Самоучитель 2.0

уроки по C++

ravesli.com

СОДЕРЖАНИЕ

Глава №2. Переменные и основные типы данных в C++

Глава №3. Операторы в C++

Глава №4. Область видимости и другие типы переменных в C++

Глава №5. Порядок выполнения кода в программе. Циклы, ветвления в C++

Глава №6. Массивы, Строки, Указатели и Ссылки в C++

Глава №7. Функции в C++

Глава №8. Основы ООП в C++

Глава №9. Перегрузка операторов в C++

Глава №10. Введение в связи между объектами в C++

Глава №11. Наследование в C++

Глава №12. Виртуальные функции в C++

Глава №13. Шаблоны в C++

Глава №14. Исключения в C++

Глава №15. Умные указатели и Семантика перемещения в C++

Глава №16. Стандартная библиотека шаблонов (STL) в C++

Глава №17. std::string в C++

Глава №18. Ввод/Вывод в C++

Дополнительные уроки

Финал

Ответы на Тесты

ПРЕДИСЛОВИЕ

Есть два основных подхода к изучению программирования. Мы рассмотрим плюсы и минусы каждого из них и постараемся выбрать "золотую середину", чтобы максимально эффективно и результативно использовать свой самый главный ресурс - время.

Метод «Снизу-Вверх»

Суть заключается в начальном изучении базиса и необходимого фундамента для дальнейшего продвижения. Подход «Снизу-Вверх» популярен не только на многих онлайн- и офлайн- курсах, но и в образовательных учреждениях (например, в университетах или колледжах). Вы начинаете с нуля и изучаете только одну концепцию или тему за раз. Идея состоит в том, чтобы получить хорошие фундаментальные основы программирования, которые в дальнейшем помогут вам развиваться в любом выбранном вами направлении программирования.

Главным преимуществом является то, что вы действительно изучаете основы. Не имеет значения, делаете ли вы 3D-игру или интерактивный веб-сайт - фундаментальные основы применимы и используются везде.

Учить каждую концепцию отдельно - легче, так как это происходит изолированно. Если выбранный вами курс с подходом «Снизу-Вверх» хорошо структурирован, то вы не будете подвергнуты бомбардировке 1000 разными терминами/концепциями за раз. Вам предоставляется каждая новая тема изолированно в «удобно-съедобном» виде. Сначала идут базовые вещи (например, что такое переменная, функция, цикл и т.д.), а затем уже происходит плавный переход к более сложным темам (например, к классам, ООП и т.д.).

Минусом (если его вообще можно так назвать) является скорость продвижения. Хотя это всё также индивидуально. Сначала вы тратите время, чтобы получить необходимые знания, и только потом применяете их на практике. Для создания чего-либо значительного вам потребуется многое узнать. Например, для реализации интерактивного веб-сайта вам могут понадобиться недели, если не месяцы обучения. А для создания более-менее хорошей 3D-игры вам понадобятся месяцы, если не годы обучения.

А когда период обучения затягивается, и счастья от мгновенного результата вы не получаете — вот именно здесь и начинается конфликт ожиданий. Большинству

новичков гораздо интереснее создать веб-сайт за день, нежели неделю разбираться со всеми нюансами циклов или массивов. Это не плохо и не хорошо, это факт.

Метод «Сверху-Вниз»

Суть заключается в создании рабочих проектов с самого старта обучения. Большинство программистов-самоучек начинали свой путь именно с этого подхода. Цель — создать готовый, рабочий проект. Например, 3D-игру или красивый интерактивный веб-сайт. Самый быстрый способ реализовать проект с нуля - следовать подробному туториалу. Повторяя в точности все шаги из выбранного вами гайда вы сможете в течении относительно небольшого срока создать готовый проект с нуля.

Главным преимуществом является то, что вы сразу что-то делаете, а не погружаетесь в изучение базиса. Вы не тратите время на то, чтобы ознакомиться и разобраться со всеми концепциями и их нюансами. Вы тупо делаете проект. И это чувство мгновенного результата от своих действий мотивирует вас на протяжении прохождения всех уроков.

Но в этом и кроется **главный недостаток - вы не учите основы**. Базис, который вам необходим для дальнейшего роста, вы просто пропускаете.

В конце туториала вы даже можете не понимать, как работает ваше творение. Если уроки недостаточно подробные, то у вас будут большие пробелы в знаниях, которые заполнить быстро уже не получится. Вам все равно придется потратить якобы сэкономленное время на изучение основ. Если же вы попытаетесь хоть малейшим образом отойти от инструкций туториала, то весь ваш проект может рухнуть в одну минуту, и вы даже не поймете, почему это произошло и как это исправить. Вам просто нужно всё копировать и повторять. Даже не пытайтесь заглянуть "под капот". Не смотрите в эту бездну :)

Какой подход выбрать?

Ни первый, ни второй. Чтобы стать опытным программистом, вам нужно совмещать оба этих подхода.

Вам нужен опыт создания программ/приложений/продуктов, даже если вы не понимаете полностью все детали. Вам нужна мотивация, чтобы продолжать обучаться, и это чувство счастья от мгновенного результата может вам помочь. Изучение только концепций и теории недостаточно, чтобы подготовить вас к работе над реальными проектами.

Но при этом вам необходимы знания основ. Вы должны уметь понимать и отлаживать как собственный код, так и код, написанный другими разработчиками. Вы должны научиться писать код сами, без помощи кого-либо и без каких-либо подсказок. Привыкать к готовым решениям в Интернете — не очень хорошая практика, так как в будущем, когда перед вами поставят конкретную задачу, решения в Интернете вы можете и не найти, а задачу решить придется.

Выход - чередовать эти 2 подхода.

Чувствуете, что перегружены и не понимаете, что делает ваш код? Переключитесь на подход "Снизу-Вверх". Вернитесь к теории и разберитесь с тем, что и как делает ваш код. Постарайтесь заполнить пробелы в своих знаниях.

Надоедает изучение абстрактных концепций программирования? Переключитесь на подход "Сверху-Вниз". Создайте что-то маленькое, например, простенький сайт или игру. Постарайтесь применить на практике полученные знания и ощутите "счастье от мгновенного результата".

Легко и просто не будет, но и не будет настолько сложно, чтобы это не осилить. Не беспокойтесь слишком много о том, что вы еще чего-то не знаете - это лишь дело времени. Но и не забывайте о том, что без действий ничего, абсолютно ничего не произойдет (по крайней мере, хорошего). А чтобы вам помочь в изучении программирования (в частности C++), и был создан данный курс по C++.

На этом курсе как раз и совмещаются 2 подхода, о которых говорилось ранее. Все темы раскрываются поочерёдно и изложены в "удобно-съедобном" виде. Вся теория сразу же иллюстрируется в примерах, а в конце уроков (не каждого!) есть тесты — это задания, которые вам нужно решить, применив на практике только что полученные знания. Кроме того, в конце каждой главы есть итоговый тест - то самое чувство мгновенного результата и самостоятельной практики. Поэтому пробуйте и находите то, что подходит именно вам, совмещайте разные подходы и не забывайте о самом главном - получать удовольствие от процесса.

Урок №1. Введение в программирование

Компьютеры понимают только очень ограниченный набор инструкций, и чтобы заставить их что-то делать, нужно четко сформулировать задание, используя эти же инструкции. **Программа** (также «*приложение*» или «*программное обеспечение*», или «*софт*») - это набор инструкций, которые указывают компьютеру, что ему нужно делать. Физическая часть компьютера, которая выполняет эти инструкции, называется ***«железом»*** или ***аппаратной частью*** (например, процессор, материнская плата и т.д.). Данный урок является началом серии уроков по программированию на языке C++ для начинающих.

Машинный язык

Процессор компьютера не способен понимать напрямую языки программирования, такие как C++, Java, Python и т.д. Очень ограниченный набор инструкций, которые изначально понимает процессор, называется **машинным кодом** (или ***«машинным языком»***). То, как эти инструкции организованы, выходит за рамки данного введения, но стоит отметить две вещи.

Во-первых, каждая команда (инструкция) состоит только из определенной последовательности (набора) цифр: `0` и `1`. Эти числа называются **битами** (сокр. от «***bi****nary digi****t***») или **двоичным кодом**.

Например, одна команда машинного кода архитектуры ×86 выглядит следующим образом:

```
10110000 01100001
```

Во-вторых, каждый набор бит переводится процессором в инструкции для выполнения определенного задания (например, *сравнить два числа* или *переместить число в определенную ячейку памяти*). Разные типы процессоров обычно имеют разные наборы инструкций, поэтому инструкции, которые будут работать на процессорах Intel (используются в персональных компьютерах), с большей долей вероятности, не будут работать на процессорах Xenon (используются в игровых приставках Xbox). Раньше, когда компьютеры только начинали массово распространяться, программисты должны были писать программы непосредственно на машинном языке, что было очень неудобно, сложно и занимало намного больше времени, чем сейчас.

Язык ассемблера

Так как программировать на машинном языке - удовольствие специфическое, то программисты изобрели язык ассемблера. В этом языке каждая команда идентифицируется коротким именем (а не набором единиц с нулями), и переменными можно управлять через их имена. Таким образом, писать/читать код стало гораздо легче. Тем не менее, процессор все равно не понимает язык ассемблера напрямую. Его также нужно переводить, с помощью ассемблера, в машинный код. **Ассемблер** — это транслятор (переводчик), который переводит код, написанный на языке ассемблера, в машинный язык. В Интернете язык ассемблера часто называют просто *"Ассемблер"*.

Преимуществом Ассемблера является его производительность (точнее скорость выполнения) и он до сих пор используется, когда это имеет решающее значение. Тем не менее, причина подобного преимущества заключается в том, что программирование на этом языке адаптируется к конкретному процессору. Программы, адаптированные под один процессор, не будут работать с другим. Кроме того, чтобы программировать на Ассемблере, по-прежнему нужно знать очень много не очень читабельных инструкций для выполнения даже простого задания.

Например, вот вышеприведенная команда, но уже на языке ассемблера:

```
mov al, 061h
```

Высокоуровневые языки программирования

Для решения проблем читабельности кода и чрезмерной сложности были разработаны высокоуровневые языки программирования. C, C++, Pascal, Java, JavaScript и Perl - это всё **языки высокого уровня**. Они позволяют писать и выполнять программы, не переживая о совместимости кода с разными архитектурами процессоров. Программы, написанные на языках высокого уровня, также должны быть переведены в машинный код перед выполнением. Есть два варианта:

- компиляция, которая выполняется компилятором;
- интерпретация, которая выполняется интерпретатором.

Компилятор — это программа, которая читает код и создает автономную (способную работать независимо от другого аппаратного или программного обеспечения) исполняемую программу, которую процессор понимает напрямую.

При запуске программы весь код компилируется целиком, а затем создается исполняемый файл и уже при повторном запуске программы компиляция не выполняется.

Проще говоря, процесс компиляции выглядит следующим образом:

Интерпретатор - это программа, которая напрямую выполняет код, без его предыдущей компиляции в исполняемый файл. Интерпретаторы более гибкие, но менее эффективные, так как процесс интерпретации выполняется повторно при каждом запуске программы.

Процесс интерпретации:

Любой язык программирования может быть компилируемым или интерпретируемым, однако, такие языки, как C, C++ и Pascal - компилируются, в то время как "скриптовые" языки, такие, как Perl и JavaScript - интерпретируются. Некоторые языки программирования (например, Java) могут как компилироваться, так и интерпретироваться.

Преимущества высокоуровневых языков программирования

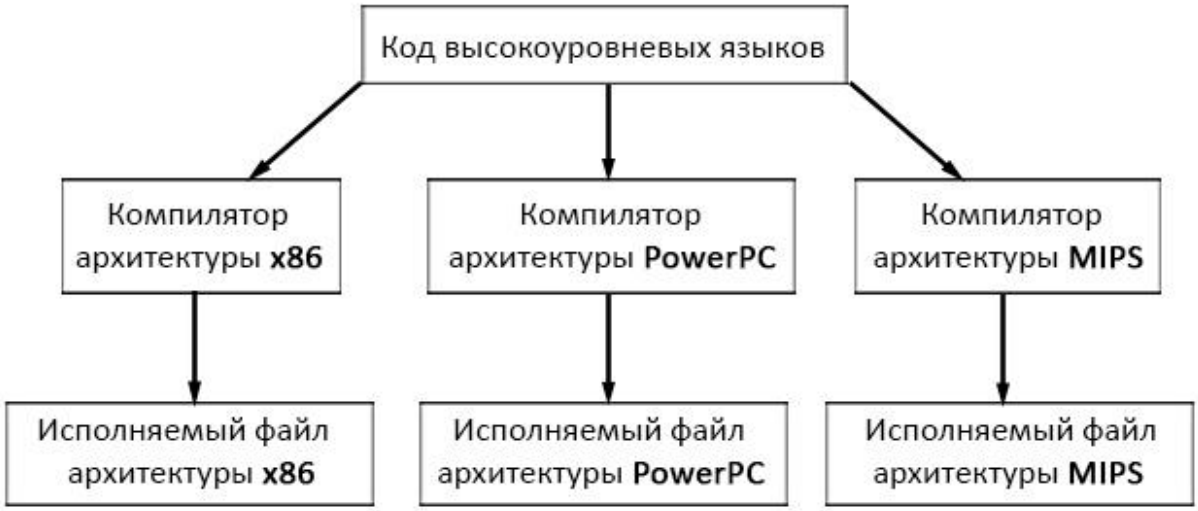
***Преимущество №1*:** Легче писать/читать код. Вот вышеприведенная команда, но уже на языке C++:

```
a = 97;
```

Преимущество №2: Требуется меньше инструкций для выполнения определенного задания. В языке C++ вы можете сделать что-то вроде `a = Ь * 2 + 5;` в одной строке. В языке ассемблера вам пришлось бы использовать 5 или 6 инструкций.

Преимущество №3: Вы не должны заботиться о таких деталях, как загрузка переменных в регистры процессора. Компилятор или интерпретатор берёт это на себя.

Преимущество №4: Высокоуровневые языки программирования более портируемые под различные архитектуры (но есть один нюанс).



Нюанс заключается в том, что многие платформы, такие как Microsoft Windows, имеют свои собственные специфические функции, с помощью которых писать код намного легче. Но в таком случае приходится жертвовать портируемостью, так как функции, специфические для одной платформы, с большей долей вероятности, не будут работать на другой платформе. Обо всем этом мы детально поговорим на следующих уроках.

Урок №2. Введение в языки программирования C и C++

Перед C++ был C. C (произносится как *"Cu"*) был разработан в 1972 году Деннисом Ритчи в *Bell Telephone Laboratories* как системный язык программирования, т.е. язык для написания операционных систем. Основной задачей Ритчи было создание легко компилируемого минималистического языка, который предоставлял бы эффективный доступ к памяти, относительно быстро выполнялся, и на котором можно было бы писать эффективный код. Таким образом, при разработке высокоуровневого языка, был создан язык Си, который во многом относился к языкам низкого уровня, оставаясь при этом независимым от платформ, для которых мог быть написан код.

Си в конечном итоге стал настолько эффективным и гибким, что в 1973 году Ритчи и Кен Томпсон переписали больше половины операционной системы UNIX, используя этот язык. Многие предыдущие операционные системы были написаны на **языке ассемблера**. В отличие от Ассемблера, на котором пишутся программы под конкретные процессоры, высокая портативность языка Си позволила перекомпилировать UNIX и на другие типы компьютеров, ускоряя его популяризацию. Язык Си и операционная система UNIX тесно связаны между собой, и популярность первого отчасти связана с успехом второго.

В 1978 году Брайан Керниган и Деннис Ритчи опубликовали книгу под названием ***"Язык программирования Си"***. Эта книга, более известна как ***"K&R"*** (первые буквы фамилий авторов), стала стандартом и своеобразной инструкцией к Си. Когда требовалась максимальная портативность, то программисты придерживались рекомендаций в *"K&R"*, поскольку большинство компиляторов в то время были реализованы в соответствии со стандартами, присутствующими в этой книге.

В 1983 году Американский национальный институт стандартов (сокр. ***"ANSI"*** от англ. *"**A**merican **N**ational **S**tandards **I**nstitute"*) сформировал комитет для утверждения официального стандарта языка Си. В 1989 году они закончили и выпустили стандарт C89, более широко известный, как ANSI C. В 1990 году Международная организация по стандартизации (сокр. ***"ISO"*** от англ. *"**I**nternational **O**rganization for **S**tandardization"*) приняла ANSI C (с небольшими изменениями). Эта версия языка Си стала известна как C90. В конечном счете, компиляторы адаптировались под требования ANSI C/C90 и программы, в которых требовалась максимальная портативность, писались в соответствие с этими стандартами.

В 1999 году комитет ANSI выпустил новую версию языка Си, которая получила название C99. Она приняла много особенностей, которые были реализованы в компиляторах (в виде различных расширений) или уже в языке C++.

Язык C++

C++ (произносится как *«Си плюс плюс»*) был разработан Бьёрном Страуструпом в *Bell Labs* в качестве дополнения к Си в 1979 г. Он добавил множество новых возможностей в язык Си. Его популярность была вызвана объектно-ориентированностью языка. Об объектно-ориентированном программировании (ООП) и его отличиях от традиционных методов программирования мы поговорим несколько позже.

Язык C++ был ратифицирован (одобрен) комитетом ISO в 1998 году и потом снова в 2003 году (под названием C++03). Потом были еще три обновления (C++11, C++14 и C++17, ратифицированные в 2011, 2014 и 2017 годах, соответственно), которые добавили больше функциональных возможностей.

Философия C и C++

Смысл философии языков C и C++ можно определить выражением "доверять программисту". Например, компилятор не будет вам мешать сделать что-то новое, что имеет смысл, но также не будет мешать вам сделать что-то такое, что может привести к сбою. Это одна из главных причин, почему так важно знать то, что вы не должны делать, как и то, что вы должны делать, создавая программы на языках C/C++.

> ***Примечание:*** Вам не нужны знания языка Си, чтобы проходить данные уроки. В процессе изучения этих уроков вы получите необходимую базу/фундамент знаний.

Урок №3. Введение в разработку программного обеспечения

Перед написанием и выполнением нашей первой программы, мы должны понять, как вообще выполняется разработка программного обеспечения на языке C++.

Схема разработки ПО (сокр. от *"**П**рограммное **О**беспечение"*):

Шаг №1: Определите проблему, которую хотели бы решить

Это шаг «**Что?**». Здесь вы должны понять, что же вы хотите, чтобы ваша программа делала. Этот шаг может быть, как самым простым, так и самым сложным. Всё, что вам нужно — это четко сформулировать идею. Только после этого вы сможете приступать к следующему шагу.

Вот несколько примеров выполнения шага №1:

- "Я хочу написать программу, которая вычисляла бы среднее арифметическое чисел, которые я введу".
- "Я хочу написать программу, в которой будет 2D-лабиринт, по которому сможет передвигаться пользователь".
- "Я хочу написать программу, которая будет анализировать цены акций на бирже и давать предсказания по поводу скачков этих цен вверх или вниз".

Шаг №2: Определитесь, как вы собираетесь решить эту проблему

Здесь мы уже отвечаем на вопрос «**Как?**». Каким образом можно решить проблему, обозначенную на шаге №1? Этим шагом довольно часто пренебрегают при разработке программного обеспечения. Суть в том, что способов решения задачи может быть много, только часть из них - хорошие решения, а часть - плохие. Вы должны научиться отделять первые от вторых. Очень часто можно наблюдать ситуацию, когда у программиста возникает идея и он сразу же садится программировать. Как вы уже могли догадаться, такой сценарий далеко не всегда приводит к эффективным результатам.

Как правило, **хорошие решения имеют следующие характеристики**:

- простота;
- хорошая документация (с инструкциями и комментариями);
- модульный принцип: любая часть программы может быть повторно использована или изменена позже, не затрагивая другие части кода;
- надежность: соответствующая обработка ошибок и экстренных ситуаций.

Когда вы садитесь и начинаете сразу программировать, вы думаете: "Я хочу сделать *это*, *вот это и еще вот это*!". Таким образом вы принимаете решения, которые позволят вам поскорее выполнить задание. Однако это может привести к тому, что вы получите программу, которую позже будет трудно изменить/модифицировать, добавить что-то новое или вам попросту придется разбираться с большим количеством ошибок.

Согласно закону Парето, программист тратит примерно 20% времени на написание программы и 80% на отладку (исправление ошибок) или поддержку (добавление новых функциональных возможностей) кода. Следовательно, лучше потратить дополнительное время на обдумывание лучшего способа решения проблемы перед процессом написания кода, нежели потом тратить оставшиеся 80% времени на поиск и исправление ошибок.

Шаг №3: Напишите программу

Для того, чтобы написать программу, необходимы две вещи:

- знание определенного языка программирования (этому мы вас научим);
- редактор кода.

Программу можно написать, используя любой редактор, даже тот же *Блокнот* в Windows или текстовый редактор *Vi* в Unix. Тем не менее, я настоятельно рекомендую использовать редактор, предназначенный для программирования. Не беспокойтесь, если у вас его еще нет. На следующем уроке мы рассмотрим процесс установки такого приложения.

Редактор типичного программиста, как правило, имеет следующие особенности, которые облегчают программирование:

- **Нумерация строк**. Это функция чрезвычайно полезна при отладке программ, когда компилятор выдаёт нам сообщения об ошибках. Типичная ошибка компиляции состоит из наименования ошибки и номера строки, где эта ошибка произошла (например, *"ошибка переопределения переменной x, строка 90"*). Без нумерации строк искать ту самую 90-ю строку кода было бы несколько затруднительно, не так ли?
- **Подсветка синтаксиса**. Подсветка синтаксиса изменяет цвет разных частей программы и кода, что улучшает восприятие как целой программы, так и её отдельных частей.
- **Специальный шрифт**. В обычных шрифтах очень часто возникает путаница между определенными символами, когда непонятно, какой символ перед вами. Например: цифра `0` или буква `O`, цифра `1` или буква `l` (нижний регистр `L`), или может буква `I` (верхний регистр `i`). Вот для этого и нужен специальный шрифт, в котором будет легко различить эти символы, предотвращая случайное использование одного символа вместо другого.

Программы на языке C++ следует называть **name.cpp**, где *name* заменяется именем вашей программы, а расширение *.cpp* сообщает компилятору (и вам тоже), что это исходный файл кода, который содержит инструкции на языке программирования C++. Следует обратить внимание, что некоторые программисты используют расширение *.cc* вместо *.cpp*, но я рекомендую использовать именно *.cpp*.

Также стоит отметить, что много программ, написанных на языке C++, могут состоять из нескольких файлов *.cpp*. Хотя большинство программ, которые вы будете создавать на протяжении этих уроков, не будут слишком большими, в

дальнейшем вы научитесь писать программы, которые будут включать десятки, если не сотни отдельных файлов *.cpp*.

Шаг №4: Компиляция

Для того, чтобы скомпилировать программу нам нужен компилятор. **Работа компилятора состоит из двух частей**:

- Проверка программы на соответствие правилам языка C++ (проверка синтаксиса). Если она будет неудачной, то компилятор выдаст сообщения об ошибках, которые нужно будет исправить.
- Конвертация каждого исходного файла с кодом в **объектный файл** (или ***"объектный модуль"***) на машинном языке. Объектные файлы, как правило, имеют названия *name.o* или *name.obj*, где *name* должно быть такое же как и имя вашего исходного файла .cpp. Если ваша программа состоит из 3-х файлов .cpp, то компилятор сгенерирует 3 объектных файла.

Стоит отметить, что такие операционные системы как Linux и macOS имеют уже встроенный компилятор C++, который называется **g++**. Для компиляции файлов из командной строки с помощью g++ вам нужно будет прописать следующее:

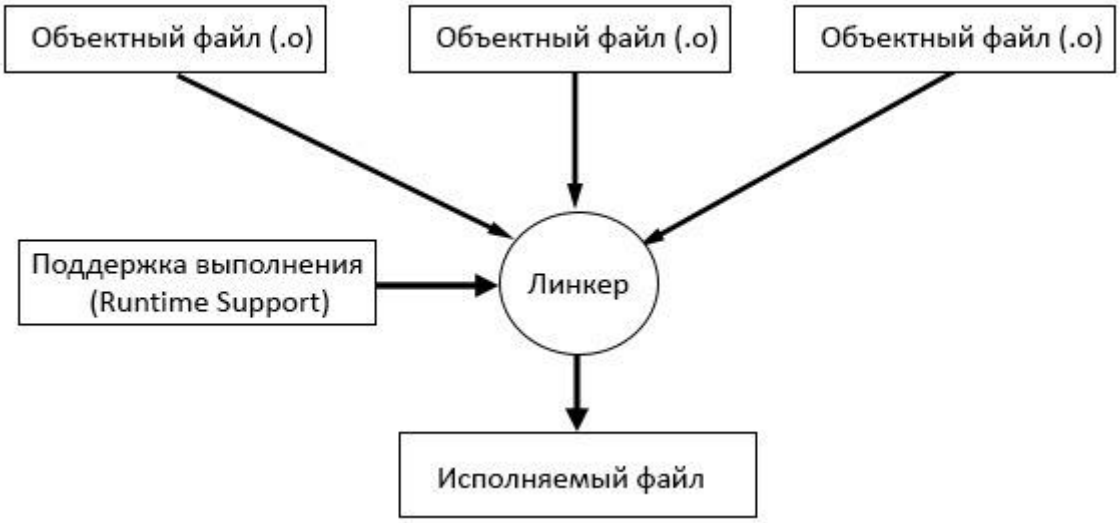
```
g++ -c file1.cpp file2.cpp file3.cpp
```

Таким образом мы создадим file1.o, file2.o и file3.o. `-c` означает "только скомпилировать", т.е. просто создать *.o* (объектные) файлы. Кроме g++, существует множество компиляторов для различных операционных систем: Linux, Windows, macOS и других.

Шаг №5: Линкинг (связывание объектных файлов)

Линкинг — это процесс связывания всех объектных файлов, генерируемых компилятором, в единую исполняемую программу, которую вы затем сможете

запустить/выполнить. Это делается с помощью программы, которая называется **линкер** (или ***"компоновщик"***).



Кроме объектных файлов, линкер также подключает файлы из Стандартной библиотеки C++ (или любой другой библиотеки, которую вы используете, например, библиотеки графики или звука). Сам по себе язык C++ довольно маленький и простой. Тем не менее, к нему подключается большая библиотека дополнительных функций, которые могут использовать ваши программы, и эти функции находятся в Стандартной библиотеке C++. Например, если вы хотите вывести что-либо на экран, то у вас в коде должна быть специальная команда, которая сообщит компилятору, что вы хотите использовать функцию вывода информации на экран из Стандартной библиотеки C++.

После того, как компоновщик закончит линкинг всех объектных файлов (при условии, что не будет ошибок), вы получите исполняемый файл. Опять же, в целях наглядности, чтобы связать *.o* файлы, которые мы создали выше в Linux или macOS, мы можем снова использовать g++:

```
g++ -o prog file1.o file2.o file3.o
```

Команда `-o` сообщает g++, что мы хотим получить исполняемый файл с именем `prog` из следующих файлов: file1.o, file2.o и file3.o. При желании, компиляцию и линкинг можно объединить в один шаг:

```
g++ -o prog file1.cpp file2.cpp file3.cpp
```

Результатом будет исполняемый файл с именем `prog`.

Шаг №6: Тестирование и отладка

Здесь начинается самое веселое! Вы уже можете запустить исполняемый файл и посмотреть, работает ли всё так, как надо. Если нет, то пришло время отладки. Более подробно об отладке мы поговорим чуть позже.

Обратите внимание, для выполнения шагов №3-№6 вам потребуется специальное программное обеспечение. Хотя вы можете использовать отдельные программы на каждом из этих шагов, один пакет программного обеспечения (**"IDE"** от англ. *"**I**ntegrated **D**evelopment **E**nvironment"*) объединяет в себе все эти программы. Обычно с IDE вы получаете редактор кода с нумерацией строк и подсветкой синтаксиса, а также компилятор и линкер. А когда вам нужно будет провести отладку программы, вы сможете использовать встроенный отладчик. Кроме того, IDE объединяет и ряд других полезных возможностей: комплексная помощь, дополнение кода, в некоторых случаях еще и система контроля версий.

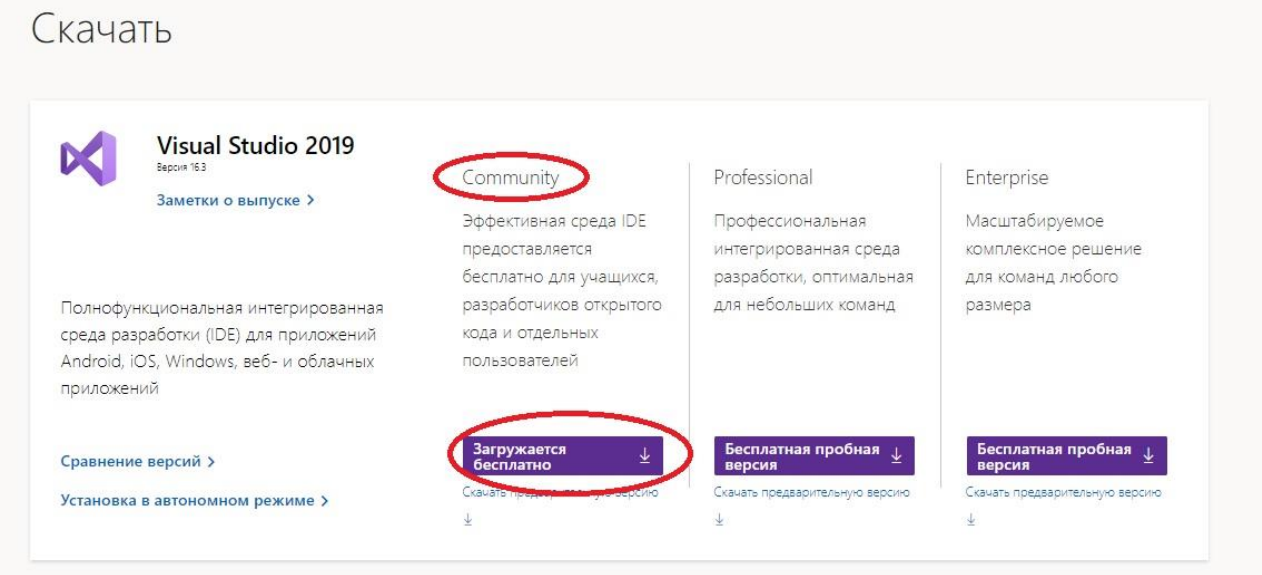
Урок №4. Установка IDE (Интегрированной Среды Разработки)

Интегрированная Среда Разработки (сокр. ***"IDE"*** от *"**I**ntegrated **D**evelopment **E**nvironment"*) — это программное обеспечение, которое содержит всё необходимое для разработки, компиляции, линкинга и отладки кода. Нам нужно установить одну такую IDE для написания программ на языке C++.

Но какую именно? Я рекомендую Visual Studio от Microsoft (для пользователей Windows) или Code::Blocks (для пользователей Linux/Windows). Также вы можете установить и любую другую IDE. Основные концепции, рассматриваемые в данных уроках, должны работать во всех средах разработки. Впрочем, иногда код может частично отличаться в разных IDE, поэтому вам придется самостоятельно искать более подробную информацию о работе в выбранной вами IDE.

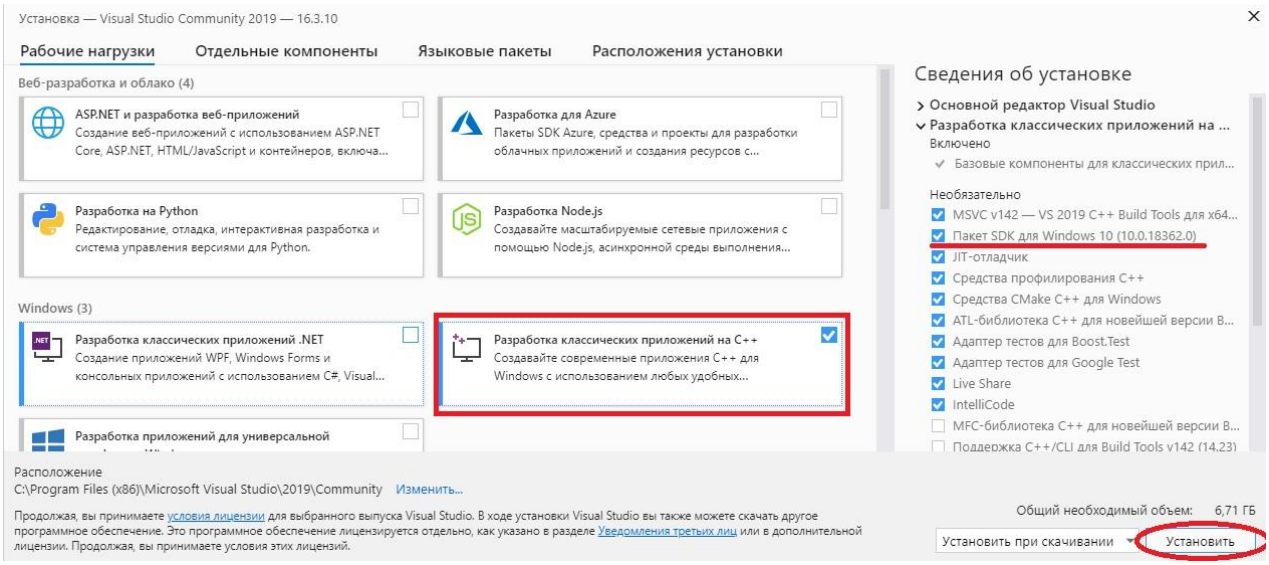
IDE для пользователей Windows

Если вы пользователь Windows (как и большинство из нас), то установите **Visual Studio 2019** версию "Community", которая является бесплатной (все остальные версии — платные):



После того, как вы скачаете и запустите установщик, вам нужно будет выбрать `"Разработка классических приложений на C++"`. Пункты, выбранные по умолчанию в правой части экрана, трогать не нужно — там всё хорошо, только убедитесь, что поставлена галочка возле пункта `"Пакет SDK для Windows 10"`. Этот пакет может использоваться и в ранних версиях Windows, поэтому не

переживайте, если у вас Windows 7 или Windows 8 — всё будет работать. Затем нажимаем `"Установить"`:

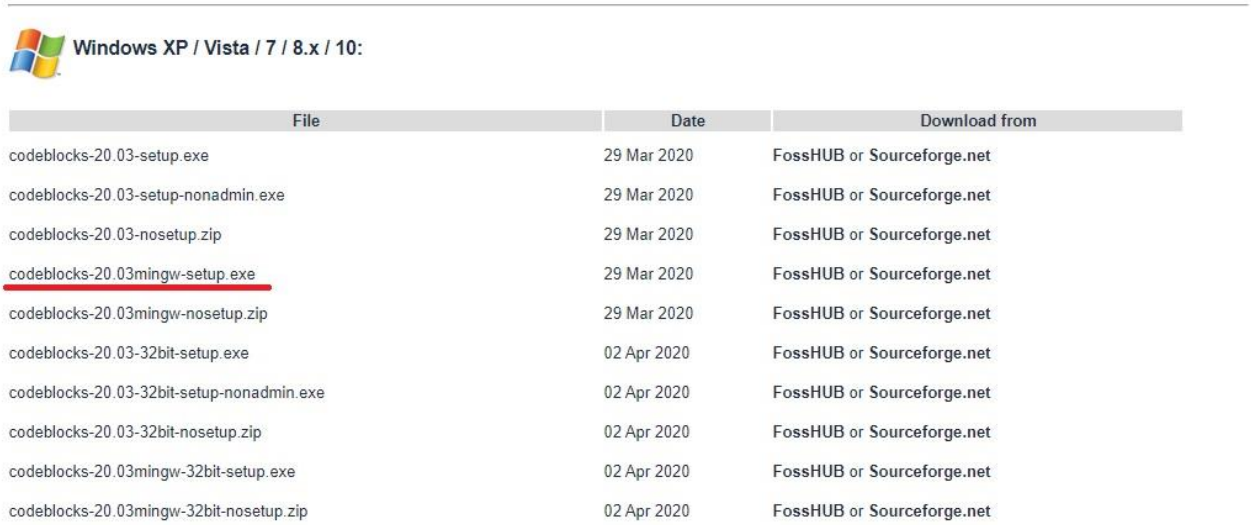


При желании вы можете указать галочки и возле других пунктов для скачивания, но учтите, что тогда размер вашей IDE будет увеличен.

IDE для пользователей Linux/Windows

Если вы пользователь Linux (или Windows, но хотите писать программы, которые затем можно будет легко портировать в Linux), то установите **Code::Blocks**. Это бесплатная, кроссплатформенная IDE, которая работает как в Linux, так и в Windows.

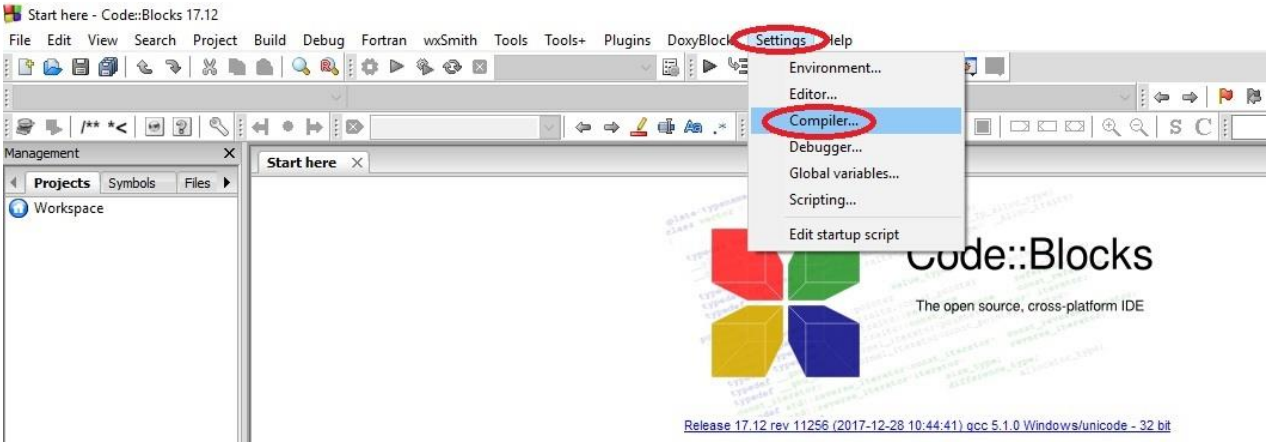
Пользователям Windows нужно загружать версию с MinGW в комплекте:



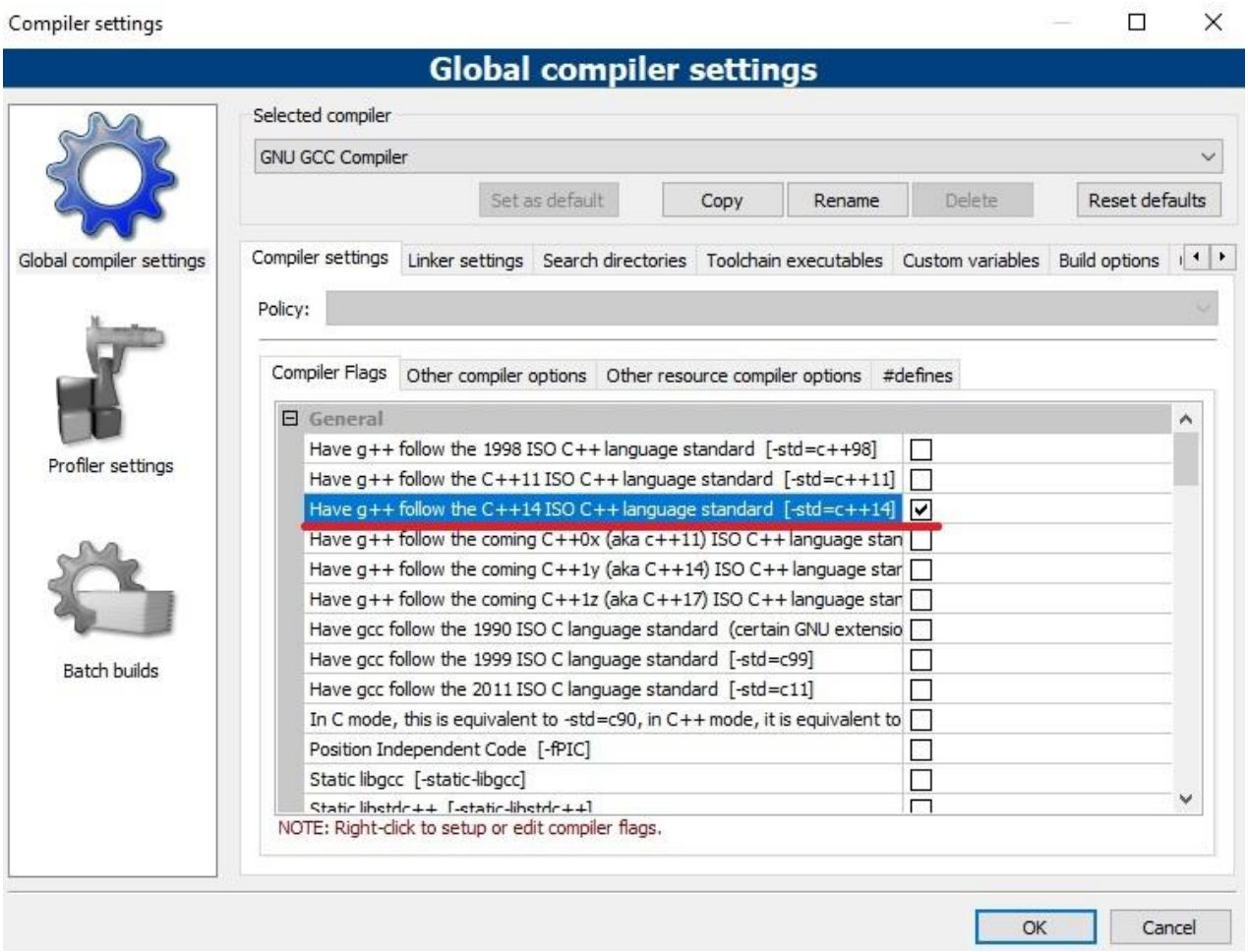
Windows XP / Vista / 7 / 8.x / 10:

File	Date	Download from
codeblocks-20.03-setup.exe	29 Mar 2020	FossHUB or Sourceforge.net
codeblocks-20.03-setup-nonadmin.exe	29 Mar 2020	FossHUB or Sourceforge.net
codeblocks-20.03-nosetup.zip	29 Mar 2020	FossHUB or Sourceforge.net
codeblocks-20.03mingw-setup.exe	29 Mar 2020	FossHUB or Sourceforge.net
codeblocks-20.03mingw-nosetup.zip	29 Mar 2020	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup.exe	02 Apr 2020	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup-nonadmin.exe	02 Apr 2020	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-nosetup.zip	02 Apr 2020	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-setup.exe	02 Apr 2020	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-nosetup.zip	02 Apr 2020	FossHUB or Sourceforge.net

Установка простая: просто соглашаетесь со всем, о чём вас спрашивают. Функционал C++11/C++14 в Code::Blocks по умолчанию может быть отключен. Чтобы его использовать, вам нужно перейти в `"Settings" > "Compiler"`:



И во вкладке `"Compiler Flags"` поставить галочку возле пункта `"Have g++ follow the C++14 ISO C++ language standard [-std=c++14]"`, затем нажать `"OK"`:



> ***Примечание:*** После установки Code::Blocks у некоторых пользователей может появиться следующее сообщение об ошибке: `Can't find compiler executable in your configured search paths for GNU GCC Compiler`. Если вы столкнулись с этим, то попробуйте выполнить следующее:
>
> - Если вы пользователь Windows, убедитесь, что вы загрузили версию Code::Blocks с MinGW (в названии скачиваемого установщика должно быть слово mingw).
> - Попробуйте полностью удалить Code::Blocks, а затем установите его заново.
> - Перейдите в `"Settings" > "Compiler"` и выберите `"Reset to defaults"`.
> - Если ничего из вышеуказанного не помогло, попробуйте установить другую IDE.
>
> В качестве альтернативы подойдет **Bloodshed's Dev-C++**, который также работает как в Windows, так и в Linux.

IDE для пользователей macOS

Пользователи техники Apple могут использовать **Xcode** или **Eclipse**. Eclipse по умолчанию не настроен на использование языка C++, поэтому вам нужно будет дополнительно установить компоненты для C++.

Веб-компиляторы

Веб-компиляторы подходят для написания простых, небольших программ. Их функционал ограничен: вы не сможете сохранять проекты, создавать исполняемые файлы или эффективно проводить отладку программ, поэтому лучше скачать полноценную IDE, если у вас действительно серьезные намерения по поводу программирования. А веб-компиляторы используйте разве что для быстрого запуска небольших программ.

Популярные веб-компиляторы:

- **OnlineGDB**
- **TutorialsPoint**
- **C++ Shell**
- **Repl.it**

Теперь, когда вы установили IDE, пора написать нашу первую программу!

Урок №5. Компиляция вашей первой программы

Перед написанием нашей первой программы мы еще должны кое-что узнать.

Теория

Во-первых, несмотря на то, что код ваших программ находится в файлах .cpp, эти файлы добавляются в **проект**. Проект содержит все необходимые файлы вашей программы, а также сохраняет указанные вами настройки вашей IDE. Каждый раз, при открытии проекта, он запускается с того момента, на котором вы остановились в прошлый раз. При компиляции программы, проект говорит компилятору и линкеру, какие файлы нужно скомпилировать, а какие связать. Стоит отметить, что файлы проекта одной IDE не будут работать в другой IDE. Вам придется создать новый проект (в другой IDE).

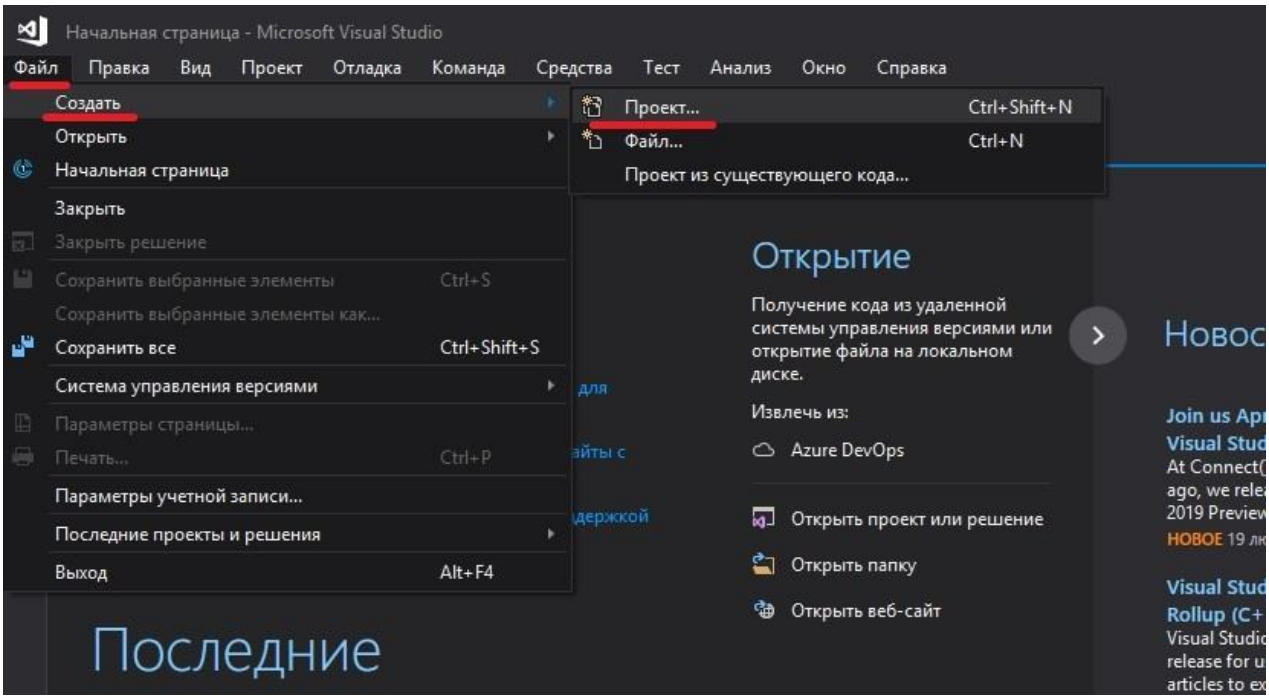
Во-вторых, есть разные типы проектов. При создании нового проекта, вам нужно будет выбрать его тип. Все проекты, которые мы будем создавать на данных уроках, будут **консольного типа**. Это означает, что они запускаются в консоли (аналог командной строки). По умолчанию, консольные приложения не имеют графического интерфейса пользователя — **GUI** (сокр. от *"**G**raphical **U**ser **I**nterface"*) и компилируются в автономные исполняемые файлы. Это идеальный вариант для изучения языка C++, так как он сводит всю сложность к минимуму.

В-третьих, при создании нового проекта большинство IDE автоматически добавят ваш проект в рабочее пространство. **Рабочее пространство** — это своеобразный контейнер, который может содержать один или несколько связанных проектов. Несмотря на то, что вы можете добавить несколько проектов в одно рабочее пространство, все же рекомендуется создавать отдельное рабочее пространство для каждой программы. Это намного упрощает работу для новичков.

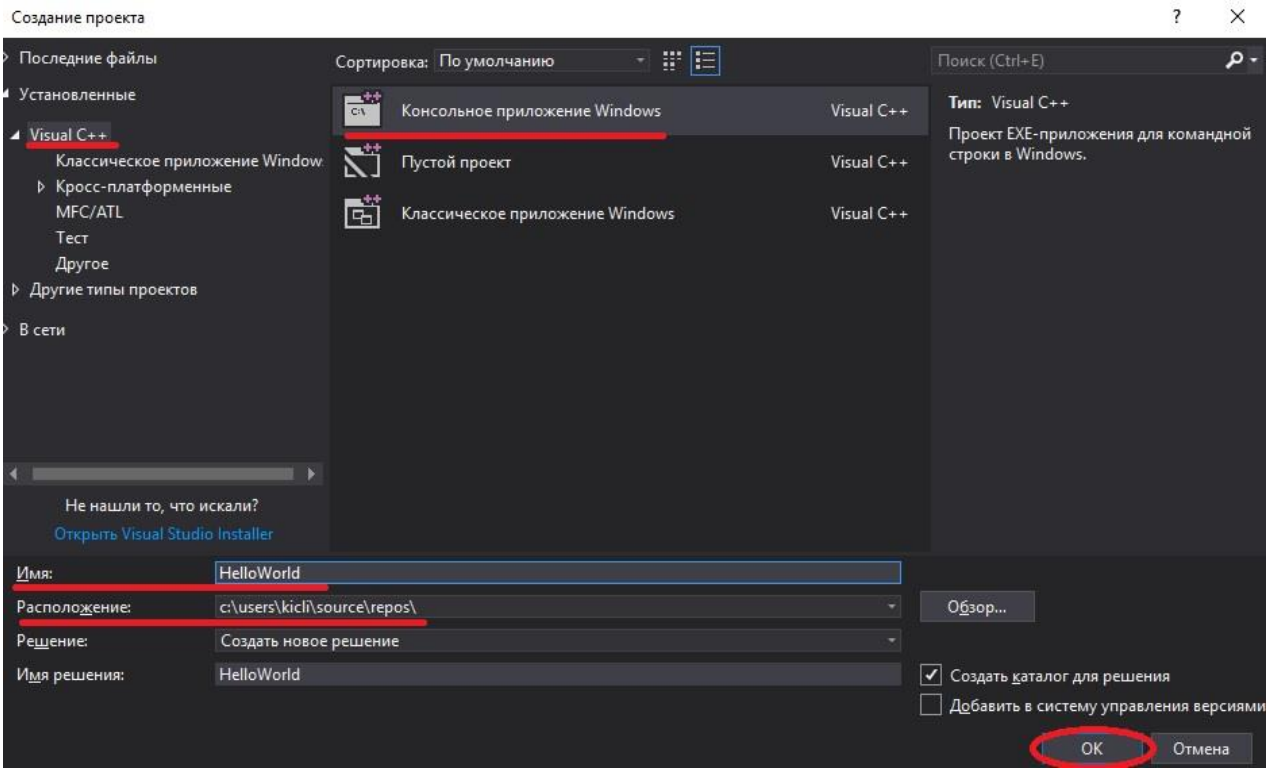
Традиционно, первой программой на новом языке программирования является всеми известная программа «Hello, world!». Мы не будем нарушать традиции :)

Пользователям Visual Studio

Для создания нового проекта в Visual Studio 2019, вам нужно сначала запустить эту IDE, затем выбрать `"Файл" > "Создать" > "Проект"`:



Дальше появится диалоговое окно, где вам нужно будет выбрать `"Консольное приложение Windows"` из вкладки `"Visual C++"` и нажать `"OK"`:

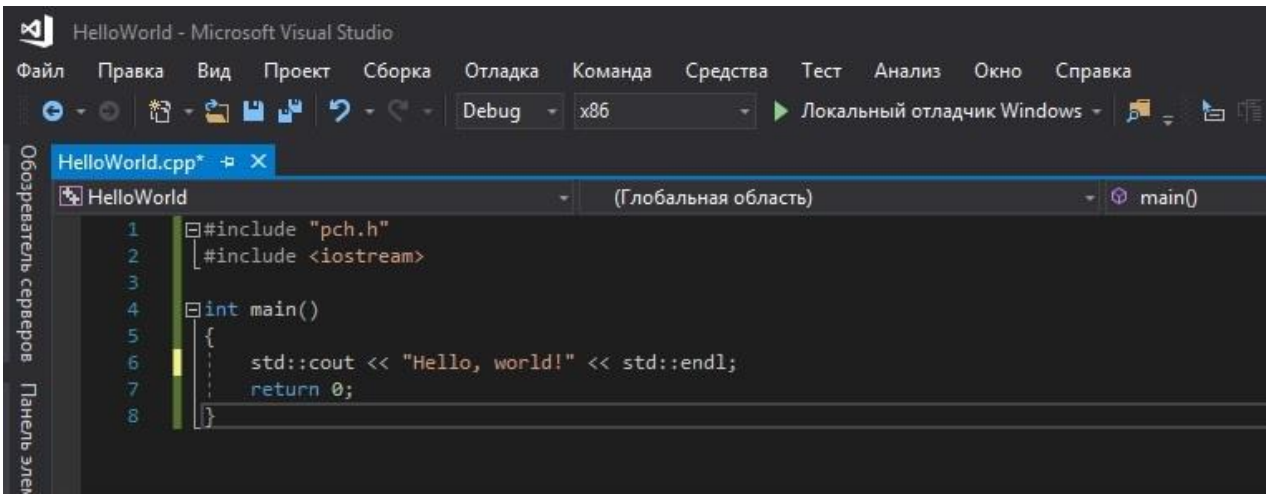


Также вы можете указать имя проекта (любое) и его расположение (рекомендую ничего не менять) в соответствующих полях.

В текстовом редакторе вы увидите, что уже есть некоторый текст и код — удалите его, а затем напечатайте или скопируйте следующий код:

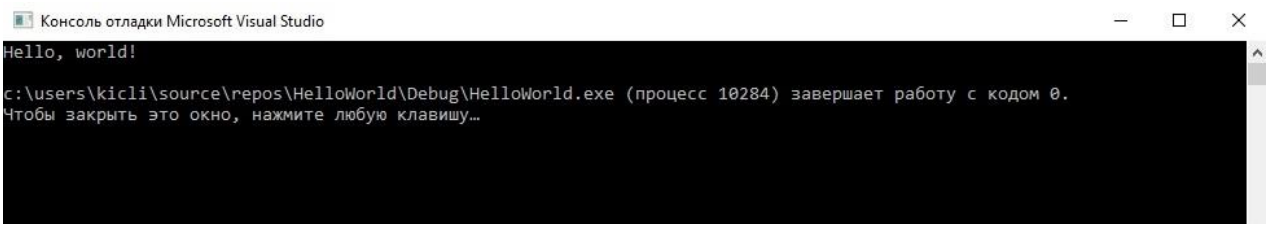
```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Hello, world!" << std::endl;
6.     return 0;
7. }
```

Вот, что у вас должно получиться:



> ***ВАЖНОЕ ПРИМЕЧАНИЕ:*** Строка `#include "pch.h"` требуется только для пользователей Visual Studio 2017. Если вы используете Visual Studio 2019 (или более новую версию), то не нужно писать эту строку вообще.

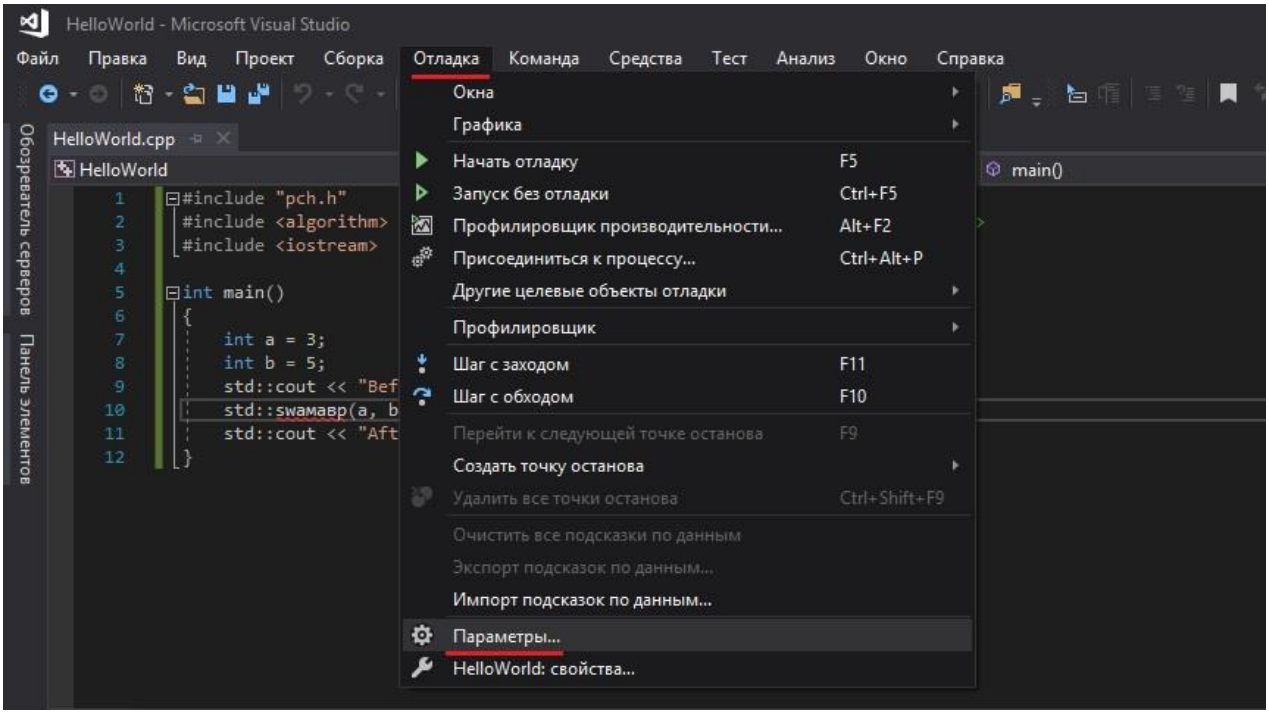
Чтобы запустить программу в Visual Studio, нажмите комбинацию Ctrl+F5. Если всё хорошо, то вы увидите следующее:



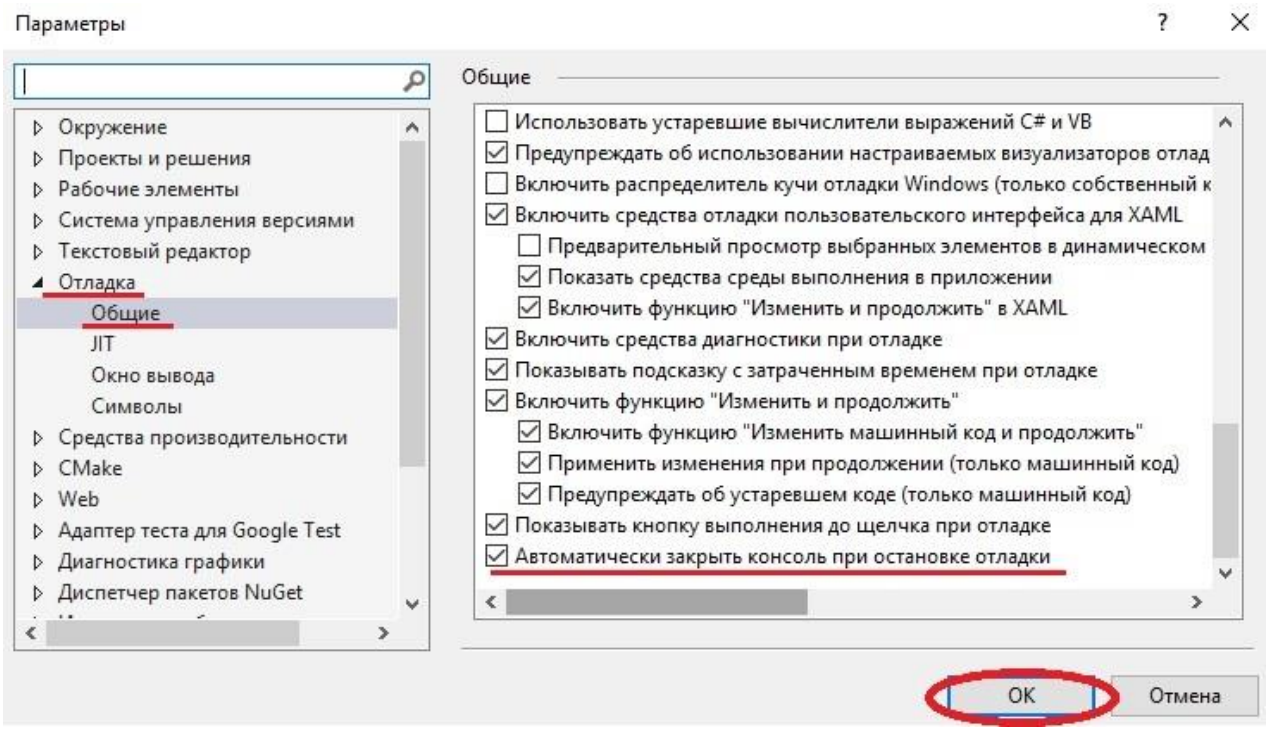
Это означает, что компиляция прошла успешно и результат выполнения вашей программы следующий:

```
Hello, world!
```

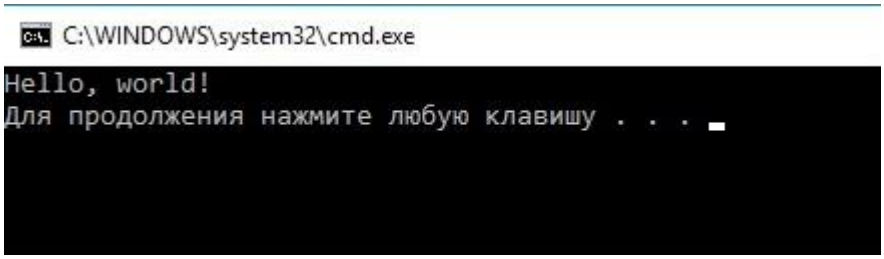
Чтобы убрать строку *"...завершает работу с кодом 0..."*, вам нужно перейти в `"Отладка" > "Параметры"`:



Затем `"Отладка" > "Общие"` и поставить галочку возле `"Автоматически закрыть консоль при остановке отладки"` и нажать `"OK"`:



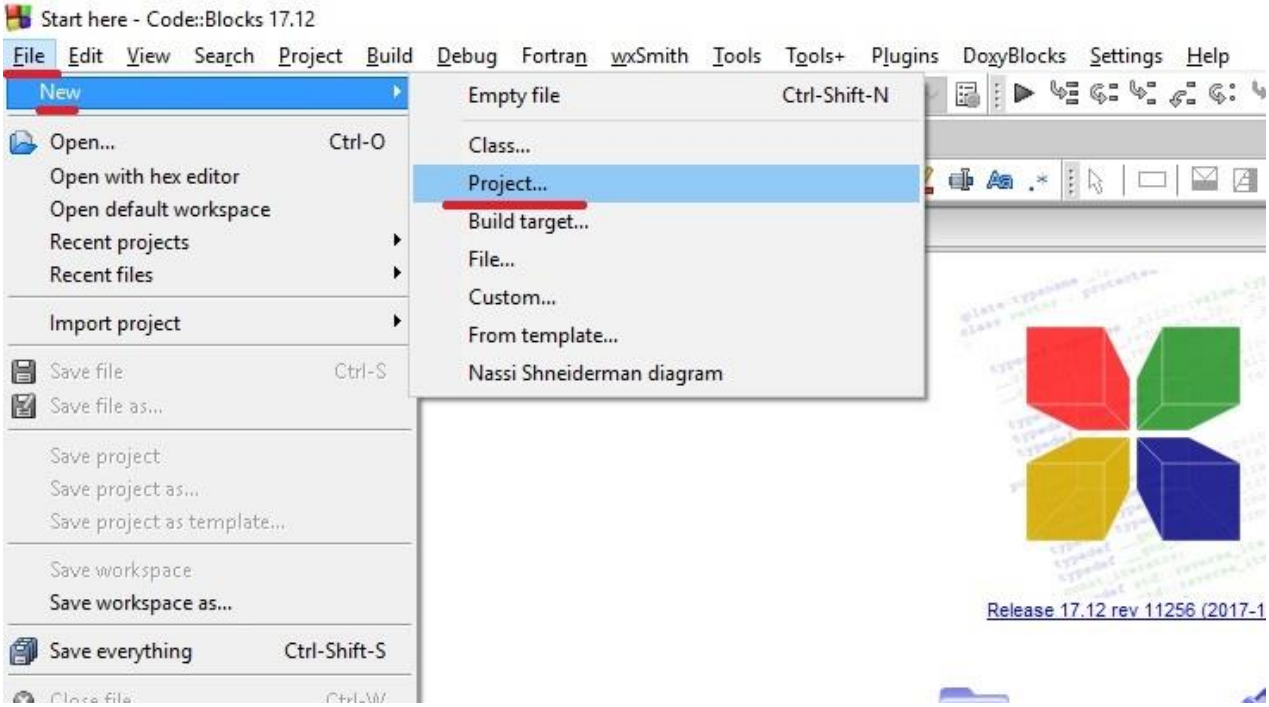
Тогда ваше консольное окно будет выглядеть следующим образом:



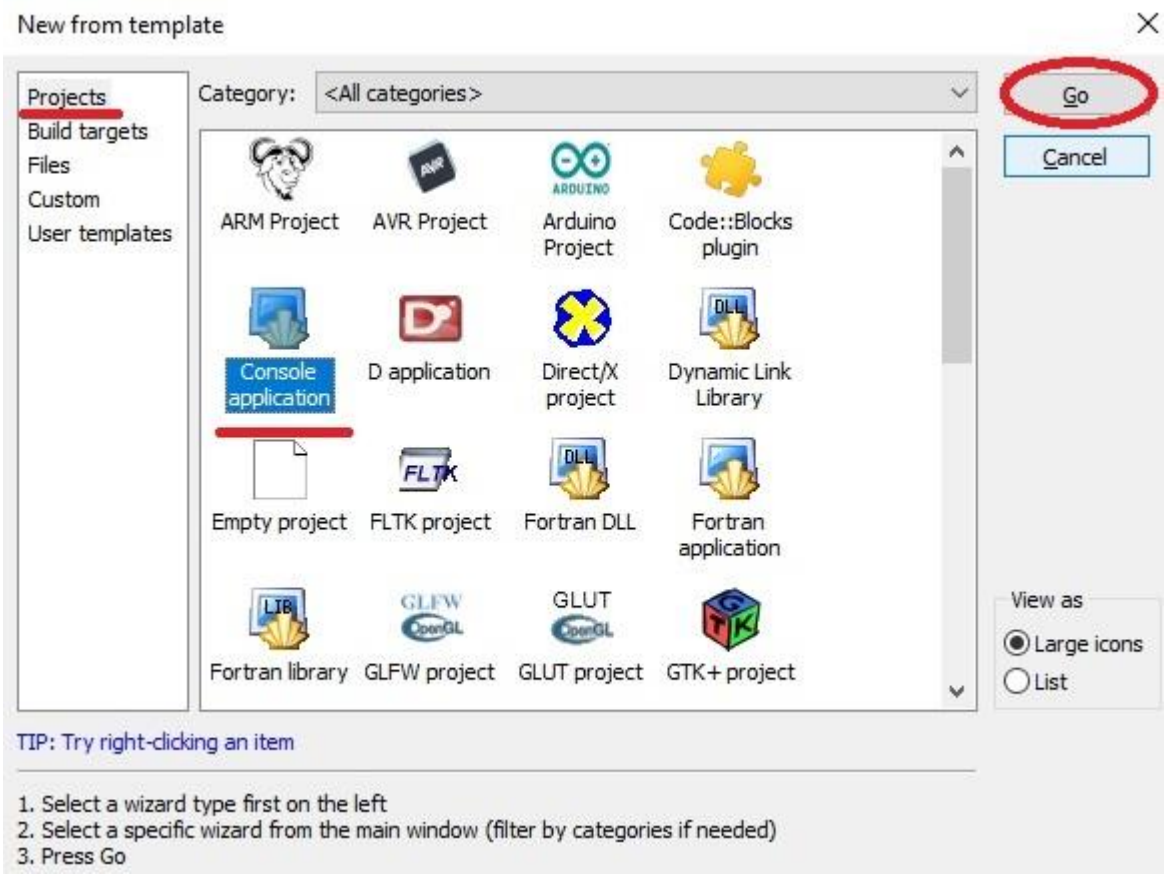
Готово! Мы научились компилировать программу в Visual Studio.

Пользователям Code::Blocks

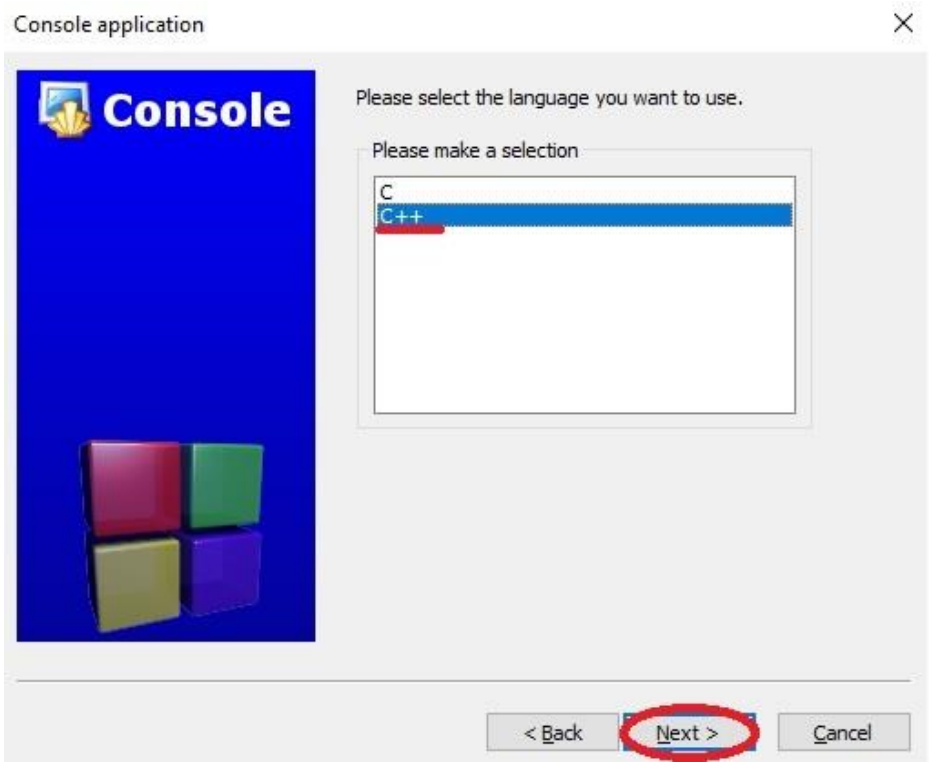
Чтобы создать новый проект, запустите Code::Blocks, выберите `"File" > "New" > "Project"`:



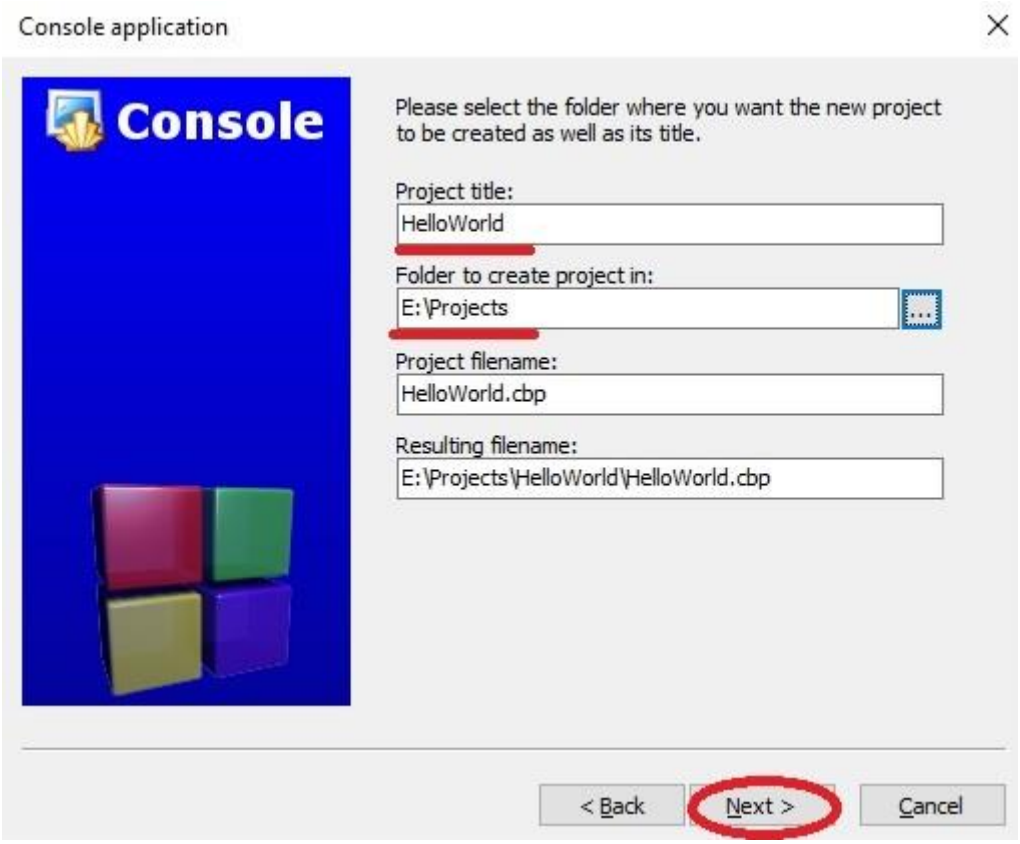
Затем появится диалоговое окно, где вам нужно будет выбрать `"Console application"` и нажать `"Go"`:



Затем выберите язык `"C++"` и нажмите `"Next"`:

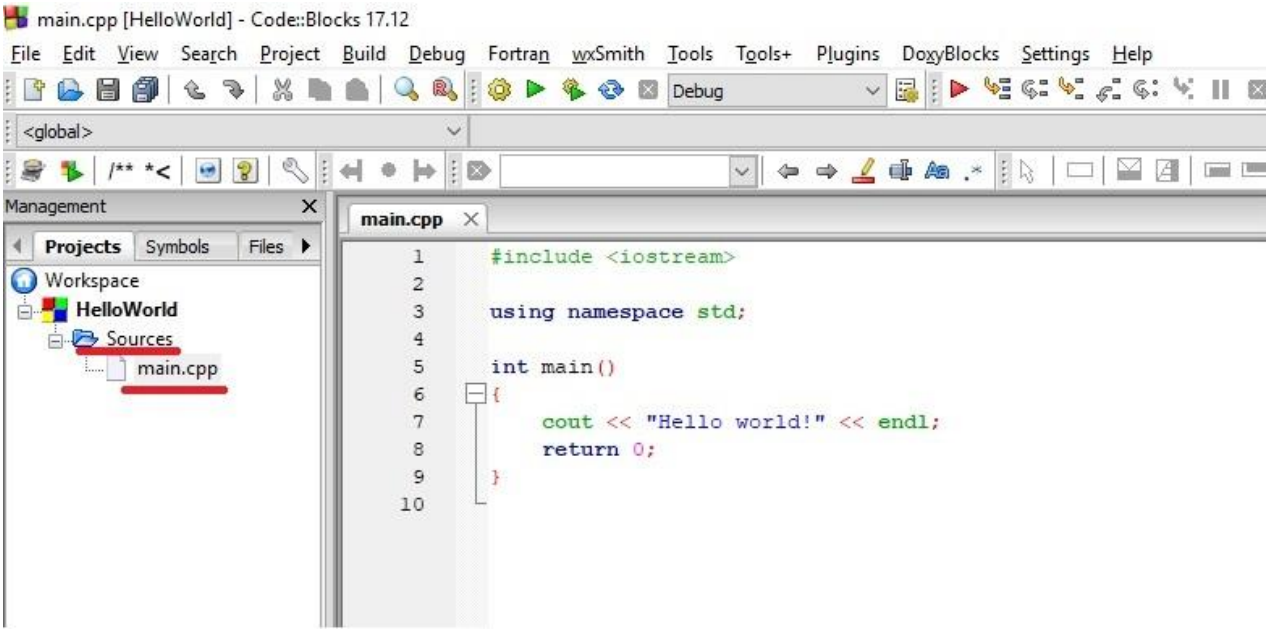


Затем нужно указать имя проекта и его расположение (можете создать отдельную папку *Projects*) и нажать `"Next"`:



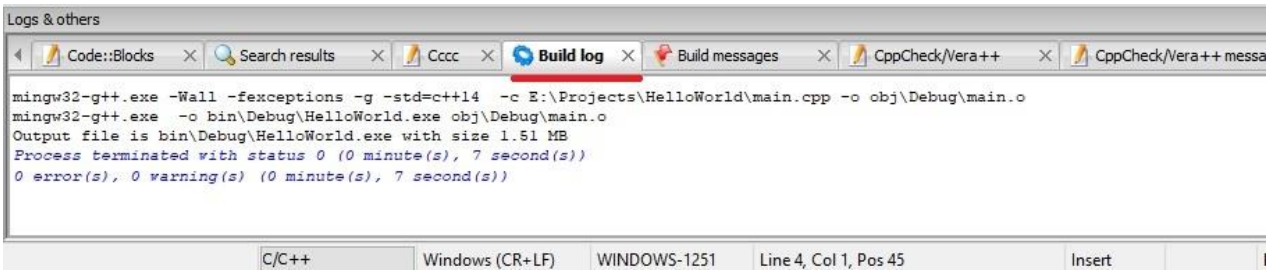
В следующем диалоговом окне нужно будет нажать `"Finish"`.

После всех этих манипуляций, вы увидите пустое рабочее пространство. Вам нужно будет открыть папку *Sources* в левой части экрана и дважды кликнуть по main.cpp:



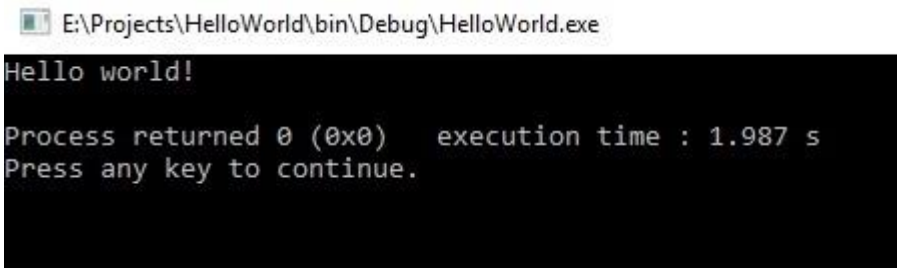
Вы увидите, что программа «Hello, world!» уже написана!

Для того, чтобы скомпилировать ваш проект в Code::Blocks, нажмите Ctrl+F9, либо перейдите в меню `"Build"` и выберите `"Build"`. Если всё пройдет хорошо, то вы увидете следующее во вкладке `"Build log"`:



Это означает, что компиляция прошла успешно!

Чтобы запустить скомпилированную программу, нажмите Ctrl+F10, либо перейдите в меню `"Build"` и выберите `"Run"`. Вы увидите следующее окно:



Это результат выполнения вашей программы.

Пользователям командной строки

Вставьте следующий код в текстовый файл с именем *HelloWorld.cpp*:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Hello, world!" << std::endl;
6.     return 0;
7. }
```

В командной строке напишите:

```
g++ -o HelloWorld HelloWorld.cpp
```

Эта команда выполнит компиляцию и линкинг файла *HelloWorld.cpp*.

Для запуска программы напишите:

```
HelloWorld
```

или

```
./HelloWorld
```

И вы увидите результат выполнения вашей программы.

Пользователям веб-компиляторов

Вставьте следующий код в рабочее пространство:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Hello, world!" << std::endl;
6.     return 0;
7. }
```

Затем нажмите `"Run"`. Вы должны увидеть результат в окне выполнения.

Пользователям других IDE

Вам нужно:

Шаг №1: Создать консольный проект.

Шаг №2: Добавить файл .cpp в проект (если нужно).

Шаг №3: Вставить следующий код в файл .cpp:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Hello, world!" << std::endl;
6.     return 0;
7. }
```

Шаг №4: Скомпилировать проект.

Шаг №5: Запустить проект.

Если компиляция прошла неудачно (a.k.a. "О Боже, что-то пошло не так!")

Всё нормально, без паники. Скорее всего, это какой-то пустяк.

- Во-первых, убедитесь, что вы написали код правильно: без ошибок и опечаток. Сообщение об ошибке компилятора может дать вам ключ к пониманию того, где и какие ошибки случились.
- Во-вторых, просмотрите Урок №10 — там есть решения наиболее распространенных проблем.
- Если всё вышесказанное не помогло — "загуглите" проблему. С вероятностью 90% кто-то уже сталкивался с этим раньше и нашел решение.

Заключение

Поздравляем, вы написали, скомпилировали и запустили свою первую программу на языке C++! Не беспокойтесь, если вы не понимаете, что означает весь этот код, приведенный выше. Мы детально всё это рассмотрим на следующих уроках.

Урок №6. Режимы конфигурации «Debug» и «Release»

Конфигурация сборки (англ. "*build configuration*") — это набор настроек проекта, которые определяют принцип его построения. Конфигурация сборки состоит из:

- имени исполняемого файла;
- имени директории исполняемого файла;
- имен директорий, в которых IDE будет искать другой код и файлы библиотек;
- информации об отладке и параметрах оптимизации вашего проекта.

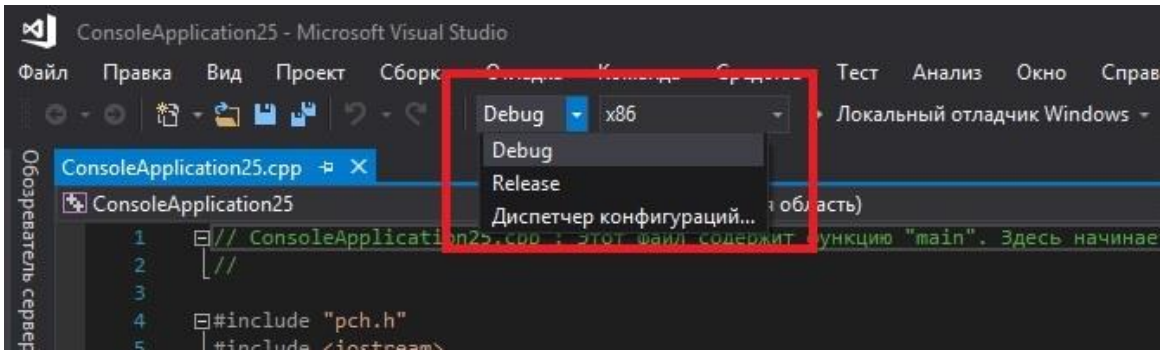
Интегрированная среда разработки имеет две конфигурации сборки: "Release" (Релиз) и "Debug" (Отладка).

- **Конфигурация "Debug"** предназначена для отладки вашей программы. Эта конфигурация отключает все настройки по оптимизации, включает информацию об отладке, что делает ваши программы больше и медленнее, но упрощает проведение отладки. Режим "Debug" обычно используется в качестве конфигурации по умолчанию.
- **Конфигурация "Release"** используется во время сборки программы для её дальнейшего выпуска. Программа оптимизируется по размеру и производительности и не содержит дополнительную информацию об отладке.

Например, исполняемый файл программы "Hello, World!" из предыдущего урока, созданный в конфигурации "Debug", у меня занимал 65КБ, в то время как исполняемый файл, построенный в конфигурации "Release", занимал всего лишь 12КБ.

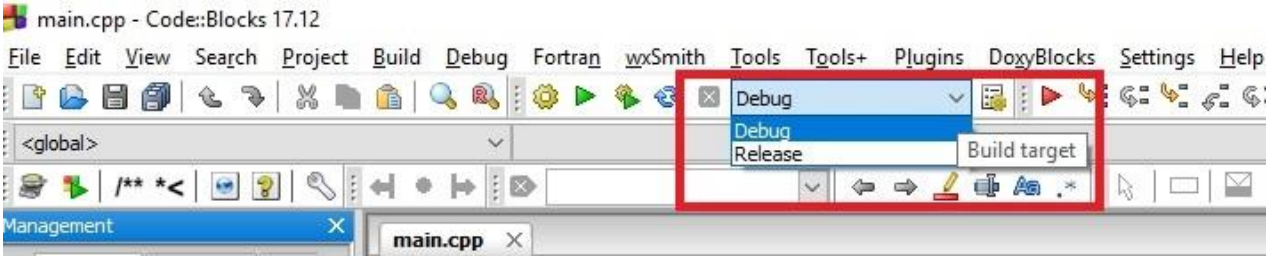
Переключение между режимами "Debug" и "Release" в Visual Studio

Самый простой способ изменить конфигурацию проекта — выбрать соответствующую из выпадающего списка на панели быстрого доступа:



Переключение между режимами "Debug" и "Release" в Code::Blocks

В Code::Blocks на панели быстрого доступа есть также выпадающий список, где вы можете выбрать соответствующий режим конфигурации:



Заключение

Используйте конфигурацию "Debug" при разработке программ, а конфигурацию "Release" при их релизе.

Урок №7. Конфигурация компилятора: Расширения компилятора

Стандарт языка C++ определяет правила поведения программ при определенных обстоятельствах. И в большинстве случаев компиляторы также будут следовать этим правилам. Однако многие компиляторы вносят свои собственные изменения в язык программирования, часто для улучшения совместимости с другими версиями языка (например, C99), или по историческим причинам. Эти специфичные для компилятора варианты поведения называются **расширениями компилятора**.

Используя расширения компилятора, вы получаете возможность писать программы, несовместимые со стандартом языка C++. Программы, использующие нестандартные расширения, обычно не компилируются другими компиляторами (которые не поддерживают эти же расширения), или вообще могут работать не так, как нужно.

К сожалению, расширения компилятора часто включены по умолчанию. Это особенно вредно для новичков в программировании, которые могут подумать, что специфическое поведение, вызванное расширениями компилятора, является частью официального стандарта языка C++ (когда, на самом деле, это не так).

Поскольку расширения компилятора очень редко требуются для решения большинства задач, и приводят к тому, что ваши программы не соответствуют стандарту языка C++, то рекомендуется отключать расширения компилятора.

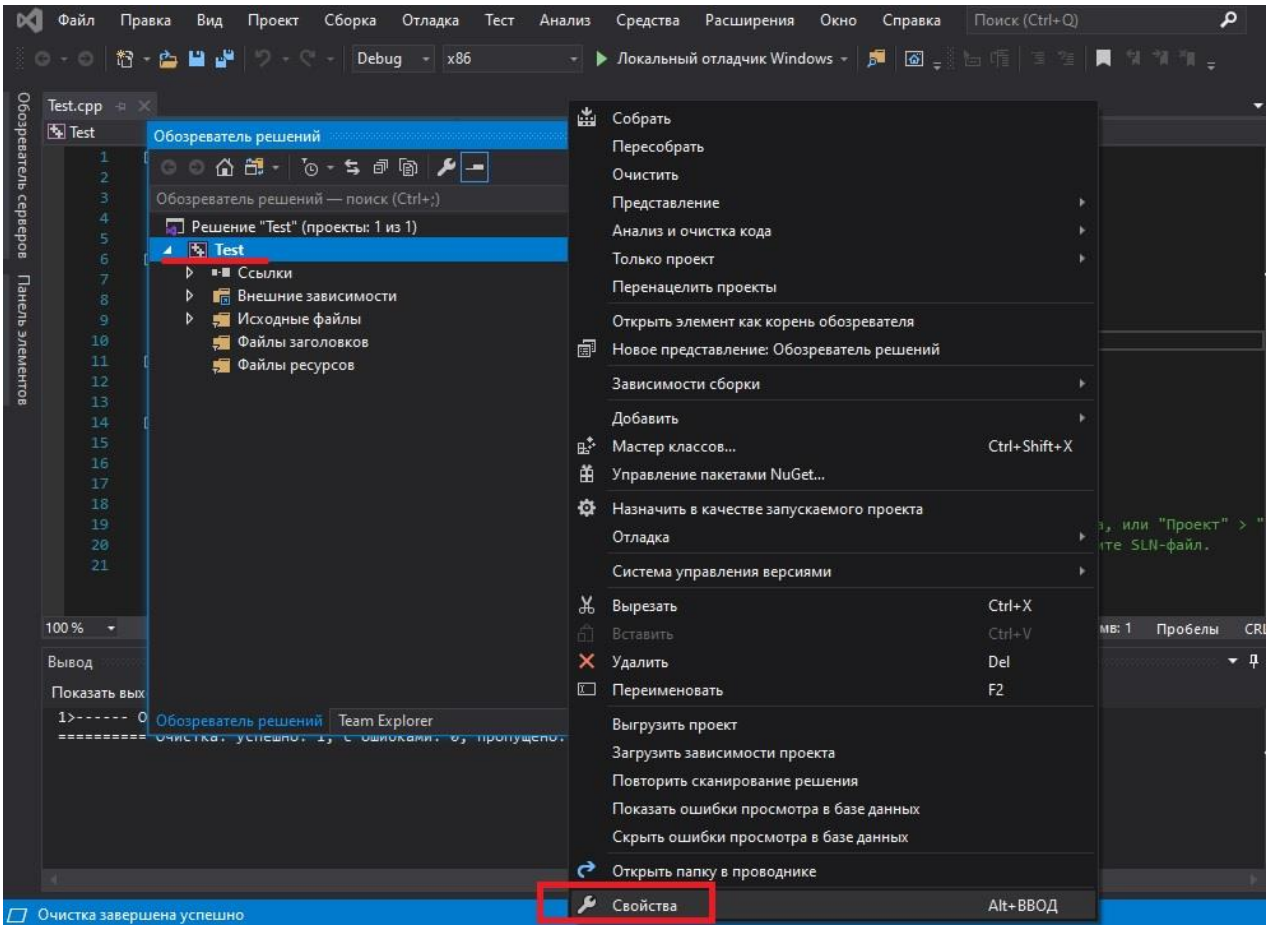
> ***Совет:*** Отключите расширения компилятора, чтобы ваши программы оставались совместимыми со стандартами языка C++ и работали в любой системе.

> ***Примечание:*** Настройки, приведенные ниже, применяются для каждого проекта отдельно. Вам нужно будет это всё проделывать при создании нового проекта, либо создать шаблон с этими настройками и уже по этому шаблону создавать новые проекты.

Отключение расширений компилятора

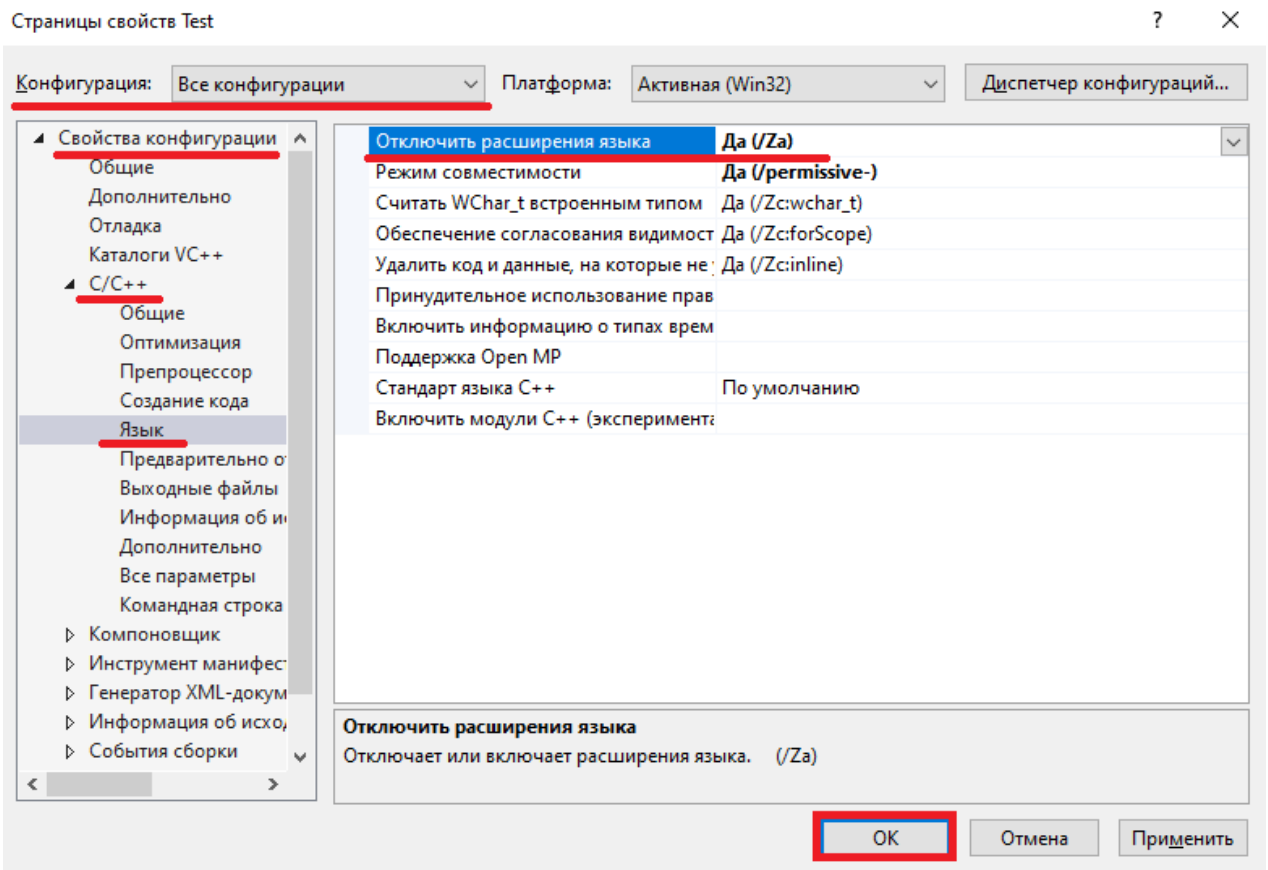
Пользователям Visual Studio

Чтобы отключить расширения компилятора в Visual Studio, щелкните правой кнопкой мыши по названию вашего проекта в `"Обозреватель решений" > "Свойства"`:



В диалоговом окне вашего проекта убедитесь, что в пункте `"Конфигурация"` установлено значение `"Все конфигурации"`.

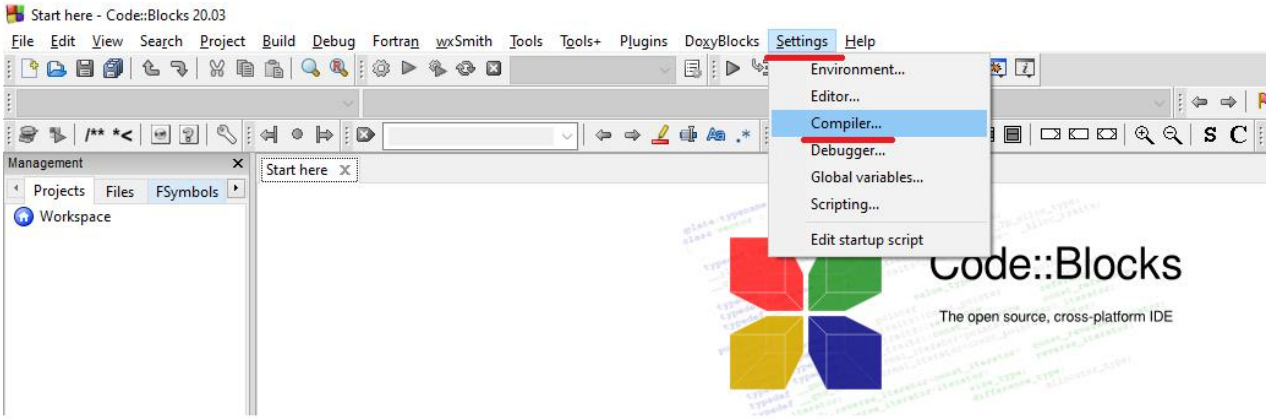
Затем перейдите на вкладку `"C/C++" > "Язык"` и в пункте `"Отключить расширения языка"` выберите значение `"Да (/Za)"`:



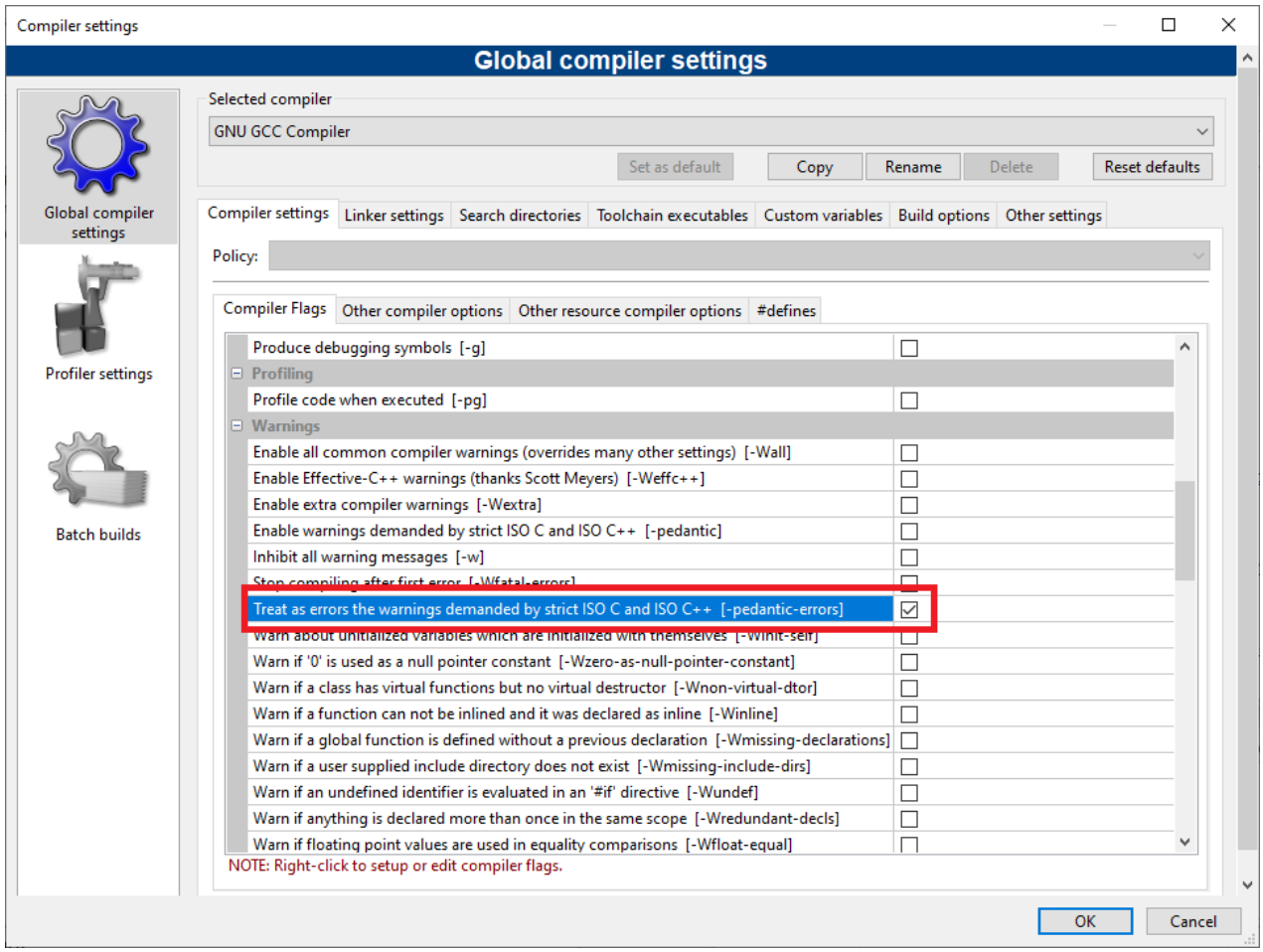
После этого нажмите `"Применить"` и `"OK"`.

Пользователям Code::Blocks

Отключить расширения компилятора можно через меню `"Settings" > "Compiler"`:



Затем на вкладке `"Compiler Flags"` найдите пункт `"Treat as errors the warnings demanded by strict ISO C and ISO C++ [-pedantic-errors]"` и поставьте возле него галочку, после чего нажмите `"OK"`:



Пользователям GCC/G++

Вы можете отключить расширения компилятора, добавив флаг `-pedantic-errors` в командную строку компиляции.

Урок №8. Конфигурация компилятора: Уровни предупреждений и ошибки

На этапе компиляции компилятор проверяет, соответствует ли ваш код правилам языка C++. Если вы сделали что-то запрещенное, что нарушило синтаксис языка C++, то компилятор выдаст ошибку, предоставив вам как номер строки, содержащий ошибку, так и некоторый текст о содержании самой ошибки. Фактически, ошибка может находиться как в этой строке (которую сообщил вам компилятор), так и в строке перед ней. После того, как вы определили и исправили ошибочные строки кода, вы можете попробовать скомпилировать вашу программу еще раз.

Еще могут быть ситуации, когда компилятор видит ошибочный код, но не до конца в этом уверен (помните, что философия языка C++ заключается в выражении «Доверяй программисту!»). В таких случаях компилятор может выдать предупреждение. Предупреждения не останавливают процесс компиляции, но сообщают программисту, что что-то пошло не так.

> ***Совет:*** Не позволяйте предупреждениям накапливаться. Решайте их по мере возникновения (так, как будто бы это были ошибки).

В большинстве случаев предупреждения могут быть устранены либо путем исправления ошибки, на которую указывает предупреждение, либо путем переписывания строки кода, генерирующей предупреждение, таким образом, чтобы предупреждение больше не генерировалось.

В редких случаях может потребоваться явно указать компилятору не генерировать конкретное предупреждение для рассматриваемой строки кода. Язык C++ не поддерживает такой способ решения предупреждений, но многие отдельные компиляторы (включая Visual Studio и GCC) предоставляют возможность (через *не портативные* директивы #pragma) временного отключения предупреждений.

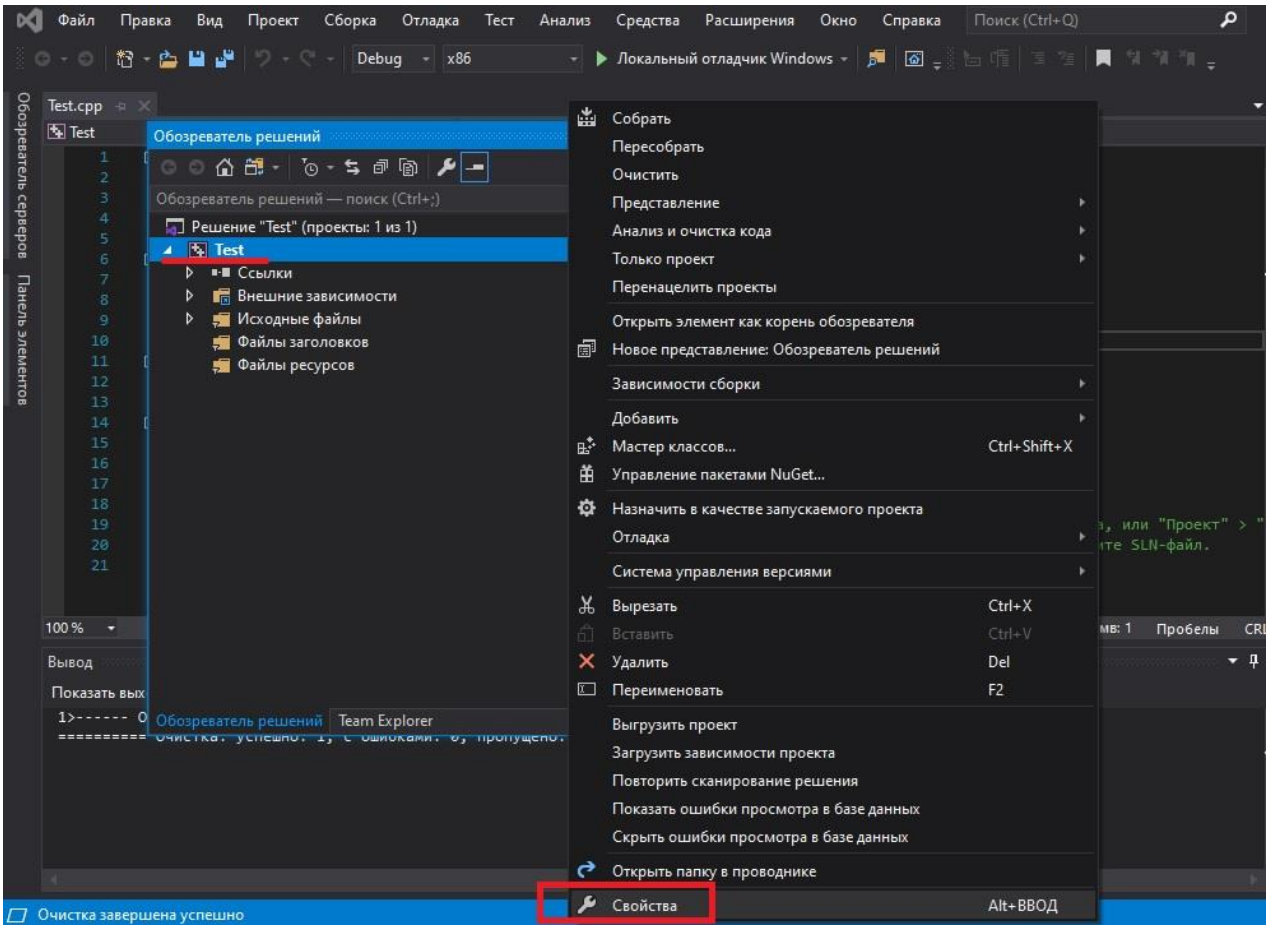
По умолчанию большинство компиляторов генерируют только предупреждения о наиболее очевидных проблемах. Однако вы можете попросить ваш компилятор быть более настойчивым в предоставлении предупреждений о вещах, которые он считает странными.

> ***Совет:*** Сделайте максимальным уровень предупреждений от компилятора (особенно во время обучения). Это поможет вам определить возможные проблемы.

Изменение уровня предупреждений

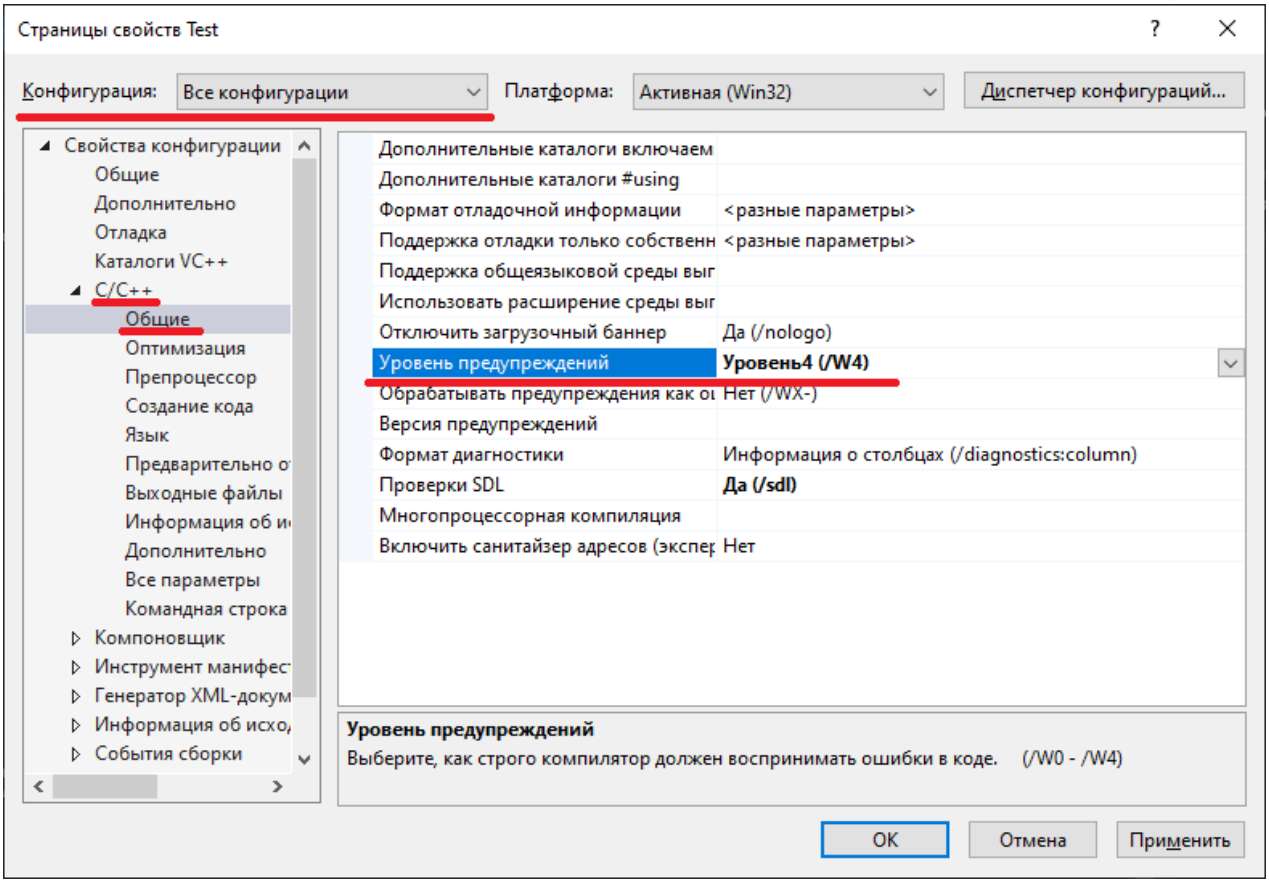
Пользователям Visual Studio

Чтобы повысить уровень предупреждений в Visual Studio, щелкните правой кнопкой мышки по названию вашего проекта в меню `"Обозреватель решений" > "Свойства"`:



В диалоговом окне вашего проекта убедитесь, что в пункте `"Конфигурация"` установлено значение `"Все конфигурации"`.

Затем перейдите на вкладку `"C/C++" > "Общие"` и в пункте `"Уровень предупреждений"` выберите значение `"Уровень4 (/W4)"`:

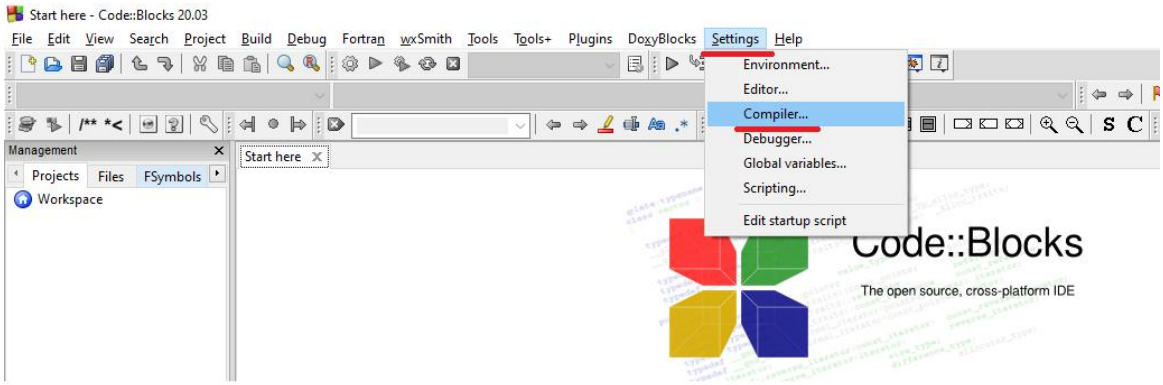


Затем нажмите `"Применить"` и `"OK"`.

> ***Примечание:*** Не выбирайте пункт `"Включить все предупреждения (/Wall)"`, иначе вы будете погребены в предупреждениях, генерируемых Стандартной библиотекой C++.

Пользователям Code::Blocks

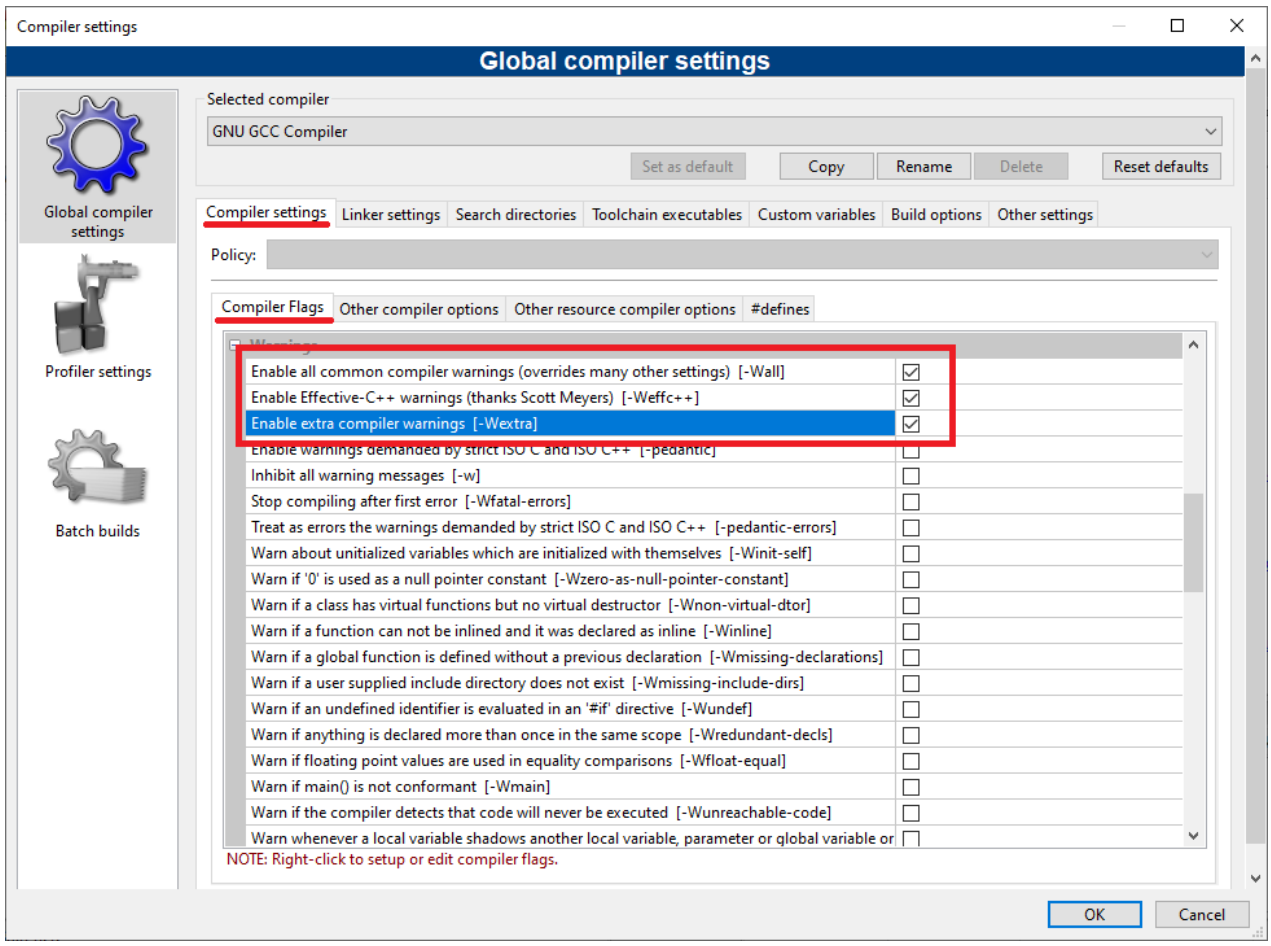
Перейдите в меню `"Settings" > "Compiler"`:



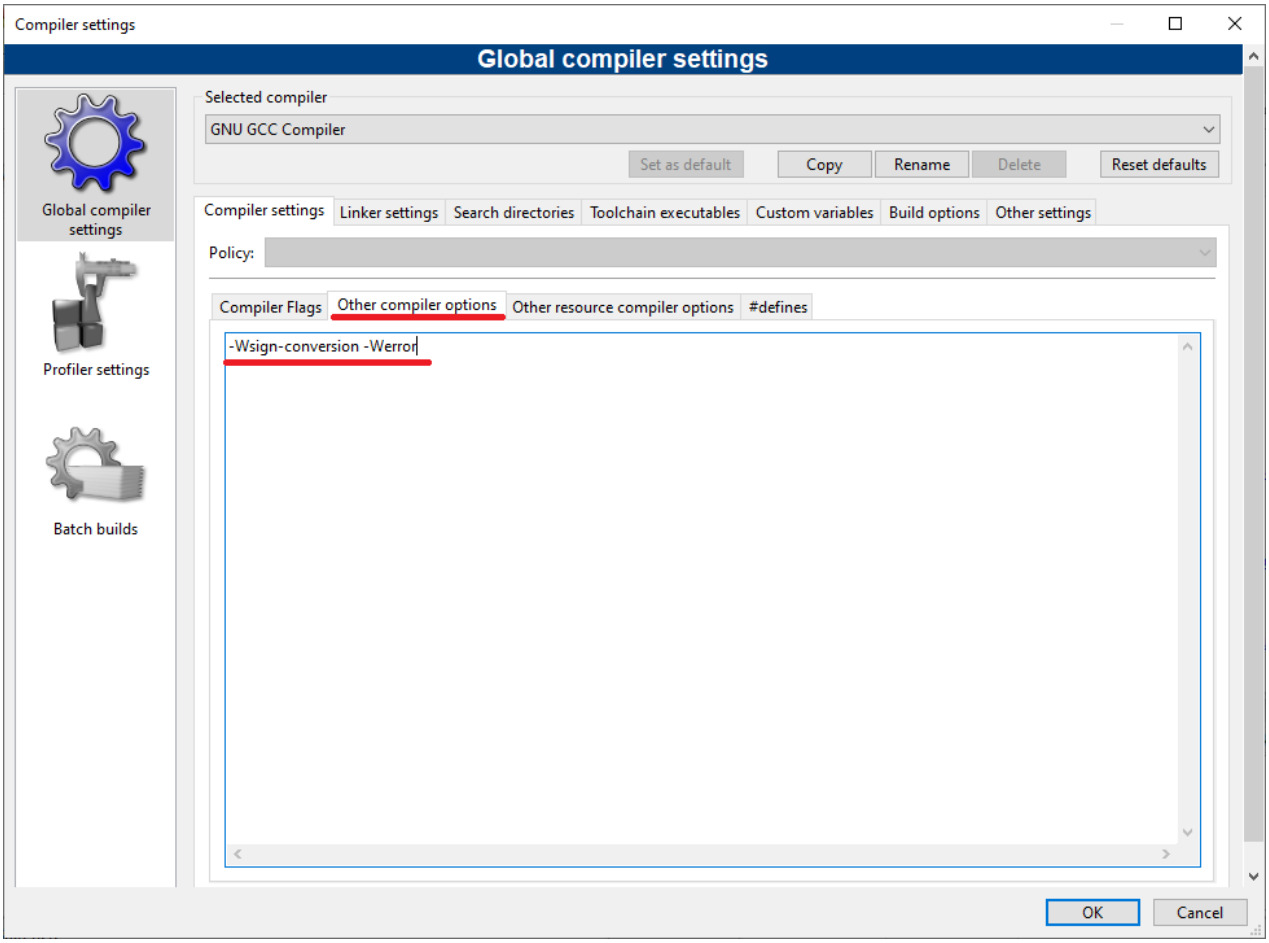
И на вкладке `"Compiler settings" > "Compiler flags"` поставьте галочки возле следующих пунктов:

- `Enable all common compiler warnings (overrides many other settings) [-Wall]`
- `Enable Effective-C++ warnings (thanks Scott Meyers) [-Weffc++]`
- `Enable extra compiler warnings [-Wextra]`

Смотрим:



Нажмите `"OK"` и затем перейдите на вкладку `"Other compiler options"` и добавьте в область редактирования текст `-Wsign-conversion -Werror`:



И нажмите `"Ок"`.

> ***Примечание:*** О параметре `-Werror` мы поговорим чуть позже.

Пользователям GCC/G++

Добавьте следующие флаги в вашу командную строку:

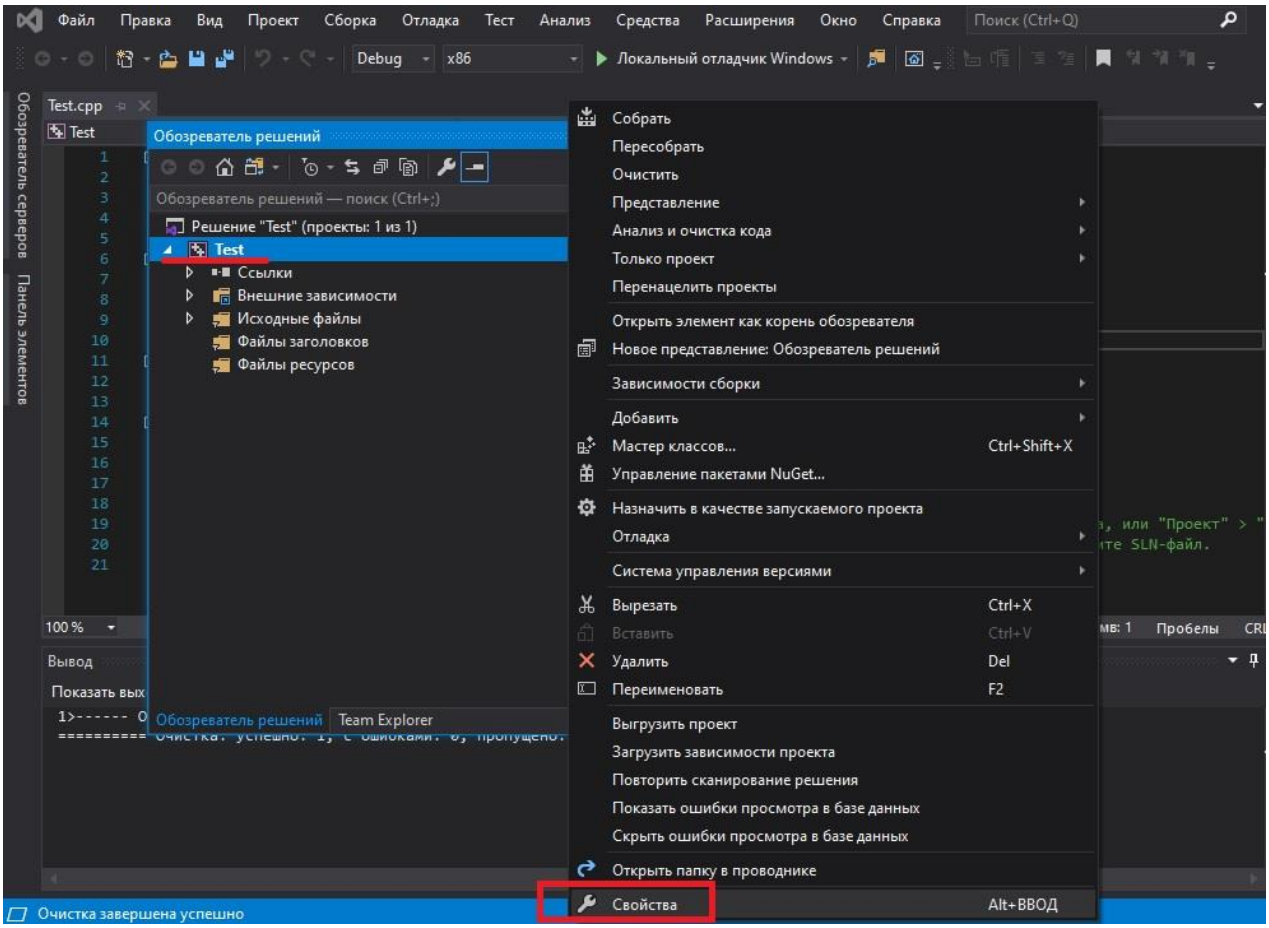
```
-Wall -Weffc++ -Wextra -Wsign-conversion
```

Обрабатывать предупреждения как ошибки

Вы также можете указать вашему компилятору обрабатывать все предупреждения так, как если бы они были ошибками (в таком случае, компилятор будет останавливать процесс компиляции, если обнаружит какие-либо предупреждения). Это хороший вариант заставить себя исправлять все предупреждения, особенно, если вам не хватает самодисциплины (как, впрочем, большинству из нас).

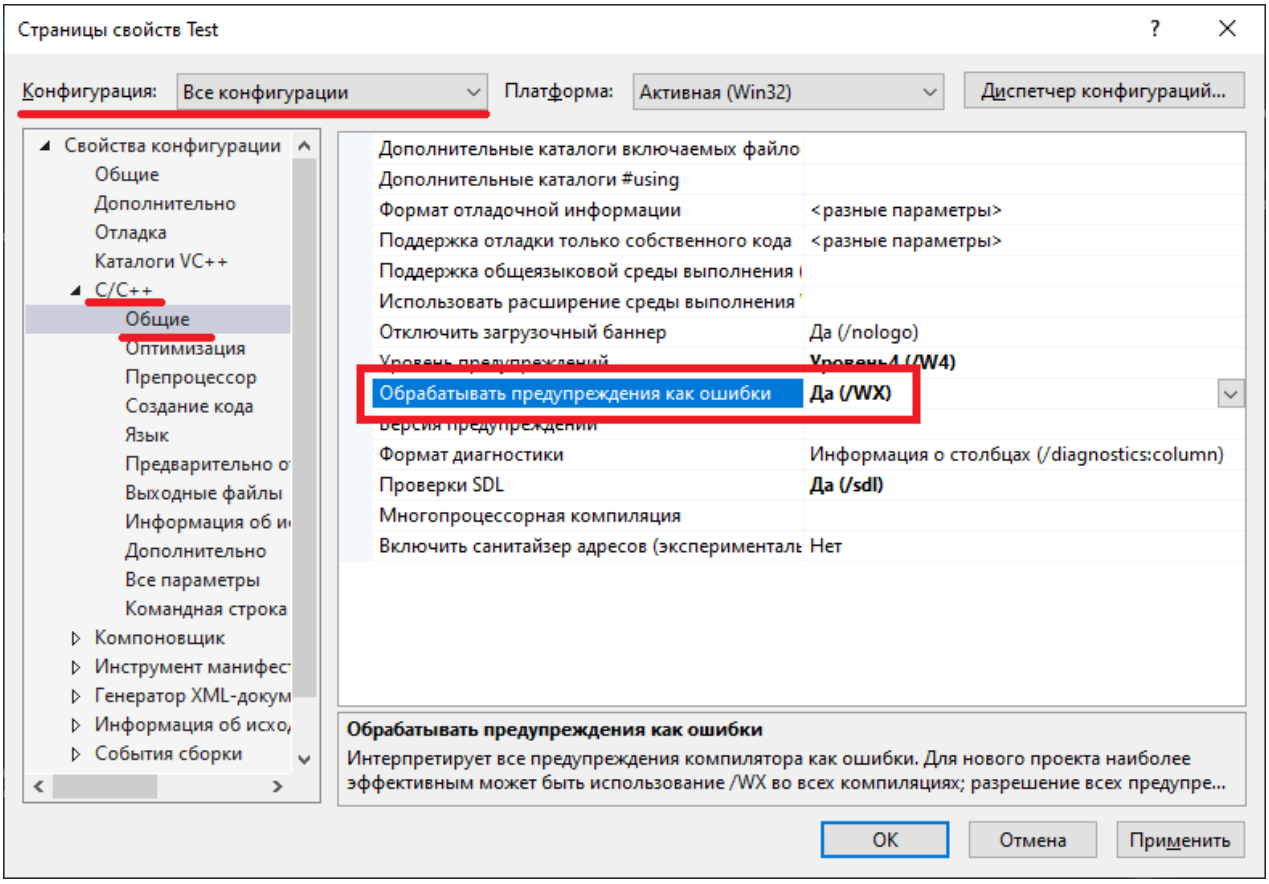
Пользователям Visual Studio

Чтобы обрабатывать все предупреждения как ошибки, щелкните правой кнопкой мышки по названию вашего проекта в меню `"Обозреватель решений" > "Свойства"`:



В диалоговом окне вашего проекта убедитесь, что в поле `"Конфигурация"` установлено значение `"Все конфигурации"`.

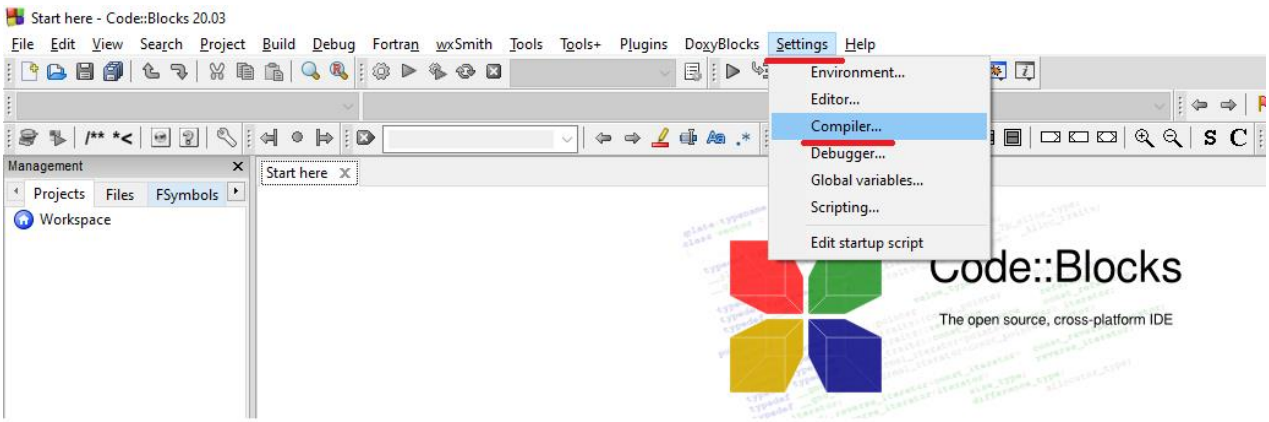
Затем перейдите на вкладку `"C/C++" > "Общие"` и в пункте `"Обрабатывать предупреждения как ошибки"` выберите значение `"Да(/WX)"`:



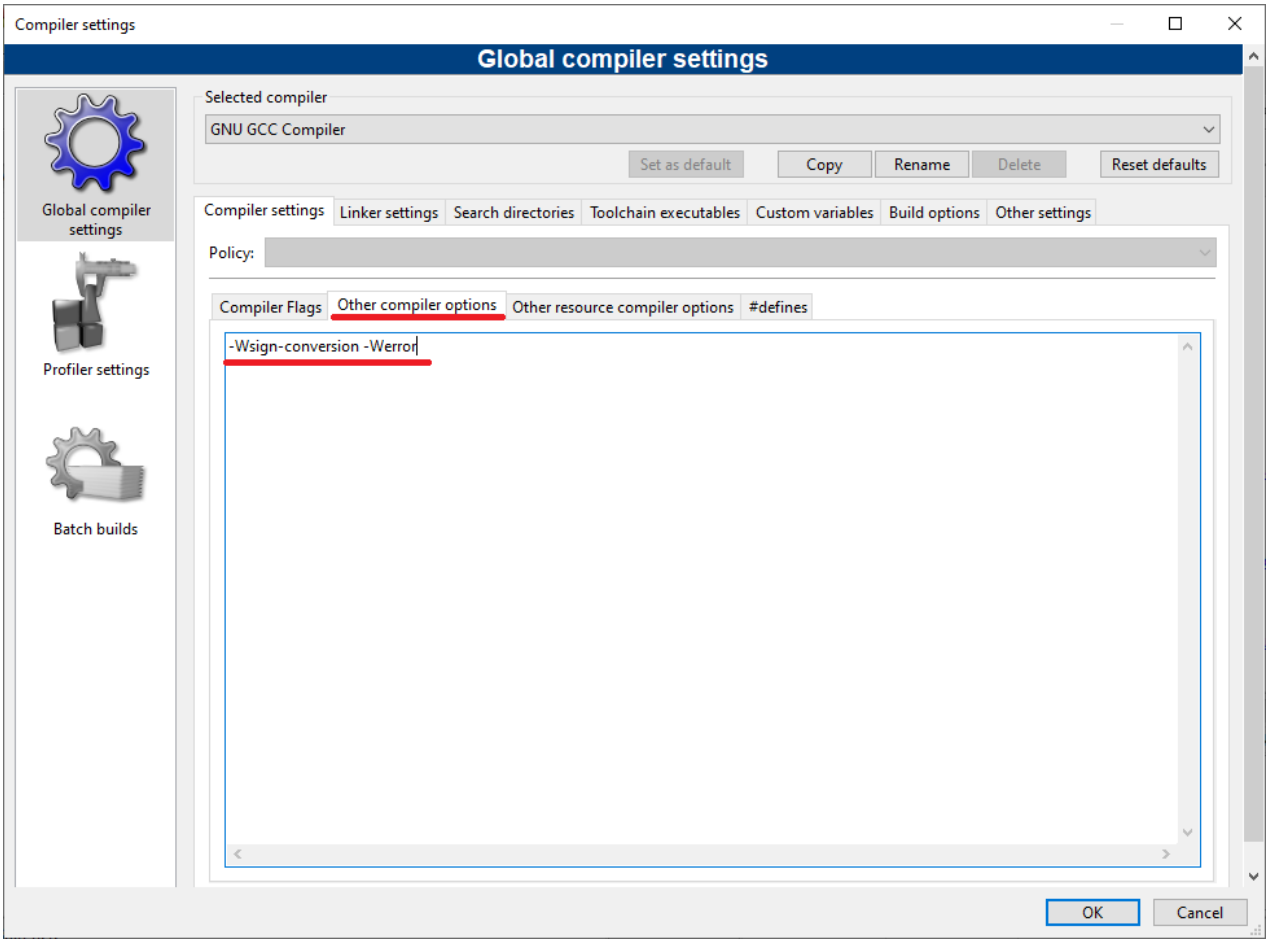
Затем нажмите `"Применить"` и `"OK"`.

Пользователям Code::Blocks

Перейдите в меню `"Settings" > "Compiler"`:



Затем `"Compiler settings" > "Other compiler options"` и добавьте `-Werror` в область редактирования:



Затем нажмите `"OK"`.

Пользователям GCC/G++

Добавьте следующий флаг в вашу командную строку:

`-Werror`

Урок №9. Конфигурация компилятора: Выбор стандарта языка C++

Как с огромным количеством различных версий языка C++ (C++98, C++03, C++11**,** C++14**,** C++17, C++20) компилятор понимает, какую из них ему следует использовать? Как правило, компилятор выбирает стандарт языка по умолчанию (часто не самый последний языковой стандарт). Если вы хотите использовать другой стандарт, то вам придется внести изменения в настройки вашей IDE/компилятора. Эти настройки применяются только к текущему проекту. При создании нового проекта вам придется всё делать заново.

Кодовые имена для версий языка C++

Обратите внимание на то, что каждый языковый стандарт имеет название, указывающее на год его принятия/утверждения (например, C++17 был принят/утвержден в 2017 году).

Однако, когда согласовывается новый языковой стандарт, неясно, в каком году удастся его принять, поэтому действующим языковым стандартам присваиваются кодовые имена, которые затем заменяются фактическими именами при доработке стандарта. Например, C++11 назывался c++1x, пока над ним вели работу. Вы можете по-прежнему видеть на просторах Интернета подобные кодовые имена (особенно, когда речь заходит о будущей версии языкового стандарта, у которого еще нет окончательного названия).

Вот сопоставление кодовых имен версий C++ с их окончательными названиями:

- c++1x = C++11
- c++1y = C++14
- c++1z = C++17
- c++2a = C++20

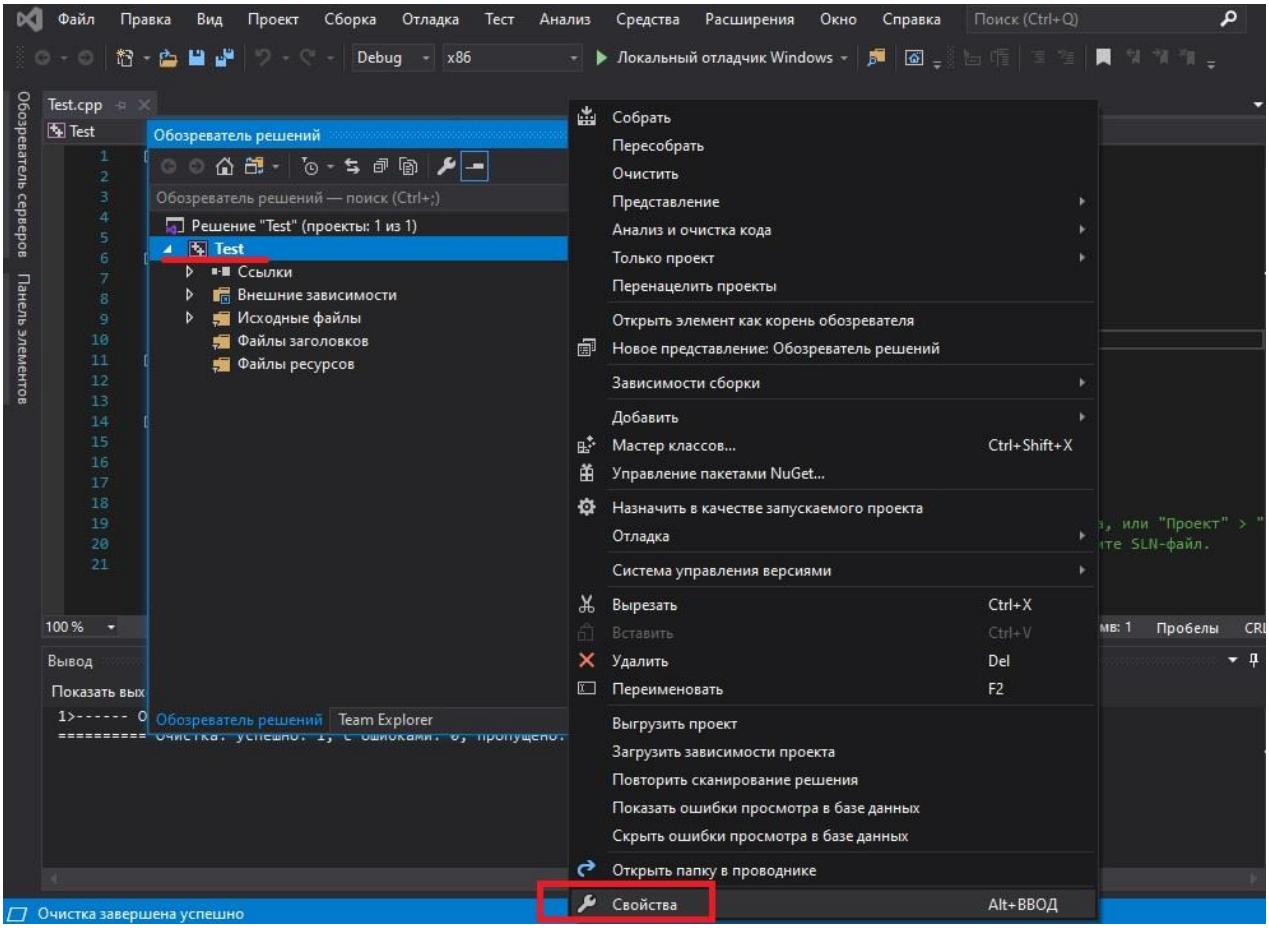
Например, если вы видите c++1z, то знайте, что речь идет о стандарте C++17.

Установка стандарта языка C++ в Visual Studio

На момент написания данной статьи, Visual Studio 2019 по умолчанию использует функционал C++14, что не позволяет использовать более новые возможности, представленные в C++17 и в C++20.

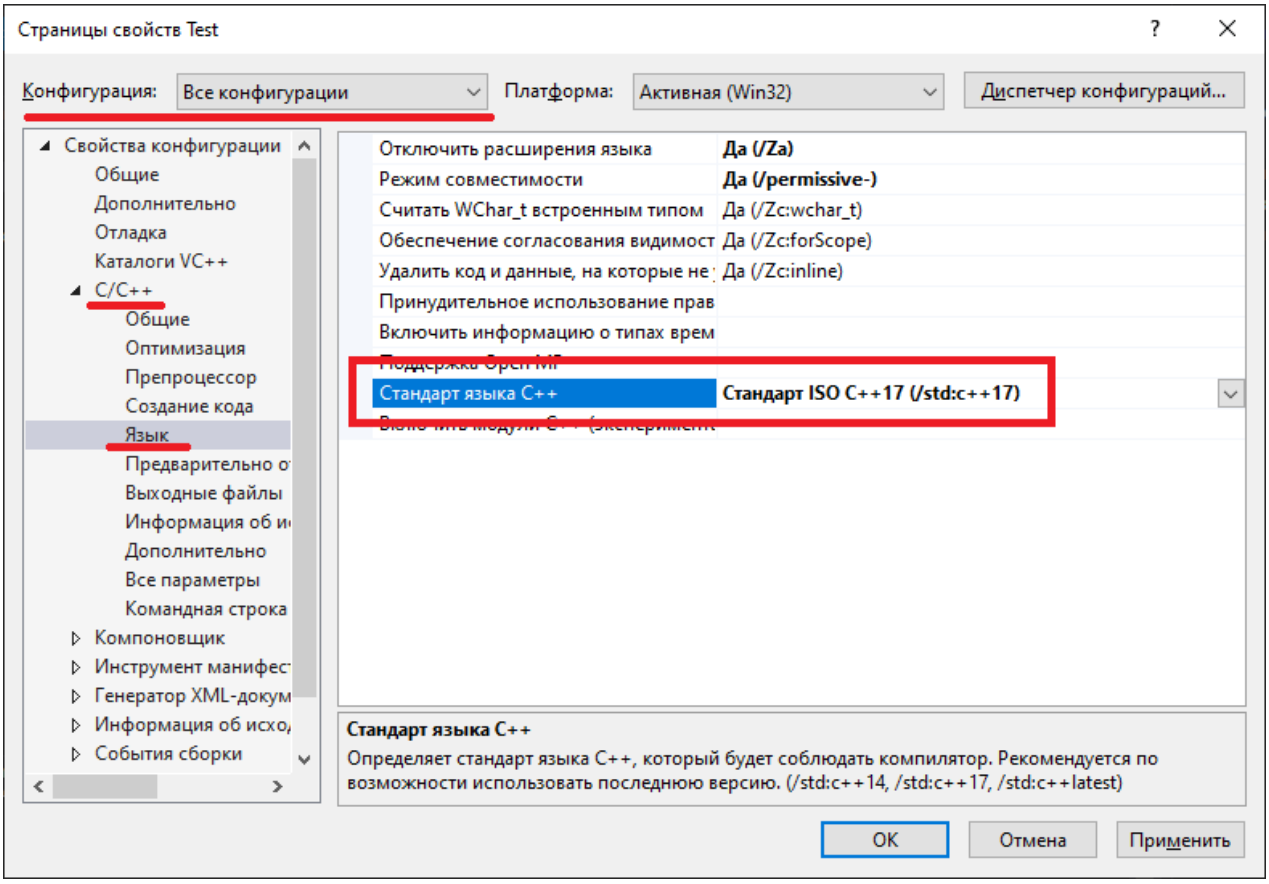
Чтобы использовать новый функционал, вам необходимо подключить новый языковой стандарт. К сожалению, сейчас нет способа сделать это глобально — вы должны делать это к каждому проекту индивидуально.

Чтобы использовать новый языковый стандарт в Visual Studio, откройте ваш проект, затем щелкните правой кнопкой мышки по названию вашего проекта в меню `"Обозреватель решений" > "Свойства"`:



В диалоговом окне вашего проекта убедитесь, что в пункте `"Конфигурация"` установлено значение `"Все конфигурации"`.

Затем перейдите на вкладку `"C/C++" > "Язык"` и в пункте `"Стандарт языка C++"` выберите ту версию языка C++, которую хотели бы использовать:



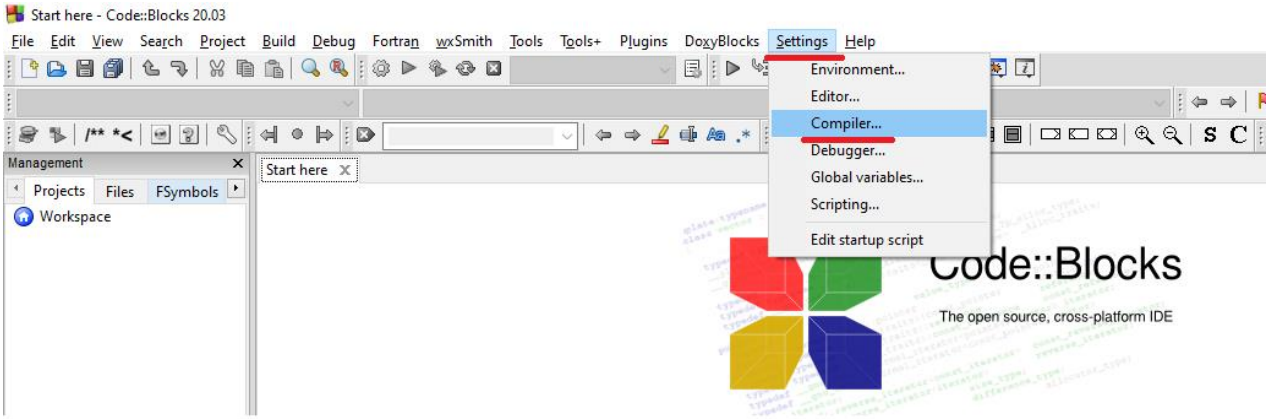
На момент написания данной статьи, я рекомендую выбрать `"Стандарт ISO C++17 (/std:c++17)"`, который является последним стабильным стандартом.

Если вы хотите поэкспериментировать с возможностями грядущего стандарта языка C++ — C++20, то вы можете выбрать пункт `"Предварительная версия ... (/std:c++latest)"`. Просто помните, что его поддержка может иметь ошибки.

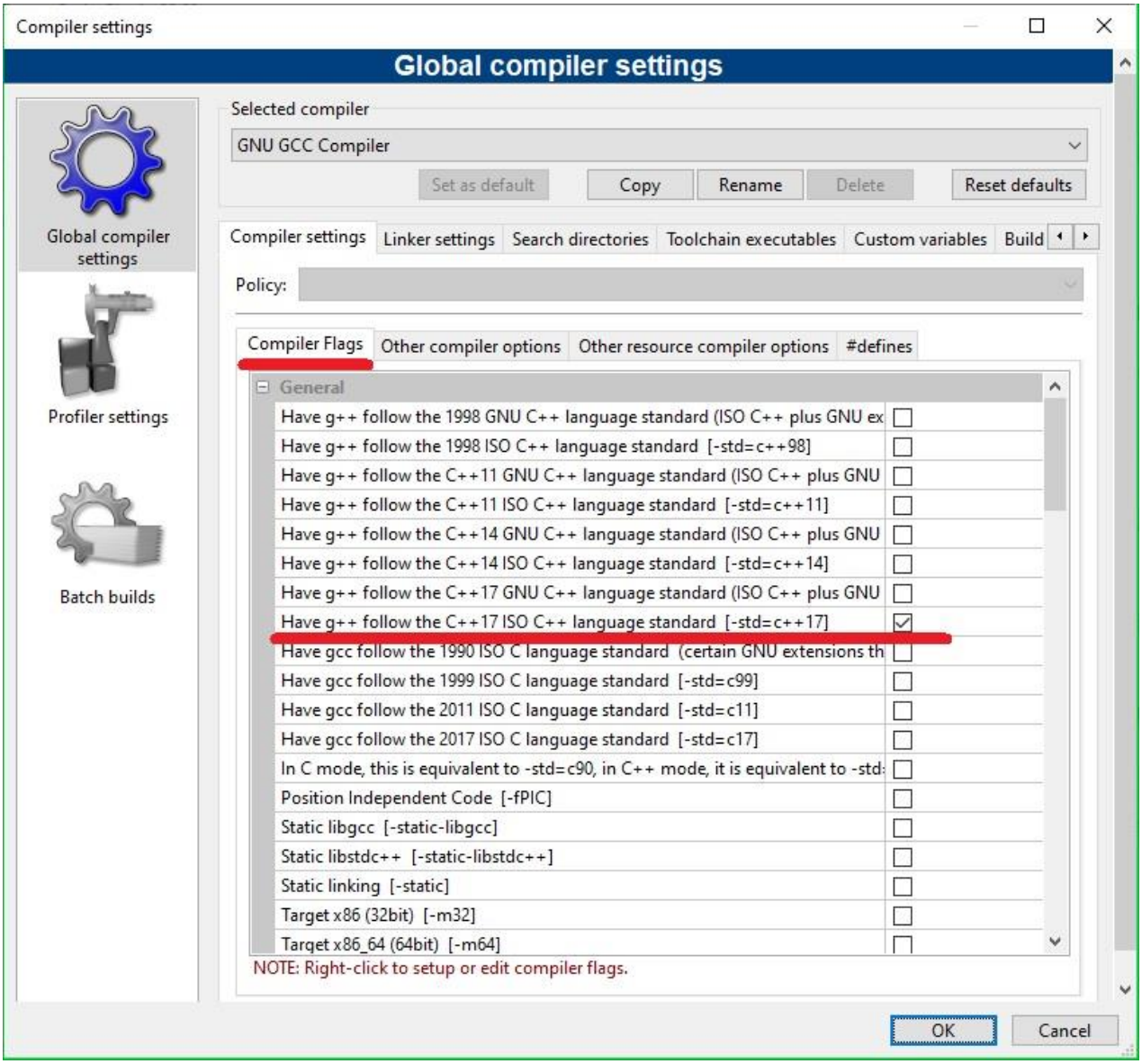
Установка стандарта языка C++ в Code::Blocks

Code::Blocks по умолчанию может использовать стандарт C++11. Хорошей новостью является то, что Code::Blocks позволяет устанавливать ваш стандарт языка C++ глобально, поэтому вы можете установить его один раз и сразу на все проекты (а не для каждого проекта в индивидуальном порядке).

Для этого перейдите в меню `"Settings" > "Compiler"`:



Затем на вкладке `"Compiler flags"` найдите следующие пункты меню:



Отметьте тот пункт, у которого число обозначает ближайший (к текущему) год утверждения стабильной версии и нажмите `"OK"` (на вышеприведенном скриншоте этим пунктом является `"Have g++ follow the C++17 ISO C++ language standard [-std=c++17]"`).

> ***Примечание:*** Если вы не нашли в ваших настройках опцию с C++17, то вам следует обновить вашу версию Code::Blocks.

Установка стандарта языка C++ в GCC/G++

В GCC/G++ вы можете прописать соответствующие флаги `-std=c++11`, `-std=c++14`, `-std=c++17` или `-std=c++2a`, чтобы подключить функционал C++11/14/17/20, соответственно.

Тестирование вашего компилятора

После подключения версии C++17 или выше, вы можете провести тест, который позволит понять, всё ли верно сделано и действительно ли подключена новая версия языка C++. Следующая программа в C++17 должна выполниться без каких-либо предупреждений или ошибок:

```
#include <array>
#include <iostream>
#include <string_view>
#include <tuple>
#include <type_traits>

namespace a::b::c
{
    inline constexpr std::string_view str{ "hello" };
}

template <class... T>
std::tuple<std::size_t, std::common_type_t<T...>> sum(T... args)
{
    return { sizeof...(T), (args + ...) };
}

int main()
{
    auto [iNumbers, iSum]{ sum(1, 2, 3) };
    std::cout << a::b::c::str << ' ' << iNumbers << ' ' << iSum << '\n';

    std::array arr{ 1, 2, 3 };

    std::cout << std::size(arr) << '\n';

    return 0;
}
```

Если вы не можете скомпилировать этот код, то либо вы не подключили C++17, либо ваш компилятор не полностью поддерживает C++17. В последнем случае обновите версию IDE или компилятора.

Урок №10. Решения самых распространенных проблем

На этом уроке мы рассмотрим наиболее частые проблемы, с которыми сталкиваются новички при написании программ на языке C++.

Проблема №1

Как использовать кириллицу в программах C++?

Ответ №1

Чтобы выводить кириллицу в языке C++ нужно подключить заголовочный файл <Windows.h>:

```
1. #include <Windows.h>
```

И прописать следующие две строки в функции main():

```
1. SetConsoleCP(1251);
2. SetConsoleOutputCP(1251);
```

В качестве альтернативного варианта можно использовать следующую строку в функции main():

```
1. setlocale(LC_ALL, "Russian");
```

Проблема №2

При выполнении программы появляется черное консольное окно, а затем сразу пропадает.

Ответ №2

Некоторые компиляторы (например, Bloodshed's Dev C++) автоматически не задерживают консольное окно после того, как программа завершила свое выполнение. Если проблема в компиляторе, то следующие два шага решат эту проблему:

***Шаг №1:* Добавьте следующую строку кода в верхнюю часть вашей программы:**

```
1. #include <iostream>
```

Шаг №2: **Добавьте следующий код в конец функции main() (прямо перед оператором return):**

```
std::cin.clear();
std::cin.ignore(32767, '\n');
std::cin.get();
```

Таким образом, программа будет ожидать нажатия клавиши, чтобы закрыть консольное окно. Вы получите дополнительное время, чтобы хорошенько всё рассмотреть/изучить. После нажатия любой клавиши, консольное окно закроется.

Другие решения, такие как `system("pause");`, могут работать только на определенных операционных системах, поэтому вариант, приведенный выше, предпочтительнее.

> ***Примечание:*** Visual Studio не задерживает консольное окно, если выполнение запущено с отладкой (`"Отладка" > "Начать отладку"` или F5). Если вы хотите, чтобы была пауза — воспользуйтесь решением выше или запустите программу без отладки (`"Отладка" > "Запуск без отладки"` или Ctrl+F5).

Проблема №3

При использовании cin, cout или endl компилятор говорит, что cin, cout или endl являются *"undeclared identifier"* (необъявленными идентификаторами).

Ответ №3

Во-первых, убедитесь, что у вас присутствует следующая строка кода в верхней части вашей программы:

```
#include <iostream>
```

Во-вторых, убедитесь, что `cin`, `cout` или `endl` имеют префикс `std::`, например:

```
std::cout << "Hello world!" << std::endl;
```

Проблема №4

При использовании endl для перехода на новую строку, появляется ошибка, что end1 является *"undeclared identifier"*.

Ответ №4

Убедитесь, что вы не перепутали букву `l` (нижний регистр `L`) в `endl` с цифрой `1`. В `endl` все символы являются буквами. Также легко можно перепутать заглавную букву `O` с цифрой `0` (ноль).

Проблема №5

Моя программа компилируется, но работает не так, как нужно. Что мне делать?

Ответ №5

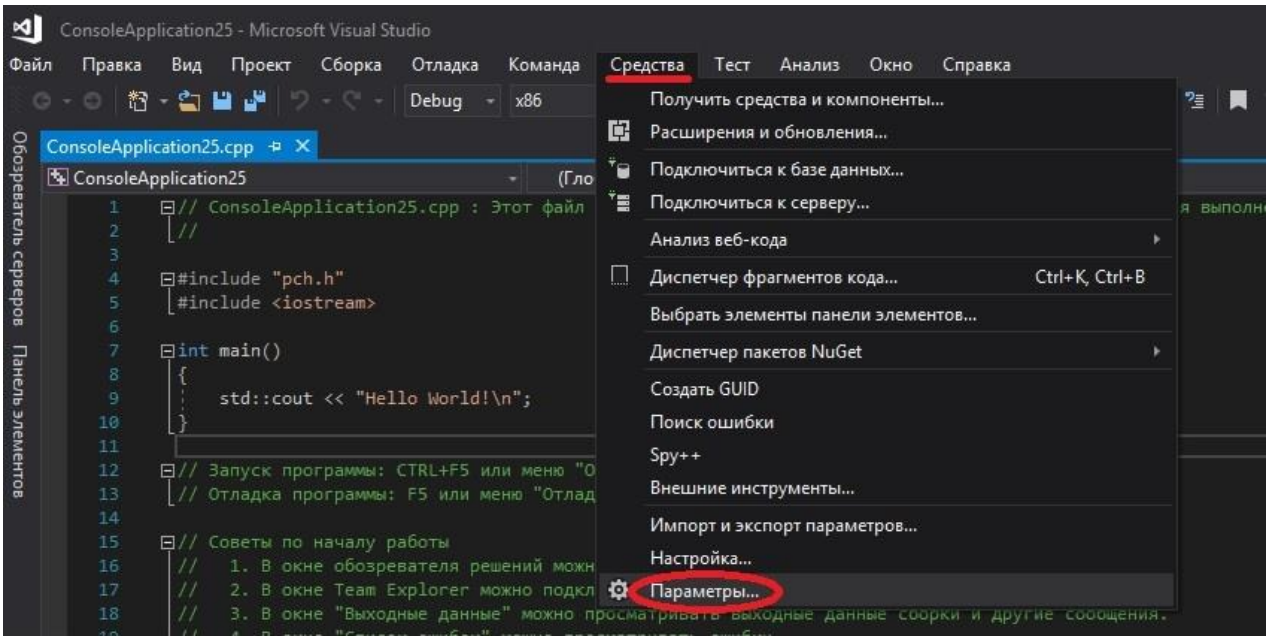
Выполните отладку программы. Детально об этом читайте на уроке №29 и на уроке №30.

Проблема №6

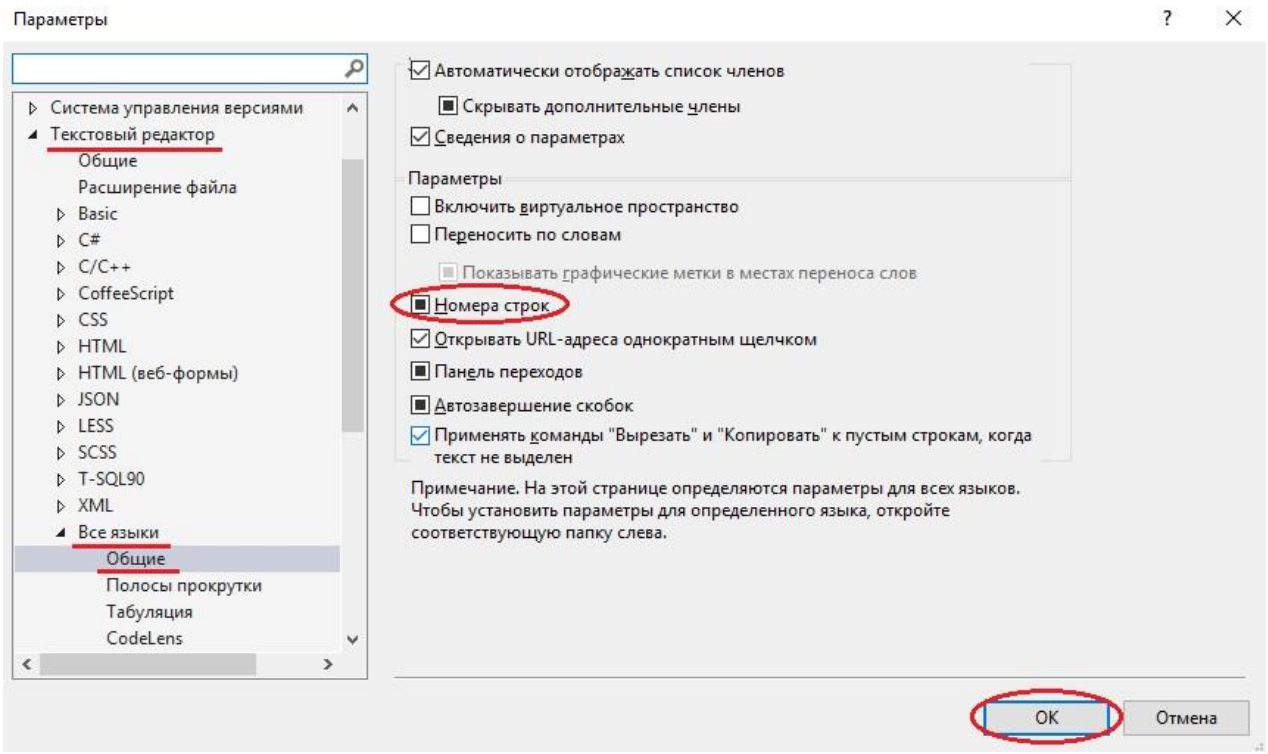
Как включить нумерацию строк в Visual Studio?

Ответ №6

Перейдите в меню `"Средства" > "Параметры"`:



Затем откройте вкладку `"Текстовый редактор" > "Все языки" > "Общие"` и поставьте галочку возле `"Номера строк"`, затем нажмите `"OK"`:



Проблема №7

При компиляции программы я получаю следующую ошибку: "unresolved external symbol _main or _WinMain@16".

Ответ №7

Это означает, что ваш компилятор не может найти главную функцию main(). Все программы должны содержать эту функцию.

Здесь есть несколько пунктов, которые нужно проверить:

- Есть ли в вашей программе функция `main()`?
- Слово `main` написано правильно?
- Подключен ли файл, который содержит функцию `main()`, к вашему проекту? (если нет, то переместите функцию `main()` в файл, который является частью вашего проекта, либо добавьте этот файл в ваш проект)
- Подключен ли файл, содержащий функцию `main()`, к компиляции?

Проблема №8

При компиляции программы я получаю следующее предупреждение: "Cannot find or open the PDB file".

Ответ №8

Это не ошибка, а предупреждение. На работоспособность вашей программы оно не повлияет. Тем не менее, в Visual Studio вы можете решить всё следующим образом: перейдите в меню `"Отладка" > "Параметры" > "Отладка" > "Символы"` и поставьте галочку возле `"Серверы символов (Майкрософт)"`, затем нажмите `"OK"`.

Проблема №9

Я использую Code::Blocks или G++, но функционал C++11/C++14 не работает.

Ответ №9

В Code::Blocks перейдите в `"Project" > "Build options" > "Compiler settings" > "Compiler flags"` и поставьте галочку возле пункта `"Have g++ follow C++14 ISO C++ language standard"`. Смотрите урок №4 — там есть скриншоты, как это сделать.

При компиляции в g++, добавьте следующий код в командную строку:

```
-std=c++14
```

Проблема №10

Я запустил программу, появилось консольное окно, но ничего не выводится.

Ответ №10

Ваш антивирус может блокировать выполнение вашей программы. Попробуйте отключить его на время и запустите программу еще раз.

У меня есть другая проблема, с которой я не могу разобраться. Как и где я могу получить ответ?

По мере прохождения данных уроков, у вас, несомненно, появятся вопросы или вы столкнетесь с проблемами. Что делать?

Во-первых, **спросите у Google**. Четко сформулируйте вашу проблему и просто **"погуглите"**. Если у вас есть сообщение об ошибке - скопируйте его и вставьте в поиск Google, используя кавычки. Скорее всего, кто-то уже задавал такой же вопрос, как у вас, и получил на него ответ.

Если Google не помог, то спросите на специализированных сервисах вопросов/ответов, либо на форумах, посвященным тематике программирования/IT. Вот самые популярные из них:

- **Stack Overflow**
- **CyberForum**
- **Хабр Q&A (раньше Toster)**

Но будьте внимательны и старайтесь максимально конкретизировать свою проблему, укажите, какую операционную систему и IDE вы используете, а также то, что вы пробовали сделать самостоятельно для решения своей проблемы.

Урок №11. Структура программ

Компьютерная программа — это последовательность инструкций, которые сообщают компьютеру, что ему нужно сделать.

Стейтменты

Стейтмент (англ. ***"statement"***) — это наиболее распространенный тип инструкций в программах. Это и есть та самая инструкция, наименьшая независимая единица в языке C++. Стейтмент в программировании — это то же самое, что и "предложение" в русском языке. Мы пишем предложения, чтобы выразить какую-то идею. В языке C++ мы пишем стейтменты, чтобы выполнить какое-то задание. **Все стейтменты в языке C++ заканчиваются точкой с запятой.**

Есть много разных видов стейтментов в языке C++. Рассмотрим самые распространенные из них:

```
1. int x;
2. x = 5;
3. std::cout << x;
```

`int x` — это **стейтмент объявления** (англ. ***"statement declaration"***). Он сообщает компилятору, что `x` является переменной. В программировании каждая переменная занимает определенное число адресуемых ячеек в памяти в зависимости от её типа. Минимальная адресуемая ячейка — байт. Переменная типа int может занимать до 4-х байт, т.е. до 4-х адресуемых ячеек памяти. Все переменные в программе должны быть объявлены, прежде чем использованы. Мы детально поговорим о переменных на следующих уроках.

`x = 5` — это **стейтмент присваивания** (англ. ***"assignment statement"***). Здесь мы присваиваем значение `5` переменной `x`.

`std::cout << x;` — это **стейтмент вывода** (англ. ***"output statement"***). Мы выводим значение переменной `x` на экран.

Выражения

Компилятор также способен обрабатывать выражения. **Выражение** (англ. ***"expression"***) — это математический объект, который создается (составляется) для проведения вычислений и нахождения соответствующего результата. Например, в математике выражение `2 + 3` имеет значение `5`.

Выражения в языке C++ могут содержать:

- отдельные цифры и числа (например, `2`, `45`);
- буквенные переменные (например, `x`, `y`);
- операторы, в т.ч. математические (например, `+`, `-`);
- функции.

Выражения могут состоять как из единичных символов — цифр или букв (например, `2` или `x`), так и из различных комбинаций этих символов с операторами (например, `2 + 3`, `2 + x`, `x + y` или `(2 + x) * (y - 3)`). Для наглядности разберем простой корректный стейтмент присваивания `x = 2 + 3;`. Здесь мы вычисляем результат сложения чисел `2 + 3`, который затем присваиваем переменной `x`.

Функции

В языке C++ стейтменты объединяются в блоки — функции. **Функция** — это последовательность стейтментов. Каждая программа, написанная на языке C++, должна содержать главную **функцию main()**. Именно с первого стейтмента, находящегося в функции main(), и начинается выполнение всей программы. Функции, как правило, выполняют конкретное задание. Например, функция max() может содержать стейтменты, которые определяют большее из заданных чисел, а функция calculateGrade() может вычислять среднюю оценку студента по какой-либо дисциплине.

> ***Совет:*** Всегда размещайте функцию main() в файле .cpp с именем, совпадающим с именем проекта. Например, если вы пишете программу Chess, то поместите вашу функцию main() в файл chess.cpp.

Библиотеки

Библиотека — это набор скомпилированного кода (например, функций), который был "упакован" для повторного использования в других программах. С помощью библиотек можно расширить возможности программ. Например, если вы пишете игру, то вам придется подключать библиотеки звука или графики (если вы самостоятельно не хотите их создавать).

Язык C++ не такой уж и большой, как вы могли бы подумать. Тем не менее, он идет в комплекте со **Стандартной библиотекой C++**, которая предоставляет дополнительный функционал. Одной из наиболее часто используемых частей Стандартной библиотеки C++ является **библиотека iostream**, которая позволяет выводить данные на экран и обрабатывать пользовательский ввод.

Пример простой программы

Теперь, когда у вас есть общее представление о том, что такое стейтменты, функции и библиотеки, давайте рассмотрим еще раз программу "Hello, world!":

```
1. #include <iostream>
2.
3. int main()
4. {
5.    std::cout << "Hello, world!";
6.    return 0;
7. }
```

Строка №1: Специальный тип инструкции, который называется **директивой препроцессора**. Директивы препроцессора сообщают компилятору, что ему нужно выполнить определенное задание. В этом случае мы говорим компилятору, что хотели бы подключить содержимое заголовочного файла <iostream> к нашей программе. Подключение заголовочного файла <iostream> дает нам возможность использовать функционал библиотеки iostream, что, в свою очередь, позволяет выводить нам данные на экран.

Строка №2: Пустое пространство, которое игнорируется компилятором.

Строка №3: Объявление главной функции main().

Строки №4 и №7: Указываем компилятору область функции main(). Всё, что находится между открывающей фигурной скобкой в строке №4 и закрывающей фигурной скобкой в строке №7 — считается содержимым функции main().

Строка №5: Наш первый стейтмент (заканчивается точкой с запятой) — стейтмент вывода. `std::cout` — это специальный объект, используя который мы можем выводить данные на экран. `<<` — это оператор вывода. Всё, что мы отправляем в `std::cout`, — выводится на экран. В этом случае, мы выводим текст `"Hello, world!"`.

Строка №6: **Оператор возврата return**. Когда программа завершает свое выполнение, функция main() передает обратно в операционную систему значение, которое указывает на результат выполнения программы: успешно ли прошло выполнение программы или нет.

Если оператор return возвращает число `0`, то это значит, что всё хорошо! Ненулевые возвращаемые значения указывают на то, что что-то пошло не так и выполнение программы было прервано. Об операторе return мы еще поговорим детально на соответствующем уроке.

Синтаксис и синтаксические ошибки

Как вы, должно быть, знаете, в русском языке все предложения подчиняются правилам грамматики. Например, каждое предложение должно заканчиваться точкой. Правила, которые регулируют построение предложений, называются **синтаксисом**. Если вы не поставили точку и записали два предложения подряд, то это является нарушением синтаксиса русского языка.

Язык C++ также имеет свой синтаксис: правила написания кода/программ. При компиляции вашей программы, компилятор отвечает за то, чтобы ваша программа соответствовала правилам синтаксиса языка C++. Если вы нарушили правила, то компилятор будет ругаться и выдаст вам ошибку.

Например, давайте посмотрим, что произойдет, если мы не укажем в конце стейтмента точку с запятой:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Hello world!"
6.     return 0;
7. }
```

Результат:

```
E0065: требуется точка с запятой ";"
C2143: синтаксическая ошибка: отсутствие ";" перед "}"
```

Допущена синтаксическая ошибка в строке №6: мы забыли указать точку с запятой перед оператором return. В этом случае ошибка на самом деле в конце строки №5. В большинстве случаев компилятор правильно определяет строку с ошибкой, но есть ситуации, когда ошибка не заметна вплоть до начала следующей строки.

Синтаксические ошибки нередко совершаются при написании программ. К счастью, большинство из них можно легко найти и исправить. Но следует помнить, что программа может быть полностью скомпилирована и выполнена только при отсутствии ошибок.

Тест

Теперь давайте проверим то, как вы усвоили материал текущего урока. Ответьте на следующие вопросы:

- В чём разница между стейтментом и выражением?

- В чём разница между функцией и библиотекой?
- Чем заканчиваются стейтменты в языке C++?
- Что такое синтаксическая ошибка?

Урок №12. Комментарии

Комментарий — это строка (или несколько строк) текста, которая вставляется в исходный код для объяснения того, что делает код. В языке C++ есть 2 типа комментариев: однострочные и многострочные.

Однострочные комментарии

Однострочные комментарии — это комментарии, которые пишутся после символов `//`. Они пишутся в отдельных строках и всё, что находится после этих символов комментирования, — игнорируется компилятором, например:

```
1. std::cout << «Hello, world!» << std::endl; // всё, что находится справа от
   двойного слеша, - игнорируется компилятором
```

Как правило, однострочные комментарии используются для объяснения одной строчки кода:

```
1. std::cout << «Hello, world!» << std::endl; // cout и endl находятся в
   библиотеке iostream
2. std::cout << «It is so exciting!» << std::endl; // эти комментарии усложняют
   чтение кода
3. std::cout << «Yeah!» << std::endl; // особенно, когда строки разной длины
```

Размещая комментарии справа от кода, мы затрудняем себе как чтение кода, так и чтение комментариев. Следовательно, однострочные комментарии лучше размещать над строками кода:

```
1. // cout и endl находятся в библиотеке iostream
2. std::cout << «Hello, world!» << std::endl;
3.
4. // теперь уже легче читать
5. std::cout << «It is so exciting!» << std::endl;
6.
7. // не так ли?
8. std::cout << «Yeah!» << std::endl;
```

Многострочные комментарии

Многострочные комментарии — это комментарии, которые пишутся между символами `/* */`. Всё, что находится между звёздочками, — игнорируется компилятором:

```
1. /* Это многострочный комментарий.
2. Эта строка игнорируется
3. и эта тоже. */
```

Так как всё, что находится между звёздочками, — игнорируется, то иногда вы можете наблюдать следующее:

```
/* Это многострочный комментарий.
* Звёздочки слева
* упрощают чтение текста
*/
```

Многострочные комментарии не могут быть вложенными (т.е. одни комментарии внутри других):

```
/* Это многострочный /* комментарий */ а это уже не комментарий */
// Верхний комментарий заканчивается перед первым */, а не перед вторым */
```

Правило: Никогда не используйте вложенные комментарии.

Как правильно писать комментарии?

Во-первых, на уровне библиотек/программ/функций комментарии отвечают на вопрос **"ЧТО?"**: "Что делают эти библиотеки/программы/функции?". Например:

```
// Эта программа вычисляет оценку студента за семестр на основе его оценок за модули

// Эта функция использует метод Ньютона для вычисления корня функции

// Следующий код генерирует случайное число
```

Все эти комментарии позволяют понять, что делает программа, без необходимости смотреть на исходный код. Это особенно важно специалистам, работающим в команде, где не каждый специалист будет знаком со всем имеющимся кодом.

Во-вторых, внутри библиотек/программ/функций комментарии отвечают на вопрос **"КАК?"**: "Как код выполняет задание?". Например:

```
/* Для расчета итоговой оценки ученика, мы складываем все его оценки за уроки и
 домашние задания, а затем делим получившееся число на общее количество оценок.
   Таким образом, мы получаем средний балл ученика. */
```

Или:

```
// Чтобы получить рандомный (случайный) элемент, мы выполняем следующее:

// 1) Составляем список всех элементов.
// 2) Вычисляем среднее значение для каждого элемента, исходя из его веса, цвета и цены.
// 3) Выбираем любое число.
// 4) Определяем соответствие элемента случайно выбранному числу.
// 5) Возвращаем случайный элемент.
```

Эти комментарии позволяют читателю понять, каким образом код выполняет поставленное ему задание.

В-третьих, на уровне стейтментов (однострочного кода) комментарии отвечают на вопрос **"ПОЧЕМУ?"**: "Почему код выполняет задание именно так, а не иначе?". Плохой комментарий на уровне стейтментов объясняет, что делает код. Если вы когда-нибудь писали код, который был настолько сложным, что нужен был комментарий, который бы объяснял, что он делает, то вам нужно было бы не писать комментарий, а переписывать этот код.

Примеры плохих и хороших однострочных комментариев:

- Плохой комментарий:

```
1. // Присваиваем переменной sight значение 0
2. sight = 0;
```

(По коду это и так понятно)

✓ Хороший комментарий:

```
1. // Игрок выпил зелье слепоты и ничего не видит
2. sight = 0;
```

(Теперь мы знаем, ПОЧЕМУ зрение у игрока равно нулю)

- Плохой комментарий:

```
1. // Рассчитываем стоимость элементов
2. cost = items / 2 * storePrice;
```

(Да, мы видим, что здесь подсчет стоимости, но почему элементы делятся на 2?)

✓ Хороший комментарий:

```
1. // Нам нужно разделить все элементы на 2, потому что они куплены по парам
2. cost = items / 2 * storePrice;
```

(Теперь понятно!)

Программистам часто приходится принимать трудные решения по поводу того, каким способом решить проблему. А комментарии и существуют для того, чтобы напомнить себе (или объяснить другим) причину, почему вы написали код именно так, а не иначе.

✓ Хорошие комментарии:

```
// Мы решили использовать список вместо массива,
// потому что массивы осуществляют медленную вставку.
```

Или:

```
// Мы используем метод Ньютона для вычисления корня функции,
// так как другого детерминистического способа решения этой задачи - нет.
```

И, наконец, комментарии нужно писать так, чтобы человек, который не имеет ни малейшего представления о том, что делает ваш код — смог в нем разобраться. Очень часто случаются ситуации, когда программист говорит: «Это же совершенно очевидно, что делает код! Я это точно не забуду!". Угадайте, что случится через несколько недель или даже дней? Это не совершенно очевидно, и вы удивитесь, как скоро вы забудете то, что делает ваш код. Вы (или кто-то другой) будете очень благодарны себе за то, что оставите комментарии, объясняя на человеческом языке что, как и почему делает ваш код. Читать отдельные строки кода — легко, понимать их логику и смысл — сложно.

Подытожим:

- На уровне библиотек/программ/функций оставляйте комментарии, отвечая на вопрос "*ЧТО?*".
- Внутри библиотек/программ/функций оставляйте комментарии, отвечая на вопрос "*КАК?*".
- На уровне стейтментов оставляйте комментарии, отвечая на вопрос "*ПОЧЕМУ?*".

Закомментировать код

Закомментировать код — это конвертировать одну или несколько строк кода в комментарии. Таким образом, вы можете (временно) исключить часть кода из компиляции.

Чтобы закомментировать одну строку кода, используйте однострочные символы комментирования `//`.

Не закомментировано:

```
std::cout << 1;
```

Закомментировано:

```
1. //    std::cout << 1;
```

Чтобы закомментировать блок кода, используйте однострочные символы комментирования `//` на каждой строке или символы многострочного комментария `/* */`.

Не закомментировано:

```
1. std::cout << 1;
2. std::cout << 2;
3. std::cout << 3;
```

Закомментировано символами однострочного комментария:

```
1. //    std::cout << 1;
2. //    std::cout << 2;
3. //    std::cout << 3;
```

Закомментировано символами многострочного комментария:

```
1. /*
2.      std::cout << 1;
3.      std::cout << 2;
4.      std::cout << 3;
5. */
```

Есть несколько причин, почему следует использовать "закомментирование":

- ***Причина №1**:* Вы работаете над новой частью кода, которая пока что не рабочая, но вам нужно запустить программу. Компилятор не позволит выполнить программу, если в ней будут ошибки. Временное отделение нерабочего кода от рабочего комментированием позволит вам запустить программу. Когда код будет рабочий, то вы сможете его легко раскомментировать и продолжить работу.
- ***Причина №2**:* Вы написали код, который компилируется, но работает не так, как нужно и сейчас у вас нет времени с этим возиться. Закомментируйте код, а затем, когда будет время, исправьте ошибки.
- ***Причина №3**:* Поиск корня ошибки. Если вас не устраивают результаты работы программы (или вообще происходит сбой), полезно будет поочерёдно "отключать" части вашего кода, чтобы понять какие из них рабочие, а какие — создают проблемы. Если вы закомментируете одну или несколько строчек кода и программа начнет корректно работать (или пропадут сбои), шансы того, что последнее, что вы закомментировали,

является ошибкой — очень велики. После этого вы сможете разобраться с тем, почему же этот код не работает так, как нужно.

- ***Причина №4**:* Тестирование нового кода. Вместо удаления старого кода, вы можете его закомментировать и оставить для справки, пока не будете уверены в том, что ваш новый код работает так, как нужно. Как только вы будете уверены в новом коде, то сможете без проблем удалить старые фрагменты кода. Если же новый код у вас будет работать не так, как нужно, то вы сможете его удалить и откатиться к старому коду.

> ***Примечание:*** Во всех следующих уроках я буду использовать комментарии в иллюстративных целях. Внимательные читатели смогут заметить, что по вышеуказанным стандартам большинство из этих комментариев будут плохими. Но помните, что использовать я их буду в образовательных целях, а не для демонстрации хороших примеров.

Урок №13. Переменные, Инициализация и Присваивание

Программируя на языке C++, мы создаем, обрабатываем и уничтожаем объекты. **Объект** — это часть памяти, которая может хранить значение. В качестве аналогии мы можем использовать почтовый ящик, куда мы помещаем информацию и откуда её извлекаем. Все компьютеры имеют **оперативную память**, которую используют программы. При создании объекта, часть оперативной памяти выделяется для этого объекта. Большинство объектов, с которыми мы будем работать в языке C++, являются переменными.

Переменные

Стейтмент `a = 8;` выглядит довольно простым: мы присваиваем значение `8` переменной `a`. Но что такое `a`? `a` — это переменная, объект с именем.

На этом уроке мы рассмотрим только целочисленные переменные. **Целое число** — это число, которое можно записать без дроби, например: -11, -2, 0, 5 или 34.

Для создания переменной используется стейтмент объявления (разницу между объявлением и определением переменной мы рассмотрим несколько позже). Вот пример объявления целочисленной переменной `a` (которая может содержать только целые числа):

```
1. int a;
```

При выполнении этой инструкции центральным процессором часть оперативной памяти выделяется под этот объект. Например, предположим, что переменной `a` присваивается ячейка памяти под номером 150. Когда программа видит переменную `a` в выражении или в стейтменте, то она понимает, что для того, чтобы получить значение этой переменной, нужно заглянуть в ячейку памяти под номером 150.

Одной из наиболее распространенных операций с переменными является операция присваивания, например:

```
1. a = 8;
```

Когда процессор выполняет эту инструкцию, он понимает её как "поместить значение 8 в ячейку памяти под номером 150".

Затем мы сможем вывести это значение на экран с помощью std::cout:

```
1. std::cout << a; // выводим значение переменной a (ячейка памяти под номером
   150) на экран
```

l-values и r-values

В языке C++ все переменные являются l-values. **l-value** (в переводе *"л-значение"*, произносится как *"ел-валью"*) — это значение, которое имеет свой собственный адрес в памяти. Поскольку все переменные имеют адреса, то они все являются l-values (например, переменные `a`, `b`, `c` — все они являются l-values). l от слова "left", так как только значения l-values могут находиться в левой стороне в операциях присваивания (в противном случае, мы получим ошибку). Например, стейтмент `9 = 10;` вызовет ошибку компилятора, так как 9 не является l-value. Число 9 не имеет своего адреса в памяти и, таким образом, мы ничего не можем ему присвоить (9 = 9 и ничего здесь не изменить).

Противоположностью l-value является r-value (в переводе *"р-значение"*, произносится как *«ер-валью»*). **r-value** — это значение, которое не имеет постоянного адреса в памяти. Примерами могут быть единичные числа (например, `7`, которое имеет значение `7`) или выражения (например, `3 + x`, которое имеет значение `x плюс 3`).

Вот несколько примеров операций присваивания с использованием r-values:

```
1. int a;      // объявляем целочисленную переменную a
2. a = 5;      // 5 имеет значение 5, которое затем присваивается переменной a
3. a = 4 + 6;  // 4 + 6 имеет значение 10, которое затем присваивается
   переменной a
4.
5. int b;      // объявляем целочисленную переменную b
6. b = a;      // a имеет значение 10 (исходя из предыдущих операций), которое
   затем присваивается переменной b
7. b = b;      // b имеет значение 10, которое затем присваивается переменной b
   (ничего не происходит)
8. b = b + 2;  // b + 2 имеет значение 12, которое затем присваивается
   переменной b
```

Давайте детально рассмотрим последнюю операцию присваивания:

```
1. b = b + 2;
```

Здесь переменная `b` используется в двух различных контекстах. Слева `b` используется как l-value (переменная с адресом в памяти), а справа `b` используется как r-value и имеет отдельное значение (в данном случае, `12`).

При выполнении этого стейтмента, компилятор видит следующее:

```
1. b = 10 + 2;
```

И здесь уже понятно, какое значение присваивается переменной `b`.

Сильно беспокоиться о l-values или r-values сейчас не нужно, так как мы еще вернемся к этой теме на следующих уроках. Всё, что вам нужно сейчас запомнить — это то, что в левой стороне операции присваивания всегда должно находиться l-value (которое имеет свой собственный адрес в памяти), а в правой стороне операции присваивания — r-value (которое имеет какое-то значение).

Инициализация vs. Присваивание

В языке C++ есть две похожие концепции, которые новички часто путают: присваивание и инициализация.

После объявления переменной, ей можно **присвоить** значение с помощью оператора присваивания (знак равенства `=`):

```
1. int a; // это объявление переменной
2. a = 8; // а это присваивание переменной a значения 8
```

В языке C++ вы можете объявить переменную и присвоить ей значение одновременно. Это называется **инициализацией** (или ***"определением"***).

```
1. int a = 8; // инициализируем переменную a значением 8
```

Переменная может быть инициализирована только после операции объявления.

Хотя эти два понятия близки по своей сути и часто могут использоваться для достижения одних и тех же целей, все же в некоторых случаях следует использовать инициализацию, вместо присваивания, а в некоторых — присваивание вместо инициализации.

Правило: Если у вас изначально имеется значение для переменной, то используйте инициализацию, вместо присваивания.

Неинициализированные переменные

В отличие от других языков программирования, языки Си и C++ не инициализируют переменные определенными значениями (например, нулем) по умолчанию. Поэтому, при создании переменной, ей присваивается ячейка в памяти, в которой

уже может находиться какой-нибудь мусор! Переменная без значения (со стороны программиста или пользователя) называется **неинициализированной переменной**.

Использование неинициализированных переменных может привести к ошибкам, например:

```
#include <iostream>

int main()
{
    // Объявляем целочисленную переменную a
    int a;

    // Выводим значение переменной a на экран (a - это неинициализированная переменная)
    std::cout << a;

    return 0;
}
```

В этом случае компилятор присваивает переменной `a` ячейку в памяти, которая в данный момент свободна (не используется). Затем значение переменной `a` отправляется на вывод. Но что мы увидим на экране? Ничего, так как компилятор это не пропустит — выведется ошибка, что переменная `a` является неинициализированной. В более старых версиях Visual Studio компилятор вообще мог бы вывести какое-то некорректное значение (например, `7177728`, т.е. мусор), которое было бы содержимым той ячейки памяти, которую он присвоил нашей переменной.

Использование неинициализированных переменных является одной из самых распространенных ошибок начинающих программистов, но, к счастью, большинство современных компиляторов выдадут ошибку во время компиляции, если обнаружат неинициализированную переменную.

Хорошей практикой считается всегда инициализировать свои переменные. Это будет гарантией того, что ваша переменная всегда имеет определенное значение и вы не получите ошибку от компилятора.

Правило: Убедитесь, что все ваши переменные в программе имеют значения (либо через инициализацию, либо через операцию присваивания).

Тест

Какой результат выполнения следующих стейтментов?

```
int a = 6;
a = a - 3;
```

```
3.  std::cout << a << std::endl; // №1
4.
5.  int b = a;
6.  std::cout << b << std::endl; // №2
7.
8.  // В этом случае a + b является r-value
9.  std::cout << a + b << std::endl; // №3
10.
11. std::cout << a << std::endl; // №4
12.
13. int c;
14. std::cout << c << std::endl; // №5
```

Урок №14. cout, cin и endl

Объект std::cout

Как мы уже говорили на предыдущих уроках, **объект std::cout** (который находится в **библиотеке iostream**) используется для вывода данных на экран (в консольное окно). В качестве напоминания, вот наша программа «Hello, world!»:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Hello, world!";
6.     return 0;
7. }
```

Для вывода нескольких предложений на одной строке **оператор вывода** `<<` нужно использовать несколько раз, например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.    int a = 7;
6.    std::cout << "a is " << a;
7.    return 0;
8. }
```

Программа выведет:

```
a is 7
```

А какой результат выполнения следующей программы?

```
1. #include <iostream>
2.
3. int main()
4. {
5.    std::cout << "Hi!";
6.    std::cout << "My name is Anton.";
7.    return 0;
8. }
```

Возможно, вы удивитесь, но:

```
Hi!My name is Anton.
```

Объект std::endl

Если текст нужно вывести раздельно (на нескольких строках) — используйте std::endl. При использовании с std::cout, **std::endl** вставляет символ новой строки. Таким образом, мы перемещаемся к началу следующей строки, например:

```
#include <iostream>

int main()
{
    std::cout << "Hi!" << std::endl;
    std::cout << "My name is Anton." << std::endl;
    return 0;
}
```

Результат:

```
Hi!
My name is Anton.
```

Объект std::cin

std::cin является противоположностью std::cout. В то время как std::cout выводит данные в консоль с помощью оператора вывода `<<`, **std::cin** получает данные от пользователя с помощью **оператора ввода** `>>`. Используя std::cin мы можем получать и обрабатывать пользовательский ввод:

```
#include <iostream>

int main()
{
    std::cout << "Enter a number: "; // просим пользователя ввести любое число
    int a = 0;
    std::cin >> a; // получаем пользовательское число и сохраняем его в переменную a
    std::cout << "You entered " << a << std::endl;
    return 0;
}
```

Попробуйте скомпилировать и запустить эту программу. При запуске вы увидите `Enter a number:`, а затем программа будет ждать, пока вы укажите число. Как только вы это сделаете и нажмете Enter, программа выведет `You entered`, а затем ваше число.

Например (я ввел `7`):

```
Enter a number: 7
You entered 7
```

Это самый простой способ получения данных от пользователя. Мы будем его использовать в дальнейших примерах.

Если ваше окно закрывается сразу после ввода числа — смотрите Урок №10 (там есть решение данной проблемы).

Если же ввести действительно большое число, то вы получите переполнение, так как переменная `a` может содержать числа только определенного размера/диапазона. Если число больше/меньше допустимых максимумов/минимумов, то происходит переполнение. Об этом мы детально поговорим на следующих уроках.

std::cin, std::cout, << и >>

Новички часто путают `std::cin` с `std::cout` и `<<` с `>>`. Вот простые способы запомнить их различия:

- `std::cin` и `std::cout` всегда находятся в левой стороне стейтмента;
- `std::cout` используется для вывода значения (с**OUT** = вывод);
- `std::cin` используется для получения значения (с**IN** = ввод);
- оператор вывода `<<` используется с `std::cout` и показывает направление, в котором данные движутся от r-value в консоль. `std::cout << 7;` (значение `7` перемещается в консоль);
- оператор ввода `>>` используется с `std::cin` и показывает направление, в котором данные движутся из консоли в переменную. `std::cin >> a;` (значение из консоли перемещается в переменную `a`).

Урок №15. Функции и оператор возврата return

Вы уже знаете, что каждая программа должна содержать функцию main() (с которой она и начинает свое выполнение). Тем не менее, большинство программ используют и много других функций.

Функции

Функция — это последовательность стейтментов для выполнения определенного задания. Часто ваши программы будут прерывать выполнение одних функций ради выполнения других. Вы делаете аналогичные вещи в реальной жизни постоянно. Например, вы читаете книгу и вспомнили, что должны были сделать телефонный звонок. Вы оставляете закладку в своей книге, берете телефон и набираете номер. После того, как вы уже поговорили, вы возвращаетесь к чтению: к той странице, на которой остановились.

Программы на языке C++ работают похожим образом. Иногда, когда программа выполняет код, она может столкнуться с вызовом функции. **Вызов функции** — это выражение, которое указывает процессору прервать выполнение текущей функции и приступить к выполнению другой функции. Процессор "оставляет закладку" в текущей точке выполнения, а затем выполняет вызываемую функцию. Когда выполнение вызываемой функции завершено, процессор возвращается к *закладке* и возобновляет выполнение прерванной функции.

Функция, в которой находится вызов, называется **caller**, а функция, которую вызывают — **вызываемая функция**, например:

```
1. #include <iostream> // для std::cout и std::endl
2.
3. // Объявление функции doPrint(), которую мы будем вызывать
4. void doPrint() {
5.    std::cout << "In doPrint()" << std::endl;
6. }
7.
8. // Объявление функции main()
9. int main()
10.{
11.   std::cout << "Starting main()" << std::endl;
12.   doPrint(); // прерываем выполнение функции main() вызовом функции doPrint().
    Функция main() в данном случае является caller-ом
13.   std::cout << "Ending main()" << std::endl;
14.   return 0;
15.}
```

Результат выполнения программы:

```
Starting main()
In doPrint()
Ending main()
```

Эта программа начинает выполнение с первой строки функции main(), где выводится на экран следующая строка: `Starting main()`. Вторая строка функции main() вызывает функцию doPrint(). На этом этапе выполнение стейтментов в функции main() приостанавливается и процессор переходит к выполнению стейтментов внутри функции doPrint(). Первая (и единственная) строка в doPrint() выводит текст `In doPrint()`. Когда процессор завершает выполнение doPrint(), он возвращается обратно в main() к той точке, на которой остановился. Следовательно, следующим стейтментом является вывод строки `Ending main()`.

Обратите внимание, для вызова функции нужно указать её имя и список параметров в круглых скобках `()`. В примере, приведенном выше, параметры не используются, поэтому круглые скобки пусты. Мы детально поговорим о параметрах функций на следующем уроке.

Правило: Не забывайте указывать круглые скобки () при вызове функций.

Возвращаемые значения

Когда функция main() завершает свое выполнение, она возвращает целочисленное значение обратно в операционную систему, используя **оператор return**.

Функции, которые мы пишем, также могут возвращать значения. Для этого нужно указать **тип возвращаемого значения** (или ***"тип возврата"***). Он указывается при объявлении функции, перед её именем. Обратите внимание, тип возврата не указывает, какое именно значение будет возвращаться. Он указывает только тип этого значения.

Затем, внутри вызываемой функции, мы используем оператор return, чтобы указать **возвращаемое значение** — какое именно значение будет возвращаться обратно в caller.

Рассмотрим простую функцию, которая возвращает целочисленное значение:

```
1. #include <iostream>
2.
3. // int означает, что функция возвращает целочисленное значение обратно в caller
4. int return7()
5. {
```

```
    // Эта функция возвращает целочисленное значение, поэтому мы должны использовать оператор return
    return 7; // возвращаем число 7 обратно в caller
}

int main()
{
    std::cout << return7() << std::endl; // выведется 7
    std::cout << return7() + 3 << std::endl; // выведется 10

    return7(); // возвращаемое значение 7 игнорируется, так как функция main() ничего с ним не делает

    return 0;
}
```

Результат выполнения программы:

```
7
10
```

Разберемся детально:

- Первый вызов функции return7() возвращает `7` обратно в caller, которое затем передается в std::cout для вывода.
- Второй вызов функции return7() опять возвращает `7` обратно в caller. Выражение `7 + 3` имеет результат `10`, который затем выводится на экран.
- Третий вызов функции return7() опять возвращает `7` обратно в caller. Однако функция main() ничего с ним не делает, поэтому ничего и не происходит (возвращаемое значение игнорируется).

> ***Примечание:*** Возвращаемые значения не выводятся на экран, если их не передать объекту std::cout. В последнем вызове функции return7() значение не отправляется в std::cout, поэтому ничего и не происходит.

Тип возврата void

Функции могут и не возвращать значения. Чтобы сообщить компилятору, что функция не возвращает значение, нужно использовать **тип возврата void**. Взглянем еще раз на функцию doPrint() из вышеприведенного примера:

```
void doPrint() // void - это тип возврата
{
    std::cout << "In doPrint()" << std::endl;
    // Эта функция не возвращает никакого значения, поэтому оператор return здесь не нужен
}
```

Эта функция имеет тип возврата void, который означает, что функция не возвращает значения. Поскольку значение не возвращается, то и оператор return не требуется.

Вот еще один пример использования функции типа void:

```
#include <iostream>

// void означает, что функция не возвращает значения
void returnNothing()
{
    std::cout << "Hi!" << std::endl;
    // Эта функция не возвращает никакого значения, поэтому оператор return здесь не нужен
}

int main()
{
    returnNothing(); // функция returnNothing() вызывается, но обратно в main() ничего не возвращает

    std::cout << returnNothing(); // ошибка, эта строчка не скомпилируется. Вам нужно будет её закомментировать
    return 0;
}
```

В первом вызове функции returnNothing() выводится `Hi!`, но ничего не возвращается обратно в caller. Точка выполнения возвращается обратно в функцию main(), где программа продолжает свое выполнение.

Второй вызов функции returnNothing() даже не скомпилируется. Функция returnNothing() имеет тип возврата void, который означает, что эта функция не возвращает значения. Однако функция main() пытается отправить это значение (которое не возвращается) в std::cout для вывода. std::cout не может обработать этот случай, так как значения на вывод не предоставлено. Следовательно, компилятор выдаст ошибку. Вам нужно будет закомментировать эту строку, чтобы компиляция прошла успешно.

Возврат значений обратно в функцию main()

Теперь у вас есть понимание того, как работает функция main(). Когда программа выполняется, операционная система делает вызов функции main() и начинается её выполнение. Стейтменты в main() выполняются последовательно. В конце функция main() возвращает целочисленное значение (обычно, `0`) обратно в операционную систему. Поэтому main() объявляется как `int main()`.

Почему нужно возвращать значения обратно в операционную систему? Дело в том, что возвращаемое значение функции main() является **кодом состояния**, который сообщает операционной системе об успешном или неудачном выполнении

программы. Обычно, возвращаемое значение `0` (ноль) означает что всё прошло успешно, тогда как любое другое значение означает неудачу/ошибку.

Обратите внимание, по стандартам языка C++ функция main() должна возвращать целочисленное значение. Однако, если вы не укажете return в конце функции main(), компилятор возвратит `0` автоматически, если никаких ошибок не будет. Но рекомендуется указывать return в конце функции main() и использовать тип возврата int для функции main().

Еще о возвращаемых значениях

Во-первых, если тип возврата функции не void, то она должна возвращать значение указанного типа (использовать оператор return). Единственно исключение — функция main(), которая возвращает `0`, если не предоставлено другое значение.

Во-вторых, когда процессор встречает в функции оператор return, он немедленно выполняет возврат значения обратно в caller и точка выполнения также переходит в caller. Любой код, который находится за return-ом в функции — игнорируется.

Функция может возвращать только одно значение через return обратно в caller. Это может быть либо число (например, `7`), либо значение переменной, либо выражение (у которого есть результат), либо определенное значение из набора возможных значений.

Но есть способы обойти правило возврата одного значения, возвращая сразу несколько значений, но об этом детально мы поговорим на соответствующем уроке.

Наконец, автор функции решает, что означает её возвращаемое значение. Некоторые функции используют возвращаемые значения в качестве кодов состояния для указания результата выполнения функции (успешно ли выполнение или нет). Другие функции возвращают определенное значение из набора возможных значений. Кроме того, существуют функции, которые вообще ничего не возвращают.

Повторное использование функций

Одну и ту же функцию можно вызывать несколько раз, даже в разных программах, что очень полезно:

```
1. #include <iostream>
2.
```

```
3. // Функция getValueFromUser() получает значение от пользователя, а затем
   возвращает его обратно в caller
4. int getValueFromUser()
5. {
6.     std::cout << "Enter an integer: ";
7.     int x;
8.     std::cin >> x;
9.     return x;
10.}
11.
12.int main()
13.{
14.    int a = getValueFromUser(); // первый вызов функции getValueFromUser()
15.    int b = getValueFromUser(); // второй вызов функции getValueFromUser()
16.
17.    std::cout << a << " + " << b << " = " << a + b << std::endl;
18.
19.    return 0;
20.}
```

Результат выполнения программы:

```
Enter an integer: 4
Enter an integer: 9
4 + 9 = 13
```

Здесь main() прерывается 2 раза. Обратите внимание, в обоих случаях, полученное пользовательское значение сохраняется в переменной `x`, а затем передается обратно в main() с помощью return, где присваивается переменной `a` или `b`!

Также main() не является единственной функцией, которая может вызывать другие функции. Любая функция может вызывать любую другую функцию!

```
1. #include <iostream>
2.
3. void printO()
4. {
5.     std::cout << "O" << std::endl;
6. }
7.
8. void printK()
9. {
10.    std::cout << "K" << std::endl;
11.}
12.
13.// Функция printOK() вызывает как printO(), так и printK()
14.void printOK()
15.{
16.    printO();
17.    printK();
18.}
19.
20.// Объявление main()
21.int main()
22.{
23.    std::cout << "Starting main()" << std::endl;
```

```
24.    printOK();
25.    std::cout << "Ending main()" << std::endl;
26.    return 0;
27. }
```

Результат выполнения программы:

```
Starting main()
O
K
Ending main()
```

Вложенные функции

В языке C++ одни функции не могут быть объявлены внутри других функций (т.е. быть вложенными). Следующий код вызовет ошибку компиляции:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.     int boo() // эта функция находится внутри функции main(), что запрещено
6.     {
7.        std::cout << "boo!";
8.        return 0;
9.     }
10.
11.    boo();
12.    return 0;
13. }
```

Правильно вот так:

```
1.  #include <iostream>
2.
3.  int boo() // теперь уже не в main()
4.  {
5.     std::cout << "boo!";
6.     return 0;
7.  }
8.
9.  int main()
10. {
11.    boo();
12.    return 0;
13. }
```

Тест

Какие из следующих программ не скомпилируются (и почему), а какие скомпилируются (и какой у них результат)?

Программа №1:

```
1. #include <iostream>
2.
3. int return5()
4. {
5.     return 5;
6. }
7.
8. int return8()
9. {
10.     return 8;
11. }
12.
13. int main()
14. {
15.     std::cout << return5() + return8() << std::endl;
16.
17.     return 0;
18. }
```

Программа №2:

```
1. #include <iostream>
2.
3. int return5()
4. {
5.     return 5;
6.
7.     int return8()
8.     {
9.         return 8;
10.    }
11. }
12.
13. int main()
14. {
15.     std::cout << return5() + return8() << std::endl;
16.
17.     return 0;
18. }
```

Программа №3:

```
1. #include <iostream>
2.
3. int return5()
4. {
5.     return 5;
6. }
7.
8. int return8()
9. {
10.     return 8;
11. }
12.
13. int main()
14. {
15.     return5();
```

```
    return8();

    return 0;
}
```

Программа №4:

```
#include <iostream>

void printO()
{
    std::cout << "O" << std::endl;
}

int main()
{
    std::cout << printO() << std::endl;

    return 0;
}
```

Программа №5:

```
#include <iostream>

int getNumbers()
{
    return 6;
    return 8;
}

int main()
{
    std::cout << getNumbers() << std::endl;
    std::cout << getNumbers() << std::endl;

    return 0;
}
```

Программа №6:

```
#include <iostream>

int return 6()
{
    return 6;
}

int main()
{
    std::cout << return 6() << std::endl;

    return 0;
}
```

Программа №7:

```
1. #include <iostream>
2. 
3. int return6()
4. {
5.     return 6;
6. }
7. 
8. int main()
9. {
10.     std::cout << return6 << std::endl;
11. 
12.     return 0;
13. }
```

Урок №16. Параметры и аргументы функций

Во многих случаях нам нужно будет передавать данные в вызываемую функцию, чтобы она могла с ними как-то взаимодействовать. Например, если мы хотим написать функцию умножения двух чисел, то нам нужно каким-то образом сообщить функции, какие это будут числа. В противном случае, как она узнает, что на что перемножать? Здесь нам на помощь приходят параметры и аргументы.

Параметр функции — это переменная, которая используется в функции, и значение которой предоставляет caller (вызывающий объект). Параметры указываются при объявлении функции в круглых скобках. Если их много, то они перечисляются через запятую, например:

```
1. // Эта функция не имеет параметров
2. void doPrint()
3. {
4.     std::cout << "In doPrint()" << std::endl;
5. }
6. 
7. // Эта функция имеет один параметр типа int: a
8. void printValue(int a)
9. {
10.     std::cout << a << std::endl;
11. }
12. 
13. // Эта функция имеет два параметра типа int: a и b
14. int add(int a, int b)
15. {
16.     return a + b;
17. }
```

Параметры каждой функции действительны только внутри этой функции. Поэтому, если printValue() и add() имеют параметр с именем `a`, то это не означает, что произойдет конфликт имен. Эти параметры считаются независимыми и никак не взаимодействуют друг с другом.

Аргумент функции — это значение, которое передается из caller-а в функцию и которое указывается в скобках при вызове функции в caller-е:

```
1. printValue(7); // 7 – это аргумент функции printValue()
2. add(4, 5); // 4 и 5 – это аргументы функции add()
```

Обратите внимание, аргументы тоже перечисляются через запятую. Количество аргументов должно совпадать с количеством параметров, иначе компилятор выдаст сообщение об ошибке.

Как работают параметры и аргументы функций?

При вызове функции, все её параметры создаются как локальные переменные, а значение каждого из аргументов копируется в соответствующий параметр (локальную переменную). Этот процесс называется **передачей по значению**. Например:

```
#include <iostream>

// Эта функция имеет два параметра типа int: a и b.
// Значения переменных a и b определяет caller
void printValues(int a, int b)
{
    std::cout << a << std::endl;
    std::cout << b << std::endl;
}

int main()
{
    printValues(8, 9); // здесь два аргумента: 8 и 9

    return 0;
}
```

При вызове функции printValues() аргументы `8` и `9` копируются в параметры `a` и `b`. Параметру `a` присваивается значение `8`, а параметру `b` — значение `9`.

Результат:

```
8
9
```

Как работают параметры и возвращаемые значения функций?

Используя параметры и возвращаемые значения, мы можем создавать функции, которые будут принимать и обрабатывать данные, а затем возвращать результат обратно в caller.

Например, простая функция, которая принимает два целых числа и возвращает их сумму:

```
#include <iostream>

// Функция add() принимает два целых числа в качестве параметров и возвращает
их сумму.
// Значения a и b определяет caller
int add(int a, int b)
{
    return a + b;
}

```

```
// Функция main() не имеет параметров
int main()
{
   std::cout << add(7, 8) << std::endl; // аргументы 7 и 8 передаются в функцию add()
   return 0;
}
```

При вызове функции add(), параметру `a` присваивается значение `7`, а параметру `b` — значение `8`. Затем функция add() вычисляет их сумму и возвращает результат обратно в main(). И тогда уже результат выводится на экран.

Результат выполнения программы:

```
15
```

Еще примеры

Рассмотрим еще несколько вызовов функций:

```
#include <iostream>

int add(int a, int b)
{
   return a + b;
}

int multiply(int c, int d)
{
   return c * d;
}

int main()
{
   std::cout << add(7, 8) << std::endl; // внутри функции add(): a = 7, b = 8, значит a + b = 15
   std::cout << multiply(4, 5) << std::endl; // внутри функции multiply(): c = 4, d = 5, значит c * d = 20

   // Мы можем передавать целые выражения в качестве аргументов
   std::cout << add(2 + 3, 4 * 5) << std::endl; // внутри функции add(): a = 5, b = 20, значит a + b = 25

   // Мы можем передавать переменные в качестве аргументов
   int x = 4;
   std::cout << add(x, x) << std::endl; // будет 4 + 4

   std::cout << add(1, multiply(2, 3)) << std::endl; // будет 1 + (2 * 3)
   std::cout << add(1, add(2, 3)) << std::endl; // будет 1 + (2 + 3)

   return 0;
}
```

Результат выполнения программы:

```
15
20
25
8
7
6
```

С первыми двумя вызовами всё понятно.

В третьем вызове параметрами являются выражения, которые сначала нужно обработать. `2 + 3 = 5` и результат `5` присваивается переменной `a`. `4 * 5 = 20` и результат `20` присваивается переменной `b`. Результатом выполнения функции `add(5, 20)` является значение `25`.

Следующая пара относительно лёгкая для понимания:

```
1. int x = 4;
2. std::cout << add(x, x) << std::endl; // будет 4 + 4
```

Здесь уже `a = x` и `b = x`. Поскольку `x = 4`, то `add(x, x) = add(4, 4)`. Результат — `8`.

Теперь рассмотрим вызов посложнее:

```
1. std::cout << add(1, multiply(2, 3)) << std::endl; // будет 1 + (2 * 3)
```

При выполнении этого стейтмента процессор должен определить значения параметров `a` и `b` функции add(). С параметром `a` всё понятно — мы передаем значение `1` (`a = 1`). А вот чтобы определить значение параметра `b`, нам необходимо выполнить операцию умножения: `multiply(2, 3)`, результат — `6`. Затем `add(1, 6)` возвращает число `7`, которое и выводится на экран.

Короче говоря:

```
add(1, multiply(2, 3)) => add(1, 6) => 7
```

Последний вызов может показаться немного сложным из-за того, что параметром функции add() является другой вызов add():

```
1. std::cout << add(1, add(2, 3)) << std::endl; // будет 1 + (2 + 3)
```

Но здесь всё аналогично вышеприведенному примеру. Перед тем, как процессор вычислит внешний вызов функции add(), он должен обработать внутренний вызов

функции `add(2, 3)`. `add(2, 3) = 5`. Затем процессор обрабатывает функцию `add(1, 5)`, результатом которой является значение `6`. Затем `6` передается в std::cout.

Короче говоря:

```
add(1, add(2, 3)) => add(1, 5) => 6
```

Тест

Задание №1: Что не так со следующим фрагментом кода?

```
1. void multiply(int a, int b)
2. {
3.     return a * b;
4. }
5.
6. int main()
7. {
8.     std::cout << multiply(7, 8) << std::endl;
9.     return 0;
10. }
```

Задание №2: Какие здесь есть две проблемы?

```
1. #include <iostream>
2.
3. int multiply(int a, int b)
4. {
5.     int product = a * b;
6. }
7.
8. int main()
9. {
10.     std::cout << multiply(5) << std::endl;
11.     return 0;
12. }
```

Задание №3: Какой результат выполнения следующей программы?

```
1. #include <iostream>
2.
3. int add(int a, int b, int c)
4. {
5.     return a + b + c;
6. }
7.
8. int multiply(int a, int b)
9. {
10.     return a * b;
11. }
12.
13. int main()
14. {
15.     std::cout << multiply(add(3, 4, 5), 5) << std::endl;
```

```
16.    return 0;
17. }
```

Задание №4: Напишите функцию doubleNumber(), которая принимает целое число в качестве параметра, удваивает его, а затем возвращает результат обратно в caller.

Задание №5: Напишите полноценную программу, которая принимает целое число от пользователя (используйте std::cin), удваивает его с помощью функции doubleNumber() из предыдущего задания, а затем выводит результат на экран.

Урок №17. Почему функции – полезны, и как их эффективно использовать?

Начинающие программисты часто спрашивают: «А можно ли обходиться без функций и весь код помещать непосредственно в функцию main()?". Если вашего кода всего 10-20 строк, то можно. Если же серьезно, то функции предназначены для упрощения кода, а не для его усложнения. Они имеют ряд преимуществ, которые делают их чрезвычайно полезными в нетривиальных программах.

- **Структура**. Как только программы увеличиваются в размере/сложности, сохранять весь код внутри main() становится трудно. Функция - это как мини-программа, которую мы можем записать отдельно от головной программы, не заморачиваясь при этом об остальных частях кода. Это позволяет разбивать сложные задачи на более мелкие и простые, что кардинально снижает общую сложность программы.
- **Повторное использование**. После объявления функции, её можно вызывать много раз. Это позволяет избежать дублирования кода и сводит к минимуму вероятность возникновения ошибок при копировании/вставке кода. Функции также могут использоваться и в других программах, уменьшая объем кода, который нужно писать с нуля каждый раз.
- **Тестирование**. Поскольку функции убирают лишний код, то и тестировать его становится проще. А так как функция - это самостоятельная единица, то нам достаточно протестировать её один раз, чтобы убедиться в её работоспособности, а затем мы можем её повторно использовать много раз без необходимости проводить тестирование (до тех пор, пока не внесем изменения в эту функцию).
- **Модернизация**. Когда нужно внести изменения в программу или расширить её функционал, то функции являются отличным вариантом. С их помощью можно внести изменения в одном месте, чтобы они работали везде.
- **Абстракция**. Для того, чтобы использовать функцию, нам нужно знать её имя, данные ввода, данные вывода и где эта функция находится. Нам не нужно знать, как она работает. Это очень полезно для написания кода, понятного другим (например, Стандартная библиотека C++ и всё, что в ней находится, созданы по этому принципу).

Каждый раз, при вызове std::cin или std::cout для ввода или вывода данных, мы используем функцию из Стандартной библиотеки C++, которая соответствует всем вышеперечисленным концепциям.

Эффективное использование функций

Одной из наиболее распространенных проблем, с которой сталкиваются новички, является понимание того, где, когда и как эффективно использовать функции. Вот **несколько основных рекомендаций при написании функций**:

- *Рекомендация №1:* Код, который появляется более одного раза в программе, лучше переписать в виде функции. Например, если мы получаем данные от пользователя несколько раз одним и тем же способом, то это отличный вариант для написания отдельной функции.
- *Рекомендация №2:* Код, который используется для сортировки чего-либо, лучше записать в виде отдельной функции. Например, если у нас есть список вещей, которые нужно отсортировать - пишем функцию сортировки, куда передаем несортированный список и откуда получаем отсортированный.
- *Рекомендация №3:* Функция должна выполнять одно (и только одно) задание.
- *Рекомендация №4:* Когда функция становится слишком большой, сложной или непонятной — её следует разбить на несколько подфункций. Это называется **рефакторингом кода**.

При изучении языка C++ вам предстоит написать много программ, которые будут включать следующие три подзадания:

- Получение данных от пользователя.
- Обработка данных.
- Вывод результата.

Для простых программ (менее, чем 30 строк кода) частично или все эти три подзадания можно записать в функции main(). Для более сложных программ (или просто для практики) каждое из этих трех подзаданий является хорошим вариантом, чтобы написать отдельные функции.

Новички часто комбинируют обработку ввода и вывод результата в одной функции. Тем не менее, это нарушает **правило "одного задания"**. Функция, которая обрабатывает значение, должна возвращать его в caller, а дальше уже пускай caller сам решает, что ему с ним делать.

Урок №18. Локальная область видимости

Как мы уже знаем из предыдущих уроков, при выполнении процессором стейтмента `int x;` создается переменная. Возникает вопрос: «Когда эта переменная уничтожается?".

Область видимости переменной определяет, кто может видеть и использовать переменную во время её существования. И параметры функции, и переменные, которые объявлены внутри функции, имеют **локальную область видимости**. Другими словами, эти параметры и переменные используются только внутри функции, в которой они объявлены. Локальные переменные создаются в точке объявления и уничтожаются, когда выходят из области видимости.

Рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int add(int a, int b) // здесь создаются переменные a и b
4. {
5.    // a и b можно видеть/использовать только внутри этой функции
6.    return a + b;
7. } // здесь a и b выходят из области видимости и уничтожаются
8.
9. int main()
10.{
11.   int x = 7; // здесь создается и инициализируется переменная x
12.   int y = 8; // здесь создается и инициализируется переменная y
13.   // x и y можно использовать только внутри функции main()
14.   std::cout << add(x, y) << std::endl; // вызов функции add() с a = x и b = y
15.   return 0;
16.} // здесь x и y выходят из области видимости и уничтожаются
```

Параметры `a` и `b` функции add() создаются при вызове этой функции, используются только внутри нее и уничтожаются по завершении выполнения этой функции.

Переменные `x` и `y` функции main() можно использовать только внутри main() и они также уничтожаются по завершении выполнения функции main().

Для лучшего понимания давайте детально разберем ход выполнения этой программы:

- выполнение начинается с функции main();
- создается переменная `x` в функции main() и ей присваивается значение `7`;
- создается переменная `y` в функции main() и ей присваивается значение `8`;
- вызывается функция add() с параметрами `7` и `8`;
- создается переменная `a` в функции add() и ей присваивается значение `7`;

- создается переменная `b` в функции add() и ей присваивается значение `8`;
- выполняется операция сложения чисел `7` и `8`, результатом является значение `15`;
- функция add() возвращает значение `15` обратно в caller (в функцию main());
- переменные `a` и `b` функции add() уничтожаются;
- функция main() выводит значение `15` на экран;
- функция main() возвращает `0` в операционную систему;
- переменные `x` и `y` функции main() уничтожаются.

Всё!

Обратите внимание, если бы функция add() вызывалась дважды, параметры `a` и `b` создавались и уничтожались бы также дважды. В программе с большим количеством функций, переменные создаются и уничтожаются часто.

Локальная область видимости предотвращает возникновение конфликтов имен

Из примера, приведенного выше, понятно, что переменные `x` и `y` отличаются от переменных `a` и `b`.

Теперь давайте рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int add(int a, int b) // здесь создаются переменные a и b функции add()
4. {
5.    return a + b;
6. } // здесь a и b функции add() выходят из области видимости и уничтожаются
7.
8. int main()
9. {
10.    int a = 7; // здесь создается переменная a функции main()
11.    int b = 8; // здесь создается переменная b функции main()
12.    std::cout << add(a, b) << std::endl; // значения переменных a и b функции
   main() копируются в переменные a и b функции add()
13.    return 0;
14. } // здесь a и b функции main() выходят из области видимости и уничтожаются
```

Здесь мы изменили имена переменных `x` и `y` функции main() на `a` и `b`. Программа по-прежнему работает корректно, несмотря на то, что функция add() также имеет переменные `a` и `b`. Почему это не вызывает конфликта имен? Дело в том, что `a` и `b`, принадлежащие функции main(), являются локальными переменными, функция add() не может их видеть, точно так же, как функция main() не может видеть переменные `a` и `b`, принадлежащие функции add(). Ни add(), ни main() не знают, что они имеют переменные с одинаковыми именами!

Это значительно снижает возможность возникновения конфликта имен. Любая функция не должна знать или заботиться о том, какие переменные находятся в другой функции. Это также предотвращает возникновение ситуаций, когда одни функции могут непреднамеренно (или намеренно) изменять значения переменных других функций.

Правило: Имена, которые используются внутри функции (включая параметры), доступны/видны только внутри этой же функции.

Тест

Каким будет результат выполнения следующей программы?

```
#include <iostream>

void doMath(int a)
{
    int b = 5;
    std::cout << "doMath: a = " << a << " and b = " << b << std::endl;
    a = 4;
    std::cout << "doMath: a = " << a << " and b = " << b << std::endl;
}

int main()
{
    int a = 6;
    int b = 7;
    std::cout << "main: a = " << a << " and b = " << b << std::endl;
    doMath(a);
    std::cout << "main: a = " << a << " and b = " << b << std::endl;
    return 0;
}
```

Урок №19. Ключевые слова и идентификаторы

Язык C++ имеет зарезервированный набор из 84 слов (включая версию C++17) для собственного использования. Эти слова называются **ключевыми словами**, каждое из которых имеет свое особое значение.

Вот **список всех ключевых слов в языке C++** (включая C++17):

alignas (C++11)	decltype (C++11)	namespace	struct
alignof (C++11)	default	new	switch
and	delete	noexcept (C++11)	template
and_eq	do	not	this
asm	double	not_eq	thread_local (C++11)
auto	dynamic_cast	nullptr (C++11)	throw
bitand	else	operator	true
bitor	enum	or	try
bool	explicit	or_eq	typedef
break	export	private	typeid
case	extern	protected	typename
catch	false	public	union
char	float	register	unsigned
char16_t (C++11)	for	reinterpret_cast	using
char32_t (C++11)	friend	return	virtual
class	goto	short	void
compl	if	signed	volatile
const	inline	sizeof	wchar_t
constexpr (C++11)	int	static	while
const_cast	long	static_assert (C++11)	xor
continue	mutable	static_cast	xor_eq

C++11 также добавил два специальных идентификатора: **override** и **final**. Они имеют особое значение при использовании в определенных контекстах, но не являются зарезервированными ключевыми словами.

Мы уже сталкивались с некоторыми ключевыми словами, такими как int, void и return. Вместе с набором операторов, ключевые слова определяют весь язык C++ (не включая команд препроцессора). Поскольку они имеют особые значения, то IDE всегда выделяют/подсвечивают их другим цветом.

После изучения материалов всех уроков по C++ на Ravesli, вы узнаете суть большинства ключевых слов языка C++, приведенных выше.

Идентификаторы

Идентификатор — это имя переменной, функции, класса или другого объекта в языке C++. Мы можем определять идентификаторы любыми словами/именами. Тем не менее, есть несколько общих правил, которые необходимо соблюдать:

- Идентификатор не может быть ключевым словом. Ключевые слова зарезервированы.
- Идентификатор может состоять только из букв (нижнего и верхнего регистра) латинского алфавита, цифр и символов подчёркивания. Это означает, что все другие символы и пробелы - запрещены.
- Идентификатор должен начинаться с буквы (нижнего или верхнего регистра). Он не может начинаться с цифры.
- Язык C++ различает нижний регистр от верхнего. `nvalue` отличается от `nValue` и отличается от `NVALUE`.

Теперь, когда вы знаете, как *можно* называть объекты, давайте поговорим о том, как их *нужно* называть.

Во-первых, в языке C++ имена переменных начинаются с буквы в нижнем регистре. Если имя переменной состоит из одного слова, то это слово должно быть записано в нижнем регистре:

```
1. int value; // корректно
2.
3. int Value; // некорректно (должно начинаться с буквы в нижнем регистре)
4. int VALUE; // некорректно (должно начинаться с буквы в нижнем регистре)
5. int VaLuE; // некорректно (должно начинаться с буквы в нижнем регистре)
```

Как правило, имена функций также начинаются с буквы в нижнем регистре (хотя есть некоторые разногласия по этому вопросу). Мы будем придерживаться этого стиля, поскольку даже функция main() (главная функция всех программ) начинается с буквы в нижнем регистре, как и все функции из Стандартной библиотеки C++.

Имена идентификаторов, которые начинаются с заглавной буквы, используются для структур, классов или перечислений (об этом позже).

Если имя переменной или функции состоит из нескольких слов, то здесь есть два варианта: разделить подчёркиванием или использовать **CamelCase** — принцип,

когда несколько слов пишутся слитно, без пробелов, и каждое новое слово пишется с заглавной буквы.

CamelCase (в переводе как «*ВерблюжийСтиль*») получил свое название из-за заглавных букв, которые напоминают верблюжьи горбы.

```
int my_variable_name; // корректно (разделяется символом подчёркивания)
void my_function_name(); // корректно (разделяется символом подчёркивания)

int myVariableName; // корректно (используется CamelCase)
void myFunctionName(); // корректно (используется CamelCase)

int my variable name; // некорректно (пробелы запрещены)
void my function name(); // некорректно (пробелы запрещены)

int MyVariableName; // работает, но не рекомендуется (следует начинать с буквы в нижнем регистре)
void MyFunctionName(); // работает, но не рекомендуется
```

Хотя даже Стандартная библиотека C++ использует символ подчёркивания для переменных и функций, мы же будем использовать CamelCase - для лучшей читабельности кода. Иногда вы будете видеть сочетание двух способов: подчёркивание для переменных и CamelCase для функций.

Стоит отметить, что, если вы работаете с чужим кодом, хорошей практикой будет придерживаться стиля, в котором написан этот код, даже если он не соответствует рекомендациям, приведенным выше.

Во-вторых, не начинайте ваши имена с символа подчёркивания, так как такие имена уже зарезервированы для ОС, библиотеки и/или используются компилятором.

В-третьих, (это, пожалуй, самое важное правило из всех) используйте в качестве идентификаторов только те имена, которые реально описывают то, чем является объект. Очень характерно для неопытных программистов сокращать имена переменных, чтобы сэкономить время при наборе кода или потому, что они думают, что всё и так понятно. В большинстве случаев не всё всегда является понятным и очевидным. В идеале переменные нужно называть так, чтобы человек, который первый раз увидел ваш код, понял как можно скорее, что этот код делает. Через 3 месяца, когда вы будете пересматривать свои программы, вы забудете, как они работают, и будете благодарны самому себе за то, что называли переменные по сути, а не как попало.

Чем сложнее код, тем проще и понятнее должны быть идентификаторы.

int ccount	Плохо	Никто не знает, что такое ccount
int customerCount	Хорошо	Теперь понятно, что мы считаем
int i	Плохо*	В большинстве нетривиальных случаев — плохо, в простых примерах — может быть (например, в циклах)
int index	50/50	Хорошо, если очевидно, индексом чего является переменная
int totalScore	Хорошо	Всё понятно
int _count	Плохо	Не начинайте имена переменных с символов подчёркивания
int count	50/50	Хорошо, если очевидно, что мы считаем
int data	Плохо	Какой тип данных?
int value1, value2	50/50	Может быть трудно понять разницу между переменными
int numberOfApples	Хорошо	Всё понятно
int monstersKilled	Хорошо	Всё понятно
int x, y	Плохо*	В большинстве нетривиальных случаев — плохо, в простых примерах — может быть (например, в математических функциях)

> ***Примечание:*** Можно использовать тривиальные имена для переменных, которые имеют тривиальное использование (например, для переменных в цикле, в простых математических функциях и т.д.).

В-четвертых, уточняющий комментарий всегда будет только плюсом. Например, мы объявили переменную с именем `numberOfChars`, которая должна хранить количество символов определенной части строки. Сколько символов в строке `Hello, world!`: 10, 11, 12 или 13? Это зависит от того, учитываем ли мы пробелы и знаки препинания или нет. Вместо названия переменной `numberOfCharsIncludingWhitespaceAndPunctuation` лучше оставить хороший комментарий, который прояснит ситуацию:

```
1. // Эта переменная подсчитывает количество символов части строки, включая
   пробелы и знаки препинания
2. int numberOfChars;
```

Тест

Какие из переменных неправильно названы и почему?

- `int result;`
- `int _oranges;`
- `int NUMBER;`
- `int the name of a variable;`
- `int TotalCustomers;`
- `int void;`
- `int countFruit;`
- `int 4orYou;`
- `int kilograms_of_pipe;`

Урок №20. Операторы

Как мы уже знаем из предыдущих уроков, выражение - это математический объект, который имеет определенное значение. Однако, термин "математический объект" несколько расплывчатый. Точнее будет так: **выражение** - это комбинация литералов, переменных, функций и операторов, которая генерирует (создает) определенное значение.

Литералы

Литерал - это фиксированное значение, которое записывается непосредственно в исходном коде (например, `7` или `3.14159`). Вот пример программы, которая использует литералы:

```
#include <iostream>

int main()
{
    int a = 3; // a – это переменная, 3 – это литерал
    std::cout << 5 + 2; // 5 + 2 – это выражение, 5 и 2 – это литералы
    std::cout << "Hello, world!"; // "Hello, world" – это тоже литерал
}
```

Литералы, переменные и функции еще известны как операнды. **Операнды** - это данные, с которыми работает выражение. Литералы имеют фиксированные значения, переменным можно присваивать значения, функции же генерируют определенные значения (в зависимости от типа возврата, исключением являются функции типа void).

Операторы

Последним пазлом в выражениях являются операторы. С их помощью мы можем объединить операнды для получения нового значения. Например, в выражении `5 + 2`, `+` является оператором. С помощью `+` мы объединили операнды `5` и `2` для получения нового значения (`7`).

Вы, вероятно, уже хорошо знакомы со стандартными арифметическими операторами из школьной математики: сложение (`+`), вычитание (`-`), умножение (`*`) и деление (`/`). Знак равенства `=` является оператором присваивания. Некоторые операторы состоят более чем из одного символа, например, оператор равенства `==`, который позволяет сравнивать между собой два определенных значения.

> ***Примечание:*** Очень часто новички путают оператор присваивания (=) с оператором равенства (==). С помощью оператора присваивания (=) мы присваиваем переменной определенное значение. С помощью оператора равенства (==) мы проверяем, равны ли между собой два определенных операнда.

Операторы бывают 3-х типов:

- **Унарные**. Работают с одним операндом. Например, оператор `-` (минус). В выражении `-7`, оператор `-` применяется только к одному операнду (`7`), чтобы создать новое значение (`-7`).
- **Бинарные**. Работают с двумя операндами (левым и правым). Например, оператор `+`. В выражении `5 + 2`, оператор `+` работает с левым операндом (`5`) и правым (`2`), чтобы создать новое значение (`7`).
- **Тернарные**. Работают с тремя операндами (в языке C++ есть только один тернарный оператор).

Обратите внимание, некоторые операторы могут иметь несколько значений. Например, оператор `-` (минус) может использоваться в двух контекстах: как унарный для изменения знака числа (например, конвертировать `7` в `-7` и наоборот), и как бинарный для выполнения арифметической операции вычитания (например, `4 - 3`).

Заключение

Это только верхушка айсберга. Более детально об операторах мы обязательно поговорим на следующих уроках.

Урок №21. Базовое форматирование кода

Пробелы относятся к символам, которые используются в форматировании кода, вместе с символами табуляции и, иногда, разрывом строки. Компилятор, как правило, игнорирует пробелы, но все же есть небольшие исключения.

В следующем примере все строки кода выполняют одно и то же:

```
std::cout << "Hello, world!";

std::cout               <<            "Hello, world!";

std::cout << "Hello, world!";

std::cout
    << "Hello, world!";
```

Даже последний стейтмент с разрывом строки успешно скомпилируется.

Аналогично:

```
int add(int x, int y) { return x + y; }

int add(int x, int y) {
    return x + y; }

int add(int x, int y)
{    return x + y; }

int add(int x, int y)
{
    return x + y;
}
```

Исключением, где компилятор учитывает пробелы, является цитируемый текст, например: `"Hello, world!"`.

```
"Hello, world!"
```

отличается от

```
"Hello,      world!"
```

Разрыв/перевод строки не допускается в цитируемом тексте:

```
std::cout << "Hello,
     world!" << std::endl; // Не допускается!
```

Еще одним исключением, где компилятор обращает внимание на пробелы, являются однострочные комментарии: они занимают только одну строку. Перенос

однострочного комментария на вторую строку вызовет ошибку компиляции, например:

```
1. std::cout << "Hello, world!" << std::endl; // это однострочный комментарий
2. А это уже не комментарий
```

Основные рекомендации

В отличие от других языков программирования, C++ не имеет каких-либо ограничений в форматировании кода со стороны программистов. Основное правило заключается в том, чтобы использовать только те способы, которые максимально улучшают читабельность и логичность кода.

Вот 6 основных рекомендаций:

***Рекомендация №1*: Вместо символа табуляции (клавиша "Tab") используйте 4 пробела**. В некоторых IDE по умолчанию используются три пробела в качестве одного символа табуляции — это тоже нормально (количество пробелов можно легко настроить в соответствующих пунктах меню вашей IDE).

Причиной использования пробелов вместо символов табуляции является то, что если вы откроете свой код в другом редакторе, то он сохранит правильные отступы, в отличие от использования символов табуляции.

***Рекомендация №2:* Открытие и закрытие фигурных скобок функции должно находиться на одном уровне на отдельных строках**:

```
1. int main()
2. {
3. }
```

Хотя есть еще и следующий вариант (вы также можете его использовать):

```
1. int main() {
2.     //...
3. }
```

Первый вариант хорош тем, что в случае возникновения ошибки несоответствия скобок, найти те самые проблемные скобки визуально будет проще.

***Рекомендация №3:* Каждый стейтмент функции должен быть с соответствующим отступом** (клавиша Tab или 4 пробела):

```
1. int main()
2. {
3.     std::cout << "Hello world!" << std::endl; // один Tab (4 пробела)
4.     std::cout << "Nice to meet you." << std::endl; // один Tab(4 пробела)
```

```
}
```

Рекомендация №4: **Строки не должны быть слишком длинными**. 72, 78 или 80 символов — это оптимальный максимум строки. Если она будет длиннее, то её следует разбить на несколько отдельных строк:

```
int main()
{
    std::cout << "This is a really, really, really, really, really, really, really, " <<
        "really long line" << std::endl; // один дополнительный отступ для строки-продолжения

    std::cout << "This is another really, really, really, really, really, really, really, " <<
            "really long line" << std::endl; // отступ + выравнивание с учетом главной строки

    std::cout << "This one is short" << std::endl;
}
```

Рекомендация №5: **Если длинная строка разбита на части с помощью определенного оператора (например, << или +), то этот оператор должен находиться в конце этой же строки, а не в начале следующей (так читабельнее).**

Правильно:

```
std::cout << "This is a really, really, really, really, really, really, really, " <<
        "really long line" << std::endl;
```

Неправильно:

```
std::cout << "This is a really, really, really, really, really, really, really, "
        << "really long line" << std::endl;
```

Рекомендация №6: **Используйте пробелы и пропуски строк между стейтментами для улучшения читабельности вашего кода**.

Менее читабельно:

```
nCost = 57;
nPricePerItem = 24;
nValue = 5;
nNumberOfItems = 17;
```

Более читабельно:

```
nCost          = 57;
nPricePerItem  = 24;
nValue         = 5;
nNumberOfItems = 17;
```

Менее читабельно:

```
1. std::cout << "Hello world!" << std::endl; // cout и endl находятся в
   библиотеке iostream
2. std::cout << "It is very nice to meet you!" << std::endl; // эти комментарии
   ухудшают читабельность кода
3. std::cout << "Yeah!" << std::endl; // особенно, когда строки разной длины
```

Более читабельно:

```
1. std::cout << "Hello world!" << std::endl;                  // cout и endl
   находятся в библиотеке iostream
2. std::cout << "It is very nice to meet you!" << std::endl;  // эти комментарии
   более читабельны
3. std::cout << "Yeah!" << std::endl;                         // не так ли?
```

Менее читабельно:

```
1. // cout и endl находятся в библиотеке iostream
2. std::cout << "Hello world!" << std::endl;
3. // эти комментарии ухудшают читабельность кода
4. std::cout << "It is very nice to meet you!" << std::endl;
5. // особенно, когда они в одной куче
6. std::cout << "Yeah!" << std::endl;
```

Более читабельно:

```
1. // cout и endl находятся в библиотеке iostream
2. std::cout << "Hello world!" << std::endl;
3.
4. // эти комментарии читать легче
5. std::cout << "It is very nice to meet you!" << std::endl;
6.
7. // ведь они разделены дополнительными строками
8. std::cout << "Yeah!" << std::endl;
```

Язык C++ позволяет выбрать вам тот стиль форматирования вашего кода, в котором вам будет наиболее комфортно работать.

Урок №22. Прототип функции и Предварительное объявление

Посмотрите на этот, казалось бы, невинный кусочек кода под названием add.cpp:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;
6.     return 0;
7. }
8.
9. int add(int x, int y)
10.{
11.    return x + y;
12.}
```

Вы, наверное, ожидаете увидеть примерно следующий результат:

```
The sum of 3 and 4 is: 7
```

Но в действительности эта программа даже не скомпилируется. Причиной этому является то, что компилятор читает код последовательно. Когда он встречает вызов функции add() в строке №5 функции main(), он даже не знает, что такое add(), так как это еще не определили! В результате чего мы получим следующую ошибку:

```
add: идентификатор не найден
```

Чтобы устранить эту проблему, мы должны учитывать тот факт, что компилятор не знает, что такое add(). Есть 2 решения.

***Решение №1*: Поместить определение функции add() выше её вызова (т.е. перед функцией main())**:

```
1. #include <iostream>
2.
3. int add(int x, int y)
4. {
5.     return x + y;
6. }
7.
8. int main()
9. {
10.    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;
11.    return 0;
12.}
```

Таким образом, при вызове функции add() в функции main(), компилятор будет знать, что это такое. Так как это простая программа, то внести подобные изменения

несложно. Однако в программах, содержащих большое количество строк кода, это может быть утомительно - узнавать кто кого вызывает и в каком порядке (чтобы соблюсти правильную последовательность).

Кроме того, этот вариант не всегда возможен. Например, мы пишем программу, которая имеет две функции: A и B. Если функция A вызывает функцию B, а функция B вызывает функцию A, то нет никакого способа упорядочить эти функции таким образом, чтобы они обе одновременно знали о существовании друг друга. Если вы объявите сначала A, то компилятор будет жаловаться, что не знает, что такое B. Если вы объявите сначала B, то компилятор будет жаловаться, что не знает, что такое A.

Прототипы функций и Предварительное объявление

***Решение №2*: Использовать предварительное объявление.**

Предварительное объявление сообщает компилятору о существовании идентификатора ДО его фактического определения.

В случае функций, мы можем сообщить компилятору о существовании функции до её фактического определения. Для этого нам следует использовать прототип этой функции. **Прототип функции** (полноценный) состоит из типа возврата функции, её имени и параметров (тип + имя параметра). В кратком прототипе отсутствуют имена параметров функции. Основная часть (между фигурными скобками) опускается. А поскольку прототип функции является стейтментом, то он также заканчивается точкой с запятой.

Вот прототип функции add():

```
1. int add(int x, int y); // прототип функции состоит из типа возврата функции, её
    имени, параметров и точки с запятой
```

А вот вышеприведенная программа, но уже с прототипом функции в качестве предварительного объявления add():

```
1. #include <iostream>
2.
3. int add(int x, int y); // предварительное объявление функции add()
   (используется её прототип)
4.
5. int main()
6. {
7.     std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl; // это
   работает, так как мы предварительно (выше функции main()) объявили функцию
   add()
8.     return 0;
9. }
10.
```

```
11. int add(int x, int y) // хотя определение функции add() находится ниже её
    вызова
12. {
13.     return x + y;
14. }
```

Теперь, когда компилятор встречает вызов функции add() в main(), он знает, что это такое и где это искать.

Стоит отметить, что в прототипах функций можно и не указывать имена параметров. Например, прототип выше мы можем записать следующим образом:

```
1. int add(int, int);
```

Тем не менее, предпочтительнее указывать имена параметров, чтобы не путаться лишний раз.

> ***Лайфхак***: Прототипы функций можно легко создавать с помощью копирования/вставки из фактического определения функции. Просто не забывайте указывать точку с запятой в конце.

Предварительно объявили, но не определили

Вопрос: «А что произойдет, если мы предварительно объявим функцию, но не запишем её определение?". Однозначного ответа нет. Если предварительное объявление записано, но функция никогда не вызывается, то программа может запуститься без ошибок. Однако, если предварительное объявление записано, функция вызывается, но её определения нет, то вы получите ошибку на этапе линкинга: программа просто не сможет обработать вызов этой функции.

Рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int add(int x, int y); // предварительное объявление функции add()
   (используется её прототип)
4.
5. int main()
6. {
7.     std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;
8.     return 0;
9. }
```

В этой программе мы предварительно объявили функцию add(), вызвали её в main(), но не записали её определения. При попытке компиляции этой программы мы получим ошибку от линкера.

Объявление vs. Определение

В языке C++ вы часто будете слышать слова "объявление" и "определение". Что это такое?

Определение фактически реализует (вызывает выделение памяти) идентификатор. Вот примеры определений:

```
1. int add(int x, int y) // определяем функцию add()
2. {
3.     int z = x + y; // определяем переменную z
4.
5.     return z;
6. }
```

Определение необходимо для корректной работы линкера. Если вы используете идентификатор без его определения, то линкер выдаст вам ошибку.

В языке C++ есть **правило одного определения**, которое состоит из 3-х частей:

- *Внутри файла* функция, объект, тип или шаблон могут иметь только одно определение.
- *Внутри программы* объект или обычная функция могут иметь только одно определение.
- *Внутри программы* типы, шаблоны функций и встроенные функции могут иметь несколько определений, если они идентичны.

Нарушение первой части правила приведет к ошибке компиляции. Нарушение второй или третьей части правила приведет к ошибке линкинга.

Объявление — это стейтмент, который сообщает компилятору о существовании идентификатора и о его типе. Вот примеры объявлений:

```
1. int add(int x, int y); // сообщаем компилятору о функции add(), которая имеет
   два параметра типа int и возвращает целочисленное значение
2. int x; // объявляем целочисленную переменную x
```

Объявление — это всё, что необходимо для корректной работы компилятора, но недостаточно для корректной работы линкера. Определение - это то, что обеспечит корректную работу как компилятора, так и линкера.

Тест

Задание №1: В чём разница между прототипом функции и предварительным объявлением?

Задание №2: Запишите прототип следующей функции:

```
int doMath(int first, int second, int third, int fourth)
{
    return first + second * third / fourth;
}
```

Задание №3: Выясните, какие из следующих программ не пройдут этап компиляции, какие не пройдут этап линкинга, а какие не пройдут и то, и другое?

Программа №1:

```
#include <iostream>

int add(int x, int y);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << std::endl;
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

Программа №2:

```
#include <iostream>

int add(int x, int y);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << std::endl;
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Программа №3:

```
#include <iostream>

int add(int x, int y);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4) << std::endl;
    return 0;
}

int add(int x, int y, int z)
```

```
12. {
13.     return x + y + z;
14. }
```

Программа №4:

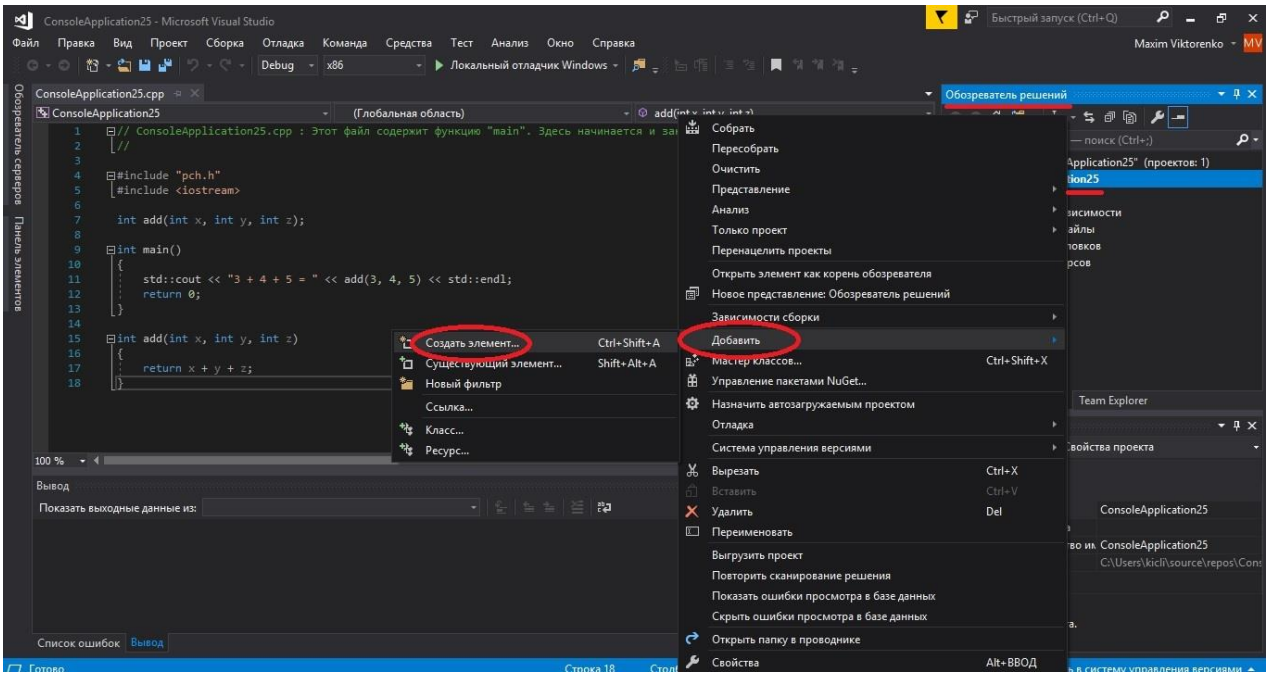
```
1.  #include <iostream>
2.
3.  int add(int x, int y, int z);
4.
5.  int main()
6.  {
7.      std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << std::endl;
8.      return 0;
9.  }
10.
11. int add(int x, int y, int z)
12. {
13.     return x + y + z;
14. }
```

Урок №23. Многофайловые программы

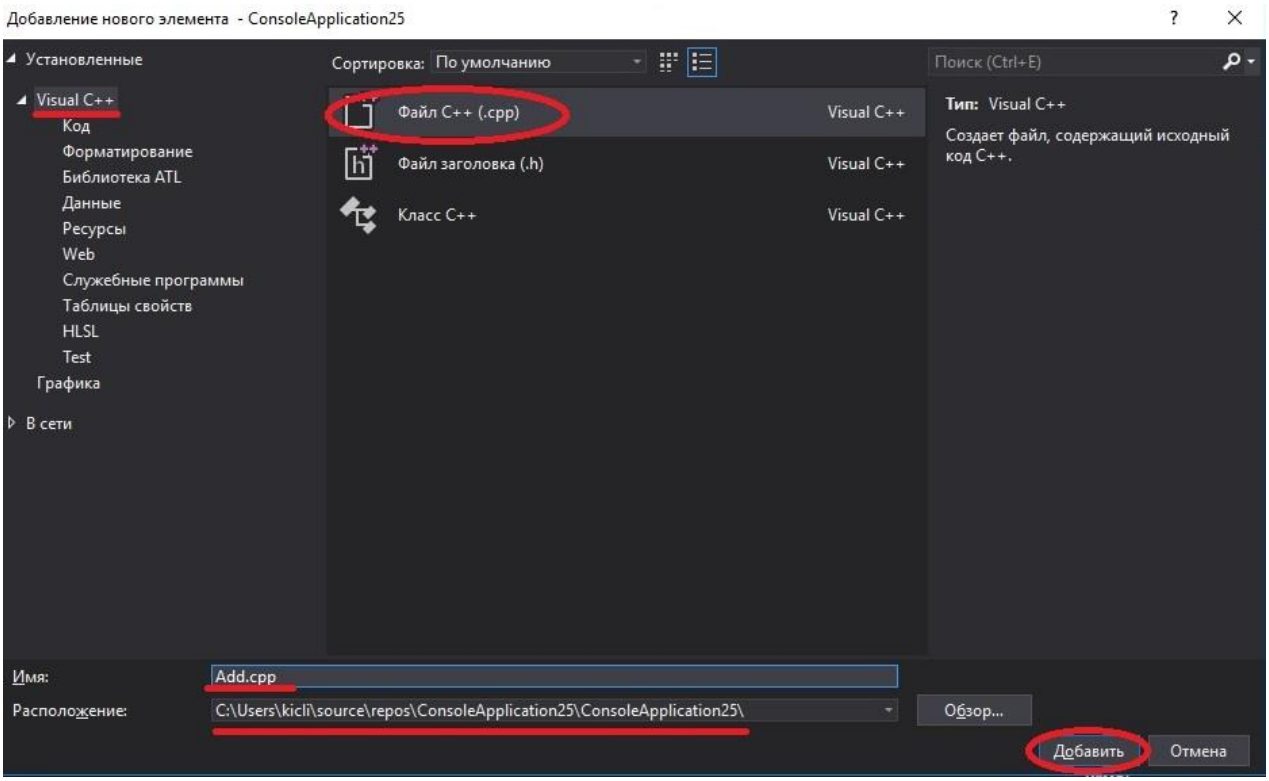
Как только программы становятся больше, их следует разбивать на несколько файлов (в целях удобства и улучшения функциональности). Одним из преимуществ использования IDE является легкость в работе с n-ным количеством файлов. Мы уже знаем, как создавать и компилировать однофайловые проекты, добавление новых файлов не составит труда.

Многофайловые проекты в Visual Studio

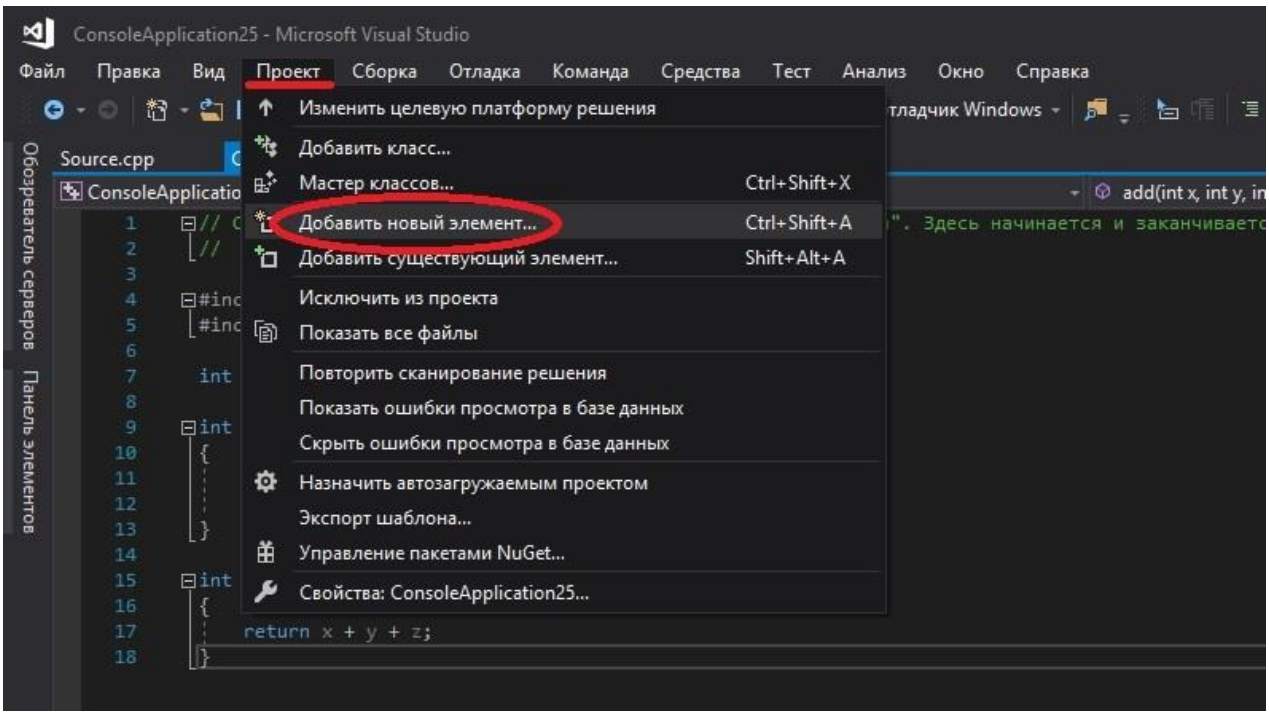
В Visual Studio щелкните правой кнопкой мыши по имени вашего проекта в меню `"Обозреватель решений"`, затем `"Добавить" > "Создать элемент..."`:



Во всплывающем диалоговом окне выберите тип файла, укажите его имя, расположение, а затем нажмите `"Добавить"`:

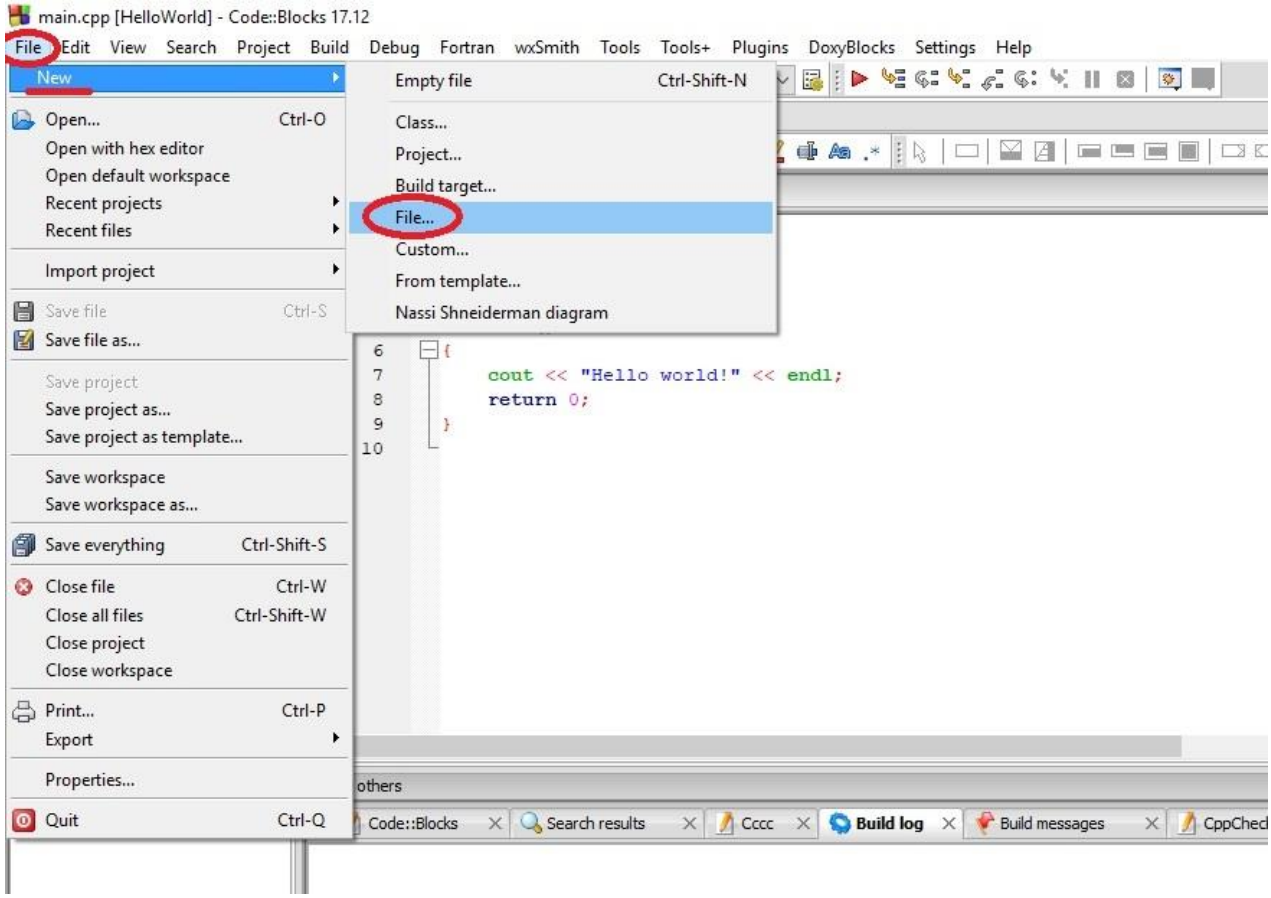


Также вы можете добавлять файлы к вашему проекту через `"Проект" > "Добавить новый элемент..."`:

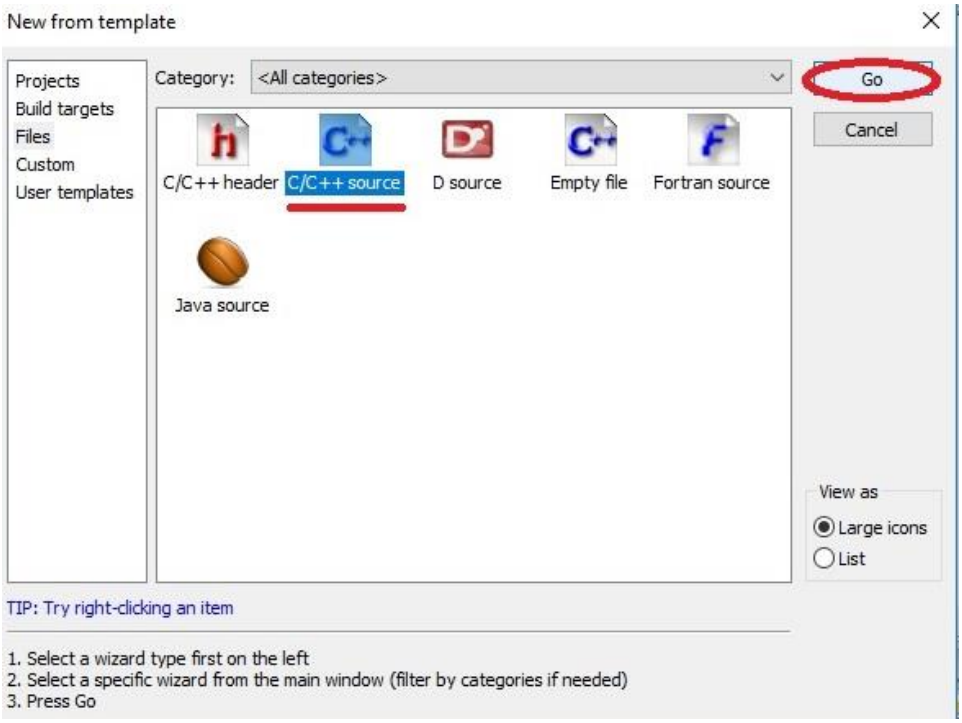


Многофайловые проекты в Code::Blocks

В Code::Blocks перейдите в `"File" > "New" > "File..."`:

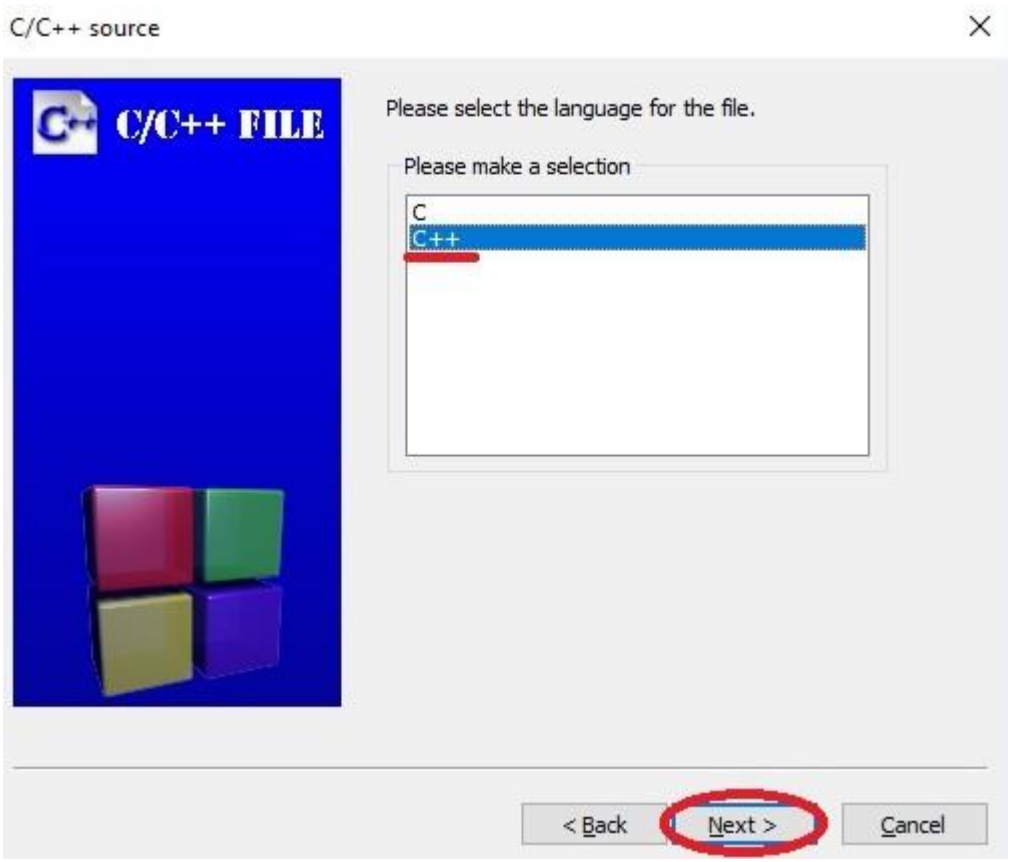


Затем выберите `"C/C++ source"` и нажмите `"Go"`:

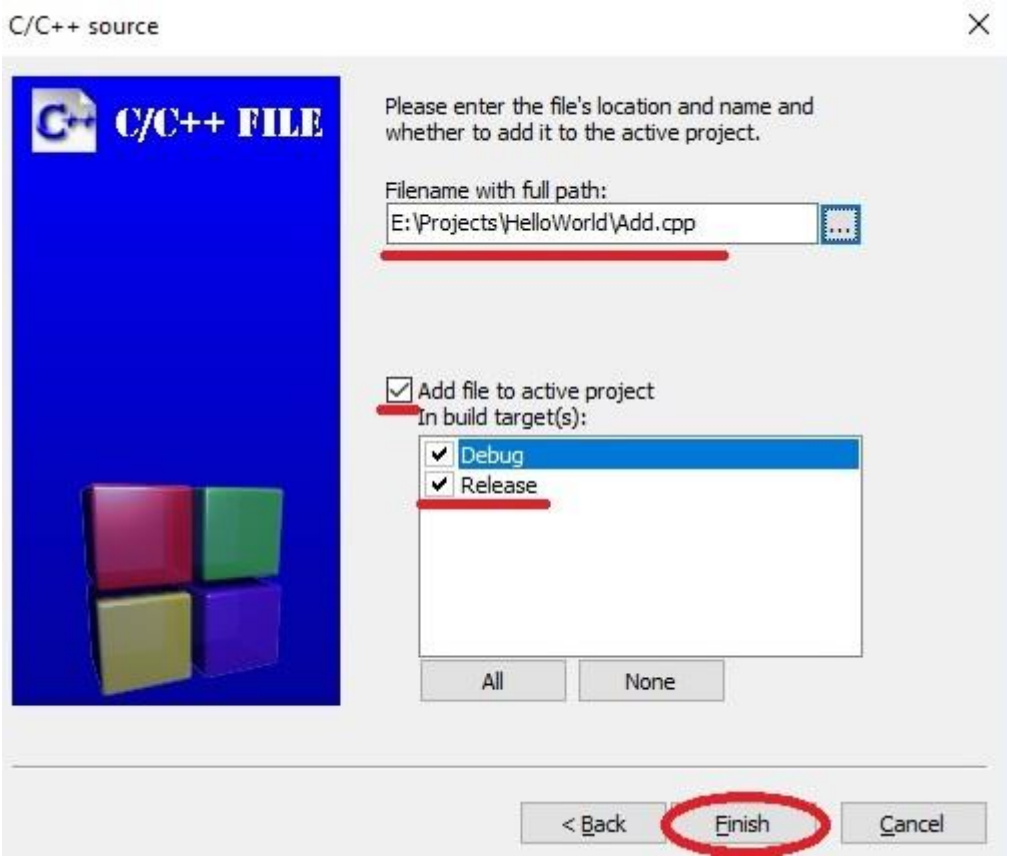


Затем `"Next"` (этого окна может и не быть):

Затем `"C++"` и опять `"Next"`:



Затем укажите имя нового файла (не забудьте расширение .cpp) и его расположение (нажмите на троеточие и выберите путь). Убедитесь, что поставлены все три галочки (они отвечают за конфигурации сборки). Затем нажмите `"Finish"`:



Готово! Файл добавлен.

Многофайловые проекты в GCC/G++

В командной строке вам нужно будет создать файл, указать его имя и подключить к компиляции, например:

```
g++ main.cpp add.cpp -o main
```

(где *main.cpp* и *add.cpp* - это имена файлов с кодом, а *main* - это имя файла-результата)

Пример многофайловой программы

Рассмотрим следующую программу, которая состоит из двух файлов.

add.cpp:

```
1. int add(int x, int y)
2. {
```

```
3.      return x + y;
4.  }
```

main.cpp:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;
6.      return 0;
7.  }
```

Попробуйте запустить эту программу. Она не скомпилируется, вы получите следующую ошибку:

```
add: идентификатор не найден
```

При компиляции кода, компилятор не знает о существовании функций, которые находятся в других файлах. Это сделано специально, чтобы функции и переменные с одинаковыми именами, но в разных файлах, не вызывали конфликт имен.

Тем не менее, в данном случае, мы хотим, чтобы main.cpp знал (и использовал) функцию add(), которая находится в add.cpp. Для предоставления доступа main.cpp к функциям add.cpp, нам нужно использовать предварительное объявление:

```
1.  #include <iostream>
2.
3.  int add(int x, int y); // это нужно для того, чтобы main.cpp знал, что функция
    add() определена в другом месте
4.
5.  int main()
6.  {
7.      std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;
8.      return 0;
9.  }
```

Теперь, когда компилятор будет компилировать main.cpp, он будет знать, что такое add(). Попробуйте запустить эту программу еще раз.

Что-то пошло не так!

Есть много вещей, которые могут пойти не так, особенно, если вы это делаете в первый раз. Главное - не паниковать:

Пункт №1: Если вы получили ошибку от компилятора, что функция add() не определена в main(), то, скорее всего, вы забыли записать предварительное объявление функции add() в main.cpp.

Пункт №2: Если вы получили следующую ошибку от линкера:

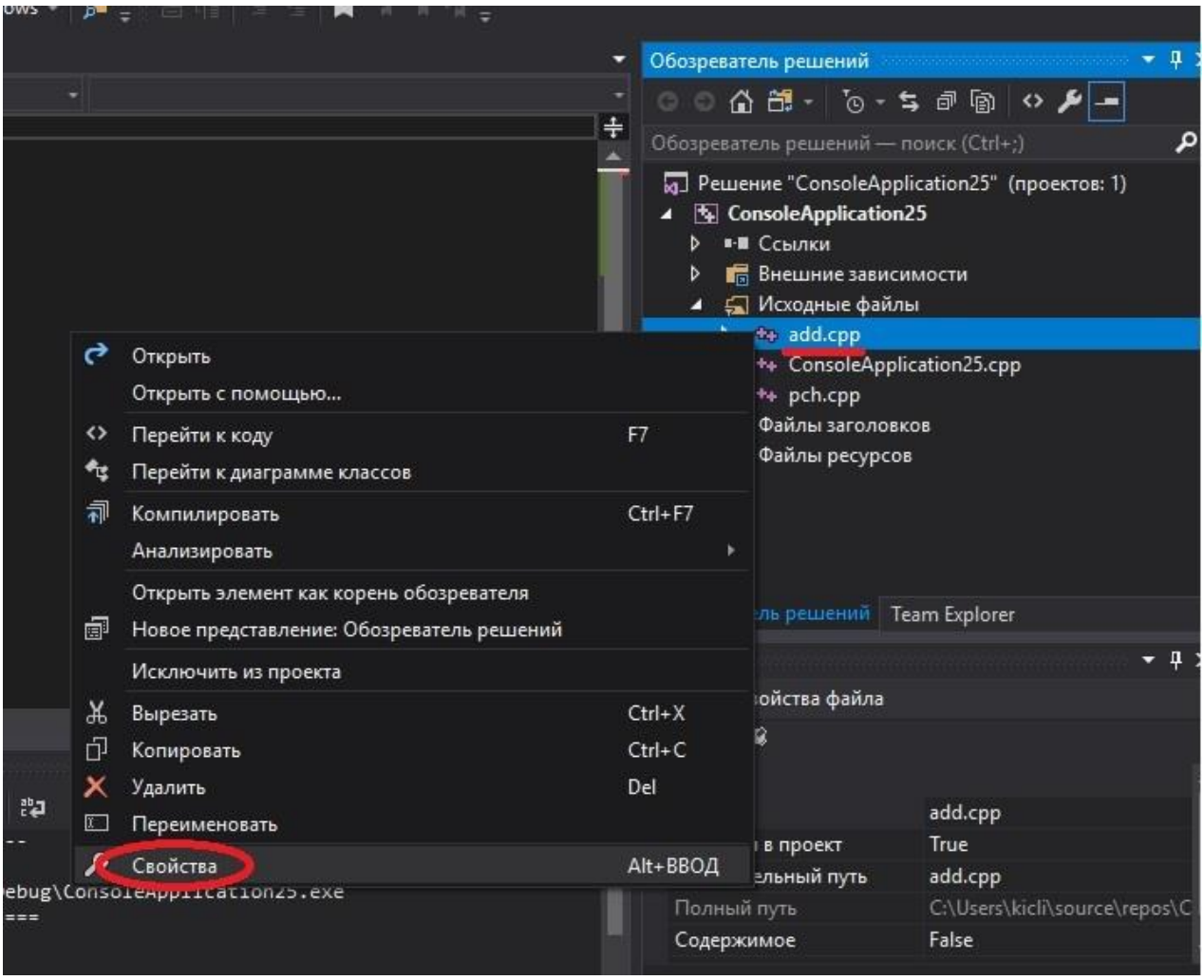
```
unresolved external symbol "int __cdecl add(int,int)"
(?add@@YAHHH@Z) referenced in function _main
```

то возможных решений есть несколько:

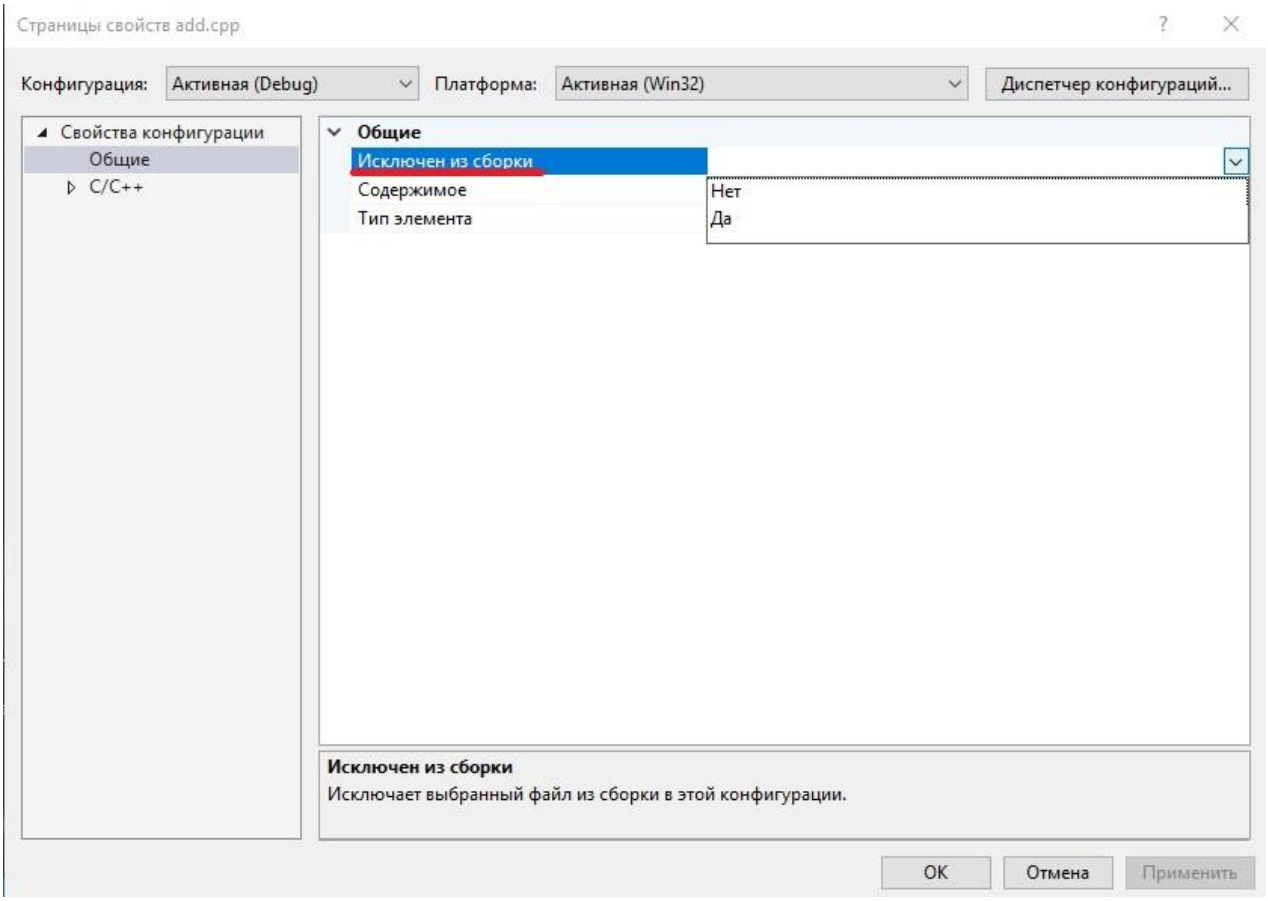
а) Скорее всего, add.cpp некорректно добавлен в ваш проект. Если вы используете Visual Studio или Code::Blocks, то вы должны увидеть add.cpp в `"Обозревателе решений"` в списке файлов вашего проекта или в панели проекта IDE. Если добавленного файла нет, то щелкните правой кнопкой мыши по вашему проекту и добавьте файл, как это показано выше, а затем повторите попытку компиляции вашего проекта.

б) Вполне возможно, что вы добавили add.cpp к другому проекту.

в) Вполне возможно, что добавленный файл не подключен к компиляции/линкингу. Щелкните правой кнопкой мыши по имени вашего добавленного файла и выберите `"Свойства"`:



Убедитесь, что пункт `"Исключен из сборки"` оставлен пустым или выбрано значение `"Нет"`:



Пункт №3: Не следует писать следующую строку в main.cpp:

```
1. #include "add.cpp"
```

Наличие этой строки приведет к тому, что компилятор вставит всё содержимое add.cpp непосредственно в main.cpp вместо того, чтобы рассматривать эти файлы как отдельные.

Тест

Разделите следующую программу на два файла (main.cpp и input.cpp): main.cpp должен содержать функцию main(), а input.cpp — функцию getInteger().

Помните, что для функции getInteger() вам понадобится предварительное объявление в main.cpp.

```
1. #include <iostream>
2.
3. int getInteger()
4. {
5.     std::cout << "Enter an integer: ";
```

```
6.      int x;
7.      std::cin >> x;
8.      return x;
9.  }
10.
11. int main()
12. {
13.     int x = getInteger();
14.     int y = getInteger();
15.
16.     std::cout << x << " + " << y << " is " << x + y << '\n';
17.     return 0;
18. }
```

Урок №24. Заголовочные файлы

По мере увеличения размера программ весь код уже не помещается в нескольких файлах, записывать каждый раз предварительные объявления для функций, которые мы хотим использовать, но которые находятся в других файлах, становится всё утомительнее и утомительнее. Хорошо было бы, если бы все предварительные объявления находились в одном месте, не так ли?

Файлы .cpp не являются единственными файлами в проектах. Есть еще один тип файлов - **заголовочные файлы** (или ***"заголовки"***), которые имеют расширение `.h`. Целью заголовочных файлов является удобное хранение набора объявлений объектов для их последующего использования в других программах.

Заголовочные файлы из Стандартной библиотеки C++

Рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Hello, world!" << std::endl;
6.     return 0;
7. }
```

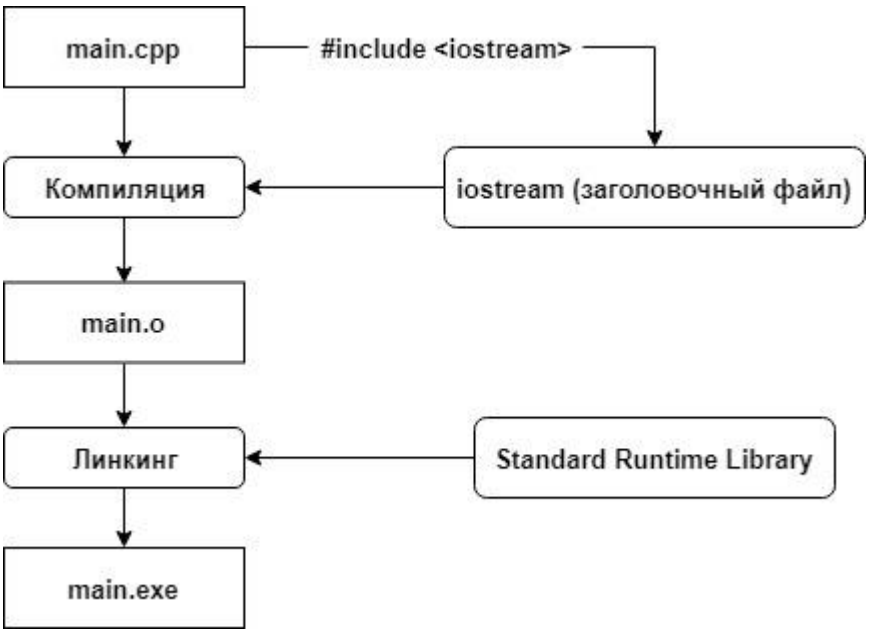
Результат выполнения программы:

```
Hello, world!
```

В этой программе мы используем cout, который нигде не определяем. Откуда компилятор знает, что это такое? Дело в том, что cout объявлен в заголовочном файле iostream. Когда мы пишем `#include <iostream>`, мы делаем запрос, чтобы всё содержимое заголовочного файла iostream было скопировано в наш файл. Таким образом, всё содержимое библиотеки iostream становится доступным для использования.

Как правило, в заголовочных файлах записываются только объявления, без определений. Следовательно, если cout только объявлен в заголовочном файле iostream, то где же он определяется?

Ответ: в Стандартной библиотеке C++, которая автоматически подключается к вашему проекту на этапе линкинга.



Подумайте о последствиях отсутствия заголовочного файла iostream. Каждый раз, при использовании cout, вам бы приходилось вручную копировать все предварительные объявления, связанные с cout в верхнюю часть вашего файла! Хорошо ведь, что можно просто указать `#include <iostream>`, не так ли?

Пишем свои собственные заголовочные файлы

Теперь давайте вернемся к примеру, который мы обсуждали на предыдущем уроке. У нас было два файла: add.cpp и main.cpp.

add.cpp:

```
1. int add(int x, int y)
2. {
3.     return x + y;
4. }
```

main.cpp:

```
1. #include <iostream>
2.
3. int add(int x, int y); // предварительное объявление с использованием прототипа
    функции
4.
5. int main()
6. {
7.     std::cout << "The sum of 3 and 4 is " << add(3, 4) << std::endl;
8.     return 0;
9. }
```

> ***Примечание:*** Если вы создаете все файлы заново, то не забудьте добавить add.cpp в свой проект, чтобы он был подключен к компиляции.

Мы использовали предварительное объявление, чтобы сообщить компилятору, что такое add(). Как мы уже говорили, записывать в каждом файле предварительные объявления используемых функций - дело не слишком увлекательное.

И здесь нам на помощь приходят заголовочные файлы. Достаточно просто написать один заголовочный файл и его можно будет повторно использовать в любом количестве программ. Также и вносить изменения в такой код (например, добавление еще одного параметра) гораздо легче, нежели чем шерстить по всем файлам в поисках используемых функций.

Написать свой собственный заголовочный файл не так уж и сложно. Заголовочные файлы состоят из двух частей:

- **Директивы препроцессора** — в частности, header guards, которые предотвращают вызов заголовочного файла больше одного раза из одного и того же файла (об этом детально на следующем уроке).
- **Содержимое заголовочного файла** — набор объявлений.

Все ваши заголовочные файлы (которые вы написали самостоятельно) должны иметь расширение `.h`.

add.h:

```
1. // Начнем с директив препроцессора. ADD_H – это произвольное уникальное имя (обычно используется имя заголовочного файла)
2. #ifndef ADD_H
3. #define ADD_H
4.
5. // А это уже содержимое заголовочного файла
6. int add(int x, int y); // прототип функции add() (не забывайте точку с запятой в конце!)
7.
8. // Заканчиваем директивой препроцессора
9. #endif
```

Чтобы использовать этот файл в main.cpp, вам сначала нужно будет подключить его к проекту.

main.cpp, в котором мы подключаем add.h:

```
1. #include <iostream>
2. #include "add.h"
3.
4. int main()
```

```
5. {
6.     std::cout << "The sum of 3 and 4 is " << add(3, 4) << std::endl;
7.     return 0;
8. }
```

add.cpp остается без изменений:

```
1. int add(int x, int y)
2. {
3.     return x + y;
4. }
```

Когда компилятор встречает `#include "add.h"`, он копирует всё содержимое add.h в текущий файл. Таким образом, мы получаем предварительное объявление функции add().

> ***Примечание**:* При подключении заголовочного файла, всё его содержимое вставляется сразу же после строки `#include ....`

Если вы получили ошибку от компилятора, что add.h не найден, то убедитесь, что имя вашего файла точно "add.h". Вполне возможно, что вы могли сделать опечатку, например, просто "add" (без ".h") или "add.h.txt" или "add.hpp".

Если вы получили ошибку от линкера, что функция add() не определена, то убедитесь, что вы корректно подключили add.cpp к вашему проекту (и к компиляции тоже)!

Угловые скобки (<>) vs. Двойные кавычки ("")

Вы, наверное, хотите узнать, почему используются угловые скобки для iostream и двойные кавычки для add.h. Дело в том, что, используя угловые скобки, мы сообщаем компилятору, что подключаемый заголовочный файл написан не нами (он является "системным", т.е. предоставляется Стандартной библиотекой C++), так что искать этот заголовочный файл следует в системных директориях. Двойные кавычки сообщают компилятору, что мы подключаем наш собственный заголовочный файл, который мы написали самостоятельно, поэтому искать его следует в текущей директории нашего проекта. Если файла там не окажется, то компилятор начнет проверять другие пути, в том числе и системные директории.

Правило: Используйте угловые скобки для подключения "системных" заголовочных файлов и двойные кавычки для ваших заголовочных файлов.

Стоит отметить, что одни заголовочные файлы могут подключать другие заголовочные файлы. Тем не менее, так делать не рекомендуется.

Почему iostream пишется без окончания .h?

Еще один часто задаваемый вопрос: "Почему iostream (или любой другой из стандартных заголовочных файлов) при подключении пишется без окончания ".h"?". Дело в том, что есть 2 отдельных файла: iostream.h (заголовочный файл) и просто iostream! Для объяснения потребуется краткий экскурс в историю.

Когда C++ только создавался, все файлы библиотеки Runtime имели окончание .h. Оригинальные версии cout и cin объявлены в iostream.h. При стандартизации языка C++ комитетом ANSI, решили перенести все функции из библиотеки Runtime в пространство имен std, чтобы предотвратить возможность возникновения конфликтов имен с пользовательскими идентификаторами (что, между прочим, является хорошей идеей). Тем не менее, возникла проблема: если все функции переместить в пространство имен std, то старые программы переставали работать!

Для обеспечения обратной совместимости ввели новый набор заголовочных файлов с теми же именами, но без окончания ".h". Весь их функционал находится в пространстве имен std. Таким образом, старые программы с `#include <iostream.h>` не нужно было переписывать, а новые программы уже могли использовать `#include <iostream>`.

Когда вы подключаете заголовочный файл из Стандартной библиотеки C++, убедитесь, что вы используете версию без .h (если она существует). В противном случае, вы будете использовать устаревшую версию заголовочного файла, который уже больше не поддерживается.

Кроме того, многие библиотеки, унаследованные от языка Си, которые до сих пор используются в C++, также были продублированы с добавлением префикса `c` (например, stdlib.h стал ***c***stdlib). Функционал этих библиотек также перенесли в пространство имен std, чтобы избежать возможность возникновения конфликтов имен с пользовательскими идентификаторами.

Правило: При подключении заголовочных файлов из Стандартной библиотеки C++, используйте версию без ".h" (если она существует). Пользовательские заголовочные файлы должны иметь окончание ".h".

Можно ли записывать определения в заголовочных файлах?

C++ не будет жаловаться, если вы это сделаете, но так делать не принято.

Как уже было сказано выше, при подключении заголовочного файла, всё его содержимое вставляется сразу же после строки с #include. Это означает, что любые определения, которые есть в заголовочном файле, скопируются в ваш файл.

Для небольших проектов, это, скорее всего, не будет проблемой. Но для более крупных это может способствовать увеличению времени компиляции (так как код будет повторно компилироваться) и размеру исполняемого файла. Если внести изменения в определения, которые находятся в файле .cpp, то перекомпилировать придется только этот файл. Если же внести изменения в определения, которые записаны в заголовочном файле, то перекомпилировать придется каждый файл, который подключает этот заголовок, используя директиву препроцессора #include. И вероятность того, что из-за одного небольшого изменения вам придется перекомпилировать весь проект, резко возрастает!

Иногда делаются исключения для простых функций, которые вряд ли изменятся (например, где определение состоит всего лишь из одной строки).

Советы

Вот несколько советов по написанию собственных заголовочных файлов:

- Всегда используйте директивы препроцессора.
- Не определяйте переменные в заголовочных файлах, если это не константы. Заголовочные файлы следует использовать только для объявлений.
- Не определяйте функции в заголовочных файлах.
- Каждый заголовочный файл должен выполнять свое конкретное задание и быть как можно более независимым. Например, вы можете поместить все ваши объявления, связанные с файлом A.cpp в файл A.h, а все ваши объявления, связанные с B.cpp — в файл B.h. Таким образом, если вы будете работать только с A.cpp, то вам будет достаточно подключить только A.h и наоборот.
- Используйте имена ваших рабочих файлов в качестве имен для ваших заголовочных файлов (например, grades.h работает с grades.cpp).
- Не подключайте одни заголовочные файлы из других заголовочных файлов.
- Не подключайте файлы .cpp, используя директиву препроцессора #include.

Урок №25. Директивы препроцессора

Препроцессор лучше всего рассматривать как отдельную программу, которая выполняется перед компиляцией. При запуске программы, препроцессор просматривает код сверху вниз, файл за файлом, в поиске директив. **Директивы** - это специальные команды, которые начинаются с символа `#` и НЕ заканчиваются точкой с запятой. Есть несколько типов директив, которые мы рассмотрим ниже.

Директива #include

Вы уже видели директиву #include в действии. Когда вы подключаете файл с помощью директивы #include, препроцессор копирует содержимое подключаемого файла в текущий файл сразу после строки с #include. Это очень полезно при использовании определенных данных (например, предварительных объявлений функций) сразу в нескольких местах.

Директива #include имеет две формы:

- `#include <filename>`, которая сообщает препроцессору искать файл в системных путях (в местах хранения системных библиотек языка C++). Чаще всего вы будете использовать эту форму при подключении заголовочных файлов из Стандартной библиотеки C++.
- `#include "filename"`, которая сообщает препроцессору искать файл в текущей директории проекта. Если его там не окажется, то препроцессор начнет проверять системные пути и любые другие, которые вы указали в настройках вашей IDE. Эта форма используется для подключения пользовательских заголовочных файлов.

Директива #define

Директиву #define можно использовать для создания макросов. **Макрос** - это правило, которое определяет конвертацию идентификатора в указанные данные.

Есть два основных типа макросов: макросы-функции и макросы-объекты.

Макросы-функции ведут себя как функции и используются в тех же целях. Мы не будем сейчас их обсуждать, так как их использование, как правило, считается опасным, и почти всё, что они могут сделать, можно осуществить с помощью простой (линейной) функции.

Макросы-объекты можно определить одним из следующих двух способов:

```
#define идентификатор
```

или

```
#define идентификатор текст_замена
```

Верхнее определение не имеет никакого `текст_замена`, в то время как нижнее - имеет. Поскольку это директивы препроцессора (а не простые стейтменты), то ни одна из форм не заканчивается точкой с запятой.

Макросы-объекты с текст_замена

Когда препроцессор встречает макросы-объекты с `текст_замена`, то любое дальнейшее появление `идентификатор` заменяется на `текст_замена`. `идентификатор` обычно пишется заглавными буквами с символами подчёркивания вместо пробелов.

Рассмотрим следующий фрагмент кода:

```
1. #define MY_FAVORITE_NUMBER 9
2.
3. std::cout << "My favorite number is: " << MY_FAVORITE_NUMBER << std::endl;
```

Препроцессор преобразует вышеприведенный код в:

```
1. std::cout << "My favorite number is: " << 9 << std::endl;
```

Результат выполнения:

```
My favorite number is: 9
```

Мы обсудим это детально, и почему так не стоит делать, на следующих уроках.

Макросы-объекты без текст_замена

Макросы-объекты также могут быть определены без `текст_замена`, например:

```
1. #define USE_YEN
```

Любое дальнейшее появление идентификатора `USE_YEN` удаляется и заменяется «ничем» (пустым местом)!

Это может показаться довольно бесполезным, однако, это не основное предназначение подобных директив. В отличие от макросов-объектов с `текст_замена`, эта форма макросов считается приемлемой для использования.

Условная компиляция

Директивы препроцессора условной компиляции позволяют определить, при каких условиях код будет компилироваться, а при каких - нет. На этом уроке мы рассмотрим только **три директивы условной компиляции**:

- `#ifdef`
- `#ifndef`
- `#endif`

Директива #ifdef (сокр. от *"**if def**ined" = "если определено"*) позволяет препроцессору проверить, было ли значение ранее определено с помощью директивы #define. Если да, то код между #ifdef и #endif скомпилируется. Если нет, то код будет проигнорирован. Например:

```
#define PRINT_JOE

#ifdef PRINT_JOE
std::cout << "Joe" << std::endl;
#endif

#ifdef PRINT_BOB
std::cout << "Bob" << std::endl;
#endif
```

Поскольку `PRINT_JOE` уже был определен, то строка `std::cout << "Joe" << std::endl;` скомпилируется и выполнится. А поскольку `PRINT_BOB` не был определен, то строка `std::cout << "Bob" << std::endl;` не скомпилируется и, следовательно, не выполнится.

Директива #ifndef (сокр. от *"**if n**ot **def**ined" = "если не определено"*) — это полная противоположность к #ifdef, которая позволяет проверить, не было ли значение ранее определено. Например:

```
#ifndef PRINT_BOB
std::cout << "Bob" << std::endl;
#endif
```

Результатом выполнения этого фрагмента кода будет `Bob`, так как `PRINT_BOB` ранее никогда не был определен. Условная компиляция очень часто используется в качестве header guards (о них мы поговорим на следующем уроке).

Область видимости директивы #define

Директивы выполняются перед компиляцией программы: сверху вниз, файл за файлом. Рассмотрим следующую программу:

```
#include <iostream>

void boo()
{
#define MY_NAME "Alex"
}

int main()
{
    std::cout << "My name is: " << MY_NAME;

    return 0;
}
```

Несмотря на то, что директива `#define MY_NAME "Alex"` определена внутри функции boo(), препроцессор этого не заметит, так как он не понимает такие понятия языка C++, как функции. Следовательно, выполнение этой программы будет идентично той, в которой бы `#define MY_NAME "Alex"` было определено ДО, либо сразу ПОСЛЕ функции boo(). Для лучше читабельности кода определяйте идентификаторы (с помощью #define) вне функций.

После того, как препроцессор завершит свое выполнение, все идентификаторы (определенные с помощью #define) из этого файла — отбрасываются. Это означает, что директивы действительны только с точки определения и до конца файла, в котором они определены. Директивы, определенные в одном файле кода, не влияют на директивы, определенные внутри других файлов этого же проекта.

Рассмотрим следующий пример:

function.cpp:

```
#include <iostream>

void doSomething()
{
#ifdef PRINT
    std::cout << "Printing!";
#endif
#ifndef PRINT
    std::cout << "Not printing!";
#endif
}
```

main.cpp:

```
1. void doSomething(); // предварительное объявление функции doSomething()
2. 
3. int main()
4. {
5. #define PRINT
6. 
7.     doSomething();
8. 
9.     return 0;
10. }
```

Результат выполнения программы:

```
Not printing!
```

Несмотря на то, что мы объявили `PRINT` в main.cpp (`#define PRINT`), это все равно не имеет никакого влияния на что-либо в function.cpp. Поэтому, при выполнении функции doSomething(), у нас выводится `Not printing!`, так как в файле function.cpp мы не объявляли идентификатор `PRINT` (с помощью директивы #define). Это связано с header guards.

Урок №26. Header guards и #pragma once

Как мы уже знаем из урока о предварительных объявлениях, идентификатор может иметь только одно объявление. Таким образом, программа с двумя объявлениями одной переменной получит ошибку компиляции:

```
1. int main()
2. {
3.     int x; // это объявление идентификатора x
4.     int x; // ошибка компиляции: дублирование объявлений
5.
6.     return 0;
7. }
```

То же самое касается и функций:

```
1. #include <iostream>
2.
3. int boo()
4. {
5.     return 7;
6. }
7.
8. int boo() // ошибка компиляции: дублирование определений
9. {
10.     return 7;
11. }
12.
13. int main()
14. {
15.     std::cout << boo();
16.     return 0;
17. }
```

Хотя вышеприведенные ошибки легко исправить (достаточно просто удалить дублирование), с заголовочными файлами дела обстоят несколько иначе. Довольно легко можно попасть в ситуацию, когда определения одних и тех же заголовочных файлов будут подключаться больше одного раза в файл .cpp. Очень часто это случается при подключении одного заголовочного файла другим.

Рассмотрим следующую программу:

math.h:

```
1. int getSquareSides()
2. {
3.     return 4;
4. }
```

geometry.h:

```
#include "math.h"
```

main.cpp:

```
#include "math.h"
#include "geometry.h"

int main()
{
    return 0;
}
```

Эта, казалось бы, невинная программа, не скомпилируется! Проблема кроется в определении функции в файле math.h. Давайте детально рассмотрим, что здесь происходит:

- Сначала main.cpp подключает заголовочный файл math.h, вследствие чего определение функции getSquareSides копируется в main.cpp.
- Затем main.cpp подключает заголовочный файл geometry.h, который, в свою очередь, подключает math.h.
- В geometry.h находится копия функции getSquareSides (из файла math.h), которая уже во второй раз копируется в main.cpp.

Таким образом, после выполнения всех директив #include, main.cpp будет выглядеть следующим образом:

```
int getSquareSides()  // из math.h
{
    return 4;
}

int getSquareSides() // из geometry.h
{
    return 4;
}

int main()
{
    return 0;
}
```

Мы получим дублирование определений и ошибку компиляции. Если рассматривать каждый файл по отдельности, то ошибок нет. Однако в main.cpp, который подключает сразу два заголовочных файла с одним и тем же определением функции, мы столкнемся с проблемами. Если для geometry.h нужна функция getSquareSides(), а для main.cpp нужен как geometry.h, так и math.h, то какое же решение?

Header guards

На самом деле решение простое — использовать header guards (защиту подключения в языке C++). **Header guards** — это директивы условной компиляции, которые состоят из следующего:

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// Основная часть кода

#endif
```

Если подключить этот заголовочный файл, то первое, что он сделает - это проверит, был ли ранее определен идентификатор `SOME_UNIQUE_NAME_HERE`. Если мы впервые подключаем этот заголовок, то `SOME_UNIQUE_NAME_HERE` еще не был определен. Следовательно, мы определяем `SOME_UNIQUE_NAME_HERE` (с помощью директивы #define) и выполняется основная часть заголовочного файла. Если же мы раньше подключали этот заголовочный файл, то `SOME_UNIQUE_NAME_HERE` уже был определен. В таком случае, при подключении этого заголовочного файла во второй раз, его содержимое будет проигнорировано.

Все ваши заголовочные файлы должны иметь header guards. `SOME_UNIQUE_NAME_HERE` может быть любым идентификатором, но, как правило, в качестве идентификатора используется имя заголовочного файла с окончанием `_H`. Например, в файле math.h идентификатор будет `MATH_H`:

math.h:

```
#ifndef MATH_H
#define MATH_H

int getSquareSides()
{
    return 4;
}

#endif
```

Даже заголовочные файлы из Стандартной библиотеки C++ используют header guards.

Если бы вы взглянули на содержимое заголовочного файла iostream, то вы бы увидели:

```
#ifndef _IOSTREAM_
#define _IOSTREAM_

```

```
4.  // Основная часть кода
5.
6.  #endif
```

Но сейчас вернемся к нашему примеру с math.h, где мы попытаемся исправить ситуацию с помощью header guards:

math.h:

```
1.  #ifndef MATH_H
2.  #define MATH_H
3.
4.  int getSquareSides()
5.  {
6.      return 4;
7.  }
8.
9.  #endif
```

geometry.h:

```
1.  #include "math.h"
```

main.cpp:

```
1.  #include "math.h"
2.  #include "geometry.h"
3.
4.  int main()
5.  {
6.      return 0;
7.  }
```

Теперь, при подключении в main.cpp заголовочного файла math.h, препроцессор увидит, что `MATH_H` не был определен, следовательно, выполнится директива определения `MATH_H` и содержимое math.h скопируется в main.cpp. Затем main.cpp подключает заголовочный файл geometry.h, который, в свою очередь, подключает math.h. Препроцессор видит, что `MATH_H` уже был ранее определен и содержимое geometry.h не будет скопировано в main.cpp.

Вот так можно бороться с дублированием определений с помощью header guards.

#pragma once

Большинство компиляторов поддерживают более простую, альтернативную форму header guards - **директиву #pragma**:

```
1.  #pragma once
2.
3.  // Основная часть кода
```

`#pragma once` используется в качестве header guards, но имеет дополнительные преимущества — она короче и менее подвержена ошибкам.

Однако, `#pragma once` не является официальной частью языка C++, и не все компиляторы её поддерживают (хотя большинство современных компиляторов поддерживают).

Я же рекомендую использовать header guards, чтобы сохранить максимальную совместимость вашего кода.

Тест

Добавьте header guards к следующему заголовочному файлу:

add.h:

```
1. int add(int x, int y);
```

Урок №27. Конфликт имен и std namespace

Допустим, что вам нужно съездить к дальним родственникам в другой город. У вас есть только их адрес: *г. Ржев, ул. Вербовая, 13*. Попав в город Ржев, вы открываете Google Карты/Яндекс.Карты и видите, что есть две улицы с названием *Вербовая*, еще и в противоположных концах города! Какая из них нужна вам? Если у вас нет никакой дополнительной информации (например, вы знаете, что их дом находится возле аптеки или школы), вам придется позвонить им и спросить. Чтобы подобной путаницы не возникало, все названия улиц в городе должны быть уникальными.

Аналогично и в языке C++ все идентификаторы (имена переменных/функций/классов и т.д.) должны быть уникальными. Если в вашей программе находятся два одинаковых идентификатора, то будьте уверены, что ваша программа не скомпилируется: вы получите ошибку **конфликта имен**.

Пример конфликта имен:

a.cpp:

```
#include <iostream>

void doSomething(int x)
{
    std::cout << x;
}
```

b.cpp:

```
#include <iostream>

void doSomething(int x)
{
    std::cout << x * 2;
}
```

main.cpp:

```
void doSomething(int x); // предварительное объявление функции doSomething()

int main()
{
    doSomething(5);

    return 0;
}
```

По отдельности файлы a.cpp, b.cpp и main.cpp скомпилируются. Однако, если a.cpp и b.cpp разместить в одном проекте — произойдет конфликт имен, так как определение функции doSomething() находится сразу в обоих файлах.

Большинство конфликтов имен происходят в двух случаях:

- Файлы, добавленные в один проект, имеют функцию (или глобальную переменную) с одинаковыми именами (ошибка на этапе линкинга).
- Файл .cpp подключает заголовочный файл, в котором идентификатор конфликтует с идентификатором из файла .cpp (ошибка на этапе компиляции).

Как только программы становятся больше, то и идентификаторов используется больше. Следовательно, вероятность возникновения конфликта имен значительно возрастает. Хорошая новость заключается в том, что язык C++ предоставляет достаточно механизмов для предотвращения возникновения конфликтов имен (например, локальная область видимости или пространства имен).

Пространство имен

В первых версиях языка C++ все идентификаторы из Стандартной библиотеки C++ (такие как cin/cout и т.д.) можно было использовать напрямую. Тем не менее, это означало, что любой идентификатор из Стандартной библиотеки C++ потенциально мог конфликтовать с именем, которое вы выбрали для ваших собственных идентификаторов. Код, который работал, мог внезапно получить конфликт имен при подключении нового заголовочного файла из Стандартной библиотеки C++. Или, что еще хуже, код, написанный по стандартам одной версии языка C++, мог уже не работать в новой версии языка C++. Чтобы устранить данную проблему, весь функционал Стандартной библиотеки C++ перенесли в специальную область - **пространство имен** (англ. ***"namespace"***).

Аналогично тому, как город гарантирует, что все улицы в его пределах имеют уникальные названия, так и пространство имен гарантирует, что все его идентификаторы — уникальны.

Таким образом, `std::cout` состоит из двух частей: идентификатор `cout` и пространство имен `std`. Весь функционал Стандартной библиотеки C++ определен внутри пространства имен `std` (сокр. от *«**st**and**ard**»*).

Мы еще поговорим о пространствах имен на следующих уроках, а также рассмотрим создание своего собственного пространства имен. Сейчас, главное, что

вам нужно запомнить, — это то, что всякий раз, когда вы используете идентификаторы из Стандартной библиотеки C++ (например, cout), вы должны сообщать компилятору, что этот идентификатор находится внутри пространства имен std.

Правило: При использовании идентификаторов из пространства имен - указывайте используемое пространство имен.

Оператор разрешения области видимости ::

Самый простой способ сообщить компилятору, что определенный идентификатор находится в определенном пространстве имен — использовать **оператор разрешения области видимости** `::`. Например:

```
1. std::cout << "Hello, world!";
```

Здесь мы сообщаем компилятору, что хотим использовать объект cout из пространства имен `std`.

Урок №28. Разработка ваших первых программ

При написании программ, у вас, как правило, есть какая-то проблема, которую нужно решить. Новички очень часто спотыкаются на этапе преобразования идеи решения проблемы в реальный код. Но самое главное, что вам нужно запомнить — разработка программы выполняется перед этапом написания её кода.

Во многих отношениях, программирование - это как архитектура. Что произойдет, если вы попытаетесь построить дом без соблюдения архитектурного плана? Дом может вы и сумеете построить, но какой он будет: кривые стены, протекающая крыша и т.д. Аналогично, если вы начнете программировать что-нибудь серьезное перед тем, как составите план, то очень скоро обнаружите, что ваш код имеет очень много проблем, на решения которых вы потратите гораздо больше времени/усилий/нервов, нежели на изначальное составление хорошего плана.

Шаг №1: Определите проблему

Первое, что вам нужно сделать — это определить проблему, которую решит ваша программа. В идеале, вы должны сформулировать это одним или двумя предложениями. Например:

- Я хочу написать программу-справочник для удобного хранения и редактирования всех телефонных номеров и звонков.
- Я хочу написать программу для генерации случайных чисел, с помощью которой можно будет определять победителей разных конкурсов.
- Я хочу написать программу, которая будет отслеживать и анализировать акции на фондовом рынке для генерации выигрышных прогнозов.

Хотя этот шаг кажется очевидным, но он также очень важен. Самое худшее, что вы можете сделать - это написать программу, которая делает не то, что вам нужно!

Шаг №2: Определите свой инструментарий, цели и план бэкапа

Для опытных программистов на этом этапе будет еще немало дополнительных пунктов:

- Какая ваша целевая аудитория и какие у нее потребности?
- На какой архитектуре/ОС ваша программа будет работать?
- Какой инструментарий вы будете использовать?
- Будете ли вы разрабатывать программу в одиночку или в составе команды?

- Анализ требований.
- Определение стратегий тестирования/обратной связи/релиза.
- Создание плана бэкапа в случае неожиданных проблем.

Новички, как правило, большим количеством вопросов не задаются: "Пишу программу для собственного использования, в одиночку, на своей операционной системе, с помощью своей IDE, пользоваться этой программой буду только я". Всё просто.

Если же вы будете работать над чем-нибудь посерьезнее, то стоит еще задуматься о плане бэкапа вашей программы/проекта. Это не просто скопировать код в другую папку ноутбука (хотя это уже лучше, чем ничего). Если ваша ОС "сломается", то вы потеряете все данные. Хорошая стратегия резервного копирования включает в себя создание копии вашего кода вне вашей операционной системы, например:

- Отправить самому себе E-mail с кодом (прикрепить как файл).
- Скопировать в Dropbox или в любое другое облако.
- Перенести на внешнее запоминающее устройство (например, на портативный жёсткий диск).
- Скопировать на другой компьютер в локальной сети.
- Воспользоваться системами контроля версий (например, **GitHub**, **GitLab** или **Bitbucket**).

Шаг №3: Разбейте проблему на части

В реальной жизни нам часто приходится выполнять очень сложные задачи. Понять, как их решить, также бывает очень трудно. В таких случаях можно использовать **метод деления на части** (или ***метод "от большого к малому"***). То есть, вместо того, чтобы решать одну большую сложную задачу, мы разбиваем её на несколько подзадач, каждую из которых проще решить. Если эти подзадачи все еще слишком сложные, то их также нужно еще раз разбить. И так до тех пор, пока мы не доберемся до точки, где каждая отдельно взятая задача — легко решаема.

Рассмотрим это на примере. Допустим, что нам нужно написать доклад о картошке. Наша иерархия задач в настоящее время выглядит следующим образом:

- Написать доклад о картошке

Это довольно большая задача, давайте разделим её на подзадачи:

- Написать доклад о картошке
    - Поиск информации о картошке
    - Создание плана
    - Заполнение каждого пункта плана подробной информацией
    - Заключение

Это уже проще, так как теперь мы имеем список подзадач, на которых можем сосредоточиться в индивидуальном порядке. Тем не менее, в данном случае, "Поиск информации о картошке" звучит немного расплывчато, нужно дополнительно разбить и этот пункт:

- Написать доклад о картошке
    - Поиск информации о картошке
        - Сходить в библиотеку за книжками о картошке
        - Поискать информацию в Интернете
        - Делать заметки в соответствующих разделах из справочного материала
    - Создание плана
        - Информация о выращивании
        - Информация об обработке
        - Информация об удобрениях
    - Заполнение каждого пункта плана подробной информацией
    - Заключение

Выполняя каждый подпункт этого задания, мы решим одну большую задачу.

Есть еще один метод создания иерархии - **метод "от малого к большому"**. Рассмотрим на примере.

Большинство из нас вынуждены ходить на работу (школу/университет) по будням. Предположим, что нам нужно решить проблему "от постели к работе". Если бы вас спросили, что вы делаете перед тем, как добраться на работу, вы бы ответили примерно следующее:

- Выбрать одежду
- Одеться
- Позавтракать
- Ехать на работу
- Почистить зубы

- Встать с постели
- Приготовить завтрак
- Принять душ

Используя метод "от малого к большому", мы можем сгруппировать задания и создать иерархию:

- От постели к работе
    - Спальня
        - Встать с постели
        - Выбрать одежду
        - Одеться
    - Ванная
        - Принять душ
        - Почистить зубы
    - Завтрак
        - Сделать завтрак
        - Позавтракать
    - Транспорт
        - Ехать на работу

Использование подобных иерархий чрезвычайно полезно в программировании для определения структуры всей программы. Задача верхнего уровня (например, "Написать доклад о картошке" или "От постели к работе") становится функцией main() (так как это основная проблема, которую нужно решить). Подзадачи становятся функциями в программе.

Шаг №4: Определите последовательность событий

Теперь, когда ваша программа имеет структуру, пришло время ответить на вопрос: "А как же связать все эти пункты вместе?". Первый шаг заключается в определении последовательности событий. Например, когда вы просыпаетесь утром, в какой последовательности вы выполняете вышеперечисленные дела? Скорее всего, в следующей:

- Встать с постели
- Выбрать одежду
- Принять душ
- Одеться
- Приготовить завтрак

- Позавтракать
- Почистить зубы
- Ехать на работу

Если бы мы создавали калькулятор, то последовательность заданий выглядела бы следующим образом:

- Получить первое значение от пользователя
- Получить математическую операцию от пользователя
- Получить второе значение от пользователя
- Вычислить результат
- Вывести результат

Этот список определяет содержимое функции main():

```
1. int main()
2. {
3.     getOutOfBed();
4.     pickOutClothes();
5.     takeAShower();
6.     getDressed();
7.     prepareBreakfast();
8.     eatBreakfast();
9.     brushTeeth();
10.    driveToWork();
11.}
```

Или, в случае с калькулятором:

```
1. int main()
2. {
3.     // Получить первое значение от пользователя
4.     getUserInput();
5.
6.     // Получить математическую операцию от пользователя
7.     getMathematicalOperation();
8.
9.     // Получить второе значение от пользователя
10.    getUserInput();
11.
12.    // Вычислить результат
13.    calculateResult();
14.
15.    // Вывести результат
16.    printResult();
17.}
```

Шаг №5: Определите данные ввода и данные вывода на каждом этапе

После определения иерархии задач и последовательности событий, нам нужно определить, какими будут данные ввода и данные вывода на каждом этапе.

Например, первая функция из вышеприведенного примера - getUserInput(), довольно проста. Мы собираемся получить число от пользователя и вернуть его обратно в caller. Таким образом, прототип функции будет выглядеть следующим образом:

```
1. int getUserInput();
```

В примере с калькулятором, функции calculateResult() требуется 3 ввода данных: два числа и 1 математический оператор. При вызове calculateResult() у нас уже должны быть 3 фрагмента данных, которые мы будем использовать в качестве параметров функции. Функция calculateResult() вычисляет значение результата, но не выводит его. Следовательно, нам необходимо вернуть этот результат в качестве возвращаемого значения обратно в caller, чтобы другие функции также имели возможность его использовать.

Учитывая это, прототип функции становится следующим:

```
1. int calculateResult(int input1, int op, int input2);
```

Шаг №6: Позаботьтесь о деталях

На этом этапе нам нужно реализовать каждый подпункт из нашей иерархии задач. Если вы разбили задание на достаточно мелкие кусочки, то выполнить каждый из этих кусочков будет несложно. Например:

```
1. int getMathematicalOperation()
2. {
3.     std::cout << "Please enter which operator you want (1 = +, 2 = -
   , 3 = *, 4 = /): ";
4. 
5.     int op;
6.     std::cin >> op;
7. 
8.     // А что, если пользователь введет некорректный символ?
9.     // Пока что мы это проигнорируем
10. 
11.    return op;
12. }
```

Шаг №7: Соедините данные ввода с данными вывода в программе

И, наконец, последний шаг - это соединение данных ввода с данными вывода. Например, вы можете отправить выходные данные функции calculateResult() во входные данные функции printResult(), чтобы вторая функция могла вывести результат. Чаще всего в таких случаях используются промежуточные переменные для временного хранения результата и его перемещения между функциями.

Например:

```
// Переменная result - это промежуточная переменная, которая используется для передачи выходного значения функции calculateResult() во входные данные функции printResult()
int result = calculateResult(input1, op, input2);
printResult(result);
```

Согласитесь, что вариант, приведенный выше, читабельнее варианта без использования временных переменных (см. ниже):

```
printResult(calculateResult(input1, op, input2));
```

Рассмотрим готовую версию программы-калькулятора, её структуру и перемещение данных:

```
#include <iostream>

int getUserInput()
{
    std::cout << "Please enter an integer: ";
    int value;
    std::cin >> value;
    return value;
}

int getMathematicalOperation()
{
    std::cout << "Please enter which operator you want (1 = +, 2 = -, 3 = *, 4 = /): ";

    int op;
    std::cin >> op;

    // А что, если пользователь введет некорректный символ?
    // Пока что мы это проигнорируем

    return op;
}

int calculateResult(int x, int op, int y)
{
    // Обратите внимание, оператор == используется для сравнения двух значений

    if (op == 1) // если пользователь выбрал операцию сложения (№1)
        return x + y; // то выполняем эту строку
    if (op == 2) // если пользователь выбрал операцию вычитания (№2)
        return x - y; // то выполняем эту строку
    if (op == 3) // если пользователь выбрал операцию умножения (№3)
        return x * y; // то выполняем эту строку
    if (op == 4) // если пользователь выбрал операцию деления (№4)
        return x / y; // то выполняем эту строку

    return -1; // вариант, если пользователь ввел некорректный символ
}

void printResult(int result)
```

```
{
    std::cout << "Your result is: " << result << std::endl;
}

int main()
{
    // Получаем первое значение от пользователя
    int input1 = getUserInput();

    // Получаем математическую операцию от пользователя
    int op = getMathematicalOperation();

    // Получаем второе значение от пользователя
    int input2 = getUserInput();

    // Вычисляем результат и сохраняем его во временной переменной
    int result = calculateResult(input1, op, input2 );

    // Выводим результат
    printResult(result);
}
```

Здесь есть несколько концепций, которые мы еще не рассматривали: условное ветвление if и использование оператора равенства == для сравнения двух элементов. Не беспокойтесь, если вы это не понимаете - мы всё это детально рассмотрим на следующих уроках.

Советы

Пускай ваши первые программы будут простыми. Очень часто новички ставят слишком высокие планки для своих первых более-менее серьезных программ. Например, "Я хочу написать игру с графикой, звуком, монстрами, подземельями и городом, в котором можно будет продавать найденные вещи". Если вы попытаетесь написать что-нибудь подобное в начале вашего пути как программиста, то очень скоро у вас пропадет любое желание программировать. Вместо этого, пускай ваши первые цели/задания/программы будут попроще, например, "Я хочу написать программу, которая отображала бы 2D-поле на экране".

Добавляйте новый функционал со временем. Как только вы написали простенькую программу, которая работает (даже без сбоев), то следующим вашим шагом должно стать добавление нового функционала. Например, когда вы сможете отображать 2D-поле на экране - добавьте персонажа, который сможет ходить по этому полю. После того, как вы уже сможете ходить — добавьте стены, которые будут препятствовать вашему движению. После того, как у вас будут стены — постройте из них город. После того, как у вас будет город — добавьте персонажей-продавцов. При таком подходе на вас не наваливается всё сразу и вы знаете с чего начинать, что делать дальше, в какой последовательности и т.д.

Фокусируйтесь только на одном задании в определенный промежуток времени. Не пытайтесь сделать всё и сразу, не распыляйтесь на несколько задач одновременно. Сосредоточьтесь на одном. Лучше иметь одно выполненное задание и пять невыполненных, нежели шесть частично выполненных заданий. Если вы рассеиваете свое внимание в нескольких направлениях, то и ошибок будет много.

Тестируйте каждую новую часть кода. Начинающие программисты часто пишут программу за один присест. Затем, при компиляции проекта, получают сотни ошибок. Поэтому, после написания определенной части кода - сразу компилируйте и тестируйте её. Если ваш код не будет работать, то вы будете знать, где находится проблема, и исправить её будет намного легче. Как только вы убедились, что ваш код рабочий - переходите к написанию следующей части, а затем *repeat*. Тестирование может занять больше времени, но в конечном итоге ваш код будет работать так, как вам нужно.

Большинство новичков пропустят некоторые из этих шагов и советов, так как это не столь захватывающе, как, собственно, сам процесс кодинга. Хорошая новость заключается в том, что как только вы освоите все эти концепции - они станут для вас естественными в процессе разработки ваших программных продуктов.

Урок №29. Отладка программ: степпинг и точки останова

Как ни странно, программирование может быть сложным и ошибок может быть очень много. Ошибки, как правило, попадают в одну из двух категорий: синтаксические или семантические/смысловые.

Типы ошибок

Синтаксическая ошибка возникает, когда вы пишете код, который не соответствует правилам грамматики языка C++. Например, пропущенные точки с запятой, необъявленные переменные, непарные круглые или фигурные скобки и т.д. В следующей программе есть несколько синтаксических ошибок:

```
1. #include <iostream>; // директивы препроцессора не заканчиваются точкой с
   запятой
2.
3. int main()
4. {
5.     std:cout < "Hi there; << x; // недействительный оператор (:), незаконченное
    предложение (пропущено ") и необъявленная переменная
6.     return 0 // пропущена точка с запятой в конце стейтмента
7. }
```

К счастью, компилятор ловит подобные ошибки и сообщает о них в виде предупреждений или ошибок.

Семантическая ошибка возникает, когда код является синтаксически правильным, но делает не то, что задумал программист.

Иногда это может привести к сбою в программе, например, если делить на ноль:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int a = 10;
6.     int b = 0;
7.     std::cout << a << " / " << b << " = " << a / b; // делить на 0 нельзя
8.     return 0;
9. }
```

Иногда это может привести к неверным результатам:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Hello, word!"; // орфографическая ошибка
6.     return 0;
7. }
```

Либо делать вообще не то, что нужно:

```
#include <iostream>

int add(int x, int y)
{
    return x - y; // функция должна выполнять сложение, но выполняет вычитание

}

int main()
{
    std::cout << add(5, 3); // должно быть 8, но результат - 2
    return 0;
}
```

К сожалению, компилятор не ловит подобные ошибки, так как он проверяет только то, что вы написали, а не то, что вы хотели этим сделать.

В примерах, приведенных выше, ошибки довольно легко обнаружить. Но в большинстве программ (в которых больше 40 строк кода), семантические ошибки увидеть с помощью простого просмотра кода будет не так-то и легко.

И здесь нам на помощь приходит отладчик.

Отладчик

Отладчик (или ***"дебаггер"***, от англ. ***"debugger"***) - это компьютерная программа, которая позволяет программисту контролировать выполнение кода. Например, программист может использовать отладчик для выполнения программы пошагово, последовательно изучая значения переменных в программе.

Более ранние дебаггеры, такие как **GDB**, имели интерфейс командной строки, где программисту приходилось вводить специальные команды для старта работы. Более современные дебаггеры уже имеют «графический» интерфейс, что значительно упрощает работу с ними. Сейчас почти все современные IDE имеют встроенные отладчики. То есть, вы можете использовать одну среду разработки как для написания кода, так и для его отладки (вместо постоянного переключения между разными программами).

Базовый функционал у всех отладчиков один и тот же. Отличаются они, как правило, тем, как этот функционал и доступ к нему организованы, горячими клавишами и дополнительными возможностями.

> ***Примечание:*** Перед тем как продолжить, убедитесь, что вы находитесь в режиме конфигурации «Debug». Все скриншоты данного урока выполнены в Visual Studio 2019.

Степпинг

Степпинг (англ. ***"stepping"***) - это возможность отладчика выполнять код пошагово (строка за строкой). Есть три команды степпинга:

- Команда "Шаг с заходом"
- Команда "Шаг с обходом"
- Команда "Шаг с выходом"

Мы сейчас рассмотрим каждую из этих команд в индивидуальном порядке.

Команда "Шаг с заходом"

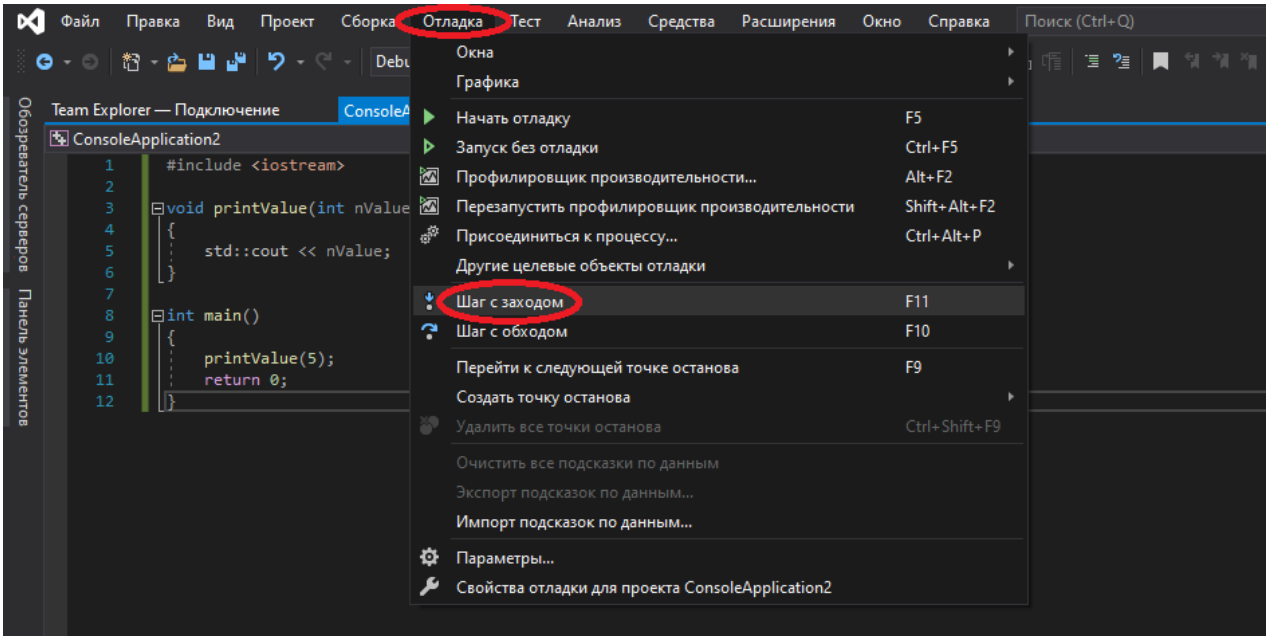
Команда **"Шаг с заходом"** (англ. ***"Step into"***) выполняет следующую строку кода. Если этой строкой является вызов функции, то "Шаг с заходом" открывает функцию и выполнение переносится в начало этой функции.

Давайте рассмотрим очень простую программу:

```
1. #include <iostream>
2.
3. void printValue(int nValue)
4. {
5.     std::cout << nValue;
6. }
7.
8. int main()
9. {
10.    printValue(5);
11.    return 0;
12. }
```

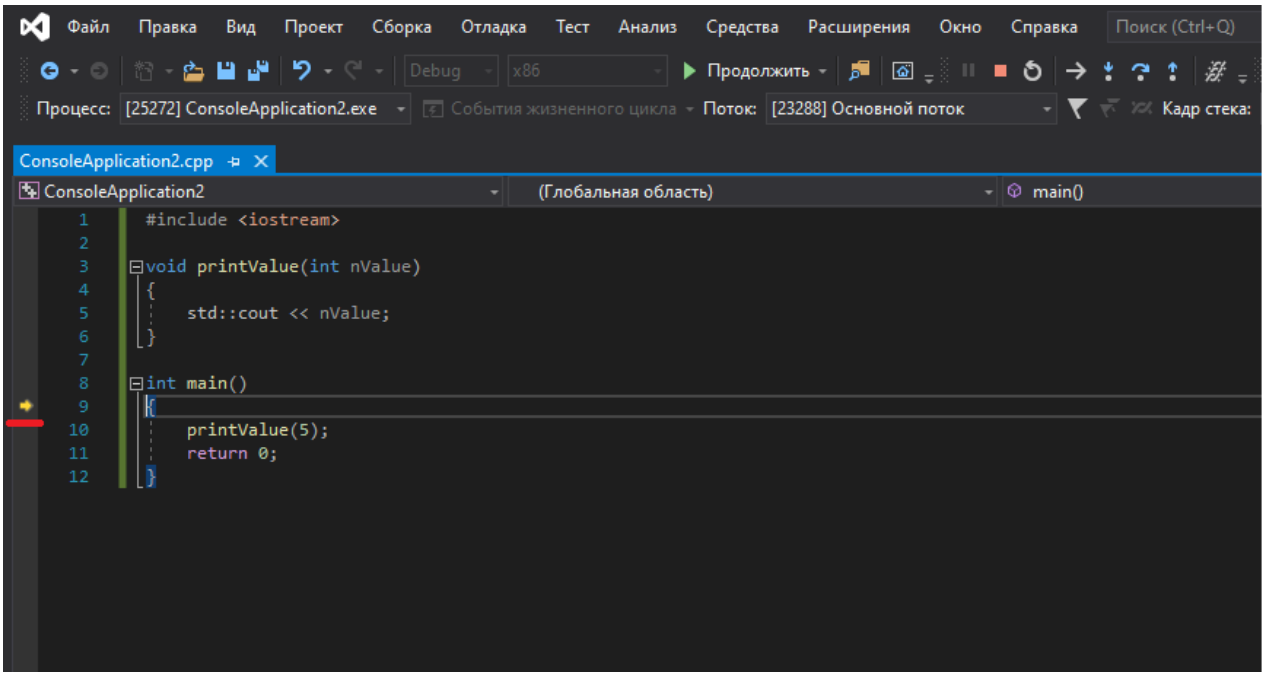
Как вы уже знаете, при запуске программы выполнение начинается с вызова главной функции main(). Так как мы хотим выполнить отладку внутри функции main(), то давайте начнем с использования команды "Шаг с заходом".

В Visual Studio, перейдите в меню `"Отладка" > "Шаг с заходом"` (либо нажмите F11):

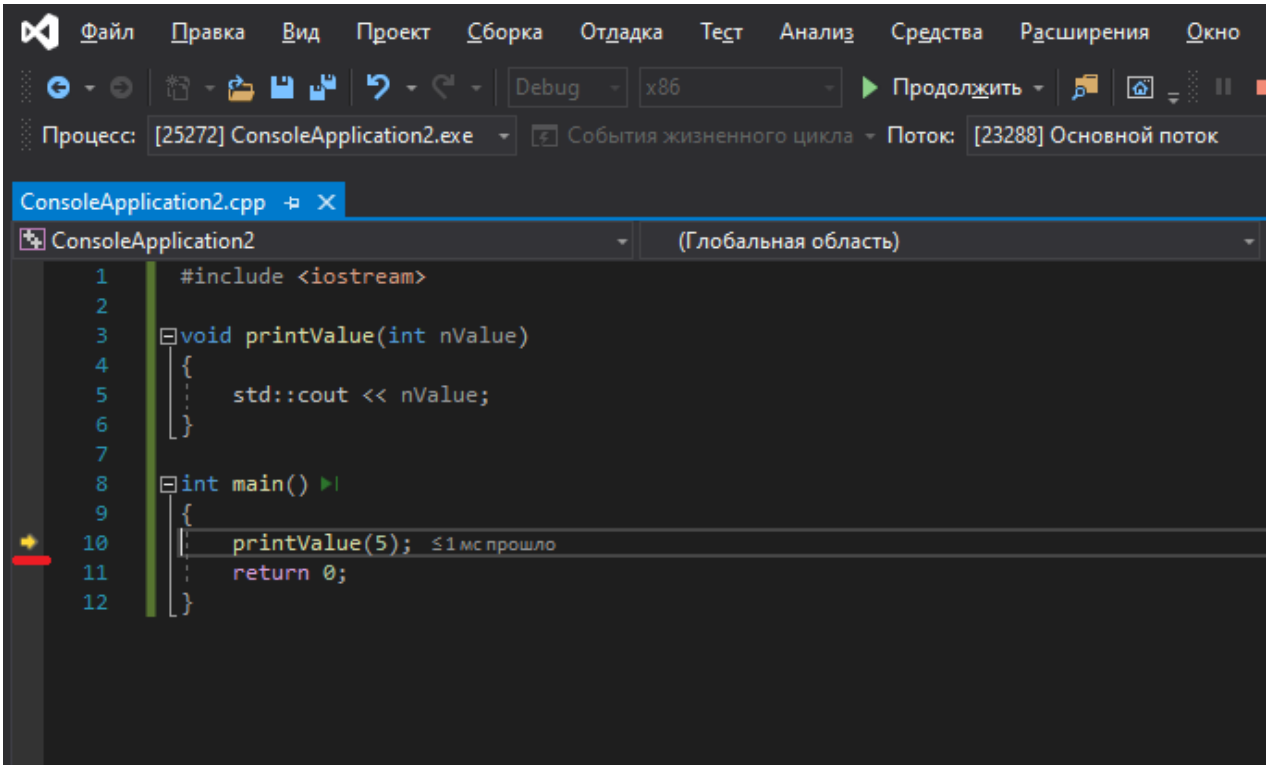


Если вы используете другую IDE, то найдите в меню команду `"Step Into/Шаг с заходом"` и выберите её.

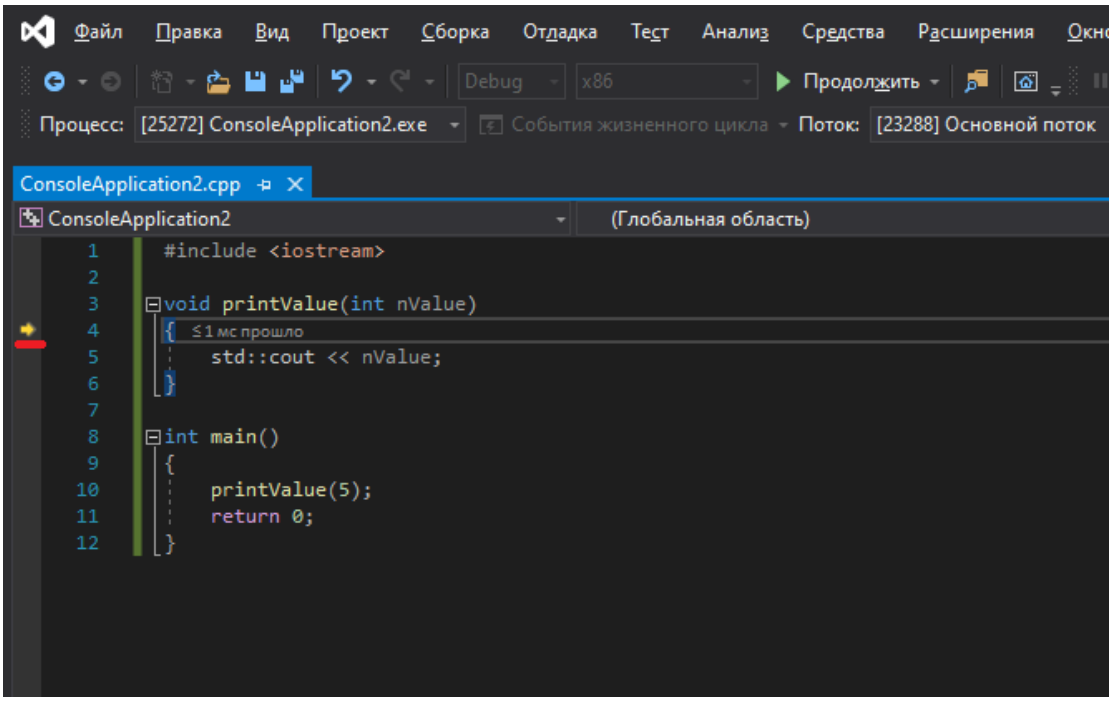
Когда вы это сделаете, должны произойти две вещи. Во-первых, так как наше приложение является консольной программой, то должно открыться консольное окно. Оно будет пустым, так как мы еще ничего не выводили. Во-вторых, вы должны увидеть специальный маркер слева возле открывающей скобки функции main(). В Visual Studio этим маркером является жёлтая стрелочка (если вы используете другую IDE, то должно появиться что-нибудь похожее):



Стрелка-маркер указывает на следующую строку, которая будет выполняться. В этом случае отладчик говорит нам, что следующей строкой, которая будет выполняться, — будет открывающая фигурная скобка функции main(). Выберите "Шаг с заходом" еще раз - стрелка переместится на следующую строку:



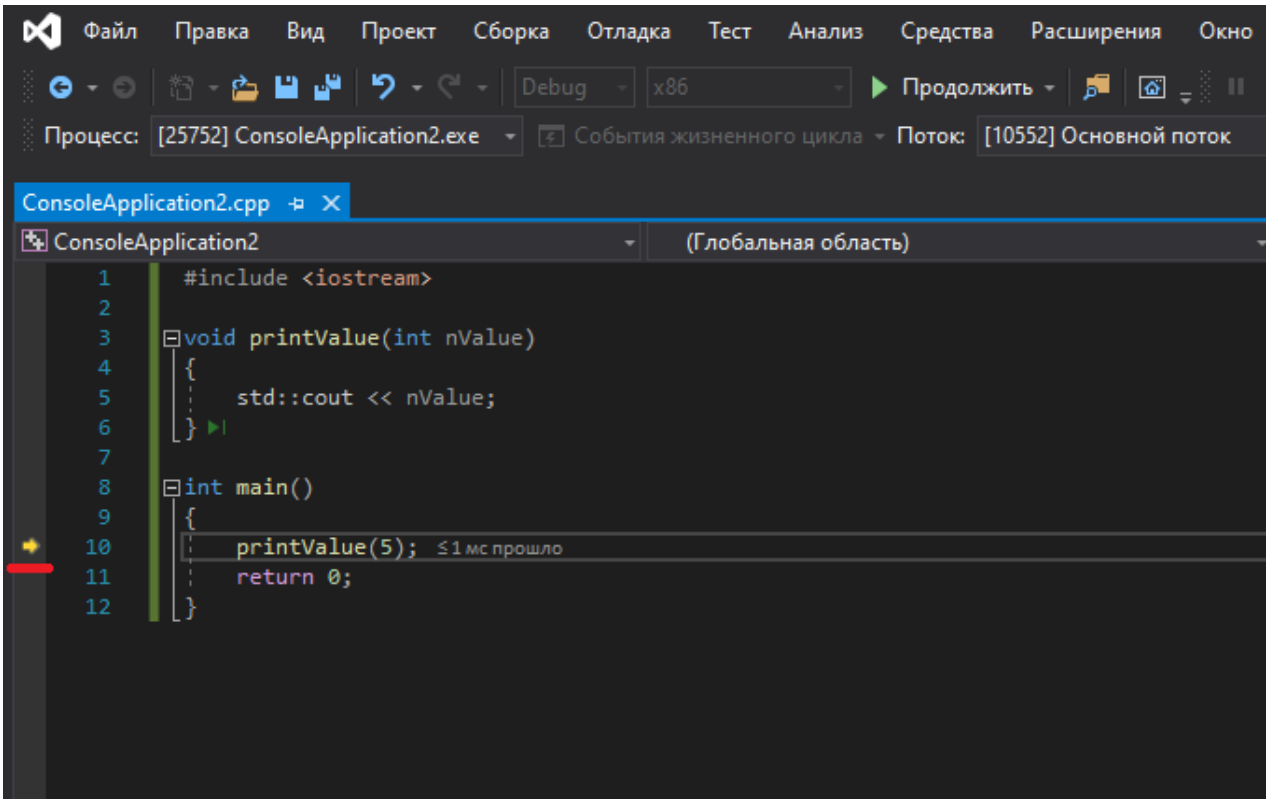
Это значит, что следующей строкой, которая будет выполняться, — будет вызов функции printValue(). Выберите "Шаг с заходом" еще раз. Поскольку printValue() - это вызов функции, то мы переместимся в начало функции printValue():



Выберите еще раз "Шаг с заходом" для выполнения открывающей фигурной скобки printValue(). Стрелка будет указывать на `std::cout << nValue;`.

Теперь выберите "Шаг с обходом" (F10). Вы увидите число `5` в консольном окне.

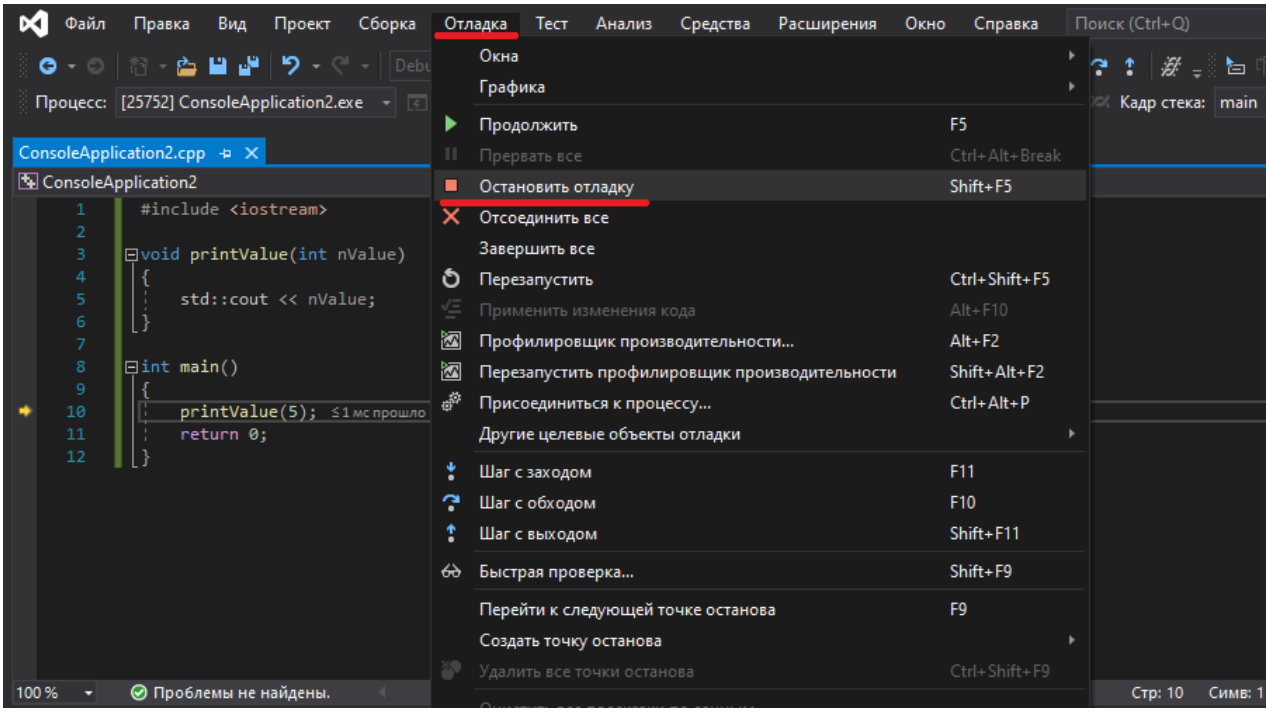
Выберите "Шаг с заходом" еще раз для выполнения закрывающей фигурной скобки printValue(). Функция printValue() завершит свое выполнение и стрелка переместиться в функцию main(). Обратите внимание, в main() стрелка снова будет указывать на вызов printValue():



Может показаться, будто отладчик намеревается еще раз повторить цикл с функцией printValue(), но в действительности он нам просто сообщает, что он только что вернулся из этой функции.

Выберите "Шаг с заходом" два раза. Готово, все строки кода выполнены. Некоторые дебаггеры автоматически прекращают сеанс отладки в этой точке.

Но Visual Studio так не делает, так что если вы используете Visual Studio, то выберите `"Отладка" > "Остановить отладку"` (или Shift+F5):



Таким образом мы полностью остановили сеанс отладки нашей программы.

Команда "Шаг с обходом"

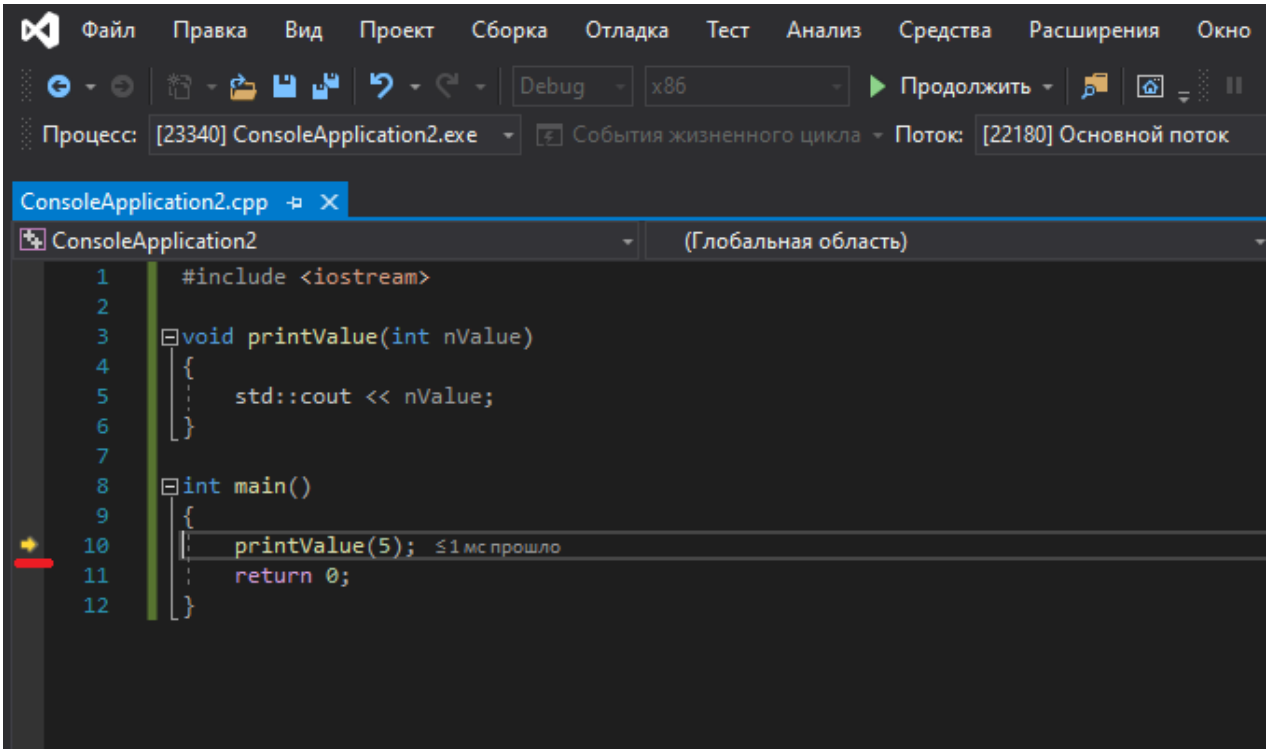
Как и команда "Шаг с заходом", команда **"Шаг с обходом"** (англ. ***"Step over"***) позволяет выполнить следующую строку кода. Только если этой строкой является вызов функции, то "Шаг с обходом" выполнит весь код функции в одно нажатие и возвратит нам контроль после того, как функция будет выполнена.

> ***Примечание для пользователей Code::Blocks***: Команда "Step over" называется "Next Line".

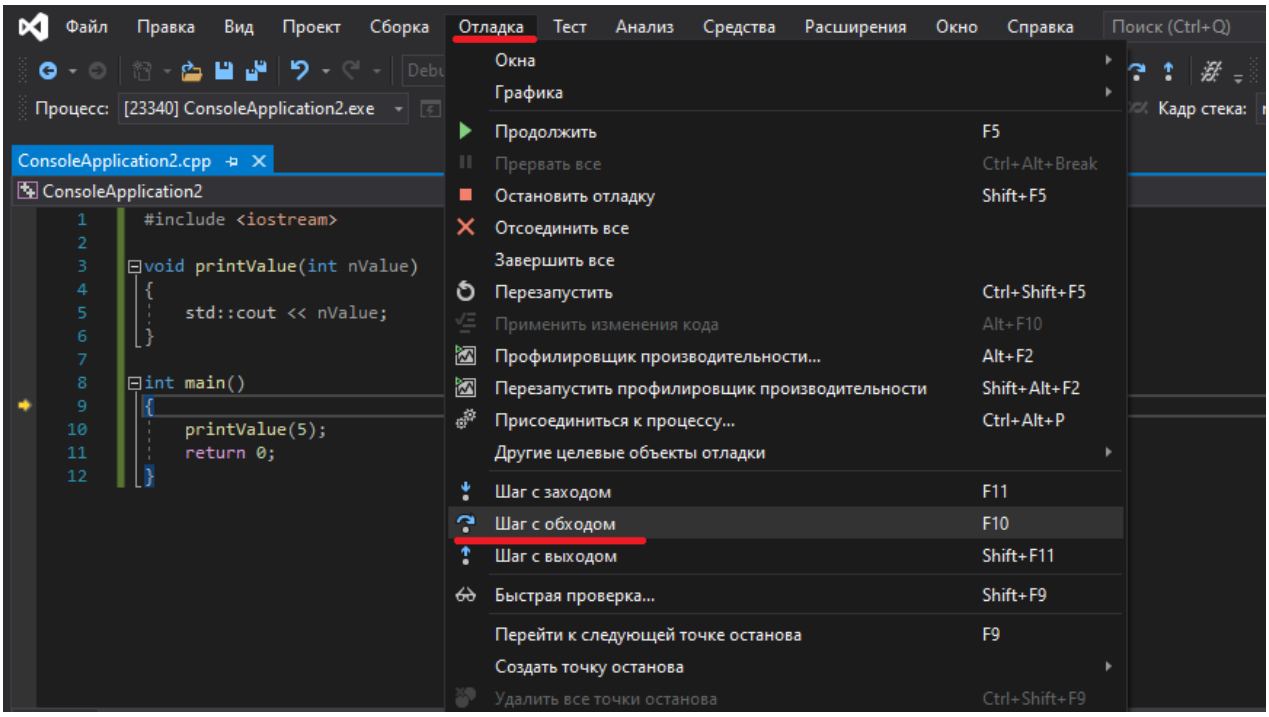
Рассмотрим пример, используя следующую программу:

```
1. #include <iostream>
2.
3. void printValue(int nValue)
4. {
5.     std::cout << nValue;
6. }
7.
8. int main()
9. {
10.     printValue(5);
11.     return 0;
12. }
```

Нажмите "Шаг с заходом", чтобы дойти до вызова функции printValue():



Теперь вместо команды "Шаг с заходом" выберите "Шаг с обходом" (или F10):



Отладчик выполнит функцию (которая выведет значение `5` в консоль), а затем возвратит нам управление на строке `return 0;`. И это всё за одно нажатие.

Команда "Шаг с обходом" позволяет быстро пропустить код функций, когда мы уверены, что они работают корректно и их не нужно отлаживать.

Команда "Шаг с выходом"

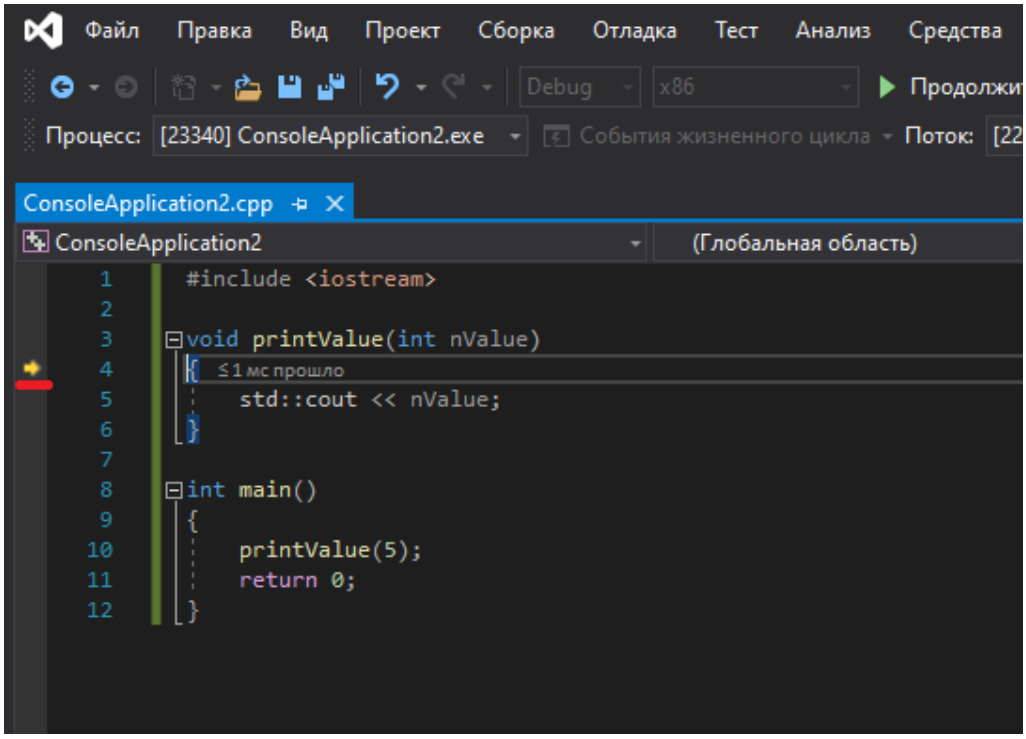
В отличие от двух предыдущих команд, команда **"Шаг с выходом"** (англ. ***"Step out"***) не просто выполняет следующую строку кода. Она выполняет весь оставшийся код функции, в которой вы сейчас находитесь, и возвращает контроль только после того, когда функция завершит свое выполнение. Проще говоря, «Шаг с выходом» позволяет выйти из функции.

Обратите внимание, команда "Шаг с выходом" появится в меню "Отладка" только после начала сеанса отладки (что делается путем использования одной из двух вышеприведенных команд).

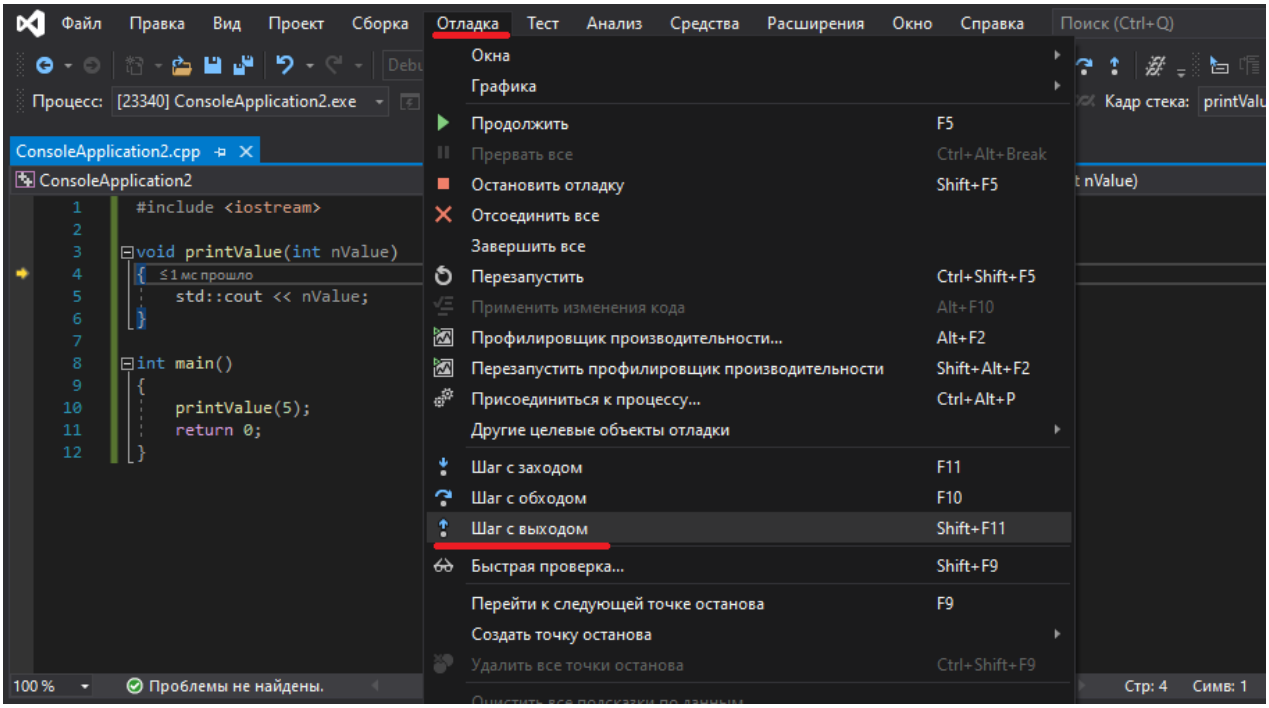
Рассмотрим все тот же пример:

```
1.  #include <iostream>
2.
3.  void printValue(int nValue)
4.  {
5.      std::cout << nValue;
6.  }
7.
8.  int main()
9.  {
10.     printValue(5);
11.     return 0;
12. }
```

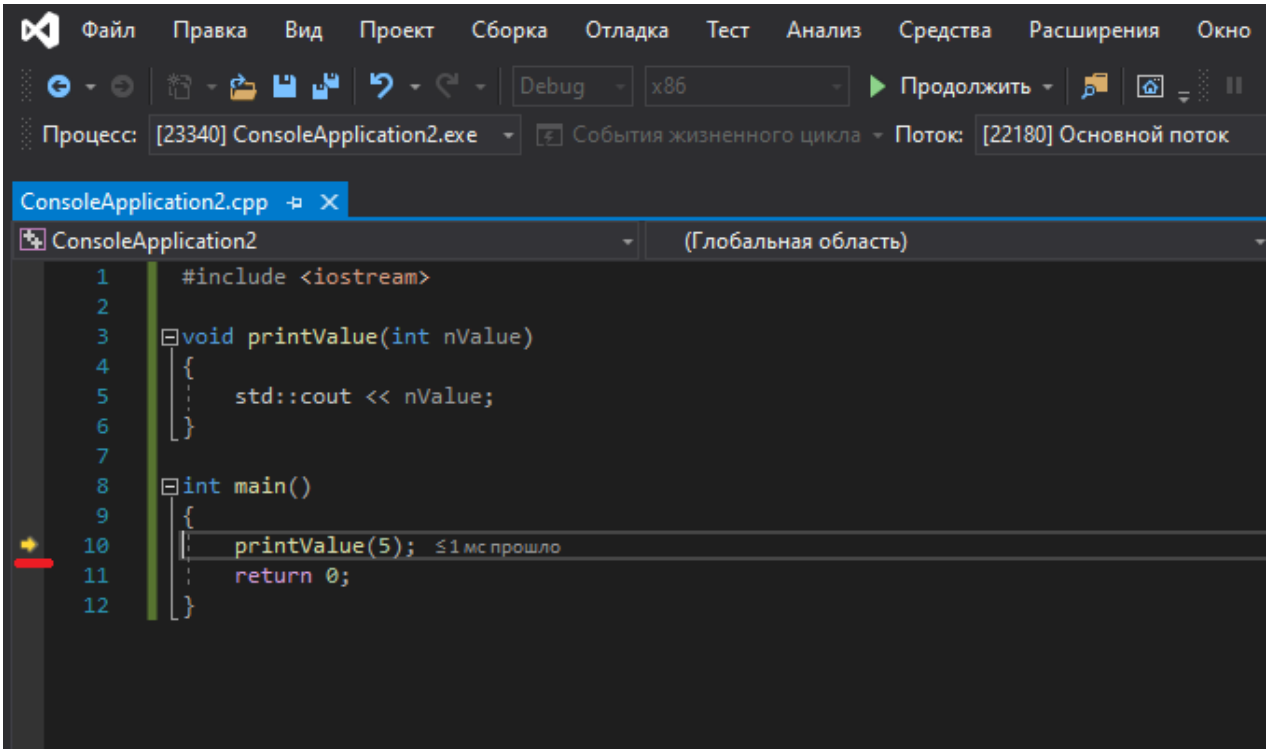
Нажимайте "Шаг с заходом" до тех пор, пока не перейдете к открывающей фигурной скобке функции printValue():



Затем выберите `"Отладка" > "Шаг с выходом"` (либо Shift+F11):



Вы заметите, что значение `5` отобразилось в консольном окне, а отладчик перешел к вызову функции printValue() в main():



Команда "Выполнить до текущей позиции"

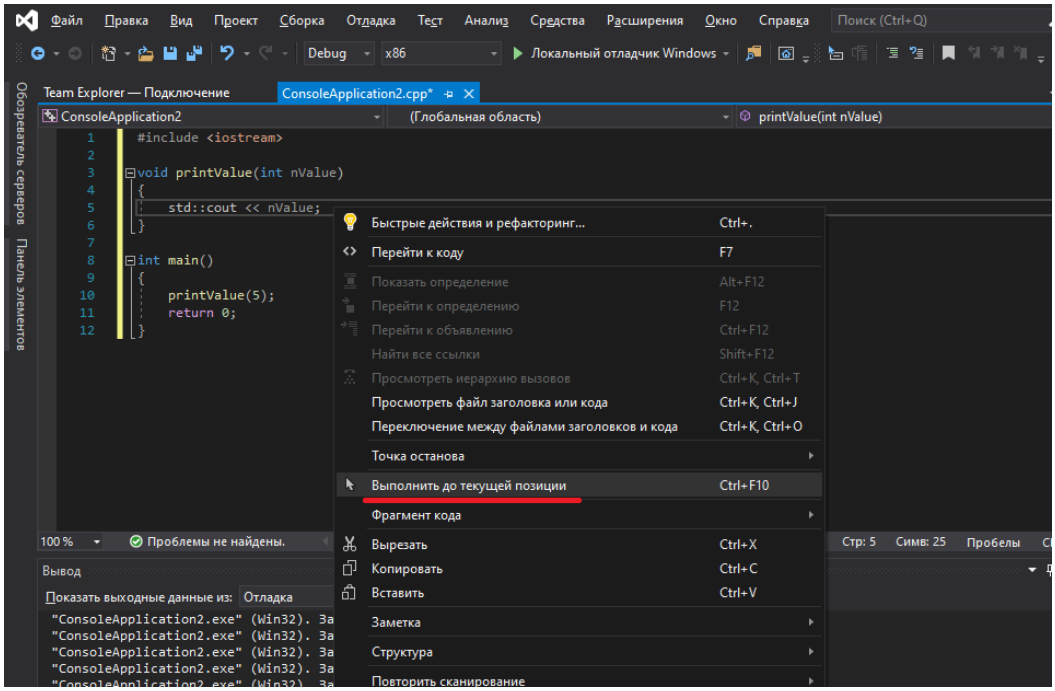
В то время как степпинг полезен для изучения каждой строки кода по отдельности, в большой программе перемещаться по коду с помощью этих команд будет не очень удобно.

Но и здесь современные отладчики предлагают еще несколько инструментов для эффективной отладки программ.

Команда "Выполнить до текущей позиции" позволяет в одно нажатие выполнить весь код до строки, обозначенной курсором. Затем контроль обратно возвращается к вам, и вы можете проводить отладку с указанной точки уже более детально. Давайте попробуем, используя уже знакомую нам программу:

```
1.  #include <iostream>
2.
3.  void printValue(int nValue)
4.  {
5.      std::cout << nValue;
6.  }
7.
8.  int main()
9.  {
10.     printValue(5);
11.     return 0;
12. }
```

Поместите курсор на строку `std::cout << nValue;` внутри функции printValue(), затем щелкните правой кнопкой мыши и выберите `"Выполнить до текущей позиции"` (либо Ctrl+F10):

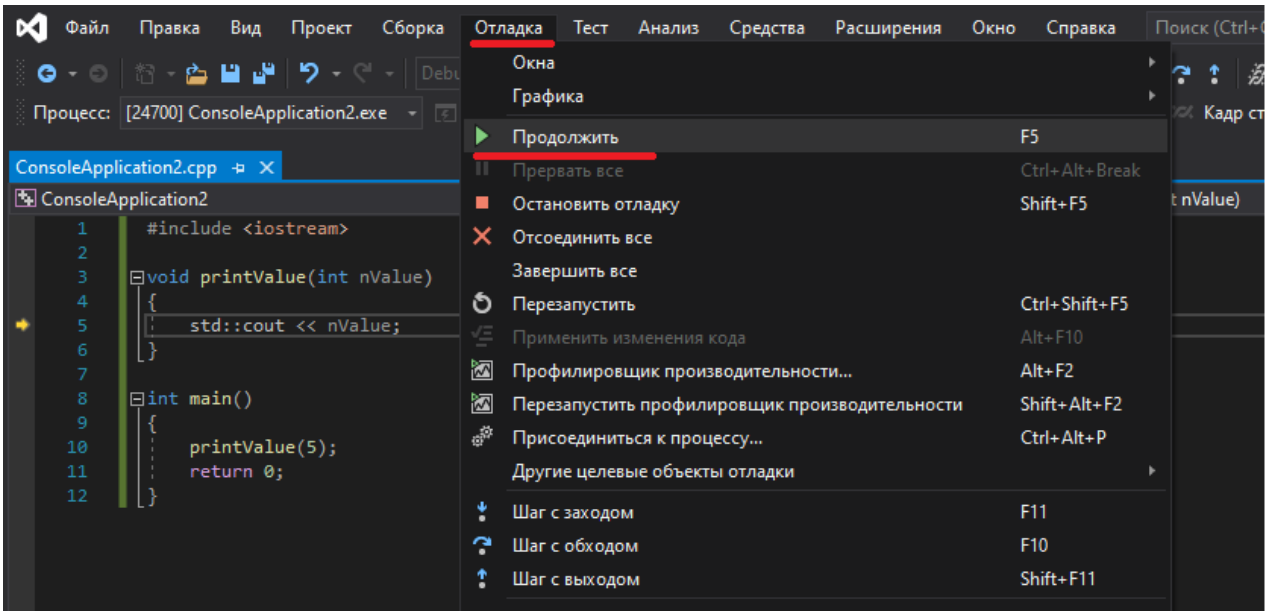


Вы заметите, что жёлтая стрелочка переместится на указанную вами строку. Выполнение программы остановится в этой точке, и программа будет ждать ваших дальнейших команд.

Команда "Продолжить"

Если вы находитесь в середине сеанса отладки вашей программы, то вы можете сообщить отладчику продолжать выполнение кода до тех пор, пока он не дойдет до конца программы (или до следующей контрольной точки). В Visual Studio эта команда называется **"Продолжить"** (англ. ***"Continue"***). В других дебаггерах она может иметь название "Run" или "Go".

Возвращаясь к вышеприведенному примеру, мы находимся как раз внутри функции printValue(). Выберите `"Отладка" > "Продолжить"` (или F5):

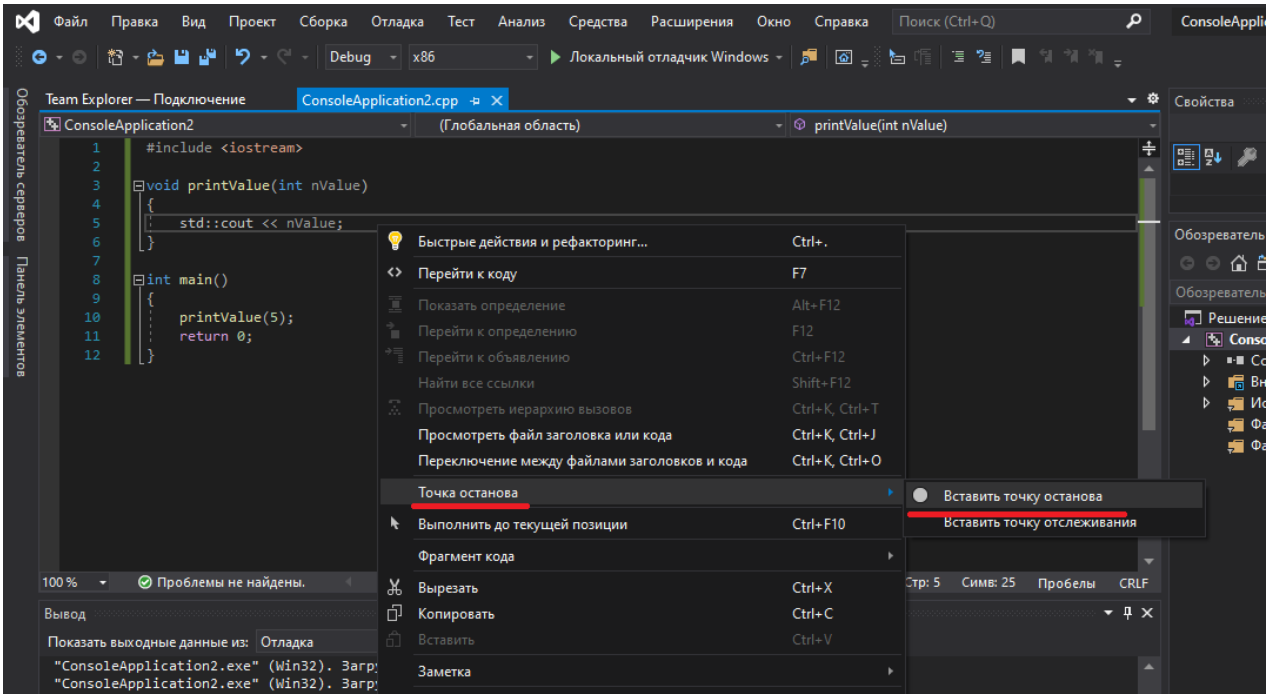


Программа завершит свое выполнение и выйдет из сеанса отладки.

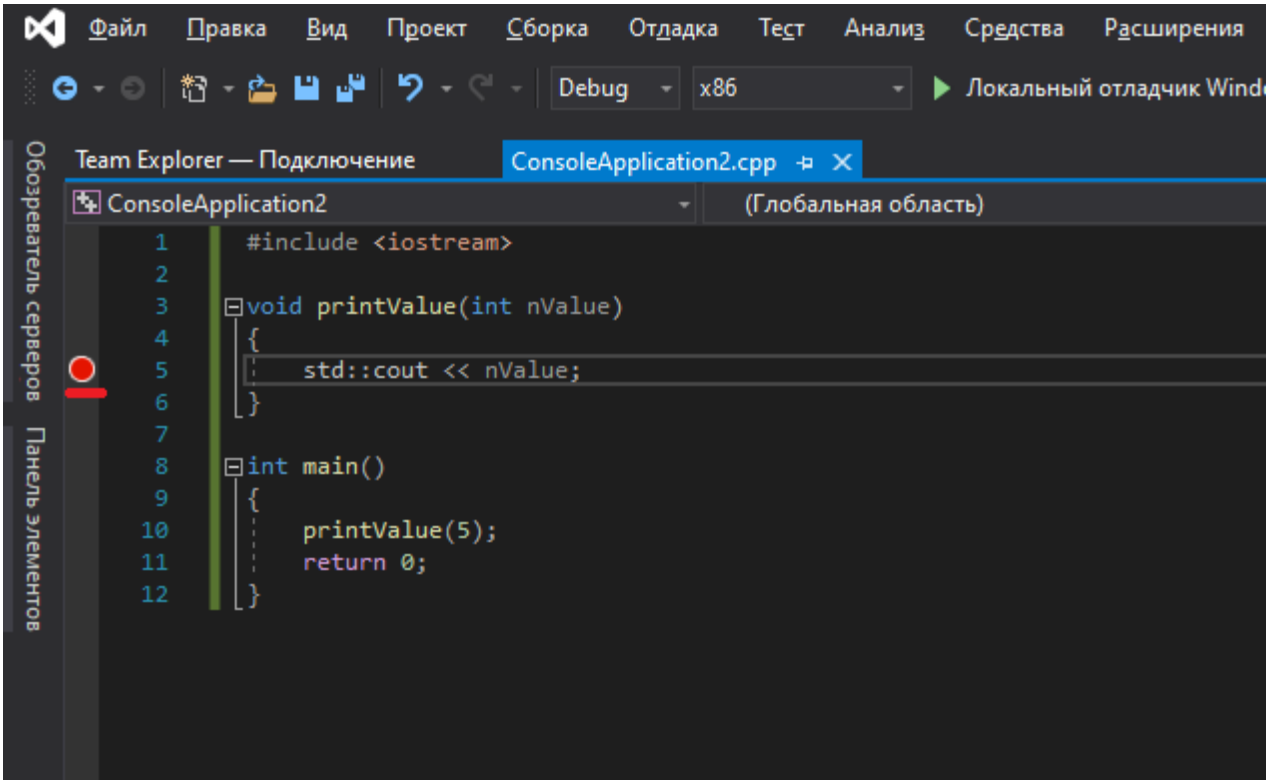
Точки останова

Точки останова (англ. ***"breakpoints"***) — это специальные маркеры, на которых отладчик останавливает процесс выполнения программы.

Чтобы задать точку останова в Visual Studio, щелкните правой кнопкой мыши по выбранной строке `> "Точка останова" > "Вставить точку останова"`:

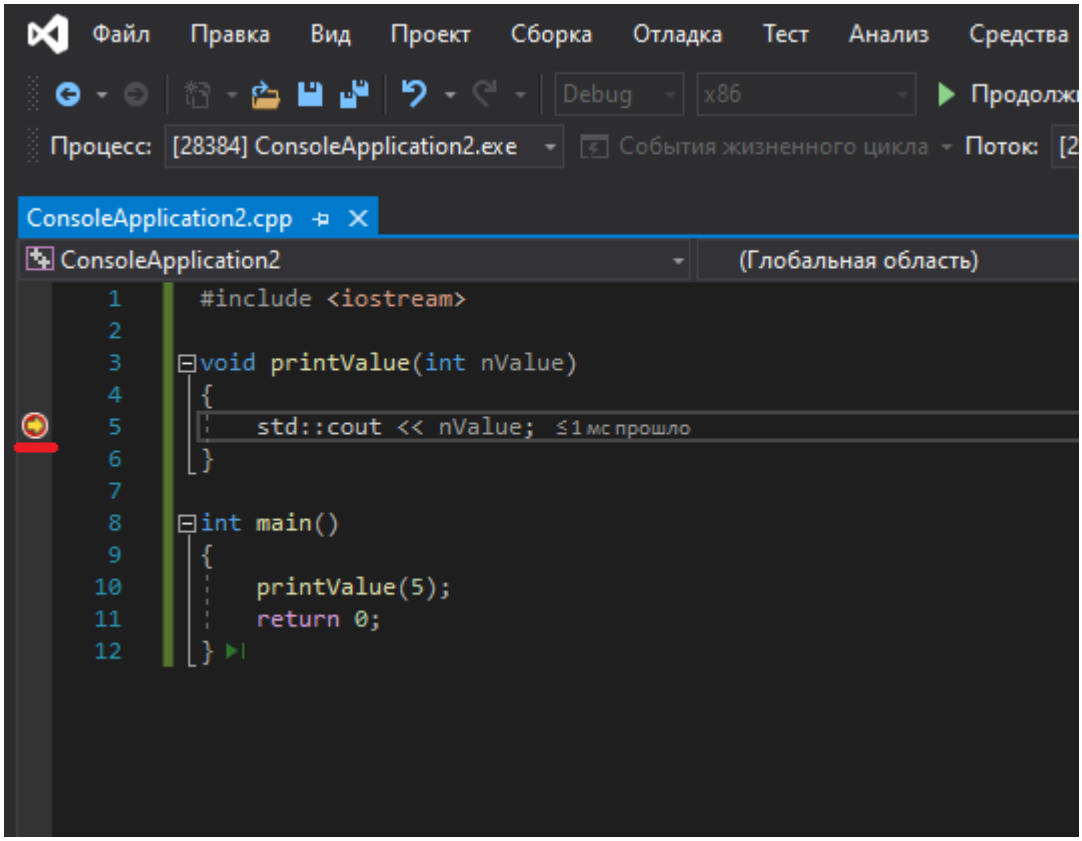


Появится кружочек возле строки:



В программе, приведенной выше, создайте точку останова на строке `std::cout << nValue;`. Затем выберите "Шаг с заходом" для старта сеанса отладки, а затем

"Продолжить". Вы увидите, что вместо завершения выполнения программы и остановки сеанса отладки, отладчик остановится в указанной вами точке:



Точки останова чрезвычайно полезны, если вы хотите изучить только определенную часть кода. Просто задайте точку останова в выбранном участке кода, выберите команду "Продолжить" и отладчик автоматически остановится возле указанной строки. Затем вы сможете использовать команды степпинга для более детального просмотра/изучения кода.

Урок №30. Отладка программ: стек вызовов и отслеживание переменных

На предыдущем уроке о степпинге и точках останова, мы узнали, как, с их использованием, контролировать выполнение программы шаг за шагом. Тем не менее, на этом не заканчиваются все полезные возможности отладчика. Он также позволяет отслеживать значения переменных.

> ***Примечание:*** Перед тем как продолжить, убедитесь, что вы находитесь в режиме конфигурации «Debug».

Отслеживание переменных

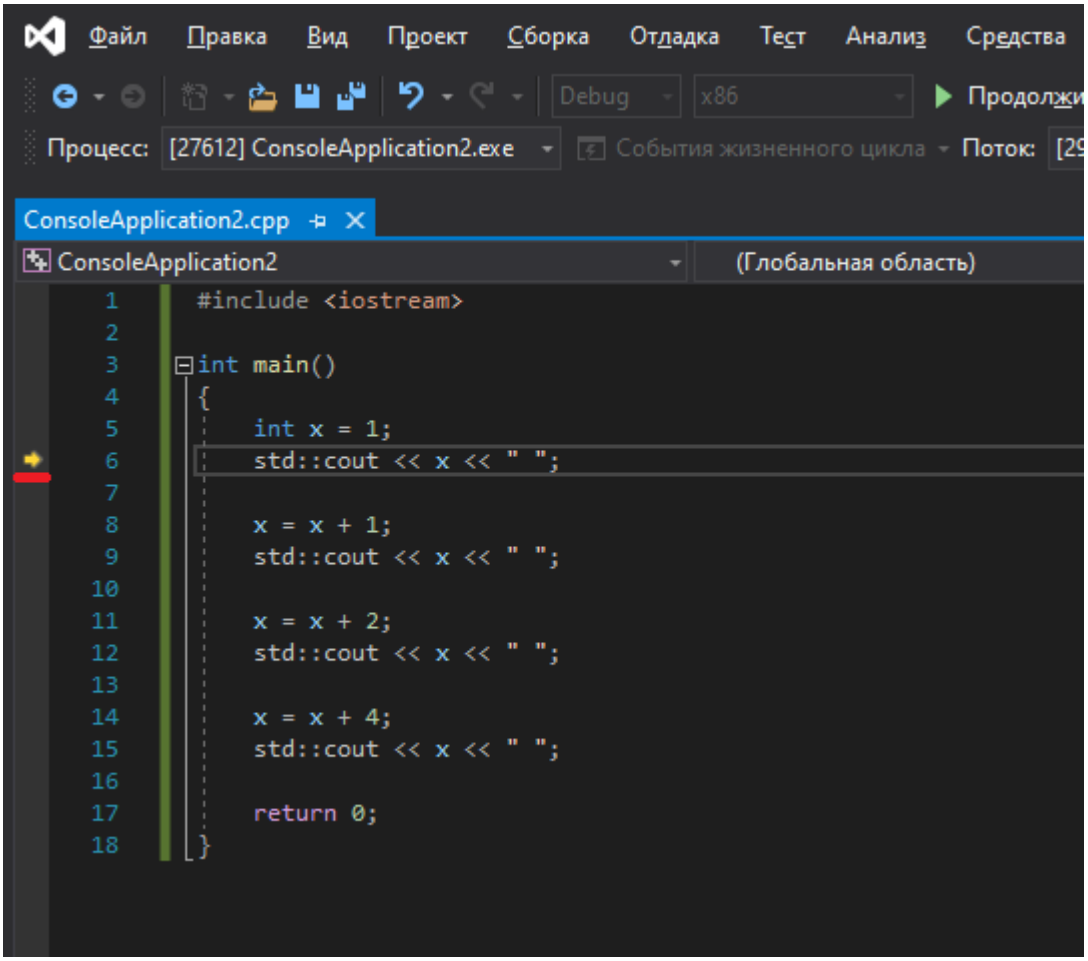
Отслеживание переменных - это процесс проверки значений переменных во время отладки. Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int x = 1;
6.     std::cout << x << " ";
7.
8.     x = x + 1;
9.     std::cout << x << " ";
10.
11.    x = x + 2;
12.    std::cout << x << " ";
13.
14.    x = x + 4;
15.    std::cout << x << " ";
16.
17.    return 0;
18.}
```

Результат выполнения программы:

```
1 2 4 8
```

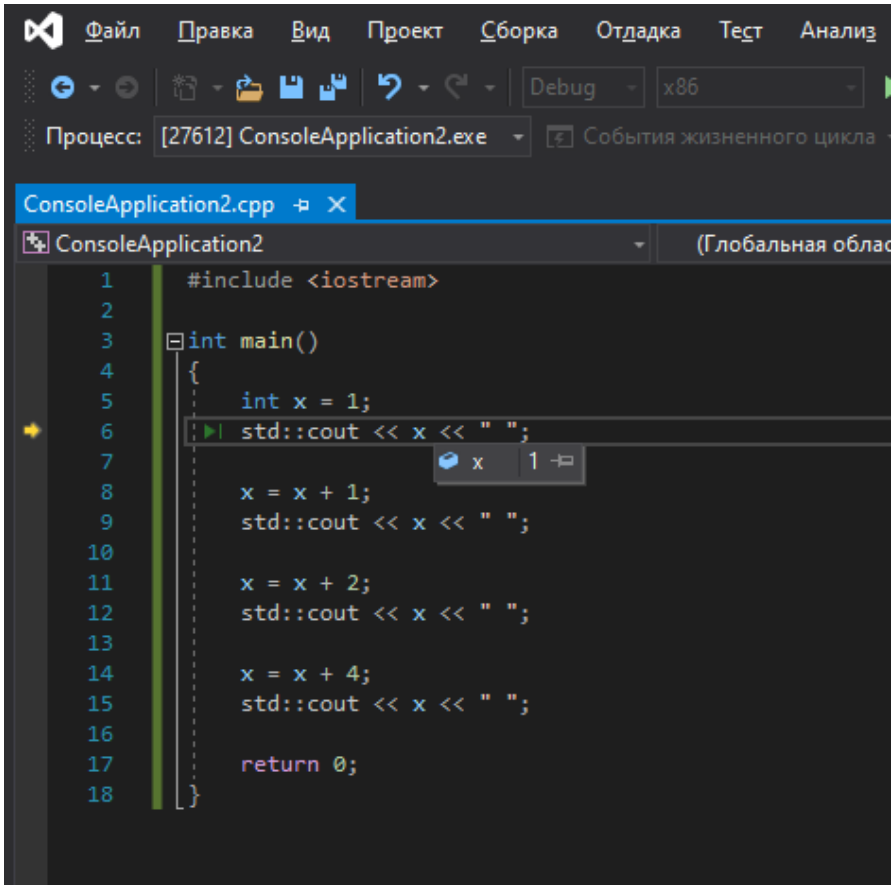
Используя команду "Выполнить до текущей позиции" переместитесь к строке `std::cout << x << " ";`:



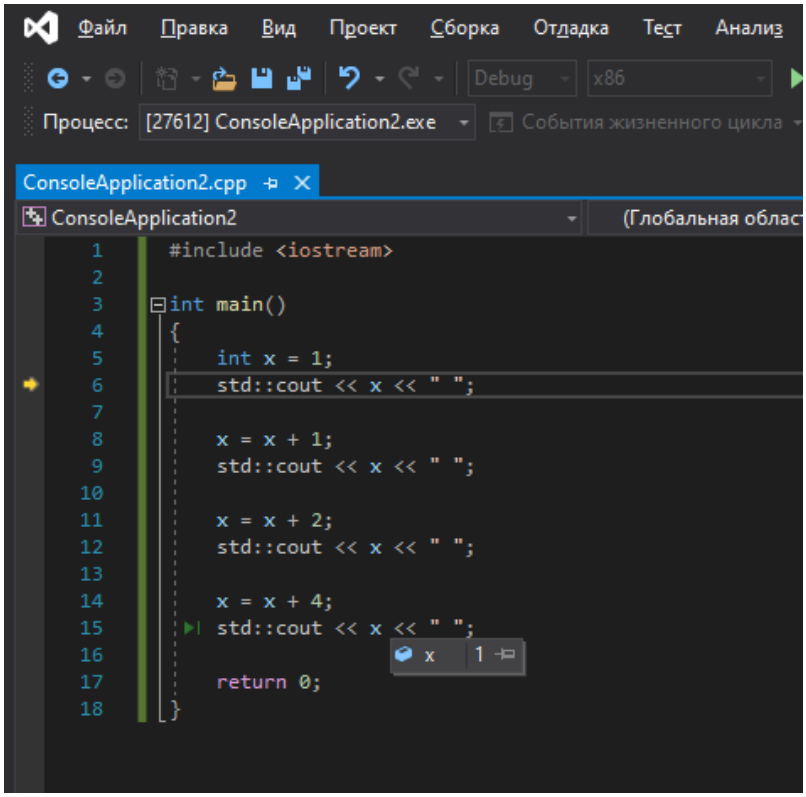
К этому моменту переменная `x` уже создана и инициализирована, поэтому, при проверке этой переменной, вы должны увидеть число `1`.

Самый простой способ отслеживания простых переменных (как `x`) - это наведение курсора мыши на элемент.

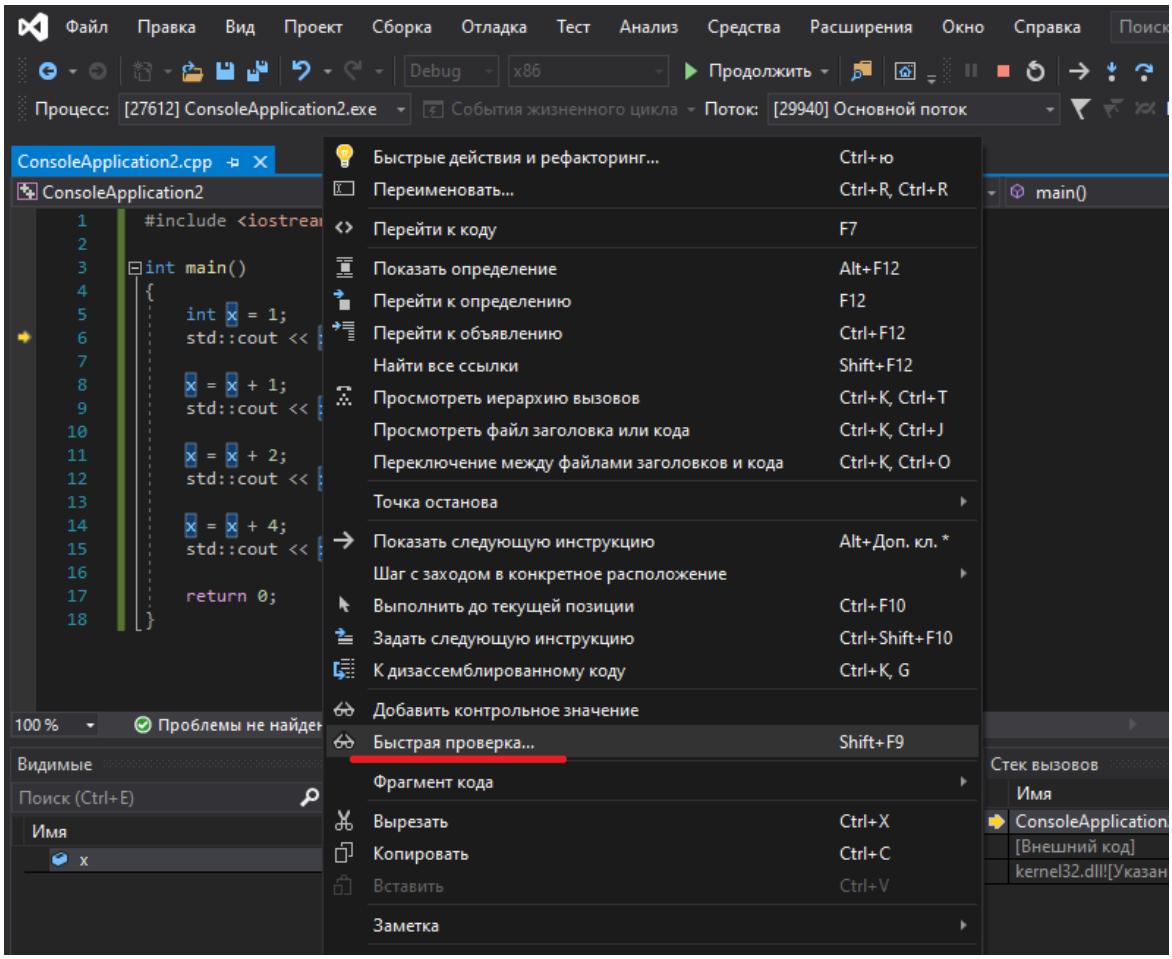
Большинство современных отладчиков поддерживают эту возможность:



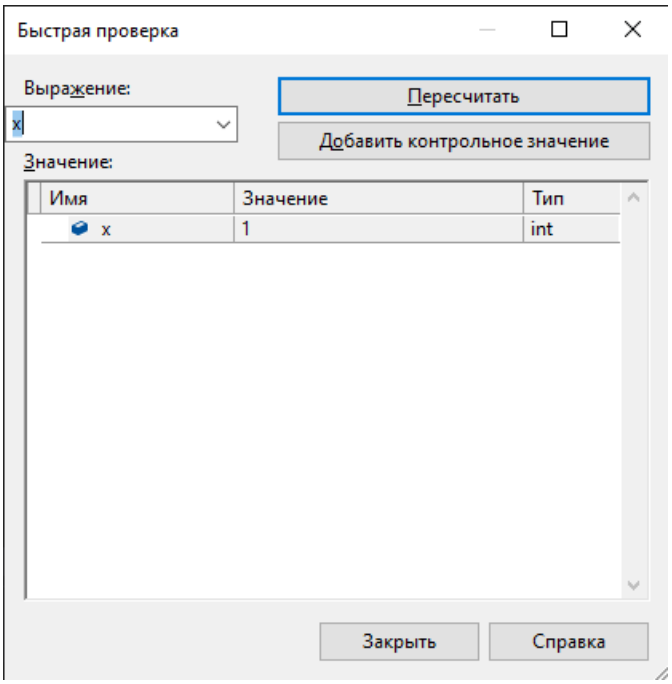
Обратите внимание, вы можете навести курсор мыши на любую другую переменную (и на любой строке):



В Visual Studio есть еще возможность использовать "Быструю проверку". Выделите переменную `x` с помощью мыши `> ПКМ (правая кнопка мыши) > "Быстрая проверка..."`:

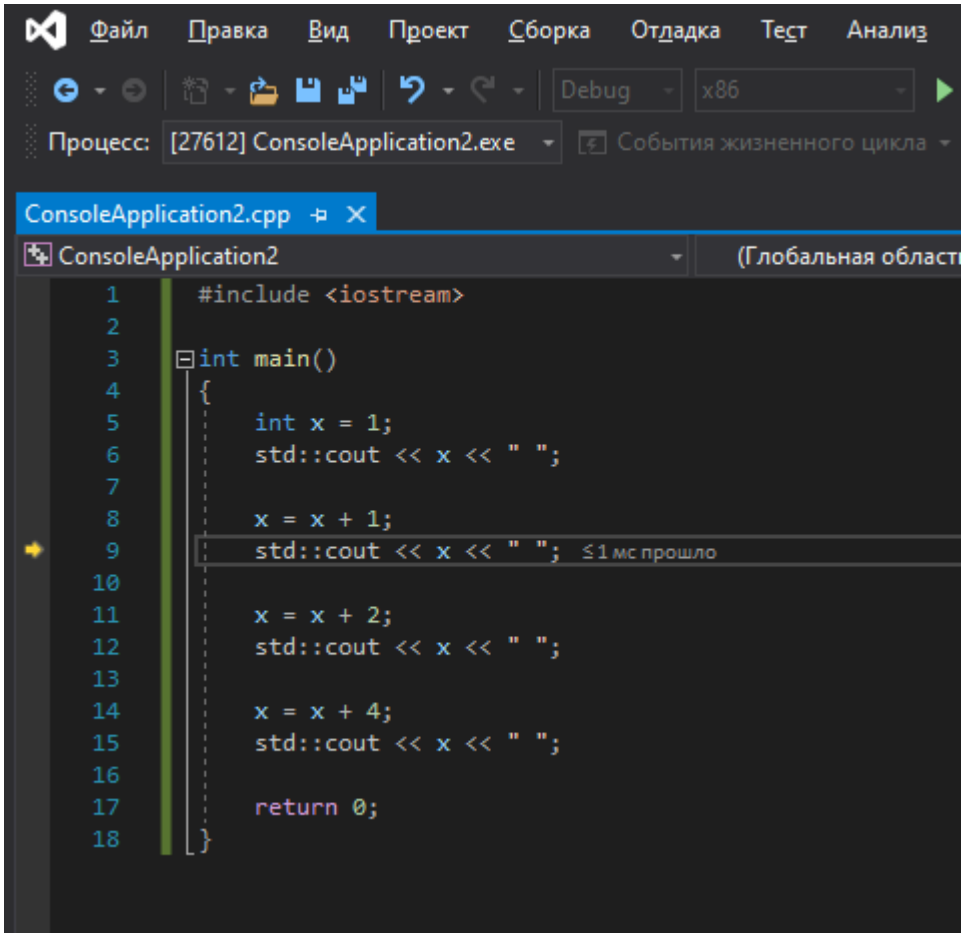


Появится специальное окно с текущим значением переменной:



Хорошо, теперь закройте это окно.

Значения переменных можно отслеживать и во время выполнения отладки. Переместитесь с помощью команды «Шаг с обходом» к строке `std::cout << x << " ";`:



Значение переменной `x` должно поменяться на `2`. Проверьте!

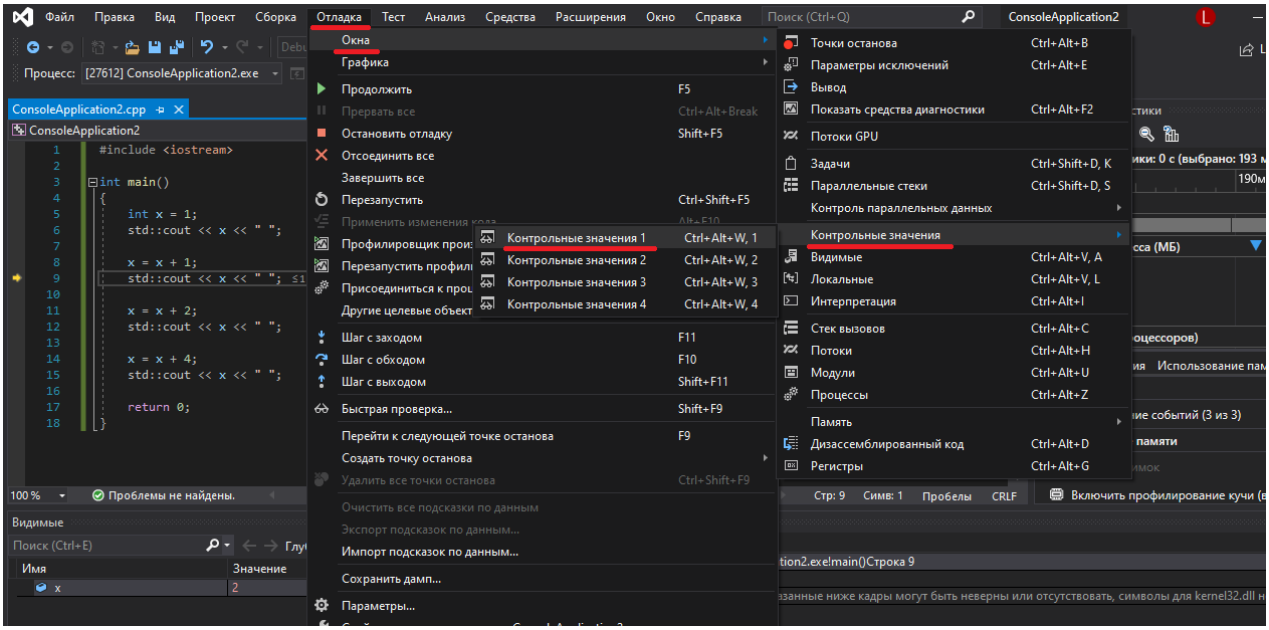
Окно просмотра

Команда "Быстрая проверка" или наведение курсора на элемент подходят для статического просмотра переменных, но не очень подходят для отслеживания изменений переменной во время выполнения программы, так как с каждой новой выполненной строкой придется заново наводить курсор на элемент или использовать "Быструю проверку".

Для решения этой проблемы, все современные отладчики предлагают еще один инструмент - окно просмотра. **Окно просмотра** — это окно, в которое можно добавлять переменные для постоянного отслеживания. Данные переменные будут автоматически обновляться при последовательном выполнении программы. Окно

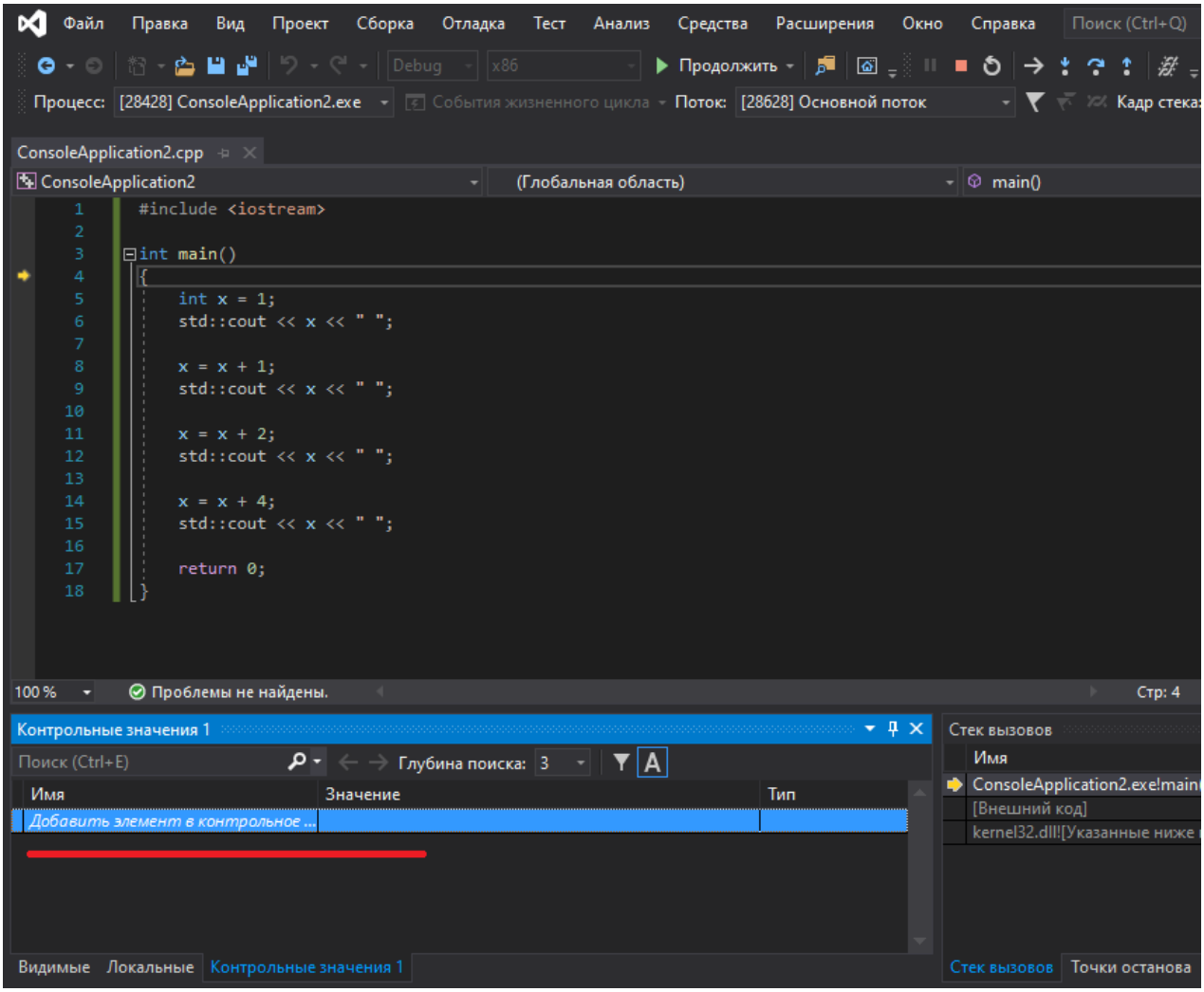
просмотра уже может быть подключено и отображаться в вашей рабочей области, но если это не так, то вы можете это исправить, перейдя в настройки вашей IDE.

В Visual Studio для отображения окна просмотра вам нужно перейти в `"Отладка" > "Окна" > "Контрольные значения" > "Контрольные значения 1"`:



> ***Примечание:*** Вы должны находиться в режиме отладки — используйте для этого команду «Шаг с заходом».

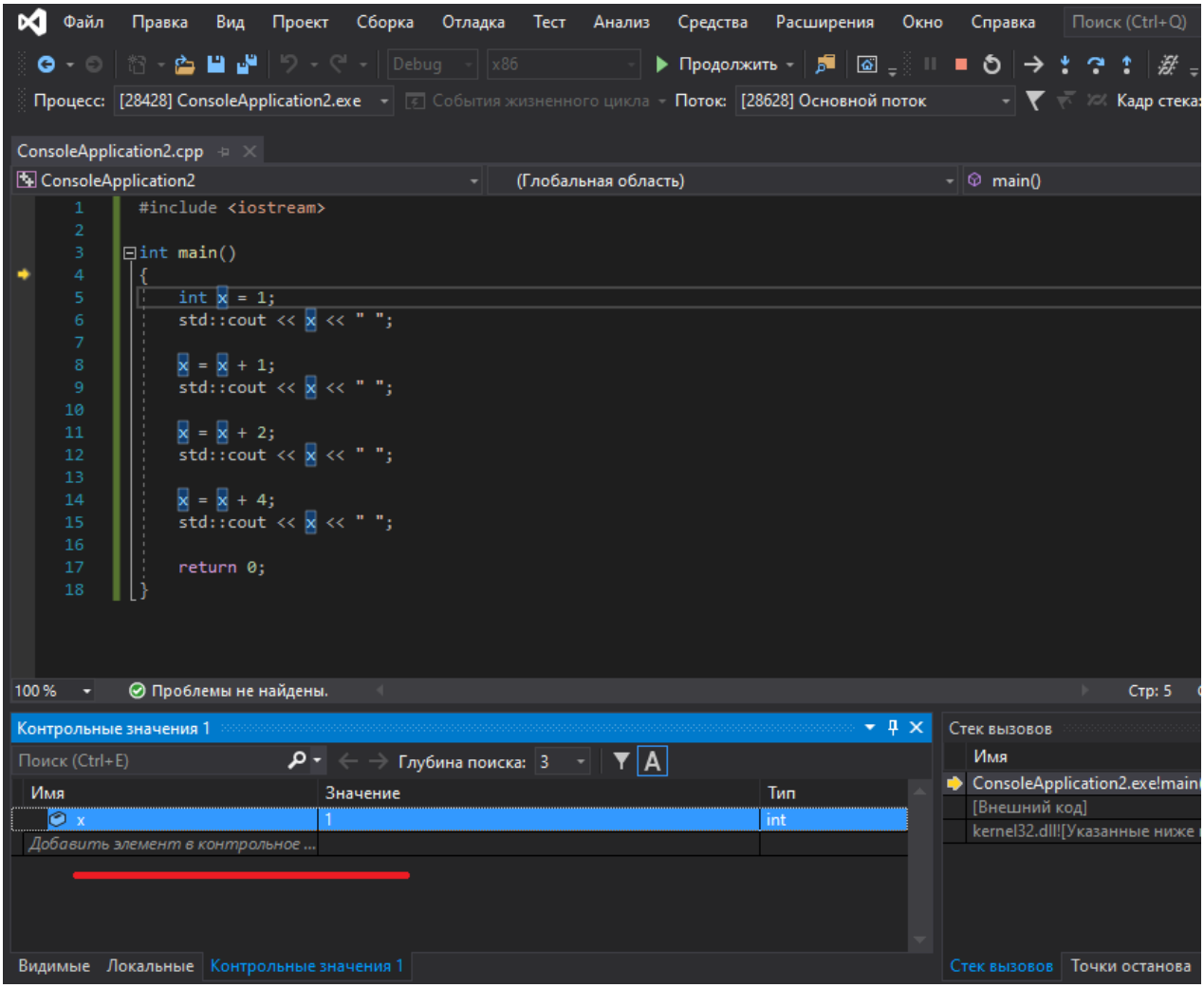
Вы должны увидеть следующее:



Пока что в этом окне ничего нет, так как вы еще ничего в него не добавили. Есть 2 пути:

- Ввести имя переменной, которую нужно отслеживать, в колонку `"Имя"` в окне просмотра.
- Выделить переменную, которую нужно отслеживать `> ПКМ > "Добавить контрольное значение"`.

Попробуйте добавить переменную `x` в окно просмотра:



Теперь выберите команду "Шаг с обходом" несколько раз и следите за изменениями значения вашей переменной!

Стек вызовов

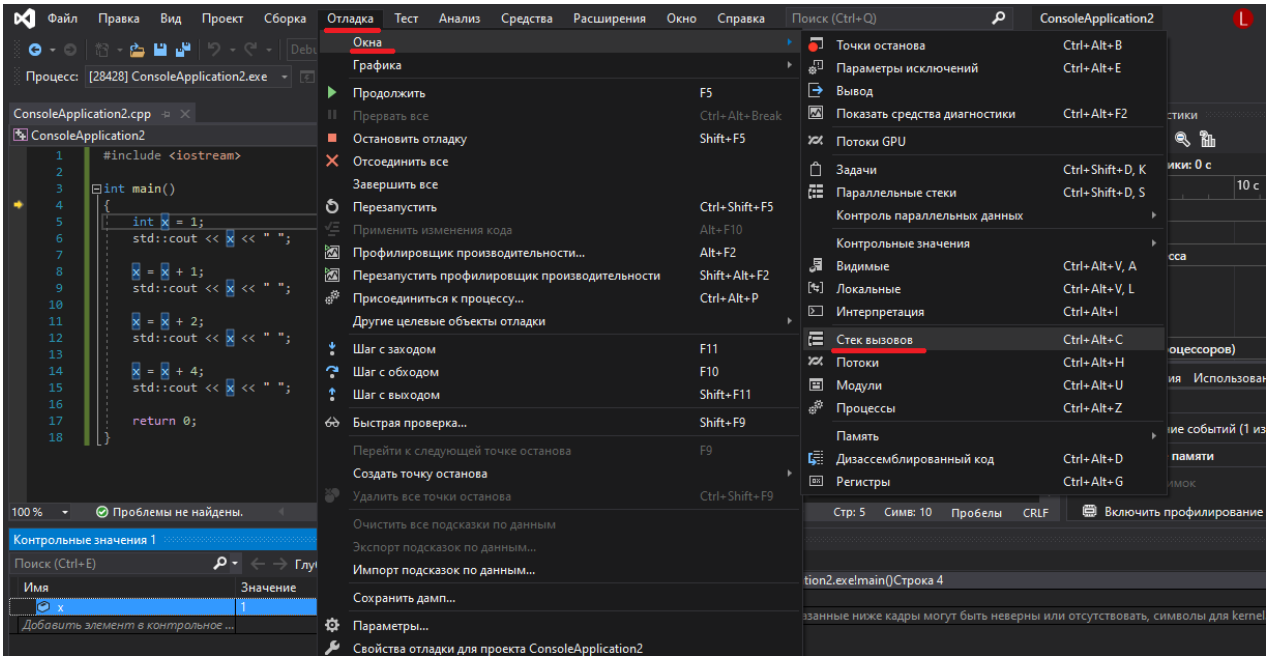
Современные отладчики имеют еще одно информационное окно, которое может быть очень полезным при отладке программ — "Стек вызовов".

Как вы уже знаете, при вызове функции программа оставляет закладку в текущем местоположении, выполняет функцию, а затем возвращается в место закладки. Программа отслеживает все вызовы функций в стеке вызовов.

Стек вызовов - это список всех активных функций, которые вызывались до текущего местоположения. В стек вызовов записывается вызываемая функция и выполняемая строка. Всякий раз, когда происходит вызов новой функции, эта новая функция добавляется в верх стека. Когда выполнение текущей функции прекращается, она

удаляется из верхней части стека и управление переходит к функции ниже (второй по счету).

Отобразить окно "Стека вызовов" в Visual Studio можно через `"Отладка" > "Окна"> "Стек вызовов"`:



> ***Примечание:*** Вы должны находиться в режиме отладки - используйте для этого команду "Шаг с заходом".

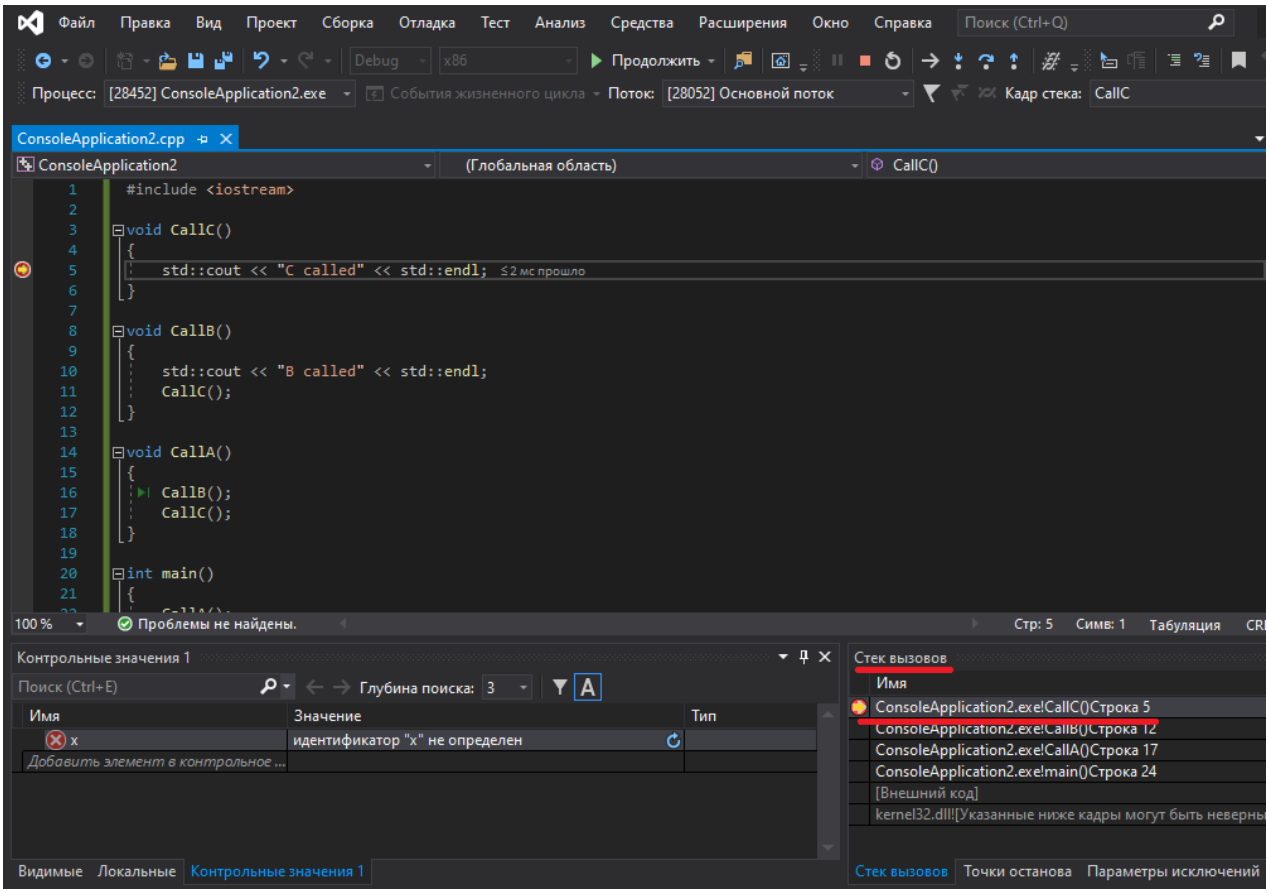
Рассмотрим пример:

```
1.  #include <iostream>
2.
3.  void CallC()
4.  {
5.      std::cout << "C called" << std::endl;
6.  }
7.
8.  void CallB()
9.  {
10.     std::cout << "B called" << std::endl;
11.     CallC();
12. }
13.
14. void CallA()
15. {
16.     CallB();
17.     CallC();
18. }
19.
20. int main()
21. {
22.     CallA();
23.
24.     return 0;
```

```
25. }
```

Укажите точку останова в функции CallC(), а затем запустите отладку. Программа выполнится до точки останова.

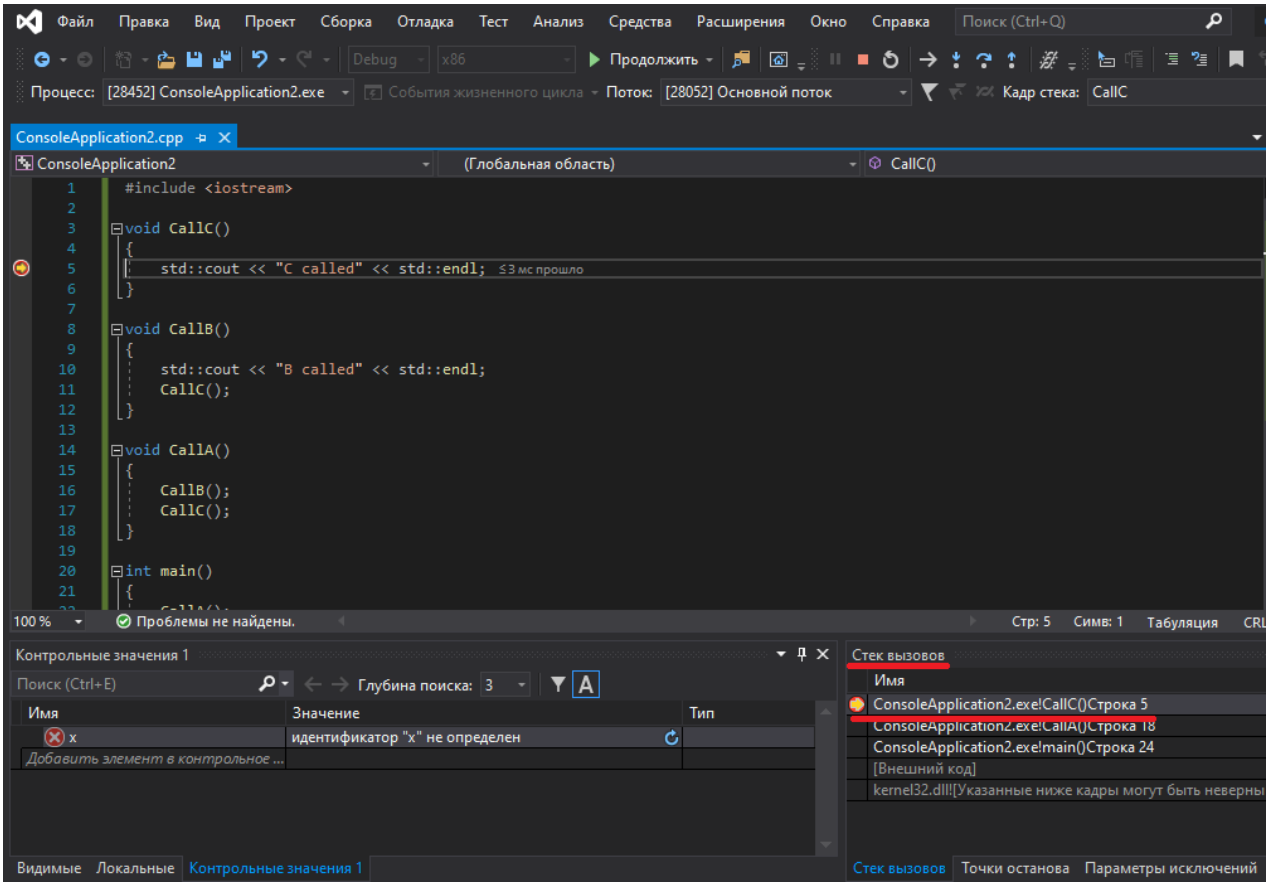
Несмотря на то, что вы знаете, что сейчас выполняется CallC(), в программе есть два вызова CallC(): в функции CallB() и в функции CallA(). Какая функция ответственна за вызов CallC() в данный момент? Стек вызовов нам это покажет:



Сначала выполняется main(). Затем main() вызывает CallA(), которая, в свою очередь, вызывает CallB(). Функция CallB() вызывает CallC(). Вы можете щелкнуть дважды по разным строкам в окне "Стек вызовов", чтобы увидеть больше информации о вызываемых функциях. Некоторые IDE переносят курсор непосредственно к вызову указанной функции. Visual Studio переносит курсор к следующей строке, которая находится после вызова функции. Попробуйте! Для того, чтобы возобновить степпинг, щелкните дважды по самой верхней (первой) строке в окне "Стек вызовов" и вы вернетесь к текущей точке выполнения.

Выберите команду "Продолжить". Точка останова должна сработать во второй раз, когда будет повторный вызов функции CallC() (на этот раз из функции CallA()).

Всё происходящее вы должны увидеть в окне "Стек вызовов":



Заключение

Теперь вы знаете об основных возможностях встроенных отладчиков! Используя степпинг, точки останова, отслеживание переменных и окно "Стек вызовов" вы можете успешно проводить отладку программ.

Глава №1. Итоговый тест

Данные задания призваны улучшить запоминание пройденного материала и показать его применение на практике.

Задание №1

Напишите однофайловую программу (с именем main.cpp), которая запрашивает у пользователя два целых числа, складывает их, а затем выводит результат. В программе должно быть 3 функции:

- функция readNumber(), которая запрашивает у пользователя целое число, а затем возвращает его в main();
- функция writeAnswer(), которая выводит результат на экран. Функция должна быть без возвращаемого значения и иметь только один параметр;
- функция main(), которая соединяет всё и вся.

> ***Подсказки:***
>
> - Для выполнения операции сложения не нужно создавать отдельную функцию (просто используйте оператор `+`).
> - Функцию readNumber() нужно вызывать дважды.

Задание №2

Измените программу из задания №1 так, чтобы функции readNumber() и writeAnswer() находились в отдельном файле io.cpp. Используйте предварительные объявления для доступа к этим функциям с функции main().

> ***Подсказка*:** Если у вас возникли проблемы, убедитесь, что io.cpp правильно добавлен к вашему проекту и подключен к компиляции.

Задание №3

Измените программу из задания №2 так, чтобы она использовала заголовочный файл io.h для доступа к функциям (вместо использования предварительных объявлений). Убедитесь, что ваш заголовочный файл использует header guards.

Урок №31. Инициализация, присваивание и объявление переменных

Этот урок является более детальным продолжением урока №13.

Адреса и переменные

Как вы уже знаете, переменные - это имена кусочков памяти, которые могут хранить информацию. Помним, что компьютеры имеют оперативную память, которая доступна программам для использования. Когда мы определяем переменную, часть этой памяти отводится ей.

Наименьшая единица памяти - **бит** (англ. *"**bit**"* от ***"bi**nary digi**t"***), который может содержать либо значение `0`, либо значение `1`. Вы можете думать о бите, как о переключателе света - либо свет выключен (`0`), либо включен (`1`). Чего-то среднего между ними нет. Если просмотреть случайный кусочек памяти, то всё, что вы увидите, — будет `...011010100101010...` или что-то в этом роде. Память организована в последовательные части, каждая из которых имеет свой **адрес**. Подобно тому, как мы используем адреса в реальной жизни, чтобы найти определенный дом на улице, так и здесь: адреса позволяют найти и получить доступ к содержимому, которое находится в определенном месте памяти. Возможно, это удивит вас, но в современных компьютерах, у каждого бита по отдельности нет своего собственного адреса. Наименьшей единицей с адресом является **байт** (который состоит из 8 битов).

Поскольку все данные компьютера — это лишь последовательность битов, то мы используем **тип данных** (или просто ***"тип"***), чтобы сообщить компилятору, как интерпретировать содержимое памяти. Вы уже видели пример типа данных: int (целочисленный тип данных). Когда мы объявляем целочисленную переменную, то мы сообщаем компилятору, что "кусочек памяти, который находится по *такому-то* адресу, следует интерпретировать как целое число".

Когда вы указываете тип данных для переменной, то компилятор и процессор заботятся о деталях преобразования вашего значения в соответствующую последовательность бит определенного типа данных. Когда вы просите ваше значение обратно, то оно "восстанавливается" из этой же последовательности бит.

Кроме int, есть много других типов данных в языке C++, большинство из которых мы детально рассмотрим на соответствующих уроках.

Фундаментальные типы данных в C++

В языке C++ есть встроенная поддержка определенных типов данных. Их называют **основными типами данных** (или ***"фундаментальные/базовые/встроенные типы данных"***).

Вот список основных типов данных в языке C++:

Категория	Тип	Значение	Пример
Логический тип данных	bool	true или false	true
Символьный тип данных	char, wchar_t, char16_t, char32_t	Один из ASCII-символов	‘c’
Тип данных с плавающей запятой	float, double, long double	Десятичная дробь	3.14159
Целочисленный тип данных	short, int, long, long long	Целое число	64
Пустота	void	Пустота	

Объявление переменных

Вы уже знаете, как объявить целочисленную переменную:

```
1. int nVarName; // int - это тип, а nVarName - это имя переменной
```

Принцип объявления переменных других типов аналогичен:

```
1. type varName; // type - это тип (например, int), а varName - это имя
   переменной
```

Объявление пяти переменных разных типов:

```
1. bool bValue;
2. char chValue;
3. int nValue;
4. float fValue;
5. double dValue;
```

Обратите внимание, переменной типа void здесь нет (о типе void мы поговорим детально на следующем уроке).

```
1. void vValue; // не будет работать, так как void не может использоваться в качестве типа переменной
```

Инициализация переменных

При объявлении переменной мы можем присвоить ей значение в этот же момент. Это называется **инициализацией переменной**.

Язык C++ поддерживает 2 основных способа инициализации переменных.

Способ №1: **Копирующая инициализация** (или ***"инициализация копированием"***) с помощью знака равенства `=`:

```
1. int nValue = 5; // копирующая инициализация
```

Способ №2: **Прямая инициализация** с помощью круглых скобок `()`:

```
1. int nValue(5); // прямая инициализация
```

Прямая инициализация лучше работает с одними типами данных, копирующая инициализация - с другими.

uniform-инициализация

Прямая или копирующая инициализация работают не со всеми типами данных (например, вы не сможете использовать эти способы для инициализации списка значений).

В попытке обеспечить единый механизм инициализации, который будет работать со всеми типами данных, в C++11 добавили новый способ инициализации, который называется **uniform-инициализация**:

```
1. int value{5};
```

Инициализация переменной с пустыми фигурными скобками указывает на инициализацию по умолчанию (переменной присваивается `0`):

```
1. int value{}; // инициализация переменной по умолчанию значением 0 (ноль)
```

В uniform-инициализации есть еще одно дополнительное преимущество: вы не сможете присвоить переменной значение, которое не поддерживает её тип данных - компилятор выдаст предупреждение или сообщение об ошибке.

Например:

```
1. int value{4.5}; // ошибка: целочисленная переменная не может содержать нецелочи
   сленные значения
```

Правило: Используйте uniform-инициализацию.

Присваивание переменных

Когда переменной присваивается значение после её объявления (не в момент объявления), то это **копирующее присваивание** (или просто ***"присваивание"***):

```
1. int nValue;
2. nValue = 5; // копирующее присваивание
```

В языке C++ нет встроенной поддержки способов прямого/uniform-присваивания, есть только копирующее присваивание.

Объявление нескольких переменных

В одном стейтменте можно объявить сразу несколько переменных одного и того же типа данных, разделяя их имена запятыми. Например, следующие 2 фрагмента кода выполняют одно и то же:

```
1. int a, b;
```

И:

```
1. int a;
2. int b;
```

Кроме того, вы даже можете инициализировать несколько переменных в одной строке:

```
1. int a = 5, b = 6;
2. int c(7), d(8);
3. int e{9}, f{10};
```

Есть 3 ошибки, которые совершают новички при объявлении нескольких переменных в одном стейтменте:

Ошибка №1*:* Указание каждой переменной одного и того же типа данных при инициализации нескольких переменных в одном стейтменте. Это не критичная ошибка, так как компилятор легко её обнаружит и сообщит вам об этом:

```
1. int a, int b; // неправильно (ошибка компиляции)
2.
3. int a, b; // правильно
```

***Ошибка №2**:* Использование разных типов данных в одном стейтменте. Переменные разных типов должны быть объявлены в разных стейтментах. Эту ошибку компилятор также легко обнаружит:

```
int a, double b; // неправильно (ошибка компиляции)

int a; double b; // правильно (но не рекомендуется)

// Правильно и рекомендуется (+ читабельнее)
int a;
double b;
```

Ошибка №3: Инициализация двух переменных с помощью одной операции:

```
int a, b = 5; // неправильно (переменная a остаётся неинициализированной)

int a = 5, b = 5; // правильно
```

Хороший способ запомнить эту ошибку и не допускать в будущем - использовать прямую или uniform-инициализацию:

```
int a, b(5);
int c, d{5};
```

Этот вариант наглядно показывает, что значение `5` присваивается только переменным `b` и `d`.

Так как инициализация нескольких переменных в одной строке является отличным поводом для совершения ошибок, то я рекомендую определять несколько переменных в одной строке только в том случае, если вы будете каждую из них инициализировать.

Правило: Избегайте объявления нескольких переменных в одной строке, если не собираетесь инициализировать каждую из них.

Где объявлять переменные?

Более старые версии компиляторов языка Си вынуждали пользователей объявлять все переменные в верхней части функции:

```
#include <iostream>

int main()
{
    // Все переменные в самом верху функции
    int x;
    int y;

    // А затем уже весь остальной код
    std::cout << "Enter a number: ";
```

```
11.     std::cin >> x;
12.
13.     std::cout << "Enter another number: ";
14.     std::cin >> y;
15.
16.     std::cout << "The sum is: " << x + y << std::endl;
17.     return 0;
18. }
```

Сейчас это уже неактуально. Современные компиляторы не требуют, чтобы все переменные обязательно были объявлены в самом верху функции. В языке C++ приоритетным является объявление (а также инициализация) переменных как можно ближе к их первому использованию:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      std::cout << "Enter a number: ";
6.      int x; // мы используем x в следующей строке, поэтому объявляем эту
    переменную здесь (как можно ближе к её первому использованию)
7.      std::cin >> x; // первое использование переменной x
8.
9.      std::cout << "Enter another number: ";
10.     int y; // переменная y понадобилась нам только здесь, поэтому здесь её и
    объявляем
11.     std::cin >> y; // первое использование переменной y
12.
13.     std::cout << "The sum is: " << x + y << std::endl;
14.     return 0;
15. }
```

Такой стиль написания кода имеет несколько преимуществ:

- Во-первых, возле переменных, которые объявлены как можно ближе к их первому использованию, находится другой код, который способствует лучшему представлению и пониманию происходящего. Если бы переменная `x` была объявлена в начале функции main(), то мы бы не имели ни малейшего представления, для чего она используется. Для понимания смысла переменной пришлось бы целиком просматривать всю функцию. Объявление переменной `x` возле операций ввода/вывода данных позволяет нам понять, что эта переменная используется для ввода/вывода данных.
- Во-вторых, объявление переменных только там, где они необходимы, сообщает нам о том, что эти переменные не влияют на код, расположенный выше, что делает программу проще для понимания.
- И, наконец, уменьшается вероятность случайного создания неинициализированных переменных.

В большинстве случаев, вы можете объявить переменную непосредственно в строке перед её первым использованием. Тем не менее, иногда, могут быть случаи, когда

это будет не желательно (по соображениям производительности) или невозможно. Детально об этом мы поговорим на следующих уроках.

Правило: Объявляйте переменные как можно ближе к их первому использованию.

Урок №32. Тип данных void

Тип void — это самый простой тип данных, который означает "отсутствие любого типа данных". Следовательно, переменные не могут быть типа void:

```
1. void value; // не будет работать, так как переменная не может иметь тип void
```

Тип void, как правило, используется в 3-х случаях:

Использование №1: Указать, что функция не возвращает значение:

```
1. void writeValue(int x) // здесь void означает, что функция не возвращает
   никакое значение
2. {
3.     std::cout << "The value of x is: " << x << std::endl;
4.     // Нет стейтмента return, так как тип функции - void
5. }
```

Использование №2: Указать, что функция не имеет никаких параметров (перешло из языка Си):

```
1. int getValue(void) // здесь void означает, что функция не имеет никаких
   параметров
2. {
3.     int x;
4.     std::cin >> x;
5.     return x;
6. }
```

Указание типа void как "никаких параметров" является пережитком, сохранившимся еще со времен языка Си. Следующий код равнозначен и более предпочтителен для использования в языке C++:

```
1. int getValue() // пустые скобки означают то же, что и void
2. {
3.     int x;
4.     std::cin >> x;
5.     return x;
6. }
```

Правило: Используйте пустой список параметров вместо void для указания отсутствия параметров в функции.

Использование №3: Ключевое слово void имеет третий (более продвинутый) способ использования в языке C++, который мы будем рассматривать на соответствующем уроке.

Урок №33. Размер типов данных

Память на современных компьютерах, как правило, организована в блоки, которые состоят из байтов, причем каждый блок имеет свой уникальный адрес. До этого момента, память можно было сравнивать с почтовыми ящиками (с теми, которые находятся в каждом подъезде), куда мы можем поместить информацию и откуда мы её можем извлечь, а имена переменных - это всего лишь номера этих почтовых ящиков.

Тем не менее, эта аналогия не совсем подходит к программированию, так как переменные могут занимать больше 1 байта памяти. Следовательно, одна переменная может использовать 2, 4 или даже 8 последовательных адресов. Объем памяти, который использует переменная, зависит от типа данных этой переменной. Так как мы, как правило, получаем доступ к памяти через имена переменных, а не через адреса памяти, то компилятор может скрывать от нас все детали работы с переменными разных размеров.

Есть несколько причин по которым полезно знать, сколько памяти занимает определенная переменная/тип данных.

Во-первых, чем больше она занимает, тем больше информации сможет хранить. Так как каждый бит содержит либо `0`, либо `1`, то 1 бит может иметь 2 возможных значения.

2 бита могут иметь 4 возможных значения:

бит 0	бит 1
0	0
0	1
1	0
1	1

3 бита могут иметь 8 возможных значений:

бит 0	бит 1	бит 2
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

По сути, переменная с n-ным количеством бит может иметь 2^n возможных значений. Поскольку байт состоит из 8 бит, то он может иметь 2^8 (256) возможных значений.

Размер переменной накладывает ограничения на количество информации, которую она может хранить. Следовательно, переменные, которые используют больше байт, могут хранить более широкий диапазон значений.

Во-вторых, компьютеры имеют ограниченное количество свободной памяти. Каждый раз, когда мы объявляем переменную, небольшая часть этой свободной памяти выделяется до тех пор, пока переменная существует. Поскольку современные компьютеры имеют много памяти, то в большинстве случаев это не является проблемой, особенно когда в программе всего лишь несколько переменных. Тем не менее, для программ с большим количеством переменных (например, 100 000), разница между использованием 1-байтовых или 8-байтовых переменных может быть значительной.

Размер основных типов данных в C++

Возникает вопрос: "Сколько памяти занимают переменные разных типов данных?". Вы можете удивиться, но размер переменной с любым типом данных зависит от компилятора и/или архитектуры компьютера!

Язык C++ гарантирует только их минимальный размер:

Категория	Тип	Минимальный размер
Логический тип данных	bool	1 байт
Символьный тип данных	char	1 байт
	wchar_t	1 байт
	char16_t	2 байта
	char32_t	4 байта
Целочисленный тип данных	short	2 байта
	int	2 байта
	long	4 байта
	long long	8 байт
Тип данных с плавающей запятой	float	4 байта
	double	8 байт
	long double	8 байт

Фактический размер переменных может отличаться на разных компьютерах, поэтому для его определения используют оператор sizeof.

Оператор sizeof — это унарный оператор, который вычисляет и возвращает размер определенной переменной или определенного типа данных в байтах. Вы можете скомпилировать и запустить следующую программу, чтобы выяснить, сколько занимают разные типы данных на вашем компьютере:

```
#include <iostream>

int main()
{
    std::cout << "bool:\t\t" << sizeof(bool) << " bytes" << std::endl;
    std::cout << "char:\t\t" << sizeof(char) << " bytes" << std::endl;
    std::cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes" << std::endl;
    std::cout << "char16_t:\t" << sizeof(char16_t) << " bytes" << std::endl;
    std::cout << "char32_t:\t" << sizeof(char32_t) << " bytes" << std::endl;
    std::cout << "short:\t\t" << sizeof(short) << " bytes" << std::endl;
    std::cout << "int:\t\t" << sizeof(int) << " bytes" << std::endl;
    std::cout << "long:\t\t" << sizeof(long) << " bytes" << std::endl;
    std::cout << "long long:\t" << sizeof(long long) << " bytes" << std::endl;
    std::cout << "float:\t\t" << sizeof(float) << " bytes" << std::endl;
    std::cout << "double:\t\t" << sizeof(double) << " bytes" << std::endl;
    std::cout << "long double:\t" << sizeof(long double) << " bytes" << std::endl;
    return 0;
}
```

Вот результат, полученный на моем компьютере:

```
bool:           1 bytes
char:           1 bytes
wchar_t:        2 bytes
char16_t:       2 bytes
char32_t:       4 bytes
short:          2 bytes
int:            4 bytes
long:           4 bytes
long long:      8 bytes
float:          4 bytes
double:         8 bytes
long double:    8 bytes
```

Ваши результаты могут отличаться, если у вас другая архитектура, или другой компилятор. Обратите внимание, оператор sizeof не используется с типом void, так как последний не имеет размера.

Если вам интересно, что значит `\t` в коде, приведенном выше, то это специальный символ, который используется вместо клавиши TAB. Мы его использовали для выравнивания столбцов. Детально об этом мы еще поговорим на соответствующих уроках.

Интересно то, что sizeof — это один из 3-х операторов в языке C++, который является словом, а не символом (еще есть new и delete).

Вы также можете использовать оператор sizeof и с переменными:

```
1. #include <iostream>
2. 
3. int main()
4. {
5.     int x;
6.     std::cout << "x is " << sizeof(x) << " bytes" << std::endl;
7. }
```

Результат выполнения программы:

```
x is 4 bytes
```

На следующих уроках мы рассмотрим каждый из фундаментальных типов данных языка C++ по отдельности.

Урок №34. Целочисленные типы данных: short, int и long

Целочисленный тип данных — это тип, переменные которого могут содержать только целые числа (без дробной части, например: -2, -1, 0, 1, 2). В языке C++ есть 5 основных целочисленных типов, доступных для использования:

Категория	Тип	Минимальный размер
Символьный тип данных	char	1 байт
Целочисленный тип данных	short	2 байта
	int	2 байта (но чаще всего 4 байта)
	long	4 байта
	long long	8 байт

> ***Примечание:*** Тип char - это особый случай: он является как целочисленным, так и символьным типом данных. Об этом детально мы поговорим на одном из следующих уроков.

Основным различием между целочисленными типами, перечисленными выше, является их размер, чем он больше, тем больше значений сможет хранить переменная этого типа.

Определение целочисленных переменных

Определение происходит следующим образом:

```
1. char c;
2. short int si; // допустимо
3. short s;      // предпочтительнее
4. int i;
5. long int li; // допустимо
6. long l;      // предпочтительнее
7. long long int lli; // допустимо
8. long long ll;      // предпочтительнее
```

В то время как полные названия `short int`, `long int` и `long long int` могут использоваться, их сокращенные версии (без `int`) более предпочтительны для использования. К тому же постоянное добавление `int` затрудняет чтение кода (легко перепутать с именем переменной).

Диапазоны значений и знак целочисленных типов данных

Как вы уже знаете из предыдущего урока, переменная с n-ным количеством бит может хранить 2^n возможных значений. Но что это за значения? Это значения, которые находятся в диапазоне. **Диапазон** — это значения от и до, которые может хранить определенный тип данных. Диапазон целочисленной переменной определяется двумя факторами: её размером (измеряется в битах) и её **знаком** (который может быть *signed* или *unsigned*).

Целочисленный тип signed (со знаком) означает, что переменная может содержать как положительные, так и отрицательные числа. Чтобы объявить переменную как signed, используйте ключевое слово `signed`:

```
1. signed char c;
2. signed short s;
3. signed int i;
4. signed long l;
5. signed long long ll;
```

По умолчанию, ключевое слово `signed` пишется перед типом данных.

1-байтовая целочисленная переменная со знаком (signed) имеет диапазон значений от -128 до 127, т.е. любое значение от -128 до 127 (включительно) может храниться в ней безопасно.

В некоторых случаях мы можем заранее знать, что отрицательные числа в программе использоваться не будут. Это очень часто встречается при использовании переменных для хранения количества или размера чего-либо (например, ваш рост или вес не может быть отрицательным).

Целочисленный тип unsigned (без знака) может содержать только положительные числа. Чтобы объявить переменную как **unsigned**, используйте ключевое слово `unsigned`:

```
1. unsigned char c;
2. unsigned short s;
3. unsigned int i;
4. unsigned long l;
5. unsigned long long ll;
```

1-байтовая целочисленная переменная без знака (unsigned) имеет диапазон значений от 0 до 255.

Обратите внимание, объявление переменной как unsigned означает, что она не сможет содержать отрицательные числа (только положительные).

Теперь, когда вы поняли разницу между signed и unsigned, давайте рассмотрим диапазоны значений разных типов данных:

Размер/Тип	Диапазон значений
1 байт signed	от -128 до 127
1 байт unsigned	от 0 до 255
2 байта signed	от -32 768 до 32 767
2 байта unsigned	от 0 до 65 535
4 байта signed	от -2 147 483 648 до 2 147 483 647
4 байта unsigned	от 0 до 4 294 967 295
8 байтов signed	от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
8 байтов unsigned	от 0 до 18 446 744 073 709 551 615

Для математиков: Переменная signed с n-ным количеством бит имеет диапазон от $-(2^{n-1})$ до $2^{n-1}-1$. Переменная unsigned с n-ным количеством бит имеет диапазон от 0 до $(2^n)-1$.

Для нематематиков: Используем таблицу :)

Начинающие программисты иногда путаются между signed и unsigned переменными. Но есть простой способ запомнить их различия. Чем отличается отрицательное число от положительного? Правильно! Минусом спереди. Если минуса нет, значит число - положительное. Следовательно, целочисленный тип со

знаком (signed) означает, что минус может присутствовать, т.е. числа могут быть как положительными, так и отрицательными. Целочисленный тип без знака (unsigned) означает, что минус спереди отсутствует, т.е. числа могут быть только положительными.

Что используется по умолчанию: signed или unsigned?

Так что же произойдет, если мы объявим переменную без указания signed или unsigned?

Категория	Тип	По умолчанию
Символьный тип данных	char	signed или unsigned (в большинстве случаев signed)
Целочисленный тип данных	short	signed
	int	signed
	long	signed
	long long	signed

Все целочисленные типы данных, кроме char, являются signed по умолчанию. Тип char может быть как signed, так и unsigned (но, обычно, signed).

В большинстве случаев ключевое слово signed не пишется (оно и так используется по умолчанию).

Программисты, как правило, избегают использования целочисленных типов unsigned, если в этом нет особой надобности, так как с переменными unsigned ошибок, по статистике, возникает больше, нежели с переменными signed.

Правило: Используйте целочисленные типы signed, вместо unsigned.

Переполнение

Вопрос: "Что произойдет, если мы попытаемся использовать значение, которое находится вне диапазона значений определенного типа данных?". Ответ: "Переполнение".

Переполнение (англ. ***"overflow"***) случается при потере бит из-за того, что переменной не было выделено достаточно памяти для их хранения.

Ранее мы говорили о том, что данные хранятся в бинарном (двоичном) формате и каждый бит может иметь только 2 возможных значения (`0` или `1`). Вот как выглядит диапазон чисел от 0 до 15 в десятичной и двоичной системах:

Десятичная система	Двоичная система
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Как вы можете видеть, чем больше число, тем больше ему требуется бит. Поскольку наши переменные имеют фиксированный размер, то на них накладываются ограничения на количество данных, которые они могут хранить.

Примеры переполнения

Рассмотрим переменную unsigned, которая состоит из 4-х бит. Любое из двоичных чисел, перечисленных в таблице выше, поместится внутри этой переменной.

"Но что произойдет, если мы попытаемся присвоить значение, которое занимает больше 4-х бит?". Правильно! Переполнение. Наша переменная будет хранить только 4 наименее значимых (те, что справа) бита, все остальные - потеряются.

Например, если мы попытаемся поместить число 21 в нашу 4-битную переменную:

Десятичная система	Двоичная система
21	10101

Число 21 занимает 5 бит (10101). 4 бита справа (0101) поместятся в переменную, а крайний левый бит (1) просто потеряется. Т.е. наша переменная будет содержать 0101, что равно 101 (нуль спереди не считается), а это уже число 5, а не 21.

> ***Примечание**:* О конвертации чисел из двоичной системы в десятичную и наоборот будет отдельный урок, где мы всё детально рассмотрим и обсудим.

Теперь рассмотрим пример в коде (тип short занимает 16 бит):

```
#include <iostream>

int main()
{
    unsigned short x = 65535; // наибольшее значение, которое может хранить 16-битная unsigned переменная
    std::cout << "x was: " << x << std::endl;
    x = x + 1; // 65536 - это число больше максимально допустимого числа из диапазона допустимых значений. Следовательно, произойдет переполнение, так как переменнная x не может хранить 17 бит
    std::cout << "x is now: " << x << std::endl;
    return 0;
}
```

Результат выполнения программы:

```
x was: 65535
x is now: 0
```

Что случилось? Произошло переполнение, так как мы попытались присвоить переменной `x` значение больше, чем она способна в себе хранить.

> ***Для тех, кто хочет знать больше:*** Число 65 535 в двоичной системе счисления представлено как 1111 1111 1111 1111. 65 535 — это наибольшее число, которое может хранить 2-байтовая (16 бит) целочисленная переменная без знака, так как это число использует все 16 бит. Когда мы добавляем 1, то получаем число 65 536. Число 65 536 представлено в двоичной системе как 1 0000 0000 0000 0000, и занимает 17 бит! Следовательно, самый главный бит (которым является 1) теряется, а все 16 бит справа - остаются. Комбинация 0000 0000 0000 0000 соответствует десятичному 0, что и является нашим результатом.

Аналогичным образом, мы получим переполнение, использовав число меньше минимального из диапазона допустимых значений:

```
#include <iostream>

int main()
{
    unsigned short x = 0; // наименьшее значение, которое 2-байтовая unsigned переменная может хранить
    std::cout << "x was: " << x << std::endl;
    x = x - 1; // переполнение!
    std::cout << "x is now: " << x << std::endl;
    return 0;
}
```

Результат выполнения программы:

```
x was: 0
x is now: 65535
```

Переполнение приводит к потере информации, а это никогда не приветствуется. Если есть хоть малейшее подозрение или предположение, что значением переменной может быть число, которое находится вне диапазона допустимых значений используемого типа данных — используйте тип данных побольше!

Правило: Никогда не допускайте возникновения переполнения в ваших программах!

Деление целочисленных переменных

В языке C++ при делении двух целых чисел, где результатом является другое целое число, всё довольно предсказуемо:

```
#include <iostream>

int main()
{
    std::cout << 20 / 4 << std::endl;
    return 0;
}
```

Результат:

```
5
```

Но что произойдет, если в результате деления двух целых чисел мы получим дробное число? Например:

```
#include <iostream>

int main()
{
    std::cout << 8 / 5 << std::endl;
    return 0;
}
```

Результат:

```
1
```

В языке C++ при делении целых чисел результатом всегда будет другое целое число. А такие числа не могут иметь дробь (она просто отбрасывается, **не округляется!**).

Рассмотрим детально вышеприведенный пример: `8 / 5 = 1.6`. Но как мы уже знаем, при делении целых чисел результатом является другое целое число. Таким образом, дробная часть (`0.6`) значения отбрасывается и остается `1`.

Правило: Будьте осторожны при делении целых чисел, так как любая дробная часть всегда отбрасывается.

Урок №35. Фиксированный размер целочисленных типов данных

На уроке о целочисленных типах данных мы говорили, что C++ гарантирует только их минимальный размер - они могут занимать и больше, в зависимости от компилятора и/или архитектуры компьютера.

Почему размер целочисленных типов не является фиксированным?

Если говорить в общем, то все еще началось с языка Си, когда производительность имела первостепенное значение. В языке Си намеренно оставили размер целочисленных типов нефиксированным для того, чтобы компилятор мог самостоятельно подобрать наиболее подходящий размер для определенного типа данных в зависимости от компьютерной архитектуры.

Разве это не глупо?

Возможно. Программистам не всегда удобно иметь дело с переменными, размер которых варьируется в зависимости от компьютерной архитектуры.

Целочисленные типы фиксированного размера

Чтобы решить вопрос кроссплатформенности, в язык C++ добавили набор **целочисленных типов фиксированного размера**, которые гарантированно имеют один и тот же размер на любой архитектуре:

Название	Тип	Диапазон значений
int8_t	1 байт signed	от -128 до 127
uint8_t	1 байт unsigned	от 0 до 255
int16_t	2 байта signed	от -32 768 до 32 767
uint16_t	2 байта unsigned	от 0 до 65 535

int32_t	4 байта signed	от -2 147 483 648 до 2 147 483 647
uint32_t	4 байта unsigned	от 0 до 4 294 967 295
int64_t	8 байт signed	от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
uint64_t	8 байт unsigned	от 0 до 18 446 744 073 709 551 615

Начиная с C++11 доступ к этим типам осуществляется через подключение заголовочного файла cstdint (находятся эти типы данных в пространстве имен std). Рассмотрим пример на практике:

```
1. #include <iostream>
2. #include <cstdint>
3.
4. int main()
5. {
6.     std::int16_t i(5); // прямая инициализация
7.     std::cout << i << std::endl;
8.     return 0;
9. }
```

Поскольку целочисленные типы фиксированного размера были добавлены еще до C++11, то некоторые старые компиляторы предоставляют доступ к ним через подключение заголовочного файла stdint.h.

Если ваш компилятор не поддерживает cstdint или stdint.h, то вы можете скачать кроссплатформенный заголовочный файл **pstdint.h**. Просто подключите его к вашему проекту, и он самостоятельно определит целочисленные типы фиксированного размера для вашей системы/архитектуры.

Предупреждение насчет std::int8_t и std::uint8_t

По определенным причинам в C++ большинство компиляторов определяют и обрабатывают типы int8_t и uint8_t идентично типам char signed и char unsigned (соответственно), но это происходит далеко не во всех случаях. Следовательно, std::cin и std::cout могут работать не так, как вы ожидаете. Например:

```
1. #include <iostream>
2. #include <cstdint>
3.
4. int main()
```

```
{
    std::int8_t myint = 65;
    std::cout << myint << std::endl;

    return 0;
}
```

На большинстве компьютеров с различными архитектурами результат выполнения этой программы следующий:

```
A
```

Т.е. программа, приведенная выше, обрабатывает `myint` как переменную типа char. Однако на некоторых компьютерах результат может быть следующим:

```
65
```

Поэтому идеальным вариантом будет избегать использования std::int8_t и std::uint8_t вообще (используйте вместо них std::int16_t или std::uint16_t). Однако, если вы все же используете std::int8_t или std::uint8_t, то будьте осторожны с любой функцией, которая может интерпретировать std::int8_t или std::uint8_t как символьный тип, вместо целочисленного (например, с объектами std::cin и std::cout).

Правило: Избегайте использования std::int8_t и std::uint8_t. Если вы используете эти типы, то будьте внимательны, так как в некоторых случаях они могут быть обработаны как тип char.

Недостатки целочисленных типов фиксированного размера

Целочисленные типы фиксированного размера могут не поддерживаться на определенных архитектурах (где они не имеют возможности быть представлены). Также эти типы могут быть менее производительными, чем фундаментальные типы данных, на определенных архитектурах.

Спор насчет unsigned

Многие разработчики (и даже большие организации) считают, что программисты должны избегать использования целочисленных типов unsigned вообще. Главная причина - непредсказуемое поведение и результаты, которые могут возникнуть при "смешивании" целочисленных типов signed и unsigned в программе.

Рассмотрим следующий фрагмент кода:

```
void doSomething(unsigned int x)
```

```
2. {
3.     // Выполнение некоего кода x раз
4. }
5.
6. int main()
7. {
8.     doSomething(-1);
9. }
```

Что произойдет в этом случае? `-1` преобразуется в другое большое число (скорее всего в `4 294 967 295`). Но самое грустное в этом то, что предотвратить это мы не сможем. Язык C++ свободно конвертирует числа с типами unsigned в типы signed и наоборот без проверки диапазона допустимых значений определенного типа данных. А это, в свою очередь, может привести к переполнению.

Бьёрн Страуструп, создатель языка C++, считает, что: "Использовать тип unsigned (вместо signed) для получения еще одного бита для представления положительных целых чисел, почти никогда не является хорошей идеей".

Это не означает, что вы должны избегать использования типов unsigned вообще - нет. Но если вы их используете, то используйте только там, где это действительно имеет смысл, а также позаботьтесь о том, чтобы не допустить "смешивания" типов unsigned с типами signed (как в вышеприведенном примере).

Урок №36. Типы данных с плавающей точкой: float, double и long double

Целочисленные типы данных отлично подходят для работы с целыми числами, но есть ведь еще и дробные числа. И тут нам на помощь приходит **тип данных с плавающей точкой** (или ***"тип данных с плавающей запятой"***, англ. ***"floating point"***). Переменная такого типа может хранить любые действительные дробные значения, например: 4320.0, -3.33 или 0.01226. Почему точка «плавающая»? Дело в том, точка/запятая перемещается (*«плавает»*) между цифрами, разделяя целую и дробную части значения.

Есть три типа данных с плавающей точкой: **float**, **double** и **long double**. Язык C++ определяет только их минимальный размер (как и с целочисленными типами). Типы данных с плавающей точкой всегда являются signed (т.е. могут хранить как положительные, так и отрицательные числа).

Категория	Тип	Минимальный размер	Типичный размер
Тип данных с плавающей точкой	float	4 байта	4 байта
	double	8 байт	8 байт
	long double	8 байт	8, 12 или 16 байт

Объявление переменных разных типов данных с плавающей точкой:

```
1. float fValue;
2. double dValue;
3. long double dValue2;
```

Если нужно использовать целое число с переменной типа с плавающей точкой, то тогда после этого числа нужно поставить разделительную точку и нуль. Это позволяет различать переменные целочисленных типов от переменных типов с плавающей запятой:

```
1. int n(5); // 5 - это целочисленный тип
```

```
2. double d(5.0); // 5.0 - это тип данных с плавающей точкой (по умолчанию
   double)
3. float f(5.0f); // 5.0 - это тип данных с плавающей точкой ("f" от "float")
```

Обратите внимание, литералы типа с плавающей точкой по умолчанию относятся к типу double. `f` в конце числа означает тип float.

Экспоненциальная запись

Экспоненциальная запись очень полезна для написания длинных чисел в краткой форме. Числа в экспоненциальной записи имеют следующий вид: **`мантисса × 10`$^{\textbf{\textit{экспонент}}}$**. Например, рассмотрим выражение `1.2 × 10`^{4}. Значение `1.2` — это **мантисса** (или ***"значащая часть числа"***), а `4` — это **экспонент** (или ***"порядок числа"***). Результатом этого выражения является значение 12000.

Обычно, в экспоненциальной записи, в целой части находится только одна цифра, все остальные пишутся после разделительной точки (в дробной части).

Рассмотрим массу Земли. В десятичной системе счисления она представлена как `5973600000000000000000000 кг`. Согласитесь, очень большое число (даже слишком большое, чтобы поместиться в целочисленную переменную размером 8 байт). Это число даже трудно читать (там 19 или 20 нулей?). Но используя экспоненциальную запись, массу Земли можно представить, как `5.9736 × 10`^{24}`кг` (что гораздо легче воспринимается, согласитесь). Еще одним преимуществом экспоненциальной записи является сравнение двух очень больших или очень маленьких чисел — для этого достаточно просто сравнить их экспоненты.

В языке C++ буква `e`/`E` означает, что число 10 нужно возвести в степень, которая следует за этой буквой. Например, `1.2 × 10`^{4} эквивалентно `1.2e4`, значение `5.9736 × 10`^{24} еще можно записать как `5.9736e24`.

Для чисел меньше единицы экспонент может быть отрицательным. Например, `5e-2` эквивалентно `5 * 10`$^{-2}$, что, в свою очередь, означает `5 / 10`^{2} или `0.05`. Масса электрона равна `9.1093822e-31 кг`.

На практике экспоненциальная запись может использоваться в операциях присваивания следующим образом:

```
1. double d1(5000.0);
2. double d2(5e3); // другой способ присвоить значение 5000
3.
4. double d3(0.05);
5. double d4(5e-2); // другой способ присвоить значение 0.05
```

Конвертация чисел в экспоненциальную запись

Для конвертации чисел в экспоненциальную запись необходимо следовать процедуре, указанной ниже:

- Ваш экспонент начинается с нуля.
- Переместите разделительную точку (которая разделяет целую и дробную части) влево, чтобы слева от нее осталась только одна ненулевая цифра:
  - каждое перемещение точки влево увеличивает экспонент на `1`;
  - каждое перемещение точки вправо уменьшает экспонент на `1`.
- Откиньте все нули перед первой ненулевой цифрой в целой части.
- Откиньте все конечные нули в правой (дробной) части, *только если исходное число является целым (без разделительной точки)*.

Рассмотрим примеры:

```
Исходное число: 42030
Перемещаем разделительную точку на 4 цифры влево: 4.2030e4
Слева (в целой части) нет нулей: 4.2030e4
Отбрасываем конечный нуль в дробной части: 4.203e4 (4 значащие цифры)

Исходное число: 0.0078900
Перемещаем разделительную точку на 3 цифры вправо: 0007.8900e-3
Отбрасываем нули слева: 7.8900e-3
Не отбрасываем нули справа (исходное число является дробным): 7.8900e-3 (5 значащих цифр)

Исходное число: 600.410
Перемещаем разделительную точку на 2 цифры влево: 6.00410e2
Слева нет нулей: 6.00410e2
Нули справа оставляем: 6.00410e2 (6 значащих цифр)
```

Самое главное, что нужно запомнить — это то, что цифры в мантиссе (часть перед `e`) называются **значащими цифрами**. Количество значащих цифр определяет **точность** самого значения. Чем больше цифр в мантиссе, тем точнее значение.

Точность и диапазон типов с плавающей точкой

Рассмотрим дробь 1/3. Десятичное представление этого числа — 0.33333333333333... (с тройками до бесконечности). Бесконечное число требует бесконечной памяти для хранения, а у нас в запасе, как правило, 4 или 8 байт. Переменные типа с плавающей запятой могут хранить только определенное

количество значащих цифр, остальные — отбрасываются. **Точность определяется количеством значащих цифр**, которые представляют число без потери данных.

Когда мы выводим переменные типа с плавающей точкой, то точность объекта cout, по умолчанию, составляет 6. Т.е. на экране мы увидим только 6 значащих цифр, остальные - потеряются. Например:

```
#include <iostream>

int main()
{
    float f;
    f = 9.87654321f;
    std::cout << f << std::endl;
    f = 987.654321f;
    std::cout << f << std::endl;
    f = 987654.321f;
    std::cout << f << std::endl;
    f = 9876543.21f;
    std::cout << f << std::endl;
    f = 0.0000987654321f;
    std::cout << f << std::endl;
    return 0;
}
```

Результат выполнения программы:

```
9.87654
987.654
987654
9.87654e+06
9.87654e-05
```

Обратите внимание, каждое из вышеприведенных значений имеет только 6 значащих цифр (цифры перед `e`, а не перед точкой).

Также, в некоторых случаях, cout сам может выводить числа в экспоненциальной записи. В зависимости от компилятора, экспонент может быть дополнен нулями. Например, `9.87654e+06` — это то же самое, что и `9.87654e6` (просто с добавленным нулем и знаком `+`). Минимальное количество цифр экспонента определяется компилятором (Visual Studio использует 2, другие компиляторы могут использовать 3).

Также мы можем переопределить точность cout, используя **функцию std::setprecision()**, которая находится в заголовочном файле iomanip:

```
#include <iostream>
#include <iomanip> // для std::setprecision()

int main()
```

```
{
    std::cout << std::setprecision(16); // задаем точность в 16 цифр
    float f = 3.33333333333333333333333333333333333333f;
    std::cout << f << std::endl;
    double d = 3.33333333333333333333333333333333333333;
    std::cout << d << std::endl;
    return 0;
}
```

Результат выполнения программы:

```
3.3333332538604736
3.3333333333333333
```

Так как мы увеличили точность до 16, то каждая переменная выводится 16-ю цифрами. Но, как вы можете видеть, исходные числа имеют больше цифр!

Точность зависит от размера типа данных (в типе float точность меньше, чем в типе double) и от присваиваемого значения:

- **точность float**: от 6 до 9 цифр (в основном 7);
- **точность double**: от 15 до 18 цифр (в основном 16);
- **точность long double**: 15, 18 или 33 цифры (в зависимости от того, сколько байт занимает тип данных на компьютере).

Этот принцип относится не только к дробным числам, но и ко всем значениям, которые имеют слишком большое количество значащих цифр. Например:

```
#include <iostream>
#include <iomanip> // для std::setprecision()

int main()
{
    float f(123456789.0f); // переменная f имеет 10 значащих цифр
    std::cout << std::setprecision(9); // задаем точность в 9 цифр
    std::cout << f << std::endl;
    return 0;
}
```

Результат:

```
123456792
```

Но ведь `123456792` больше чем `123456789`, не так ли? Значение `123456789.0` имеет 10 значащих цифр, но точность float равна 7. Поэтому мы и получили другое число, произошла потеря данных!

Следовательно, нужно быть осторожными, когда вы используете переменные типа с плавающей точкой вместе с очень большими/очень маленькими числами, которые требуют большей точности, чем их текущий тип данных может предложить.

Диапазон и точность типов данных с плавающей точкой, согласно **стандарту IEEE 754**:

Размер	Диапазон	Точность
4 байта	от ±1.18 x 10^{-38} до ±3.4 x 10^{38}	6-9 значащих цифр (в основном 7)
8 байт	от ±2.23 x 10^{-308} до ±1.80 x 10^{308}	15-18 значащих цифр (в основном 16)
80 бит (12 байт)	от ±3.36 x 10^{-4932} до ±1.18 x 10^{4932}	18-21 значащих цифр
16 байт	от ±3.36 x 10^{-4932} до ±1.18 x 10^{4932}	33-36 значащих цифр

Может показаться немного странным, что 12-байтовая переменная типа с плавающей точкой имеет тот же диапазон, что и 16-байтовая переменная. Это потому, что они имеют одинаковое количество бит, выделенных для экспонента (только в 16-байтовой переменной точность будет выше).

Правило: Используйте по умолчанию тип double вместо типа float, так как его точность выше.

Ошибки округления

Рассмотрим дробь 1/10. В десятичной системе счисления эту дробь можно представить, как 0.1. В двоичной системе счисления эта дробь представлена в виде бесконечной последовательности — 0.00011001100110011... Именно из-за подобных разногласий в представлении чисел в разных системах счисления, у нас могут возникать проблемы с точностью. Например:

```
1. #include <iostream>
2. #include <iomanip> // для std::setprecision()
3.
```

```
4. int main()
5. {
6.     double d(0.1);
7.     std::cout << d << std::endl; // используем точность cout по умолчанию (6 ци
   фр)
8.     std::cout << std::setprecision(17);
9.     std::cout << d << std::endl;
10.    return 0;
11.}
```

Результат выполнения программы:

```
0.1
0.10000000000000001
```

Первый cout выводит `0.1` (что и ожидаемо). После того, как мы изменили для объекта cout точность вывода до 17 цифр, мы увидели, что значением переменной `d` является не совсем `0.1`! Подобное происходит из-за ограничений в количестве выделяемой памяти для переменных типа double, а также из-за необходимости "округлять" числа. По факту мы получили типичную **ошибку округления**.

Подобные ошибки могут иметь неожиданные последствия:

```
1. #include <iostream>
2. #include <iomanip> // для std::setprecision()
3.
4. int main()
5. {
6.     std::cout << std::setprecision(17);
7.
8.     double d1(1.0);
9.     std::cout << d1 << std::endl;
10.
11.    double d2(0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1); //
   должно получиться 1.0
12.    std::cout << d2 << std::endl;
13.}
```

Результат выполнения программы:

```
1
0.99999999999999989
```

Хотя мы ожидали, что `d1` и `d2` окажутся равными, но это не так. А что, если бы нам довелось сравнивать эти переменные и, исходя из результата, выполнять определенный сценарий? В таком случае ошибок нам не миновать.

Математические операции (например, сложение или умножение), как правило, только увеличивают масштаб этих ошибок. Даже если `0.1` имеет погрешность в 17-

й значащей цифре, то при выполнении операции сложения десять раз, ошибка округления переместится к 16-й значащей цифре.

nan и inf

Есть две специальные категории чисел типа с плавающей запятой:

- **inf** (или ***"бесконечность"***, от англ *"**inf**inity"*), которая может быть либо положительной, либо отрицательной.
- **nan** (или ***"не число"***, от англ *"**n**ot **a** **n**umber"*). Их есть несколько видов (обсуждать все виды здесь мы не будем).

Рассмотрим примеры на практике:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     double zero = 0.0;
6.     double posinf = 5.0 / zero; // положительная бесконечность
7.     std::cout << posinf << "\n";
8.
9.     double neginf = -5.0 / zero; // отрицательная бесконечность
10.    std::cout << neginf << "\n";
11.
12.    double nan = zero / zero; // не число (математически некорректно)
13.    std::cout << nan << "\n";
14.
15.    return 0;
16. }
```

Результат выполнения программы:

```
inf
-inf
-nan(ind)
```

`inf` означает "бесконечность", а `ind` означает "неопределенный" (от англ. *"**ind**eterminate"*). Обратите внимание, результаты вывода `inf` и `nan` зависят от компилятора/архитектуры компьютера, поэтому ваш результат выполнения вышеприведенной программы может отличаться от моего результата.

Заключение

Переменные типа с плавающей точкой отлично подходят для хранения очень больших или очень маленьких (в том числе и дробных) чисел до тех пор, пока они имеют ограниченное количество значащих цифр (не превышают точность определенного типа данных).

Переменные типа с плавающей точкой могут иметь небольшие ошибки округления, даже если точность типа не превышена. В большинстве случаев такие ошибки остаются незамеченными, так как они не столь значительны. Но следует помнить, что сравнение переменных типов с плавающей точкой может иметь неопределенные последствия/результаты (а выполнение математических операций с такими переменными может только увеличить масштаб этих ошибок).

Тест

Запишите следующие числа в экспоненциальной записи в стиле языка C++ (используя букву `e`, как символ экспонента) и определите, сколько значащих цифр имеет каждое из следующих чисел:

- `34.50`
- `0.004000`
- `123.005`
- `146000`
- `146000.001`
- `0.0000000008`
- `34500.0`

Урок №37. Логический тип данных bool

В реальной жизни перед нами очень часто возникают вопросы, на которые можно ответить однозначно: "Да" или "Нет". Яблоко является фруктом? Да! Вам нравится спаржа? Нет!

Рассмотрим утверждение: "Яблоко — это фрукт". Является ли это правдой? Да! Яблоко действительно является фруктом. Или как насчет: "Я люблю спаржу". Абсолютная ложь (тьфу!).

Подобные стейтменты, которые имеют только два возможных исхода — да/правда или нет/ложь, настолько распространены, что многие языки программирования добавили специальный тип для работы с ними — логический тип данных. В языке C++ это **логический тип данных bool** (от англ. ***"boolean"***).

Переменные логического типа данных

Логические переменные - это переменные, диапазон которых состоит только из двух возможный значений: `true` (`1`) и `false` (`0`).

Для объявления логической переменной используется **ключевое слово bool**:

```
1. bool b;
```

Инициализировать логическую переменную или выполнить операцию присваивания можно с помощью **ключевых слов true** или **false**:

```
1. bool b1 = true; // копирующая инициализация
2. bool b2(false); // прямая инициализация
3. bool b3 { true }; // uniform-инициализация (C++11)
4. b3 = false; // операция присваивания
```

Аналогично работе унарного оператора минус (`-`), с помощью которого мы можем сделать число отрицательным, с помощью логического оператора НЕ (`!`) мы можем изменить `true` на `false` и наоборот (`false` на `true`):

```
1. bool b1 = !true; // значение b1 - false
2. bool b2(!false); // значение b2 - true
```

На самом деле, логические значения не сохраняются как `true` или `false`. Они обрабатываются в виде целых чисел: вместо `true` — единица, вместо `false` — ноль.

Следовательно, если мы попытаемся вывести логические значения с помощью std::cout, то увидим либо `0`, либо `1`:

```
#include <iostream>

int main()
{
    std::cout << true << std::endl; // вместо true единица
    std::cout << !true << std::endl; // вместо !true ноль

    bool b(false);
    std::cout << b << std::endl; // b - false (0)
    std::cout << !b << std::endl; // !b - true (1)
    return 0;
}
```

Результат выполнения программы:

```
1
0
0
1
```

Если вы хотите, чтобы std::cout выводил `true` или `false` (вместо целых чисел), то тогда используйте манипулятор форматирования **std::boolalpha**:

```
#include <iostream>

int main()
{
    std::cout << true << std::endl;
    std::cout << false << std::endl;

    std::cout << std::boolalpha; // выводим логические значения как "true" или "false"

    std::cout << true << std::endl;
    std::cout << false << std::endl;
    return 0;
}
```

Результат выполнения программы:

```
1
0
true
false
```

Использование логического типа данных в ветвлениях if

Очень часто логические переменные используются в ветвлениях if.

Ветвление if выглядит следующим образом:

```
if (выражение) стейтмент1;
```

Либо:

```
if (выражение) стейтмент1;
else стейтмент2;
```

`(выражение)` еще называется **"условием"**, либо **"условным выражением"**.

В обоих случаях, если результатом условия является ненулевое значение, то выполняется `стейтмент1`. Если же результатом условия является нулевое значение, то выполняется `стейтмент2`.

Помните, что `true` - это `1` (ненулевое значение), а `false` — это `0` (нулевое значение).

Теперь рассмотрим пример в коде:

```
1. if (true) // true - это условие
2.     std::cout << "The condition is true!" << std::endl;
3. else
4.     std::cout << "The condition is false!" << std::endl;
```

Результат:

```
The condition is true!
```

Что здесь делается? Во-первых, мы начинаем с условия if, которым является логическое значение `true`, т.е. `1` (ненулевое значение), что означает, что выполняться будет `стейтмент1`.

Следующая программа работает аналогично:

```
1. bool b(false);
2. if (b)
3.     std::cout << "b is true!" << std::endl;
4. else
5.     std::cout << "b is false!" << std::endl;
```

Результат:

```
b is false!
```

Здесь, при проверке условия, переменная `b` имеет значение `false`. `false` — это `0`. Следовательно, первый стейтмент под if (который `true`) пропускается, а второй, который под else (`false`) — выполняется.

А теперь рассмотрим пример посложнее. Оператор равенства (`==`) используется для сравнения двух чисел (являются ли они равными). Оператор `==` возвращает `true`, если операнды равны и `false`, если таковыми не являются:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Enter an integer: ";
6.     int x;
7.     std::cin >> x;
8.
9.     if (x == 0)
10.        std::cout << "The value is zero" << std::endl;
11.    else
12.        std::cout << "The value is non-zero" << std::endl;
13.    return 0;
14. }
```

Результат выполнения программы:

```
Enter an integer: 4
The value is non-zero
```

Давайте разберемся, что и как здесь работает. Во-первых, мы просим пользователя ввести целое число. После этого, с помощью оператора `==`, мы проверяем, является ли пользовательское число нулевым. В вышеприведенном примере `4` не равно `0`, поэтому оператор `==` определяет условие как `false`. Следовательно, выполняется `стейтмент2` (тот, который под else), где мы выводим `The value is non-zero`.

Возвращаемые значения логического типа данных

Логические значения часто используются в качестве возвращаемых значений в функциях. Названия таких функций очень часто начинаются со слов `is` (например, `isEqual`) или `has` (например, `hasCommonDivisor`).

Рассмотрим следующий пример:

```
1. #include <iostream>
2.
3. // Возвращаем true, если x и y равны, в противном случае - возвращаем false
4. bool isEqual(int x, int y)
5. {
6.     return (x == y); // оператор == возвращает true, если x равно y, в
   противном случае - false
7. }
```

```
8.
9.  int main()
10. {
11.     std::cout << "Enter an integer: ";
12.     int x;
13.     std::cin >> x;
14.
15.     std::cout << "Enter another integer: ";
16.     int y;
17.     std::cin >> y;
18.
19.     if (isEqual(x, y))
20.         std::cout << x << " and " << y << " are equal" << std::endl;
21.     else
22.         std::cout << x << " and " << y << " are not equal" << std::endl;
23.
24.     return 0;
25. }
```

Результат выполнения программы:

```
Enter an integer: 5
Enter another integer: 5
5 and 5 are equal
```

Как это работает? Во-первых, мы указываем значения переменным `x` и `y`. Затем проверяется условие, что приводит к вызову функции isEqual(5, 5). Внутри этой функции наши два числа сравниваются между собой (`5 == 5`), что приводит к возврату значения `true` (так как `5 = 5`). Значение `true` возвращается обратно в caller. Так как условие истинно, то выполняется `стейтмент1`, который выводит `5 and 5 are equal`.

К логическим значениям нужно немного привыкнуть, но как только вы это сделаете, то сами удивитесь, насколько они удобны и просты.

Во всех примерах, приведенных выше, в наших условиях были либо логические значения (`true` или `false`), либо логические переменные, либо функции, которые возвращают логическое значение. А что произойдет, если мы не будем использовать логическое значение в условиях? Правильно! Если результатом условия будет любое ненулевое значение, то выполняться будет `стейтмент1`.

Поэтому, если попробовать сделать что-то вроде следующего:

```
1. if (4)
2.     std::cout << "hi";
3. else
4.     std::cout << "bye";
```

То результатом будет `hi`, так как `4` является ненулевым значением.

Тест

Что такое простое число? Правильно! Это целое положительное число больше единицы, которое делится без остатка либо на себя, либо на единицу. Напишите программу, которая просит пользователя ввести простое целое число, меньшее 10. Если пользователь ввел одно из следующих чисел: 2, 3, 5 или 7 — программа должна вывести `The digit is prime`, в противном случае — `The digit is not prime`.

> ***Подсказка*:** Используйте ветвление if для сравнения чисел и логические значения для отслеживания того, является ли пользовательское число простым или нет.

Урок №38. Символьный тип данных char

Хоть тип char и относится к целочисленным типам данных (и, таким образом, следует всем их правилам), работа с char несколько отличается от работы с обычными целочисленными типами.

Тип данных char

Переменная типа char занимает 1 байт. Однако вместо конвертации значения типа char в целое число, оно *интерпретируется* как ASCII-символ.

ASCII (сокр. от *"**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange"*) - это американский стандартный код для обмена информацией, который определяет способ представления символов английского языка (+ несколько других) в виде чисел от 0 до 127. Например: код буквы `'a'` - 97, код буквы `'b'` - 98. Символы всегда помещаются в одинарные кавычки.

Таблица ASCII-символов:

Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NUL (null)	32	(space)	64	@	96	`
1	SOH (start of header)	33	!	65	A	97	a
2	STX (start of text)	34	”	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f

7	BEL (bell)	39	’	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	HT (horizontal tab)	41	)	73	I	105	i
10	LF (line feed/new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (form feed / new page)	44	,	76	L	108	l
13	CR (carriage return)	45	—	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (data control 1)	49	1	81	Q	113	q
18	DC2 (data control 2)	50	2	82	R	114	r
19	DC3 (data control 3)	51	3	83	S	115	s
20	DC4 (data control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v

23	ETB (end of transmission block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL (delete)

Символы от 0 до 31 в основном используются для форматирования вывода. Большинство из них уже устарели.

Символы от 32 до 127 используются для вывода. Это буквы, цифры, знаки препинания, которые большинство компьютеров использует для отображения текста (на английском языке).

Следующие два стейтмента выполняют одно и то же (присваивают переменным типа char целое число `97`):

```
1. char ch1(97); // инициализация переменной типа char целым числом 97
2. char ch2('a'); // инициализация переменной типа char символом 'a' (97)
```

Будьте внимательны при использовании фактических чисел с числами, которые используются для представления символов (из ASCII-таблицы). Следующие два стейтмента выполняют не одно и то же:

```
1. char ch(5); // инициализация переменной типа char целым числом 5
2. char ch('5'); // инициализация переменной типа char символом '5' (53)
```

Вывод символов

При выводе переменных типа char, объект cout выводит символы вместо цифр:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     char ch(97); // несмотря на то, что мы инициализируем переменную ch целым ч
   ислом
6.     std::cout << ch << std::endl; // cout выводит символ
7.     return 0;
8. }
```

Результат:

```
a
```

Также вы можете выводить литералы типа char напрямую:

```
1. std::cout << 'b' << std::endl;
```

Результат:

```
b
```

Оператор static_cast

Если вы хотите вывести символы в виде цифр, а не в виде букв, то вам нужно сообщить cout выводить переменные типа char в виде целочисленных значений. Не очень хороший способ это сделать - присвоить переменной типа int переменную типа char и вывести её:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     char ch(97);
6.     int i(ch); // присваиваем значение переменной ch переменной типа int
7.     std::cout << i << std::endl; // выводим значение переменной типа int
8.     return 0;
9. }
```

Результат:

```
97
```

Лучшим способом является конвертация переменной из одного типа данных в другой с помощью **оператора static_cast**.

Синтаксис static_cast выглядит следующим образом:

```
static_cast<новый_тип_данных>(выражение)
```

Оператор static_cast принимает значение из `(выражения)` в качестве входных данных и конвертирует его в указанный вами `<новый_тип_данных>`.

Пример использования оператора static_cast для конвертации типа char в тип int:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     char ch(97);
6.     std::cout << ch << std::endl;
7.     std::cout << static_cast<int>(ch) << std::endl;
8.     std::cout << ch << std::endl;
9.     return 0;
10.}
```

Результат выполнения программы:

```
a
97
a
```

Запомните, static_cast принимает `(выражение)` в качестве входных данных. Если мы используем переменную в `(выражении)`, то эта переменная изменяет свой тип только в стейтменте с оператором static_cast. Процесс конвертации никак не влияет на исходную переменную с её значением! В вышеприведенном примере, переменная `ch` остается переменной типа char с прежним значением, чему является подтверждением последний стейтмент с cout.

Также в static_cast нет никакой проверки по диапазону, так что если вы попытаетесь использовать числа, которые будут слишком большие или слишком маленькие для конвертируемого типа, то произойдет переполнение.

Более подробно о static_cast мы еще поговорим на соответствующем уроке.

Ввод символов

Следующая программа просит пользователя ввести символ. Затем она выводит этот символ и его ASCII-код:

```
1. #include <iostream>
2.
3. int main()
4. {
```

```
    std::cout << "Input a keyboard character: ";

    char ch;
    std::cin >> ch;
    std::cout << ch << " has ASCII code " << static_cast<int>(ch) << std::endl;

    return 0;
}
```

Результат выполнения программы:

```
Input a keyboard character: q
q has ASCII code 113
```

Обратите внимание, даже если cin позволит вам ввести несколько символов, переменная `ch` будет хранить только первый символ (именно он и помещается в переменную). Остальная часть пользовательского ввода останется во входном буфере, который использует cin, и будет доступна для использования последующим вызовам cin.

Рассмотрим это всё на практике:

```
#include <iostream>

int main()
{
    std::cout << "Input a keyboard character: "; // предположим, что пользователь ввел abcd

    char ch;
    std::cin >> ch; // ch = 'a', "bcd" останется во входном буфере
    std::cout << ch << " has ASCII code " << static_cast<int>(ch) << std::endl;

    // Обратите внимание, следующий cin не просит пользователя что-либо ввести, данные берутся из входного буфера!
    std::cin >> ch; // ch = 'b', "cd" останется в буфере
    std::cout << ch << " has ASCII code " << static_cast<int>(ch) << std::endl;

    return 0;
}
```

Результат выполнения программы:

```
Input a keyboard character: abcd
a has ASCII code 97
b has ASCII code 98
```

Размер, диапазон и знак типа char

В языке C++ для переменных типа char всегда выделяется 1 байт. По умолчанию, char может быть как signed, так и unsigned (хотя обычно signed). Если вы используете char для хранения ASCII-символов, то вам не нужно указывать знак переменной (поскольку signed и unsigned могут содержать значения от 0 до 127).

Но если вы используете тип char для хранения небольших целых чисел, то тогда следует уточнить знак. Переменная типа char signed может хранить числа от -128 до 127. Переменная типа char unsigned имеет диапазон от 0 до 255.

Управляющие символы

В языке C++ есть **управляющие символы** (или ***"escape-последовательности"***). Они начинаются с бэкслеша (`\`), а затем следует определенная буква или цифра.

Наиболее распространенным управляющим символов в языке C++ является `'\n'`, который обозначает символ новой строки:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "First line\nSecond line" << std::endl;
6.     return 0;
7. }
```

Результат:

```
First line
Second line
```

Еще одним часто используемым управляющим символом является `\t`, который заменяет клавишу TAB, вставляя большой отступ:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "First part\tSecond part";
6.     return 0;
7. }
```

Результат:

```
First part        Second part
```

Таблица всех управляющих символов в языке C++:

Название	Символ	Значение
Предупреждение (**a**lert)	\a	Предупреждение (звуковой сигнал)
Backspace	\b	Перемещение курсора на одну позицию назад
formfeed	\f	Перемещение курсора к следующей логической странице
Символ новой строки (**n**ewline)	\n	Перемещение курсора на следующую строку
Возврат каретки (carriage **r**eturn)	\r	Перемещение курсора в начало строки
Горизонтальный таб (horizontal **t**ab)	\t	Вставка горизонтального TAB-а
Вертикальный таб (**v**ertical tab)	\v	Вставка вертикального TAB-а
Одинарная кавычка	\’	Вставка одинарной кавычки (или апострофа)
Двойная кавычка	\”	Вставка двойной кавычки
Бэкслеш	\\	Вставка обратной косой черты (бэкслэша)
Вопросительный знак	\?	Вставка знака вопроса
Восьмеричное число	\(number)	Перевод числа из восьмеричной системы счисления в тип char

Шестнадцатеричное число	\x(number)	Перевод числа из шестнадцатеричной системы счисления в тип char

Рассмотрим пример в коде:

```
#include <iostream>

int main()
{
    std::cout << "\"This is quoted text\"\n";
    std::cout << "This string contains a single backslash \\" << std::endl;
    std::cout << "6F in hex is char \'\x6F\'" << std::endl;
    return 0;
}
```

Результат выполнения программы:

```
"This is quoted text"
This string contains a single backslash \
6F in hex is char 'o'
```

Что использовать: '\n' или std::endl?

Вы могли заметить, что в последнем примере мы использовали `'\n'` для перемещения курсора на следующую строку. Но мы могли бы использовать и `std::endl`. Какая между ними разница? Сейчас разберемся.

При использовании std::cout, данные для вывода могут помещаться в буфер, т.е. std::cout может не отправлять данные сразу же на вывод. Вместо этого он может оставить их *при себе* на некоторое время (в целях улучшения производительности).

И `'\n'`, и `std::endl` оба переводят курсор на следующую строку. Только `std::endl` еще гарантирует, что все данные из буфера будут выведены, перед тем, как продолжить.

Так когда же использовать `'\n'`, а когда `std::endl`?

- Используйте `std::endl`, когда нужно, чтобы ваши данные выводились сразу же (например, при записи в файл или при обновлении индикатора состояния какого-либо процесса). Обратите внимание, это может повлечь за собой незначительное снижение производительности, особенно если запись на устройство происходит медленно (например, запись файла на диск).
- Используйте `'\n'` во всех остальных случаях.

Другие символьные типы: wchar_t, char16_t и char32_t

Тип wchar_t следует избегать практически во всех случаях (кроме тех, когда происходит взаимодействие с Windows API).

Так же, как и стандарт ASCII использует целые числа для представления символов английского языка, так и другие кодировки используют целые числа для представления символов других языков. Наиболее известный стандарт (после ASCII) - **Unicode**, который имеет в запасе более 110 000 целых чисел для представления символов из разных языков.

Существуют следующие кодировки Unicode:

- **UTF-32** — требует 32 бита для представления символа.
- **UTF-16** — требует 16 бит для представления символа.
- **UTF-8** — требует 8 бит для представления символа.

Типы char16_t и char32_t были добавлены в C++11 для поддержки 16-битных и 32-битных символов Unicode (8-битные символы и так поддерживаются типом char).

В чём разница между одинарными и двойными кавычками при использовании с символами?

Как вы уже знаете, символы всегда помещаются в одинарные кавычки (например, `'a'`, `'+'`, `'5'`). Переменная типа char представляет только один символ (например, буква `a`, символ `+`, число `5`). Следующий стейтмент не является корректным:

```
1. char ch('56'); // переменная типа char может хранить только один символ
```

Текст, который находится в двойных кавычках, называется строкой (например, `"Hello, world!"`). **Строка** (тип **string**) — это набор последовательных символов.

Вы можете использовать литералы типа string в коде:

```
1. std::cout << "Hello, world!"; // "Hello, world!" - это литерал типа string
```

Более подробно о типе string мы поговорим на соответствующем уроке.

Урок №39. Литералы и магические числа

В языке C++ есть два вида констант: литеральные и символьные. На этом уроке мы рассмотрим литеральные константы.

Литеральные константы

Литеральные константы (или просто ***"литералы"***) — это значения, которые вставляются непосредственно в код. Поскольку они являются константами, то их значения изменить нельзя. Например:

```
1. bool myNameIsAlex = true; // true - это литеральная константа типа bool
2. int x = 5; // 5 - это литеральная константа типа int
3. int y = 2 * 3; // 2 и 3 - это литеральные константы типа int
```

С литералами типов bool и int всё понятно, а вот для литералов типа с плавающей точкой есть два способа объявления:

```
1. double pi = 3.14159; // 3.14159 - это литерал типа double
2. double avogadro = 6.02e23; // число avogadro - 6.02 x 10^23
```

Во втором способе объявления, число после экспонента может быть и отрицательным:

```
1. double electron = 1.6e-19; // заряд электрона - 1.6 x 10^-19
```

Числовые литералы могут иметь суффиксы, которые определяют их типы. Эти суффиксы не являются обязательными, так как компилятор понимает из контекста, константу какого типа данных вы хотите использовать.

Тип данных	Суффикс	Значение
int	u или U	unsigned int
int	l или L	long
int	ul, uL, Ul, UL, lu, lU, Lu или LU	unsigned long
int	ll или LL	long long

int	ull, uLL, Ull, ULL, llu, llU, LLu или LLU	unsigned long long
double	f или F	float
double	l или L	long double

Суффиксы есть даже для целочисленных типов (но они почти не используются):

```
1. unsigned int nValue = 5u; // тип int unsigned
2. long nValue2 = 5L; // тип long
```

По умолчанию литеральные константы типа с плавающей точкой являются типа double. Для конвертации литеральных констант в тип float можно использовать суффикс `f` или `F`:

```
1. float fValue = 5.0f; // тип float
2. double d = 6.02e23; // тип double (по умолчанию)
```

Язык C++ также поддерживает литералы типов string и char:

```
1. char c = 'A'; // 'A' - это литерал типа char
2. std::cout << "Hello, world!"; // "Hello, world!" - это литерал строки C-style
3. std::cout << "Hello," " world!"; // C++ связывает последовательные литералы
   типа string
```

Литералы хорошо использовать в коде до тех пор, пока их значения понятны и однозначны. Это выполнение операций присваивания, математических операций или вывода текста в консоль.

Литералы в восьмеричной и шестнадцатеричной системах счисления

В повседневной жизни мы используем **десятичную систему счисления**, которая состоит из десяти цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. По умолчанию язык C++ использует десятичную систему счисления для чисел в программах:

```
1. int x = 12; // предполагается, что 12 является числом десятичной системы
   счисления
```

В **двоичной (бинарной) системе счисления** всего 2 цифры: 0 и 1. Значения: 0, 1, 10, 11, 100, 101, 110, 111 и т.д.

Есть еще две другие системы счисления: восьмеричная и шестнадцатеричная.

Восьмеричная система счисления состоит из 8 цифр: 0, 1, 2, 3, 4, 5, 6 и 7. Значения: 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12 и т.д.

> ***Примечание***: В восьмеричной системе счисления нет цифр 8 и 9, так что сразу перескакиваем от 7 к 10.

Десятичная система счисления	0	1	2	3	4	5	6	7	8	9	10	11
Восьмеричная система счисления	0	1	2	3	4	5	6	7	10	11	12	13

Для использования литерала из восьмеричной системы счисления, используйте префикс `0` (ноль):

```
#include <iostream>

int main()
{
    int x = 012; // 0 перед значением означает, что это восьмеричный литерал
    std::cout << x;
    return 0;
}
```

Результат выполнения программы:

```
10
```

Почему 10 вместо 12? Потому что std::cout выводит числа в десятичной системе счисления, а 12 в восьмеричной системе = 10 в десятичной.

Восьмеричная система счисления используется крайне редко.

Шестнадцатеричная система счисления состоит из 16 символов: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Десятичная система	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Шестнадцатеричная система	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11

Для использования литерала из шестнадцатеричной системы счисления, используйте префикс `0x`:

```
#include <iostream>

int main()
{
    int x = 0xF; // 0x перед значением означает, что это шестнадцатеричный литерал
    std::cout << x;
    return 0;
}
```

Результат выполнения программы:

```
15
```

Поскольку в этой системе 16 символов, то одна шестнадцатеричная цифра занимает 4 бита. Следовательно, две шестнадцатеричные цифры занимают 1 байт.

Рассмотрим 32-битное целое число из двоичной системы счисления: `0011 1010 0111 1111 1001 1000 0010 0110`. Из-за длины и повторения цифр его сложно прочесть. В шестнадцатеричной системе счисления это же значение будет выглядеть следующим образом: `3A7F 9826`. Такой удобный/сжатый формат является преимуществом шестнадцатеричной системы счисления, поэтому шестнадцатеричные значения часто используются для представления адресов памяти или необработанных значений в памяти.

До C++14 использовать литерал из двоичной системы счисления было невозможно. Тем не менее, шестнадцатеричная система счисления может нам в этом помочь:

```
#include <iostream>

int main()
{
    int bin(0);
    bin = 0x01; // присваиваем переменной бинарный литерал 0000 0001
    bin = 0x02; // присваиваем переменной бинарный литерал 0000 0010
    bin = 0x04; // присваиваем переменной бинарный литерал 0000 0100
    bin = 0x08; // присваиваем переменной бинарный литерал 0000 1000
    bin = 0x10; // присваиваем переменной бинарный литерал 0001 0000
    bin = 0x20; // присваиваем переменной бинарный литерал 0010 0000
    bin = 0x40; // присваиваем переменной бинарный литерал 0100 0000
    bin = 0x80; // присваиваем переменной бинарный литерал 1000 0000
    bin = 0xFF; // присваиваем переменной бинарный литерал 1111 1111
    bin = 0xB3; // присваиваем переменной бинарный литерал 1011 0011
    bin = 0xF770; // присваиваем переменной бинарный литерал 1111 0111 0111 0000

    return 0;
}
```

Бинарные литералы и разделитель цифр в C++14

В C++14 мы можем использовать бинарные (двоичные) литералы, добавляя префикс `0b`:

```
#include <iostream>

int main()
{
    int bin(0);
    bin = 0b1; // присваиваем переменной бинарный литерал 0000 0001
    bin = 0b11; // присваиваем переменной бинарный литерал 0000 0011
    bin = 0b1010; // присваиваем переменной бинарный литерал 0000 1010
    bin = 0b11110000; // присваиваем переменной бинарный литерал 1111 0000

    return 0;
}
```

Поскольку длинные литералы читать трудно, то в C++14 добавили возможность использовать одинарную кавычку `'` в качестве разделителя цифр:

```
#include <iostream>

int main()
{
    int bin = 0b1011'0010; // присваиваем переменной бинарный литерал 1011 0010

    long value = 2'532'673'462; // намного проще читать, нежели 2532673462

    return 0;
}
```

Если ваш компилятор не поддерживает C++14, то использовать бинарные литералы и разделитель цифр вы не сможете — компилятор выдаст ошибку.

Магические числа. Что с ними не так?

Рассмотрим следующий фрагмент кода:

```
int maxStudents = numClassrooms * 30;
```

В вышеприведенном примере число 30 является магическим числом. **Магическое число** — это хорошо закодированный литерал (обычно, число) в строке кода, который не имеет никакого контекста. Что это за 30, что оно означает/обозначает? Хотя из вышеприведенного примера можно догадаться, что число 30 обозначает максимальное количество учеников, находящихся в одном кабинете - в большинстве случаев, это не будет столь очевидным и понятным. В более сложных программах контекст подобных чисел разгадать намного сложнее (если только не будет соответствующих комментариев).

Использование магических чисел является плохой практикой, так как в дополнение к тому, что они не предоставляют никакого контекста (для чего и почему используются), они также могут создавать проблемы, если их значения необходимо будет изменить. Предположим, что школа закупила новые парты. Эта покупка, соответственно, увеличила максимально возможное количество учеников, находящихся в одном кабинете, с 30 до 36 - это нужно будет продумать и отобразить в нашей программе.

Рассмотрим следующий фрагмент кода:

```
1. int maxStudents = numClassrooms * 30;
2. setMax(30);
```

Чтобы обновить число учеников в кабинете, нам нужно изменить значение константы с 30 на 36. Но что делать с вызовом функции setMax(30)? Аргумент 30 и константа 30 в коде, приведенном выше, являются одним и тем же, верно? Если да, то нам нужно будет обновить это значение. Если нет, то нам не следует вообще трогать этот вызов функции. Если же проводить автоматический глобальный поиск и замену числа 30, то можно ненароком изменить и аргумент функции setMax(), в то время, когда его вообще не следовало бы трогать. Поэтому вам придется просмотреть весь код "вручную", в поисках числа 30, а затем в каждом конкретном случае определить — изменить ли 30 на 36 или нет. Это может занять очень много времени, кроме того, вероятность возникновения новых ошибок повышается в разы.

К счастью, есть лучший вариант — использовать символьные константы. О них мы поговорим на следующем уроке.

Правило: Старайтесь свести к минимуму использование магических чисел в ваших программах.

Урок №40. const, constexpr и символьные константы

До этого момента, все переменные, которые мы рассматривали, были *обычными.* Их значения можно было изменить в любое время, например:

```
1. int x { 4 }; // инициализация переменной x значением 4
2. x = 5; // изменяем значение x на 5
```

Тем не менее, иногда полезно использовать переменные, значения которых изменить нельзя — **константы**.

Константы

Возьмем к примеру величину силы тяжести на Земле: `9.8м/с^2`. Она вряд ли поменяется в ближайшее время. Использовать константу в этом случае будет наилучшим вариантом, так как мы предотвратим, таким образом, любое (даже случайное) изменение этого значения.

Чтобы сделать переменную константой — используйте **ключевое слово const** перед типом переменной или после него. Например:

```
1. const double gravity { 9.8 }; // предпочтительнее использовать const перед
   типом данных
2. int const sidesInSquare { 4 }; // ок, но вариант выше - лучше
```

Несмотря на то, что язык C++ позволяет размещать const как перед типом данных, так и после него, хорошей практикой считается размещать const *перед* типом данных.

Константы должны быть инициализированы при объявлении. Изменить их значения с помощью операции присваивания нельзя:

```
1. const double gravity { 9.8 };
2. gravity = 9.9; // не допускается - ошибка компиляции
```

Объявление константы без её инициализации также вызовет ошибку компиляции:

```
1. const double gravity; // ошибка компиляции, константа должна быть
   инициализирована
```

Обратите внимание, константы могут быть инициализированы и с помощью неконстантных значений:

```
1. std::cout << "Enter your age: ";
2. int age;
3. std::cin >> age;
```

```
4.
5. const int usersAge (age); // в дальнейшем значение переменной usersAge не может
    быть изменено
```

Ключевое слово const является наиболее полезным (и наиболее часто используемым) с параметрами функций:

```
1. void printInteger(const int myValue)
2. {
3.     std::cout << myValue;
4. }
```

Таким образом, при вызове функции константа-параметр сообщает и гарантирует нам то, что функция не изменит значение переменной `myValue`.

Время компиляции и время выполнения

Когда вы находитесь в процессе компиляции программы, то это **время компиляции** (англ. ***"compile time"***). Компилятор проверяет вашу программу на синтаксические ошибки и, если их нет, конвертирует код в объектные файлы.

Временной промежуток с момента старта выполнения программы и до момента окончания её работы называется **временем выполнения программы** (англ. ***"runtime"***). Код выполняется строка за строкой.

Спецификатор constexpr

В языке C++ есть два вида констант:

- **Константы времени выполнения**. Их значения определяются только во время выполнения программы. Переменные типа `usersAge` и `myValue` выше являются константами времени выполнения, так как компилятор не может определить их значения во время компиляции. `usersAge` зависит от пользовательского ввода (который можно получить только во время выполнения программы), а `myValue` зависит от значения, переданного в функцию (это значение также определится только во время выполнения программы).
- **Константы времени компиляции**. Их значения определяются во время компиляции программы. Например, переменная со значением силы тяжести на Земле является константой времени компиляции, так как мы её определяем во время написания программы (до начала её выполнения).

В большинстве случаев не важно какой тип константы вы используете: времени выполнения или времени компиляции. Однако, все же есть несколько ситуаций,

когда C++ может потребовать константу времени компиляции вместо времени выполнения (например, при определении длины массива фиксированного размера - мы рассмотрим это несколько позже). Так как есть 2 типа констант, то компилятору нужно постоянно отслеживать, к какому из них относится какая переменная. Чтобы упростить это задание, в C++11 добавили **спецификатор constexpr**, который сообщает компилятору, что текущая переменная является константой времени компиляции:

```
1. constexpr double gravity (9.8); // ок, значение определяется во время
   компиляции программы
2. constexpr int sum = 4 + 5; // ок, результат выражения 4 + 5 определяется во
   время компиляции программы
3.
4. std::cout << "Enter your age: ";
5. int age;
6. std::cin >> age;
7. constexpr int myAge = age; // неправильно, переменная age не определяется во
   время компиляции программы
```

Использовать его вы, скорее всего, не будете, но знать о нем не помешает.

Правило: Любая переменная, которая не должна изменять свое значение после инициализации, должна быть объявлена с помощью спецификатора const (или constexpr).

Имена констант

Некоторые программисты пишут имена констант заглавными буквами. Другие используют обычные имена, только с префиксом `k`. Мы же не будем их как-то выделять, так как константы — это те же обычные переменные, просто с фиксированными значениями, вот и всё. Особой причины их выделять — нет. Однако, это дело привычки.

Символьные константы

На предыдущем уроке мы говорили о магических числах — литералы, которые используются в программе в виде констант. "Поскольку использовать их не рекомендуется, то какой выход?" — спросите вы. А я отвечу: "Использовать символьные константы". **Символьная константа** — это тот же литерал (магическое число), только с идентификатором. Есть 2 способа объявления символьных констант в языке C++. Первый способ хороший, а второй — не очень. Рассмотрим сначала плохой способ.

Плохой способ: Использовать макросы-объект с текст_замена в качестве символьных констант.

Раньше этот способ широко использовался, так что вы все еще можете его увидеть в старых программах.

Как мы уже знаем, макросы-объекты имеют две формы: с `текст_замена` и без `текст_замена`. Рассмотрим первый вариант с `текст_замена`. Он выглядит следующим образом:

```
#define идентификатор текст_замена
```

Как только препроцессор встретит эту директиву, все дальнейшие появления `идентификатор` будут заменены на `текст_замена`. `идентификатор` обычно пишется заглавными буквами с нижним подчёркиванием вместо пробелов.

Например:

```
1. #define MAX_STUDENTS_PER_CLASS 30
2.
3. //...
4. int max_students = numClassrooms * MAX_STUDENTS_PER_CLASS;
5. //...
```

Во время компиляции программы, препроцессор заменит все идентификаторы `MAX_STUDENTS_PER_CLASS` на литерал `30`.

Согласитесь, это гораздо лучше, нежели использовать магические числа, как минимум, по нескольким причинам. `MAX_STUDENTS_PER_CLASS` дает понимание того, что это за значение и зачем оно используется (это понятно даже без комментариев). Во-вторых, если число нужно будет изменить - достаточно будет внести правки только в директиву #define, все остальные идентификаторы `MAX_STUDENTS_PER_CLASS` в программе будут автоматически заменены новым значением при повторной компиляции.

Рассмотрим еще один пример:

```
1. #define MAX_STUDENTS_PER_CLASS 30
2. #define MAX_NAME_LENGTH 30
3.
4. int max_students = numClassrooms * MAX_STUDENTS_PER_CLASS;
5. setMax(MAX_NAME_LENGTH);
```

Здесь понятно, что `MAX_STUDENTS_PER_CLASS` и `MAX_NAME_LENGTH` не являются одним и тем же объектом, хоть и имеют одинаковые значения.

Так почему же этот способ плохой? На это есть две причины:

- Во-первых, макросы обрабатываются препроцессором, который заменяет идентификаторы на определенные значения. Эти значения не отображаются в отладчике (во время отладки вашей программы). При компиляции `int max_students = numClassrooms * 30;` в отладчике вы увидите `int max_students = numClassrooms * MAX_STUDENTS_PER_CLASS;`. "А как тогда узнать значение `MAX_STUDENTS_PER_CLASS`?" — спросите вы. А я отвечу: "Вам придется самостоятельно найти это в коде". А процесс поиска может занять некоторое время (в зависимости от размеров вашей программы).
- Во-вторых, эти директивы всегда имеют глобальную область видимости (о ней мы поговорим позже). Это означает, что значения #define в одной части кода могут конфликтовать со значениями #define в другой части кода.

Правило: Не используйте директиву #define для создания символьных констант.

Хороший способ: Использовать переменные со спецификатором const.

Например:

```
1. const int maxStudentsPerClass { 30 };
2. const int maxNameLength { 30 };
```

Такие значения отображаются в отладчике, а также следуют всем правилам обычных переменных (в том числе и по области видимости).

Правило: Используйте спецификатор const для создания символьных констант.

Использование символьных констант в программе

Во многих программах символьные константы используются часто (в разных местах кода). Это могут быть физические или математические константы (например, число Пи или число Авогадро), или специфические значения в вашей программе. Чтобы не писать их каждый раз, когда они необходимы - просто определите их в одном месте и используйте везде, где они нужны. Таким образом, если вам придется изменить их значения — достаточно будет зайти в один файл и внести в него правки, а не искать эти константы по всей программе.

Алгоритм использования символьных констант в вашей программе:

- *Шаг №1:* Создайте заголовочный файл для хранения констант.
- *Шаг №2:* В заголовочном файле объявите пространство имен.

- *Шаг №3:* Добавьте все ваши константы в созданное пространство имен (убедитесь, что все константы имеют спецификатор const).
- *Шаг №4:* #include заголовочный файл везде, где нужны константы.

Например, файл constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// Определите собственное пространство имен для хранения констант
namespace constants
{
    const double pi(3.14159);
    const double avogadro(6.0221413e23);
    const double my_gravity(9.2);
    // ... другие константы
}
#endif
```

Используйте оператор разрешения области видимости (`::`) для доступа к константам в файлах .cpp:

```
#include "constants.h"

//...
double circumference = 2 * radius * constants::pi;
//...
```

Если в программе много констант разных типов, то сделайте отдельные файлы для каждого из этих типов. Таким образом, отделив математические константы от специфических значений (которые могут быть разными в разных программах), вы сможете подключать файл с математическими константами к любой другой программе.

Глава №2. Итоговый тест

Полученные знания всегда нужно применять на практике, поэтому мы сначала повторим теорию, а затем перейдем к практике.

Теория

Целочисленные типы данных используются для хранения целых чисел. Не забывайте о проблемах деления и переполнения с ними. Используйте целочисленные типы с фиксированным размером.

Типы данных с плавающей точкой используются для хранения вещественных чисел. Не забывайте о проблемах с точностью, ошибках округления и неточном сравнении чисел.

Логический тип данных содержит 2 значения: `true` и `false`.

Символьный тип данных содержит целые числа, которые могут интерпретироваться в символы, соответствующие стандарту ASCII. Будьте осторожны при использовании фактических чисел и цифр, которые используются для представления символов. Также помните о проблемах переполнения.

Используйте **спецификатор const** для объявления символьных констант вместо использования директив #define. Это безопаснее.

Задание №1

Почему символьные константы лучше литеральных (магических чисел)? Почему использование const лучше использования директив #define?

Задание №2

Выберите подходящий тип данных для переменных в каждой из следующих ситуаций. Будьте как можно более конкретными. Если ответом является целочисленный тип данных, то используйте соответствующий тип с фиксированным размером (например, int16_t). Если переменная должна быть константной, то так и отвечайте.

- Возраст пользователя.
- Нравится ли определенный цвет пользователю?
- Число Пи.

- Количество страниц в учебнике.
- Цена акций в долларах (дробь присутствует).
- Сколько раз вы моргнули за всю свою жизнь? (*Примечание:* Ответ исчисляется в миллионах)
- Пользователь выбирает опцию с помощью ввода определенной буквы.

Задание №3

Напишите следующую программу. Сначала пользователю предлагается ввести 2 числа типа с плавающей точкой (используйте тип double). Затем предлагается ввести один из следующих математических символов: `+`, `-`, `*` или `/`. Программа выполняет выбранную пользователем математическую операцию между двумя числами, а затем выводит результат на экран. Если пользователь ввел некорректный символ, то программа ничего не должна выводить. Например:

```
Enter a double value: 7
Enter a double value: 5
Enter one of the following: +, -, *, or /: *
7 * 5 = 35
```

> ***Подсказка*:** Вы можете использовать ветвление if для того, чтобы распознать, ввел ли пользователь определенный математический символ (например, `+`) или нет.

Задание №4

Это уже немного сложнее. Напишите небольшую программу-симулятор падения мячика с башни. Сначала пользователю предлагается ввести высоту башни в метрах. Не забывайте о гравитации (`9,8м/с`^{2}) и о том, что у мячика нет начальной скорости (его держат в руках). Программа должна выводить расстояние от земли, на котором находится мячик после 0, 1, 2, 3, 4 и 5 секунд падения. Минимальная высота составляет 0 метров (ниже мячику падать нельзя).

В вашей программе должен быть заголовочный файл constants.h с пространством имен `myConstants`. В `myConstants` определите символьную константу для хранения значения силы тяжести на Земле (`9.8`).

Напишите функцию для вычисления высоты мячика через `x` секунд падения. Используйте следующую формулу: *высота мячика над землей = константа_гравитации * x_секунд*^{2}*/2*.

Пример результата выполнения программы:

```
Enter the initial height of the tower in meters: 100
At 0 seconds, the ball is at height: 100 meters
At 1 seconds, the ball is at height: 95.1 meters
At 2 seconds, the ball is at height: 80.4 meters
At 3 seconds, the ball is at height: 55.9 meters
At 4 seconds, the ball is at height: 21.6 meters
At 5 seconds, the ball is on the ground.
```

> ***Примечания:***
>
> - В зависимости от начальной высоты, мячик может и не достичь земли в течение 5 секунд — это нормально. Мы усовершенствуем эту программу, когда будем рассматривать циклы.
> - Символ `^` не является экспонентом в языке C++. В формуле вместо него используйте знак умножения `*`.

Урок №41. Приоритет операций и правила ассоциативности

Чтобы правильно вычислять выражения (например, `4 + 2 * 3`), мы должны знать, что делают определенные операторы и в каком порядке они выполняются. Последовательность, в которой они выполняются, называется **приоритетом операций**. Следуя обычным правилам математики (в которой умножение следует перед сложением), выражение, приведенное выше в данном абзаце, как пример, обрабатывается следующим образом: `4 + (2 * 3) = 10`.

В языке C++ все операторы (операции) имеют свой уровень приоритета. Те, в которых он выше, выполняются первыми. В таблице, приведенной ниже, можно увидеть, что приоритет операций умножения и деления (5) выше, чем в операциях сложения и вычитания (6). Компилятор использует приоритет операторов для определения порядка обработки выражений.

А что делать, если у двух операторов в выражении одинаковый уровень приоритета, и они размещены рядом? Какую операцию компилятор выполнит первой? А здесь уже компилятор будет использовать **правила ассоциативности**, которые указывают направление выполнения операций: слева направо или справа налево. Например, в выражении `3 * 4 / 2` операции умножения и деления имеют одинаковый уровень приоритета (5-й уровень). А ассоциативность пятого уровня соответствует выполнению операций слева направо, таким образом: `(3 * 4) / 2 = 6`.

Таблица приоритета и ассоциативности операций

> ***Несколько примечаний***:
>
> - 1 означает самый высокий уровень приоритета, а 17 — самый низкий. Операции с более высоким уровнем приоритета выполняются первыми.
> - `L -> R` означает слева направо.
> - `R -> L` означает справа налево.

Ассоциативность	Оператор	Описание	Пример
1. Нет	::	Глобальная область видимости (унарный)	::name

	::	Область видимости класса (бинарный)	class_name::member_name
2. L -> R	()	Круглые скобки	(expression)
	()	Вызов функции	function_name(parameters)
	()	Инициализация	type name(expression)
	{}	uniform-инициализация (C++11)	type name{expression}
	type()	Конвертация типа	new_type(expression)
	type{}	Конвертация типа (C++11)	new_type{expression}
	[]	Индекс массива	pointer[expression]
	.	Доступ к члену объекта	object.member_name
	->	Доступ к члену объекта через указатель	object_pointer->member_name
	++	Пост-инкремент	lvalue++
	--	Пост-декремент	Lvalue--
	typeid	Информация о типе во время выполнения	typeid(type) or typeid(expression)

3. R -> L	const_cast	Cast away const	const_cast(expression)
	dynamic_cast	Type-checked cast во время выполнения	dynamic_cast(expression)
	reinterpret_cast	Конвертация одного типа в другой	reinterpret_cast(expression)
	static_cast	Type-checked cast во время компиляции	static_cast(expression)
	+	Унарный плюс	+expression
	-	Унарный минус	-expression
	++	Пре-инкремент	++lvalue
	--	Пре-декремент	--lvalue
	!	Логическое НЕ (NOT)	!expression
	~	Побитовое НЕ (NOT)	~expression
	(type)	C-style cast	(new_type)expression
	sizeof	Размер в байтах	sizeof(type) or sizeof(expression)
	&	Адрес	&lvalue
	*	Разыменование	*expression

4. L -> R	new	Динамическое выделение памяти	new type
	new[]	Динамическое выделение массива	new type[expression]
	delete	Динамическое удаление памяти	delete pointer
	delete[]	Динамическое удаление массива	delete[] pointer
	->*	Member pointer selector	object_pointer->*pointer_to_member
	.*	Member object selector	object.*pointer_to_member
5. L -> R	*	Умножение	expression * expression
	/	Деление	expression / expression
	%	Деление с остатком	expression % expression
6. L -> R	+	Сложение	expression + expression
	-	Вычитание	expression - expression
7. L -> R	<<	Побитовый сдвиг влево	expression << expression
	>>	Побитовый сдвиг вправо	expression >> expression
8. L -> R	<	Сравнение: меньше чем	expression < expression

9. L -> R	<=	Сравнение: меньше чем или равно	expression <= expression
	>	Сравнение: больше чем	expression > expression
	>=	Сравнение: больше чем или равно	expression >= expression
	==	Равно	expression == expression
	!=	Не равно	expression != expression
10. L -> R	&	Побитовое И (AND)	expression & expression
11. L -> R	^	Побитовое исключающее ИЛИ (XOR)	expression ^ expression
12. L -> R	|	Побитовое ИЛИ (OR)	expression | expression
13. L -> R	&&	Логическое И (AND)	expression && expression
14. L -> R	||	Логическое ИЛИ (OR)	expression || expression
15. R -> L	?:	Тернарный условный оператор	expression ? expression : expression
	=	Присваивание	lvalue = expression
	*=	Умножение с присваиванием	lvalue *= expression

	/=	Деление с присваиванием	lvalue /= expression
	%=	Деление с остатком и с присваиванием	lvalue %= expression
	+=	Сложение с присваиванием	lvalue += expression
	-=	Вычитание с присваиванием	lvalue -= expression
	<<=	Присваивание с побитовым сдвигом влево	lvalue <<= expression
	>>=	Присваивание с побитовым сдвигом вправо	lvalue >>= expression
	&=	Присваивание с побитовой операцией И (AND)	lvalue &= expression
	|=	Присваивание с побитовой операцией ИЛИ (OR)	lvalue |= expression
	^=	Присваивание с побитовой операцией «Исключающее ИЛИ» (XOR)	lvalue ^= expression
16. R -> L	throw	Генерация исключения	throw expression

17. L -> R	,	Оператор Запятая	expression, expression

Некоторые операторы вы уже знаете из предыдущих уроков: `+`, `-`, `*`, `/`, `()`, `=`, `<` и `>`. Их значения одинаковы как в математике, так и в языке C++.

Однако, если у вас нет опыта работы с другими языками программирования, то большинство из этих операторов вам сейчас могут быть непонятны. Это нормально. Мы рассмотрим большую их часть на уроках этой главы, а об остальных расскажем по мере необходимости.

Эта таблица предназначена в первую очередь для того, чтобы вы могли в любой момент обратиться к ней для решения возможных проблем приоритета или ассоциативности.

Как возвести число в степень в C++?

Вы уже должны были заметить, что оператор `^`, который обычно используется для обозначения возведения в степень в обычной математике, не является таковым в языке C++. В языке C++ это побитовая операция XOR. А для возведения числа в степень в языке C++ используется функция pow(), которая находится в заголовочном файле cmath:

```
#include <cmath>

double x = pow(3.0, 4.0); // 3 в степени 4
```

Обратите внимание, параметры и возвращаемые значения функции pow() являются типа double. А поскольку типы с плавающей точкой известны ошибками округления, то результаты pow() могут быть неточными (чуть меньше или чуть больше).

Если вам нужно возвести в степень целое число, то лучше использовать собственную функцию, например:

```
// Примечание: Экспонент не должен быть отрицательным
int pow(int base, int exp)
{
    int result = 1;
    while (exp)
    {
        if (exp & 1)
            result *= base;
        exp >>= 1;
        base *= base;
    }

    return result;
```

```
14.}
```

Не переживайте, если здесь что-то не понятно. Просто помните о проблеме переполнения, которая может произойти, если один из аргументов будет слишком большим.

Тест

Из школьной математики нам известно, что выражения внутри скобок выполняются первыми. Например, в выражении `(2 + 3) * 4` часть `(2 + 3)` выполняется первой.

В этом задании есть 4 выражения, в которых отсутствуют какие-либо скобки. Используя приоритет операций и правила ассоциативности, приведенные выше, добавьте скобки в каждое выражение так, как если бы их обрабатывал компилятор.

> ***Подсказка*:** Используйте колонку *"Пример"* в таблице приоритета и ассоциативности операций, чтобы определить, является ли оператор унарным (имеет один операнд) или бинарным (имеет два операнда).

***Например**:* `x = 2 + 3 % 4`

Бинарный оператор `%` имеет более высокий приоритет, чем оператор `+` или `=`, поэтому он выполняется первым: `x = 2 + (3 % 4)`. Затем выполняется бинарный оператор `+`, так как он имеет более высокий приоритет, чем оператор `=`.

Ответ: `x = (2 + (3 % 4))`.

Дальше нам уже не нужна таблица, чтобы понять ход обработки этого выражения компилятором.

Задания:

- *Выражение №1:* `x = 3 + 4 + 5`
- *Выражение №2:* `x = y = z`
- *Выражение №3:* `z *= ++y + 5`
- *Выражение №4:* `a || b && c || d`

Урок №42. Арифметические операторы

Существуют два унарных арифметических оператора: плюс (`+`) и минус (`–`). Унарные операторы — это операторы, которые применяются только к одному операнду.

Оператор	Символ	Пример	Операция
Унарный плюс	+	+x	Значение x
Унарный минус	–	–x	Отрицательное значение x

Унарный оператор `+` возвращает значение операнда. Другими словами, `+5 = 5` или `+x = x`. Унарный плюс вам, скорее всего, не придется использовать. Его по большей части добавили в качестве симметрии с унарным оператором минус. Унарный оператор минус возвращает операнд, умноженный на `-1`. Например, если `x = 5`, то `-x = -5`.

Оба этих оператора пишутся непосредственно перед самим операндом, без пробела (`-x`, а не `- x`).

Не следует путать унарный оператор минус с бинарным оператором вычитания, хоть они и используют один и тот же символ. Например, в выражении `x = 5 --3;`, первый минус — это оператор вычитания, а второй — унарный минус.

Бинарные арифметические операторы

Бинарные операторы — это операторы, которые применяются к двум операндам (слева и справа). Существует 5 бинарных операторов.

Оператор	Символ	Пример	Операция
Сложение	+	x + y	x плюс y
Вычитание	–	x – y	x минус y
Умножение	*	x * y	x умножить на y

Деление	/	x / y	x разделить на y
Деление с остатком	%	x % y	Остаток от деления x на y

Операторы сложения, вычитания и умножения работают так же, как и в обычной математике. А вот деление и деление с остатком рассмотрим детально.

Деление целых чисел и чисел типа с плавающей точкой

Оператор деления имеет два режима. Если оба операнда являются целыми числами, то оператор выполняет целочисленное деление. Т.е. любая дробь (больше/меньше) отбрасывается и возвращается целое значение без остатка, например, `7 / 4 = 1`.

Если один или оба операнда типа с плавающей точкой, то тогда будет выполняться деление типа с плавающей точкой. Здесь уже дробь присутствует. Например, выражения `7.0 / 3 = 2.333`, `7 / 3.0 = 2.333` или `7.0 / 3.0 = 2.333` имеют один и тот же результат.

Попытки деления на `0` (или на `0.0`) станут причиной сбоя в вашей программе, и это правило не следует забывать!

Использование оператора static_cast в операциях деления

Ранее мы уже использовали оператор static_cast для вывода ASCII-символов в виде целых чисел.

Аналогичным образом мы можем использовать static_cast для конвертации целого числа в число типа с плавающей точкой. Таким образом, вместо целочисленного деления выполнится деление типа с плавающей точкой. Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int x = 7;
6.     int y = 4;
7.
8.     std::cout << "int / int = " << x / y << "\n";
9.     std::cout << "double / int = " << static_cast<double>(x) / y << "\n";
10.    std::cout << "int / double = " << x / static_cast<double>(y) << "\n";
11.    std::cout << "double / double = " << static_cast<double>(x) / static_cast<d
   ouble>(y) << "\n";
12.
13.    return 0;
```

```
14. }
```

Результат выполнения программы:

```
int / int = 1
double / int = 1.75
int / double = 1.75
double / double = 1.75
```

Деление с остатком

Оператор деления с остатком (`%`) работает только с целочисленными операндами и возвращает остаток от целочисленного деления. Например:

- *Пример №1:* `7 / 4 = 1` с остатком `3`, таким образом, `7 % 4 = 3`.
- *Пример №2:* `25 / 7 = 3` с остатком `4`, таким образом, `25 % 7 = 4`. Остаток составляет не дробь, а целое число.
- *Пример №3:* `36 % 5 = 7` с остатком `1`. В числе 36 только 35 делится на 5 без остатка, поэтому `36 - 35 = 1`, `1` — это остаток и результат.

Данный оператор чаще всего используют для проверки деления без остатка одних чисел на другие. Если `x % y == 0`, то `x` делится на `y` без остатка.

Например, мы хотим написать программу, которая выводит числа от 1 до 100 по 20 значений в каждой строке. Мы можем использовать оператор деления с остатком для создания разрыва строк. Несмотря на то, что мы еще не рассматривали цикл while, в следующей программе всё максимально просто и понятно:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     // Переменная count хранит текущее число для вывода
6.     int count = 1; // начинаем с 1
7.
8.     // Повторение операции (цикл) до тех пор, пока count не будет равен 100
9.     while (count <= 100)
10.    {
11.        std::cout << count << " "; // вывод текущего числа
12.
13.        // Если count делится на 20 без остатка, то вставляем разрыв строки и
   продолжаем с новой строки
14.        if (count % 20 == 0)
15.            std::cout << "\n";
16.
17.        count = count + 1; // переходим к следующему числу
18.    } // конец while
19.
20.    return 0;
21. } // конец main()
```

Результат выполнения программы:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

О while мы еще поговорим на соответствующем уроке.

Отрицательные числа в операциях деления до C++11

До C++11, если любой из операндов целочисленного деления является отрицательным, то компилятор округляет результат самостоятельно! Например, результатом `-5 / 2` может быть либо `-3`, либо `-2`. Однако большинство современных компиляторов округляют числа в сторону нуля (например, в `-5 / 2` результатом будет `-2`). В спецификации C++11 определили, что компилятор должен всегда округлять к нулю (или, проще говоря, просто отбрасывать дробь).

Также до C++11, если один из операндов оператора деления с остатком является отрицательным, то результат может быть как положительным, так и отрицательным! Например, результатом `-5 % 2` может быть как `1`, так и `-1`. В спецификации C++11 решили сделать так, чтобы результат `a % b` был того же знака, что и значение `a`.

Арифметические операторы присваивания

Оператор	Символ	Пример	Операция
Присваивание	=	x = y	Присваиваем значение y переменной x
Сложение с присваиванием	+=	x += y	Добавляем y к x
Вычитание с присваиванием	-=	x -= y	Вычитаем y из x
Умножение с присваиванием	*=	x *= y	Умножаем x на y

Деление с присваиванием	/=	x /= y	Делим x на y
Деление с остатком и с присваиванием	%=	x %= y	Присваиваем остаток от деления x на y переменной x

До этого момента, когда нам нужно было добавить число 5 к определенной переменной, мы делали следующее:

```
1. x = x + 5;
```

Это работает, но требуется два оператора для выполнения.

Так как стейтменты типа `x = x + 5` являются очень распространенными, то C++ предоставляет 5 арифметических операторов присваивания для нашего удобства. Вместо `x = x + 5`, мы можем записать:

```
1. x += 5;
```

Вместо:

```
1. x = x * y;
```

Мы можем записать:

```
1. x *= y;
```

Где оператор возведения в степень?

В языке C++ вместо оператора возведения в степень есть **функция pow()**, которая находится в заголовочном файле cmath. `pow(base, exponent)` эквивалентно $base^{exponent}$. Стоит отметить, что параметры pow() имеют тип double, поэтому вы можете использовать не только целые числа, но и дробные. Например:

```
1. #include <iostream>
2. #include <cmath> // подключаем pow()
3.
4. int main()
5. {
6.     std::cout << "Enter the base: ";
7.     double base;
8.     std::cin >> base;
9.
10.    std::cout << "Enter the exponent: ";
11.    double exp;
12.    std::cin >> exp;
```

```
13.
14.     std::cout << base << "^" << exp << " = " << pow(base, exp) << "\n";
15.
16.     return 0;
17. }
```

Тест

Задание №1

Вычислите результат следующего выражения: `6 + 5 * 4 % 3`.

Задание №2

Напишите программу, которая просит пользователя ввести целое число, а затем сообщает, является ли его число чётным или нечётным. Напишите функцию isEven(), которая возвращает true, если целое число является чётным. Используйте оператор деления с остатком, чтобы определить чётность числа.

> ***Подсказка***: Используйте ветвление if и оператор сравнения (==).

Урок №43. Инкремент, декремент и побочные эффекты

Операции **инкремента** (увеличение на `1`) и **декремента** (уменьшение на `1`) переменных настолько используемые, что у них есть свои собственные операторы в языке C++. Кроме того, каждый из этих операторов имеет две версии применения: префикс и постфикс.

Оператор	Символ	Пример	Операция
Префиксный инкремент (пре-инкремент)	++	++x	Инкремент x, затем вычисление x
Префиксный декремент (пре-декремент)	--	--x	Декремент x, затем вычисление x
Постфиксный инкремент (пост-инкремент)	++	x++	Вычисление x, затем инкремент x
Постфиксный декремент (пост-декремент)	--	x--	Вычисление x, затем декремент x

С операторами инкремента/декремента версии префикс всё просто. Значение переменной `x` сначала увеличивается/уменьшается, а затем уже вычисляется. Например:

```
1. int x = 5;
2. int y = ++x; // x = 6 и 6 присваивается переменной y
```

А вот с операторами инкремента/декремента версии постфикс несколько сложнее. Компилятор создает временную копию переменной `x`, увеличивает или уменьшает оригинальный `x` (не копию), а затем возвращает копию. Только после возврата копия `x` удаляется. Например:

```
1. int x = 5;
2. int y = x++; // x = 6, но переменной y присваивается 5
```

Рассмотрим вышеприведенный код детально. Во-первых, компилятор создает временную копию `x`, которая имеет то же значение, что и оригинал (`5`). Затем увеличивается первоначальный `x` с `5` до `6`. После этого компилятор возвращает временную копию, значением которой является `5`, и присваивает её переменной `y`.

Только после этого копия `x` уничтожается. Следовательно, в вышеприведенном примере мы получим `y = 5` и `x = 6`.

Вот еще один пример, показывающий разницу между версиями префикс и постфикс:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int x = 5, y = 5;
6.     std::cout << x << " " << y << std::endl;
7.     std::cout << ++x << " " << --y << std::endl; // версия префикс
8.     std::cout << x << " " << y << std::endl;
9.     std::cout << x++ << " " << y-- << std::endl; // версия постфикс
10.    std::cout << x << " " << y << std::endl;
11.
12.    return 0;
13. }
```

Результат выполнения программы:

```
5 5
6 4
6 4
6 4
7 3
```

В строке №7 переменные `x` и `y` увеличиваются/уменьшаются на единицу непосредственно перед обработкой компилятором, так что сразу выводятся их новые значения. А в строке №9 временные копии (`x = 6` и `y = 4`) отправляются в cout, а только после этого исходные `x` и `y` увеличиваются/уменьшаются на единицу. Именно поэтому изменения значений переменных после выполнения операторов версии постфикс не видно до следующей строки.

Версия префикс увеличивает/уменьшает значения переменных перед обработкой компилятором, версия постфикс - после обработки компилятором.

Правило: Используйте префиксный инкремент и префиксный декремент вместо постфиксного инкремента и постфиксного декремента. Версии префикс не только более производительны, но и ошибок с ними (по статистике) меньше.

Побочные эффекты

Функция или выражение имеет **побочный эффект**, если она/оно изменяет состояние чего-либо, делает ввод/вывод или вызывает другие функции, которые имеют побочные эффекты.

В большинстве случаев побочные эффекты являются полезными:

```
#include <iostream>

int main()
{
    int x = 5;
    ++x;
    std::cout << x;

    return 0;
}
```

В примере, приведенном выше, оператор присваивания имеет побочный эффект, который проявляется в изменении значения переменной `x`. Оператор `++` имеет побочный эффект инкремента переменной `x`. Вывод `x` имеет побочный эффект внесения изменений в консольное окно.

Также побочные эффекты могут приводить и к неожиданным результатам:

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    int x = 5;
    int value = add(x, ++x); // здесь 5 + 6 или 6 + 6?  Это зависит от компилятора и от того, в каком порядке он будет обрабатывать аргументы функции

    std::cout << value; // результатом может быть 11 или 12

    return 0;
}
```

Язык C++ не определяет порядок, в котором вычисляются аргументы функции. Если левый аргумент будет вычисляться первым, то `add(5, 6)` и результат — `11`. Если правый аргумент будет вычисляться первым, то `add(6, 6)` и результат — `12`! А проблема то кроется в побочном эффекте одного из аргументов функции add().

Вот еще один пример:

```
#include <iostream>

int main()
{
    int x = 1;
    x = x++;
    std::cout << x;

```

```
9.      return 0;
10.}
```

Какой результат выполнения этой программы? Если инкремент переменной `x` выполняется до операции присваивания, то ответ - `1`. Если же после операции присваивания, то ответ - `2`.

Есть и другие случаи, в которых C++ не определяет порядок обработки данных, поэтому в разных компиляторах могут быть разные результаты. Но даже в тех случаях, когда C++ и уточняет порядок обработки данных, некоторые компиляторы все равно вычисляют переменные с побочными эффектами некорректно. Этого всего можно избежать, если использовать переменные с побочными эффектами не более одного раза в одном стейтменте.

Правило: Не используйте переменную с побочным эффектом больше одного раза в одном стейтменте.

Урок №44. Условный тернарный оператор, оператор sizeof и Запятая

Мы уже ранее рассматривали оператор sizeof.

Оператор	Символ	Пример	Операция
sizeof	sizeof	sizeof(type) sizeof(variable)	Возвращает размер типа данных или переменной в байтах

Тогда мы использовали его для определения размера конкретных типов данных. Но также sizeof можно использовать и с переменными:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     double t = 7.0;
6.     std::cout << sizeof(t); // выводим размер переменной t в байтах
7. }
```

Оператор Запятая

Оператор Запятая (или ***"оператор Comma"***) позволяет обрабатывать несколько выражений (в то время, когда, обычно, позволяется только одно).

Оператор	Символ	Пример	Операция
Запятая	,	x, y	Вычисляется x, затем вычисляется y, а затем возвращается значение y

Выражение, в котором находится этот оператор, будет иметь значение правого операнда. Например:

```
1. int x = 0;
2. int y = 2;
3. int z = (++x, ++y); // инкремент переменных x и y
```

Переменной `z` будет присвоен результат вычисления `++y` (правого операнда), что равно `3`.

Почти в каждом случае, стейтмент, в котором есть оператор Запятая, лучше записывать в виде отдельных инструкций. Вышеприведенный код корректнее будет записать следующим образом:

```
1. int x = 0;
2. int y = 2;
3. ++x;
4. ++y;
5. int z = y;
```

Обратите внимание, оператор Запятая имеет самый низкий приоритет из всех операторов (даже ниже, чем в оператора присваивания), поэтому следующие две строки кода делают не одно и то же:

```
1. z = (a, b); // сначала вычисляется выражение (a, b), которое равняется значению b, а затем результат присваивается переменной z
2. z = a, b; // вычисляется как "(z = a), b", поэтому переменной z присваивается значение a, переменная b - игнорируется
```

Большинство программистов не используют оператор Comma вообще (разве что только в циклах for).

Обратите внимание, запятая, которая используется в вызовах функций, не является оператором Comma:

```
1. int sum = add(x, y); // эта запятая не является оператором Comma
```

Аналогично, при объявлении нескольких переменных в одной строке, запятая используется как разделитель, а не как оператор:

```
1. int x(3), y(5); // эта запятая не является оператором Comma
```

Правило: Избегайте использования оператора Comma (исключением являются циклы for).

Условный тернарный оператор

Условный (тернарный) оператор (обозначается как `?:`) является единственным тернарным оператором в языке C++, который работает с 3-мя операндами.

Из-за этого его часто называют просто ***"тернарный оператор"***.

Оператор	Символ	Пример	Операция
Условный	?:	c ? x : y	Если c — ненулевое значение (true), то вычисляется x, в противном случае — y

Оператор `?:` предоставляет сокращенный способ (альтернативу) ветвления if/else.

Стейтменты if/else:

```
if (условие)
   выражение;
else
   другое_выражение;
```

Можно записать как:

```
(условие) ? выражение : другое_выражение;
```

Обратите внимание, операнды условного оператора должны быть выражениями (а не стейтментами).

Например, ветвление if/else, которое выглядит следующим образом:

```
if (условие)
   x = значение1;
else
   x = значение2;
```

Можно записать как:

```
x = (условие) ? значение1 : значение2;
```

Большинство программистов предпочитают последний вариант, так как он читабельнее.

Давайте рассмотрим еще один пример. Чтобы определить, какое значение поместить в переменную `larger`, мы можем сделать так:

```
1. if (x > y)
2.     larger = x;
3. else
4.     larger = y;
```

Или вот так:

```
1. larger = (x > y) ? x : y;
```

Обычно, часть с условием помещают внутри скобок, чтобы убедиться, что приоритет операций корректно сохранен и так удобнее читать.

Помните, что оператор `?:` имеет очень низкий приоритет, из-за этого его следует записывать в круглых скобках.

Например, для вывода `x` или `y`, мы можем сделать следующее:

```
1. if (x > y)
2.     std::cout << x;
3. else
4.     std::cout << y;
```

Или с помощью тернарного оператора:

```
1. std::cout << ((x > y) ? x : y);
```

Давайте рассмотрим, что произойдет, если мы не заключим в скобки весь условный оператор в вышеприведенном случае. Поскольку оператор `<<` имеет более высокий приоритет, чем оператор `?:`, то следующий стейтмент (где мы не заключили весь тернарный оператор в круглые скобки, а только лишь условие):

```
1. std::cout << (x > y) ? x : y;
```

Будет обрабатываться как:

```
1. (std::cout << (x > y)) ? x : y;
```

Таким образом, в консольном окне мы увидим `1` (true), если `x > y`, в противном случае — выведется `0` (false).

> ***Совет:*** Всегда заключайте в скобки условную часть тернарного оператора, а лучше весь тернарный оператор.

Условный тернарный оператор — это удобное упрощение ветвления if/else, особенно при присваивании результата переменной или возврате определенного значения. Но его не следует использовать вместо сложных ветвлений if/else, так как в таких случаях читабельность кода резко ухудшается и вероятность возникновения ошибок только растет.

Правило: Используйте условный тернарный оператор только в тривиальных случаях.

Условный тернарный оператор вычисляется как выражение

Стоит отметить, что условный оператор вычисляется как выражение, в то время как ветвление if/else обрабатывается как набор стейтментов. Это означает, что тернарный оператор `?:` может быть использован там, где if/else применить невозможно, например, при инициализации константы:

```
1. bool inBigClassroom = false;
2. const int classSize = inBigClassroom ? 30 : 20;
```

Здесь нельзя использовать if/else, так как константы должны быть инициализированы при объявлении, а стейтмент не может быть значением для инициализации.

Урок №45. Операторы сравнения

В языке C++ есть 6 операторов сравнения:

Оператор	Символ	Пример	Операция
Больше	>	x > y	true, если x больше y, в противном случае — false
Меньше	<	x < y	true, если x меньше y, в противном случае — false
Больше или равно	>=	x >= y	true, если x больше/равно y, в противном случае — false
Меньше или равно	<=	x <= y	true, если x меньше/равно y, в противном случае — false
Равно	==	x == y	true, если x равно y, в противном случае — false
Не равно	!=	x != y	true, если x не равно y, в противном случае — false

Вы уже могли их видеть в коде. Они довольно простые. Каждый из этих операторов вычисляется в логическое значение true (1) или false (0).

Вот несколько примеров использования этих операторов на практике:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;

    std::cout << "Enter another integer: ";
    int y;
    std::cin >> y;

    if (x == y)
        std::cout << x << " equals " << y << "\n";
```

```
15.     if (x != y)
16.         std::cout << x << " does not equal " << y << "\n";
17.     if (x > y)
18.         std::cout << x << " is greater than " << y << "\n";
19.     if (x < y)
20.         std::cout << x << " is less than " << y << "\n";
21.     if (x >= y)
22.         std::cout << x << " is greater than or equal to " << y << "\n";
23.     if (x <= y)
24.         std::cout << x << " is less than or equal to " << y << "\n";
25.
26.     return 0;
27. }
```

Результат выполнения программы:

```
Enter an integer: 4
Enter another integer: 5
4 does not equal 5
4 is less than 5
4 is less than or equal to 5
```

Всё просто!

Сравнение чисел типа с плавающей точкой

Сравнение значений типа с плавающей точкой с помощью любого из этих операторов — дело опасное. Почему? Из-за тех самых небольших ошибок округления, которые могут привести к неожиданным результатам. Например:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      double d1(100 - 99.99); // должно быть 0.01
6.      double d2(10 - 9.99); // должно быть 0.01
7.
8.      if (d1 == d2)
9.          std::cout << "d1 == d2" << "\n";
10.     else if (d1 > d2)
11.         std::cout << "d1 > d2" << "\n";
12.     else if (d1 < d2)
13.         std::cout << "d1 < d2" << "\n";
14.
15.     return 0;
16. }
```

Вот так раз:

```
d1 > d2
```

В вышеприведенной программе, `d1 = 0.010000000000000005116`, а `d2 = 0.009999999999997868`. Оба этих числа очень близки к `0.1`, но `d1` больше `d2`. Они не равны.

Иногда сравнение чисел типа с плавающей точкой бывает неизбежным. В таком случае следует использовать операторы `>`, `<`, `>=` и `<=` только если значения не очень близки. А вот если два операнда очень близки значениями, то результат уже может быть неожиданный. В вышеприведенном примере последствия неправильного результата незначительны, а вот с оператором равенства дела обстоят хуже, так как даже при самой маленькой неточности результат сразу меняется на противоположный ожидаемому. Не рекомендуется использовать операторы `==` или `!=` с числами типа с плавающей точкой. Вместо них следует использовать функцию, которая вычисляет, насколько *близки* эти два значения. Если они "достаточно близки", то мы считаем их равными. Значение, используемое для представления термина "достаточно близки", называется **эпсилоном**. Оно, обычно, небольшое (например, `0.0000001`).

Очень часто начинающие разработчики пытаются писать свои собственные функции определения равенства чисел:

```
1. #include <cmath> // для функции fabs()
2.
3. bool isAlmostEqual(double a, double b, double epsilon)
4. {
5.     // Если разница между a и b меньше значения эпсилона, то тогда a и b - "достаточно близки"
6.     return fabs(a - b) <= epsilon;
7. }
```

> ***Примечание:*** Функция fabs() - это функция из заголовочного файла cmath, которая возвращает абсолютное значение (модуль) параметра. `fabs(a - b)` возвращает положительное число, как разницу между `a` и `b`.

Функция isAlmostEqual() из примера, приведенного выше, сравнивает разницу `(a - b)` и эпсилон, вычисляя, таким образом, *близость* чисел. Если `a` и `b` достаточно близки, то функция возвращает true.

Хоть это и рабочий вариант, но он не идеален. Эпсилон 0.00001 подходит для чисел около 1.0, но будет слишком большим для чисел типа 0.0000001 и слишком малым для чисел типа 10000. Это означает, что каждый раз, при вызове функции, нам нужно будет выбирать наиболее соответствующий входным данным функции эпсилон.

Дональд Кнут, известный учёный, предложил следующий способ в своей книге "Искусство программирования, том 2: Получисленные алгоритмы" (1968):

```
#include <cmath> // для функции fabs()

// Возвращаем true, если разница между a и b в пределах процента эпсилона
bool approximatelyEqual(double a, double b, double epsilon)
{
    return fabs(a - b) <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}
```

Здесь, вместо использования эпсилона как абсолютного числа, мы используем его как умножитель, чтобы подстроиться под входные данные.

Рассмотрим детально, как работает функция approximatelyEqual(). Слева от оператора `<=` абсолютное значение `(a - b)` сообщает нам разницу между `a` и `b` (положительное число). Справа от `<=` нам нужно вычислить эпсилон, т.е. наибольшее значение "близости чисел", которое мы готовы принять. Для этого алгоритм выбирает большее из чисел `a` и `b` (как приблизительный показатель общей величины чисел), а затем умножает его на эпсилон. В этой функции эпсилон представляет собой процентное соотношение. Например, если термин "достаточно близко" означает, что `a` и `b` находятся в пределах 1% разницы (больше или меньше), то мы вводим эпсилон 1% (1% = 1/100 = 0.01). Его значение можно легко регулировать, в зависимости от обстоятельств (например, 0.01% = эпсилон 0.0001). Чтобы сделать неравенство (`!=`) вместо равенства - просто вызовите эту функцию, используя логический оператор НЕ (`!`), чтобы *перевернуть* результат:

```
if (!approximatelyEqual(a, b, 0.001))
    std::cout << a << " is not equal to " << b << "\n";
```

Но и функция approximatelyEqual() тоже не идеальна, особенно, когда дело доходит до чисел, близких к нулю:

```
#include <iostream>
#include <cmath> // для функции fabs()

// Возвращаем true, если разница между a и b в пределах процента эпсилона
bool approximatelyEqual(double a, double b, double epsilon)
{
    return fabs(a - b) <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

int main()
{
    // Значение a очень близкое к 1.0, но, из-за ошибок округления, чуть меньше 1.0
    double a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

```

```
    // Во-первых, давайте сравним значение a (почти 1.0) с 1.0
    std::cout << approximatelyEqual(a, 1.0, 1e-8) << "\n";

    // Во-вторых, давайте сравним значение a - 1.0 (почти 0.0) с 0.0
    std::cout << approximatelyEqual(a - 1.0, 0.0, 1e-8) << "\n";
}
```

Возможно, вы удивитесь, но результат:

```
1
0
```

Второй вызов не сработал так, как ожидалось. Математика просто ломается, когда дело доходит до нулей.

Но и этого можно избежать, используя как абсолютный эпсилон (то, что мы делали в первом способе), так и относительный (способ Кнута) вместе:

```
// Возвращаем true, если разница между a и b меньше absEpsilon или в пределах relEpsilon
bool approximatelyEqualAbsRel(double a, double b, double absEpsilon, double relEpsilon)
{
    // Проверяем числа на их близость - это нужно в тех случаях, когда сравниваемые числа являются нулевыми или "около нуля"
    double diff = fabs(a - b);
    if (diff <= absEpsilon)
        return true;

    // В противном случае, возвращаемся к алгоритму Кнута
    return diff <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * relEpsilon);
}
```

Здесь мы добавили новый параметр - `absEpsilon`. Сначала мы сравниваем `a` и `b` с `absEpsilon`, который должен быть задан как очень маленькое число (например, `1e-12`). Таким образом, мы решаем случаи, когда `a` и `b` — нулевые значения или близки к нулю. Если это не так, то мы возвращаемся к алгоритму Кнута.

Протестируем:

```
#include <iostream>
#include <cmath> // для функции fabs()

// Возвращаем true, если разница между a и b в пределах процента эпсилона
bool approximatelyEqual(double a, double b, double epsilon)
{
    return fabs(a - b) <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);

}

// Возвращаем true, если разница между a и b меньше absEpsilon или в пределах relEpsilon
bool approximatelyEqualAbsRel(double a, double b, double absEpsilon, double relEpsilon)
```

```
12. {
13.     // Проверяем числа на их близость - это нужно в случаях, когда
    сравниваемые числа являются нулевыми или около нуля
14.     double diff = fabs(a - b);
15.     if (diff <= absEpsilon)
16.         return true;
17. 
18.     // В противном случае, возвращаемся к алгоритму Кнута
19.     return diff <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * relEpsilon);
20. }
21. 
22. int main()
23. {
24.     // Значение a очень близко к 1.0, но, из-
    за ошибок округления, чуть меньше 1.0
25.     double a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
26. 
27.     std::cout << approximatelyEqual(a, 1.0, 1e-
    8) << "\n";     // сравниваем "почти 1.0" с 1.0
28.     std::cout << approximatelyEqual(a - 1.0, 0.0, 1e-
    8) << "\n"; // сравниваем "почти 0.0" с 0.0
29.     std::cout << approximatelyEqualAbsRel(a - 1.0, 0.0, 1e-12, 1e-
    8) << "\n"; // сравниваем "почти 0.0" с 0.0
30. }
```

Результат:

```
1
0
1
```

С удачно подобранным `absEpsilon`, функция approximatelyEqualAbsRel() обрабатывает близкие к нулю и нулевые значения корректно.

Сравнение чисел типа с плавающей точкой - сложная тема, и нет одного идеального алгоритма, который подойдет в любой ситуации. Однако для большинства случаев с которыми вы будете сталкиваться, функции approximatelyEqualAbsRel() должно быть достаточно.

Урок №46. Логические операторы: И, ИЛИ, НЕ

В то время как операторы сравнения используются для проверки конкретного условия: ложное оно или истинное, они могут проверить только одно условие за определенный промежуток времени. Но бывают ситуации, когда нужно протестировать сразу несколько условий. Например, чтобы узнать, выиграли ли мы в лотерею, нам нужно сравнить все цифры купленного билета с выигрышными. Если в лотерее 6 цифр, то нужно выполнить 6 сравнений, все из которых должны быть true.

Также иногда нам нужно знать, является ли хоть одно из нескольких условий истинным. Например, мы не пойдем сегодня на работу, если больны или слишком устали, или если выиграли в лотерею. :) Нам нужно проверить, является ли хоть одно из этих 3-х условий истинным. Как это сделать? С помощью логических операторов! Они позволяют проверить сразу несколько условий за раз.

В языке C++ есть 3 логических оператора:

Оператор	Символ	Пример	Операция
Логическое НЕ	!	!x	true, если x — false и false, если x — true
Логическое И	&&	x && y	true, если x и y — true, в противном случае — false
Логическое ИЛИ	||	x || y	true, если x или y — true, в противном случае — false

Логический оператор НЕ

Мы уже с ним ранее сталкивались.

Логический оператор НЕ (!)	
Операнд	**Результат**
true	false
false	true

Если операндом является true, то, после применения логического НЕ, результатом будет false. Если же операнд до применения оператора НЕ был false, то после его применения станет true. Другими словами, логический оператор НЕ меняет результат на противоположный начальному значению. Он часто используется в условных выражениях:

```
1. bool bTooLarge = (x > 100); // переменная bTooLarge будет true, если x > 100
2. if (!bTooLarge)
3.     // Делаем что-нибудь с x
4. else
5.     // Выводим ошибку
```

Следует помнить, что логический оператор НЕ имеет очень высокий уровень приоритета. Новички часто совершают следующую ошибку:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int x = 5;
6.     int y = 7;
7.
8.     if (!x == y)
9.         std::cout << "x does not equal y";
10.    else
11.        std::cout << "x equals y";
12.
13.    return 0;
14.}
```

Результат выполнения программы:

```
x equals y
```

Но `x` ведь не равно `y`, как это возможно? Поскольку приоритет логического оператора НЕ выше, чем приоритет оператора равенства, то выражение `! x == y` обрабатывается как `(! x) == y`. Так как `x` — это `5`, то `!x` — это `0`. Условие `0 == y` ложное, поэтому выполняется часть else!

> ***Напоминание***: Любое ненулевое целое значение в логическом контексте является true. Так как `x = 5`, то `x` вычисляется как true, а вот `!x = false`, т.е. `0`. Использование целых чисел в логических операциях подобным образом может запутать не только пользователя, но и самого разработчика, поэтому так не рекомендуется делать!

Правильный способ написания программы, приведенной выше:

```
#include <iostream>

int main()
{
    int x = 5;
    int y = 7;

    if (!(x == y))
        std::cout << "x does not equal y";
    else
        std::cout << "x equals y";

    return 0;
}
```

Сначала обрабатывается `x == y`, а затем уже оператор НЕ изменяет результат на противоположный.

Правило: Если логический оператор НЕ должен работать с результатами работы других операторов, то другие операторы и их операнды должны находиться в круглых скобках.

Логический оператор ИЛИ

Если хоть одно из двух условий является истинным, то логический оператор ИЛИ является true.

Логический оператор ИЛИ (||)		
Левый операнд	**Правый операнд**	**Результат**
false	false	false
false	true	true
true	false	true
true	true	true

Рассмотрим следующую программу:

```
#include <iostream>

int main()
{
    std::cout << "Enter a number: ";
    int value;
    std::cin >> value;

    if (value== 0 || value== 1)
        std::cout << "You picked 0 or 1" << std::endl;
    else
        std::cout << "You did not pick 0 or 1" << std::endl;

    return 0;
}
```

Здесь мы использовали логический оператор ИЛИ, чтобы проверить, является ли хоть одно из двух условий истинным: левое (`value == 0`) или правое (`value == 1`). Если хоть одно из условий — true или оба сразу true, то выполняться будет стейтмент if. Если ни одно из условий не является true, то результат — false и выполняться будет стейтмент else.

Вы можете связать сразу несколько условий:

```
if (value == 0 || value == 1 || value == 2 || value == 3)
```

```
2.      std::cout << "You picked 0, 1, 2, or 3" << std::endl;
```

Новички иногда путают логическое ИЛИ (`||`) с побитовым ИЛИ (`|`). Хоть у них и одинаковые названия, функции они выполняют разные.

Логический оператор И

Только при условии, что оба операнда будут истинными, логический оператор И будет true. Если нет, тогда — false.

Логический оператор И (&&)		
Левый операнд	**Правый операнд**	**Результат**
false	false	false
false	true	false
true	false	false
true	true	true

Например, мы хотим узнать, находится ли значение переменной `x` в диапазоне от 10 до 20. Здесь у нас есть два условия: мы должны проверить, является ли `x` больше 10 и является ли `x` меньше 20.

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      std::cout << "Enter a number: ";
6.      int value;
7.      std::cin >> value ;
8.
9.      if (value > 10 && value < 20)
10.         std::cout << "Your value is between 10 and 20" << std::endl;
11.     else
12.         std::cout << "Your value is not between 10 and 20" << std::endl;
13.
14.     return 0;
15. }
```

Если оба условия истинны, то выполняется часть if. Если же хоть одно или сразу оба условия ложные, то выполняется часть else.

Как и с логическим ИЛИ, мы можем комбинировать сразу несколько условий И:

```
1. if (value > 10 && value < 20 && value != 16)
2.     // Делаем что-нибудь
3. else
4.     // Делаем что-нибудь другое
```

Короткий цикл вычислений

Для того, чтобы логическое И возвращало true, оба операнда должны быть истинными. Если первый операнд вычисляется как false, то оператор И должен сразу возвращать false независимо от результата второго операнда (даже без его обработки). Это называется **коротким циклом вычисления** (англ. ***"short circuit evaluation"***) и выполняется он, в первую очередь, в целях оптимизации.

Аналогично, если первый операнд логического ИЛИ является true, то и всё условие будет true (даже без обработки второго операнда).

Как и в случае с оператором ИЛИ, новички иногда путают логическое И (`&&`) с побитовым И (`&`).

Использование логических операторов И/ИЛИ

Иногда возникают ситуации, когда смешивания логических операторов И и ИЛИ в одном выражении не избежать. Тогда следует знать о возможных проблемах, которые могут произойти.

Многие программисты думают, что логические И и ИЛИ имеют одинаковый приоритет (или забывают, что это не так), так же как и сложение/вычитание или умножение/деление. Тем не менее, приоритет логического И выше приоритета ИЛИ. Таким образом, операции с оператором И всегда будут вычисляться первыми (если только операции с ИЛИ не находятся в круглых скобках).

Рассмотрим следующее выражение: `value1 || value2 && value3`. Поскольку приоритет логического И выше, то обрабатываться выражение будет так:

```
value1 || (value2 && value3)
```

А не так:

```
(value1 || value2) && value3
```

Хорошей практикой является использование круглых скобок с операциями. Это предотвратит ошибки приоритета, увеличит читабельность кода и чётко даст понять

компилятору, как следует обрабатывать выражения. Например, вместо того, чтобы писать `value1 && value2 || value3 && value4`, лучше записать `(value1 && value2) || (value3 && value4)`.

Законы Де Моргана

Многие программисты совершают ошибку, думая, что `!(x && y)` - это то же самое, что и `!x && !y`. К сожалению, вы не можете использовать логическое НЕ подобным образом.

Законы Де Моргана гласят, что `!(x && y)` эквивалентно `!x || !y`, а `!(x || y)` эквивалентно `!x && !y`.

Другими словами, логические операторы И и ИЛИ меняются местами! В некоторых случаях, это даже полезно, так как улучшает читабельность.

А где же побитовое исключающее ИЛИ (XOR)?

Побитовое исключающее ИЛИ (XOR) - это логический оператор, который используется в некоторых языках программирования для проверки на истинность нечётного количества условий.

Побитовое исключающее ИЛИ (XOR)		
Левый операнд	**Правый операнд**	**Результат**
false	false	false
false	true	true
true	false	true
true	true	false

В языке C++ нет такого оператора. В отличии от логических И/ИЛИ, к XOR не применяется короткий цикл вычислений. Однако его легко можно сымитировать, используя оператор неравенства (`!=`):

```
1. if (a != b) ... // a XOR b (предполагается, что a и b имеют тип bool)
```

Можно также расширить количество операндов:

```
1. if (a != b != c != d) ... // a XOR b XOR c XOR d (предполагается, что a, b, c и d имеют тип bool)
```

Следует отметить, что вышеприведенные шаблоны XOR работают только, если операнды имеют логический (а не целочисленный) тип данных. Если вы хотите, чтобы это работало и с целыми числами, то используйте оператор static_cast.

Форма XOR, которая работает и с другими типами данных (с помощью оператора static_cast мы можем конвертировать любой тип данных в тип bool):

```
1. if (static_cast<bool>(a) != static_cast<bool>(b) != static_cast<bool>(c) != static_cast<bool>(d)) ... // a XOR b XOR c XOR d, для любого типа, который может быть конвертирован в тип bool
```

Тест

Какой результат следующих выражений?

- *Выражение №1:* `(true && true) || false`
- *Выражение №2:* `(false && true) || true`
- *Выражение №3:* `(false && true) || false || true`
- *Выражение №4:* `(5 > 6 || 4 > 3) && (7 > 8)`
- *Выражение №5:* `!(7 > 6 || 3 > 4)`

Урок №47. Конвертация чисел из двоичной системы в десятичную и наоборот

Чтобы научиться конвертировать числа из двоичной (бинарной) системы счисления в десятичную и наоборот, прежде всего необходимо понять, как целые числа представлены в двоичной системе.

Представление чисел в двоичной системе

Рассмотрим обычное десятичное число, например, число 5623. Интуитивно понятно, что означают все эти цифры: `(5 * 1000) + (6 * 100) + (2 * 10) + (3 * 1)`. Так как в десятичной системе счисления всего 10 цифр, то каждое значение умножается на множитель 10 в степени ^{n}. Выражение, приведенное выше, можно записать следующим образом: $(5 * 10^3) + (6 * 10^2) + (2 * 10^1) + (3 * 1)$.

Двоичные числа работают по аналогичной схеме, за исключением того, что в системе всего 2 числа (0 и 1) и множитель не 10, а 2. Так же как запятые (или пробелы) используются для улучшения читабельности больших десятичных чисел (например, `1, 427, 435`), двоичные числа пишутся группами — в каждой по 4 цифры (например, `1101 0101`).

Десятичное значение	Двоичное значение
0	0
1	1
2	10
3	11
4	100
5	101

6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Конвертация чисел из двоичной системы в десятичную

В примерах, приведенных ниже, предполагается, что мы работаем с целочисленными значениями unsigned. Рассмотрим 8-битное (1-байтовое) двоичное число: 0101 1110. Оно означает `(0 * 128) + (1 * 64) + (0 * 32) + (1 * 16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1)`. Если суммировать, то получим десятичное `64 + 16 + 8 + 4 + 2 = 94`.

Вот тот же процесс, но в таблице. Мы умножаем каждую двоичную цифру на её значение, которое определяется её положением.

Выполним конвертацию двоичного числа 0101 1110 в десятичную систему:

Двоичный символ	0	1	0	1	1	1	1	0
*** Значение символа**	128	64	32	16	8	4	2	1
= Результат (94)	0	64	0	16	8	4	2	0

А теперь конвертируем двоичное 1001 0111 в десятичную систему:

Двоичный символ	1	0	0	1	0	1	1	1
*** Значение символа**	128	64	32	16	8	4	2	1
= Результат (151)	128	0	0	16	0	4	2	1

Получается:

```
1001 0111 (двоичное) = 151 (десятичное)
```

Таким способом можно легко конвертировать и 16-битные, и 32-битные двоичные числа, просто добавляя столбцы. Обратите внимание, проще всего начинать отсчет справа налево, умножая на 2 каждое последующее значение.

Способ №1: Конвертация чисел из десятичной системы в двоичную

Первый способ конвертации чисел из десятичной системы счисления в двоичную заключается в непрерывном делении числа на 2 и записывании остатков. Если остаток (**"r"** от англ. *"**r**emainder"*) есть, то пишем `1`, если нет — пишем `0`. Затем, читая остатки снизу вверх, мы получим готовое двоичное число.

Например, конвертация десятичного числа 148 в двоичную систему счисления:

```
148 / 2 = 74 r0
74 / 2 = 37 r0
37 / 2 = 18 r1
18 / 2 = 9 r0
9 / 2 = 4 r1
4 / 2 = 2 r0
```

```
2 / 2 = 1 r0
1 / 2 = 0 r1
```

Записываем остатки снизу вверх: 1001 0100.

```
148 (десятичное) = 1001 0100 (двоичное)
```

Вы можете проверить этот ответ путем конвертации двоичного числа обратно в десятичную систему:

```
(1 * 128) + (0 * 64) + (0 * 32) + (1 * 16) + (0 * 8) + (1 * 4)
+ (0 * 2) + (0 * 1) = 148
```

Способ №2: Конвертация чисел из десятичной системы в двоичную

Этот способ хорошо подходит для небольших двоичных чисел. Рассмотрим десятичное число 148 еще раз. Какое наибольшее число, умноженное на 2 (из ряда 1, 2, 4, 8, 16, 32, 64, 128, 256 и т.д.), меньше 148? Ответ: 128.

```
148 >= 128? Да, поэтому 128-й бит равен 1. 148 - 128 = 20
20 >= 64? Нет, поэтому 64-й бит равен 0.
20 >= 32? Нет, поэтому 32-й бит равен 0.

20 >= 16? Да, поэтому 16-й бит равен 1. 20 - 16 = 4
4 >= 8? Нет, поэтому 8-й бит равен 0.
4 >= 4? Да, поэтому 4-й бит равен 1. 4 - 4 = 0, что означает,
что все остальные биты равны 0.
```

> ***Примечание***: Если ответом является "Да", то мы имеем true, что означает 1. Если ответом является "Нет", то мы имеем false, что означает 0.

Результат:

```
148 = (1 * 128) + (0 * 64) + (0 * 32) + (1 * 16) + (0 * 8) + (1
* 4) + (0 * 2) + (0 * 1) = 1001 0100
```

То же самое, но в таблице:

Двоичный символ	1	0	0	1	0	1	0	0
* Значение символа	128	64	32	16	8	4	2	1
= Результат (148)	128	0	0	16	0	4	0	0

Еще один пример

Конвертируем десятичное число 117 в двоичную систему счисления, **используя способ №1**:

```
117 / 2 = 58 r1
58 / 2 = 29 r0
29 / 2 = 14 r1
14 / 2 = 7 r0
7 / 2 = 3 r1
3 / 2 = 1 r1
1 / 2 = 0 r1
```

Запишем число, состоящее из остатков (снизу вверх):

```
117 (десятичное) = 111 0101 (двоичное)
```

А теперь выполним ту же конвертацию, но с **использованием способа №2:**

Наибольшее число, умноженное на 2, но которое меньше 117 - это 64.

```
117 >= 64? Да, поэтому 64-й бит равен 1. 117 - 64 = 53.
53 >= 32? Да, поэтому 32-й бит равен 1. 53 - 32 = 21.
21 >= 16? Да, поэтому 16-й бит равен 1. 21 - 16 = 5.

5 >= 8? Нет, поэтому 8-й бит равен 0.
5 >= 4? Да, поэтому 4-й бит равен 1. 5 - 4 = 1.
1 >= 2? Нет, поэтому 2-й бит равен 0.
1 >= 1? Да, поэтому 1-й бит равен 1.
```

Результат:

```
117 (десятичное) = 111 0101 (двоичное)
```

Сложение двоичных чисел

В некоторых случаях (один из них мы рассмотрим ниже) вам может понадобиться выполнить сложение двух двоичных чисел. Это на удивление легко (может быть даже проще, чем сложение десятичных чисел), хотя поначалу может показаться немного странным, но вы быстро к этому привыкните.

Рассмотрим сложение следующих двух небольших двоичных чисел:

```
0110 (6 в десятичной системе) +
0111 (7 в десятичной системе)
```

Во-первых, числа нужно записать в столбик (как показано выше). Затем справа налево и сверху вниз мы добавляем каждый столбец с цифрами, как будто это десятичные числа. Так как в бинарной системе есть только два числа: 0 и 1, то всего есть 4 возможных исхода:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0, и 1 переносим в следующую колонку

Начнем с первой колонки (столбца):

```
0110 (6 в десятичной системе) +
0111 (7 в десятичной системе)
----
   1
```

0 + 1 = 1. Легко.

Вторая колонка:

```
 1
0110 (6 в десятичной системе) +
0111 (7 в десятичной системе)
----
  01
```

1 + 1 = 0, и 1 остается в памяти до следующей колонки.

Третья колонка:

```
11
0110 (6 в десятичной системе) +
0111 (7 в десятичной системе)
----
 101
```

А вот здесь уже немного сложнее. Обычно, 1 + 1 = 0 и остается единица, которую мы переносим в следующую колонку. Тем не менее, у нас уже есть 1 из предыдущего столбца и нам нужно добавить еще 1. Что делать? А вот что: 1 остается, а еще 1 мы переносим дальше.

Последняя колонка:

```
11
0110 (6 в десятичной системе) +
0111 (7 в десятичной системе)
----
1101
```

0 + 0 = 0, но так как есть еще 1, то результат — 1101.

```
13 (десятичное) = 1101 (двоичное)
```

Вы спросите: "А как добавить десятичную единицу к любому другому двоичному числу (например, к 1011 0011)?". Точно так же, как мы это делали выше, только числом снизу является двоичная единица. Например:

```
       1 (переносим в следующую колонку)
1011 0011 (двоичное число)
0000 0001 (1 в двоичной системе)
---------
1011 0100
```

Числа signed и метод "two’s complement"

В примерах, приведенных выше, мы работаем только с целыми числами unsigned, которые могут быть только положительными. Сейчас же мы рассмотрим то, как работать с числами signed, которые могут быть как положительными, так и отрицательными.

С целыми числами signed используется **метод "two’s complement"**. Он означает, что самый левый (самый главный) бит используется в качестве **знакового бита**. Если

значением знакового бита является 0, то число положительное, если 1 — число отрицательное.

Положительные числа signed хранятся так же, как и положительные числа unsigned (с 0 в качестве знакового бита). А вот отрицательные числа signed хранятся в виде обратных положительных чисел + 1. Например, выполним конвертацию `-5` из десятичной системы счисления в двоичную, используя метод "two’s complement":

```
Сначала выясняем бинарное представление 5: 0000 0101
Затем инвертируем все биты (конвертируем в противоположные): 1111 1010
Затем добавляем к числу единицу: 1111 1011
```

Конвертация `-76` из десятичной системы счисления в двоичную:

```
Представление положительного 76: 0100 1100
Инвертируем все биты: 1011 0011
Добавляем к числу единицу: 1011 0100
```

Почему мы добавляем единицу? Рассмотрим это на примере 0 (нуля). Если противоположностью отрицательного числа является его положительная форма, то 0 имеет два представления: 0000 0000 (положительный ноль) и 1111 1111 (отрицательный ноль). При добавлении единицы, в 1111 1111 произойдет переполнение, и значение изменится на 0000 0000. Добавление единицы позволяет избежать наличия двух представлений нуля и упрощает внутреннюю логику, необходимую для выполнения арифметических вычислений с отрицательными числами.

Перед тем, как конвертировать двоичное число (используя метод "two’s complement") обратно в десятичную систему счисления, нужно сначала посмотреть на знаковый бит. Если это 0, то смело используйте способы, приведенные выше, для целых чисел unsigned. Если же знаковым битом является 1, то тогда нужно инвертировать все биты, затем добавить единицу, затем конвертировать в десятичную систему, и уже после этого менять знак десятичного числа на отрицательный (потому что знаковый бит изначально был отрицательным).

Например, выполним конвертацию двоичного 1001 1110 (используя метод "two’s complement") в десятичную систему счисления:

```
Имеем: 1001 1110
Инвертируем биты: 0110 0001
Добавляем единицу: 0110 0010
Конвертируем в десятичную систему счисления: (0 * 128) + (1 *
```

```
64) + (1 * 32) + (0 * 16) + (0 * 8) + (0 * 4) + (1 * 2) + (0 *
1 ) = 64 + 32 + 2 = 98
```

Так как исходный знаковый бит был отрицательным, то результатом является `-98`.

Почему так важен тип данных?

Рассмотрим двоичное число 1011 0100. Что это за число в десятичной системе счисления? Вы, наверное, подумаете, что это 180, и, если бы это было стандартное двоичное число unsigned, то вы были бы правы. Однако, если здесь используется метод "two’s complement", то результат будет другой: -76. Также значение еще может быть другое, если оно закодировано каким-то третьим способом.

Так как же язык C++ понимает в какое число конвертировать 1011 0100: в 180 или в -76?

На предыдущих уроках мы говорили: "Когда вы указываете тип данных переменной, компилятор и процессор заботятся о деталях конвертации этого значения в соответствующую последовательность бит определенного типа данных. Когда вы просите ваше значение обратно, то оно "восстанавливается" из соответствующей последовательности бит в памяти".

Тип переменной используется для конвертации бинарного представления числа обратно в ожидаемую форму. Поэтому, если вы указали целочисленный тип данных unsigned, то компилятор знает, что 1011 0100 - это стандартное двоичное число, а его представление в десятичной системе счисления — 180. Если же типом переменной является целочисленный тип signed, то компилятор знает, что 1011 0100 закодирован с помощью метода "two’s complement" и его представлением в десятичной системе счисления является число -76.

Тест

Задание №1

Конвертируйте двоичное число 0100 1101 в десятичную систему счисления.

Задание №2

Конвертируйте десятичное число 93 в 8-битное двоичное число unsigned.

Задание №3

Конвертируйте десятичное число -93 в 8-битное двоичное число signed (используя метод "two’s complement").

Задание №4

Конвертируйте двоичное число 1010 0010 в десятичное unsigned.

Задание №5

Конвертируйте двоичное число 1010 0010 в десятичное signed (используя метод "two’s complement").

Задание №6

Напишите программу, которая просит пользователя ввести число от 0 до 255. Выведите его как 8-битное двоичное число (в парах по 4 цифры). Не используйте побитовые операторы.

> ***Подсказки:***
>
> - Воспользуйтесь способом конвертации №2. Предполагается, что наименьшим числом для сравнения является 128.
> - Напишите функцию для проверки входных чисел: являются ли они больше чисел, умноженных на 2 (т.е. чисел 1, 2, 4, 8, 16, 32, 64 и 128). Если это так, то выводится 1, если нет — выводится 0.

Урок №48. Побитовые операторы

Побитовые операторы манипулируют отдельными битами в пределах переменной.

> ***Примечание*:** Для некоторых этот материал может показаться сложным. Если вы застряли или что-то не понятно - пропустите этот урок (и следующий), в будущем сможете вернуться и разобраться детально. Он не столь важен для прогресса в изучении языка C++, как другие уроки, и изложен здесь в большей мере для общего развития.

Зачем нужны побитовые операторы?

В далеком прошлом компьютерной памяти было очень мало и ею сильно дорожили. Это было стимулом максимально разумно использовать каждый доступный бит. Например, в логическом типе данных bool есть всего лишь два возможных значения (true и false), которые могут быть представлены одним битом, но по факту занимают целый байт памяти! А это, в свою очередь, из-за того, что переменные используют уникальные адреса памяти, а они выделяются только в байтах. Переменная bool занимает 1 бит, а другие 7 бит — тратятся впустую.

Используя побитовые операторы, можно создавать функции, которые позволят уместить 8 значений типа bool в переменную размером 1 байт, что значительно сэкономит потребление памяти. В прошлом такой трюк был очень популярен. Но сегодня, по крайней мере, в прикладном программировании, это не так.

Теперь памяти стало существенно больше и программисты обнаружили, что лучше писать код так, чтобы было проще и понятнее его поддерживать, нежели усложнять его ради незначительной экономии памяти. Поэтому спрос на использование побитовых операторов несколько уменьшился, за исключением случаев, когда необходима уж максимальная оптимизация (например, научные программы, которые используют огромное количество данных; игры, где манипуляции с битами могут быть использованы для дополнительной скорости; встроенные программы, где память по-прежнему ограничена).

В языке C++ есть 6 побитовых операторов:

Оператор	Символ	Пример	Операция
Побитовый сдвиг влево	<<	x << y	Все биты в x смещаются влево на y бит
Побитовый сдвиг вправо	>>	x >> y	Все биты в x смещаются вправо на y бит
Побитовое НЕ	~	~x	Все биты в x меняются на противоположные
Побитовое И	&	x & y	Каждый бит в x И каждый соответствующий ему бит в y
Побитовое ИЛИ	|	x | y	Каждый бит в x ИЛИ каждый соответствующий ему бит в y
Побитовое исключающее ИЛИ (XOR)	^	x ^ y	Каждый бит в x XOR с каждым соответствующим ему битом в y

В побитовых операциях следует использовать только целочисленные типы данных unsigned, так как C++ не всегда гарантирует корректную работу побитовых операторов с целочисленными типами signed.

Правило: При работе с побитовыми операторами используйте целочисленные типы данных unsigned.

Побитовый сдвиг влево (<<) и побитовый сдвиг вправо (>>)

В языке C++ количество используемых бит основывается на размере типа данных (в 1 байте находятся 8 бит). Оператор побитового сдвига влево (`<<`) сдвигает биты влево. Левый операнд является выражением, в котором они сдвигаются, а правый — количество мест, на которые нужно сдвинуть. Поэтому в выражении `3 << 1` мы имеем в виду "сдвинуть биты влево в литерале 3 на одно место".

> ***Примечание:*** В следующих примерах мы будем работать с 4-битными двоичными значениями.

Рассмотрим число 3, которое в двоичной системе равно 0011:

```
3 = 0011
3 << 1 = 0110 = 6
3 << 2 = 1100 = 12
3 << 3 = 1000 = 8
```

В последнем третьем случае один бит перемещается за пределы самого литерала! Биты, сдвинутые за пределы двоичного числа, теряются навсегда.

Оператор побитового сдвига вправо (`>>`) сдвигает биты вправо. Например:

```
12 = 1100
12 >> 1 = 0110 = 6
12 >> 2 = 0011 = 3
12 >> 3 = 0001 = 1
```

В третьем случае мы снова переместили бит за пределы литерала. Он также потерялся навсегда.

Хотя в примерах, приведенных выше, мы смещаем биты только в литералах, мы также можем смещать биты и в переменных:

```
1. unsigned int x = 4;
2. x = x << 1; // x должен стать равным 8
```

Следует помнить, что результаты операций с побитовыми сдвигами в разных компиляторах могут отличаться.

Что!? Разве операторы << и >> используются не для вывода и ввода данных?

И для этого тоже.

Сейчас польза от использования побитовых операторов не так велика, как это было раньше. Сейчас в большинстве случаев оператор побитового сдвига влево используется для вывода данных. Например, рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     unsigned int x = 4;
6.     x = x << 1; // оператор << используется для побитового сдвига влево
7.     std::cout << x; // оператор << используется для вывода данных в консоль
```

```
8.
9.      return 0;
10.}
```

Результат выполнения программы:

```
8
```

А как компилятор понимает, когда нужно применить оператор побитового сдвига влево, а когда выводить данные? Всё очень просто. std::cout **переопределяет** значение оператора `<<` по умолчанию на новое (вывод данных в консоль). Когда компилятор видит, что левым операндом оператора `<<` является std::cout, то он понимает, что должен произойти вывод данных. Если левым операндом является переменная целочисленного типа данных, то компилятор понимает, что должен произойти побитовый сдвиг влево (операция по умолчанию).

Побитовый оператор НЕ

Побитовый оператор НЕ (`~`), пожалуй, самый простой для объяснения и понимания. Он просто меняет каждый бит на противоположный, например, с 0 на 1 или с 1 на 0. Обратите внимание, результаты побитового НЕ зависят от размера типа данных!

Предположим, что размер типа данных составляет 4 бита:

```
4 = 0100
~ 4 = 1011 (двоичное) = 11 (десятичное)
```

Предположим, что размер типа данных составляет 8 бит:

```
4 = 0000 0100
~ 4 = 1111 1011 (двоичное) = 251 (десятичное)
```

Побитовые операторы И, ИЛИ и исключающее ИЛИ (XOR)

Побитовые операторы И (`&`) и ИЛИ (`|`) работают аналогично логическим операторам И и ИЛИ. Однако, побитовые операторы применяются к каждому биту отдельно! Например, рассмотрим выражение `5 | 6`. В двоичной системе это `0101 | 0110`. В любой побитовой операции операнды лучше всего размещать следующим образом:

```
0 1 0 1 // 5
0 1 1 0 // 6
```

А затем применять операцию к каждому столбцу с битами по отдельности. Как вы помните, логическое ИЛИ возвращает true (1), если один из двух или оба операнды истинны (1). Аналогичным образом работает и **побитовое ИЛИ**. Выражение `5 | 6` обрабатывается следующим образом:

```
0 1 0 1 // 5
0 1 1 0 // 6
-------
0 1 1 1 // 7
```

Результат:

```
0111 (двоичное) = 7 (десятичное)
```

Также можно обрабатывать и комплексные выражения ИЛИ, например, `1 | 4 | 6`. Если хоть один бит в столбце равен 1, то результат целого столбца - 1. Например:

```
0 0 0 1 // 1
0 1 0 0 // 4
0 1 1 0 // 6
--------
0 1 1 1 // 7
```

Результатом `1 | 4 | 6` является десятичное `7`.

Побитовое И работает аналогично логическому И — возвращается true, только если оба бита в столбце равны 1. Рассмотрим выражение `5 & 6`:

```
0 1 0 1 // 5
0 1 1 0 // 6
--------
0 1 0 0 // 4
```

Также можно решать и комплексные выражения И, например, `1 & 3 & 7`. Только при условии, что все биты в столбце равны 1, результатом столбца будет 1.

```
0 0 0 1 // 1
0 0 1 1 // 3
0 1 1 1 // 7
--------
0 0 0 1 // 1
```

Последний оператор - **побитовое исключающее ИЛИ** (`^`) (сокр. **"XOR"** от англ. *"e**X**clusive **OR**"*). При обработке двух операндов, исключающее ИЛИ возвращает true (1), только если один и только один из операндов является истинным (1). Если таких

нет или все операнды равны 1, то результатом будет false (0). Рассмотрим выражение `6 ^ 3`:

```
0 1 1 0 // 6
0 0 1 1 // 3
-------
0 1 0 1 // 5
```

Также можно решать и комплексные выражения XOR, например, `1 ^ 3 ^ 7`. Если единиц в столбце чётное количество, то результатом будет 0, если же нечётное количество, то результат - 1. Например:

```
0 0 0 1 // 1
0 0 1 1 // 3
0 1 1 1 // 7
--------
0 1 0 1 // 5
```

Побитовые операторы присваивания

Как и в случае с арифметическими операторами присваивания, язык C++ предоставляет побитовые операторы присваивания для облегчения внесения изменений в переменные.

Оператор	Символ	Пример	Операция
Присваивание с побитовым сдвигом влево	<<=	x <<= y	Сдвигаем биты в x влево на y бит
Присваивание с побитовым сдвигом вправо	>>=	x >>= y	Сдвигаем биты в x вправо на y бит
Присваивание с побитовой операцией ИЛИ	|=	x |= y	Присваивание результата выражения x | y переменной x
Присваивание с побитовой операцией И	&=	x &= y	Присваивание результата выражения x & y переменной x

Присваивание с побитовой операцией исключающего ИЛИ	^=	x ^= y	Присваивание результата выражения x ^ y переменной x

Например, вместо `x = x << 1;` мы можем написать `x <<= 1;`.

Заключение

При работе с побитовыми операторами (используя метод столбца) не забывайте о том, что:

- При вычислении побитового ИЛИ, если хоть один из битов в столбце равен 1, то результат целого столбца равен 1.
- При вычислении побитового И, если все биты в столбце равны 1, то результат целого столбца равен 1.
- При вычислении побитового исключающего ИЛИ (XOR), если единиц в столбце нечётное количество, то результат равен 1.

Тест

Задание №1

Какой результат `0110 >> 2` в двоичной системе счисления?

Задание №2

Какой результат `5 | 12` в десятичной системе счисления?

Задание №3

Какой результат `5 & 12` в десятичной системе счисления?

Задание №4

Какой результат `5 ^ 12` в десятичной системе счисления?

Урок №49. Битовые флаги и битовые маски

На этом уроке мы рассмотрим битовые флаги и битовые маски в языке C++.

> ***Примечание*:** Для некоторых этот материал может показаться немного сложным. Если вы застряли или что-то не понятно — пропустите этот урок, в будущем сможете вернуться и разобраться детально. Он не столь важен для прогресса в изучении языка C++, как другие уроки, и изложен здесь в большей мере для общего развития.

Битовые флаги

Используя целый байт для хранения значения логического типа данных, вы занимаете только 1 бит, а остальные 7 из 8 — не используются. Хотя в целом это нормально, но в особых, ресурсоёмких случаях, связанных с множеством логических значений, может быть полезно "упаковать" 8 значений типа bool в 1 байт, сэкономив при этом память и увеличив, таким образом, производительность. Эти отдельные биты и называются **битовыми флагами**. Поскольку доступ к этим битам напрямую отсутствует, то для операций с ними используются побитовые операторы.

> ***Примечание*:** На этом уроке мы будем использовать значения из шестнадцатеричной системы счисления.

Например:

```
1.  // Определяем 8 отдельных битовых флагов (они могут представлять всё, что вы захотите).
2.  // Обратите внимание, в C++11 лучше использовать "uint8_t" вместо "unsigned char"
3.  const unsigned char option1 = 0x01; // шестнадцатеричный литерал для 0000 0001
4.  const unsigned char option2 = 0x02; // шестнадцатеричный литерал для 0000 0010
5.  const unsigned char option3 = 0x04; // шестнадцатеричный литерал для 0000 0100
6.  const unsigned char option4 = 0x08; // шестнадцатеричный литерал для 0000 1000
7.  const unsigned char option5 = 0x10; // шестнадцатеричный литерал для 0001 0000
8.  const unsigned char option6 = 0x20; // шестнадцатеричный литерал для 0010 0000
9.  const unsigned char option7 = 0x40; // шестнадцатеричный литерал для 0100 0000
10. const unsigned char option8 = 0x80; // шестнадцатеричный литерал для 1000 0000
11.
12. // Байтовое значения для хранения комбинаций из 8 возможных вариантов
13. unsigned char myflags = 0; // все флаги/параметры отключены до старта
```

Чтобы *узнать битовое состояние*, используется побитовое И:

```
1.  if (myflags & option4) ... // если option4 установлено - что-нибудь делаем
```

Чтобы *включить биты*, используется побитовое ИЛИ:

```
1.  myflags |= option4; // включаем option4
```

```
2. myflags |= (option4 | option5); // включаем option4 и option5
```

Чтобы *выключить биты*, используется побитовое И с инвертированным литералом:

```
1. myflags &= ~option4; // выключаем option4
2. myflags &= ~(option4 | option5); // выключаем option4 и option5
```

Для переключения между состояниями бит, используется побитовое исключающее ИЛИ (XOR):

```
1. myflags ^= option4; // включаем или выключаем option4
2. myflags ^= (option4 | option5); // изменяем состояния option4 и option5
```

В качестве примера возьмем библиотеку 3D-графики **OpenGL**, в которой некоторые функции принимают один или несколько битовых флагов в качестве параметров:

```
1. glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // очищаем буфер цвета и
   глубины
```

`GL_COLOR_BUFFER_BIT` и `GL_DEPTH_BUFFER_BIT` определяются следующим образом (в gl2.h):

```
1. #define GL_DEPTH_BUFFER_BIT               0x00000100
2. #define GL_STENCIL_BUFFER_BIT             0x00000400
3. #define GL_COLOR_BUFFER_BIT               0x00004000
```

Вот небольшой пример:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     // Определяем набор физических/эмоциональных состояний
6.     const unsigned char isHungry = 0x01; // шестнадцатеричный литерал для 0000 0001
7.     const unsigned char isSad = 0x02; // шестнадцатеричный литерал для 0000 0010
8.     const unsigned char isMad = 0x04; // шестнадцатеричный литерал для 0000 0100
9.     const unsigned char isHappy = 0x08; // шестнадцатеричный литерал для 0000 1000
10.    const unsigned char isLaughing = 0x10; // шестнадцатеричный литерал для 0001 0000
11.    const unsigned char isAsleep = 0x20; // шестнадцатеричный литерал для 0010 0000
12.    const unsigned char isDead = 0x40; // шестнадцатеричный литерал для 0100 0000
13.    const unsigned char isCrying = 0x80; // шестнадцатеричный литерал для 1000 0000
14.
15.    unsigned char me = 0; // все флаги/параметры отключены до старта
16.    me |= isHappy | isLaughing; // Я isHappy и isLaughing
17.    me &= ~isLaughing; // Я уже не isLaughing
18.
19.    // Запрашиваем сразу несколько состояний (мы будем использовать static_cast<bool> дл
   я конвертации результатов в значения типа bool)
20.    std::cout << "I am happy? " << static_cast<bool>(me & isHappy) << '\n';
21.    std::cout << "I am laughing? " << static_cast<bool>(me & isLaughing) << '\n';
22.
23.    return 0;
24. }
```

Почему битовые флаги полезны?

Внимательные читатели заметят, что в примерах с `myflags` мы фактически не экономим память. 8 логических значений займут 8 байт. Но пример, приведенный выше, использует 9 байт (8 для определения параметров и 1 для битового флага)! Так зачем же тогда нужны битовые флаги?

Они используются в двух случаях:

Случай №1: **Если у вас много идентичных битовых флагов.**

Вместо одной переменной `myflags`, рассмотрим случай, когда у вас есть две переменные: `myflags1` и `myflags2`, каждая из которых может хранить 8 значений. Если вы определите их как два отдельных логических набора, то вам потребуется 16 логических значений и, таким образом, 16 байт. Однако с использованием битовых флагов вам потребуется только 10 байт (8 для определения параметров и 1 для каждой переменной `myflags`). А вот если у вас будет 100 переменных `myflags`, то, используя битовые флаги, вам потребуется 108 байт вместо 800. Чем больше идентичных переменных вам нужно, тем более значительной будет экономия памяти.

Давайте рассмотрим конкретный пример. Представьте, что вы создаете игру, в которой игроку придется бороться с монстрами. Монстр, в свою очередь, может быть устойчив к определенным типам атак (выбранных случайным образом). В игре есть следующие типы атак: яд, молнии, огонь, холод, кража, кислота, паралич и слепота.

Чтобы отследить, к какому типу атаки монстр устойчив, мы можем использовать одно логическое значение на сопротивление (для одного монстра). Это 8 логических значений (сопротивлений) для одного монстра = 8 байт.

Для 100 монстров это будет 800 переменных типа bool и 800 байт памяти.

А вот используя битовые флаги:

```
1. const unsigned char resistsPoison    = 0x01;
2. const unsigned char resistsLightning = 0x02;
3. const unsigned char resistsFire      = 0x04;
4. const unsigned char resistsCold      = 0x08;
5. const unsigned char resistsTheft     = 0x10;
6. const unsigned char resistsAcid      = 0x20;
7. const unsigned char resistsParalysis = 0x40;
8. const unsigned char resistsBlindness = 0x80;
```

Нам нужен будет только 1 байт для хранения сопротивления каждого монстра и одноразовая плата в 8 байт для типов атак.

Таким образом, потребуется только 108 байт или примерно в 8 раз меньше памяти.

В большинстве программ, сохраненный объем памяти с использованием битовых флагов не стоит добавленной сложности. Но в программах, где есть десятки тысяч или даже миллионы похожих объектов, их использование может значительно сэкономить память. Согласитесь, знать о таком полезном трюке не помешает.

Случай №2: Представьте, что у вас есть функция, которая может принимать любую комбинацию из 32 различных вариантов. Одним из способов написания такой функции является использование 32 отдельных логических параметров:

```
1. void someFunction(bool option1, bool option2, bool option3, bool option4,
   bool option5, bool option6, bool option7, bool option8, bool option9, bool
   option10, bool option11, bool option12, bool option13, bool option14, bool
   option15, bool option16, bool option17, bool option18, bool option19, bool
   option20, bool option21, bool option22, bool option23, bool option24, bool
   option25, bool option26, bool option27, bool option28, bool option29, bool
   option30, bool option31, bool option32);
```

Затем, если вы захотите вызвать функцию с 10-м и 32-м параметрами, установленными как true — вам придется сделать следующее:

```
1. someFunction(false, false, false, false, false, false, false, false, false,
   true, false, false, false, false, false, false, false, false, false, false,
   false, false, false, false, false, false, false, false, false, false, false,
   true);
```

Т.е. перечислить все варианты как false, кроме 10 и 32 — они true. Читать такой код сложно, да и требуется держать в памяти порядковые номера нужных параметров (10 и 32 или 11 и 33?). Такая простыня не может быть эффективной.

А вот если определить функцию с помощью битовых флагов:

```
1. void someFunction(unsigned int options);
```

То можно выбирать и передавать только нужные параметры:

```
1. someFunction(option10 | option32);
```

Кроме того, что это читабельнее, это также эффективнее и производительнее, так как включает только 2 операции (1 побитовое ИЛИ и 1 передача параметров).

Вот почему в OpenGL используют битовые флаги вместо длинной последовательности логических значений.

Также, если у вас есть неиспользуемые битовые флаги и вам нужно позже добавить параметры, вы можете просто определить битовый флаг. Нет необходимости изменять прототип функции, а это плюс к обеспечению обратной совместимости.

Введение в std::bitset

Все эти биты, битовые флаги, операции-манипуляции - всё это утомительно, не правда ли? К счастью, в Стандартной библиотеке C++ есть такой объект, как **std::bitset**, который упрощает работу с битовыми флагами.

Для его использования необходимо подключить заголовочный файл bitset, а затем определить переменную типа std::bitset, указав необходимое количество бит. Она должна быть константой типа compile time.

```
1. #include <bitset>
2.
3. std::bitset<8> bits; // нам нужно 8 бит
```

При желании std::bitset можно инициализировать начальным набором значений:

```
1. #include <bitset>
2.
3. std::bitset<8> bits(option1 | option2) ; // начнем с включенных option1 и
   option2
4. std::bitset<8> morebits(0x2) ; // начнем с битового шаблона 0000 0010
```

Обратите внимание, наше начальное значение конвертируется в двоичную систему. Так как мы ввели шестнадцатеричное 2, то std::bitset преобразует его в двоичное 0000 0010.

В std::bitset есть 4 основные функции:

- **функция test()** — позволяет узнать значение бита (0 или 1).
- **функция set()** — позволяет *включить* биты (если они уже включены, то ничего не произойдет).
- **функция reset()** — позволяет *выключить* биты (если они уже выключены, то ничего не произойдет).
- **функция flip()** — позволяет изменить значения бит на противоположные (с 0 на 1 или с 1 на 0).

Каждая из этих функций принимает в качестве параметров позиции бит. Позиция крайнего правого бита (последнего) равна 0, затем порядковый номер растет с каждым последующим битом влево (1, 2, 3, 4 и т.д.). Старайтесь давать содержательные имена битовым индексам (либо путем присваивания их

константным переменным, либо с помощью перечислений — о них мы поговорим на соответствующем уроке).

```
#include <iostream>
#include <bitset>

// Обратите внимание, используя std::bitset, наши options соответствуют порядковым номерам бит, а не их значениям
const int option_1 = 0;
const int option_2 = 1;
const int option_3 = 2;
const int option_4 = 3;
const int option_5 = 4;
const int option_6 = 5;
const int option_7 = 6;
const int option_8 = 7;

int main()
{
    // Помните, что отсчет бит начинается не с 1, а с 0
    std::bitset<8> bits(0x2); // нам нужно 8 бит, начнем с битового шаблона 0000 0010
    bits.set(option_5); // включаем 4-й бит - его значение изменится на 1 (теперь мы имеем 0001 0010)
    bits.flip(option_6); // изменяем значения 5-го бита на противоположное (теперь мы имеем 0011 0010)
    bits.reset(option_6); // выключаем 5-й бит - его значение снова 0 (теперь мы имеем 0001 0010)

    std::cout << "Bit 4 has value: " << bits.test(option_5) << '\n';
    std::cout << "Bit 5 has value: " << bits.test(option_6) << '\n';
    std::cout << "All the bits: " << bits << '\n';

    return 0;
}
```

Результат выполнения программы:

```
Bit 4 has value: 1
Bit 5 has value: 0
All the bits: 00010010
```

Обратите внимание, отправляя переменную `bits` в std::cout - выводятся значения всех бит в std::bitset.

Помните, что инициализируемое значение std::bitset рассматривается как двоичное, в то время как функции std::bitset используют позиции бит!

std::bitset также поддерживает стандартные побитовые операторы (`|`, `&` и `^`), которые также можно использовать (они полезны при проведении операций одновременно сразу с несколькими битами).

Вместо выполнения всех побитовых операций вручную, рекомендуется использовать std::bitset, так как он более удобен и менее подвержен ошибкам.

Битовые маски

Включение, выключение, переключение или запрашивание сразу нескольких бит можно осуществить в одной битовой операции. Когда мы соединяем отдельные биты вместе, в целях их модификации как группы, то это называется **битовой маской**.

Рассмотрим пример. В следующей программе мы просим пользователя ввести число. Затем, используя битовую маску, мы сохраняем только последние 4 бита, значения которых и выводим в консоль:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     const unsigned int lowMask = 0xF; // битовая маска для хранения последних
   4-х бит (шестнадцатеричный литерал для 0000 0000 0000 1111)
6.
7.     std::cout << "Enter an integer: ";
8.     int num;
9.     std::cin >> num;
10.
11.    num &= lowMask; // удаляем первые биты, оставляя последние 4
12.
13.    std::cout << "The 4 low bits have value: " << num << '\n';
14.
15.    return 0;
16.}
```

Результат выполнения программы:

```
Enter an integer: 151
The 4 low bits have value: 7
```

`151` в десятичной системе равно `1001 0111` в двоичной. `lowMask` — это `0000 1111` в 8-битной двоичной системе. `1001 0111 & 0000 1111 = 0000 0111`, что равно десятичному `7`.

Пример с RGBA

Цветные дисплейные устройства, такие как телевизоры и мониторы, состоят из миллионов пикселей, каждый из которых может отображать точку цвета. Точка цвета состоит из 3-х пучков: один красный, один зелёный и один синий (сокр. **"RGB"** от англ. *"**R**ed, **G**reen, **B**lue"*). Изменяя их интенсивность, можно воссоздать любой цвет. Количество цветов R, G и B в одном пикселе представлено 8-битным целым

числом unsigned. Например, красный цвет имеет R = 255, G = 0, B = 0; фиолетовый: R = 255, G = 0, B = 255; серый: R = 127, G = 127, B = 127.

Используется еще 4-е значение, которое называется A. **"A"** от англ. *"**A**lfa"*, которое отвечает за прозрачность. Если A = 0, то цвет полностью прозрачный. Если A = 255, то цвет непрозрачный.

В совокупности R, G, B и A составляют одно 32-битное целое число, с 8 битами для каждого компонента:

32-битное значение RGBA			
31-24 бита	23-16 бит	15-8 бит	7-0 бит
RRRRRRRR	GGGGGGGG	BBBBBBBB	AAAAAAAA
red	green	blue	alpha

Следующая программа просит пользователя ввести 32-битное шестнадцатеричное значение, а затем извлекает 8-битные цветовые значения R, G, B и A:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     const unsigned int redBits = 0xFF000000;
6.     const unsigned int greenBits = 0x00FF0000;
7.     const unsigned int blueBits = 0x0000FF00;
8.     const unsigned int alphaBits = 0x000000FF;
9.
10.    std::cout << "Enter a 32-
   bit RGBA color value in hexadecimal (e.g. FF7F3300): ";
11.    unsigned int pixel;
12.    std::cin >> std::hex >> pixel; // std::hex позволяет вводить
   шестнадцатеричные значения
13.
14.    // Используем побитовое И для изоляции красных пикселей, а затем сдвигаем
   значение вправо в диапазон 0-255
15.    unsigned char red = (pixel & redBits) >> 24;
16.    unsigned char green = (pixel & greenBits) >> 16;
17.    unsigned char blue = (pixel & blueBits) >> 8;
18.    unsigned char alpha = pixel & alphaBits;
19.
20.    std::cout << "Your color contains:\n";
21.    std::cout << static_cast<int>(red) << " of 255 red\n";
22.    std::cout << static_cast<int>(green) << " of 255 green\n";
23.    std::cout << static_cast<int>(blue) << " of 255 blue\n";
24.    std::cout << static_cast<int>(alpha) << " of 255 alpha\n";
```

```
25.
26.     return 0;
27. }
```

Результат выполнения программы:

```
Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300):
FF7F3300
Your color contains:
255 of 255 red
127 of 255 green
51 of 255 blue
0 of 255 alpha
```

В программе, приведенной выше, побитовое И используется для запроса 8-битного набора, который нас интересует, затем мы его сдвигаем вправо в диапазон от 0 до 255 для хранения и вывода.

> ***Примечание***: RGBA иногда может храниться как ARGB. В таком случае, главным байтом является альфа.

Заключение

Давайте кратко повторим то, как включать, выключать, переключать и запрашивать битовые флаги.

Для запроса битового состояния используется побитовое И:

```
1. if (myflags & option4) ... // если установлен option4, то делаем что-нибудь
```

Для включения бит используется побитовое ИЛИ:

```
1. myflags |= option4; // включаем option4
2. myflags |= (option4 | option5); // включаем option4 и option5
```

Для выключения бит используется побитовое И с инвертированным литералом:

```
1. myflags &= ~option4; // выключаем option4
2. myflags &= ~(option4 | option5); // выключаем option4 и option5
```

Для переключения между битовыми состояниями используется побитовое исключающее ИЛИ (XOR):

```
1. myflags ^= option4; // включаем или выключаем option4
2. myflags ^= (option4 | option5); // изменяем на противоположные option4 и
   option5
```

Тест

Есть следующий фрагмент кода:

```
1. int main()
2. {
3.     unsigned char option_viewed = 0x01;
4.     unsigned char option_edited = 0x02;
5.     unsigned char option_favorited = 0x04;
6.     unsigned char option_shared = 0x08;
7.     unsigned char option_deleted = 0x80;
8.
9.     unsigned char myArticleFlags;
10.
11.    return 0;
12.}
```

> ***Примечание***: Статья — это `myArticleFlags`.

Задание №1

Добавьте строку кода, чтобы пометить статью как уже прочитанную (`option_viewed`).

Задание №2

Добавьте строку кода, чтобы проверить, была ли статья удалена (`option_deleted`).

Задание №3

Добавьте строку кода, чтобы открепить статью от закрепленного места (`option_favorited`).

Задание №4

Почему следующие две строки идентичны?

```
1. myflags &= ~(option4 | option5); // выключаем option4 и option5
2. myflags &= ~option4 & ~option5; // выключаем option4 и option5
```

Глава №3. Итоговый тест

Поздравляю вас! Еще одна глава позади. Сейчас мы вкратце повторим то, чему научились в этой главе, а затем закрепим пройденный материал на практике.

Теория

Всегда используйте круглые скобки для устранения возможных проблем с приоритетами операторов и порядком их выполнения.

Арифметические операторы в языке C++ работают так же, как и в обычной математике. Оператор `%` возвращает остаток от целочисленного деления. Остерегайтесь ошибок округления, когда операнды целочисленного деления и остатка от деления - отрицательны.

Операторы инкремента (`++`) и декремента (`--`) используются для увеличения или уменьшения числа. Остерегайтесь побочных эффектов, особенно когда дело доходит до порядка, в котором будут обрабатываться параметры функции. Не используйте переменную с побочным эффектом больше одного раза в одном стейтменте.

Операторы сравнения позволяют сравнивать числа типа с плавающей точкой. Остерегайтесь использования операторов равенства и неравенства с ними.

Логические операторы позволяют формировать сложные условные стейтменты. Побитовые операторы позволяют работать на уровне отдельных бит.

Задание №1

Вычислите результат следующих выражений:

- `(5 > 3 && 4 < 8)`
- `(4 > 6 && true)`
- `(3 >= 3 || false)`
- `(true || false) ? 4 : 5`

Задание №2

Вычислите результат следующих выражений:

- `7 / 4`
- `14 % 5`

Задание №3

Конвертируйте следующие двоичные числа в десятичную систему счисления:

- 1101
- 101110

Задание №4

Конвертируйте следующие десятичные числа в двоичную систему счисления:

- 15
- 53

Задание №5

Почему вы никогда не должны делать следующее:

- `int y = foo(++x, x);`
- `int x = 7 / -2; // (до C++11)`
- `int x = -5 % 2; // (до C++11)`
- `float x = 0.1 + 0.1; if (x == 0.2) return true; else return false;`
- `int x = 3 / 0;`

Урок №50. Блоки стейтментов (составные операторы)

Блоки стейтментов (или ***"составные операторы"***) — это группа стейтментов, которые обрабатываются компилятором как одна инструкция. Блок начинается с символа `{` и заканчивается символом `}`, стейтменты находятся внутри. Блоки могут использоваться в любом месте, где разрешено использовать один стейтмент. В конце составного оператора точка с запятой не ставится.

Вы уже видели пример блоков при написании функций, поскольку тело функции является блоком:

```
1. int add(int x, int y)
2. { // начало блока
3.     return x + y;
4. } // конец блока
5.
6. int main()
7. { // начало блока
8.
9.     // Несколько стейтментов
10.    int value(0);
11.    add(3, 4);
12.
13.    return 0;
14.
15.} // конец блока (без точки с запятой)
```

Вложенные блоки

Хотя функции не могут быть вложены в другие функции, блоки *могут* быть вложены в другие блоки:

```
1. int add(int x, int y)
2. { // начало блока
3.     return x + y;
4. } // конец блока
5.
6. int main()
7. { // начало внешнего блока
8.
9.     // Несколько стейтментов
10.    int value {};
11.
12.    { // начало внутреннего/вложенного блока
13.        add(3, 4);
14.    } // конец внутреннего/вложенного блока
15.
16.    return 0;
17.
18.} // конец внешнего блока
```

При использовании вложенных блоков, блок, который содержит внутри себя другой блок, называется **внешним блоком**, а тот, который содержится внутри этого блока — **внутренний/вложенный блок**.

Блоки и операторы if

Один из наиболее распространенных вариантов использования блоков связан с операторами if. По умолчанию оператор if выполняет один стейтмент, если условие имеет значение true. С помощью блока мы можем сделать так, чтобы выполнялось сразу несколько стейтментов, если условие имеет значение true, например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Enter an integer: ";
6.     int value;
7.     std::cin >> value;
8.
9.     if (value >= 0)
10.    { // начало вложенного блока
11.        std::cout << value << " is a positive integer (or zero)" << std::endl;

12.        std::cout << "Double this number is " << value * 2 << std::endl;
13.    } // конец вложенного блока
14.    else
15.    { // начало другого вложенного блока
16.        std::cout << value << " is a negative integer" << std::endl;
17.        std::cout << "The positive of this number is " << -
   value << std::endl;
18.    } // конец другого вложенного блока
19.
20.    return 0;
21.}
```

Если ввести число `3`, то программа выведет:

```
Enter an integer: 3
3 is a positive integer (or zero)
Double this number is 6
```

Если ввести число `-4`, то программа выведет:

```
Enter an integer: -4
-4 is a negative integer
The positive of this number is 4
```

Количество уровней вложенности блоков

Можно даже размещать вложенные блоки внутри других вложенных блоков:

```
1. #include <iostream>
2.
3. int main()
4. { // 1-й уровень вложенности блоков
5.     std::cout << "Enter an integer: ";
6.     int value {};
7.     std::cin >> value;
8.
9.     if (value >  0)
10.    { // 2-й уровень вложенности блоков
11.        if ((value % 2) == 0)
12.        { // 3-й уровень вложенности блоков
13.            std::cout << value << " is positive and even\n";
14.        }
15.        else
16.        { // также 3-й уровень вложенности блоков
17.            std::cout << value << " is positive and odd\n";
18.        }
19.    }
20.
21.    return 0;
22.}
```

Уровень вложенности функции (или ***"глубина вложенности функции"***) — это максимальное количество блоков, которые могут находиться в любой точке функции (включая внешний блок). В вышеприведенной функции есть 4 блока, но уровень вложенности равен 3.

По факту, ограничений на количество вложенных блоков нет. Однако не рекомендуется делать больше 3-х уровней вложенности (максимум 4). Если ваша функция нуждается в большем количестве уровней вложенности, то эту функцию лучше разбить на несколько подфункций!

Заключение

Блоки стейтментов позволяют выполнить сразу несколько стейтментов там, где можно использовать лишь один. Они чрезвычайно полезны, когда нужно выполнить сразу несколько инструкций вместе.

Урок №51. Локальные переменные, область видимости и продолжительность жизни

Прежде чем мы начнем, нам нужно сначала разобраться с двумя терминами: область видимости и продолжительность жизни. **Область видимости** определяет, где можно использовать переменную. **Продолжительность жизни** (или ***"время жизни"***) определяет, где переменная создается и где уничтожается. Эти две концепции связаны между собой.

Переменные, определенные внутри блока, называются **локальными переменными**. Локальные переменные имеют **автоматическую продолжительность жизни**: они создаются (и инициализируются, если необходимо) в точке определения и уничтожаются при выходе из блока. Локальные переменные имеют **локальную область видимости** (или ***"блочную"***), т.е. они входят в область видимости с точки объявления и выходят в самом конце блока, в котором определены.

Например, рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int x(4); // переменная x создается и инициализируется здесь
6.     double y(5.0); // переменная y создается и инициализируется здесь
7.
8.     return 0;
9.
10.} // x и y выходят из области видимости и уничтожаются здесь
```

Поскольку переменные `x` и `y` определены внутри блока, который является главной функцией, то они обе уничтожаются, когда main() завершает свое выполнение.

Переменные, определенные внутри вложенных блоков, уничтожаются, как только заканчивается вложенный блок:

```
1. #include <iostream>
2.
3. int main() // внешний блок
4. {
5.     int m(4); // переменная m создается и инициализируется здесь
6.
7.     { // начало вложенного блока
8.         double k(5.0); // переменная k создается и инициализируется здесь
9.     } // k выходит из области видимости и уничтожается здесь
10.
11.    // Переменная k не может быть использована здесь, так как она уже
   уничтожена!
```

```
12.
13.     return 0;
14. } // переменная m выходит из области видимости и уничтожается здесь
```

Такие переменные можно использовать только внутри блоков, в которых они определены. Поскольку каждая функция имеет свой собственный блок, то переменные из одной функции никак не соприкасаются и не влияют на переменные из другой функции:

```
1.  #include <iostream>
2.
3.  void someFunction()
4.  {
5.      int value(5); // value определяется здесь
6.
7.      // value можно использовать здесь
8.
9.  } // value выходит из области видимости и уничтожается здесь
10.
11. int main()
12. {
13.     // value нельзя использовать внутри этой функции
14.
15.     someFunction();
16.
17.     // value здесь также нельзя использовать
18.
19.     return 0;
20. }
```

В разных функциях могут находиться переменные или параметры с одинаковыми именами. Это хорошо, потому что не нужно беспокоиться о возможности возникновения конфликтов имен между двумя независимыми функциями. В примере, приведенном ниже, в обеих функциях есть переменные `x` и `y`. Они даже не подозревают о существовании друг друга:

```
1.  #include <iostream>
2.
3.  // Параметр x можно использовать только внутри функции add()
4.  int add(int x, int y) // параметр x функции add() создается здесь
5.  {
6.      return x + y;
7.  } // параметр x функции add() уничтожается здесь
8.
9.  // Переменную x функции main() можно использовать только внутри функции main()
10. int main()
11. {
12.     int x = 5; // переменная x функции main() создается здесь
13.     int y = 6;
14.     std::cout << add(x, y) << std::endl; // значение x функции main()
    копируется в переменную x функции add()
15.     return 0;
16. } // переменная x функции main() уничтожается здесь
```

Вложенные блоки считаются частью внешнего блока, в котором они определены. Следовательно, переменные, определенные во внешнем блоке, могут быть видны и внутри вложенного блока:

```
#include <iostream>

int main()
{ // начало внешнего блока

    int x(5);

    { // начало вложенного блока
        int y(7);
        // Мы можем использовать x и y здесь
        std::cout << x << " + " << y << " = " << x + y;
    } // переменная y уничтожается здесь

    // Переменную y здесь нельзя использовать, поскольку она уже уничтожена!

    return 0;
} // переменная x уничтожается здесь
```

Сокрытие имен

Переменная внутри вложенного блока может иметь то же имя, что и переменная внутри внешнего блока. Когда подобное случается, то переменная во вложенном (внутреннем) блоке «скрывает» внешнюю переменную. Это называется **сокрытием имен**:

```
#include <iostream>

int main()
{ // внешний блок
    int oranges(5); // внешняя переменная oranges

    if (oranges >= 5) // относится к внешней oranges
    { // вложенный блок
        int oranges; // скрывается внешняя переменная oranges

        // Идентификатор oranges теперь относится к вложенной переменной oranges.
        // Внешняя переменная oranges временно скрыта

        oranges = 10; // здесь мы присваиваем значение 10 вложенной переменной oranges, не внешней!

        std::cout << oranges << std::endl; // выводим значение вложенной переменной oranges
    } // вложенная переменная oranges уничтожается

    // Идентификатор oranges опять относится к внешней переменной oranges

    std::cout << oranges << std::endl; // выводим значение внешней переменной oranges

    return 0;
```

```
24. } // внешняя переменная oranges уничтожается
```

Результат выполнения программы:

```
10
5
```

Здесь мы сначала объявляем переменную `oranges` во внешнем блоке. Затем объявляем вторую переменную `oranges`, но уже во вложенном (внутреннем) блоке. Когда мы присваиваем `oranges` значение `10`, то оно относится к переменной во вложенном блоке. После вывода этого значения (и окончания внутреннего блока), внутренняя переменная `oranges` уничтожается, оставляя внешнюю `oranges` с исходным значением (`5`), которое затем выводится. Результат выполнения программы был бы тот же, даже если бы мы назвали вложенную переменную по-другому (например, `nbOranges`).

Обратите внимание, если бы мы не определили вложенную переменную `oranges`, то идентификатор `oranges` относился бы к внешней переменной и значение `10` было бы присвоено внешней переменной:

```
1. #include <iostream>
2.
3. int main()
4. { // внешний блок
5.     int oranges(5); // внешняя переменная oranges
6.
7.     if (oranges >= 5) // относится к внешней переменной oranges
8.     { // вложенный блок
9.         // Никакого определения внутренней переменной oranges здесь нет
10.
11.        oranges = 10; // это применяется к внешней переменной oranges, хоть мы и находимся во вложенном блоке
12.
13.        std::cout << oranges << std::endl; // выводим значение внешней переменной oranges
14.    } // значением переменной oranges будет 10 даже после того, как мы выйдем из вложенного блока
15.
16.    std::cout << oranges << std::endl; // выводим значение переменной oranges
17.
18.    return 0;
19. } // переменная oranges уничтожается
```

Результат выполнения программы:

```
10
10
```

В обоих примерах на внешнюю переменную `oranges` никак не влияет то, что происходит с вложенной переменной `oranges`. Единственное различие между двумя программами - это то, к чему применяется выражение `oranges = 10`.

Сокрытие имен - это то, чего, как правило, следует избегать, поскольку оно может быть довольно запутанным!

Правило: Избегайте использования вложенных переменных с именами, идентичными именам внешних переменных.

Область видимости переменных

Переменные должны определяться в максимально ограниченной области видимости. Например, если переменная используется только внутри вложенного блока, то она и должна быть определена в нем:

```
#include <iostream>

int main()
{
    // Не определяйте x здесь

    {
        // Переменная x используется только внутри этого блока, поэтому определяем её здесь
        int x(7);
        std::cout << x;
    }

    // В противном случае, переменная x может быть использована и здесь

    return 0;
}
```

Ограничивая область видимости, мы уменьшаем сложность программы, поскольку число активных переменных уменьшается. Таким образом, легче увидеть, где какие переменные используются. Переменная, определенная внутри блока, может использоваться только внутри этого же блока (или вложенных в него подблоков). Этим мы упрощаем понимание и логику программы.

Если во внешнем блоке нужна переменная, то её необходимо объявлять во внешнем блоке:

```
#include <iostream>

int main()
{
    int y(5); // мы объявляем переменную y здесь, потому что она нам будет нужна как во внутреннем, так и во внешнем блоке чуть позже

```

```
    {
        int x;
        std::cin >> x;
        // Если бы мы объявили y здесь, непосредственно перед её первым
фактическим использованием,
        if (x == 4)
            y = 4;
    } // то она бы уничтожилась здесь

    std::cout << y; // а переменная y нам нужна еще здесь

    return 0;
}
```

Это один из тех редких случаев, когда вам может понадобиться объявить переменную до её первого использования.

Правило: Определяйте переменные в наиболее ограниченной области видимости.

Параметры функций

Хотя параметры функций не определяются внутри основного блока (тела) функции, в большинстве случаев они имеют локальную область видимости:

```
int max(int x, int y) // x и y определяются здесь
{
    // Присваиваем большее из значений (x или y) переменной max
    int max = (x > y) ? x : y; // max определяется здесь
    return max;
} // x, y и max уничтожаются здесь
```

Заключение

Переменные, определенные внутри блоков, называются локальными переменными. Они доступны только внутри блока, в котором определены (включая вложенные блоки) и уничтожаются при завершении этого же блока.

Определяйте переменные в наиболее ограниченной области видимости. Если переменная используется только внутри вложенного блока, то и определять её следует внутри этого же вложенного блока.

Тест

Задание №1

Напишите программу, которая просит пользователя ввести два целых числа: второе должно быть больше первого. Если пользователь введет второе число меньше первого, то используйте блок и временную переменную, чтобы поменять местами

пользовательские числа. Затем выведите значения этих переменных. Добавьте в свой код комментарии, объясняющие, где и какая переменная уничтожается.

Результат выполнения программы должен быть примерно следующим:

```
Введите число: 4
Введите большее число: 2
Меняем значения местами
Меньшее число: 2
Большее число: 4
```

> ***Подсказка***: Чтобы использовать кириллицу, добавьте следующую строчку кода в самое начало функции main():
>
> ```
> setlocale(LC_ALL, "rus");
> ```

Задание №2

В чём разница между областью видимости и продолжительностью жизни переменной? Какую область видимости и продолжительность жизни по умолчанию имеют локальные переменные (и что это значит)?

Урок №52. Глобальные переменные

Мы уже знаем, что переменные, объявленные внутри блока, называются локальными. Они имеют локальную область видимости (используются только внутри блока, в котором объявлены) и автоматическую продолжительность жизни (создаются в точке определения и уничтожаются в конце блока).

Глобальными называются переменные, которые объявлены вне блока. Они имеют **статическую продолжительность жизни**, т.е. создаются при запуске программы и уничтожаются при её завершении. Глобальные переменные имеют **глобальную область видимости** (или ***"файловую область видимости"***), т.е. их можно использовать в любом месте файла, после их объявления.

Определение глобальных переменных

Обычно глобальные переменные объявляют в верхней части кода, ниже директив #include, но выше любого другого кода. Например:

```
#include <iostream>

// Переменные, определенные вне блока, являются глобальными переменными
int g_x; // глобальная переменная g_x
const int g_y(3); // константная глобальная переменная g_y

void doSomething()
{
    // Глобальные переменные можно использовать в любом месте программы
    g_x = 4;
    std::cout << g_y << "\n";
}

int main()
{
    doSomething();

    // Глобальные переменные можно использовать в любом месте программы
    g_x = 7;
    std::cout << g_y << "\n";

    return 0;
}
```

Подобно тому, как переменные во внутреннем блоке скрывают переменные с теми же именами во внешнем блоке, локальные переменные скрывают глобальные переменные с одинаковыми именами внутри блока, в котором они определены. Однако с помощью **оператора разрешения области видимости** (`::`), компилятору можно сообщить, какую версию переменной вы хотите использовать: глобальную или локальную.

Например:

```
#include <iostream>

int value(4); // глобальная переменная

int main()
{
    int value = 8; // эта переменная (локальная) скрывает значение глобальной переменной
    value++; // увеличивается локальная переменная value (не глобальная)
    ::value--; // уменьшается глобальная переменная value (не локальная)

    std::cout << "Global value: " << ::value << "\n";
    std::cout << "Local value: " << value << "\n";
    return 0;
} // локальная переменная уничтожается
```

Результат выполнения программы:

```
Global value: 3
Local value: 9
```

Использовать одинаковые имена для локальных и глобальных переменных — это прямой путь к проблемам и ошибкам, поэтому подобное делать не рекомендуется. Многие разработчики добавляют к глобальным переменным префикс `g_` (**"g"** от англ. *"**g**lobal"*). Таким образом, можно убить сразу двух зайцев: определить глобальные переменные и избежать конфликтов имен с локальными переменными.

Ключевые слова static и extern

В дополнение к области видимости и продолжительности жизни, переменные имеют еще одно свойство — связь. **Связь переменной** определяет, относятся ли несколько упоминаний одного идентификатора к одной и той же переменной или нет.

Переменная без связей — это переменная с локальной областью видимости, которая относится только к блоку, в котором она определена. Это обычные локальные переменные. Две переменные с одинаковыми именами, но определенные в разных функциях, не имеют никакой связи — каждая из них считается независимой единицей.

Переменная, имеющая внутренние связи, называется **внутренней переменной** (или ***"статической переменной"***). Она может использоваться в любом месте файла, в котором определена, но не относится к чему-либо вне этого файла.

Переменная, имеющая внешние связи, называется **внешней переменной**. Она может использоваться как в файле, в котором определена, так и в других файлах.

Если вы хотите сделать глобальную переменную внутренней (которую можно использовать только внутри одного файла) — используйте **ключевое слово static**:

```
#include <iostream>

static int g_x; // g_x - это статическая глобальная переменная, которую можно использовать только внутри этого файла

int main()
{
    return 0;
}
```

Аналогично, если вы хотите сделать глобальную переменную внешней (которую можно использовать в любом файле программы) — используйте **ключевое слово extern**:

```
#include <iostream>

extern double g_y(9.8); // g_y - это внешняя глобальная переменная и её можно использовать и в других файлах программы

int main()
{
    return 0;
}
```

По умолчанию, неконстантные переменные, объявленные вне блока, считаются внешними. Однако константные переменные, объявленные вне блока, считаются внутренними.

Предварительные объявления переменных с использованием extern

Как мы уже знаем из предыдущих уроков, для использования функций, которые определены в другом файле, нужно применять предварительные объявления.

Аналогично, чтобы использовать внешнюю глобальную переменную, которая была объявлена в другом файле, нужно записать предварительное объявление переменной с использованием ключевого слова extern (без инициализируемого значения). Например:

global.cpp:

```
// Определяем две глобальные переменные
int g_m; // неконстантные глобальные переменные имеют внешнюю связь по умолчанию
```

```
3. int g_n(3); // неконстантные глобальные переменные имеют внешнюю связь по умолчанию
4. // g_m и g_n можно использовать в любом месте этого файла
```

main.cpp:

```
1. #include <iostream>
2.
3. extern int g_m; // предварительное объявление g_m. Теперь g_m можно использовать в любом месте этого файла
4.
5. int main()
6. {
7.     extern int g_n; // предварительное объявление g_n. Теперь g_n можно использовать только внутри main()
8.
9.     g_m = 4;
10.    std::cout << g_n; // должно вывести 3
11.
12.    return 0;
13. }
```

Если предварительное объявление находится вне блока, то оно применяется ко всему файлу. Если же внутри блока, то оно применяется только к нему.

Если переменная объявлена с помощью ключевого слова static, то получить доступ к ней с помощью предварительного объявления не получится. Например:

constants.cpp:

```
1. static const double g_gravity(9.8);
```

main.cpp:

```
1. #include <iostream>
2.
3. extern const double g_gravity; // не найдет g_gravity в constants.cpp, так как g_gravity является внутренней переменной
4.
5. int main()
6. {
7.     std:: cout << g_gravity; // вызовет ошибку компиляции, так как переменная g_gravity не была определена для использования в main.cpp
8.     return 0;
9. }
```

Обратите внимание, если вы хотите определить неинициализированную неконстантную глобальную переменную, то не используйте ключевое слово extern, иначе C++ будет думать, что вы пытаетесь записать предварительное объявление.

Связи функций

Функции имеют такие же свойства связи, что и переменные. По умолчанию они имеют внешнюю связь, которую можно сменить на внутреннюю с помощью ключевого слова static:

```
// Эта функция определена как static и может быть использована только внутри этого файла.
// Попытки доступа к ней через прототип функции будут безуспешными
static int add(int a, int b)
{
    return a + b;
}
```

Предварительные объявления функций не нуждаются в ключевом слове extern. Компилятор может определить сам (по телу функции): определяете ли вы функцию или пишете её прототип.

Файловая область видимости vs. Глобальная область видимости

Термины "файловая область видимости" и "глобальная область видимости", как правило, вызывают недоумение, и это отчасти объясняется их неофициальным использованием. В теории, в языке C++ все глобальные переменные имеют файловую область видимости. Однако, по факту, термин "файловая область видимости" чаще применяется к внутренним глобальным переменным, а "глобальная область видимости" — к внешним глобальным переменным.

Например, рассмотрим следующую программу:

global.cpp:

```
int g_y(3); // внешняя связь по умолчанию
```

main.cpp:

```
#include <iostream>

extern int g_y; // предварительное объявление g_y. Теперь g_y можно использовать в любом месте этого файла

int main()
{
    std::cout << g_y; // должно вывести 3

    return 0;
}
```

Переменная `g_y` имеет файловую область видимости внутри global.cpp. Доступ к этой переменной вне файла global.cpp отсутствует. Обратите внимание, хотя эта

переменная и используется в main.cpp, сам main.cpp не видит её, он видит только предварительное объявление `g_y` (которое также имеет файловую область видимости). Линкер отвечает за связывание определения `g_y` в global.cpp с использованием `g_y` в main.cpp.

Глобальные символьные константы

На уроке о символьных константах, мы определяли их следующим образом:

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// Определяем отдельное пространство имен для хранения констант
namespace Constants
{
    const double pi(3.14159);
    const double avogadro(6.0221413e23);
    const double my_gravity(9.2);
    // ... другие константы
}
#endif
```

Хоть это просто и отлично подходит для небольших программ, но каждый раз, когда constants.h подключается в другой файл, каждая из этих переменных копируется в этот файл. Таким образом, если constants.h подключить в 20 различных файлов, то каждая из переменных продублируется 20 раз. Header guards не остановят это, так как они только предотвращают подключение заголовочного файла более одного раза в один файл. Дублирование переменных на самом деле не является проблемой (поскольку константы зачастую не занимают много памяти), но изменение значения одной константы потребует перекомпиляции каждого файла, в котором она используется, что может привести к большим временным затратам в более крупных проектах.

Избежать этой проблемы можно, превратив эти константы в константные глобальные переменные, и изменив заголовочный файл только для хранения предварительных объявлений переменных. Например:

constants.cpp:

```
namespace Constants
{
    // Фактические глобальные переменные
    extern const double pi(3.14159);
    extern const double avogadro(6.0221413e23);
    extern const double my_gravity(9.2);
}
```

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace Constants
{
    // Только предварительные объявления
    extern const double pi;
    extern const double avogadro;
    extern const double my_gravity;
}

#endif
```

Их использование в коде остается неизменным:

```
#include "constants.h"

//...
double circumference = 2 * radius * Constants::pi;
//...
```

Теперь определение символьных констант выполняется только один раз (в constants.cpp). Любые изменения, сделанные в constants.cpp, потребуют перекомпиляции только (одного) этого файла.

Но есть и обратная сторона медали: такие константы больше не будут считаться константами типа compile-time и, поэтому, не смогут использоваться где-либо, где потребуется константа такого типа.

Поскольку глобальные символьные константы должны находиться в отдельном пространстве имен и быть доступными только для чтения, то использовать префикс `g_` уже не обязательно.

Предостережение о (неконстантных) глобальных переменных

У начинающих программистов часто возникает соблазн использовать просто множество глобальных переменных, поскольку с ними легко работать, особенно когда задействовано много функций. Тем не менее, этого следует избегать! Почему? Об этом мы поговорим на следующем уроке.

Заключение

Подытожим вышесказанное:

- Глобальные переменные имеют глобальную область видимости и могут использоваться в любом месте программы. Подобно функциям, вы должны

использовать предварительные объявления (с ключевым словом extern), чтобы использовать глобальную переменную, определенную в другом файле.

- По умолчанию, глобальные неконстантные переменные имеют внешнюю связь. Вы можете использовать ключевое слово static, чтобы сделать их внутренними.
- По умолчанию, глобальные константные переменные имеют внутреннюю связь. Вы можете использовать ключевое слово extern, чтобы сделать их внешними.
- Используйте префикс `g_` для идентификации ваших неконстантных глобальных переменных.

Тест

В чём разница между областью видимости, продолжительностью жизни и связью переменных? Какие типы продолжительности жизни, области видимости и связи имеют глобальные переменные?

Урок №53. Почему глобальные переменные – зло?

Если вы попросите ветерана-программиста дать один дельный совет о программировании, то после некоторого раздумья он ответит: «Избегайте использования глобальных переменных!». И, частично, он будет прав. Глобальные переменные являются одними из самых злоупотребляемых объектов в языке C++. Хоть они и выглядят безвредными в небольших программах, использование их в крупных проектах зачастую чрезвычайно проблематично.

Новички часто используют огромное количество глобальных переменных, потому что с ними легко работать, особенно когда задействовано много функций. Это плохая идея. Многие разработчики считают, что неконстантные глобальные переменные вообще не следует использовать!

Но прежде, чем мы разберемся с вопросом «Почему?», нужно кое-что уточнить. Когда разработчики говорят, что глобальные переменные - это зло, они НЕ подразумевают полностью ВСЕ глобальные переменные. Они говорят о неконстантных глобальных переменных.

Почему (неконстантные) глобальные переменные — это зло?

Безусловно, причина №1, почему неконстантные глобальные переменные являются опасными, — это то, что их значения могут изменять любые вызываемые функции, при этом вы можете этого и не знать. Например, рассмотрим следующую программу:

```
#include <iostream>

// Объявление глобальной переменной
int g_mode;

void doSomething()
{
    g_mode = 2; // присваиваем глобальной переменной g_mode значение 2
}

int main()
{
    g_mode = 1; // примечание: Здесь мы присваиваем глобальной переменной g_mode значение 1. Это не объявление локальной переменной g_mode!

    doSomething();

    // Программист по-прежнему ожидает, что g_mode будет 1.
    // Но функция doSomething() изменила значение этой переменной на 2!

    if (g_mode == 1)
        std::cout << "No threat detected.\n";
```

```
    else
        std::cout << "Launching nuclear missiles...\n";

    return 0;
}
```

Результат выполнения программы:

```
Launching nuclear missiles...
```

Сначала мы присваиваем переменной `g_mode` значение `1`, а затем вызываем функцию doSomething(). Если бы мы не знали заранее, что doSomething() изменит значение `g_mode`, то, вероятно, не ожидали бы дальнейшего развития событий (`g_mode = 2` => `Launching nuclear missiles...`)!

Неконстантные глобальные переменные делают каждую функцию потенциально опасной, и программист не может заранее знать, какая из используемых им функций является опасной, а какая — нет. Локальные переменные намного безопаснее, потому что другие функции не могут влиять на них напрямую.

Также есть много других веских причин не использовать неконстантные глобальные переменные. Например, нередко можно встретить примерно следующее:

```
void boo()
{
    // Некоторый код

    if (g_mode == 4) // делаем что-нибудь полезное
}
```

Предположим, что `g_mode` равно `3`, а не `4` - наша программа выдаст неверные результаты. Как это исправить? Нужно будет отыскать все места, где предположительно могло измениться значение переменной `g_mode`, а затем проследить ход выполнения кода в каждом потенциально опасном участке. Возможно, изменение глобальной переменной вы обнаружите вообще в другом коде, который, как вам казалось на первый взгляд, никак не связан с фрагментом, приведенным выше.

Одной из причин объявления локальных переменных максимально близко к месту их первого использования является уменьшение количества кода, которое нужно будет просмотреть, чтобы понять, что делает (зачем нужна?) переменная. С глобальными переменными дела обстоят несколько иначе — поскольку их можно использовать в любом месте программы, то вам придется просмотреть чуть ли не весь код, чтобы проследить логику выполнения и изменения значений переменных в вашей программе.

Например, вы можете обнаружить, что на `g_mode` ссылаются 442 раза в вашей программе. Если использования переменной `g_mode` не подкреплены комментариями, то вам придется просмотреть каждое упоминание `g_mode`, чтобы понять, как оно используется в разных случаях.

Также глобальные переменные делают вашу программу менее модульной и гибкой. Функция, которая использует только свои параметры и не имеет побочных эффектов, является идеальной в плане модульности. Модульность помогает понять структуру вашей программы, что она делает и как можно повторно использовать определенные участки кода в другой программе. Глобальные переменные значительно уменьшают эту возможность.

В частности, не используйте глобальные переменные в качестве важных переменных, которые выполняют главные или решающие функции в программе (например, переменные, которые используются в условных стейтментах, как `g_mode` выше). Ваша программа вряд ли сломается, если в ней будет глобальная переменная с информационным значением, которое может меняться (например, имя пользователя). Гораздо хуже, если изменится значение глобальной переменной, которая влияет непосредственно на результаты выполнения самой программы или на её работу.

Правило: Вместо глобальных переменных используйте локальные (когда это целесообразно).

В чём плюсы использования (неконстантных) глобальных переменных?

Их немного. Зачастую можно обойтись без использования неконстантных глобальных переменных. Но в некоторых случаях их разумное использование поможет уменьшить сложность программы, и, иногда, может быть даже лучшим вариантом для решения проблемы, чем альтернативные способы.

Например, если ваша программа использует базу данных для чтения и записи данных, то имеет смысл определить базу данных глобально, поскольку доступ к ней может понадобиться с любого места. Аналогично, если в вашей программе есть журнал ошибок (или журнал отладки), в котором вы можете читать/записывать информацию об ошибках (или об отладке), то имеет смысл определить его глобально. Звуковая библиотека может быть еще одним хорошим примером: вам, вероятно, не захочется открывать доступ к ней для каждой функции, которая делает запрос. Поскольку у вас будет только одна звуковая библиотека, управляющая всеми звуками, логично будет объявить её глобально, инициализировать при запуске программы, а затем использовать только в режиме чтения.

Как защититься от "глобального разрушения"?

Если у вас возникнет ситуация, где полезнее будет использовать неконстантные глобальные переменные, вместо локальных, то вот вам несколько полезных советов, которые помогут свести к минимуму количество потенциальных проблем, с которыми вы можете столкнуться при использовании подобных переменных.

Во-первых, добавляйте префикс `g_` ко всем вашим глобальным переменным и/или размещайте их в пространстве имен, дабы уменьшить вероятность возникновения конфликтов имен.

Например, вместо следующего:

```
1. #include <iostream>
2.
3. double gravity (9.8); // по имени переменной непонятно, глобальная ли это
   переменная или локальная
4.
5. int main()
6. {
7.     return 0;
8. }
```

Сделайте следующее:

```
1. #include <iostream>
2.
3. double g_gravity (9.8); // теперь понятно, что это глобальная переменная
4.
5. int main()
6. {
7.     return 0;
8. }
```

Во-вторых, вместо разрешения прямого доступа к глобальным переменным, лучше их «инкапсулировать». Сначала добавьте ключевое слово static, чтобы доступ к ним был возможен только из файла, в котором они объявлены. Затем напишите внешние глобальные «функции доступа» для работы с переменными. Эти функции помогут обеспечить надлежащее использование переменных (например, при проверке ввода, проверке допустимого диапазона значений и т.д.). Кроме того, если вы когда-либо решите изменить первоначальную реализацию программы (например, перейти из одной базы данных в другую), то вам нужно будет обновить только *функции доступа* вместо каждого фрагмента кода, который напрямую использует глобальные переменные.

Например, вместо следующего:

```
1. double g_gravity (9.8); // можно экспортировать и использовать напрямую в
   любом файле
```

Сделайте следующее:

```
1. static double g_gravity (9.8); // ограничиваем доступ к переменной только на
   этот файл
2.
3. double getGravity() // эта функция может быть экспортирована в другие файлы
   для доступа к глобальной переменной
4. {
5.     return g_gravity;
6. }
```

В-третьих, при написании автономной функции, использующей глобальные переменные, не используйте их непосредственно в теле функции. Передавайте их в качестве параметров. Таким образом, если в вашей функции нужно будет когда-либо использовать другое значение, то вы сможете просто изменить параметр. Это улучшит модульность вашей программы.

Вместо следующего:

```
1. // Эта функция полезна только для расчета мгновенной скорости на основе
   глобальной гравитации
2. double instantVelocity(int time)
3. {
4.     return g_gravity * time;
5. }
```

Сделайте следующее:

```
1. // Эта функция вычисляет мгновенную скорость для любого значения гравитации.
2. // Передайте возвращаемое значение из getGravity() в параметр gravity, если
   хотите использовать глобальную переменную gravity
3. double instantVelocity(int time, double gravity)
4. {
5.     return gravity * time;
6. }
```

Наконец, изменение значений глобальных переменных - это прямой путь к проблемам. Структурируйте ваш код в соответствии с тем, что ваши глобальные переменные могут измениться. Постарайтесь свести к минимуму количество случаев, где они могут изменять свои значения — обращайтесь с ними исключительно как с *доступными только для чтения* (насколько позволяет ситуация). Если вы можете инициализировать значение глобальной переменной при запуске программы, а затем не изменять его в ходе выполнения, то, таким образом, вы снизите вероятность возникновения непредвиденных проблем.

Шутка

Какой наилучший префикс для глобальных переменных?

Ответ: `//`.

Заключение

Избегайте использования неконстантных глобальных переменных, насколько это возможно! Если же используете, то используйте их максимально разумно и осторожно.

Урок №54. Статические переменные

Ключевое слово static является одним из самых запутанных в языке C++. Оно имеет разные значения в разных ситуациях.

На уроке о глобальных переменных мы узнали, что, добавляя static к переменной, объявленной вне блока, мы определяем её как внутреннюю, то есть такую, которую можно использовать только в файле, в котором она определена.

Ключевое слово static можно применять и к переменным внутри блока, но тогда его значение будет другим. На уроке о локальных переменных мы узнали, что локальные переменные имеют автоматическую продолжительность жизни, т.е. создаются, когда блок начинается, и уничтожаются при выходе из него.

Использование **ключевого слова** static с локальными переменными изменяет их свойство продолжительности жизни с автоматического на статическое (или *"фиксированное"*). **Статическая переменная** (или ***«переменная со статической продолжительностью жизни»***) сохраняет свое значение даже после выхода из блока, в котором она определена. То есть она создается (и инициализируется) только один раз, а затем сохраняется на протяжении выполнения всей программы.

Рассмотрим разницу между переменными с автоматической и статической продолжительностями жизни.

Автоматическая продолжительность жизни (по умолчанию):

```
#include <iostream>

void incrementAndPrint()
{
    int value = 1; // автоматическая продолжительность жизни (по умолчанию)
    ++value;
    std::cout << value << std::endl;
} // переменная value уничтожается здесь

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();
}
```

Каждый раз, при вызове функции incrementAndPrint(), создается переменная `value`, которой присваивается значение `1`. Функция incrementAndPrint() увеличивает значение переменной до `2`, а затем выводит его. Когда

incrementAndPrint() завершает свое выполнение, переменная выходит из области видимости и уничтожается.

Следовательно, результат выполнения программы:

```
2
2
2
```

Теперь рассмотрим статическую версию. Единственная разница между этими двумя программами только в добавлении ключевого слова static к переменной.

Статическая продолжительность жизни:

```
#include <iostream>

void incrementAndPrint()
{
    static int s_value = 1; // переменная s_value является статической
    ++s_value;
    std::cout << s_value << std::endl;
} // переменная s_value не уничтожается здесь, но становится недоступной

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();
}
```

Поскольку переменная `s_value` объявлена статической (с помощью ключевого слова static), то она создается и инициализируется только один раз. Кроме того, выходя из области видимости, она не уничтожается. Каждый раз, при вызове функции incrementAndPrint(), значение `s_value` увеличивается.

Результат выполнения программы:

```
2
3
4
```

Так же, как мы используем префикс `g_` с глобальными переменными, префикс `s_` принято использовать со статическими переменными. Обратите внимание, внутренние глобальные переменные (которые объявлены с использованием static) остаются с префиксом `g_`, а не с префиксом `s_`.

Зачем нужны статические локальные переменные? Одним из наиболее распространенных применений является генерация уникальных идентификаторов.

При работе с большим количеством одинаковых объектов внутри программы часто бывает полезно присвоить каждому объекту отдельный уникальный идентификационный номер. Это легко осуществить, используя одну статическую локальную переменную:

```
1. int generateID()
2. {
3.     static int s_itemID = 0;
4.     return s_itemID++;
5. }
```

При первом вызове функции возвращается `0`. Во второй раз возвращается `1`. Затем `2` и каждый последующий вызов будет увеличивать эту переменную на единицу. Хороший способ генерации уникальных идентификаторов для похожих объектов? Хороший! Поскольку `s_itemID` - это локальная переменная, то она не может быть «изменена» другими функциями.

Статические переменные имеют некоторые преимущества глобальных переменных (они не уничтожаются до завершения программы), сохраняя при этом локальную область видимости. Таким образом, они намного безопаснее для использования, нежели глобальные переменные.

Тест

Какой эффект от добавления ключевого слова static к глобальной переменной? Какое влияние оно имеет на локальную переменную?

Урок №55. Связи, область видимости и продолжительность жизни

Мы уже ранее рассматривали, что такое область видимости, продолжительность жизни, связи и то, какими они могут быть в языке C++. Давайте сейчас закрепим это всё.

Область видимости

Область видимости идентификатора определяет, где он доступен для использования. К идентификатору, который находится вне области видимости, доступ закрыт.

- Переменные с **локальной/блочной областью видимости** доступны только в пределах блока, в котором они объявлены. Это:
  - локальные переменные;
  - параметры функции.
- Переменные с **глобальной/файловой областью видимости** доступны в любом месте файла. Это:
  - глобальные переменные.

Продолжительность жизни

Продолжительность жизни переменной определяет, где она создается и где уничтожается.

- Переменные с **автоматической продолжительностью жизни** создаются в точке определения и уничтожаются при выходе из блока, в котором определены. Это:
  - обычные локальные переменные.
- Переменные со **статической продолжительностью жизни** создаются, когда программа запускается, и уничтожаются при её завершении. Это:
  - глобальные переменные;
  - статические локальные переменные.
- Переменные с **динамической продолжительностью жизни** создаются и уничтожаются по запросу программиста. Это:
  - динамические переменные (о них мы поговорим на соответствующем уроке).

Связи

Связь идентификатора определяет, относятся ли несколько упоминаний одного идентификатора к одному и тому же идентификатору или нет.

- Идентификаторы **без связей** — это идентификаторы, которые ссылаются сами на себя. Это:
  - обычные локальные переменные;
  - пользовательские типы данных, такие как enum, typedef и классы, объявленные внутри блока (об этом детально поговорим на соответствующих уроках).
- Идентификаторы с **внутренней связью** доступны в любом месте файла, в котором они объявлены. Это:
  - статические глобальные переменные (инициализированные или неинициализированные);
  - константные глобальные переменные;
  - статические функции (о них поговорим чуть позже).
- Идентификаторы с **внешней связью** доступны как в любом месте файла, в котором они объявлены, так и в других файлах (через предварительное объявление). Это:
  - обычные функции;
  - неконстантные глобальные переменные (инициализированные или неинициализированные);
  - внешние константные глобальные переменные;
  - определяемые пользователем типы данных, такие как enum, typedef и классы с глобальной областью видимости (о них мы поговорим чуть позже).

Идентификаторы с внешней связью могут вызвать ошибку дублирования определений, если определения скомпилированы в более чем одном файле .cpp.

Функции по умолчанию имеют внешнюю связь, что можно изменить с помощью ключевого слова static (на внутреннюю связь).

Внимательные читатели могут заметить, что глобальные типы данных имеют внешнюю связь, но их определения не вызывают ошибки линкера при использовании в нескольких файлах. Это связано с тем, что типы, шаблоны и внешние встроенные функции являются исключениями из правила, и это позволяет им быть определенными более чем в одном файле, при условии, что эти определения идентичны. В противном случае, они не были бы так полезны.

Резюмируем

Весь материал, изложенный выше:

Тип	Пример	Область видимости	Продолжительность жизни	Связь	Примечание
Локальная переменная	int x;	Локальная область видимости	Автоматическая продолжительность жизни	Нет связей	
Статическая локальная переменная	static int s_x;	Локальная область видимости	Статическая продолжительность жизни	Нет связей	
Динамическая переменная	int *x = new int;	Локальная область видимости	Динамическая продолжительность жизни	Нет связей	
Параметр функции	void foo(int x)	Локальная область видимости	Автоматическая продолжительность жизни	Нет связей	
Внешняя неконстантная глобальная переменная	int g_x;	Глобальная область видимости	Статическая продолжительность жизни	Внешняя связь	Инициализированная или неинициализированная

Внутренняя неконстантная глобальная переменная	static int g_x;	Глобальная область видимости	Статическая продолжительность жизни	Внутренняя связь	Инициализированная или неинициализированная
Внутренняя константная глобальная переменная	const int g_x(1);	Глобальная область видимости	Статическая продолжительность жизни	Внутренняя связь	Должна быть инициализирована
Внешняя константная глобальная переменная	extern const int g_x(1);	Глобальная область видимости	Статическая продолжительность жизни	Внешняя связь	Должна быть инициализирована

Предварительные объявления

С помощью предварительного объявления мы можем получить доступ к функции или переменной из другого файла:

Тип	Пример	Примечание
Предварительное объявление функции	void foo(int x);	Только прототип, без тела функции
Предварительное объявление неконстантной глобальной переменной	extern int g_x;	Переменная должна быть инициализирована
Предварительное объявление константной глобальной переменной	extern const int g_x;	Переменная должна быть инициализирована

Урок №56. Пространства имен

Конфликт имен возникает, когда два одинаковых идентификатора находятся в одной области видимости, и компилятор не может понять, какой из этих двух следует использовать в конкретной ситуации. Компилятор или линкер выдаст ошибку, так как у них недостаточно информации, чтобы решить эту неоднозначность. Как только программы увеличиваются в объемах, количество идентификаторов также увеличивается, следовательно, увеличивается и вероятность возникновения конфликтов имен.

Рассмотрим пример такого конфликта. boo.h и doo.h — это заголовочные файлы с функциями, которые выполняют разные вещи, но имеют одинаковые имена и параметры.

boo.h:

```
1. // Функция doOperation() выполняет операцию сложения своих параметров
2. int doOperation(int a, int b)
3. {
4.     return a + b;
5. }
```

doo.h:

```
1. // Функция doOperation() выполняет операцию вычитания своих параметров
2. int doOperation(int a, int b)
3. {
4.     return a - b;
5. }
```

main.cpp:

```
1. #include <iostream>
2. #include "boo.h"
3. #include "doo.h"
4.
5. int main()
6. {
7.     std::cout << doOperation(5, 4); // какая версия doOperation() выполнится
   здесь?
8.     return 0;
9. }
```

Если boo.h и doo.h скомпилировать отдельно, то всё пройдет без инцидентов. Однако, соединив их в одной программе, мы подключим две разные функции, но с одинаковыми именами и параметрами, в одну область видимости (глобальную), а это, в свою очередь, приведет к конфликту имен. В результате, компилятор выдаст

ошибку. Для решения подобных проблем и добавили в язык C++ такую концепцию, как пространства имен.

Что такое пространство имен?

Пространство имен определяет область кода, в которой гарантируется уникальность всех идентификаторов. По умолчанию, глобальные переменные и обычные функции определены в **глобальном пространстве имен.** Например:

```
1. int g_z = 4;
2.
3. int boo(int z)
4. {
5.     return -z;
6. }
```

Глобальная переменная `g_z` и функция boo() определены в глобальном пространстве имен.

В примере, приведенном выше, при подключении файлов boo.h и doo.h обе версии doOperation() были включены в глобальное пространство имен, из-за чего, собственно, и произошел конфликт имен.

Чтобы избежать подобных ситуаций, когда два независимых объекта имеют идентификаторы, которые могут конфликтовать друг с другом при совместном использовании, язык C++ позволяет объявлять собственные пространства имен через **ключевое слово namespace**. Всё, что объявлено внутри пользовательского пространства имен, — принадлежит только этому пространству имен (а не глобальному).

Перепишем заголовочные файлы из вышеприведенного примера, но уже с использованием namespace:

boo.h:

```
1. namespace Boo
2. {
3.     // Эта версия doOperation() принадлежит пространству имен Boo
4.     int doOperation(int a, int b)
5.     {
6.         return a + b;
7.     }
8. }
```

doo.h:

```
1. namespace Doo
2. {
```

```
3.      // Эта версия doOperation() принадлежит пространству имен Doo
4.      int doOperation(int a, int b)
5.      {
6.          return a - b;
7.      }
8.  }
```

Теперь doOperation() из файла boo.h находится в пространстве имен `Boo`, а doOperation() из файла doo.h — в пространстве имен `Doo`. Посмотрим, что произойдет при перекомпиляции main.cpp:

```
1.  int main()
2.  {
3.      std::cout << doOperation(5, 4); // какая версия doOperation() здесь
    выполнится?
4.      return 0;
5.  }
```

Результатом будет еще одна ошибка:

```
C:\VCProjects\Test.cpp(15) : error C2065: 'doOperation' :
undeclared identifier
```

Случилось так, что когда мы попытались вызвать функцию doOperation(), компилятор заглянул в глобальное пространство имен в поисках определения doOperation(). Однако, поскольку ни одна из наших версий doOperation() не находится в глобальном пространстве имен, компилятор не смог найти определение doOperation() вообще!

Существует два разных способа сообщить компилятору, какую версию doOperation() следует использовать: через оператор разрешения области видимости или с помощью using-стейтментов (о них мы поговорим на следующем уроке).

Доступ к пространству имен через оператор разрешения области видимости (::)

Первый способ указать компилятору искать идентификатор в определенном пространстве имен - это использовать название необходимого пространства имен вместе с оператором разрешения области видимости (`::`) и требуемым идентификатором.

Например, сообщим компилятору использовать версию doOperation() из пространства имен Boo:

```
1.  int main(void)
2.  {
3.      std::cout << Boo::doOperation(5, 4);
4.      return 0;
```

```
5. }
```

Результат:

```
9
```

Если бы мы захотели использовать версию doOperation() из пространства имен Doo:

```
1. int main(void)
2. {
3.     std::cout << Doo::doOperation(5, 4);
4.     return 0;
5. }
```

Результат:

```
1
```

Оператор разрешения области видимости хорош, так как позволяет выбрать конкретное пространство имен. Мы даже можем сделать следующее:

```
1. int main(void)
2. {
3.     std::cout << Boo::doOperation(5, 4) << '\n';
4.     std::cout << Doo::doOperation(5, 4) << '\n';
5.     return 0;
6. }
```

Результат:

```
9
1
```

Также этот оператор можно использовать без какого-либо префикса (например, `::doOperation`). В таком случае мы ссылаемся на глобальное пространство имен.

Пространства имен с одинаковыми названиями

Допускается объявление пространств имен в нескольких местах (либо в нескольких файлах, либо в нескольких местах внутри одного файла). Всё, что находится внутри одного блока имен, считается частью только этого блока.

add.h:

```
1. namespace DoMath
2. {
3.     // Функция add() является частью пространства имен DoMath
4.     int add(int x, int y)
5.     {
6.         return x + y;
```

```
    }
}
```

subtract.h:

```
namespace DoMath
{
    // Функция subtract() является частью пространства имен DoMath
    int subtract(int x, int y)
    {
        return x - y;
    }
}
```

main.cpp:

```
#include "add.h" // импортируем DoMath::add()
#include "subtract.h" // импортируем DoMath::subtract()

int main(void)
{
    std::cout << DoMath::add(5, 4) << '\n';
    std::cout << DoMath::subtract(5, 4) << '\n';

    return 0;
}
```

Всё работает, как нужно.

Стандартная библиотека C++ широко использует эту особенность, поскольку все заголовочные файлы, которые находятся в ней, реализуют свой функционал внутри пространства имен std.

Псевдонимы и вложенные пространства имен

Одни пространства имен могут быть вложены в другие пространства имен. Например:

```
#include <iostream>

namespace Boo
{
    namespace Doo
    {
        const int g_x = 7;
    }
}

int main()
{
    std::cout << Boo::Doo::g_x;
    return 0;
}
```

Обратите внимание, поскольку Doo находится внутри Boo, то доступ к `g_x` осуществляется через `Boo::Doo::g_x`.

Так как это не всегда удобно и эффективно, то C++ позволяет создавать псевдонимы для пространств имен:

```
#include <iostream>

namespace Boo
{
    namespace Doo
    {
        const int g_x = 7;
    }
}

namespace Foo = Boo::Doo; // Foo теперь считается как Boo::Doo

int main()
{
    std::cout << Foo::g_x; // это, на самом деле, Boo::Doo::g_x
    return 0;
}
```

Стоит отметить, что пространства имен в C++ не были разработаны, как способ реализации информационной иерархии — они были разработаны в качестве механизма предотвращения возникновения конфликтов имен. Как доказательство этому, вся Стандартная библиотека шаблонов находится в единственном пространстве имен `std::`.

Вложенность пространств имен не рекомендуется использовать, так как при неумелом использовании увеличивается вероятность возникновения ошибок и дополнительно усложняется логика программы.

Урок №57. using-стейтменты

Если вы часто используете Стандартную библиотеку C++, то постоянное добавление `std::` к используемым объектам может быть несколько утомительным, не правда ли? Язык C++ предоставляет альтернативы в виде using-стейтментов.

Использование "using-объявления"

Одной из альтернатив является использование **"using-объявления"**. Вот программа «Hello, world!» с "using-объявлением" в строке №5:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     using std::cout; // «using-объявление» сообщает компилятору, что cout следует обрабатывать, как std::cout
6.     cout << "Hello, world!"; // и никакого префикса std:: уже здесь не нужно!
7.     return 0;
8. }
```

Строка `using std::cout;` сообщает компилятору, что мы будем использовать объект cout из пространства имен std. И каждый раз, когда компилятор будет сталкиваться с `cout`, он будет понимать, что это `std::cout`.

Конечно, в этом случае, мы не сэкономили много усилий, но в программе, где объекты из пространства имен std используются сотни, если не тысячи раз, "using-объявление" неплохо так экономит время, усилия + улучшает читабельность кода. Также для каждого объекта нужно использовать отдельное "using-объявление" (например, отдельное для `std::cout`, отдельное для `std::cin` и отдельное для `std::endl`).

Хотя этот способ является менее предпочтительным, чем использование префикса `std::`, он все же является абсолютно безопасным и приемлемым.

Использование "using-директивы"

Второй альтернативой является использование **"using-директивы"**. Вот программа «Hello, world!» с "using-директивой" в строке №5:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     using namespace std; // «using-директива» сообщает компилятору, что мы используем все объекты из пространства имен std!
```

```
6.     cout << "Hello, world!"; // так что никакого префикса std:: здесь уже не
   нужно!
7.     return 0;
8.  }
```

Много разработчиков спорят насчет использования "using-директивы". Так как с её помощью мы подключаем ВСЕ имена из пространства имен std, то вероятность возникновения конфликтов имен значительно возрастает (но все же эта вероятность в глобальном масштабе остается незначительной). `using namespace std;` сообщает компилятору, что мы хотим использовать всё, что находится в пространстве имен std, так что, если компилятор найдет имя, которое не сможет распознать, он будет проверять его наличие в пространстве имен std.

Совет: Старайтесь избегать использования "using-директивы" (насколько это возможно).

Пример конфликта с "using-директивой"

Рассмотрим пример, где использование "using-директивы" создает неопределенность:

```
1.  #include <iostream>
2.
3.  int cout() // объявляем нашу собственную функцию "cout"
4.  {
5.      return 4;
6.  }
7.
8.  int main()
9.  {
10.     using namespace std; // делаем std::cout доступным по "cout"
11.     cout << "Hello, world!"; // какой cout компилятор здесь должен
   использовать? Тот, который из пространства имен std или тот, который мы
   определили выше?
12.
13.     return 0;
14. }
```

Здесь компилятор не сможет понять, использовать ли ему std::cout или функцию cout(), которую мы определили сами. В результате, получим ошибку неоднозначности. Хоть это и банальный пример, но если бы мы добавили префикс `std::` к cout:

```
1.  std::cout << "Hello, world!"; // сообщаем компилятору, что хотим использовать
   std::cout
```

Или использовали бы "using-объявление" вместо "using-директивы":

```
1.  using std::cout; // сообщаем компилятору, что cout означает std::cout
2.  cout << "Hello, world!"; // так что здесь следует использовать std::cout
```

Тогда наша программа была бы без ошибок.

Большинство программистов избегают использования "using-директивы" именно по этой причине. Другие считают это приемлемым до тех пор, пока "using-директива" используется только в пределах отдельных функций (что значительно сокращает масштабы возникновения конфликтов имен).

Области видимости "using-объявления" и "using-директивы"

Если "using-объявление" или "using-директива" используются в блоке, то они применяются только внутри этого блока (по обычным правилам локальной области видимости). Это хорошо, поскольку уменьшает масштабы возникновения конфликтов имен до отдельных блоков. Однако многие начинающие программисты пишут "using-директиву" в глобальной области видимости (вне функции main() или вообще вне любых функций). Этим они *вытаскивают* все имена из пространства имен std напрямую в глобальную область видимости, значительно увеличивая вероятность возникновения конфликтов имен. А это уже не хорошо.

Правило: Никогда не используйте using-стейтменты вне тела функций.

Отмена/замена using-стейтментов

Как только один using-стейтмент был объявлен, его невозможно отменить или заменить другим using-стейтментом в пределах области видимости, в которой он был объявлен. Например:

```
1. int main()
2. {
3.     using namespace Boo;
4. 
5.     // Отменить «использование пространства имен Boo» здесь невозможно!
6.     // Также нет никакого способа заменить «using namespace Boo» на другой
   using-стейтмент
7. 
8.     return 0;
9. } // действие using namespace Boo заканчивается здесь
```

Лучшее, что вы можете сделать — это намеренно ограничить область применения using-стейтментов с самого начала, используя правила локальной области видимости:

```
1. int main()
2. {
3.     {
4.         using namespace Boo;
5.         // Здесь всё относится к пространству имен Boo::
6.     } // действие using namespace Boo заканчивается здесь
```

```
7.
8.      {
9.          using namespace Foo;
10.         // Здесь всё относится к пространству имен Foo::
11.     } // действие using namespace Foo заканчивается здесь
12.
13.     return 0;
14. }
```

Конечно, всей этой головной боли можно было бы избежать, просто использовав оператор разрешения области видимости (`::`).

Урок №58. Неявное преобразование типов данных

Из предыдущих уроков мы уже знаем, что значение переменной хранится в виде последовательности бит, а тип переменной указывает компилятору, как интерпретировать эти биты в соответствующие значения.

Преобразование типов

Разные типы данных могут представлять одно значение по-разному, например, значение `4` типа int и значение `4.0` типа float хранятся как совершенно разные двоичные шаблоны.

И как вы думаете, что произойдет, если сделать следующее:

```
1. float f = 4; // инициализация переменной типа с плавающей точкой целым числом 4
```

Здесь компилятор не сможет просто скопировать биты из значения `4` типа int и переместить их в переменную `f` типа float. Вместо этого ему нужно будет преобразовать целое число `4` в число типа с плавающей точкой, которое затем можно будет присвоить переменной `f`.

Процесс конвертации значений из одного типа данных в другой называется **преобразованием типов**. Преобразование типов может выполняться в следующих случаях:

Случай №1: **Присваивание или инициализация переменной значением другого типа данных:**

```
1. double k(4); // инициализация переменной типа double целым числом 4
2. k = 7; // присваиваем переменной типа double целое число 7
```

Случай №2: **Передача значения в функцию, где тип параметра — другой:**

```
1. void doSomething(long l)
2. {
3. }
4.
5. doSomething(4); // передача числа 4 (тип int) в функцию с параметром типа long
```

Случай №3: **Возврат из функции, где тип возвращаемого значения — другой:**

```
1. float doSomething()
2. {
3.     return 4.0; // передача значения 4.0 (тип double) из функции, которая
   возвращает float
4. }
```

Случай №4: **Использование бинарного оператора с операндами разных типов:**

```
1. double division = 5.0 / 4; // операция деления со значениями типов double и int
```

Во всех этих случаях (и во многих других) C++ будет использовать преобразование типов.

Есть 2 основных способа преобразования типов:

- **Неявное преобразование типов**, когда компилятор автоматически конвертирует один фундаментальный тип данных в другой.
- **Явное преобразование типов**, когда разработчик использует один из операторов явного преобразования для выполнения конвертации объекта из одного типа данных в другой.

Неявное преобразование типов

Неявное преобразование типов (или ***"автоматическое преобразование типов"***) выполняется всякий раз, когда требуется один фундаментальный тип данных, но предоставляется другой, и пользователь не указывает компилятору, как выполнить конвертацию (не использует явное преобразование типов через операторы явного преобразования).

Есть 2 основных способа неявного преобразования типов:

- числовое расширение;
- числовая конверсия.

Числовое расширение

Когда значение из одного типа данных конвертируется в другой тип данных побольше (по размеру и по диапазону значений), то это называется **числовым расширением**. Например, тип int может быть расширен в тип long, а тип float может быть расширен в тип double:

```
1. long l(65); // расширяем значение типа int (65) в тип long
2. double d(0.11f); // расширяем значение типа float (0.11) в тип double
```

В языке C++ есть два варианта расширений:

- **Интегральное расширение** (или ***"целочисленное расширение"***). Включает в себя преобразование целочисленных типов, меньших, чем int (bool, char,

unsigned char, signed char, unsigned short, signed short) в int (если это возможно) или unsigned int.

- **Расширение типа с плавающей точкой**. Конвертация из типа float в тип double.

Интегральное расширение и расширение типа с плавающей точкой используются для преобразования "меньших по размеру" типов данных в типы int/unsigned int или double (они наиболее эффективны для выполнения разных операций).

> ***Важно:*** Числовые расширения всегда безопасны и не приводят к потере данных.

Числовые конверсии

Когда мы конвертируем значение из более крупного типа данных в аналогичный, но более мелкий тип данных, или конвертация происходит между разными типами данных, то это называется **числовой конверсией**. Например:

```
1. double d = 4; // конвертируем 4 (тип int) в double
2. short s = 3; // конвертируем 3 (тип int) в short
```

В отличие от расширений, которые всегда безопасны, конверсии могут (но не всегда) привести к потере данных. Поэтому в любой программе, где выполняется неявная конверсия, компилятор будет выдавать предупреждение.

Есть много правил по выполнению числовой конверсии, но мы рассмотрим только основные.

Во всех случаях, когда происходит конвертация значения из одного типа данных в другой, который не имеет достаточного диапазона для хранения конвертируемого значения, результаты будут неожиданные. Поэтому делать так не рекомендуется. Например, рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int i = 30000;
6.     char c = i;
7.
8.     std::cout << static_cast<int>(c);
9.
10.    return 0;
11. }
```

В этом примере мы присвоили огромное целочисленное значение типа int переменной типа char (диапазон которого составляет от -128 до 127). Это приведет к переполнению и следующему результату:

```
48
```

Однако, если число подходит по диапазону, конвертация пройдет успешно. Например:

```
#include <iostream>

int main()
{
    int i = 3;
    short s = i; // конвертируем значение типа int в тип short
    std::cout << s << std::endl;

    double d = 0.1234;
    float f = d; // конвертируем значение типа double в тип float
    std::cout << f << std::endl;

    return 0;
}
```

Здесь мы получим ожидаемый результат:

```
3
0.1234
```

В случаях со значениями типа с плавающей точкой могут произойти округления из-за худшей точности в меньших типах. Например:

```
#include <iostream>
#include <iomanip> // для std::setprecision()

int main()
{
    float f = 0.123456789; // значение типа double - 0.123456789 имеет 9 значащих цифр, но float может хранить только 7
    std::cout << std::setprecision(9) << f; // std::setprecision определен в заголовочном файле iomanip

    return 0;
}
```

В этом случае мы наблюдаем потерю в точности, так как точность типа float меньше, чем типа double:

```
0.123456791
```

Конвертация из типа int в тип float успешна до тех пор, пока значения подходят по диапазону.

Например:

```
#include <iostream>

int main()
{
    int i = 10;
    float f = i;
    std::cout << f;

    return 0;
}
```

Результат:

```
10
```

Аналогично, конвертация из float в int успешна до тех пор, пока значения подходят по диапазону. Но следует помнить, что любая дробь отбрасывается. Например:

```
#include <iostream>

int main()
{
    int i = 4.6;
    std::cout << i;

    return 0;
}
```

Дробная часть значения (`.6`) игнорируется и выводится результат:

```
4
```

Обработка арифметических выражений

При обработке выражений компилятор разбивает каждое выражение на отдельные подвыражения. Арифметические операторы требуют, чтобы их операнды были одного типа данных. Чтобы это гарантировать, **компилятор использует следующие правила**:

- Если операндом является целое число меньше (по размеру/диапазону) типа int, то оно подвергается интегральному расширению в int или в unsigned int.
- Если операнды разных типов данных, то компилятор вычисляет операнд с наивысшим приоритетом и неявно конвертирует тип другого операнда в такой же тип, как у первого.

Приоритет типов операндов:

- long double (самый высокий);
- double;
- float;
- unsigned long long;
- long long;
- unsigned long;
- long;
- unsigned int;
- int (самый низкий).

Мы можем использовать **оператор typeid** (который находится в заголовочном файле typeinfo), чтобы узнать решающий тип в выражении.

В следующем примере у нас есть две переменные типа short:

```
1. #include <iostream>
2. #include <typeinfo> // для typeid
3.
4. int main()
5. {
6.     short x(3);
7.     short y(6);
8.     std::cout << typeid(x + y).name() << " " << x + y << std::endl; //
   вычисляем решающий тип данных в выражении x + y
9.
10.    return 0;
11. }
```

Поскольку значениями переменных типа short являются целые числа и тип short меньше (по размеру/диапазону) типа int, то он подвергается интегральному расширению в тип int. Результатом сложения двух int-ов будет тип int:

```
int 9
```

Рассмотрим другой случай:

```
1. #include <iostream>
2. #include <typeinfo> // для typeid()
3.
4. int main()
5. {
6.     double a(3.0);
7.     short b(2);
8.     std::cout << typeid(a + b).name() << " " << a + b << std::endl; //
   вычисляем решающий тип данных в выражении a + b
9.
10.    return 0;
11. }
```

Здесь short подвергается интегральному расширению в int. Однако int и double по-прежнему не совпадают. Поскольку double находится выше в иерархии типов, то целое число `2` преобразовывается в `2.0` (тип double), и сложение двух чисел типа double дадут число типа double:

```
double 5
```

С этой иерархией иногда могут возникать интересные ситуации, например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << 5u - 10; // 5u означает значение 5 типа unsigned int
6.
7.     return 0;
8. }
```

Ожидается, что результатом выражения `5u - 10` будет `-5`, поскольку `5 - 10 = -5`, но результат:

```
4294967291
```

Здесь значение signed int (`10`) подвергается расширению в unsigned int (которое имеет более высокий приоритет), и выражение вычисляется как unsigned int. А поскольку unsigned — это только положительные числа, то происходит переполнение, и мы имеем то, что имеем.

Это одна из тех многих веских причин избегать использования типа unsigned int вообще.

Урок №59. Явное преобразование типов данных

Из предыдущего урока мы уже знаем, что компилятор в определенных случаях выполняет неявное преобразование типов данных.

Когда вы хотите изменить один тип данных на другой, более крупный (по размеру/диапазону), то неявное преобразование является хорошим вариантом.

Но многие начинающие программисты часто пытаются сделать что-то вроде следующего: `float x = 11 / 3;`. Однако, поскольку `11` и `3` являются целыми числами, никакого числового расширения не происходит. Выполняется целочисленное деление `11 / 3`, результатом которого будет значение `3`, которое затем неявно преобразуется в `3.0` и присвоится переменной `x`!

В случае, когда вы используете литералы (такие как `11` или `3`), замена одного или обоих целочисленных литералов значением типа с плавающей точкой (`11.0` или `3.0`) приведет к конвертации обоих операндов в значения типа с плавающей точкой и выполнится деление типа с плавающей точкой.

Но что будет, если использовать переменные? Например:

```
1. int i1 = 11;
2. int i2 = 3;
3. float x = i1 / i2;
```

Значением переменной `x` будет `3`. Как сообщить компилятору, что мы хотим использовать деление типа с плавающей точкой вместо целочисленного деления? Правильно! Использовать один из операторов явного преобразования типов данных, чтобы указать компилятору выполнить явное преобразование.

Операторы явного преобразования типов данных

В языке C++ есть 5 видов **операций явного преобразования типов**:

- **конвертация C-style**;
- **применение оператора static_cast**;
- **применение оператора const_cast**;
- **применение оператора dynamic_cast**;
- **применение оператора reinterpret_cast**.

На этом уроке мы рассмотрим конвертацию C-style и оператор static_cast. Оператор dynamic_cast мы будем рассматривать, когда дойдем до указателей и

наследования. Применения операторов const_cast и reinterpret_cast следует избегать, так как они полезны только в редких случаях и могут создать немало проблем, если их использовать неправильно.

Правило: Избегайте использования const_cast и reinterpret_cast, если у вас нет на это веских причин.

Конвертация C-style

В программировании на языке Си явное преобразование типов данных выполняется с помощью оператора `()`. Внутри круглых скобок мы пишем тип, в который нужно конвертировать. Этот способ конвертации типов называется **конвертацией C-style**. Например:

```
1. int i1 = 11;
2. int i2 = 3;
3. float x = (float)i1 / i2;
```

В программе, приведенной выше, мы используем круглые скобки, чтобы сообщить компилятору о необходимости преобразования переменной `i1` (типа int) в тип float. Поскольку переменная `i1` станет типа float, то `i2` также затем автоматически преобразуется в тип float, и выполнится деление типа с плавающей точкой!

Язык C++ также позволяет использовать этот оператор следующим образом:

```
1. int i1 = 11;
2. int i2 = 3;
3. float x = float(i1) / i2;
```

Конвертация C-style не проверяется компилятором во время компиляции, поэтому она может быть неправильно использована, например, при конвертации типов const или изменении типов данных, без учета их диапазонов (что приведет к переполнению).

Следовательно, конвертацию C-style лучше не использовать.

Правило: Не используйте конвертацию C-style.

Оператор static_cast

В языке C++ есть еще один оператор явного преобразования типов данных — **оператор static_cast**.

Ранее, на уроке о символьном типе данных char, мы уже использовали оператор static_cast для конвертации переменной типа char в тип int, выводя вместо символа целое число:

```
1. char c = 97;
2. std::cout << static_cast<int>(c) << std::endl; // в результате выведется 97, а
   не 'a'
```

Оператор static_cast лучше всего использовать для конвертации одного фундаментального типа данных в другой:

```
1. int i1 = 11;
2. int i2 = 3;
3. float x = static_cast<float>(i1) / i2;
```

Основным преимуществом оператора static_cast является проверка его выполнения компилятором во время компиляции, что усложняет возможность возникновения непреднамеренных проблем.

Использование операторов явного преобразования в неявном преобразовании

Если вы будете выполнять небезопасные неявные преобразования типов данных, то компилятор будет жаловаться. Например:

```
1. int i = 49;
2. char ch = i; // неявное преобразование
```

Конвертация переменной типа int (4 байта) в тип char (1 байт) потенциально опасна - компилятор выдаст предупреждение. Чтобы сообщить ему, что вы намеренно делаете что-то, что потенциально опасно (но хотите сделать это в любом случае), используйте оператор static_cast:

```
1. int i = 49;
2. char ch = static_cast<char>(i);
```

В следующем случае компилятор будет жаловаться, что конвертация из типа double в тип int может привести к потере данных:

```
1. int i = 90;
2. i = i / 3.6;
```

Чтобы сообщить компилятору, что мы сознательно хотим сделать это:

```
1. int i = 90;
2. i = static_cast<int>(i / 3.6);
```

Заключение

Преобразования типов данных следует избегать, если это вообще возможно, поскольку всякий раз, когда выполняется подобное изменение, есть вероятность возникновения непредвиденных проблем. Но очень часто случаются ситуации, когда этого не избежать. Поэтому в таких случаях лучше использовать оператор static_cast вместо конвертации C-style.

Тест

В чём разница между явным и неявным преобразованием типов данных?

Урок №60. Введение в std::string

Вашей первой программой на языке C++, вероятно, была всеми известная программа "Hello, world!":

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Hello, world!" << std::endl;
6.     return 0;
7. }
```

Не так ли? Но что такое `Hello, world!`? `Hello, world!` - это последовательность символов или просто **строка** (англ. ***"string"***). В языке C++ мы используем строки для представления текста (имен, адресов, слов и предложений). Строковые литералы (такие как `Hello, world!`) помещаются в двойные кавычки.

Поскольку их часто используют в программах, то большинство современных языков программирования имеют встроенный тип данных string. В языке C++ есть также этот тип, но не как часть основного языка, а как часть Стандартной библиотеки C++.

Тип данных string

Чтобы иметь возможность использовать строки в C++, сначала нужно подключить заголовочный файл string. Как только это будет сделано, мы сможем определять переменные типа string:

```
1. #include <string>
2.
3. // ...
4. std::string name;
5. // ...
```

Как и с обычными переменными, мы можем инициализировать переменные типа string или присваивать им значения:

```
1. std::string name("Sasha"); // инициализируем переменную name строковым
   литералом "Sasha"
2. name = "Masha"; // присваиваем переменной name строковый литерал "Masha"
```

Строки также могут содержать числа:

```
1. std::string myID("34"); // "34" здесь - это не целое число 34!
```

Стоит отметить, что присваиваемые числа тип string обрабатывает как текст, а не как числа, и, следовательно, ими нельзя манипулировать как обычными числами

(например, вы не сможете выполнять с ними арифметические операции). Язык C++ автоматически не преобразовывает их в значения целочисленных типов или типов с плавающей точкой.

Ввод/вывод строк

Строки можно выводить с помощью std::cout:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string name("Sasha");
7.     std::cout << "My name is " << name;
8.
9.     return 0;
10. }
```

Результат выполнения программы:

```
My name is Sasha
```

А вот с std::cin дела обстоят несколько иначе. Рассмотрим следующий пример:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::cout << "Enter your full name: ";
7.     std::string myName;
8.     std::cin >> myName; // это будет работать не так, как ожидается, поскольку
   извлечение данных из потока std::cin останавливается на первом пробеле
9.
10.    std::cout << "Enter your age: ";
11.    std::string myAge;
12.    std::cin >> myAge;
13.
14.    std::cout << "Your name is " << myName << " and your age is " << myAge;
15. }
```

Результат выполнения программы:

```
Enter your full name: Sasha Mak
Enter your age: Your name is Sasha and your age is Mak
```

Хм, что-то не так! Что же случилось? Оказывается, оператор извлечения (>>) возвращает символы из входного потока данных только до первого пробела. Все остальные символы остаются внутри cin, ожидая следующего извлечения.

Поэтому, когда мы использовали оператор `>>` для извлечения данных в переменную `myName`, только `Sasha` был извлечен, `Mak` остался внутри std::cin, ожидая следующего извлечения. Когда мы использовали оператор `>>` снова, чтобы извлечь данные в переменную `myAge`, мы получили `Mak` вместо `25`. Если бы мы сделали третье извлечение, то получили бы `25`.

Использование std::getline()

Чтобы извлечь полную строку из входного потока данных (вместе с пробелами), используйте **функцию std::getline()**. Она принимает два параметра: первый — std::cin, второй — переменная типа string.

Вот программа, приведенная выше, но уже с использованием std::getline():

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::cout << "Enter your full name: ";
7.     std::string myName;
8.     std::getline(std::cin, myName); // полностью извлекаем строку в переменную
   myName
9.
10.    std::cout << "Enter your age: ";
11.    std::string myAge;
12.    std::getline(std::cin, myAge); // полностью извлекаем строку в переменную
   myAge
13.
14.    std::cout << "Your name is " << myName << " and your age is " << myAge;
15.}
```

Теперь работает как надо:

```
Enter your full name: Sasha Mak
Enter your age: 25
Your name is Sasha Mak and your age is 25
```

Использование std::getline() с std::cin

Извлечение данных из std::cin с помощью std::getline() иногда может приводить к неожиданным результатам. Например, рассмотрим следующую программу:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::cout << "Pick 1 or 2: ";
7.     int choice;
8.     std::cin >> choice;
```

```
9.
10.     std::cout << "Now enter your name: ";
11.     std::string myName;
12.     std::getline(std::cin, myName);
13.
14.     std::cout << "Hello, " << myName << ", you picked " << choice << '\n';
15.
16.     return 0;
17. }
```

Возможно, вы удивитесь, но когда вы запустите эту программу, и она попросит вас ввести ваше имя, она не будет ожидать вашего ввода, а сразу выведет результат (просто пробел вместо вашего имени)!

Пробный запуск программы:

```
Pick 1 or 2: 2
Now enter your name: Hello, , you picked 2
```

Почему так? Оказывается, когда вы вводите числовое значение, поток cin захватывает вместе с вашим числом и символ новой строки. Поэтому, когда мы ввели `2`, cin фактически получил `2\n`. Затем он извлек значение `2` в переменную, оставляя `\n` (символ новой строки) во входном потоке. Затем, когда std::getline() извлекает данные для `myName`, он видит в потоке `\n` и думает, что мы, должно быть, ввели просто пустую строку! А это определенно не то, что мы хотим.

Хорошей практикой является удалять из входного потока данных символ новой строки. Это можно сделать следующим образом:

```
1. std::cin.ignore(32767, '\n'); // игнорируем символы перевода строки "\n" во
   входящем потоке длиной 32767 символов
```

Если мы вставим эту строку непосредственно после получения входных данных, то символ новой строки будет удален из входного потока, и программа будет работать должным образом:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::cout << "Pick 1 or 2: ";
7.     int choice;
8.     std::cin >> choice;
9.
10.    std::cin.ignore(32767, '\n'); // удаляем символ новой строки из входного
   потока данных
11.
12.    std::cout << "Now enter your name: ";
13.    std::string myName;
14.    std::getline(std::cin, myName);
```

```
15.
16.     std::cout << "Hello, " << myName << ", you picked " << choice << '\n';
17.
18.     return 0;
19. }
```

Правило: При вводе числовых значений не забывайте удалять символ новой строки из входного потока данных с помощью std::cin.ignore().

Добавление строк

Вы можете использовать оператор `+` для объединения двух строк или оператор `+=` для добавления одной строки к другой.

В следующей программе мы протестируем эти два оператора, а также покажем, что произойдет, если вы попытаетесь использовать оператор `+` для соединения двух числовых строк:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string x("44");
7.     std::string y("12");
8.
9.     std::cout << x + y << "\n"; // объединяем строки x и y (а не складываем
   числа)
10.    x += " cats";
11.    std::cout << x;
12.
13.    return 0;
14. }
```

Результат выполнения программы:

```
4412
44 cats
```

Обратите внимание, оператор `+` объединил две числовые строки в одну (44 + 12 = 4412). Он не складывал эти строки как числа.

Длина строк

Чтобы узнать длину строки, мы можем сделать следующее:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5.
```

```
{
    std::string myName("Sasha");
    std::cout << myName << " has " << myName.length() << " characters\n";
    return 0;
}
```

Результат выполнения программы:

```
Sasha has 5 characters
```

Обратите внимание, вместо запроса длины строки как `length(myName)`, мы пишем `myName.length()`.

Функция запроса длины строки не является обычной функцией как те, которые мы использовали ранее. Это особый тип функции класса std::string, который называется методом. Мы поговорим об этом детально, когда будем рассматривать классы.

Тест

Напишите программу, которая просит у пользователя ввести его имя, фамилию и возраст. В результате, укажите пользователю, сколько лет он прожил на каждую букву из его имени и фамилии (чтобы было проще, учитывайте также пробелы). Например:

```
Enter your full name: Tom Cats
Enter your age: 45
You've lived 5.625 years for each letter in your name.
```

> Уточнение*:* Возраст 45 делится на длину имени и фамилии "Tom Cats" (8 букв, учитывая пробел), что равно 5.625.

Урок №61. Перечисления

Язык C++ позволяет программистам создавать свои собственные (пользовательские) типы данных.

Перечисляемые типы

Перечисление (или ***"перечисляемый тип"***) - это тип данных, где любое значение (или ***"перечислитель"***) определяется как символьная константа. Объявить перечисление можно с помощью **ключевого слова enum**. Например:

```
1. // Объявляем новое перечисление Colors
2. enum Colors
3. {
4.     // Ниже находятся перечислители - все возможные значения этого типа данных
5.     // Каждый перечислитель отделяется запятой (НЕ точкой с запятой)
6.     COLOR_RED,
7.     COLOR_BROWN,
8.     COLOR_GRAY,
9.     COLOR_WHITE,
10.    COLOR_PINK,
11.    COLOR_ORANGE,
12.    COLOR_BLUE,
13.    COLOR_PURPLE, // о конечной запятой читайте ниже
14.}; // однако сам enum должен заканчиваться точкой с запятой
15.
16.// Определяем несколько переменных перечисляемого типа Colors
17.Colors paint = COLOR_RED;
18.Colors house(COLOR_GRAY);
```

Объявление перечислений не требует выделения памяти. Только когда переменная перечисляемого типа определена (например, как переменная `paint` в примере, приведенном выше), только тогда выделяется память для этой переменной.

Обратите внимание, каждый перечислитель отделяется запятой, а само перечисление заканчивается точкой с запятой.

До C++11, конечная запятая после последнего перечислителя (как после `COLOR_PURPLE` в примере, приведенном выше) не разрешается (хотя многие компиляторы её все равно принимают). Однако начиная с C++11 конечная запятая разрешена.

Имена перечислений

Идентификаторы перечислений часто начинаются с заглавной буквы, а имена перечислителей вообще состоят только из заглавных букв. Поскольку

перечислители вместе с перечислением находятся в едином пространстве имен, то имена перечислителей не могут повторяться в разных перечислениях:

```
enum Colors
{
    YELLOW,
    BLACK, // BLACK находится в глобальном пространстве имен
    PINK
};

enum Feelings
{
    SAD,
    ANGRY,
    BLACK // получим ошибку, так как BLACK уже используется в enum Colors
};
```

Распространено добавление названия перечисления в качестве префикса к перечислителям, например: `ANIMAL_` или `COLOR_`, как для предотвращения конфликтов имен, так и в целях комментирования кода.

Значения перечислителей

Каждому перечислителю автоматически присваивается целочисленное значение в зависимости от его позиции в списке перечисления. По умолчанию, первому перечислителю присваивается целое число `0`, а каждому следующему — на единицу больше, чем предыдущему:

```
#include <iostream>

enum Colors
{
    COLOR_YELLOW, // присваивается 0
    COLOR_WHITE, // присваивается 1
    COLOR_ORANGE, // присваивается 2
    COLOR_GREEN, // присваивается 3
    COLOR_RED, // присваивается 4
    COLOR_GRAY, // присваивается 5
    COLOR_PURPLE, // присваивается 6
    COLOR_BROWN // присваивается 7
};

int main()
{
    Colors paint(COLOR_RED);
    std::cout << paint;

    return 0;
}
```

Результат выполнения программы:

```
4
```

Можно и самому определять значения перечислителей. Они могут быть как положительными, так и отрицательными, или вообще иметь аналогичные другим перечислителям значения. Любые, не определенные вами перечислители, будут иметь значения на единицу больше, чем значения предыдущих перечислителей. Например:

```
1. // Определяем новый перечисляемый тип Animals
2. enum Animals
3. {
4.     ANIMAL_PIG = -4,
5.     ANIMAL_LION, // присваивается -3
6.     ANIMAL_CAT, // присваивается -2
7.     ANIMAL_HORSE = 6,
8.     ANIMAL_ZEBRA = 6, // имеет то же значение, что и ANIMAL_HORSE
9.     ANIMAL_COW // присваивается 7
10. };
```

Обратите внимание, `ANIMAL_HORSE` и `ANIMAL_ZEBRA` имеют одинаковые значения. Хотя C++ это не запрещает, присваивать одно значение нескольким перечислителям в одном перечислении не рекомендуется.

Совет: Не присваивайте свои значения перечислителям.

Правило: Не присваивайте одинаковые значения двум перечислителям в одном перечислении, если на это нет веской причины.

Обработка перечислений

Поскольку значениями перечислителей являются целые числа, то их можно присваивать целочисленным переменным, а также выводить в консоль (как переменные типа int):

```
1. #include <iostream>
2.
3. // Определяем новый перечисляемый тип Animals
4. enum Animals
5. {
6.     ANIMAL_PIG = -4,
7.     ANIMAL_LION, // присваивается -3
8.     ANIMAL_CAT, // присваивается -2
9.     ANIMAL_HORSE = 6,
10.    ANIMAL_ZEBRA = 6, // имеет то же значение, что и ANIMAL_HORSE
11.    ANIMAL_COW // присваивается 7
12. };
13.
14. int main()
15. {
16.    int mypet = ANIMAL_PIG;
17.    std::cout << ANIMAL_HORSE; // конвертируется в int, а затем выводится на
    экран
18.
19.    return 0;
```

```
20. }
```

Результат выполнения программы:

```
6
```

Компилятор не будет неявно конвертировать целочисленное значение в значение перечислителя. Следующее вызовет ошибку компиляции:

```
1. Animals animal = 7; // приведет к ошибке компиляции
```

Тем не менее, вы можете сделать подобное с помощью оператора static_cast:

```
1. Colors color = static_cast<Colors>(5); // но так делать не рекомендуется
```

Компилятор также не позволит вам вводить перечислители через std::cin:

```
1. #include <iostream>
2.
3. enum Colors
4. {
5.     COLOR_PURPLE, // присваивается 0
6.     COLOR_GRAY, // присваивается 1
7.     COLOR_BLUE, // присваивается 2
8.     COLOR_GREEN, // присваивается 3
9.     COLOR_BROWN, // присваивается 4
10.    COLOR_PINK, // присваивается 5
11.    COLOR_YELLOW, // присваивается 6
12.    COLOR_MAGENTA // присваивается 7
13. };
14.
15. int main()
16. {
17.     Colors color;
18.     std::cin >> color; // приведет к ошибке компиляции
19.
20.     return 0;
21. }
```

Однако, вы можете ввести целое число, а затем использовать оператор static_cast, чтобы поместить целочисленное значение в перечисляемый тип:

```
1. int inputColor;
2. std::cin >> inputColor;
3.
4. Colors color = static_cast<Colors>(inputColor);
```

Каждый перечисляемый тип считается отдельным типом. Следовательно, попытка присвоить перечислитель из одного перечисления перечислителю из другого — вызовет ошибку компиляции:

```
1. Animals animal = COLOR_BLUE; // приведет к ошибке компиляции
```

Как и в случае с константами, перечисления отображаются в отладчике, что делает их еще более полезными.

Вывод перечислителей

Попытка вывести перечисляемое значение с помощью std::cout приведет к выводу целочисленного значения самого перечислителя (т.е. его порядкового номера). Но как вывести значение перечислителя в виде текста? Один из способов - написать функцию с использованием стейтментов if:

```
enum Colors
{
    COLOR_PURPLE, // присваивается 0
    COLOR_GRAY, // присваивается 1
    COLOR_BLUE, // присваивается 2
    COLOR_GREEN, // присваивается 3
    COLOR_BROWN, // присваивается 4
    COLOR_PINK, // присваивается 5
    COLOR_YELLOW, // присваивается 6
    COLOR_MAGENTA // присваивается 7
};

void printColor(Colors color)
{
    if (color == COLOR_PURPLE)
        std::cout << "Purple";
    else if (color == COLOR_GRAY)
        std::cout << "Gray";
    else if (color == COLOR_BLUE)
        std::cout << "Blue";
    else if (color == COLOR_GREEN)
        std::cout << "Green";
    else if (color == COLOR_BROWN)
        std::cout << "Brown";
    else if (color == COLOR_PINK)
        std::cout << "Pink";
    else if (color == COLOR_YELLOW)
        std::cout << "Yellow";
    else if (color == COLOR_MAGENTA)
        std::cout << "Magenta";
    else
        std::cout << "Who knows!";
}
```

Выделение памяти для перечислений

Перечисляемые типы считаются частью семейства целочисленных типов, и компилятор сам определяет, сколько памяти выделять для переменных типа enum. По стандарту C++ размер перечисления должен быть достаточно большим, чтобы иметь возможность вместить все перечислители. Но чаще всего размеры переменных enum будут такими же, как и размеры обычных переменных типа int.

Поскольку компилятору нужно знать, сколько памяти выделять для перечисления, то использовать предварительное объявление с ним вы не сможете. Однако существует простой обходной путь. Поскольку определение перечисления само по себе не требует выделения памяти и, если перечисление необходимо использовать в нескольких файлах, его можно определить в заголовочном файле и подключать этот файл везде, где необходимо использовать перечисление.

Польза от перечислений

Перечисляемые типы невероятно полезны для документации кода и улучшения читабельности.

Например, функции часто возвращают целые числа обратно в caller в качестве кодов ошибок, если что-то пошло не так. Как правило, небольшие отрицательные числа используются для представления возможных кодов ошибок. Например:

```
1. int readFileContents()
2. {
3.     if (!openFile())
4.         return -1;
5.     if (!parseFile())
6.         return -2;
7.     if (!readFile())
8.         return -3;
9.
10.    return 0; // если всё прошло успешно
11.}
```

Однако магические числа, как в вышеприведенном примере, не очень эффективное решение. Альтернатива - использовать перечисления:

```
1. enum ParseResult
2. {
3.     SUCCESS = 0,
4.     ERROR_OPENING_FILE = -1,
5.     ERROR_PARSING_FILE = -2,
6.     ERROR_READING_FILE = -3
7. };
8.
9. ParseResult readFileContents()
10.{
11.    if (!openFile())
12.        return ERROR_OPENING_FILE;
13.    if (!parseFile())
14.        return ERROR_PARSING_FILE;
15.    if (!readfile())
16.        return ERROR_READING_FILE;
17.
18.    return SUCCESS; // если всё прошло успешно
19.}
```

Это и читать легче, и понять проще. Кроме того, функция, которая вызывает другую функцию, может проверить возвращаемое значение на соответствующий перечислитель. Это лучше, нежели самому сравнивать возвращаемый результат с конкретными целочисленными значениями, чтобы понять какая именно ошибка произошла, не так ли? Например:

```
if (readFileContents() == SUCCESS)
    {
    // Делаем что-нибудь
    }
else
    {
    // Выводим сообщение об ошибке
    }
```

Перечисляемые типы лучше всего использовать при определении набора связанных идентификаторов. Например, предположим, что вы пишете игру, в которой игрок может иметь один предмет, но этот предмет может быть нескольких разных типов:

```
#include <iostream>
#include <string>

enum ItemType
{
    ITEMTYPE_GUN,
    ITEMTYPE_ARBALET,
    ITEMTYPE_SWORD
};

std::string getItemName(ItemType itemType)
{
    if (itemType == ITEMTYPE_GUN)
        return std::string("Gun");
    if (itemType == ITEMTYPE_ARBALET)
        return std::string("Arbalet");
    if (itemType == ITEMTYPE_SWORD)
        return std::string("Sword");
}

int main()
{
    // ItemType - это перечисляемый тип, который мы объявили выше.
    // itemType (с маленькой i) - это имя переменной, которую мы определяем ниже (типа ItemType).
    // ITEMTYPE_GUN - это значение перечислителя, которое мы присваиваем переменной itemType
    ItemType itemType(ITEMTYPE_GUN);

    std::cout << "You are carrying a " << getItemName(itemType) << "\n";

    return 0;
}
```

Или, если вы пишете функцию для сортировки группы значений:

```
enum SortType
```

```
{
    SORTTYPE_FORWARD,
    SORTTYPE_BACKWARDS
};

void sortData(SortType type)
{
    if (type == SORTTYPE_FORWARD)
        // Сортировка данных в одном порядке
    else if (type == SORTTYPE_BACKWARDS)
        // Сортировка данных в обратном порядке
}
```

Многие языки программирования используют перечисления для определения логических значений. По сути, логический тип данных - это простое перечисление всего лишь с двумя перечислителями: true и false! Однако в языке C++ значения true и false определены как ключевые слова вместо перечислителей.

Тест

Задание №1

Напишите перечисление со следующими перечислителями: `ogre`, `goblin`, `skeleton`, `orc` и `troll`.

Задание №2

Объявите переменную перечисляемого типа, который вы определили в задании №1, и присвойте ей значение `ogre`.

Задание №3

Правда или ложь:

Перечислителям можно:

- присваивать целочисленные значения;
- не присваивать значения;
- явно присваивать значения типа с плавающей точкой;
- присваивать значения предыдущих перечислителей (например, `COLOR_BLUE = COLOR_GRAY`).

Перечислители могут быть:

- отрицательными;
- не уникальными.

Урок №62. Классы enum

Хотя перечисления и считаются отдельными типами данных в языке C++, они не столь безопасны, как кажутся на первый взгляд, и в некоторых случаях позволят вам делать вещи, которые не имеют смысла. Например:

```
#include <iostream>

int main()
{
    enum Fruits
    {
        LEMON, // LEMON находится внутри той же области видимости, что и Fruits
        KIWI
    };

    enum Colors
    {
        PINK, // PINK находится внутри той же области видимости, что и Colors
        GRAY
    };

    Fruits fruit = LEMON; // Fruits и LEMON доступны в одной области видимости (добавлять префикс не нужно)
    Colors color = PINK; // Colors и PINK доступны в одной области видимости (добавлять префикс не нужно)

    if (fruit == color) // компилятор будет сравнивать эти переменные как целые числа
        std::cout << "fruit and color are equal\n"; // и обнаружит, что они равны!
    else
        std::cout << "fruit and color are not equal\n";

    return 0;
}
```

Когда C++ будет сравнивать переменные `fruit` и `color`, он неявно преобразует их в целочисленные значения и сравнит эти целые числа. Так как значениями этих двух переменных являются перечислители, которым присвоено значение `0`, то это означает, что в примере, приведенном выше, `fruit = color`. А это не совсем то, что должно быть, так как `fruit` и `color` из разных перечислений и их вообще нельзя сравнивать (фрукт и цвет!). С обычными перечислителями нет способа предотвратить подобные сравнения.

Для решения этой проблемы в C++11 добавили **классы enum** (или ***"перечисления с областью видимости"***), которые добавляют перечислениям, как вы уже могли понять, локальную область видимости со всеми её правилами. Для создания такого класса нужно просто добавить **ключевое слово class** сразу после enum.

Например:

```
#include <iostream>

int main()
{
    enum class Fruits // добавление "class" к enum определяет перечисление с ограниченной областью видимости, вместо стандартного "глобального" перечисления
    {
        LEMON, // LEMON находится внутри той же области видимости, что и Fruits
        KIWI
    };

    enum class Colors
    {
        PINK, // PINK находится внутри той же области видимости, что и Colors
        GRAY
    };

    Fruits fruit = Fruits::LEMON; // примечание: LEMON напрямую не доступен, мы должны использовать Fruits::LEMON
    Colors color = Colors::PINK; // примечание: PINK напрямую не доступен, мы должны использовать Colors::PINK

    if (fruit == color) // ошибка компиляции, поскольку компилятор не знает, как сравнивать разные типы: Fruits и Colors
        std::cout << "fruit and color are equal\n";
    else
        std::cout << "fruit and color are not equal\n";

    return 0;
}
```

Стандартные перечислители находятся в той же области видимости, что и само перечисление (в глобальной области видимости), поэтому вы можете напрямую получить к ним доступ (например, `PINK`). Однако с добавлением класса, который ограничивает область видимости каждого перечислителя областью видимости его перечисления, для доступа к нему потребуется оператор разрешения области видимости (например, `Colors::PINK`). Это значительно снижает риск возникновения конфликтов имен.

Поскольку перечислители являются частью класса enum, то необходимость добавлять префиксы к идентификаторам отпадает (например, можно использовать просто `PINK` вместо `COLOR_PINK`, так как `Colors::COLOR_PINK` уже будет лишним).

А строгие правила типов классов enum означают, что каждый класс enum считается уникальным типом. Это означает, что компилятор не сможет сравнивать перечислители из разных перечислений. Если вы попытаетесь это сделать, компилятор выдаст ошибку (как в примере, приведенном выше).

Однако учтите, что вы можете сравнивать перечислители внутри одного класса enum (так как эти перечислители принадлежат одному типу):

```
#include <iostream>

int main()
{
    enum class Colors
    {
        PINK,
        GRAY
    };

    Colors color = Colors::PINK;

    if (color == Colors::PINK) // это нормально
        std::cout << "The color is pink!\n";
    else if (color == Colors::GRAY)
        std::cout << "The color is gray!\n";

    return 0;
}
```

С классами enum компилятор больше не сможет неявно конвертировать значения перечислителей в целые числа. Это хорошо! Но иногда могут быть ситуации, когда нужно будет вернуть эту особенность. В таких случаях вы можете явно преобразовать перечислитель класса enum в тип int, используя оператор static_cast:

```
#include <iostream>

int main()
{
    enum class Colors
    {
        PINK,
        GRAY
    };

    Colors color = Colors::GRAY;

    std::cout << color; // не будет работать, поскольку нет неявного
преобразования в тип int
    std::cout << static_cast<int>(color); // результатом будет 1

    return 0;
}
```

Если вы используете компилятор, поддерживающий C++11, то нет никакого смысла использовать обычные перечисления вместо классов enum.

Обратите внимание, ключевое слово class вместе с ключевым словом static являются одними из самых запутанных в языке C++, поскольку результат сильно зависит от варианта применения (контекста). Хотя классы enum используют

ключевое слово class, в C++ они не считаются традиционными «классами». О традиционных классах мы поговорим несколько позже.

Урок №63. Псевдонимы типов: typedef и type alias

Ключевое слово typedef позволяет программисту создать псевдоним для любого типа данных и использовать его вместо фактического имени типа. Чтобы объявить typedef (использовать псевдоним типа) - используйте ключевое слово typedef вместе с типом данных, для которого создается псевдоним, а затем, собственно, сам псевдоним. Например:

```
1. typedef double time_t; // используем time_t в качестве псевдонима для типа
   double
2.
3. // Следующие два стейтмента эквивалентны
4. double howMuch;
5. time_t howMuch;
```

Обычно к псевдонимам typedef добавляют окончание `_t`, указывая, таким образом, что идентификатором является тип, а не переменная.

typedef не определяет новый тип данных. Это просто псевдоним (другое имя) для уже существующего типа. Его можно использовать везде, где используется обычный тип.

Даже если следующее не имеет смысла, оно все равно разрешено в языке C++:

```
1. typedef long miles_t;
2. typedef long speed_t;
3.
4. miles_t distance = 8;
5. speed_t phr = 2100;
6.
7. // Следующее разрешено, поскольку обе переменные distance и phr являются типа
   long
8. distance = phr;
```

typedef и читабельность кода

typedef используется в улучшении документации и разборчивости кода. Имена таких типов, как char, int, long, double и bool хороши для описания того, какой тип возвращает функция, но чаще всего мы хотим знать, с какой целью возвращается значение. Например, рассмотрим следующую функцию:

```
1. int GradeTest();
```

Мы видим, что возвращаемым значением является целое число, но что оно означает? Количество пропущенных вопросов? Идентификационный номер учащегося? Код ошибки? Сам int ни о чем нам не говорит. Исправим ситуацию:

```
typedef int testScore_t;
testScore_t GradeTest();
```

С использованием возвращаемого типа `testScore_t` становится очевидным, что функция возвращает тип, значением которого является результат теста.

typedef и поддержка кода

typedef также позволяет изменить базовый тип объекта без внесения изменений в большое количество кода. Например, если вы использовали тип short для хранения идентификационного номера учащегося, но потом решили, что лучше использовать тип long, то вам придется прошерстить кучу кода для замены short-а на long. И, вероятно, было бы трудно определить, какой из типов short используется для хранения идентификационных номеров, а какой — для других целей.

С typedef же всё, что вам нужно сделать, — это изменить объявление `typedef short studentID_t` на `typedef long studentID_t`. Тем не менее, не стоит забывать об осторожности при изменении типа typedef на тип из другого семейства (например, из int на float или наоборот)! Новый тип данных может иметь проблемы со сравнением или делением целых чисел/чисел типа с плавающей точкой, которых старый тип не имел — об этом следует помнить.

typedef и кроссплатформенность

Еще одним большим преимуществом typedef является возможность скрывать специфические для определенных платформ (операционных систем) детали. На некоторых платформах тип int занимает 2 байта, на других - 4 байта. Таким образом, использование типа int для хранения более 2 байтов информации может быть потенциально опасным при написании кроссплатформенного кода.

Поскольку char, short, int и long не указывают свой размер, то для кроссплатформенных программ довольно часто используется typedef для определения псевдонимов, которые включают размер типа данных в битах. Например, int8_t - это 8-битный signed int, int16_t - это 16-битный signed int, а int32_t - это 32-битный signed int.

typedef и упрощение сложного

Хотя мы до сих пор рассматривали только простые типы данных, в языке C++ вы можете увидеть и следующие переменные/функции:

```
std::vector<std::pair<std::string, int>> pairlist;

```

```
3. boolean hasAttribute(std::vector<std::pair<std::string, int>> pairlist)
4. {
5.     // Что-то делаем
6. }
```

Писать `std::vector<std::pair<std::string, int>>` всякий раз, когда нужно использовать этот тип — не очень эффективно и затратно как по времени, так и по приложенным усилиям. Гораздо проще использовать typedef:

```
1. typedef std::vector<std::pair<std::string, int>> pairlist_t; // используем
   pairlist_t в качестве псевдонима для этого длиннющего типа данных
2.
3. pairlist_t pairlist; // объявляем pairlist_t
4.
5. boolean hasAttribute(pairlist_t pairlist) // используем pairlist_t в качестве
   типа параметра функции
6. {
7.     // Что-то делаем
8. }
```

Вот! Другое дело! Ведь проще использовать `pairlist_t` вместо `std::vector<std::pair<std::string, int>>`, не так ли?

Не переживайте, если вы еще не знаете, что такое std::vector, std::pair и прочее. Гораздо важнее сейчас усвоить, что с помощью typedef вы можете давать простые имена сложным типам данных, что сделает их проще как для использования, так и для понимания.

type alias

У typedef есть также свои нюансы. Во-первых, легко забыть, что пишется первым: псевдоним типа или имя типа:

```
1. typedef time_t double; // неправильно
2. typedef double time_t; // правильно
```

Во-вторых, синтаксис typedef становится уже менее привлекательным в связке со сложными типами данных (об этом мы поговорим детально, когда будем рассматривать указатели на функции).

Для решения этих проблем, в C++11 ввели новый улучшенный синтаксис для typedef, который имитирует способ объявления переменных. Этот синтаксис называется **type alias**. С помощью **type alias** мы пишем имя, которое затем используется как синоним конкретного типа данных (т.е. принцип тот же, но синтаксис более удобен).

Следующий typedef:

```
1. typedef double time_t; // используем time_t в качестве псевдонима для типа
   double
```

В C++11 можно объявить как:

```
1. using time_t = double; // используем time_t в качестве псевдонима для типа
   double
```

Эти два способа функционально эквивалентны.

Обратите внимание, что хоть мы и используем ключевое слово using, оно не имеет ничего общего с using-стейтментами. Это ключевое слово имеет различный функционал в зависимости от контекста.

Новый синтаксис создания псевдонимов создает меньше проблем при использовании в сложных ситуациях, и его рекомендуется применять вместо обычного typedef, если ваш компилятор поддерживает C++11.

Правило: Используйте type alias вместо typedef, если ваш компилятор поддерживает C++11.

Тест

Задание №1

Используя следующий прототип функции:

```
1. int editData();
```

Преобразуйте тип возвращаемого значения int в status_t, используя ключевое слово typedef. В ответе к этому заданию укажите стейтмент typedef и обновленный прототип функции.

Задание №2

Используя прототип функции из задания №1, преобразуйте тип возвращаемого значения int в status_t, используя ключевое слово using (C++11). В ответе к этому заданию укажите стейтмент создания псевдонима типа и обновленный прототип функции.

Урок №64. Структуры

В программировании есть много случаев, когда может понадобиться больше одной переменной для представления определенного объекта. Например, для представления самого себя, вы, скорее всего, захотите указать свое имя, день рождения, рост, вес или любую другую информацию:

```
1. std::string myName;
2. int myBirthDay;
3. int myBirthMonth;
4. int myBirthYear;
5. int myHeight;
6. int myWeight;
```

Но теперь у вас есть 6 отдельных независимых переменных. Если вы захотите передать информацию о себе в функцию, то вам придется передавать каждую переменную по отдельности. Кроме того, если вы захотите хранить информацию о ком-то еще, то вам придется дополнительно объявить еще 6 переменных на каждого человека! Невооруженным глазом видно, что такая реализация не очень эффективна.

К счастью, язык C++ позволяет программистам создавать свои собственные **пользовательские типы данных** — типы, которые группируют несколько отдельных переменных вместе. Одним из простейших пользовательских типов данных является структура. **Структура позволяет сгруппировать переменные разных типов в единое целое**.

Объявление и определение структур

Поскольку структуры определяются программистом, то вначале мы должны сообщить компилятору, как она вообще будет выглядеть. Для этого используется **ключевое слово struct**:

```
1. struct Employee
2. {
3.     short id;
4.     int age;
5.     double salary;
6. };
```

Мы определили структуру с именем `Employee`.

Она содержит 3 переменные:

- `id` типа short;

- `age` типа int;
- `salary` типа double.

Эти переменные, которые являются частью структуры, называются **членами структуры** (или ***"полями структуры"***). `Employee` - это простое объявление структуры. Хотя мы и указали компилятору, что она имеет переменные-члены, память под нее сейчас не выделяется. Имена структур принято писать с заглавной буквы, чтобы отличать их от имен переменных.

> ***Предупреждение:*** Одна из самых простых ошибок в C++ — забыть точку с запятой в конце объявления структуры. Это приведет к ошибке компиляции в следующей строке кода. Современные компиляторы, такие как Visual Studio версии 2010, а также более новых версий, укажут вам, что вы забыли точку с запятой в конце, но более старые компиляторы могут этого и не сделать, из-за чего такую ошибку будет трудно найти.

Чтобы использовать структуру `Employee`, нам нужно просто объявить переменную типа `Employee`:

```
1. Employee john; // имя структуры Employee начинается с заглавной буквы, а
   переменная john - с маленькой
```

Здесь мы определили переменную типа `Employee` с именем `john`. Как и в случае с обычными переменными, определение переменной, типом которой является структура, приведет к выделению памяти для этой переменной.

Объявить можно и несколько переменных одной структуры:

```
1. Employee john; // создаем отдельную структуру Employee для John-a
2. Employee james; // создаем отдельную структуру Employee для James-a
```

Доступ к членам структур

Когда мы объявляем переменную структуры, например, `Employee john`, то `john` ссылается на всю структуру. Для того, чтобы получить доступ к отдельным её членам, используется **оператор выбора члена** (`.`). Например, в коде, приведенном ниже, мы используем оператор выбора членов для инициализации каждого члена структуры:

```
1. Employee john; // создаем отдельную структуру Employee для John-a
2. john.id = 8; // присваиваем значение члену id структуры john
3. john.age = 27; // присваиваем значение члену age структуры john
4. john.salary = 32.17; // присваиваем значение члену salary структуры john
5.
6. Employee james; // создаем отдельную структуру Employee для James-a
```

```
james.id = 9; // присваиваем значение члену id структуры james
james.age = 30; // присваиваем значение члену age структуры james
james.salary = 28.35; // присваиваем значение члену salary структуры james
```

Как и в случае с обычными переменными, переменные-члены структуры не инициализируются автоматически и обычно содержат мусор. Инициализировать их нужно вручную.

В примере, приведенном выше, легко определить, какая переменная относится к структуре `John`, а какая — к структуре `James`. Это обеспечивает гораздо более высокий уровень организации, чем в случае с обычными отдельными переменными.

Переменные-члены структуры работают так же, как и простые переменные, поэтому с ними можно выполнять обычные арифметические операции и операции сравнения:

```
int totalAge = john.age + james.age;

if (john.salary > james.salary)
    cout << "John makes more than James\n";
else if (john.salary < james.salary)
    cout << "John makes less than James\n";
else
    cout << "John and James make the same amount\n";

// James получил повышение в должности
james.salary += 3.75;

// Сегодня день рождения у John-а
++john.age; // используем пре-инкремент для увеличения возраста John-а на 1 год
```

Инициализация структур

Инициализация структур путем присваивания значений каждому члену по порядку — занятие довольно громоздкое (особенно, если этих членов много), поэтому в языке C++ есть более быстрый способ инициализации структур — с помощью **списка инициализаторов**. Он позволяет инициализировать некоторые или все члены структуры во время объявления переменной типа struct:

```
struct Employee
{
    short id;
    int age;
    double salary;
};

Employee john = { 5, 27, 45000.0 }; // john.id = 5, john.age = 27, john.salary = 45000.0
```

```
9. Employee james = { 6, 29}; // james.id = 6, james.age = 29, james.salary = 0.0 (инициализация по умолчанию)
```

В C++11 также можно использовать uniform-инициализацию:

```
1. Employee john { 5, 27, 45000.0 }; // john.id = 5, john.age = 27, john.salary = 45000.0
2. Employee james { 6, 29 }; // james.id = 6, james.age = 29, james.salary = 0.0 (инициализация по умолчанию)
```

Если в списке инициализаторов не будет одного или нескольких элементов, то им присвоятся значения по умолчанию (обычно, `0`). В примере, приведенном выше, члену `james.salary` присваивается значение по умолчанию `0.0`, так как мы сами не предоставили никакого значения во время инициализации.

C++11/14: Инициализация нестатических членов структур

В C++11 добавили возможность присваивать нестатическим (обычным) членам структуры значения по умолчанию. Например:

```
1. #include <iostream>
2.
3. struct Triangle
4. {
5.     double length = 2.0;
6.     double width = 2.0;
7. };
8.
9. int main()
10. {
11.     Triangle z; // длина = 2.0, ширина = 2.0
12.
13.     z.length = 3.0; // вы также можете присваивать членам структуры и другие значения
14.
15.     return 0;
16. }
```

К сожалению, в C++11 синтаксис инициализации нестатических членов структуры несовместим с синтаксисом списка инициализаторов или uniform-инициализацией. В C++11 следующая программа не скомпилируется:

```
1. #include <iostream>
2.
3. struct Triangle
4. {
5.     double length = 2.0; // нестатическая инициализация членов
6.     double width = 2.0;
7. };
8.
9. int main()
10. {
11.     Triangle z{ 3.0, 3.0 }; // uniform-инициализация
12.
```

```
13.     return 0;
14. }
```

Следовательно, вам нужно будет решить, хотите ли вы использовать инициализацию нестатических полей или uniform-инициализацию. uniform-инициализация более гибкая, поэтому рекомендуется использовать именно её.

Однако в C++14 это ограничение было снято, и оба варианта можно использовать. Мы еще поговорим детально о статических членах структуры на соответствующем уроке.

Присваивание значений членам структур

До C++11, если бы мы захотели присвоить значения членам структуры, то нам бы пришлось это делать вручную каждому члену по отдельности:

```
1. struct Employee
2. {
3.     short id;
4.     int age;
5.     double salary;
6. };
7.
8. Employee john;
9. john.id = 5;
10. john.age = 27;
11. john.salary = 45000.0;
```

Это боль, особенно когда членов в структуре много. В C++11 вы можете присваивать значения членам структур, используя список инициализаторов:

```
1. struct Employee
2. {
3.     short id;
4.     int age;
5.     double salary;
6. };
7.
8. Employee john;
9. john = { 5, 27, 45000.0 }; // только C++11
```

Структуры и функции

Большим преимуществом использования структур, нежели отдельных переменных, является возможность передать всю структуру в функцию, которая должна работать с её членами:

```
1. #include <iostream>
2.
3. struct Employee
4. {
```

```
    short id;
    int age;
    double salary;
};

void printInformation(Employee employee)
{
    std::cout << "ID: " << employee.id << "\n";
    std::cout << "Age: " << employee.age << "\n";
    std::cout << "Salary: " << employee.salary << "\n";
}

int main()
{
    Employee john = { 21, 27, 28.45 };
    Employee james = { 22, 29, 19.29 };

    // Выводим информацию о John-е
    printInformation(john);

    std::cout << "\n";

    // Выводим информацию о James-е
    printInformation(james);

    return 0;
}
```

В примере, приведенном выше, мы передали структуру `Employee` в функцию printInformation(). Это позволило нам не передавать каждую переменную по отдельности. Более того, если мы когда-либо захотим добавить новых членов в структуру `Employee`, то нам не придется изменять объявление или вызов функции!

Результат выполнения программы:

```
ID: 21
Age: 27
Salary: 28.45

ID: 22
Age: 29
Salary: 19.29
```

Функция также может возвращать структуру (это один из тех немногих случаев, когда функция может возвращать несколько переменных). Например:

```
#include <iostream>

struct Point3d
{
    double x;
    double y;
    double z;
};

```

```
Point3d getZeroPoint()
{
    Point3d temp = { 0.0, 0.0, 0.0 };
    return temp;
}

int main()
{
    Point3d zero = getZeroPoint();

    if (zero.x == 0.0 && zero.y == 0.0 && zero.z == 0.0)
        std::cout << "The point is zero\n";
    else
        std::cout << "The point is not zero\n";

    return 0;
}
```

Результат выполнения программы:

```
The point is zero
```

Вложенные структуры

Одни структуры могут содержать другие структуры. Например:

```
struct Employee
{
    short id;
    int age;
    double salary;
};

struct Company
{
    Employee CEO; // Employee - это структура внутри структуры Company
    int numberOfEmployees;
};

Company myCompany;
```

В этом случае, если бы мы хотели узнать, какая зарплата у CEO (исполнительного директора), то нам бы пришлось использовать оператор выбора членов дважды:

```
myCompany.CEO.salary
```

Сначала мы выбираем поле `CEO` из структуры `myCompany`, а затем поле `salary` из структуры `Employee`.

Вы можете использовать вложенные списки инициализаторов с вложенными структурами:

```
struct Employee
{
```

```
    short id;
    int age;
    float salary;
};

struct Company
{
    Employee CEO; // Employee является структурой внутри структуры Company
    int numberOfEmployees;
};

Company myCompany = {{ 3, 35, 55000.0f }, 7 };
```

Размер структур

Как правило, размер структуры - это сумма размеров всех её членов, но не всегда! Например, рассмотрим структуру `Employee`. На большинстве платформ тип short занимает 2 байта, тип int - 4 байта, а тип double - 8 байт. Следовательно, ожидается, что `Employee` будет занимать `2 + 4 + 8 = 14` байт. Чтобы узнать точный размер `Employee`, мы можем воспользоваться оператором sizeof:

```
#include <iostream>

struct Employee
{
    short id;
    int age;
    double salary;
};

int main()
{
    std::cout << "The size of Employee is " << sizeof(Employee) << "\n";

    return 0;
}
```

Результат выполнения программы (на компьютере автора):

```
The size of Employee is 16
```

Оказывается, мы можем сказать только, что размер структуры будет, *по крайней мере,* не меньше суммы размеров всех её членов. Но он может быть и больше! По соображениям производительности компилятор иногда может добавлять "пробелы/промежутки" в структуры.

В структуре `Employee` компилятор неявно добавил 2 байта после члена `id`, увеличивая размер структуры до 16 байтов вместо 14. Описание причины, по которой это происходит, выходит за рамки этого урока, но если вы хотите знать больше, то можете прочитать о **выравнивании данных в Википедии**.

Доступ к структурам из нескольких файлов

Поскольку объявление структуры не провоцирует выделение памяти, то использовать предварительное объявление для нее вы не сможете. Но есть обходной путь: если вы хотите использовать объявление структуры в нескольких файлах (чтобы иметь возможность создавать переменные этой структуры в нескольких файлах), поместите объявление структуры в заголовочный файл и подключайте этот файл везде, где необходимо использовать структуру.

Переменные типа struct подчиняются тем же правилам, что и обычные переменные. Следовательно, если вы хотите сделать переменную структуры доступной в нескольких файлах, то вы можете использовать ключевое слово extern.

Заключение

Структуры очень важны в языке C++, поскольку их понимание - это первый большой шаг на пути к объектно-ориентированному программированию! Чуть позже мы рассмотрим другой пользовательский тип данных — класс (который является продолжением темы структур).

Тест

Задание №1

У вас есть веб-сайт и вы хотите отслеживать, сколько денег вы зарабатываете в день от размещенной на нем рекламы. Объявите структуру `Advertising`, которая будет отслеживать:

- сколько объявлений вы показали посетителям (1);
- сколько процентов посетителей нажали на объявления (2);
- сколько вы заработали в среднем за каждое нажатие на объявления (3).

Значения этих 3-х полей должен вводить пользователь. Передайте структуру `Advertising` в функцию, которая выведет каждое из этих значений, а затем подсчитает, сколько всего денег вы заработали за день (перемножьте все 3 поля).

Задание №2

Создайте структуру для хранения дробных чисел. Структура должна иметь 2 члена: целочисленный числитель и целочисленный знаменатель. Объявите две дробные переменные и получите их значения от пользователя. Напишите функцию multiply()

(параметрами которой будут эти две переменные), которая будет перемножать эти числа и выводить результат в виде десятичного числа.

Урок №65. Вывод типов: ключевое слово auto

До C++11 ключевое слово auto было наименее используемым ключевым словом в языке C++. Как мы уже знаем из предыдущих уроков, локальные переменные имеют автоматическую продолжительность жизни (создаются в точке определения и уничтожаются в конце блока, в котором определены).

Ключевое слово auto использовалось для явного указания, что переменная должна иметь автоматическую продолжительность жизни:

```
1. int main()
2. {
3.     auto int boo(7); // явно указываем, что переменная boo типа int должна
   иметь автоматическую продолжительность жизни
4.
5.     return 0;
6. }
```

Однако, поскольку все переменные в новых версиях языка C++ по умолчанию имеют автоматическую продолжительность жизни (если явно не указать другой тип продолжительности жизни), ключевое слово auto стало лишним и, следовательно, устаревшим.

В C++11 значение ключевого слова auto изменилось. Рассмотрим следующий стейтмент:

```
1. double x = 4.0;
```

Если C++ и так знает, что `4.0` является литералом типа double, то зачем нам дополнительно указывать, что переменная `x` должна быть типа double? Правда, было бы неплохо, если бы мы могли указать переменной принять соответствующий тип данных, основываясь на инициализируемом значении?

Начиная с C++11, **ключевое слово auto** при инициализации переменной может использоваться вместо типа переменной, чтобы сообщить компилятору, что он должен присвоить тип переменной исходя из инициализируемого значения. Это называется **выводом типа** (или ***"автоматическим определением типа данных компилятором"***).

Например:

```
1. auto x = 4.0; // 4.0 - это литерал типа double, поэтому и x должен быть типа
   double
2. auto y = 3 + 4; // выражение 3 + 4 обрабатывается как целочисленное, поэтому и
   переменная y должна быть типа int
```

Это работает даже с возвращаемыми значениями функций:

```
1. int subtract(int a, int b)
2. {
3.     return a - b;
4. }
5.
6. int main()
7. {
8.     auto result = subtract(4, 3); // функция subtract() возвращает значение типа int и, следовательно, переменная result также должна быть типа int
9.     return 0;
10. }
```

Обратите внимание, это работает только с инициализированными переменными. Переменные, объявленные без инициализации, не могут использовать эту особенность (поскольку нет инициализируемого значения, и компилятор не может знать, какой тип данных присвоить переменной).

Используя ключевое слово auto вместо фундаментальных типов данных, мы не сэкономим много времени или усилий, но на следующих уроках, когда типы данных будут более сложными и длинными, ключевое слово auto может очень пригодиться.

Ключевое слово auto и параметры функций

Многие новички пытаются сделать что-то вроде следующего:

```
1. void mySwap(auto a, auto b)
2. {
3.     auto x = a;
4.     a = b;
5.     b = x;
6. }
```

Это не сработает, так как компилятор не может определить типы данных для параметров функции `a` и `b` во время компиляции.

Если вы хотите создать функцию, которая будет работать с разными типами данных, то вам лучше воспользоваться шаблонами функций, а не выводом типа. Это ограничение, возможно, отменят в будущих версиях C++ (когда auto будет использоваться как сокращенный способ создания шаблонов функций), но в C++14 это не работает. Единственное исключение – лямбда-выражения (но это уже другая тема).

Вывод типов в C++14

В C++14 функционал ключевого слова auto был расширен до автоматического определения типа возвращаемого значения функции. Например:

```
1. auto subtract(int a, int b)
2. {
3.     return a - b;
4. }
```

Так как выражение `a - b` является типа int, то компилятор делает вывод, что и функция должна быть типа int.

Хотя это может показаться удобным, так делать не рекомендуется. Тип возвращаемого значения функции помогает понять caller-у, что именно функция должна возвращать. Если конкретный тип не указан, то caller может неверно интерпретировать тип возвращаемого значения, что может привести к непреднамеренным ошибкам.

Так почему же использование auto при инициализации переменных — это хорошо, а с функциями — плохо? Дело в том, что auto можно использовать при определении переменной, так как значение, из которого компилятор делает выводы о типе переменной, находится прямо там — в правой части стейтмента. Однако с функциями это не так - нет контекста, который бы указывал, какого типа данных будет возвращаемое значение. Фактически, пользователю придется лезть в тело функции, чтобы определить тип возвращаемого значения. Следовательно, такой способ не только не практичен, но и более подвержен ошибкам.

trailing-синтаксис в C++11

В C++11 появилась возможность использовать **синтаксис типа возвращаемого значения trailing** (или просто ***"trailing-синтаксис"***), когда компилятор делает выводы о типе возвращаемого значения по конечной части прототипа функции. Например, рассмотрим следующее объявление функции:

```
1. int subtract(int a, int b);
```

В C++11 это можно записать как:

```
1. auto subtract(int a, int b) -> int;
```

В этом случае auto не выполняет вывод типа - это всего лишь часть синтаксиса типа возвращаемого значения trailing. Зачем это стоит использовать? В основном, из-за удобства. Например, можно выстроить в колонку все названия ваших функций:

```
auto subtract(int a, int b) -> int;
auto divide(double a, double b) -> double;
auto printThis() -> void;
auto calculateResult(int a, double x) -> std::string;
```

Но этот синтаксис более полезен в сочетании с некоторыми другими продвинутыми особенностями языка C++ такими, как классы и ключевое слово decltype.

На данный момент я рекомендую придерживаться традиционного (обычного) синтаксиса для определения типа возвращаемого значения функции (без использования ключевого слова auto).

Заключение

Начиная с C++11 ключевое слово auto может использоваться вместо типа переменной при инициализации для выполнения вывода типа. Во всех других случаях использования ключевого слова auto следует избегать, если на это нет веских причин.

Глава №4. Итоговый тест

В этой главе мы рассмотрели много материала. Если вы дошли до этого момента, то я вас поздравляю — вы проделали немало работы и это уже хороший шаг на пути к изучению языка C++ в частности и программированию в целом! Сейчас же давайте закрепим пройденный материал.

Теория

Блок стейтментов (или ***"составной оператор"***) обрабатывается компилятором так, как если бы это был один стейтмент. Составные операторы помещаются в фигурные скобки `{` и `}` и используются почти везде.

Локальные переменные создаются в точке определения и уничтожаются при выходе из блока, в котором они объявлены. Доступ к ним возможен только внутри этого же блока.

Глобальные переменные создаются, когда программа запускается, и уничтожаются, когда она завершает свое выполнение. Они могут использоваться в любом месте программы. Неконстантные глобальные переменные следует избегать, потому что это — зло.

Ключевое слово static может использоваться для преобразования глобальной переменной во внутреннюю (с внутренней связью), чтобы её можно было использовать только в том файле, в котором она объявлена. Также ключевое слово static используют для указания того, что локальная переменная должна иметь статическую продолжительность жизни. А это означает, что она будет сохранять свое значение даже после выхода из своей области видимости.

Пространство имен - это область, в которой гарантируется уникальность всех имен. Отличный способ избежать конфликтов имен. Не используйте using-стейтменты вне тела функций.

Неявное преобразование типов данных происходит, когда один тип данных конвертируется в другой тип без использования одного из операторов явного преобразования. **Явное преобразование типа** происходит, когда один тип данных конвертируется в другой с помощью одного из операторов явного преобразования. В некоторых случаях это абсолютно безопасно, в других случаях данные могут быть потеряны. Избегайте использования конвертации C-style, вместо нее используйте оператор static_cast.

std::string — это простой способ работы с текстовыми строками (текст помещается в двойные кавычки).

Перечисления позволяют создавать собственные (пользовательские) типы данных. **Классы enum** — это те же перечисления, но надежнее и безопаснее. Используйте их вместо обычных перечислений, если ваш компилятор поддерживает C++11.

typedef позволяет создавать псевдонимы для типов данных. Целочисленные типы данных с фиксированным размером реализованы с помощью typedef. Псевдонимы типов полезны для присваивания простых имен сложным типам данных.

И, наконец, **структуры**. Они позволяют сгруппировать отдельные переменные в единое целое. Доступ к членам структуры осуществляется через оператор выбора членов (`.`). Объектно-ориентированное программирование в значительной степени основывается именно на структурах, поэтому, если вы изучили только одну вещь из этой главы, то лучше, чтобы это были структуры.

Тест

При разработке игры мы решили, что в ней должны быть монстры, потому что всем нравится сражаться с монстрами. Объявите структуру, которая представляет вашего монстра. Монстр может быть разным: `ogre`, `goblin`, `skeleton`, `orc` и `troll`. Если ваш компилятор поддерживает C++11, то используйте классы enum, если нет — обычные перечисления.

Каждый монстр также должен иметь имя (используйте std::string) и количество здоровья, которое отображает, сколько урона он может получить, прежде чем умрет. Напишите функцию printMonster(), которая выведет все члены структуры. Объявите монстров типа `goblin` и `orc`, инициализируйте их, используя список инициализаторов, и передайте в функцию printMonster().

Пример результата выполнения вашей программы:

```
This Goblin is named John and has 170 health.
This Orc is named James and has 35 health.
```

Урок №66. Операторы управления потоком выполнения программ

При запуске программы, центральный процессор (сокр. *"ЦП"*) начинает выполнение кода с первой строки функции main(), выполняя определенное количество стейтментов, а затем завершает выполнение при завершении блока main(). Последовательность стейтментов, которые выполняет ЦП, называется **порядком выполнения программы** (или ***"потоком выполнения программы"***).

Порядок выполнения программ

Большинство программ, которые мы рассматривали до этого момента, были **линейными** с **последовательным выполнением**, то есть порядок выполнения у них один и тот же каждый раз: выполняются одни и те же стейтменты, даже если значения, которые вводит пользователь, — меняются.

Но на практике это не всегда может быть то, что нам нужно. Например, если мы попросим пользователя сделать выбор между `+` и `/`, а пользователь введет некорректный символ (например, `*`), то в идеале нам нужно было бы попросить его ввести символ еще раз. Но это невозможно в линейной программе. Кроме того, бывают случаи, когда нужно выполнить что-то несколько раз, но количество этих повторений наперед неизвестно. Например, если бы мы хотели вывести все целые числа от `0` до числа, которое введет пользователь, то в линейной программе мы бы не смогли это сделать, не зная наперед число, которое введет пользователь.

К счастью, в языке C++ есть **операторы управления порядком выполнения программы**, которые позволяют программисту изменить поток выполнения программы центральным процессором.

Остановка

Самый простой оператор управления порядком выполнения программы - это **остановка**, которая сообщает программе немедленно прекратить свое выполнение. В языке C++ остановка осуществляется с помощью **функции exit()**, которая определена в заголовочном файле cstdlib. Функция exit() принимает целочисленный параметр, который затем возвращает обратно в операционную систему в качестве кода выхода. Например:

```
1. #include <iostream>
2. #include <cstdlib> // для функции exit()
3.
4. int main()
```

```
{
    std::cout << 5;
    exit(0); // завершаем выполнение программы и возвращаем 0 обратно в операционную систему

    // Следующие стейтменты никогда не выполнятся
    std::cout << 3;
    return 0;
}
```

Прыжок

Следующим оператором управления порядком выполнения программы является **прыжок** (или ***"переход"***). Он безоговорочно сообщает компилятору во время выполнения перейти от одного стейтмента к другому, т.е. выполнить прыжок. **Ключевые слова goto**, **break** и **continue** являются разными типами прыжков, об их различиях мы поговорим несколько позже.

Вызовы функций — это тоже, в некоторой степени, прыжки. При выполнении вызова функции, ЦП переходит к началу вызываемой функции. Когда вызываемая функция заканчивается, выполнение возвращается к следующему стейтменту, который находится после вызова этой функции.

Условные ветвления

Условное ветвление заставляет программу изменить свой порядок выполнения, основываясь на значении результата выражения. Одним из основных операторов условного ветвления является if, который вы уже могли видеть в программах раньше. Например:

```
int main()
{
    // Делаем A
    if (expression)
        // Делаем B
    else
        // Делаем C

    // Делаем D
}
```

Здесь есть два возможных пути выполнения программы. Если результатом выражения будет true, то программа выполнит `A`, `B` и `D`. Если же результатом выражения будет false, то программа выполнит `A`, `C` и `D`. Таким образом, выполнится либо `B`, либо `C`. Оба варианта выполняться вместе не будут. Это уже не линейная программа.

Ключевое слово switch также предоставляет механизм для выполнения условного ветвления. Более подробно об операторах if и switch мы поговорим на соответствующих уроках.

Циклы

Цикл заставляет программу многократно выполнять определенное количество стейтментов до тех пор, пока заданное условие не станет ложным. Например:

```
1. int main()
2. {
3.     // Делаем A
4.     // B делается в цикле 0 или больше раз
5.     // Делаем C
6. }
```

Эта программа может выполняться как `ABC`, `ABBC`, `ABBBC`, `ABBBBC` или даже `AC`. Опять же, она больше не является линейной, её порядок выполнения зависит от того, сколько раз выполнится цикл (если вообще выполнится).

В языке C++ есть 4 типа циклов:

- цикл while;
- цикл do while;
- цикл for;
- цикл foreach (добавили в C++11).

Мы подробно рассмотрим каждый из них в этой главе, кроме foreach (о нем немного позже).

Исключения

Исключения предлагают механизм обработки ошибок, возникающих в функции. Если в функции возникает ошибка, с которой она не может справиться, то она может выбросить исключение. Это заставит ЦП перейти к ближайшему блоку кода, который обрабатывает исключения данного типа.

Обработка исключений - это довольно сложная тема и это единственный тип оператора управления порядком выполнения, который мы не будем рассматривать в этой главе.

Заключение

Используя операторы управления порядком выполнения программы, вы можете повлиять на поток выполнения программы центральным процессором, а также на то, из-за чего он может быть прерван. До этого момента функционал наших программ был очень ограничен. Теперь же с операторами управления порядком выполнения программ мы сможем осуществить огромное количество разных интересных вещей, например, отображение меню до тех пор, пока пользователь не сделает правильный выбор; вывод каждого числа между `x` и `y`, и много-много чего еще.

Как только вы разберетесь с этой темой, вы перейдете на новый, более качественный уровень. Больше вы не будете ограничены игрушечными программами или простенькими упражнениями - вы сможете писать полноценные программы. Вот именно здесь и начинается всё самое интересное.

Погнали!

Урок №67. Операторы условного ветвления if/else

Самыми простыми условными ветвлениями в языке C++ являются стейтменты if/else. Они выглядят следующим образом:

```
if (выражение)
    стейтмент1
```

Либо так:

```
if (выражение)
    стейтмент1
else
    стейтмент2
```

`выражение` называется **условием** (или ***"условным выражением"***). Если результатом `выражения` является true (любое ненулевое значение), то выполняться будет `стейтмент1`. Если же результатом `выражения` является false (0), то выполняться будет `стейтмент2`. Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Enter a number: ";
6.     int a;
7.     std::cin >> a;
8.
9.     if (a > 15)
10.        std::cout << a << " is greater than 15\n";
11.    else
12.        std::cout << a << " is not greater than 15\n";
13.
14.    return 0;
15.}
```

Использование нескольких операций в ветвлениях if/else

Оператор if выполняет *только одну* операцию, если `выражение` является true, и также *только одну* операцию else, если `выражение` — false. Чтобы выполнить несколько операций подряд, используйте блок стейтментов:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Enter a number: ";
6.     int a;
7.     std::cin >> a;
8.
```

```
    if (a > 15)
        {
        // Обе операции будут выполнены, если a > 15
        std::cout << "You entered " << a << "\n";
        std::cout << a << " is greater than 15\n";
        }
    else
        {
        // Обе операции будут выполнены, если a <= 15
        std::cout << "You entered " << a << "\n";
        std::cout << a << " is not greater than 15\n";
        }

    return 0;
}
```

Неявное указание блоков

Если программист не указал скобки для блока стейтментов if или else, то компилятор неявно сделает это за него. Таким образом, следующее:

```
if (выражение)
    стейтмент1
else
    стейтмент2
```

Будет выполняться как:

```
if (выражение)
{
    стейтмент1
}
else
{
    стейтмент2
}
```

По сути, это не имеет значения. Однако начинающие программисты иногда пытаются сделать что-то вроде следующего:

```
#include <iostream>

int main()
{
    if (1)
        int a = 4;
    else
        int a = 5;

    std::cout << a;

    return 0;
}
```

Программа не скомпилируется, и в итоге мы получим ошибку, что идентификатор `a` не определен. А произойдет это из-за того, что программа будет выполняться следующим образом:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     if (1)
6.     {
7.         int a = 4;
8.     } // переменная a уничтожается здесь
9.     else
10.    {
11.        int a = 5;
12.    } // переменная a уничтожается здесь
13.
14.    std::cout << a; // переменная a здесь не определена
15.
16.    return 0;
17. }
```

В этом контексте становится понятным, что переменная `a` имеет локальную область видимости и уничтожается в конце блока, в котором выполняется её инициализация. И, когда мы дойдем до строчки с std::cout, переменная `a` уже перестанет существовать.

Связывание стейтментов if

Стейтменты if/else можно использовать в связке:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Enter a number: ";
6.     int a;
7.     std::cin >> a;
8.
9.     if (a > 15)
10.        std::cout << a << " is greater than 15\n";
11.    else if (a < 15)
12.        std::cout << a << " is less than 15\n";
13.    else
14.        std::cout << a << " is exactly 15\n";
15.
16.    return 0;
17. }
```

Вложенные ветвления if/else

Одни стейтменты if могут быть вложены в другие стейтменты if:

```
1. #include <iostream>
```

```
2.
3.  int main()
4.  {
5.      std::cout << "Enter a number: ";
6.      int a;
7.      std::cin >> a;
8.
9.      if (a > 15) // внешний оператор if
10.         // Это плохой способ написания вложенных стейтментов if
11.         if (a < 25) // внутренний оператор if
12.             std::cout << a << " is between 15 and 25\n";
13.
14.         // К какому if относится следующий else?
15.         else
16.             std::cout << a << " is greater than or equal to 25\n";
17.
18.     return 0;
19. }
```

Обратите внимание, в программе, приведенной выше, мы можем наблюдать **потенциальную ошибку двусмысленности оператора else**. К какому if относится оператор else: к внешнему или к внутреннему?

Дело в том, что оператор else всегда относится к последнему незакрытому оператору if в блоке, в котором находится сам else. Т.е. в программе, приведенной выше, else относится к внутреннему if.

Чтобы избежать таких вот неоднозначностей при вложенности операторов условного ветвления, рекомендуется использовать блоки стейтментов (указывать скобки). Например, вот та же программа, приведенная выше, но уже без двусмысленности:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      std::cout << "Enter a number: ";
6.      int a;
7.      std::cin >> a;
8.
9.      if (a > 15)
10.     {
11.         if (a < 25)
12.             std::cout << a << " is between 15 and 25\n";
13.         else // относится к внутреннему оператору if
14.             std::cout << a << " is greater than or equal to 25\n";
15.     }
16.
17.     return 0;
18. }
```

Теперь понятно, что оператор else относится к внутреннему оператору if.

Использование скобок также позволяет явно указать привязку else к внешнему стейтменту if:

```
#include <iostream>

int main()
{
    std::cout << "Enter a number: ";
    int a;
    std::cin >> a;

    if (a > 15)
    {
        if (a < 25)
            std::cout << a << " is between 15 and 25\n";
    }
    else // относится к внешнему оператору if
        std::cout << a << " is less than 15\n";

    return 0;
}
```

Используя блоки стейтментов, мы уточняем, к какому if следует прикреплять определенный else. Без блоков оператор else будет прикрепляться к ближайшему незакрытому оператору if.

Использование логических операторов в ветвлениях if/else

Также вы можете проверить сразу несколько условий в ветвлениях if/else, используя логические операторы:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int a;
    std::cin >> a;

    std::cout << "Enter another integer: ";
    int b;
    std::cin >> b;

    if (a > 0 && b > 0) // && - это логическое И. Проверяем, являются ли оба условия истинными
        std::cout << "Both numbers are positive\n";
    else if (a > 0 || b > 0) // || - это логическое ИЛИ. Проверяем, является ли истинным хоть одно из условий
        std::cout << "One of the numbers is positive\n";
    else
        std::cout << "Neither number is positive\n";

    return 0;
}
```

Основные использования ветвлений if/else

Ветвления if/else активно используются для проверки ошибок. Например, чтобы вычислить квадратный корень значения, параметр, который передается в функцию для вычисления, — обязательно должен быть положительным:

```
#include <iostream>
#include <cmath> // для функции sqrt()

void printSqrt(double value)
{
    if (value >= 0.0)
        std::cout << "The square root of " << value << " is " << sqrt(value) << "\n";
    else
        std::cout << "Error: " << value << " is negative\n";
}
```

Также операторы if используют для **ранних возвратов** — когда функция возвращает управление обратно в caller еще до завершения выполнения самой функции. В программе, приведенной ниже, если значением параметра является отрицательное число, то функция сразу же возвращает в caller символьную константу или перечислитель в качестве кода ошибки:

```
#include <iostream>

enum class ErrorCode
{
    ERROR_SUCCESS = 0,
    ERROR_NEGATIVE_NUMBER = -1
};

ErrorCode doSomething(int value)
{
    // Если параметром value является отрицательное число,
    if (value < 0)
       // то сразу же возвращаем код ошибки
        return ErrorCode::ERROR_NEGATIVE_NUMBER;

    // Что-нибудь делаем

    return ErrorCode::ERROR_SUCCESS;
}

int main()
{
    std::cout << "Enter a positive number: ";
    int a;
    std::cin >> a;

    if (doSomething(a) == ErrorCode::ERROR_NEGATIVE_NUMBER)
    {
        std::cout << "You entered a negative number!\n";
    }
    else
    {
```

```
33.         std::cout << "It worked!\n";
34.     }
35.
36.     return 0;
37. }
```

Ветвления if/else также обычно используют для выполнения простых математических операций. Например, рассмотрим функцию min(), которая возвращает минимальное из 2-х чисел:

```
1. int min(int a, int b)
2. {
3.     if (a > b)
4.         return b;
5.     else
6.         return a;
7. }
```

Эта функция настолько проста, что её можно записать с помощью условного тернарного оператора:

```
1. int min(int a, int b)
2. {
3.     return (a > b) ? b : a;
4. }
```

Нулевые стейтменты

Также в C++ можно не указывать основную часть оператора if. Такие стейтменты называются **нулевыми стейтментами** (или ***"null-стейтментами"***). Объявить их можно, используя точку с запятой вместо выполняемой операции. В целях улучшения читабельности кода, точка с запятой нулевого стейтмента обычно пишется с новой строки. Таким образом, мы *явно указываем*, что хотим использовать null-стейтмент, уменьшая вероятность не заметить его использования:

```
1. if (a > 15)
2.     ; // это нулевой стейтмент
```

Хотя нулевые стейтменты редко используются в сочетании с оператором if, но, из-за неосторожности, это может привести к проблемам. Рассмотрим следующий фрагмент кода:

```
1. if (a == 0);
2.     a = 1;
```

В вышеприведенном примере мы случайно указали точку с запятой в конце оператора if.

Эта неосмотрительность приведет к тому, что код будет выполняться следующим образом:

```
1. if (a == 0)
2.     ; // точка с запятой указывает, что это нулевой стейтмент
3. a = 1; // и эта строка выполнится в любом случае!
```

Предупреждение: Всегда проверяйте, не "закрыли" ли вы случайно оператор if точкой с запятой.

Урок №68. Оператор switch

Хоть мы и можем использовать сразу несколько операторов if/else вместе — читается и смотрится это не очень. Например:

```
#include <iostream>

enum Colors
{
    COLOR_GRAY,
    COLOR_PINK,
    COLOR_BLUE,
    COLOR_PURPLE,
    COLOR_RED
};

void printColor(Colors color)
{
    if (color == COLOR_GRAY)
        std::cout << "Gray";
    else if (color == COLOR_PINK)
        std::cout << "Pink";
    else if (color == COLOR_BLUE)
        std::cout << "Blue";
    else if (color == COLOR_PURPLE)
        std::cout << "Purple";
    else if (color == COLOR_RED)
        std::cout << "Red";
    else
        std::cout << "Unknown";
}

int main()
{
    printColor(COLOR_BLUE);

    return 0;
}
```

Использование ветвления if/else для проверки значения одной переменной — практика распространенная, но язык C++ предоставляет альтернативный и более эффективный **условный оператор ветвления switch**. Вот вышеприведенная программа, но уже с использованием оператора switch:

```
#include <iostream>

enum Colors {
    COLOR_GRAY,
    COLOR_PINK,
    COLOR_BLUE,
    COLOR_PURPLE,
    COLOR_RED
};

void printColor(Colors color)
{
```

```
13.     switch (color)
14.     {
15.     case COLOR_GRAY:
16.         std::cout << "Gray";
17.         break;
18.     case COLOR_PINK:
19.         std::cout << "Pink";
20.         break;
21.     case COLOR_BLUE:
22.         std::cout << "Blue";
23.         break;
24.     case COLOR_PURPLE:
25.         std::cout << "Purple";
26.         break;
27.     case COLOR_RED:
28.         std::cout << "Red";
29.         break;
30.     default:
31.         std::cout << "Unknown";
32.         break;
33.     }
34. }
35.
36. int main()
37. {
38.     printColor(COLOR_BLUE);
39.     return 0;
40. }
```

Общая идея операторов switch проста: выражение оператора switch (например, `switch(color)`) должно производить значение, а каждый **кейс** (англ. ***"case"***) проверяет это значение на соответствие. Если кейс совпадает с выражением switch, то выполняются инструкции под соответствующим кейсом. Если ни один кейс не соответствует выражению switch, то выполняются инструкции после кейса `default` (если он вообще указан).

Из-за своей реализации, операторы switch обычно более эффективны, чем цепочки if/else. Давайте рассмотрим это более подробно.

Оператор switch

Сначала пишем **ключевое слово switch** за которым следует выражение, с которым мы хотим работать. Обычно это выражение представляет собой только одну переменную, но это может быть и нечто более сложное, например, `nX + 2` или `nX - nY`. Единственное ограничение к этому выражению - оно должно быть интегрального типа данных (т.е. типа char, short, int, long, long long или enum). Переменные типа с плавающей точкой или неинтегральные типы использоваться не могут.

После выражения switch мы объявляем блок. Внутри блока мы используем **лейблы** (англ. ***"labels"***) для определения всех значений, которые мы хотим проверять на соответствие выражению. Существуют два типа лейблов.

Лейблы case

Первый вид лейбла - это **case** (или просто ***"кейс"***), который объявляется с использованием **ключевого слова case** и имеет константное выражение. Константное выражение - это то, которое генерирует константное значение, другими словами: либо литерал (например, `5`), либо перечисление (например, `COLOR_RED`), либо константу (например, переменную `x`, которая была объявлена с ключевым словом const).

Константное выражение, находящееся после ключевого слова case, проверяется на равенство с выражением, находящимся возле ключевого слова switch. Если они совпадают, то тогда выполняется код под соответствующим кейсом.

Стоит отметить, что все выражения case должны производить уникальные значения. То есть вы не сможете сделать следующее:

```
switch (z)
{
    case 3:
    case 3: // нельзя, значение 3 уже используется!
    case COLOR_PURPLE: // нельзя, COLOR_PURPLE вычисляется как 3!
};
```

Можно использовать сразу несколько кейсов для одного выражения. Следующая функция использует несколько кейсов для проверки, соответствует ли параметр `p` числу из ASCII-таблицы:

```
bool isDigit(char p)
{
    switch (p)
    {
        case '0': // если p = 0
        case '1': // если p = 1
        case '2': // если p = 2
        case '3': // если p = 3
        case '4': // если p = 4
        case '5': // если p = 5
        case '6': // если p = 6
        case '7': // если p = 7
        case '8': // если p = 8
        case '9': // если p = 9
            return true; // возвращаем true
        default: // в противном случае, возвращаем false
            return false;
    }
}
```

В случае, если `p` является числом из ASCII-таблицы, то выполнится первый стейтмент после кейса — `return true;`.

Лейбл по умолчанию

Второй тип лейбла - это **лейбл по умолчанию** (так называемый ***"default case"***), который объявляется с использованием **ключевого слова default**. Код под этим лейблом выполняется, если ни один из кейсов не соответствует выражению switch. Лейбл по умолчанию является необязательным. В одном switch может быть только один default. Обычно его объявляют самым последним в блоке switch.

В вышеприведенном примере, если `p` не является числом из ASCII-таблицы, то тогда выполняется лейбл по умолчанию и возвращается false.

switch и fall-through

Одна из самых каверзных вещей в switch - это последовательность выполнения кода. Когда кейс совпал (или выполняется default), то выполнение начинается с первого стейтмента, который находится после соответствующего кейса и продолжается до тех пор, пока не будет выполнено одно из следующих условий завершения:

- Достигнут конец блока switch.
- Выполняется оператор return.
- Выполняется оператор goto.
- Выполняется оператор break.

Обратите внимание, если ни одного из этих условий завершения не будет, то выполняться будут все кейсы после того кейса, который совпал с выражением switch. Например:

```
1.  switch (2)
2.  {
3.      case 1: // Не совпадает!
4.          std::cout << 1 << '\n'; // пропускается
5.      case 2: // Совпало!
6.          std::cout << 2 << '\n'; // выполнение кода начинается здесь
7.      case 3:
8.          std::cout << 3 << '\n'; // это также выполнится
9.      case 4:
10.         std::cout << 4 << '\n'; // и это
11.     default:
12.         std::cout << 5 << '\n'; // и это
13. }
```

Результат:

```
2
3
4
5
```

А это точно не то, что нам нужно! Когда выполнение переходит из одного кейса в следующий, то это называется **fall-through**. Программисты почти никогда не используют fall-through, поэтому в редких случаях, когда это все-таки используется - программист оставляет комментарий, в котором сообщает, что fall-through является преднамеренным.

switch и оператор break

Оператор break (объявленный с использованием **ключевого слова break**) сообщает компилятору, что мы уже сделали всё, что хотели с определенным switch-ом (или циклом while, do while или for) и больше не намерены с ним работать. Когда компилятор встречает оператор break, то выполнение кода переходит из switch на следующую строку после блока switch. Рассмотрим вышеприведенный пример, но уже с корректно вставленными операторами break:

```
1. switch (2)
2. {
3.     case 1: // не совпадает - пропускается
4.         std::cout << 1 << '\n';
5.         break;
6.     case 2: // совпало!  Выполнение начинается со следующего стейтмента
7.         std::cout << 2 << '\n'; // выполнение начинается здесь
8.         break; // оператор break завершает выполнение switch-a
9.     case 3:
10.        std::cout << 3 << '\n';
11.        break;
12.    case 4:
13.        std::cout << 4 << '\n';
14.        break;
15.    default:
16.        std::cout << 5 << '\n';
17.        break;
18.}
19.// Выполнение продолжается здесь
```

Поскольку второй кейс соответствует выражению switch, то выводится `2`, и оператор break завершает выполение блока switch. Остальные кейсы пропускаются.

> ***Предупреждение:*** Не забывайте использовать оператор break в конце каждого кейса. Его отсутствие - одна из наиболее распространенных ошибок новичков!

Несколько стейтментов внутри блока switch

Еще одна странность в switch заключается в том, что вы можете использовать несколько стейтментов под каждым кейсом, не определяя новый блок:

```
switch (3)
{
    case 3:
        std::cout << 3;
        boo();
        std::cout << 4;
        break;
    default:
        std::cout << "default case\n";
        break;
}
```

Объявление переменной и её инициализация внутри case

Вы можете объявлять, но не инициализировать переменные внутри блока case:

```
switch (x)
{
    case 1:
        int z; // ок, объявление разрешено
        z = 5; // ок, операция присваивания разрешена
        break;

    case 2:
        z = 6; // ок, переменная z была объявлена выше, поэтому мы можем использовать её здесь
        break;

    case 3:
        int c = 4; // нельзя, вы не можете инициализировать переменные внутри case
        break;

    default:
        std::cout << "default case" << std::endl;
        break;
}
```

Обратите внимание, что, хотя переменная `z` была определена в первом кейсе, она также используется и во втором кейсе. Все кейсы считаются частью одной и той же области видимости, поэтому, объявив переменную в одном кейсе, мы можем спокойно использовать её без объявления и в других кейсах.

Это может показаться немного нелогичным, поэтому давайте рассмотрим это детально. Когда мы определяем локальную переменную, например, `int y;`, то переменная не создается в этой точке - она фактически создается в начале блока, в котором объявлена. Однако, она не видна в программе до точки объявления. Само

объявление не выполняется, оно просто сообщает компилятору, что переменная уже может использоваться в коде. Поэтому переменная, объявленная в одном кейсе, может использоваться в другом кейсе, даже если кейс, объявляющий переменную, никогда не выполняется.

Однако инициализация переменных непосредственно в кейсах запрещена и вызовет ошибку компиляции. Это связано с тем, что инициализация переменной *требует выполнения*, а кейс, содержащий инициализацию, может никогда не выполниться!

Если в кейсе нужно объявить и/или инициализировать новую переменную, то это лучше всего сделать, используя блок стейтментов внутри кейса:

```
1. switch (1)
2. {
3.     case 1:
4.     { // обратите внимание, здесь указан блок
5.         int z = 5; // хорошо, переменные можно инициализировать внутри блока, который находится внутри кейса
6.         std::cout << z;
7.         break;
8.     }
9.     default:
10.        std::cout << "default case" << std::endl;
11.        break;
12.}
```

Правило: Если нужно инициализировать и/или объявить переменные внутри кейса — используйте блоки стейтментов.

Тест

Задание №1

Напишите функцию calculate(), которая принимает две переменные типа int и одну переменную типа char, которая, в свою очередь, представляет одну из следующих математических операций: `+`, `-`, `*`, `/` или `%` (остаток от числа). Используйте switch для выполнения соответствующей математической операции над целыми числами, а результат возвращайте обратно в main(). Если в функцию передается недействительный математический оператор, то функция должна выводить ошибку. С оператором деления выполняйте целочисленное деление.

Задание №2

Определите перечисление (или класс enum) `Animal`, которое содержит следующих животных: `pig`, `chicken`, `goat`, `cat`, `dog` и `ostrich`. Напишите функцию

getAnimalName(), которая принимает параметр `Animal` и использует switch для возврата типа животного в качестве строки. Напишите еще одну функцию - printNumberOfLegs(), которая использует switch для вывода количества лап соответствующего типа животного. Убедитесь, что обе функции имеют кейс default, который выводит сообщение об ошибке. Вызовите printNumberOfLegs() в main(), используя в качестве параметров `cat` и `chicken`.

Пример результата выполнения вашей программы:

```
A cat has 4 legs.
A chicken has 2 legs.
```

Урок №69. Оператор goto

Оператор goto – это оператор управления потоком выполнения программ, который заставляет центральный процессор выполнить переход из одного участка кода в другой (осуществить прыжок). Другой участок кода идентифицируется с помощью **лейбла**. Например:

```
#include <iostream>
#include <cmath> // для функции sqrt()

int main()
{
    double z;
tryAgain: // это лейбл
    std::cout << "Enter a non-negative number: ";
    std::cin >> z;

    if (z < 0.0)
        goto tryAgain; // а это оператор goto

    std::cout << "The sqrt of " << z << " is " << sqrt(z) << std::endl;
    return 0;
}
```

В этой программе пользователю предлагается ввести неотрицательное число. Однако, если пользователь введет отрицательное число, программа, используя оператор goto, выполнит переход обратно к лейблу `tryAgain`. Затем пользователю снова нужно будет ввести число. Таким образом, мы можем постоянно запрашивать у пользователя ввод числа, пока он не введет корректное число.

Ранее мы рассматривали два типа области видимости: локальная (или *"блочная"*) и глобальная (или *"файловая"*). Лейблы используют третий тип области видимости: **область видимости функции**. Оператор goto и соответствующий лейбл должны находиться в одной и той же функции.

Существуют некоторые ограничения на использование операторов goto. Например, вы не сможете перепрыгнуть вперед через переменную, которая инициализирована в том же блоке, что и goto:

```
int main()
{
    goto skip; // прыжок вперед недопустим
    int z = 7;
skip: // лейбл
    z += 4; // какое значение будет в этой переменной?
    return 0;
}
```

В целом, программисты избегают использования оператора goto в языке C++ (и в большинстве других высокоуровневых языков программирования). Основная проблема с ним заключается в том, что он позволяет программисту управлять выполнением кода так, что точка выполнения может прыгать по коду произвольно. А это, в свою очередь, создает то, что опытные программисты называют «спагетти-кодом». **Спагетти-код** - это код, порядок выполнения которого напоминает тарелку со спагетти (всё запутано и закручено), что крайне затрудняет следование порядку и понимание логики выполнения такого кода.

Как говорил один известный специалист в информатике и программировании, Эдсгер Дейкстра: «Качество программистов - это уменьшающаяся функция плотности использования операторов goto в программах, которые они пишут».

Оператор goto часто используется в некоторых старых языках, таких как Basic или Fortran, или даже в языке Си. Однако в C++ goto почти никогда не используется, поскольку любой код, написанный с ним, можно более эффективно переписать с использованием других объектов в языке C++, таких как циклы, обработчики исключений или деструкторы (всё перечисленное мы рассмотрим чуть позже).

Правило: Избегайте использования операторов goto, если на это нет веских причин.

Урок №70. Цикл while

Цикл while является самым простым из 4-х циклов, которые есть в языке C++. Он очень похож на ветвление if/else:

```
while (условие)
    тело цикла;
```

Цикл while объявляется с использованием **ключевого слова while**. В начале цикла обрабатывается `условие`. Если его значением является true (любое ненулевое значение), то тогда выполняется `тело цикла`.

Однако, в отличие от оператора if, после завершения выполнения `тела цикла`, управление возвращается обратно к while и процесс проверки условия повторяется. Если условие опять является true, то тогда `тело цикла` выполняется еще раз.

Например, следующая программа выводит все числа от 0 до 9:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int count = 0;
6.     while (count < 10)
7.         {
8.         std::cout << count << " ";
9.         ++count;
10.        }
11.    std::cout << "done!";
12.
13.    return 0;
14. }
```

Результат выполнения программы:

```
0 1 2 3 4 5 6 7 8 9 done!
```

Рассмотрим детально эту программу. Во-первых, инициализируется переменная: `int count = 0;`. Условие `0 < 10` имеет значение true, поэтому выполняется тело цикла. В первом стейтменте мы выводим `0`, а во втором — выполняем инкремент переменной `count`. Затем управление возвращается к началу цикла while для повторной проверки условия. Условие `1 < 10` имеет значение true, поэтому тело цикла выполняется еще раз. Тело цикла будет повторно выполняться до тех пор, пока переменная `count` не будет равна `10`, только в том случае, когда результат условия `10 < 10` будет false, цикл завершится.

Тело цикла while может и вообще не выполняться, например:

```
#include <iostream>

int main()
{
    int count = 15;
    while (count < 10)
        {
        std::cout << count << " ";
        ++count;
        }
    std::cout << "done!";

    return 0;
}
```

Условие `15 < 10` сразу принимает значение false, и тело цикла пропускается. Единственное, что выведет эта программа:

```
done!
```

Бесконечные циклы

С другой стороны, если условие цикла всегда принимает значение true, то и сам цикл будет выполняться бесконечно. Это называется **бесконечным циклом**. Например:

```
#include <iostream>

int main()
{
    int count = 0;
    while (count < 10) // это условие никогда не будет false
        std::cout << count << " "; // поэтому эта строка будет выполняться постоянно

    return 0; // а эта строка никогда не выполнится
}
```

Поскольку переменная `count` не увеличивается на единицу в этой программе, то условие `count < 10` всегда будет true. Следовательно, цикл никогда не будет завершен, и программа будет постоянно выводить `0 0 0 0 0...`.

Мы можем преднамеренно объявить бесконечный цикл следующим образом:

```
while (1) // или while (true)
{
  // Этот цикл будет выполняться бесконечно
}
```

Единственный способ выйти из бесконечного цикла — использовать операторы return, break, goto, выбросить исключение или воспользоваться функцией exit().

Программы, которые работают до тех пор, пока пользователь не решит остановить их, иногда преднамеренно используют бесконечные циклы вместе с операторами return, break или функцией exit() для завершения цикла. Распространена такая практика в серверных веб-приложениях, которые работают непрерывно и постоянно обслуживают веб-запросы.

Счетчик цикла while

Часто нам нужно будет, чтобы цикл выполнялся определенное количество раз. Для этого обычно используется переменная в виде счетчика цикла. **Счетчик цикла** - это целочисленная переменная, которая объявляется с единственной целью: считать, сколько раз выполнился цикл. В вышеприведенных примерах переменная `count` является счетчиком цикла.

Счетчикам цикла часто дают простые имена, такие как `i`, `j` или `k`. Однако в этих именах есть одна серьезная проблема. Если вы захотите узнать, где в вашей программе используется счетчик цикла и воспользуетесь функцией поиска символов `i`, `j` или `k`, то в результате получите половину своей программы, так как `i`, `j` или `k` используются во многих именах. Следовательно, лучше использовать `iii`, `jjj` или `kkk` в качестве имен для счетчиков. Они более уникальны, их значительно проще найти, и они выделяются в коде. А еще лучше использовать «реальные» имена для переменных, например, `count` или любое другое имя, которое предоставляет контекст использования этой переменной.

Также для счетчиков цикла лучше использовать тип signed int. Использование unsigned int может привести к неожиданным результатам. Например:

```
#include <iostream>

int main()
{
    unsigned int count = 10;

    // Считаем от 10 к 0
    while (count >= 0)
    {
        if (count == 0)
            std::cout << "blastoff!";
        else
            std::cout << count << " ";
        --count;
    }

    return 0;
```

```
18. }
```

Взгляните на эту программу еще раз и постарайтесь найти ошибку.

Оказывается, эта программа представляет собой бесконечный цикл. Она начинается с вывода `10 9 8 7 6 5 4 3 2 1 blastoff!`, как и предполагалось, но затем «сходит с рельсов» и начинает отсчет с `4294967295`. Почему? Потому что условие цикла `count >= 0` никогда не будет ложным! Когда `count = 0`, то и условие `0 >= 0` имеет значение true, выводится `blastoff`, а затем выполняется декремент переменной `count`, происходит переполнение и значением переменной становится `4294967295`. И так как условие `4294967295 >= 0` является истинным, то программа продолжает свое выполнение. А поскольку счетчик цикла является типа unsigned, то он никогда не сможет быть отрицательным, а так как он никогда не сможет быть отрицательным, то цикл никогда не завершится.

Правило: Всегда используйте тип signed int для счетчиков цикла.

Итерации

Каждое выполнение цикла называется **итерацией** (или ***"повтором"***).

Поскольку тело цикла обычно является блоком, и поскольку этот блок выполняется по новой с каждым повтором, то любые переменные, объявленные внутри тела цикла, создаются, а затем и уничтожаются по новой. В следующем примере переменная `z` создается и уничтожается 6 раз:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int count = 1;
6.     int result = 0; // переменная result определена здесь, поскольку она нам
   понадобится позже (вне тела цикла)
7.
8.     while (count <= 6) // итераций будет 6
9.     {
10.        int z; // z создается здесь по новой с каждой итерацией
11.
12.        std::cout << "Enter integer #" << count << ':';
13.        std::cin >> z;
14.
15.        result += z;
16.
17.        // Увеличиваем значение счетчика цикла на единицу
18.        ++count;
19.    } // z уничтожается здесь по новой с каждой итерацией
20.
21.    std::cout << "The sum of all numbers entered is: " << result;
22.
23.    return 0;
```

```
}
```

Для фундаментальных типов переменных это нормально. Для не фундаментальных типов переменных (таких как структуры или классы) это может сказаться на производительности. Следовательно, не фундаментальные типы переменных лучше определять перед циклом.

Обратите внимание, переменная `count` объявлена вне тела цикла. Это важно и необходимо, поскольку нам нужно, чтобы значение переменной сохранялось на протяжении всех итераций (не уничтожалось по новой с каждым повтором цикла).

Иногда нам может понадобиться выполнить что-то при достижении определенного количества итераций, например, вставить символ новой строки. Это легко осуществить, используя оператор остатка от деления со счетчиком цикла:

```
#include <iostream>

int main()
{
    int count = 1;
    while (count <= 50)
    {
        // Выводим числа до 10 (перед каждым числом добавляем 0)
        if (count < 10)
            std::cout << "0" << count << " ";
        else
            std::cout << count << " "; // выводим остальные числа

        // Если счетчик цикла делится на 10 без остатка, то тогда вставляем символ новой строки
        if (count % 10 == 0)
            std::cout << "\n";

        // Увеличиваем значение счетчика цикла на единицу
        ++count;
    }

    return 0;
}
```

Результат выполнения программы:

```
01 02 03 04 05 06 07 08 09 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

Вложенные циклы while

Также одни циклы while могут быть вложены внутри других циклов while. В следующем примере внутренний и внешний циклы имеют свои собственные счетчики. Однако, обратите внимание, условие внутреннего цикла использует счетчик внешнего цикла!

```
#include <iostream>

int main()
{
    int outer = 1;
    while (outer <= 5)
    {
        int inner = 1;
        while (inner <= outer)
            std::cout << inner++ << " ";

        // Вставляем символ новой строки в конце каждого ряда
        std::cout << "\n";
        ++outer;
    }

    return 0;
}
```

Результат выполнения программы:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Тест

Задание №1

Почему в программе, приведенной выше, переменная `inner` объявлена внутри блока while, а не сразу после объявления переменной `outer` (вне блока while)?

Задание №2

Напишите программу, которая выводит буквы английского алфавита от `a` до `z` вместе с кодами из **ASCII-таблицы**.

> ***Подсказка:*** Чтобы выводить символы как целые числа — используйте оператор static_cast.

Задание №3

Измените программу из последнего подраздела «Вложенные циклы» так, чтобы она выводила следующее:

```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

Задание №4

Теперь сделайте так, чтобы цифры выводились следующим образом (используя программу из предыдущего задания):

```
        1
      2 1
    3 2 1
  4 3 2 1
5 4 3 2 1
```

> ***Подсказка:*** Разберитесь сначала, как вывести числа следующим образом:
>
> ```
> X X X X 1
> X X X 2 1
> X X 3 2 1
> X 4 3 2 1
> 5 4 3 2 1
> ```

Урок №71. Цикл do while

Одна интересная вещь в цикле while заключается в том, что если условие цикла изначально равно false, то тело цикла не будет выполняться вообще. Но иногда бывают случаи, когда нужно, чтобы цикл выполнился хотя бы один раз, например, при отображении меню. Для решения этой проблемы C++ предоставляет цикл do while.

Синтаксис do while в языке C++:

```
do
    тело цикла;
while (условие);
```

Тело цикла do while всегда выполняется хотя бы один раз. После выполнения тела цикла проверяется условие. Если оно истинно, то выполнение переходит к началу блока do и тело цикла выполняется снова.

Ниже приведен пример использования цикла do while для отображения меню:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     // Переменная choice должна быть объявлена вне цикла do while
6.     int choice;
7.
8.     do
9.     {
10.        std::cout << "Please make a selection: \n";
11.        std::cout << "1) Addition\n";
12.        std::cout << "2) Subtraction\n";
13.        std::cout << "3) Multiplication\n";
14.        std::cout << "4) Division\n";
15.        std::cin >> choice;
16.    }
17.    while (choice != 1 && choice != 2 &&
18.        choice != 3 && choice != 4);
19.
20.    // Что-
   то делаем с переменной choice, например, используем оператор switch
21.
22.    std::cout << "You selected option #" << choice << "\n";
23.
24.    return 0;
25.}
```

Интересно, что переменная `choice` должна быть объявлена вне блоков do while. Почему так?

Если бы переменная `choice` была объявлена внутри блока do, то она была бы уничтожена при завершении этого блока еще до проверки условия while. Но нам нужна переменная, которая будет использоваться в условии while, следовательно, переменная `choice` должна быть объявлена вне блока do.

В целом, использовать do while вместо while, когда нужно, чтобы цикл выполнился хотя бы один раз, является хорошей практикой.

Урок №72. Цикл for

Безусловно, наиболее используемым циклом в языке C++ является цикл for.

Цикл for

Цикл for в языке C++ идеален, когда известно необходимое количество итераций. Выглядит он следующим образом:

```
for (объявление переменных; условие; инкремент/декремент счетчика)
    тело цикла;
```

Или, преобразуя for в эквивалентный цикл while:

```
{ // обратите внимание, цикл находится в блоке
    объявление переменных;
    while (условие)
    {
        тело цикла;
        инкремент/декремент счетчика;
    }
} // переменные, объявленные внутри цикла, выходят из области видимости здесь
```

Переменные, определенные внутри цикла for, имеют специальный тип области видимости: **область видимости цикла**. Такие переменные существуют только внутри цикла и недоступны за его пределами.

Выполнение цикла for

Цикл for в C++ выполняется в 3 шага:

- ***Шаг №1:*** **Объявление переменных**. Как правило, здесь выполняется определение и инициализация счетчиков цикла, а точнее — одного счетчика цикла. Эта часть выполняется только один раз, когда цикл выполняется впервые.
- ***Шаг №2:*** **Условие**. Если оно равно false, то цикл немедленно завершает свое выполнение. Если же условие равно true, то выполняется тело цикла.
- ***Шаг №3:*** **Инкремент/декремент счетчика цикла**. Переменная увеличивается или уменьшается на единицу. После этого цикл возвращается к шагу №2.

Рассмотрим пример цикла for и разберемся детально, как он работает:

```
#include <iostream>

int main()
{
    for (int count = 0; count < 10; ++count)
        std::cout << count << " ";

    return 0;
}
```

Сначала мы объявляем переменную `count` и присваиваем ей значение `0`. Далее проверяется условие `count < 10`, а так как `count` равен `0`, то условие `0 < 10` имеет значение true. Следовательно, выполняется тело цикла, в котором мы выводим в консоль переменную `count` (т.е. значение `0`).

Затем выполняется выражение `++count`, т.е. инкремент переменной. Затем цикл снова возвращается к проверке условия. Условие `1 < 10` имеет значение true, поэтому тело цикла выполняется опять. Выводится `1`, а переменная `count` увеличивается уже до значения `2`. Условие `2 < 10` является истинным, поэтому выводится `2`, а `count` увеличивается до `3` и так далее.

В конце концов, `count` увеличивается до `10`, а условие `10 < 10` является ложным, и цикл завершается. Следовательно, результат выполнения программы:

```
0 1 2 3 4 5 6 7 8 9
```

Циклы for могут быть несколько сложны для новичков, однако опытные кодеры любят их, так как эти циклы очень компактны и удобны. Для наглядности, давайте преобразуем цикл for, приведенный выше, в эквивалентный цикл while:

```
#include <iostream>

int main()
{
    { // внешние скобки нужны для обеспечения области видимости цикла
        int count = 0;
        while (count < 10)
        {
            std::cout << count << " ";
            ++count;
        }
    }

    return 0;
}
```

Обратите внимание, внешние фигурные скобки здесь необходимы, так как переменная `count` выходит из области видимости при завершении цикла.

Еще примеры циклов for

Давайте, используя цикл for, напишем функцию вычисления значений в степени `n`:

```
1. int pow(int base, int exponent)
2. {
3.     int total = 1;
4. 
5.     for (int count=0; count < exponent; ++count)
6.         total *= base;
7. 
8.     return total;
9. }
```

Функция возвращает значение `base^exponent` (число в степени `n`). `base` — это число, которое нужно возвести в степень, а `exponent` — это степень, в которую нужно возвести `base`.

- Если экспонент равен `0`, то цикл for выполняется 0 раз, и функция возвращает `1`.
- Если экспонент равен `1`, то цикл for выполняется 1 раз, и функция возвращает `1 * base`.
- Если экспонент равен `2`, то цикл for выполняется 2 раза, и функция возвращает `1 * base * base`.

Хотя в большинстве циклов используется инкремент счетчика, мы также можем использовать и декремент счетчика:

```
1. #include <iostream>
2. 
3. int main()
4. {
5.     for (int count = 8; count >= 0; --count)
6.         std::cout << count << " ";
7. 
8.     return 0;
9. }
```

Результат:

```
8 7 6 5 4 3 2 1 0
```

Также с каждой новой итерацией мы можем увеличить или уменьшить значение счетчика больше, чем на единицу:

```
1. #include <iostream>
2. 
3. int main()
4. {
5.     for (int count = 9; count >= 0; count -= 2)
```

```
6.          std::cout << count << " ";
7.
8.      return 0;
9.  }
```

Результат:

```
9 7 5 3 1
```

Ошибка неучтенной единицы

Одна из самых больших проблем с которой приходится сталкиваться начинающим программистам в циклах for (а также и в других типах циклов) - это **ошибка на единицу** (или ***"ошибка неучтенной единицы"***). Она возникает, когда цикл повторяется на 1 раз больше или на 1 раз меньше нужного количества итераций. Это обычно происходит из-за того, что в условии используется некорректный оператор сравнения (например, `>` вместо `>=` или наоборот). Как правило, эти ошибки трудно отследить, так как компилятор не будет жаловаться на них, программа будет работать, но её результаты будут неправильными.

При написании циклов for помните, что цикл будет выполняться до тех пор, пока условие является истинным. Рекомендуется тестировать циклы, используя разные значения для проверки работоспособности цикла. Хорошей практикой является проверять циклы с помощью данных ввода (чисел, символов и прочего), которые заставляют цикл выполниться 0, 1 и 2 раза. Если цикл работает исправно, значит всё ОК.

Правило: Тестируйте свои циклы, используя входные данные, которые заставляют цикл выполниться 0, 1 и 2 раза.

Пропущенные выражения в цикле

Также в циклах можно пропускать одно или сразу все выражения, например:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      int count = 0;
6.      for (; count < 10; )
7.      {
8.          std::cout << count << " ";
9.          ++count;
10.     }
11.
12.     return 0;
13. }
```

Результат:

```
0 1 2 3 4 5 6 7 8 9
```

Инициализацию счетчика мы прописали вне тела цикла, а инкремент счетчика — внутри тела цикла. В самом операторе for мы указали только условие. Иногда бывают случаи, когда не требуется объявлять счетчик цикла (потому что у нас он уже есть) или увеличивать его (так как мы увеличиваем его каким-то другим способом).

Хоть это и не часто можно наблюдать, но в операторе for можно вообще ничего не указывать. Стоит отметить, что подобное приведет к бесконечному циклу:

```
for (;;)
    тело цикла;
```

Вышеприведенный пример эквивалентен:

```
while (true)
    тело цикла;
```

Объявления переменных в цикле for

Хотя в циклах for обычно используется только один счетчик, иногда могут возникать ситуации, когда нужно работать сразу с несколькими переменными. Для этого используется оператор Запятая. Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int aaa, bbb;
6.     for (aaa = 0, bbb = 9; aaa < 10; ++aaa, --bbb)
7.         std::cout << aaa << " " << bbb << std::endl;
8.
9.     return 0;
10.}
```

Этот цикл присваивает значения двум ранее объявленным переменным: `aaa = 0` и `bbb = 9`. Только с каждой итерацией переменная `aaa` увеличивается на единицу, а `bbb` — уменьшается на единицу.

Результат выполнения программы:

```
0 9
1 8
2 7
```

```
3 6
4 5
5 4
6 3
7 2
8 1
9 0
```

> ***Примечание:*** Вышеприведенный цикл можно переписать следующим образом:
>
> ```
> #include <iostream>
>
> int main()
> {
>     for (int aaa = 0, bbb = 9; aaa < 10; ++aaa, --bbb)
>         std::cout << aaa << " " << bbb << std::endl;
>
>     return 0;
> }
> ```
>
> В этом случае запятая в объявлении переменных является частью синтаксиса, а не использованием оператора Запятая. Но эффект идентичен.

Использование циклов for в старых версиях C++

В старых версиях C++ переменные, объявленные в цикле for, не уничтожались при завершении этого цикла. Т.е. у вас могло получиться что-то вроде следующего:

```
1. for (int count=0; count < 10; ++count)
2.     std::cout << count << " ";
3.
4. // В старых версиях C++ переменная count здесь не уничтожается
5.
6. std::cout << "\n";
7. std::cout << "I counted to: " << count << "\n"; // поэтому мы можем
   использовать count даже здесь
```

Позднее это было запрещено, но вы все еще можете увидеть подобное в старом коде.

Вложенные циклы for

Подобно другим типам циклов, одни циклы for могут быть вложены в другие циклы for. В следующем примере мы разместили один for внутри другого for:

```
1. #include <iostream>
2.
```

```
3.  int main()
4.  {
5.      for (char c = 'a'; c <= 'e'; ++c) // внешний цикл по буквам
6.      {
7.          std::cout << c; // сначала выводим букву
8.
9.          for (int i = 0; i < 3; ++i) // внутренний цикл по числам
10.             std::cout << i;
11.
12.         std::cout << '\n';
13.     }
14.
15.     return 0;
16. }
```

С одной итерацией внешнего цикла выполняется три итерации внутреннего цикла. Следовательно, результат выполнения программы:

```
a012
b012
c012
d012
e012
```

Заключение

Циклы for являются наиболее часто используемыми циклами в языке C++. Несмотря на то, что их синтаксис, как правило, немного запутывает начинающих программистов, вы очень скоро к нему привыкните и ощутите всю мощь и удобство этих циклов.

Тест

Задание №1

Напишите цикл for, который выводит каждое четное число в диапазоне от 0 до 20.

Задание №2

Напишите функцию sumTo(), которая принимает целочисленный параметр с именем `value` и возвращает сумму всех чисел от `1` до значения `value`.

Например, если значением `value` является `5`, то `sumTo(5)` должен возвратить `15`, исходя из `1 + 2 + 3 + 4 + 5`.

> ***Подсказка:*** Используйте переменную вне тела цикла для хранения суммы чисел, как в примере с функцией pow() в подзаголовке «Еще примеры циклов for».

Задание №3

Что не так со следующим циклом?

```
1. // Выводим все числа от 8 до 0
2. for (unsigned int count=8; count >= 0; --count)
3.     cout << count << " ";
```

Урок №73. Операторы break и continue

Хотя вы уже видели оператор break в связке с оператором switch, все же он заслуживает большего внимания, поскольку может использоваться и с циклами. **Оператор break** приводит к завершению выполнения циклов do, for или while.

break и switch

В контексте оператора switch оператор break обычно используется в конце каждого кейса для его завершения (предотвращая fall-through):

```
switch (op)
{
    case '+':
        doAddition(a, b);
        break;
    case '-':
        doSubtraction(a, b);
        break;
    case '*':
        doMultiplication(a, b);
        break;
    case '/':
        doDivision(a, b);
        break;
}
```

break и циклы

В контексте циклов оператор break используется для завершения работы цикла раньше времени:

```
#include <iostream>

int main()
{
    int sum = 0;

    // Разрешаем пользователю ввести до 10 чисел
    for (int count=0; count < 10; ++count)
    {
        std::cout << "Enter a number to add, or 0 to exit: ";
        int val;
        std::cin >> val;

        // Выходим из цикла, если пользователь введет 0
        if (val == 0)
            break;

        // В противном случае, добавляем число к общей сумме
        sum += val;
    }

```

```
22.     std::cout << "The sum of all the numbers you entered is " << sum << "\n";
23.
24.     return 0;
25. }
```

Эта программа позволяет пользователю ввести до 10 чисел и в конце подсчитывает их сумму. Если пользователь введет `0`, то выполнится break и цикл завершится (не важно, сколько чисел в этот момент успел ввести пользователь).

Обратите внимание, оператор break может использоваться и для выхода из бесконечного цикла:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      while (true) // бесконечный цикл
6.      {
7.          std::cout << "Enter 0 to exit or anything else to continue: ";
8.          int val;
9.          std::cin >> val;
10.
11.         // Выходим из цикла, если пользователь ввел 0
12.         if (val == 0)
13.             break;
14.     }
15.
16.     std::cout << "We're out!\n";
17.
18.     return 0;
19. }
```

break и return

Новички часто путают или не понимают разницы между операторами break и return. Оператор break завершает работу switch или цикла, а выполнение кода продолжается с первого стейтмента, который находится сразу же после этого switch или цикла. Оператор return завершает выполнение всей функции, в которой находится цикл, а выполнение продолжается в точке после вызова функции:

```
1.  #include <iostream>
2.
3.  int breakOrReturn()
4.  {
5.      while (true) // бесконечный цикл
6.      {
7.          std::cout << "Enter 'b' to break or 'r' to return: ";
8.          char sm;
9.          std::cin >> sm;
10.
11.         if (sm == 'b')
12.             break; // выполнение кода продолжится с первого стейтмента после
    цикла
13.
```

```
14.         if (sm == 'r')
15.             return 1; // выполнение return приведет к тому, что управление
    сразу возвратится в caller (в этом случае, в функцию main())
16.     }
17.
18.     // Использование оператора break приведет к тому, что выполнение цикла
    продолжится здесь
19.
20.     std::cout << "We broke out of the loop\n";
21.
22.     return 0;
23. }
24.
25. int main()
26. {
27.     int returnValue = breakOrReturn();
28.     std::cout << "Function breakOrContinue returned " << returnValue << '\n';
29.
30.     return 0;
31. }
```

Оператор continue

Оператор continue позволяет сразу перейти в конец тела цикла, пропуская весь код, который находится под ним. Это полезно в тех случаях, когда мы хотим завершить текущую итерацию раньше времени. Например:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      for (int count = 0; count < 20; ++count)
6.      {
7.          // Если число делится нацело на 4, то пропускаем весь код в этой
    итерации после continue
8.          if ((count % 4) == 0)
9.              continue; // пропускаем всё и переходим в конец тела цикла
10.
11.         // Если число не делится нацело на 4, то выполнение кода продолжается
12.         std::cout << count << std::endl;
13.
14.         // Точка выполнения после оператора continue перемещается сюда
15.     }
16.
17.     return 0;
18. }
```

Эта программа выведет все числа от 0 до 19, которые не делятся нацело на 4.

В случае с циклом for часть инкремента/декремента счетчика по-прежнему выполняется даже после выполнения continue (так как инкремент/декремент происходит вне тела цикла).

Будьте осторожны при использовании оператора continue с циклами while или do while. Поскольку в этих циклах инкремент счетчиков выполняется непосредственно

в теле цикла, то использование continue может привести к тому, что цикл станет бесконечным! Например:

```
#include <iostream>

int main()
{
    int count(0);
    while (count < 10)
    {
        if (count == 5)
            continue; // переходим в конец тела цикла
        std::cout << count << " ";
        ++count;

        // Точка выполнения после оператора continue перемещается сюда
    }

    return 0;
}
```

Предполагается, что программа выведет все числа от 0 до 9, за исключением 5. Но на самом деле:

```
0 1 2 3 4
```

А затем цикл станет бесконечным. Когда значением `count` становится `5`, то условие оператора if станет true, затем выполнится continue и мы, минуя вывод числа и инкремент счетчика, перейдем к следующей итерации. Переменная `count` так и не увеличится. Как результат, в следующей итерации переменная `count` по-прежнему останется со значением `5`, а оператор if по-прежнему останется true, и цикл станет бесконечным.

А вот правильное решение, но с использованием цикла do while:

```
#include <iostream>

int main()
{
    int count(0);
    do
    {
        if (count == 5)
            continue; // переходим в конец тела цикла
        std::cout << count << " ";

        // Точка выполнения после оператора continue перемещается сюда
    } while (++count < 10); // этот код выполняется, так как он находится вне тела цикла

    return 0;
}
```

Результат выполнения программы:

```
0 1 2 3 4 6 7 8 9
```

break и continue

Многие учебники рекомендуют не использовать операторы break и continue, поскольку они приводят к произвольному перемещению точки выполнения программы по всему коду, что усложняет понимание и следование логике выполнения такого кода.

Тем не менее, разумное использование операторов break и continue может улучшить читабельность циклов в программе, уменьшив при этом количество вложенных блоков и необходимость наличия сложной логики выполнения циклов. Например, рассмотрим следующую программу:

```
#include <iostream>

int main()
{
    int count(0); // считаем количество итераций цикла
    bool exitLoop(false); // контролируем завершение выполнения цикла
    while (!exitLoop)
    {
        std::cout << "Enter 'e' to exit this loop or any other key to continue: ";
        char sm;
        std::cin >> sm;

        if (sm == 'e')
            exitLoop = true;
        else
        {
            ++count;
            std::cout << "We've iterated " << count << " times\n";
        }
    }

    return 0;
}
```

Эта программа использует логическую переменную для выхода из цикла, а также вложенный блок, который запускается только в том случае, если пользователь не использует символ выхода.

А вот более читабельная версия, но с использованием оператора break:

```
#include <iostream>

int main()
{
    int count(0); // считаем количество итераций цикла
```

```
6.      while (true) // выполнение цикла продолжается, если его не завершит
    пользователь
7.      {
8.          std::cout << "Enter 'e' to exit this loop or any other key to continue:
     ";
9.          char sm;
10.         std::cin >> sm;
11.
12.         if (sm == 'e')
13.             break;
14.
15.         ++count;
16.         std::cout << "We've iterated " << count << " times\n";
17.     }
18.
19.     return 0;
20. }
```

Здесь (с одним оператором break) мы избежали использования как логической переменной (а также понимания того, зачем она и где используется), так и оператора else с вложенным блоком.

Уменьшение количества используемых переменных и вложенных блоков улучшают читабельность и понимание кода намного больше, чем операторы break или continue могут нанести вред. По этой причине считается приемлемым их разумное использование.

Урок №74. Генерация случайных чисел

Возможность генерировать случайные числа очень полезна в некоторых видах программ, в частности, в играх, программах научного или статистического моделирования. Возьмем, к примеру, игры без **рандомных** (или ***"случайных"***) **событий** - монстры всегда будут атаковать вас одинаково, вы всегда будете находить одни и те же предметы/артефакты, макеты темниц и подземелий никогда не будут меняться и т.д. В общем, сюжет такой игры не очень интересен и вряд ли вы будете в нее долго играть.

Генератор псевдослучайных чисел

Так как же генерировать случайные числа? В реальной жизни мы часто бросаем монетку (орел/решка), кости или перетасовываем карты. Эти события включают в себя так много физических переменных (например, сила тяжести, трение, сопротивление воздуха и т.д.), что они становятся почти невозможными для прогнозирования/контроля и выдают результаты, которые во всех смыслах являются случайными.

Однако компьютеры не предназначены для использования физических переменных — они не могут подбросить монетку, бросить кости или перетасовать реальные карты. Компьютеры живут в контролируемом электрическом мире, где есть только два значения (true и false), чего-то среднего между ними нет. По своей природе компьютеры предназначены для получения прогнозируемых результатов. Когда мы говорим компьютеру посчитать, сколько будет `2 + 2`, мы *всегда* хотим, чтобы ответом было `4` (не `3` и не `5`).

Следовательно, компьютеры неспособны генерировать случайные числа. Вместо этого они могут имитировать случайность, что достигается с помощью генераторов псевдослучайных чисел.

Генератор псевдослучайных чисел (сокр. ***"ГПСЧ"***) — это программа, которая принимает стартовое/начальное значение и выполняет с ним определенные математические операции, чтобы конвертировать его в другое число, которое совсем не связано со стартовым. Затем программа использует новое сгенерированное значение и выполняет с ним те же математические операции, что и с начальным числом, чтобы конвертировать его в еще одно новое число - третье, которое не связано ни с первым, ни со вторым. Применяя этот алгоритм к последнему сгенерированному значению, программа может генерировать целый

ряд новых чисел, которые будут казаться случайными (при условии, что алгоритм будет достаточно сложным).

На самом деле, написать простой ГПСЧ не так уж и сложно. Вот небольшая программа, которая генерирует 100 рандомных чисел:

```
#include <iostream>

unsigned int PRNG()
{
    // Наше стартовое число - 4 541
    static unsigned int seed = 4541;

    // Берем стартовое число и, с его помощью, генерируем новое значение.
    // Из-за использования очень больших чисел (и переполнения) угадать следующее число исходя из предыдущего - очень сложно
    seed = (8253729 * seed + 2396403);

    // Берем стартовое число и возвращаем значение в диапазоне от 0 до 32767
    return seed % 32768;
}

int main()
{
    // Выводим 100 случайных чисел
    for (int count=0; count < 100; ++count)
    {
        std::cout << PRNG() << "\t";

        // Если вывели 5 чисел, то вставляем символ новой строки
        if ((count+1) % 5 == 0)
            std::cout << "\n";
    }
}
```

Результат выполнения программы:

```
18256   4675    32406   6217    27484
975     28066   13525   25960   2907
12974   26465   13684   10471   19898
12269   23424   23667   16070   3705
22412   9727    1490    773     10648
1419    8926    3473    20900   31511
5610    11805   20400   1699    24310
25769   9148    10287   32258   12597
19912   24507   29454   5057    19924
11591   15898   3149    9184    4307
24358   6873    20460   2655    22066
16229   20984   6635    9022    31217
10756   16247   17994   19069   22544
31491   16214   12553   23580   19599
3682    11669   13864   13339   13166
```

```
16417   26164   12711   11898   26797
27712   17715   32646   10041   18508
28351   9874    31685   31320   11851
9118    26193   612     983     30378
26333   24688   28515   8118    32105
```

Каждое число кажется случайным по отношению к предыдущему. Главный недостаток этого алгоритма — его примитивность.

Функции srand() и rand()

Языки Си и C++ имеют свои собственные встроенные генераторы случайных чисел. Они реализованы в 2-х отдельных функциях, которые находятся в заголовочном файле cstdlib:

- **Функция srand()** устанавливает передаваемое пользователем значение в качестве стартового. srand() следует вызывать только один раз — в начале программы (обычно в верхней части функции main()).
- **Функция rand()** генерирует следующее случайное число в последовательности. Оно будет находиться в диапазоне от `0` до `RAND_MAX` (константа в cstdlib, значением которой является `32767`).

Вот пример программы, в которой используются обе эти функции:

```
1. #include <iostream>
2. #include <cstdlib> // для функций rand() и srand()
3.
4. int main()
5. {
6.     srand(4541); // устанавливаем стартовое значение - 4 541
7.
8.     // Выводим 100 случайных чисел
9.     for (int count=0; count < 100; ++count)
10.    {
11.        std::cout << rand() << "\t";
12.
13.        // Если вывели 5 чисел, то вставляем символ новой строки
14.        if ((count+1) % 5 == 0)
15.            std::cout << "\n";
16.    }
17.}
```

Результат выполнения программы:

```
14867   24680   8872    25432   21865
17285   18997   10570   16397   30572
22339   31508   1553    124     779
6687    23563   5754    25989   16527
19808   10702   13777   28696   8131
```

```
18671   27093   8979    4088    31260
31016   5073    19422   23885   18222
3631    19884   10857   30853   32618
31867   24505   14240   14389   13829
13469   11442   5385    9644    9341
11470   189     3262    9731    25676
1366    24567   25223   110     24352
24135   459     7236    17918   1238
24041   29900   24830   1094    13193
10334   6192    6968    8791    1351
14521   31249   4533    11189   7971
5118    19884   1747    23543   309
28713   24884   1678    22142   27238
6261    12836   5618    17062   13342
14638   7427    23077   25546   21229
```

Стартовое число и последовательности в ГПСЧ

Если вы запустите вышеприведенную программу (генерация случайных чисел) несколько раз, то заметите, что в результатах всегда находятся одни и те же числа! Это означает, что, хотя каждое число в последовательности кажется случайным относительно предыдущего, вся последовательность не является случайной вообще! А это, в свою очередь, означает, что наша программа полностью предсказуема (одни и те же значения ввода приводят к одним и тем же значениям вывода). Бывают случаи, когда это может быть полезно или даже желательно (например, если вы хотите, чтобы научная симуляция повторялась, или вы пытаетесь исправить причины сбоя вашего генератора случайных подземелий в игре).

Но в большинстве случаев это не совсем то, что нам нужно. Если вы пишете игру типа *Hi-Lo* (где у пользователя есть 10 попыток угадать число, а компьютер говорит ему, насколько его предположения близки или далеки от реального числа), вы бы не хотели, чтобы программа выбирала одни и те же числа каждый раз. Поэтому давайте более подробно рассмотрим, почему это происходит и как это можно исправить.

Помните, что каждое новое число в последовательности ГПСЧ генерируется исходя из предыдущего определенным способом. Таким образом, при постоянном начальном числе ГПСЧ всегда будет генерировать одну и ту же последовательность! В программе, приведенной выше, последовательность чисел всегда одинакова, так как стартовое число всегда равно `4541`.

Чтобы это исправить нам нужен способ выбрать стартовое число, которое не будет фиксированным значением. Первое, что приходит на ум - использовать рандомное число! Это хорошая мысль, но если нам нужно случайное число для генерации случайных чисел, то это какой-то замкнутый круг, вам не кажется? Оказывается, нам не обязательно использовать случайное стартовое число - нам просто нужно выбрать что-то, что будет меняться каждый раз при новом запуске программы. Затем мы сможем использовать наш ГПСЧ для генерации уникальной последовательности рандомных чисел исходя из уникального стартового числа.

Общепринятым решением является использование системных часов. Каждый раз, при запуске программы, время будет другое. Если мы будем использовать значение времени в качестве стартового числа, то наша программа всегда будет генерировать разную последовательность чисел при каждом новом запуске!

В языке Си есть **функция time()**, которая возвращает в качестве времени общее количество секунд, прошедшее от полуночи 1 января 1970 года. Чтобы использовать эту функцию, нам просто нужно подключить заголовочный файл ctime, а затем инициализировать функцию srand() вызовом функции `time(0)`.

Вот вышеприведенная программа, но уже с использованием функции time() в качестве стартового числа:

```
#include <iostream>
#include <cstdlib> // для функций rand() и srand()
#include <ctime> // для функции time()

int main()
{
    srand(static_cast<unsigned int>(time(0))); // устанавливаем значение системных часов в качестве стартового числа

    for (int count=0; count < 100; ++count)
    {
        std::cout << rand() << "\t";

        // Если вывели 5 чисел, то вставляем символ новой строки
        if ((count+1) % 5 == 0)
            std::cout << "\n";
    }
}
```

Теперь наша программа будет генерировать разные последовательности случайных чисел! Попробуйте сами.

Генерация случайных чисел в заданном диапазоне

В большинстве случаев нам не нужны рандомные числа между `0` и `RAND_MAX` - нам нужны числа между двумя другими значениями: `min` и `max`. Например, если нам нужно сымитировать бросок кубика, то диапазон значений будет невелик: от `1` до `6`.

Вот небольшая функция, которая конвертирует результат функции rand() в нужный нам диапазон значений:

```
1. // Генерируем рандомное число между значениями min и max.
2. // Предполагается, что функцию srand() уже вызывали
3. int getRandomNumber(int min, int max)
4. {
5.     static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);

6.     // Равномерно распределяем рандомное число в нашем диапазоне
7.     return static_cast<int>(rand() * fraction * (max - min + 1) + min);
8. }
```

Чтобы сымитировать бросок кубика, вызываем функцию `getRandomNumber(1, 6)`.

Какой ГПСЧ является хорошим?

Как мы уже говорили, генератор случайных чисел, который мы написали выше, не является очень хорошим. Сейчас рассмотрим почему.

Хороший ГПСЧ должен иметь ряд свойств:

Свойство №1: **ГПСЧ должен генерировать каждое новое число с примерно одинаковой вероятностью**. Это называется **равномерностью распределения**. Если некоторые числа генерируются чаще, чем другие, то результат программы, использующей ГПСЧ, будет предсказуем! Например, предположим, вы пытаетесь написать генератор случайных предметов для игры. Вы выбираете случайное число от 1 до 10, и, если результатом будет 10, игрок получит крутой предмет вместо среднего. Шансы должны быть 1 к 10. Но, если ваш ГПСЧ неравномерно генерирует числа, например, десятки генерируются чаще, чем должны, то ваши игроки будут получать более редкие предметы чаще, чем предполагалось, и сложность, и интерес к такой игре автоматически уменьшаются.

Создать ГПСЧ, который бы генерировал равномерные результаты - сложно, и это одна из главных причин, по которым ГПСЧ, который мы написали в начале этого урока, не является очень хорошим.

***Свойство №2**:* **Метод, с помощью которого генерируется следующее число в последовательности, не должен быть очевиден или предсказуем**. Например, рассмотрим следующий алгоритм ГПСЧ: `num = num + 1`. У него есть равномерность распределения рандомных чисел, но это не спасает его от примитивности и предсказуемости!

***Свойство №3**:* **ГПСЧ должен иметь хорошее диапазонное распределение чисел**. Это означает, что маленькие, средние и большие числа должны возвращаться случайным образом. ГПСЧ, который возвращает все маленькие числа, а затем все большие — предсказуем и приведет к предсказуемым результатам.

***Свойство №4**:* **Период циклического повторения значений ГПСЧ должен быть максимально большим.** Все ГПСЧ являются циклическими, т.е. в какой-то момент последовательность генерируемых чисел начнет повторяться. Как упоминалось ранее, ГПСЧ являются детерминированными, и с одним значением ввода мы получим одно и то же значение вывода. Подумайте, что произойдет, когда ГПСЧ сгенерирует число, которое уже ранее было сгенерировано. С этого момента начнется дублирование последовательности чисел между первым и последующим появлением этого числа. Длина этой последовательности называется **периодом**.

Например, вот представлены первые 100 чисел, сгенерированные ГПСЧ с плохой периодичностью:

```
112    9      130     97     64
31     152    119     86     53
20     141    108     75     42
9      130    97      64     31
152    119    86      53     20
141    108    75      42      9
130    97     64      31    152
119    86     53      20    141
108    75     42       9    130
97     64     31     152    119
86     53     20     141    108
75     42      9     130     97
64     31    152     119     86
53     20    141     108     75
42      9    130      97     64
31    152    119      86     53
20    141    108      75     42
9     130     97      64     31
152   119     86      53     20
141   108     75      42      9
```

Заметили, что он сгенерировал `9`, как второе число, а затем как шестнадцатое? ГПСЧ застревает, генерируя последовательность между этими двумя 9-ми: `9-130-97-64-31-152-119-86-53-20-141-108-75-42-` (повтор).

Хороший ГПСЧ должен иметь длинный период для *всех* стартовых чисел. Разработка алгоритма, соответствующего этому требованию, может быть чрезвычайно сложной - большинство ГПСЧ имеют длинные периоды для одних начальных чисел и короткие для других. Если пользователь выбрал начальное число, которое имеет короткий период, то и результаты будут соответствующие.

Несмотря на сложность разработки алгоритмов, отвечающих всем этим критериям, в этой области было проведено большое количество исследований, так как разные ГПСЧ активно используются в важных отраслях науки.

Почему rand() является посредственным ГПСЧ?

Алгоритм, используемый для реализации rand(), может варьироваться в разных компиляторах, и, соответственно, результаты также могут быть разными. В большинстве реализаций rand() используется **Линейный Конгруэнтный Метод** (сокр. ***"ЛКМ"***). Если вы посмотрите на первый пример в этом уроке, то заметите, что там, на самом деле, используется ЛКМ, хоть и с намеренно подобранными плохими константами.

Одним из основных недостатков функции rand() является то, что `RAND_MAX` обычно устанавливается как `32767` (15-битное значение). Это означает, что если вы захотите сгенерировать числа в более широком диапазоне (например, 32-битные целые числа), то функция rand() не подойдет. Кроме того, она не подойдет, если вы захотите сгенерировать случайные числа типа с плавающей точкой (например, между `0.0` и `1.0`), что часто используется в статистическом моделировании. Наконец, функция rand() имеет относительно короткий период по сравнению с другими алгоритмами.

Тем не менее, этот алгоритм отлично подходит для изучения программирования и для программ, в которых высококлассный ГПСЧ не является необходимостью.

Для приложений, где требуется высококлассный ГПСЧ, рекомендуется использовать **алгоритм Вихрь Мерсенна** (англ. ***"Mersenne Twister"***), который генерирует отличные результаты и относительно прост в использовании.

Отладка программ, использующих случайные числа

Программы, которые используют случайные числа, трудно отлаживать, так как при каждом запуске такой программы мы будем получать разные результаты. А чтобы успешно проводить отладку программ, нужно удостовериться, что наша программа выполняется одинаково при каждом её запуске. Таким образом, мы сможем быстро узнать расположение ошибки и изолировать этот участок кода.

Поэтому, проводя отладку программы, полезно использовать в качестве стартового числа (с использованием функции srand()) определенное значение (например, `0`), которое вызовет ошибочное поведение программы. Это будет гарантией того, что наша программа каждый раз генерирует одни и те же результаты (что значительно облегчит процесс отладки). После того, как мы найдем и исправим ошибку, мы сможем снова использовать системные часы для генерации рандомных результатов.

Рандомные числа в C++11

В C++11 добавили тонну нового функционала для генерации случайных чисел, включая алгоритм Вихрь Мерсенна, а также разные виды генераторов случайных чисел (например, равномерные, генератор Poisson и пр.). Доступ к ним осуществляется через подключение заголовочного файла random. Вот пример генерации случайных чисел в C++11 с использованием Вихря Мерсенна:

```
#include <iostream>
// #include <ctime> // раскомментируйте, если используете Code::Blocks
#include <random> // для std::random_device и std::mt19937

int main()
{
    std::random_device rd;
    std::mt19937 mersenne(rd()); // инициализируем Вихрь Мерсенна случайным стартовым числом

// Примечание: Из-за одного бага в компиляторе Code::Blocks (если вы используете Code::Blocks в Windows) - удалите две строки кода выше и раскомментируйте следующую строку:
// std::mt19937 mersenne(static_cast<unsigned int>(time(0))); // инициализируем Вихрь Мерсенна случайным стартовым числом

    // Выводим несколько случайных чисел
    for (int count = 0; count < 48; ++count)
    {
        std::cout << mersenne() << "\t";

        // Если вывели 5 чисел, то вставляем символ новой строки
        if ((count + 1) % 5 == 0)
            std::cout << "\n";
    }
}
```

Вихрь Мерсенна генерирует случайные 32-битные целые числа unsigned (а не 15-битные целые числа, как в случае с rand()), что позволяет использовать гораздо больший диапазон значений. Существует также версия (`std::mt19937_64`) для генерации 64-битных целых чисел unsigned!

> ***Примечание для пользователей Visual Studio**:* Реализация функции rand() в Visual Studio имеет один существенный недостаток - первое генерируемое случайное число не сильно отличается от стартового. Это означает, что, при использовании time() для генерации начального числа, первое число не будет сильно отличаться/изменяться от стартового и при последующих запусках. Есть простое решение: вызовите функцию rand() один раз и сбросьте результат. Затем вы сможете использовать rand(), как обычно в вашей программе.

Урок №75. Обработка некорректного пользовательского ввода

Большинство программ, имеющих хоть какой-либо пользовательский интерфейс, сталкиваются с обработкой пользовательского ввода. В программах, которые мы писали раньше, используется std::cin для получения пользовательского ввода. А так как ввод данных имеет свободную форму (пользователь может ввести всё, что пожелает), то пользователю очень легко ввести данные, которые от него никак не ожидаются.

При написании программ, вы всегда должны учитывать, как пользователи (преднамеренно или непреднамеренно) могут использовать ваши программы. Хорошо написанная программа предвидит, как ею могут злоупотребить, и либо будет обрабатывать такие случаи, либо предотвращать их (если это возможно). Программа, которая имеет обработку некорректного ввода, называется **надежной**.

На этом уроке мы рассмотрим, как пользователи могут вводить некорректные данные через std::cin, а также как с этим бороться.

std::cin, буфер данных и извлечение

Прежде чем разбираться с обработкой некорректного ввода через std::cin и оператор `>>`, давайте сначала рассмотрим их принцип работы.

Процесс, когда мы используем оператор `>>` для получения данных от пользователя и размещение этих данных в определенной переменной, называется **извлечением**. Соответственно, **оператор** `>>` называется **оператором извлечения**.

Когда пользователь вводит данные в ответ на операцию извлечения, то эти данные помещаются в буфер std::cin. **Буфер данных** - это просто часть памяти, зарезервированная для временного хранения данных, когда они перемещаются из одного места в другое. В этом случае буфер используется для хранения пользовательского ввода, пока он находится в режиме ожидания выделения для него переменных.

При использовании оператора извлечения, выполняется следующая процедура:

- Если во входном буфере есть данные, то эти данные используются для извлечения.

- Если во входном буфере нет данных, то пользователю предлагается ввести данные (обычно именно это и происходит в большинстве случаев). Когда пользователь нажимает Enter, символ новой строки `\n` помещается во входной буфер.
- Оператор `>>` извлекает столько данных из входного буфера в переменную, сколько позволяет размер самой переменной (игнорируя любые пробелы, табы или `\n`).
- Любые данные, которые не были извлечены, остаются во входном буфере для последующего извлечения.

Извлечение данных считается успешным, если по крайней мере один символ был извлечен из входного буфера. Оставшиеся данные во входном буфере используются для последующих извлечений. Например:

```
1. int a;
2. std::cin >> a;
```

Если пользователь введет `7d`, то `7` будет извлечено, преобразовано в целое число и присвоено переменной `a`. А `d\n` останется во входном буфере дожидаться следующего извлечения.

Извлечение не выполняется, если данные ввода не соответствуют типу переменной, выделенной для них. Например:

```
1. int a;
2. std::cin >> a;
```

Если бы пользователь ввел `z`, то извлечение не выполнилось бы, так как `z` не может быть извлечено в целочисленную переменную.

Проверка пользовательского ввода

Существует три основных способа проверки пользовательского ввода:

- До ввода
  - Предотвращение некорректного пользовательского ввода.
- После ввода
  - Пользователь может вводить всё, что хочет, но осуществляется последующая проверка данных. Если проверка прошла успешно, то выполняется перемещение данных в переменную.
  - Пользователь может вводить всё, что хочет, но при операции извлечения данных оператором `>>` параллельно решаются возможные ошибки.

Некоторые графические пользовательские интерфейсы или расширенные текстовые интерфейсы позволяют проверять ввод пользователя сразу (символ за символом). Программист создает функцию проверки данных, которая принимает и проверяет пользовательский ввод, и, если данные ввода корректны, то возвращается true, если нет — false. Эта функция вызывается каждый раз, когда пользователь нажимает на клавишу. Если функция проверки возвращает true, то символ, который пользователь ввел — принимается. Если функция возвращает false, то символ, который только что ввел пользователь — отбрасывается (и не отображается на экране). Используя этот метод, мы можем гарантировать, что любой пользовательский ввод будет корректным, так как любые неверные нажатия клавиш будут обнаружены и немедленно удалены. К сожалению, std::cin не поддерживает этот тип проверки.

Поскольку строки не имеют никаких ограничений на то, какие символы вводятся, то извлечение гарантированно будет успешным (хотя помним, что std::cin останавливает процесс извлечения при первом обнаружении символа пробела). После ввода строки программа сразу может её проанализировать. Однако этот анализ и последующая конвертация данных в другие типы данных (например, числа) может быть сложной, поэтому это делается только в редких случаях.

Чаще всего мы позволяем std::cin и оператору извлечения выполнять тяжелую работу. В соответствии с этим методом пользователь может вводить всё, что хочет, а далее std::cin и оператор `>>` пытаются извлечь данные и, если что-то пойдет не так, выполняется **обработка возможных ошибок**. Это самый простой способ, и его мы и будем рассматривать.

Рассмотрим следующую программу «Калькулятор», которая не имеет обработки ошибок:

```
1. #include <iostream>
2.
3. double getValue()
4. {
5.     std::cout << "Enter a double value: ";
6.     double a;
7.     std::cin >> a;
8.     return a;
9. }
10.
11.char getOperator()
12.{
13.    std::cout << "Enter one of the following: +, -, *, or /: ";
14.    char sm;
15.    std::cin >> sm;
16.    return sm;
17.}
18.
```

```
19. void printResult(double a, char sm, double b)
20. {
21.     if (sm == '+')
22.         std::cout << a << " + " << b << " is " << a + b << '\n';
23.     else if (sm == '-')
24.         std::cout << a << " - " << b << " is " << a - b << '\n';
25.     else if (sm == '*')
26.         std::cout << a << " * " << b << " is " << a * b << '\n';
27.     else if (sm == '/')
28.         std::cout << a << " / " << b << " is " << a / b << '\n';
29. }
30. 
31. int main()
32. {
33.     double a = getValue();
34.     char sm = getOperator();
35.     double b = getValue();
36. 
37.     printResult(a, sm, b);
38. 
39.     return 0;
40. }
```

Здесь мы просим пользователя ввести два числа и арифметический оператор. Результат выполнения программы:

```
Enter a double value: 4
Enter one of the following: +, -, *, or /: *
Enter a double value: 5
4 * 5 is 20
```

Теперь рассмотрим, где некорректный ввод пользователя может привести к сбою в программе.

Сначала мы просим пользователя ввести первое число. А что будет, если он введет что-либо другое (например, `q`)? В этом случае извлечение данных не произойдет.

Во-вторых, мы просим пользователя ввести один из 4-х возможных символов (арифметических операторов). Что будет, если он введет какой-то другой символ? Мы сможем извлечь данные, но не сможем их обработать.

В-третьих, что будет, если пользователь вместо символа введет строку, например, `*q hello`. Хотя мы можем извлечь символ `*`, но в буфере останется лишний балласт, который в будущем может привести к проблемам.

Основные типы некорректного пользовательского ввода

Я выделил 4 типа:

- Извлечение выполняется успешно, но значения бесполезны для программы (например, вместо математического оператора введен символ `k`).
- Извлечение выполняется успешно, но пользователь вводит лишний текст (например, `*q hello` вместо одного символа математического оператора).
- Извлечение не выполняется (например, вместо числового значения ввели символ `q`).
- Извлечение выполняется успешно, но пользователь ввел слишком большое числовое значение.

Таким образом, чтобы наши программы были надежными, то всякий раз, когда мы просим пользователя ввести данные, мы должны предугадать возможность возникновения любого из вышеуказанных событий, и, если это произойдет, в программе должна быть обработка таких случаев.

Давайте рассмотрим каждую из ситуаций, приведенных выше, а также алгоритм действий, если такая ситуация случилась.

Ошибка №1: Извлечение выполняется успешно, но данные бесполезны

Это самый простой случай. Рассмотрим следующий фрагмент выполнения программы, приведенной выше:

```
Enter a double value: 4
Enter one of the following: +, -, *, or /: k
Enter a double value: 5
```

Здесь мы просим пользователя ввести один из 4-х арифметических операторов, но он вводит `k`. Символ `k` является допустимым символом, поэтому std::cin извлекает его в переменную `sm`, и всё это добро возвращается обратно в функцию main(). Но в программе не предусмотрен случай обработки подобного ввода, поэтому мы получаем сбой, и в результате ничего не выводится.

Решение простое: выполнить проверку пользовательского ввода. Обычно она состоит из 3-х шагов:

- Проверить пользовательский ввод на ожидаемые значения.
- Если ввод совпадает с ожидаемым, то значения благополучно возвращаются.

- Если нет, то сообщаем пользователю что что-то пошло не так, и просим повторить ввод снова.

Вот обновленная функция getOperator() с проверкой пользовательского ввода:

```
1. char getOperator()
2. {
3.     while (true) // цикл продолжается до тех пор, пока пользователь не введет корректное значение
4.     {
5.         std::cout << "Enter one of the following: +, -, *, or /: ";
6.         char sm;
7.         std::cin >> sm;
8.
9.         // Выполняем проверку значений
10.        if (sm == '+' || sm == '-' || sm == '*' || sm == '/')
11.            return sm; // возвращаем данные в функцию main()
12.        else // в противном случае, сообщаем пользователю, что что-то пошло не так
13.            std::cout << "Oops, that input is invalid.  Please try again.\n";
14.        }
15. }
```

Мы используем цикл while для гарантии корректного ввода. Если пользователь введет один из ожидаемых символов, то всё хорошо - символ возвратится обратно в функцию main(), если нет — пользователю выведется просьба повторить ввод снова. И вводить данные он будет до тех пор, пока не введет корректное значение, не закроет программу или не уничтожит свой компьютер :)

Ошибка №2: Извлечение выполняется успешно, но пользователь вводит лишний текст

Рассмотрим следующее выполнение программы «Калькулятор»:

```
Enter a double value: 4*5
```

Как вы думаете, что произойдет дальше?

```
Enter a double value: 4*5
Enter one of the following: +, -, *, or /: Enter a double value: 4 * 5 is 20
```

Программа выведет верный результат, но её порядок выполнения неправильный. Почему? Сейчас разберемся.

Когда пользователь вводит `4*5`, то эти данные поступают в буфер. Затем оператор `>>` извлекает `4` в переменную `a`, оставляя `*5\n` в буфере. Затем программа выводит `Enter one of the following: +, -, *, or /:`. Однако, когда вызывается

оператор извлечения, он видит, что в буфере находится `*5\n`, поэтому он использует эти данные вместо того, чтобы запрашивать их у пользователя еще раз. Следовательно, извлекается символ `*`, а `5\n` остается во входном буфере.

После того, как пользователя просят ввести другое число, `5` извлекается из буфера, не дожидаясь ответа от пользователя. Поскольку у пользователя не было возможности ввести другие значения и нажать Enter (вставляя символ новой строки), то всё происходит на одной строке.

Хотя результат программы правильный, но её выполнение — нет. Согласитесь, что с наличием возможности просто проигнорировать лишние символы было бы намного лучше. К счастью, это можно сделать следующим образом:

```
1. std::cin.ignore(32767, '\n'); // удаляем до 32767 символов из входного буфера
   вплоть до появления символа '\n' (который мы также удаляем)
```

Поскольку последний символ, введенный пользователем, должен быть `\n` (так как пользователь в конце ввода нажимает Enter), то мы можем сообщить std::cin игнорировать все символы в буфере до тех пор, пока не будет найден символ новой строки (который мы также удаляем). Таким образом, всё лишнее будет успешно проигнорировано.

Обновим нашу функцию getDouble(), добавив эту строчку:

```
1. double getValue()
2. {
3.     std::cout << "Enter a double value: ";
4.     double a;
5.     std::cin >> a;
6.     std::cin.ignore(32767, '\n'); // удаляем до 32767 символов из входного
   буфера вплоть до появления символа '\n' (который мы также удаляем)
7.     return a;
8. }
```

Теперь наша программа будет работать так, как надо, даже если мы введем `4*5` в первой строке. Число `4` будет извлечено в переменную, а все остальные символы будут удалены из входного буфера. Поскольку входной буфер теперь пуст, то при последующем выполнении операции извлечения всё пройдет гладко и порядок выполнения программы не будет нарушен.

Ошибка №3: Извлечение не выполняется

Рассмотрим следующее выполнение программы «Калькулятор»:

```
Enter a double value: a
```

Теперь уже не удивительно, что программа работает не так, как надо, но её дальнейший ход выполнения — вот что интересно:

```
Enter a double value: a
Enter one of the following: +, -, *, or /: Enter a double value:
```

И программа внезапно обрывается.

Этот случай похож на ошибку №2, но все же несколько отличается. Давайте рассмотрим детально, что здесь происходит.

Когда пользователь вводит `a`, то этот символ помещается в буфер. Затем оператор `>>` пытается извлечь `a` в переменную `a` типа double. Поскольку `a` нельзя конвертировать в тип double, то оператор `>>` не может выполнить извлечение. На этом этапе случаются две вещи: `a` остается в буфере, а std::cin переходит в «**режим отказа**». Как только установлен этот режим, то все последующие запросы на извлечение данных будут проигнорированы.

К счастью, мы можем определить, удачно ли прошло извлечение или нет. Если нет, то мы можем исправить ситуацию следующим образом:

```
if (std::cin.fail()) // если предыдущее извлечение было неудачным,
{
    std::cin.clear(); // то возвращаем cin в 'обычный' режим работы
    std::cin.ignore(32767,'\n'); // и удаляем значения предыдущего ввода из входного буфера
}
```

Вот так! Теперь давайте интегрируем это в нашу функцию getValue():

```
double getValue()
{
    while (true) // цикл продолжается до тех пор, пока пользователь не введет корректное значение
    {
        std::cout << "Enter a double value: ";
        double a;
        std::cin >> a;

        if (std::cin.fail()) // если предыдущее извлечение оказалось неудачным,
        {
            std::cin.clear(); // то возвращаем cin в 'обычный' режим работы
            std::cin.ignore(32767,'\n'); // и удаляем значения предыдущего ввода из входного буфера
        }
        else // если всё хорошо, то возвращаем a
            return a;
    }
}
```

Ошибка №4: Извлечение выполняется успешно, но пользователь ввел слишком большое числовое значение

Рассмотрим следующий код:

```
#include <iostream>
#include <cstdint>

int main()
{
    std::int16_t x { 0 }; // переменная x занимает 16 бит, её диапазон значений: от -32768 до 32767
    std::cout << "Enter a number between -32768 and 32767: ";
    std::cin >> x;

    std::int16_t y { 0 }; // переменная y занимает 16 бит, её диапазон значений: от -32768 до 32767
    std::cout << "Enter another number between -32768 and 32767: ";
    std::cin >> y;

    std::cout << "The sum is: " << x + y << '\n';
    return 0;
}
```

Что случится, если пользователь введет слишком большое число (например, `40000`)?

```
Enter a number between -32768 and 32767: 40000
Enter another number between -32768 and 32767: The sum is: 32767
```

В вышеприведенном примере std::cin немедленно перейдет в «режим отказа», и значение не будет присвоено переменной `x`. Следовательно, переменной `x` присваивается максимально возможное значение типа данных (в этом случае, `32767`). Все следующие данные ввода будут проигнорированы, а `y` останется с инициализированным значением `0`. Решение такое же, как и в случае с неудачным извлечением (см. ошибка №3).

Объединяем всё вместе

Вот программа «Калькулятор», но уже с полным механизмом обработки/проверки ошибок:

```
#include <iostream>

double getValue()
{
    while (true) // цикл продолжается до тех пор, пока пользователь не введет корректное значение
    {
        std::cout << "Enter a double value: ";
```

```
8.          double a;
9.          std::cin >> a;
10.
11.         // Проверка на предыдущее извлечение
12.         if (std::cin.fail()) // если предыдущее извлечение оказалось неудачным,
13.         {
14.             std::cin.clear(); // то возвращаем cin в 'обычный' режим работы
15.             std::cin.ignore(32767,'\n'); // и удаляем значения предыдущего
    ввода из входного буфера
16.             std::cout << "Oops, that input is invalid.  Please try again.\n";
17.         }
18.         else
19.         {
20.             std::cin.ignore(32767,'\n'); // удаляем лишние значения
21.
22.             return a;
23.         }
24.     }
25. }
26.
27. char getOperator()
28. {
29.     while (true) // цикл продолжается до тех пор, пока пользователь не введет
    корректное значение
30.     {
31.         std::cout << "Enter one of the following: +, -, *, or /: ";
32.         char sm;
33.         std::cin >> sm;
34.
35.         // Переменные типа char могут принимать любые символы из
    пользовательского ввода, поэтому нам не стоит беспокоиться по поводу
    возникновения неудачного извлечения
36.
37.         std::cin.ignore(32767,'\n'); // удаляем лишний балласт
38.
39.         // Выполняем проверку пользовательского ввода
40.         if (sm == '+' || sm == '-' || sm == '*' || sm == '/')
41.             return sm; // возвращаем обратно в caller
42.         else // в противном случае, сообщаем пользователю что что-
    то пошло не так
43.             std::cout << "Oops, that input is invalid.  Please try again.\n";
44.         }
45. }
46.
47. void printResult(double a, char sm, double b)
48. {
49.     if (sm == '+')
50.         std::cout << a << " + " << b << " is " << a + b << '\n';
51.     else if (sm == '-')
52.         std::cout << a << " - " << b << " is " << a - b << '\n';
53.     else if (sm == '*')
54.         std::cout << a << " * " << b << " is " << a * b << '\n';
55.     else if (sm == '/')
56.         std::cout << a << " / " << b << " is " << a / b << '\n';
57.     else
58.         std::cout << "Something went wrong: printResult() got an invalid operat
    or.";
59.
60. }
61.
62. int main()
63. {
```

```
64.     double a = getValue();
65.     char sm = getOperator();
66.     double b = getValue();
67.
68.     printResult(a, sm, b);
69.
70.     return 0;
71. }
```

Заключение

При написании программ, всегда думайте о том, как пользователи могут злоупотребить ими или использовать не по назначению, особенно то, что касается ввода данных. Подумайте:

- Может ли извлечение не выполниться?
- Может ли пользователь ввести значение больше ожидаемого?
- Может ли пользователь ввести бессмысленные значения?
- Может ли ввод пользователя привести к переполнению переменных?

Используйте ветвление if и логические переменные для проверки пользовательского ввода.

Следующий код осуществляет проверку пользовательского ввода и исправляет проблемы с переполнением или неудачным извлечением данных:

```
1. if (std::cin.fail()) // если предыдущее извлечение не выполнилось или произошло
    переполнение,
2. {
3.     std::cin.clear(); // то возвращаем cin в 'обычный' режим работы
4.     std::cin.ignore(32767,'\n'); // и удаляем значения предыдущего ввода из
   входного буфера
5. }
```

> ***Примечание:*** Проверка пользовательского ввода очень важна и полезна, но она, к сожалению, приводит к усложнению кода, что, в свою очередь, затрудняет его понимание. Поэтому на следующих уроках мы не будем её использовать, дабы всё оставалось максимально простым и доступным.

Урок №76. Введение в тестирование кода

Итак, вы написали программу, она компилируется, и даже работает! Что дальше? Есть несколько вариантов.

Если вы написали программу, чтобы её один раз запустить и забыть, то дальше ничего делать не нужно. Здесь не столь важно, что ваша программа может некорректно работать в некоторых случаях: если при первом запуске она работает, так как вы и ожидали, и если вы дальше запускать и использовать её не планируете, тогда всё - финиш.

Если ваша программа полностью линейна (не имеет условного ветвления: операторов if или switch), не принимает входных данных и выводит правильный результат, тогда тоже финиш. В этом случае вы уже протестировали всю программу, запустив её один раз и сверив результаты.

Но если вы написали программу, которую собираетесь запускать много раз и которая имеет циклы и условные ветвления, принимает пользовательский ввод, то здесь уже немного по-другому. Возможно, вы написали функцию, которую хотите повторно использовать в других программах. Возможно, вы даже намереваетесь распространять эту программу в дальнейшем. В таком случае вам действительно нужно будет проверить, как ваша программа работает в самых разных условиях.

Только потому, что она корректно выполнилась с одними значениями пользовательского ввода, совсем не значит, что она будет работать корректно и с другими значениями.

Тестирование программного обеспечения - это процесс определения работоспособности программного обеспечения согласно ожиданиям разработчика.

Прежде чем мы будем говорить о некоторых практических способах тестирования вашего кода, давайте поговорим о том, почему комплексное тестирование может быть сложным. Например, рассмотрим следующую программу:

```
1. #include <iostream>
2. #include <string>
3.
4. void compare(int a, int b)
5. {
6.     if (a > b)
7.         std::cout << a << " is greater than " << b << '\n'; // случай №1
8.     else if (a < b)
9.         std::cout << a << " is less than " << b << '\n'; // случай №2
10.    else
```

```
11.         std::cout << a << " is equal to " << b << '\n'; // случай №3
12. }
13.
14. int main()
15. {
16.     std::cout << "Enter a number: ";
17.     int a;
18.     std::cin >> a;
19.
20.     std::cout << "Enter another number: ";
21.     int b;
22.     std::cin >> b;
23.
24.         compare(a, b);
25. }
```

Учитывая 4-байтовый тип int и его диапазон значений, для тестирования всех возможных значений нам потребуется 18 446 744 073 709 551 616 (~ 18 квинтиллионов) раз запустить эту программу. Понятно, что это абсурд.

Каждый раз, когда мы запрашиваем пользовательский ввод или используем условное ветвление в программе - мы увеличиваем в разы количество возможных способов выполнения нашей программы. Для всех программ, кроме простейших, тестировать каждую комбинацию входных данных, да еще и вручную — как-то не логично, вам не кажется?

Сейчас ваша интуиция должна подсказывать вам, что для того, чтобы убедиться в полной работоспособности программы, приведенной выше, не нужно будет её запускать 18 квинтиллионов раз. Вы можете прийти к выводу, что если код выполняется, когда выражение `x > y` равно true при одной паре значений `x` и `y`, то код должен корректно работать и с любыми другими парами `x` и `y`, где `x > y`. Учитывая это, становится очевидным, что для тестирования программы нам потребуется запустить её всего лишь три раза (по одному для каждого случая: `x > y`, `x < y`, `x = y`), чтобы убедиться, что она работает корректно. Есть и другие трюки, позволяющие упростить процесс тестирования кода.

Про методологии тестирования можно долго рассказывать, но так как эта тема не является специфической именно для языка C++, то мы будем придерживаться краткого и понятного изложения материала для начинающего разработчика, который тестирует свой собственной код.

Выполнение неофициального тестирования

Большинство разработчиков проводят **неофициальное тестирование**, когда пишут свои программы. После написания части кода (функции, класса или какого-либо другого "куска кода") разработчик пишет некий код для проверки только что

добавленной части, и, если тест пройден успешно, разработчик удаляет код этого теста. Например, для следующей функции isLowerVowel() мы можем написать следующий код для проверки:

```
#include <iostream>

bool isLowerVowel(char c)
{
    switch (c)
    {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        return true;
    default:
        return false;
    }
}

int main()
{
    std::cout << isLowerVowel('a'); // временный тестовый код, результатом которого должна быть 1
    std::cout << isLowerVowel('q'); // временный тестовый код, результатом которого должен быть 0
}
```

Если при выполнении программы вы получите `1` и `0`, тогда всё хорошо. Вы знаете, что ваша функция работает, поэтому можно удалить временный тестовый код и продолжить процесс программирования.

Совет №1: Пишите свою программу по частям: в небольших, чётко определенных единицах (функциях)

Возьмем, к примеру, автопроизводителя, который создает автомобиль. Как вы думаете, что из следующего он делает?

- Создает (или покупает) и проверяет каждый компонент автомобиля отдельно перед его установкой. Как только компонент успешно проходит проверку, автопроизводитель интегрирует его в автомобиль и повторяет проверку, чтобы убедиться, что интеграция прошла успешно. В конце, перед презентацией, проводится генеральный тест работоспособности всего автомобиля.
- Создает автомобиль из всех компонентов без какой-либо предварительной проверки (за один присест). Затем, в конце, проводится первое и окончательное тестирование работоспособности уже собранного автомобиля.

Не кажется вам, что более правильным является первый вариант? И все же большинство начинающих программистов пишут свой код в соответствии со вторым вариантом!

Во втором случае, если какая-либо из частей автомобиля будет работать неправильно, то механику придется провести диагностику всего автомобиля, чтобы определить, что пошло не так - проблема может находиться где угодно. Например, автомобиль может не заводиться из-за неисправной свечи зажигания, аккумулятора, топливного насоса или чего-то еще. Это приведет к потере большого количества потраченного впустую времени в попытках точного определения корня проблемы. И, если проблема будет найдена, последствия могут быть катастрофическими: изменения в одной части автомобиля могут привести к «эффекту бабочки» - серьезным изменениям в других частях автомобиля. Например, слишком маленький топливный насос может привести к изменению двигателя, что приведет к реорганизации каркаса автомобиля. В конечном итоге вам придется переделывать большую часть авто, просто чтобы исправить то, что изначально было небольшой проблемой!

В первом случае автопроизводитель проверяет все детали по мере поступления. Если какой-либо из компонентов оказался бракованным, то механики сразу понимают, в чем проблема и как её решить. Ничто не интегрируется в автомобиль, пока не будет успешно протестировано. К тому времени, когда они уже соберут весь автомобиль, у них будет разумная уверенность в его работоспособности - в конце концов, все его части были успешно протестированы. Все же есть вероятность, что что-то может пойти не так при соединении всех частей, но по сравнению со вторым вариантом - это очень малая вероятность, о которой и не следует серьезно беспокоиться.

Вышеупомянутая аналогия справедлива и для программистов, хотя, по некоторым причинам, новички часто этого не осознают. Гораздо лучше писать небольшие функции, а затем сразу их компилировать и тестировать. Таким образом, если вы допустили ошибку, вы будете знать, что она находится в небольшом количестве кода, который вы только что написали/изменили. А это, в свою очередь, означает, что площадь поиска ошибки невелика, и времени на отладку будет потрачено намного меньше.

Правило: Часто компилируйте свой код и всегда тестируйте все нетривиальные функции, которые вы пишете.

Совет №2: Нацеливайтесь на 100%-ное покрытие кода

Термин **"покрытие кода"** относится к количеству исходного кода программы, который был задействован во время тестирования. Есть много разных показателей покрытия кода, но лишь о нескольких из них стоит упомянуть.

Покрытие стейтментов — это процент стейтментов в вашем коде, которые были задействованы во время выполнения тестирования. Например:

```
1. int boo(int a, int b)
2. {
3.     bool z = b;
4.     if (a > b)
5.     {
6.         z = a;
7.     }
8.     return z;
9. }
```

Вызов `boo(1, 0)` даст вам полный охват стейтментов этой функции, так как выполнится каждая строка кода.

В случае с функцией isLowerVowel():

```
1. bool isLowerVowel(char c)
2. {
3.     switch (c) // стейтмент №1
4.     {
5.     case 'a':
6.     case 'e':
7.     case 'i':
8.     case 'o':
9.     case 'u':
10.        return true; // стейтмент №2
11.    default:
12.        return false; // стейтмент №3
13.    }
14. }
```

Здесь потребуется два вызова для проверки всех стейтментов, так как определить работу стейтментов №2 и №3 в одном вызове функции мы не сможем.

Правило: Убедитесь, что во время тестирования задействованы все стейтменты вашей функции.

Совет №3: Нацеливайтесь на 100%-ное покрытие ветвлений

Термин **"покрытие ветвлений"** относится к проценту ветвлений, которые были выполнены в каждом случае (положительном и отрицательном) отдельно.

Оператор switch может иметь много ветвлений. Оператор if имеет два ветвления: случай true и случай false (даже если нет оператора else). Например:

```
1. int boo(int a, int b)
2. {
3.     bool z = b;
4.     if (a > b)
5.     {
6.         z = a;
7.     }
8.     return z;
9. }
```

Предыдущий вызов `boo(1, 0)` дал нам 100%-ный охват стейтментов и ветвление true. Но это всего лишь 50%-ный охват ветвлений. Нам нужен еще один вызов — `boo(0, 1)`, чтобы протестировать ветвление false.

В функции isLowerVowel() нужны два вызова (например, `isLowerVowel('a')` и `isLowerVowel('q')`), чтобы убедиться в 100%-ном охвате ветвлений (все буквы, которые находятся в switch, тестировать не обязательно, если сработала одна — сработают и другие):

```
1. #include <iostream>
2.
3. bool isLowerVowel(char c)
4. {
5.     switch (c)
6.     {
7.     case 'a':
8.     case 'e':
9.     case 'i':
10.    case 'o':
11.    case 'u':
12.        return true;
13.    default:
14.        return false;
15.    }
16.}
17.
18.int main()
19.{
20.    std::cout << isLowerVowel('a'); // временный тестовый код, результатом
   которого должна быть 1
21.    std::cout << isLowerVowel('q'); // временный тестовый код, результатом
   которого должен быть 0
22.}
```

Пересмотрим функцию сравнения из первого примера данного урока:

```
1. void compare(int a, int b)
2. {
3.     if (a > b)
4.         std::cout << a << " is greater than " << b << '\n'; // случай №1
5.     else if (a < b)
6.         std::cout << a << " is less than " << b << '\n'; // случай №2
```

```
7.      else
8.          std::cout << a << " is equal to " << b << '\n'; // случай №3
9.  }
```

Здесь необходимы 3 вызова функции, чтобы получить 100%-ный охват ветвлений:

- `compare(1,0)` проверяет вариант true для первого оператора if.
- `compare(0, 1)` проверяет вариант false для первого оператора if и вариант true для второго оператора if (else if).
- `compare(0, 0)` проверяет вариант false для второго оператора if и выполняет инструкцию else.

Таким образом, мы можем сказать, что эту функцию можно протестировать с помощью всего лишь 3-х вызовов функции (а не 18 квинтиллионов раз).

Правило: Тестируйте каждый случай ветвления в вашей программе.

Совет №4: Нацеливайтесь на 100%-ное покрытие циклов

Покрытие циклов (неофициально называемый ***«тест 0, 1, 2»***) сообщает, что если у вас есть цикл в коде, то, чтобы убедиться в его работоспособности, нужно его выполнить 0, 1 и 2 раза. Если он работает корректно во второй итерации, то должен работать корректно и для всех последующих итераций (3, 4, 10, 100 и т.д.). Например:

```
1.  #include <iostream>
2.
3.  int spam(int timesToPrint)
4.  {
5.      for (int count=0; count < timesToPrint; ++count)
6.          std::cout << "Spam!!!";
7.  }
```

Чтобы протестировать цикл внутри функции, нам придется вызвать его 3 раза:

- `spam(0)`, чтобы проверить случай нулевой итерации.
- `spam(1)` для проверки итерации №1 и `spam(2)` для проверки итерации №2.
- Если `spam(2)` работает, тогда и `spam(n)` будет работать (где `n > 2`).

Правило: Используйте "тест 0, 1, 2" для проверки циклов на корректную работу с разным количеством итераций.

Совет №5: Убедитесь, что вы тестируете разные типы ввода

Когда вы пишете функции, которые принимают параметры или пользовательский ввод, то посмотрите, что происходит с разными типами ввода. Например, если я

написал функцию вычисления квадратного корня из целого числа, то какие значения имело бы смысл протестировать? Вероятнее всего, вы бы начали с обычных значений, например, с `4`. Но также было бы неплохо протестировать и `0`, и какое-нибудь отрицательное число.

Вот несколько основных рекомендаций по тестированию разных типов ввода:

- Для целых чисел убедитесь, что вы проверили, как ваша функция обрабатывает `0`, отрицательные и положительные значения. При наличии пользовательского ввода вы также должны проверить вариант возникновения переполнения.
- Для чисел типа с плавающей запятой убедитесь, что вы рассмотрели варианты, как ваша функция обрабатывает значения, которые имеют неточности (значения, которые немного больше/меньше ожидаемых). Хорошие тестовые значения — это `0.1` и `-0.1` (для проверки чисел, которые немного больше ожидаемых) и `0.6` и `-0.6` (для проверки чисел, которые немного меньше ожидаемых).
- Для строк убедитесь, что вы рассмотрели вариант, как ваша функция обрабатывает пустую строку, строку с допустимыми значениями, строку с пробелами и строку, содержимым которой являются одни пробелы.

Правило: Тестируйте разные типы ввода, чтобы убедиться, что ваш "кусок кода" правильно их обрабатывает.

Сохранение ваших тестов

Хотя написание тестов и последующее их удаление - достаточно хороший вариант для быстрого и временного тестирования, но для кода, который вы намереваетесь повторно использовать или модифицировать в будущем, имеет смысл сохранять эти тесты. Например, вместо удаления вашего временного теста, вы можете переместить его в функцию test():

```
1. #include <iostream>
2.
3. bool isLowerVowel(char c)
4. {
5.     switch (c)
6.     {
7.     case 'a':
8.     case 'e':
9.     case 'i':
10.    case 'o':
11.    case 'u':
12.        return true;
13.    default:
14.        return false;
```

```
15.     }
16. }
17.
18. // Эта функция сейчас нигде не вызывается, но находится здесь в случае, если
    вы захотите повторно провести тестирование
19. void test()
20. {
21.     std::cout << isLowerVowel('a'); // временный тестовый код, результатом
    которого должна быть 1
22.     std::cout << isLowerVowel('q'); // временный тестовый код, результатом
    которого должен быть 0
23. }
24.
25. int main()
26. {
27.     return 0;
28. }
```

Автоматизация тестирования

Одна из проблем с вышеупомянутой тестовой функцией заключается в том, что вам придется вручную проверять результаты теста. А можно сделать лучше — добавить к тесту правильные ожидаемые результаты, которые должны получиться при успешном тестировании:

```
1.  #include <iostream>
2.
3.  bool isLowerVowel(char c)
4.  {
5.      switch (c)
6.      {
7.      case 'a':
8.      case 'e':
9.      case 'i':
10.     case 'o':
11.     case 'u':
12.         return true;
13.     default:
14.         return false;
15.     }
16. }
17.
18. // Возвращается номер теста, который не был пройден или 0, если все тесты
    были пройдены успешно
19. int test()
20. {
21.     if (isLowerVowel('a') != true) return 1;
22.     if (isLowerVowel('q') != false) return 2;
23.
24.     return 0;
25. }
26.
27. int main()
28. {
29.     return 0;
30. }
```

Теперь вы можете вызывать test() в любое время и функция сама всё сделает за вас.

Тест

Задание №1

Когда вы должны начинать тестировать свой код?

Задание №2

Сколько тестов потребуется для минимального подтверждения работоспособности следующей функции?

```
1. bool isLowerVowel(char c, bool yIsVowel)
2. {
3.     switch (c)
4.     {
5.     case 'a':
6.     case 'e':
7.     case 'i':
8.     case 'o':
9.     case 'u':
10.        return true;
11.    case 'y':
12.        return (yIsVowel ? true : false);
13.    default:
14.        return false;
15.    }
16.}
```

Глава №5. Итоговый тест

Поздравляю! Мы продвинулись еще на одну главу вперед. Чтобы закрепить пройденный материал, давайте быстренько повторим теорию и выполним 2 практических задания.

Теория

Операторы if позволяют выполнить код, основываясь на результате условия (истинно оно или нет). Если условие ложное, то выполняется **оператор else**. Можно связывать несколько операторов if и else вместе.

Оператор switch обеспечивает более удобный и быстрый способ использования условий/ветвлений в коде. Он отлично сочетается с перечислениями.

Оператор goto позволяет переносить точку выполнения в программе из одного места в другое. Использовать этот оператор не рекомендуется.

Цикл while выполняет определенный код до тех пор, пока условие истинно. Сначала обрабатывается условие, а затем выполняется код.

Цикл do while — это тот же цикл while, только сначала выполняется код, а затем уже проверяется условие. Он отлично подходит для вывода меню или других элементов, так как позволяет выполнить код хотя бы один раз.

Циклы for наиболее используемые циклы. Они идеальны, когда нужно выполнить код определенное количество раз.

Оператор break позволяет немедленно завершить выполнение оператора switch, циклов while, do while или for.

Оператор continue позволяет немедленно перейти к следующей итерации цикла. Будьте осторожны при использовании этого оператора в связке с циклами while или do while — не забывайте о возникновении проблемы с инкрементом счетчика цикла.

И, наконец, **рандомные числа** позволяют получить разные результаты при выполнении одной и той же программы.

Тест

Задание №1

В итоговом тесте главы №2 мы написали программу имитации мячика, падающего с башни. Так как тогда мы еще ничего не знали о циклах и не умели их использовать, то время полета мячика составляло всего лишь 5 секунд.

Измените программу, приведенную ниже, таким образом, чтобы мячик падал ровно то количество секунд, которое необходимо ему для достижения земли.

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace myConstants
{
    const double gravity(9.8);
}
#endif
```

Основной файл:

```
#include <iostream>
#include "constants.h"

// Получаем начальную высоту от пользователя и возвращаем её
double getInitialHeight()
{
    std::cout << "Enter the initial height of the tower in meters: ";
    double initialHeight;
    std::cin >> initialHeight;
    return initialHeight;
}

// Возвращаем расстояние от земли после "..." секунд падения
double calculateHeight(double initialHeight, int seconds)
{
    // Используем формулу: [ s = u * t + (a * t^2) / 2 ], где u(начальная скорость) = 0
    double distanceFallen = (myConstants::gravity * seconds * seconds) / 2;
    double currentHeight = initialHeight - distanceFallen;

    return currentHeight;
}

// Выводим высоту, на которой находится мячик после каждой секунды падения
void printHeight(double height, int seconds)
{
    if (height > 0.0)
    {
        std::cout << "At " << seconds << " seconds, the ball is at height:\t" << height <<
            " meters\n";
```

```
    }
    else
        std::cout << "At " << seconds << " seconds, the ball is on the ground.\n";
}

void calculateAndPrintHeight(double initialHeight, int seconds)
{
    double height = calculateHeight(initialHeight, seconds);
    printHeight(height, seconds);
}

int main()
{
    const double initialHeight = getInitialHeight();

    calculateAndPrintHeight(initialHeight, 0);
    calculateAndPrintHeight(initialHeight, 1);
    calculateAndPrintHeight(initialHeight, 2);
    calculateAndPrintHeight(initialHeight, 3);
    calculateAndPrintHeight(initialHeight, 4);
    calculateAndPrintHeight(initialHeight, 5);

    return 0;
}
```

Задание №2

Напишите программу-игру типа *Hi-Lo*:

- Во-первых, ваша программа должна выбрать случайное целое число в диапазоне от 1 до 100.
- Пользователю дается 7 попыток, чтобы угадать это число.
- Если пользователь не угадал число, то программа должна подсказывать, была ли его догадка слишком большой или слишком маленькой.
- Если пользователь угадал число, то программа должна сообщить, что всё верно — вы выиграли.
- Если же у пользователя кончились попытки, и он не угадал число, то программа должна сообщить ему, что он проиграл и показать правильный результат.
- В конце игры программа должна спросить у пользователя, не хочет ли он сыграть еще раз. Если пользователь не введет ни `y`, ни `n` (а что-то другое), то программа должна спросить его еще раз.

Пример результата выполнения вашей программы:

```
Let's play a game. I'm thinking of a number. You have 7 tries to guess what it is.
Guess #1: 64
Your guess is too high.
```

```
Guess #2: 32
Your guess is too low.
Guess #3: 54
Your guess is too high.
Guess #4: 51
Correct! You win!
Would you like to play again (y/n)? y
Let's play a game. I'm thinking of a number. You have 7 tries
to guess what it is.
Guess #1: 64
Your guess is too high.
Guess #2: 32
Your guess is too low.
Guess #3: 54
Your guess is too high.
Guess #4: 51
Your guess is too high.
Guess #5: 36
Your guess is too low.
Guess #6: 45
Your guess is too low.
Guess #7: 48
Your guess is too low.
Sorry, you lose. The correct number was 49.
Would you like to play again (y/n)? q
Would you like to play again (y/n)? f
Would you like to play again (y/n)? n
Thank you for playing.
```

> ***Подсказки***:
>
> - Используйте в качестве стартового числа во время генерации случайных чисел вызов функции `time(0)`.
> - ***Пользователям Visual Studio:*** Из-за плохой реализации функции rand() (первое рандомное число не сильно отличается от стартового) - вызовите rand() сразу после установки стартового числа, чтобы сбросить первый результат.
> - Используйте функцию getRandomNumber() из урока о генерации случайных чисел для генерации случайного числа.
> - В функции, которая будет спрашивать у пользователя, не хочет ли он сыграть еще раз, используйте механизм обработки некорректного пользовательского ввода.

Урок №77. Массивы

На уроке о структурах мы узнали, что с их помощью можно объединять переменные разных типов под одним идентификатором. Это идеально, когда нужно смоделировать объект, который имеет много разных свойств. Однако удобство работы со структурами при наличии большого количества элементов оставляет желать лучшего.

Что такое массив?

К счастью, структуры не являются единственным агрегированным типом данных в языке C++. Есть еще **массив** - совокупный тип данных, который позволяет получить доступ ко всем переменным одного и того же типа данных через использование одного идентификатора.

Рассмотрим случай, когда нужно записать результаты тестов 30 студентов в классе. Без использования массива нам придется выделить почти 30 одинаковых переменных!

```
1. // Выделяем 30 целочисленных переменных (каждая с разным именем)
2. int testResultStudent1;
3. int testResultStudent2;
4. int testResultStudent3;
5. // ...
6. int testResultStudent30;
```

С использованием массива всё гораздо проще. Следующая строка эквивалентна коду, приведенному выше:

```
1. int testResult[30]; // выделяем 30 целочисленных переменных, используя
   фиксированный массив
```

В объявлении переменной массива мы используем квадратные скобки `[]`, чтобы сообщить компилятору, что это переменная массива (а не обычная переменная), а в скобках — количество выделяемых элементов (это называется **длиной** или **размером массива**).

В примере, приведенном выше, мы объявили фиксированный массив с именем `testResult` и длиной 30. **Фиксированный массив** (или ***"массив фиксированной длины"***) представляет собой массив, размер которого известен во время компиляции. При создании `testResult`, компилятор выделит 30 целочисленных переменных.

Элементы массива

Каждая из переменных в массиве называется **элементом**. Элементы не имеют своих собственных уникальных имен. Вместо этого для доступа к ним используется имя массива вместе с **оператором индекса** `[]` и параметром, который называется **индексом**, и который сообщает компилятору, какой элемент мы хотим выбрать. Этот процесс называется **индексированием массива**.

В вышеприведенном примере первым элементом в нашем массиве является `testResult[0]`, второй - `testResult[1]`, десятый - `testResult[9]`, последний - `testResult[29]`. Хорошо, что уже не нужно отслеживать и помнить кучу разных (хоть и похожих) имен переменных — для доступа к разным элементам нужно изменять только индекс.

> ***Важно:*** В отличие от повседневной жизни, отсчет в программировании и в языке C++ всегда начинается с 0, а не с 1!

В массиве длиной `N` элементы массива будут пронумерованы от `0` до `N-1`! Это называется **диапазоном массива**.

Пример программы с использованием массива

Здесь мы можем наблюдать как определение, так и индексирование массива:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int array[5]; // массив из пяти чисел
6.     array[0] = 3; // индекс первого элемента - 0 (нулевой элемент)
7.     array[1] = 2;
8.     array[2] = 4;
9.     array[3] = 8;
10.    array[4] = 12; // индекс последнего элемента - 4
11.
12.    std::cout << "The array element with the smallest index has the value " << array[0] << "\n";
13.    std::cout << "The sum of the first 5 numbers is " << array[0] + array[1] + array[2] + array[3] + array[4] << "\n";
14.
15.    return 0;
16. }
```

Результат выполнения программы:

```
The array element with the smallest index has the value 3
The sum of the first 5 numbers is 29
```

Типы данных и массивы

Массив может быть любого типа данных. Например, объявляем массив типа double:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     double array[3]; // выделяем 3 переменные типа double
6.     array[0] = 3.5;
7.     array[1] = 2.4;
8.     array[2] = 3.4;
9.
10.    std::cout << "The average is " << (array[0] + array[1] + array[2]) / 3 << "
   \n";
11.
12.    return 0;
13. }
```

Результат выполнения программы:

```
The average is 3.1
```

Массивы также можно сделать из структур, например:

```
1. struct Rectangle
2. {
3.     int length;
4.     int width;
5. };
6. Rectangle rects[4]; // объявляем массив с 4-мя прямоугольниками
```

Чтобы получить доступ к члену структуры из элемента массива, сначала нужно выбрать элемент массива, затем использовать оператор выбора члена структуры, а затем требуемый член структуры:

```
1. rects[0].length = 15;
```

Индексы массивов

В языке C++ индексы массивов всегда должны быть интегрального типа данных (т.е. типа char, short, int, long, long long, bool и т.д.). Эти индексы могут быть либо константными значениями, либо неконстантными значениями. Например:

```
1. int array[4]; // объявляем массив длиной 4
2.
3. // Используем литерал (константу) в качестве индекса
4. array[2] = 8; // хорошо
5.
6. // Используем перечисление (константу) в качестве индекса
7. enum Animals
8. {
9.     ANIMAL_CAT = 3
```

```
};
array[ANIMAL_CAT] = 5; // хорошо

// Используем переменную (не константу) в качестве индекса
short index = 4;
array[index] = 8; // хорошо
```

Объявление массивов фиксированного размера

При объявлении массива фиксированного размера, его длина (между квадратными скобками) должна быть **константой типа compile-time** (которая определяется во время компиляции). Вот несколько разных способов объявления массивов с фиксированным размером:

```
// Используем литерал
int array[7]; // хорошо

// Используем макрос-объект с текст_замена в качестве символьной константы
#define ARRAY_WIDTH 4
int array[ARRAY_WIDTH]; // синтаксически хорошо, но не делайте этого

// Используем символьную константу
const int arrayWidth = 7;
int array[arrayWidth]; // хорошо

// Используем перечислитель
enum ArrayElements
{
    MIN_ARRAY_WIDTH = 3
};
int array[MIN_ARRAY_WIDTH]; // хорошо

// Используем неконстантную переменную
int width;
std::cin >> width;
int array[width]; // плохо: width должна быть константой типа compile-time!

// Используем константную переменную типа runtime
int temp = 8;
const int width = temp;
int array[width]; // плохо: здесь width является константой типа runtime, но должна быть константой типа compile-time!
```

Обратите внимание, в двух последних случаях мы должны получить ошибку, так как длина массива не является константой типа compile-time. Некоторые компиляторы могут разрешить использование таких массивов, но они являются некорректными в соответствии со стандартами языка C++ и не должны использоваться в программах, написанных на C++.

Чуть-чуть о динамических массивах

Поскольку массивам фиксированного размера память выделяется во время компиляции, то здесь мы имеем два ограничения:

- Массивы фиксированного размера не могут иметь длину, основанную на любом пользовательском вводе или другом значении, которое вычисляется во время выполнения программы (runtime).
- Фиксированные массивы имеют фиксированную длину, которую нельзя изменить.

Во многих случаях эти ограничения являются проблематичными. К счастью, C++ поддерживает еще один тип массивов, известный как **динамический массив**. Размер такого массива может быть установлен во время выполнения программы и его можно изменить. Однако создание динамических массивов несколько сложнее фиксированных, поэтому мы поговорим об этом несколько позже.

Заключение

Фиксированные массивы обеспечивают простой способ выделения и использования нескольких переменных одного типа данных до тех пор, пока размер массива известен во время компиляции.

На следующем уроке мы рассмотрим больше тем, связанных с фиксированными массивами.

Урок №78. Фиксированные массивы

Элементы массива обрабатываются так же, как и обычные переменные, поэтому они не инициализируются при создании. Одним из способов инициализации массива является присваивание значений каждому элементу поочерёдно:

```
1. int array[5]; // массив содержит 5 простых чисел
2. array[0] = 4;
3. array[1] = 5;
4. array[2] = 8;
5. array[3] = 9;
6. array[4] = 12;
```

Однако это не совсем удобно, особенно когда массив большой.

К счастью, язык C++ поддерживает более удобный способ инициализации массивов с помощью **списка инициализаторов**. Следующий пример эквивалентен примеру выше:

```
1. int array[5] = { 4, 5, 8, 9, 12 }; // используется список инициализаторов для
   инициализации фиксированного массива
```

Если в этом списке инициализаторов больше, чем может содержать массив, то компилятор выдаст ошибку.

Однако, если в списке инициализаторов меньше, чем может содержать массив, то остальные элементы будут проинициализированы значением `0`. Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int array[5] = { 5, 7, 9 }; // инициализируем только первые 3 элемента
6.
7.     std::cout << array[0] << '\n';
8.     std::cout << array[1] << '\n';
9.     std::cout << array[2] << '\n';
10.    std::cout << array[3] << '\n';
11.    std::cout << array[4] << '\n';
12.
13.    return 0;
14. }
```

Результат выполнения программы:

```
5
7
9
0
0
```

Следовательно, чтобы инициализировать все элементы массива значением `0`, нужно:

```
1. // Инициализируем все элементы массива значением 0
2. int array[5] = { };
```

В C++11 вместо этого мы можем воспользоваться синтаксисом uniform-инициализации:

```
1. int array[5] { 4, 5, 8, 9, 12 }; // используем uniform-инициализацию для
   инициализации фиксированного массива
```

Длина массива

Если вы инициализируете фиксированный массив с помощью списка инициализаторов, то компилятор может определить длину массива вместо вас, и вам уже не потребуется её объявлять.

Следующие две строки выполняют одно и то же:

```
1. int array[5] = { 0, 1, 2, 3, 4 }; // явно указываем длину массива
2. int array[] = { 0, 1, 2, 3, 4 }; // список инициализаторов автоматически
   определит длину массива
```

Это не только сэкономит время, но также вам не придется обновлять длину массива, если вы захотите добавить или удалить элементы позже.

Массивы и перечисления

Одна из основных проблем при использовании массивов состоит в том, что целочисленные индексы не предоставляют никакой информации программисту об их значении. Рассмотрим класс из 5 учеников:

```
1. const int numberOfStudents(5);
2. int testScores[numberOfStudents];
3. testScores[3] = 65;
```

Кто представлен элементом `testScores[3]`? Непонятно!

Это можно решить, используя перечисление, в котором перечислители сопоставляются каждому из возможных индексов массива:

```
1. enum StudentNames
2. {
3.     SMITH, // 0
4.     ANDREW, // 1
5.     IVAN, // 2
6.     JOHN, // 3
7.     ANTON, // 4
```

```
8.      MAX_STUDENTS // 5
9.  };
10.
11. int main()
12. {
13.     int testScores[MAX_STUDENTS]; // всего 5 студентов
14.     testScores[JOHN] = 65;
15.
16.     return 0;
17. }
```

Вот теперь понятно, что представляет собой каждый из элементов массива. Обратите внимание, добавлен дополнительный перечислитель с именем `MAX_STUDENTS`. Он используется во время объявления массива для гарантирования того, что массив имеет корректную длину (она должна быть на единицу больше самого большого индекса). Это полезно как для подсчета элементов, так и для возможности автоматического изменения длины массива, если добавить еще один перечислитель:

```
1.  enum StudentNames
2.  {
3.      SMITH, // 0
4.      ANDREW, // 1
5.      IVAN, // 2
6.      JOHN, // 3
7.      ANTON, // 4
8.      MISHA, // 5
9.      MAX_STUDENTS // 6
10. };
11.
12. int main()
13. {
14.     int testScores[MAX_STUDENTS]; // всего 6 студентов
15.     testScores[JOHN] = 65; // всё работает
16.
17.     return 0;
18. }
```

Обратите внимание, этот трюк работает только в том случае, если вы не изменяете значения перечислителей вручную!

Массивы и классы enum

Классы enum не имеют неявного преобразования в целочисленный тип, поэтому, если вы попробуете сделать следующее:

```
1.  enum class StudentNames
2.  {
3.      SMITH, // 0
4.      ANDREW, // 1
5.      IVAN, // 2
6.      JOHN, // 3
7.      ANTON, // 4
8.      MISHA, // 5
```

```
    MAX_STUDENTS // 6
};

int main()
{
    int testScores[StudentNames::MAX_STUDENTS]; // всего 6 студентов
    testScores[StudentNames::JOHN] = 65;
}
```

То получите ошибку от компилятора. Это можно решить, используя оператор static_cast для конвертации перечислителя в целое число:

```
int main()
{
    int testScores[static_cast<int>(StudentNames::MAX_STUDENTS)]; // всего 6 студентов
    testScores[static_cast<int>(StudentNames::JOHN)] = 65;
}
```

Однако, это также не очень удобно, поэтому лучше использовать стандартное перечисление внутри пространства имен:

```
namespace StudentNames
{
    enum StudentNames
    {
        SMITH, // 0
        ANDREW, // 1
        IVAN, // 2
        JOHN, // 3
        ANTON, // 4
        MISHA, // 5
        MAX_STUDENTS // 6
    };
}

int main()
{
    int testScores[StudentNames::MAX_STUDENTS]; // всего 6 студентов
    testScores[StudentNames::JOHN] = 65;
}
```

Передача массивов в функции

Хотя передача массива в функцию на первый взгляд выглядит так же, как передача обычной переменной, но "под капотом" C++ обрабатывает массивы несколько иначе.

Когда обычная переменная передается по значению, то C++ копирует значение аргумента в параметр функции. Поскольку параметр является копией, то изменение значения параметра не изменяет значение исходного аргумента.

Однако, поскольку копирование больших массивов - дело трудоёмкое, то C++ не копирует массив при его передаче в функцию. Вместо этого передается фактический массив. И здесь мы получаем побочный эффект, позволяющий функциям напрямую изменять значения элементов массива!

Следующий пример хорошо иллюстрирует эту концепцию:

```
#include <iostream>

void passValue(int value) // здесь value - это копия аргумента
{
    value = 87; // изменения value здесь не повлияют на фактическую переменную value
}

void passArray(int array[5]) // здесь array - это фактический массив
{
    array[0] = 10; // изменения array здесь изменят исходный массив array
    array[1] = 8;
    array[2] = 6;
    array[3] = 4;
    array[4] = 1;
}

int main()
{
    int value = 1;
    std::cout << "before passValue: " << value << "\n";
    passValue(value);
    std::cout << "after passValue: " << value << "\n";

    int array[5] = { 1, 4, 6, 8, 10 };
    std::cout << "before passArray: " << array[0] << " " << array[1] << " " << array[2] << " " << array[3] << " " << array[4] << "\n";
    passArray(array);
    std::cout << "after passArray: " << array[0] << " " << array[1] << " " << array[2] << " " << array[3] << " " << array[4] << "\n";

    return 0;
}
```

Результат выполнения программы:

```
before passValue: 1
after passValue: 1
before passArray: 1 4 6 8 10
after passArray: 10 8 6 4 1
```

В примере, приведенном выше, значение переменной `value` не изменяется в функции main(), так как параметр `value` в функции passValue() был лишь копией фактической переменной `value`. Однако, поскольку массив в параметре функции passArray() является фактическим массивом, то passArray() напрямую изменяет значения его элементов!

***Примечание*:** Если вы не хотите, чтобы функция изменяла значения элементов массива, переданного в нее в качестве параметра, то нужно сделать массив константным:

```
1. // Даже если array является фактическим массивом, внутри этой функции он должен рассматриваться как константный
2. void passArray(const int array[5])
3. {
4.     // Поэтому каждая из следующих строк вызовет ошибку компиляции!
5.     array[0] = 11;
6.     array[1] = 7;
7.     array[2] = 5;
8.     array[3] = 3;
9.     array[4] = 2;
10. }
```

Оператор sizeof и массивы

Оператор sizeof можно использовать и с массивами: он возвращает общий размер массива (длина массива умножена на размер одного элемента) в байтах. Обратите внимание, из-за того, как C++ передает массивы в функции, следующая операция не будет корректно выполнена с массивами, переданными в функции:

```
1. #include <iostream>
2.
3. void printSize(int array[])
4. {
5.     std::cout << sizeof(array) << '\n'; // выводится размер указателя (об этом поговорим на соответствующем уроке), а не массива
6. }
7.
8. int main()
9. {
10.     int array[] = { 1, 3, 3, 4, 5, 9, 14, 17 };
11.     std::cout << sizeof(array) << '\n'; // выводится размер массива
12.     printSize(array);
13.
14.     return 0;
15. }
```

Результат выполнения программы:

```
32
4
```

По этой причине будьте осторожны при использовании оператора sizeof с массивами!

Определение длины фиксированного массива

Чтобы определить длину фиксированного массива, поделите размер всего массива на размер одного элемента массива:

```
#include <iostream>

int main()
{
    int array[] = { 1, 3, 3, 4, 5, 9, 14, 17 };
    std::cout << "The array has: " << sizeof(array) / sizeof(array[0]) << " elements\n";

    return 0;
}
```

Результат выполнения программы:

```
The array has 8 elements
```

Как это работает? Во-первых, размер всего массива равен длине массива, умноженной на размер одного элемента. Формула: `размер массива = длина массива * размер одного элемента`.

Используя алгебру, мы можем изменить это уравнение: `длина массива = размер массива / размер одного элемента`. `sizeof(array)` - это размер массива, а `sizeof(array [0])` - это размер одного элемента массива. Соответственно, `длина массива = sizeof(array) / sizeof(array[0])`. Обычно используется нулевой элемент в качестве элемента массива в уравнении, так как только он является единственным элементом, который гарантированно существует в массиве, независимо от его длины.

Это работает только если массив фиксированной длины, и вы выполняете эту операцию в той же функции, в которой объявлен массив (мы поговорим больше об этом ограничении на следующих уроках).

На следующих уроках мы будем использовать термин «длина» для обозначения общего количества элементов в массиве, и термин «размер», когда речь будет идти о байтах.

Индексирование массива вне диапазона

Помните, что массив длиной `N` содержит элементы от `0` до `N-1`. Итак, что произойдет, если мы попытаемся получить доступ к индексу массива за пределами этого диапазона?

Рассмотрим следующую программу:

```
1. int main()
2. {
3.     int array[5]; // массив содержит 5 простых чисел
4.     array[5] = 14;
5.
6.     return 0;
7. }
```

Здесь наш массив имеет длину 5, но мы пытаемся записать значение в 6-й элемент (индекс 5).

Язык C++ не выполняет никаких проверок корректности вашего индекса. Таким образом, в вышеприведенном примере значение `14` будет помещено в ячейку памяти, где 6-й элемент существовал бы (если бы вообще был). Но, как вы уже догадались, это будет иметь свои последствия. Например, произойдет перезаписывание значения другой переменной или вообще сбой программы.

Хотя это происходит реже, но C++ также позволяет использовать отрицательный индекс, что тоже приведет к нежелательным результатам.

Правило: При использовании массивов убедитесь, что ваши индексы корректны и соответствуют диапазону вашего массива.

Тест

Задание №1

Объявите массив для хранения температуры (дробное число) каждого дня в году (всего 365 дней). Проинициализируйте массив значением `0.0` для каждого дня.

Задание №2

Создайте перечисление со следующими перечислителями: `chicken`, `lion`, `giraffe`, `elephant`, `duck` и `snake`. Поместите перечисление в пространство имен. Объявите массив, где элементами будут эти перечислители и, используя список инициализаторов, инициализируйте каждый элемент соответствующим количеством лап определенного животного. В функции main() выведите количество ног у слона, используя перечислитель.

Урок №79. Массивы и циклы

Рассмотрим случай, когда нужно вычислить средний балл всех студентов в группе. Используя отдельные переменные:

```
1. const int numStudents = 5;
2. int student0 = 73;
3. int student1 = 85;
4. int student2 = 84;
5. int student3 = 44;
6. int student4 = 78;
7.
8. int totalScore = student0 + student1 + student2 + student3 + student4;
9. double averageScore = static_cast<double>(totalScore) / numStudents;
```

Мы получим много объявлений переменных и, следовательно, много кода - а это всего лишь 5 студентов! А представьте, если бы их было 30 или 150.

Кроме того, чтобы добавить нового студента, нам придется объявить новую переменную, инициализировать её и добавить в переменную `totalScore`. И это всё вручную. А каждый раз при изменении старого кода есть риск наделать новых ошибок. А вот с использованием массива:

```
1. const int numStudents = 5;
2. int students[numStudents] = { 73, 85, 84, 44, 78};
3. int totalScore = students[0] + students[1] + students[2] + students[3] +
   students[4];
4. double averageScore = static_cast<double>(totalScore) / numStudents;
```

Количество объявленных переменных сократится, но в `totalScore` по-прежнему придется заносить каждый элемент массива вручную. И, как указано выше, изменение количества студентов означает, что формулу `totalScore` необходимо будет изменять также вручную.

Если бы был только способ автоматизировать этот процесс.

Циклы и массивы

Из предыдущего урока мы уже знаем, что индекс массива не обязательно должен быть константным значением - он может быть и обычной переменной. Это означает, что мы можем использовать счетчик цикла в качестве индекса массива для доступа к элементам и выполнения с ними необходимых математических и других операций. Это настолько распространенная практика, что почти всегда при обнаружении массива, вы найдете рядом с ним цикл! Когда цикл используется для доступа к каждому элементу массива поочередно, то это называются **итерацией по массиву**.

Например:

```
int students[] = { 73, 85, 84, 44, 78};
const int numStudents = sizeof(students) / sizeof(students[0]);
int totalScore = 0;

// Используем цикл для вычисления totalScore
for (int person = 0; person < numStudents; ++person)
    totalScore += students[person];

double averageScore = static_cast<double>(totalScore) / numStudents;
```

Это решение идеально подходит как в плане удобства и чтения, так и поддержки. Поскольку доступ к каждому элементу массива выполняется через цикл, то формула подсчета суммы всех значений автоматически настраивается с учетом количества элементов в массиве. И для вычисления средней оценки нам уже не нужно будет вручную добавлять новых студентов и индексы новых элементов массива!

А вот пример использования цикла для поиска в массиве наибольшего значения (наилучшей оценки среди всех студентов в группе):

```
#include <iostream>

int main()
{
    int students[] = { 73, 85, 84, 44, 78};
    const int numStudents = sizeof(students) / sizeof(students[0]);

    int maxScore = 0; // отслеживаем самую высокую оценку
    for (int person = 0; person < numStudents; ++person)
        if (students[person] > maxScore)
            maxScore = students[person];

    std::cout << "The best score was " << maxScore << '\n';

    return 0;
}
```

Здесь уже используется переменная `maxScore` (не из цикла) для отслеживания самого большого значения массива. Сначала инициализируем `maxScore` значением `0`, что означает, что мы еще не видели никаких оценок. Затем перебираем каждый элемент массива и, если находим оценку, которая выше предыдущей, присваиваем её значение переменной `maxScore`. Таким образом, `maxScore` всегда будет хранить наибольшее значение из всех элементов массива.

Использование циклов с массивами

Циклы с массивами обычно используются для выполнения одной из 3-х следующих задач:

- Вычислить значение (например, среднее или сумму всех значений).
- Найти значение (например, самое большое или самое маленькое).
- Отсортировать элементы массива (например, по возрастанию или по убыванию).

При вычислении значения, переменная обычно используется для хранения промежуточного результата, который необходим для вычисления конечного значения. В примере, приведенном выше, где мы вычисляем средний балл, переменная `totalScore` содержит сумму значений всех рассмотренных элементов.

При поиске значения, переменная обычно используется для хранения наилучшего варианта (или индекса наилучшего варианта) из всех просмотренных. В примере, приведенном выше, где мы используем цикл для поиска наивысшей оценки, переменная `maxScore` используется для хранения наибольшего количества баллов из просмотренных ранее элементов массива.

Сортировка массива происходит несколько сложнее, так как в этом деле используются вложенные циклы (но об этом уже на следующем уроке).

Массивы и «ошибка неучтенной единицы»

Одной из самых сложных задач при использовании циклов с массивами является убедиться, что цикл выполняется правильное количество раз. **Ошибку на единицу** (или ***«ошибку неучтенной единицы»***) сделать легко, а попытка получить доступ к элементу, индекс которого больше, чем длина массива, может иметь самые разные последствия. Рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int students[] = { 73, 85, 84, 44, 78 };
6.     const int numStudents = sizeof(students) / sizeof(students[0]);
7.
8.     int maxScore = 0; // отслеживаем самую высокую оценку
9.     for (int person = 0; person <= numStudents; ++person)
10.        if (students[person] > maxScore)
11.            maxScore = students[person];
12.
13.    std::cout << "The best score was " << maxScore << '\n';
```

```
14.
15.     return 0;
16. }
```

Здесь проблема состоит в неверном условии оператора if в цикле for! Объявленный массив содержит 5 элементов, проиндексированных от 0 до 4. Однако цикл внутри перебирает элементы от 0 до 5. Следовательно, на последней итерации в цикле for выполнится:

```
1. if (students[5] > maxScore)
2.     maxScore = students[5];
```

Но ведь `students[5]` не определен! Его значением, скорее всего, будет простой мусор. И в итоге результатом выполнения цикла может быть ошибочный `maxScore`.

Однако представьте, что бы произошло, если бы мы ненароком присвоили значение элементу `students[5]`! Мы бы могли перезаписать другую переменную (или её часть) или испортить что-либо - эти типы ошибок очень трудно отследить!

Следовательно, при использовании циклов с массивами, всегда перепроверяйте условия в циклах, чтобы убедиться, что их выполнение не приведет к ошибке неучтенной единицы.

Тест

Задание №1

Выведите на экран следующий массив с помощью цикла:

```
1. int array[] = { 7, 5, 6, 4, 9, 8, 2, 1, 3 };
```

> ***Подсказка:*** Используйте трюк с sizeof (из предыдущего урока) для определения длины массива.

Задание №2

Используя массив из задания №1:

Попросите пользователя ввести число от 1 до 9. Если пользователь введет что-либо другое - попросите его снова ввести число и так до тех пор, пока он не введет корректное значение из заданного диапазона. Как только пользователь введет число от 1 до 9, выведите массив на экран. Затем найдите в массиве элемент с числом, которое ввел пользователь, и выведите его индекс.

Для обработки некорректного ввода используйте следующий код:

```
// Если пользователь ввел некорректное значение
if (std::cin.fail())
{
    std::cin.clear();
    std::cin.ignore(32767, '\n');
}
```

Задание №3

Измените следующую программу так, чтобы вместо `maxScore` с наибольшим значением, переменная `maxIndex` содержала индекс элемента с наибольшим значением:

```
#include <iostream>

int main()
{
    int students[] = { 73, 85, 84, 44, 78};
    const int numStudents = sizeof(students) / sizeof(students[0]);

    int maxScore = 0; // отслеживаем самую высокую оценку

    for (int person = 0; person < numStudents; ++person)
        if (students[person] > maxScore)
            maxScore = students[person];

    std::cout << "The best score was " << maxScore << '\n';

    return 0;
}
```

Урок №80. Сортировка массивов методом выбора

Сортировка массива - это процесс распределения всех элементов массива в определенном порядке. Очень часто это бывает полезным. Например, в вашем почтовом ящике электронные письма отображаются в зависимости от времени получения; новые письма считаются более релевантными, чем те, которые вы получили полчаса, час, два или день назад; когда вы переходите в свой список контактов, имена обычно находятся в алфавитном порядке, потому что так легче что-то найти. Все эти случаи включают в себя сортировку данных перед их фактическим выводом.

Как работает сортировка?

Сортировка данных может сделать поиск внутри массива более эффективным не только для людей, но и для компьютеров. Например, рассмотрим случай, когда нам нужно узнать, отображается ли определенное имя в списке имен. Чтобы это узнать, нужно проверить каждый элемент массива на соответствие нашему значению. Поиск в массиве с множеством элементов может оказаться слишком неэффективным (затратным).

Однако, предположим, что наш массив с именами отсортирован в алфавитном порядке. Тогда наш поиск начинается с первой буквы нашего значения и заканчивается буквой, которая идет следующей по алфавиту. В таком случае, если мы дошли до этой буквы и не нашли имя, то точно знаем, что оно не находится в остальной части массива, так как в алфавитном порядке нашу букву мы уже прошли!

Не секрет, что есть алгоритмы поиска внутри отсортированных массивов и получше. Применяя простой алгоритм, мы можем искать определенный элемент в отсортированном массиве, содержащем 1 000 000 элементов, используя всего лишь 20 сравнений! Недостатком, конечно же, является то, что сортировка массива с таким огромным количеством элементов - дело сравнительно затратное, и оно точно не выполняется ради одного поискового запроса.

В некоторых случаях сортировка массива делает поиск ненужным. Например, мы ищем наилучший результат прохождения теста среди студентов. Если массив не отсортирован, то нам придется просмотреть каждый элемент массива, чтобы найти наивысшую оценку. Если же массив отсортирован, то наивысшая оценка будет находиться либо на первой позиции, либо на последней (в зависимости от метода сортировки массива: в порядке возрастания или в порядке убывания), поэтому нам не нужно искать вообще!

Сортировка обычно выполняется путем повторного сравнения пар элементов массива и замены значений, если они отвечают заданным критериям. Порядок, в котором эти элементы сравниваются, зависит от того, какой алгоритм сортировки используется. Критерии определяют, как будет сортироваться массив (например, в порядке возрастания или в порядке убывания).

Чтобы поменять два элемента местами, мы можем использовать **функцию std::swap()** из Стандартной библиотеки C++, которая определена в заголовочном файле algorithm. В C++11 функция std::swap() была перенесена в заголовочный файл utility:

```
#include <iostream>
#include <algorithm> // для std::swap. В C++11 используйте заголовок <utility>

int main()
{
    int a = 3;
    int b = 5;
    std::cout << "Before swap: a = " << a << ", b = " << b << '\n';
    std::swap(a, b); // меняем местами значения переменных a и b
    std::cout << "After swap:  a = " << a << ", b = " << b << '\n';
}
```

Результат выполнения программы:

```
Before swap: a = 3, b = 5
After swap: a = 5, b = 3
```

После выполнения операции замены значения переменных `a` и `b` поменялись местами.

Сортировка массивов методом выбора

Существует множество способов сортировки массивов. Сортировка массивов методом выбора, пожалуй, самая простая для понимания, хотя и одна из самых медленных.

Для **сортировки массива методом выбора от наименьшего до наибольшего элемента** выполняются следующие шаги:

- Начиная с элемента под индексом 0, ищем в массиве наименьшее значение.
- Найденное значение меняем местами с нулевым элементом.
- Повторяем шаги №1 и №2 уже для следующего индекса в массиве (отсортированный элемент больше не трогаем).

Другими словами, мы ищем наименьший элемент в массиве и перемещаем его на первое место. Затем ищем второй наименьший элемент и перемещаем его уже на второе место после первого наименьшего элемента. Этот процесс продолжается до тех пор, пока в массиве не закончатся неотсортированные элементы.

Вот пример работы этого алгоритма в массиве с 5-ю элементами:

```
{ 30, 50, 20, 10, 40 }
```

Сначала ищем наименьший элемент, начиная с индекса 0:

```
{ 30, 50, 20, 10, 40 }
```

Затем меняем местами наименьший элемент с элементом под индексом 0:

```
{ 10, 50, 20, 30, 40 }
```

Теперь, когда первый элемент массива отсортирован, мы его игнорируем. Ищем следующий наименьший элемент, но уже начиная с индекса 1:

```
{ 10, 50, 20, 30, 40 }
```

И меняем его местами с элементом под индексом 1:

```
{ 10, 20, 50, 30, 40 }
```

Теперь мы игнорируем первые два элемента. Ищем следующий наименьший элемент, начиная с индекса 2:

```
{ 10, 20, 50, 30, 40 }
```

И меняем его местами с элементом под индексом 2:

```
{ 10, 20, 30, 50, 40 }
```

Ищем следующий наименьший элемент, начиная с индекса 3:

```
{ 10, 20, 30, 50, 40 }
```

И меняем его местами с элементом под индексом 3:

```
{ 10, 20, 30, 40, 50 }
```

Ищем следующий наименьший элемент, начиная с индекса 4:

```
{ 10, 20, 30, 40, 50 }
```

И меняем его местами с элементом под индексом 4 (выполняется самозамена, т.е. ничего не делаем):

```
{ 10, 20, 30, 40 50 }
```

Готово!

```
{ 10, 20, 30, 40, 50 }
```

Обратите внимание, последнее сравнение всегда будет одиночным (т.е. самозамена), что является лишней операцией, поэтому, фактически, мы можем остановить выполнение сортировки перед последним элементом массива.

Сортировка массивов методом выбора в C++

Вот как этот алгоритм реализован в C++:

```
1. #include <iostream>
2. #include <algorithm> // для std::swap. В C++11 используйте заголовок <utility>
3. 
4. int main()
5. {
6.     const int length = 5;
7.     int array[length] = { 30, 50, 20, 10, 40 };
8. 
9.     // Перебираем каждый элемент массива (кроме последнего, он уже будет отсортирован к тому времени, когда мы до него доберемся)
10.     for (int startIndex = 0; startIndex < length - 1; ++startIndex)
11.     {
12.         // В переменной smallestIndex хранится индекс наименьшего значения, которое мы нашли в этой итерации.
13.         // Начинаем с того, что наименьший элемент в этой итерации – это первый элемент (индекс 0)
14.         int smallestIndex = startIndex;
15. 
16.         // Затем ищем элемент поменьше в остальной части массива
17.         for (int currentIndex = startIndex + 1; currentIndex < length; ++currentIndex)
18.         {
19.             // Если мы нашли элемент, который меньше нашего наименьшего элемента,
20.             if (array[currentIndex] < array[smallestIndex])
21.                 // то запоминаем его
22.                 smallestIndex = currentIndex;
23.         }
24. 
25.         // smallestIndex теперь наименьший элемент.
26.                 // Меняем местами наше начальное наименьшее число с тем, которое мы обнаружили
27.         std::swap(array[startIndex], array[smallestIndex]);
28.     }
29. 
30.     // Теперь, когда весь массив отсортирован - выводим его на экран
31.     for (int index = 0; index < length; ++index)
```

```
32.         std::cout << array[index] << ' ';
33.
34.     return 0;
35. }
```

Наиболее запутанной частью этого алгоритма является цикл внутри другого цикла (так называемый "вложенный цикл"). Внешний цикл (`startIndex`) перебирает элементы один за другим (поочередно). В каждой итерации внешнего цикла внутренний цикл (`currentIndex`) используется для поиска наименьшего элемента среди элементов, которые остались в массиве (начиная со `startIndex + 1`). `smallestIndex` отслеживает индекс наименьшего элемента, найденного внутренним циклом. Затем `smallestIndex` меняется значением с `startIndex`. И, наконец, внешний цикл (`startIndex`) переходит к следующему индексу массива, и процесс повторяется.

> ***Подсказка:*** Если у вас возникли проблемы с пониманием того, как работает программа, приведенная выше, то попробуйте записать её выполнение на листке бумаги. Запишите начальные (неотсортированные) элементы массива горизонтально в строке в верхней части листа. Нарисуйте стрелки, указывающие на то, какими элементами являются `startIndex`, `currentIndex` и `smallestIndex` на данный момент. Прокрутите выполнение программы вручную и перерисуйте стрелки по мере изменения индексов. После каждой итерации внешнего цикла нарисуйте новую строку, показывающую текущее состояние массива (расположение его элементов).

Сортировка текста выполняется с помощью того же алгоритма. Просто измените тип массива с int на std::string и инициализируйте его с помощью соответствующих значений.

Функция std::sort()

Поскольку операция сортировки массивов очень распространена, то Стандартная библиотека C++ предоставляет встроенную **функцию сортировки std::sort()**. Она находится в заголовочном файле algorithm и вызывается следующим образом:

```
1. #include <iostream>
2. #include <algorithm> // для std::sort()
3.
4. int main()
5. {
6.     const int length = 5;
7.     int array[length] = { 30, 50, 20, 10, 40 };
8.
9.     std::sort(array, array+length);
10.
11.    for (int i=0; i < length; ++i)
```

```
12.         std::cout << array[i] << ' ';
13.
14.     return 0;
15. }
```

Тест

Задание №1

Напишите на листке бумаги выполнение сортировки следующего массива методом выбора (так, как мы это делали выше):

```
{30, 60, 20, 50, 40, 10}
```

Задание №2

Перепишите код программы из подзаголовка "Сортировка массивов методом выбора в C++" так, чтобы сортировка выполнялась в порядке убывания (от наибольшего числа к наименьшему). Хотя это может показаться сложным на первый взгляд, но на самом деле это очень просто.

Задание №3

Это задание уже немного сложнее.

Еще одним простым методом сортировки элементов является **«сортировка пузырьком»** (или ***"пузырьковая сортировка"***). Суть заключается в сравнении пары значений, которые находятся рядом, и, если удовлетворены заданные критерии, значения из этой пары меняются местами. И таким образом элементы «скачут пузырьком» до конца массива. Хотя есть несколько способов оптимизировать сортировку пузырьком, в этом задании мы будем придерживаться неоптимизированной версии, так как она проще.

При неоптимизированной версии сортировки пузырьком выполняются следующие шаги для **сортировки массива от наименьшего до наибольшего значения**:

- Сравнивается элемент массива под индексом 0 с элементом массива под индексом 1. Если элемент под индексом 0 больше элемента под индексом 1, то значения меняются местами.
- Затем мы перемещаемся к следующей паре значений: элемент под индексом 1 и элемент под индексом 2 и так до тех пор, пока не достигнем конца массива.

- Повторяем шаг №1 и шаг №2 до тех пор, пока весь массив не будет отсортирован.

Напишите программу, которая отсортирует следующий массив сортировкой пузырьком в соответствии с правилами, указанными выше:

```
1. const int length(9);
2. int array[length] = { 7, 5, 6, 4, 9, 8, 2, 1, 3 };
```

В конце программы выведите отсортированные элементы массива.

> ***Подсказка:*** Если мы можем отсортировать только один элемент за одну итерацию, то это означает, что нам нужно будет повторить выполнение цикла столько раз, сколько есть чисел в нашем массиве (его длина), дабы гарантировать выполнение сортировки всего массива.

Задание №4

Реализуйте следующие два решения оптимизации алгоритма сортировки пузырьком, который вы написали в предыдущем задании:

- Обратите внимание, с каждым выполнением сортировки пузырьком наибольшее значения в массиве "пузырится" до конца. После первой итерации последний элемент массива уже отсортирован. После второй итерации отсортирован предпоследний элемент массива и т.д. С каждой новой итерацией нам не нужно перепроверять элементы, которые уже были отсортированы. Измените свой цикл соответствующим образом.
- Если на протяжении всей итерации не выполнится ни одной замены, то мы знаем, что массив уже отсортирован. Внедрите проверку того, были ли сделаны какие-либо замены в текущей итерации, и, если нет — завершите выполнение цикла. Если цикл был завершен, то выведите информацию о том, на какой итерации сортировка элементов завершилась.

Пример результата выполнения вашей программы:

```
Early termination on iteration: 8
1 2 3 4 5 6 7 8 9
```

Урок №81. Многомерные массивы

Элементы массива могут быть любого типа данных, даже массива!

Многомерные массивы

Массив массивов называется **многомерным массивом**:

```
1. int array[2][4]; // 2-элементный массив из 4-элементных массивов
```

Поскольку у нас есть 2 индекса, то это **двумерный массив**.

В двумерном массиве первый (левый) индекс принято читать как количество строк, а второй (правый) как количество столбцов. Массив выше можно представить следующим образом:

```
[0][0]  [0][1]  [0][2]  [0][3]  // строка №0
[1][0]  [1][1]  [1][2]  [1][3]  // строка №1
```

Чтобы получить доступ к элементам двумерного массива, просто используйте два индекса:

```
1. array[1][3] = 7; // без приставки int (типа данных)
```

Инициализация двумерных массивов

Для инициализации двумерного массива проще всего использовать вложенные фигурные скобки, где каждый набор значений соответствует определенной строке:

```
1. int array[3][5] =
2. {
3. { 1, 2, 3, 4, 5 }, // строка №0
4. { 6, 7, 8, 9, 10 }, // строка №1
5. { 11, 12, 13, 14, 15 } // строка №2
6. };
```

Хотя некоторые компиляторы могут позволить вам упустить внутренние фигурные скобки, все же рекомендуется указывать их в любом случае: улучшается читабельность и уменьшается вероятность получения незапланированных нулевых элементов массива из-за того, что C++ заменяет отсутствующие инициализаторы значением `0`:

```
1. int array[3][5] =
2. {
3. { 2, 4  }, // строка №0 = 2, 4, 0, 0, 0
4. { 1, 3, 7 }, // строка №1 = 1, 3, 7, 0, 0
5. { 8, 9, 11, 12 } // строка №2 = 8, 9, 11, 12, 0
```

```
6. };
```

В двумерном массиве со списком инициализаторов можно не указывать только левый индекс (длину массива):

```
1. int array[][5] =
2. {
3. { 1, 2, 3, 4, 5 },
4. { 6, 7, 8, 9, 10 },
5. { 11, 12, 13, 14, 15 }
6. };
```

Компилятор может сам вычислить количество строк в массиве. Однако не указывать два индекса — это уже ошибка:

```
1. int array[][] =
2. {
3. { 3, 4, 7, 8 },
4. { 1, 2, 6, 9 }
5. };
```

Подобно обычным массивам, многомерные массивы можно инициализировать значением `0` следующим образом:

```
1. int array[3][5] = { 0 };
```

Обратите внимание, это работает только в том случае, если вы явно объявляете длину массива (указываете левый индекс)! В противном случае, вы получите двумерный массив с 1 строкой.

Доступ к элементам в двумерном массиве

Для доступа ко всем элементам двумерного массива требуется два цикла: один для строк и один для столбцов. Поскольку доступ к двумерным массивам обычно выполняется по строкам, то левый индекс используется в качестве внешнего цикла:

```
1. for (int row = 0; row < numRows; ++row) // доступ по строкам
2.     for (int col = 0; col < numCols; ++col) // доступ к каждому элементу в
   строке
3.         std::cout << array[row][col];
```

Многомерные массивы более двух измерений

Многомерные массивы могут быть более двух измерений. Например, объявление трехмерного массива:

```
1. int array[4][3][2];
```

Трехмерные массивы трудно инициализировать любым интуитивным способом с использованием списка инициализаторов, поэтому лучше инициализировать весь массив значением `0` и явно присваивать элементам значения с помощью вложенных циклов.

Доступ к элементам трехмерного массива осуществляется так же, как и к элементам двумерного массива:

```
1. std::cout << array[3][2][1];
```

Пример двумерного массива

Рассмотрим пример использования двумерного массива:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     // Объявляем массив 10x10
6.     const int numRows = 10;
7.     const int numCols = 10;
8.     int product[numRows][numCols] = { 0 };
9.
10.    // Создаем таблицу умножения
11.    for (int row = 0; row < numRows; ++row)
12.        for (int col = 0; col < numCols; ++col)
13.            product[row][col] = row * col;
14.
15.    // Выводим таблицу умножения
16.    for (int row = 1; row < numRows; ++row)
17.    {
18.        for (int col = 1; col < numCols; ++col)
19.            std::cout << product[row][col] << "\t";
20.
21.        std::cout << '\n';
22.    }
23.
24.    return 0;
25. }
```

Эта программа вычисляет и выводит таблицу умножения от 1 до 9 (включительно). Обратите внимание, при выводе таблицы в цикле for мы начинаем с 1 вместо 0. Это делается с целью предотвращения вывода нулевой строки и нулевого столбца, содержащих одни нули!

Результат выполнения программы:

```
1    2    3    4    5    6    7    8    9
2    4    6    8    10   12   14   16   18
3    6    9    12   15   18   21   24   27
4    8    12   16   20   24   28   32   36
```

```
5    10   15   20   25   30   35   40   45
6    12   18   24   30   36   42   48   54
7    14   21   28   35   42   49   56   63
8    16   24   32   40   48   56   64   72
9    18   27   36   45   54   63   72   81
```

Двумерные массивы обычно используются в играх типа *tile-based*, где каждый элемент массива представляет собой один фрагмент/плитку. Они также используются в компьютерной 3D-графике (в виде *матриц*) для вращения, масштабирования и отражения фигур.

Урок №82. Строки C-style

Ранее мы определили термин «строка», как набор последовательных символов (например, `Hello, world!`). Строки - это основной способ работы с текстом в языке C++, а std::string упрощает этот способ работы.

Современный C++ поддерживает два разных типа строк:

- **std::string** (как часть Стандартной библиотеки C++);
- **строки C-style** (изначально унаследованные от языка Си).

std::string реализован с помощью строк C-style.

Строки C-style

Строка C-style - это простой массив символов, который использует нуль-терминатор. **Нуль-терминатор** - это специальный символ (ASCII-код которого равен 0), используемый для обозначения конца строки. Строка C-style еще называется **"нуль-терминированной строкой"**.

Для её определения нужно просто объявить массив типа char и инициализировать его литералом (например, `string`):

```
1. char mystring[] = "string";
```

Хотя `string` имеет только 6 букв, C++ автоматически добавляет нуль-терминатор в конец строки (нам не нужно добавлять его вручную). Следовательно, длина массива `mystring` на самом деле равна 7!

В качестве примера рассмотрим следующую программу, которая выводит длину строки, а затем ASCII-коды всех символов литерала `string`:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     char mystring[] = "string";
6.     std::cout << mystring << " has " << sizeof(mystring) << " characters.\n";
7.     for (int index = 0; index < sizeof(mystring); ++index)
8.         std::cout << static_cast<int>(mystring[index]) << " ";
9.
10.    return 0;
11. }
```

Результат выполнения программы:

```
string has 7 characters.
115 116 114 105 110 103 0
```

Нуль в конце является ASCII-кодом нуль-терминатора, который был добавлен в конец строки.

При таком объявлении строк рекомендуется использовать квадратные скобки `[]`, чтобы позволить компилятору определить длину массива самому. Таким образом, если вы измените строку позже, вам не придется вручную изменять значение длины массива.

Важно отметить, что строки C-style следуют всем тем же правилам, что и массивы. Это означает, что вы можете инициализировать строку при создании, но после этого не сможете присваивать ей значения с помощью оператора присваивания:

```
1. char mystring[] = "string"; // ок
2. mystring = "cat"; // не ок!
```

Это то же самое, как если бы мы сделали следующее:

```
1. int array[] = { 4, 6, 8, 2 }; // ок
2. array = 7; // что это значит?
```

Поскольку строки C-style являются массивами, то вы можете использовать оператор `[]` для изменения отдельных символов в строке:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     char mystring[] = "string";
6.     mystring[1] = 'p';
7.     std::cout << mystring;
8.
9.     return 0;
10.}
```

Результат выполнения программы:

```
spring
```

При выводе строки C-style объект std::cout выводит символы до тех пор, пока не встретит нуль-терминатор. Если бы вы случайно перезаписали нуль-терминатор в конце строки (например, присвоив что-либо для `mystring[6]`), то от std::cout вы бы получили не только все символы строки, но и всё, что находится в соседних ячейках памяти до тех пор, пока не попался бы `0`!

Обратите внимание, это нормально, если длина массива больше строки, которую он хранит:

```
#include <iostream>

int main()
{
    char name[15] = "Max"; // используется только 4 символа (3 буквы + нуль-терминатор)
    std::cout << "My name is: " << name << '\n';

    return 0;
}
```

В этом случае строка `Max` будет выведена, а std::cout остановится на нуль-терминаторе. Остальные символы в массиве будут проигнорированы.

Строки C-style и std::cin

Есть много случаев, когда мы не знаем заранее, насколько длинной будет наша строка. Например, рассмотрим проблему написания программы, где мы просим пользователя ввести свое имя. Насколько длинным оно будет? Это неизвестно до тех пор, пока пользователь его не введет!

В таком случае мы можем объявить массив размером больше, чем нам нужно:

```
#include <iostream>

int main()
{
    char name[255]; // объявляем достаточно большой массив (для хранения 255 символов)
    std::cout << "Enter your name: ";
    std::cin >> name;
    std::cout << "You entered: " << name << '\n';

    return 0;
}
```

В программе, приведенной выше, мы объявили массив из 255 символов, предполагая, что пользователь не введет имя длиннее 255 символов. Хоть это и распространенная практика, но она не очень эффективна, так как пользователю ничего не мешает ввести имя, содержащее более 255 символов (случайно или намеренно).

Намного лучше сделать следующим образом:

```
#include <iostream>

int main()
{
```

```
5.      char name[255]; // объявляем достаточно большой массив (для хранения 255 символов)
6.      std::cout << "Enter your name: ";
7.      std::cin.getline(name, 255);
8.      std::cout << "You entered: " << name << '\n';
9.
10.     return 0;
11. }
```

Вызов `cin.getline()` будет принимать до 254 символов в массив `name` (оставляя место для нуль-терминатора!). Любые лишние символы будут проигнорированы. Таким образом, мы можем гарантировать, что массив не будет переполнен!

Управление строками C-style

Язык C++ предоставляет множество функций для управления строками C-style, которые подключаются с помощью заголовочного файла cstring. Вот **несколько самых полезных функций**:

Функция strcpy_s() позволяет копировать содержимое одной строки в другую. Чаще всего это используется для присваивания значений строке:

```
1. #include <iostream>
2. #include <cstring>
3.
4. int main()
5. {
6.      char text[] = "Print this!";
7.      char dest[50];
8.      strcpy_s(dest, text);
9.      std::cout << dest; // выводим "Print this!"
10.
11.     return 0;
12. }
```

Тем не менее, использование функции strcpy_s() может легко вызвать переполнение массива, если не быть осторожным! В следующей программе, длина массива `dest` меньше длины копируемой строки, поэтому в результате мы получим переполнение массива:

```
1. #include <iostream>
2. #include <cstring>
3.
4. int main()
5. {
6.      char text[] = "Print this!";
7.      char dest[5]; // обратите внимание, длина массива dest всего 5 символов!
8.      strcpy_s(dest, text); // переполнение!
9.      std::cout << dest;
10.
11.     return 0;
12. }
```

Еще одной полезной функцией управления строками является **функция strlen()**, которая возвращает длину строки C-style (без учета нуль-терминатора):

```
#include <iostream>
#include <cstring>

int main()
{
    char name[15] = "Max"; // используется только 4 символа (3 буквы + нуль-терминатор)
    std::cout << "My name is " << name << '\n';
    std::cout << name << " has " << strlen(name) << " letters.\n";
    std::cout << name << " has " << sizeof(name) << " characters in the array.\n";

    return 0;
}
```

Результат выполнения программы:

```
My name is Max
Max has 3 letters.
Max has 15 characters in the array.
```

Обратите внимание на разницу между функцией strlen() и оператором sizeof. strlen() выводит количество символов до нуль-терминатора, тогда как оператор sizeof возвращает размер целого массива, независимо от того, что в нем находится.

Вот еще **полезные функции для управления строками C-style**:

- **функция strcat()** - добавляет одну строку к другой (опасно);
- **функция strncat()** - добавляет одну строку к другой (с проверкой размера места назначения);
- **функция strcmp()** - сравнивает две строки (возвращает `0`, если они равны);
- **функция strncmp()** - сравнивает две строки до определенного количества символов (возвращает `0`, если до указанного символа не было различий).

Например:

```
#include <iostream>
#include <cstring>

int main()
{
    // Просим пользователя ввести строку
    char buffer[255];
    std::cout << "Enter a string: ";
    std::cin.getline(buffer, 255);

    int spacesFound = 0;
    // Перебираем каждый символ, который ввел пользователь
```

```
13.     for (int index = 0; index < strlen(buffer); ++index)
14.     {
15.         // Подсчитываем количество пробелов
16.         if (buffer[index] == ' ')
17.             spacesFound++;
18.     }
19.
20.     std::cout << "You typed " << spacesFound << " spaces!\n";
21.
22.     return 0;
23. }
```

Стоит ли использовать строки C-style?

Знать о строках C-style стоит, так как они используются не так уж и редко, но использовать их без веской на то причины не рекомендуется. Вместо строк C-style используйте std::string (подключая заголовочный файл string), так как он проще, безопаснее и гибче.

Правило: Используйте std::string вместо строк C-style.

Урок №83. Введение в класс std::string_view

На уроке о строках C-style мы говорили об опасностях, которые возникают при их использовании. Конечно, строки C-style работают быстро, но при этом их использование не является таким уж простым и безопасным в сравнении с std::string.

Правда, стоит отметить, что и у std::string имеются свои недостатки, особенно когда речь заходит об использовании константных строк.

Рассмотрим следующий пример:

```
#include <iostream>
#include <string>

int main()
{
  char text[]{ "hello" };
  std::string str{ text };
  std::string more{ str };

  std::cout << text << ' ' << str << ' ' << more << '\n';

  return 0;
}
```

Результат выполнения программы:

```
hello hello hello
```

Внутри функции main() выполняется копирование строки `hello` 3 раза, в результате чего мы имеем 4 копии исходной строки:

- первая копия — это непосредственно сам строковый литерал `hello`, который создается на этапе компиляции и хранится в бинарном виде;
- еще одна копия создается при инициализации массива типа char;
- далее идут объекты `str` и `more` класса std::string, каждый из которых, в свою очередь, создает еще по одной копии строки.

Из-за того, что класс std::string спроектирован так, чтобы его объекты могли быть изменяемыми, каждому объекту класса std::string приходится хранить свою собственную копию строки. Благодаря этому, исходная строка может быть изменена без влияния на другие объекты std::string.

Это также справедливо и для константных строк (`const std::string`), несмотря на то, что подобные объекты не могут быть изменены.

Введение в класс std::string_view

В качестве следующего примера возьмем окно в вашем доме и автомобиль, стоящий на улице неподалеку. Вы можете посмотреть через окно и увидеть машину, но при этом вы не можете дотронуться до машины или передвинуть её. Ваше окно лишь обеспечивает вид на автомобиль, который является отдельным независимым от вас объектом.

В стандарте C++17 вводится еще один способ использования строк — с помощью **класса std::string_view**, который находится в заголовочном файле string_view.

В отличие от объектов класса std::string, которые хранят свою собственную копию строки, класс std::string_view обеспечивает **представление** (англ. ***"view"***) для заданной строки, которая может быть определена где-нибудь в другом месте.

Попробуем переписать код предыдущего примера, заменив каждое вхождение `std::string` на `std::string_view`:

```
#include <iostream>
#include <string_view>

int main()
{
  std::string_view text{ "hello" }; // представление для строки "hello", которое хранится в бинарном виде
  std::string_view str{ text }; // представление этой же строки - "hello"
  std::string_view more{ str }; // представление этой же строки - "hello"

  std::cout << text << ' ' << str << ' ' << more << '\n';

  return 0;
}
```

В результате мы получим точно такой же вывод на экран, как и в предыдущем примере, но при этом у нас не будут созданы лишние копии строки `hello`. Когда мы копируем объект класса std::string_view, то новый объект std::string_view будет "смотреть" на ту же самую строку, на которую "смотрел" исходный объект. Ко всему прочему, класс std::string_view не только быстр, но и обладает многими функциями, которые мы изучили при работе с классом std::string:

```
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "Trains are fast!" };

  std::cout << str.length() << '\n'; // 16
  std::cout << str.substr(0, str.find(' ')) << '\n'; // Trains
  std::cout << (str == "Trains are fast!") << '\n'; // 1
```

```
11.
12.   // Начиная с C++20
13.   std::cout << str.starts_with("Boats") << '\n'; // 0
14.   std::cout << str.ends_with("fast!") << '\n'; // 1
15.
16.   std::cout << str << '\n'; // Trains are fast!
17.
18.   return 0;
19. }
```

Т.к. объект класса std::string_view не создает копии строки, то, изменив исходную строку, мы, тем самым, повлияем и на её представление в связанном с ней объектом std::string_view:

```
1.  #include <iostream>
2.  #include <string_view>
3.
4.  int main()
5.  {
6.    char arr[]{ "Gold" };
7.    std::string_view str{ arr };
8.
9.    std::cout << str << '\n'; // Gold
10.
11.   // Изменяем 'd' на 'f' в arr
12.   arr[3] = 'f';
13.
14.   std::cout << str << '\n'; // Golf
15.
16.   return 0;
17. }
```

Изменяя `arr`, можно видеть, как изменяется и `str`. Это происходит из-за того, что исходная строка является общей для этих переменных. Стоит отметить, что при использовании объектов класса std::string_view лучше избегать модифицирования исходной строки, пока существуют связанные с ней объекты класса std::string_view, так как в противном случае, это может привести к путанице и ошибкам.

> ***Совет***: Используйте std::string_view вместо строк C-style. Для строк, которые не планируете изменять в дальнейшем, предпочтительнее использовать класс std::string_view вместо std::string.

Функции, модифицирующие представление

Вернемся к нашей аналогии с окном, только теперь рассмотрим окно с занавесками. Мы можем закрыть часть окна левой или правой занавеской, тем самым уменьшив то, что можно увидеть сквозь окно. Заметьте, мы не изменяем объекты, находящиеся снаружи окна, изменяется лишь сектор наблюдения из окна.

Аналогично и с классом std::string_view: в нем содержатся функции, позволяющие нам управлять представлением строки. Благодаря этому мы можем изменять представление строки без изменения исходной строки.

Для этого используются следующие функции:

- `remove_prefix()` — удаляет символы из левой части представления;
- `remove_suffix()` — удаляет символы из правой части представления.

Например:

```
1. #include <iostream>
2. #include <string_view>
3.
4. int main()
5. {
6.   std::string_view str{ "Peach" };
7.
8.   std::cout << str << '\n';
9.
10.  // Игнорируем первый символ
11.  str.remove_prefix(1);
12.
13.  std::cout << str << '\n';
14.
15.  // Игнорируем последние 2 символа
16.  str.remove_suffix(2);
17.
18.  std::cout << str << '\n';
19.
20.  return 0;
21. }
```

Результат выполнения программы:

```
Peach
each
ea
```

В отличие от настоящих занавесок, с помощью которых мы закрыли часть окна, объекты класса std::string_view нельзя "открыть обратно". Изменив однажды область видимости, вы уже не сможете вернуться к первоначальным значениям (стоит отметить, что есть приемы, которые позволяют решить данную проблему, но вдаваться в них мы не будем).

std::string_view и обычные строки

В отличие от строк C-Style, объекты классов std::string и std::string_view не используют **нулевой символ** (**нуль-терминатор**) в качестве метки для обозначения

конца строки. Данные объекты знают, где заканчивается строка, т.к. отслеживают её длину:

```
#include <iostream>
#include <iterator> // для функции std::size()
#include <string_view>

int main()
{
  // Нет нуль-терминатора
  char vowels[]{ 'a', 'e', 'i', 'o', 'u' };

  // Массив vowels не является нуль-терминированным. Мы должны передавать длину вручную.
  // Поскольку vowels является массивом, то мы можем использовать функцию std::size(), чтобы получить его длину
  std::string_view str{ vowels, std::size(vowels) };

  std::cout << str << '\n'; // это безопасно, так как std::cout знает, как выводить std::string_view

  return 0;
}
```

Результат выполнения программы:

```
aeiou
```

Проблемы владения и доступа

Поскольку std::string_view является всего лишь представлением строки, его время жизни не зависит от времени жизни строки, которую он представляет. Если отображаемая строка выйдет за пределы области видимости, то std::string_view больше не сможет её отображать и при попытке доступа к ней мы получим неопределенные результаты:

```
#include <iostream>
#include <string>
#include <string_view>

std::string_view askForName()
{
  std::cout << "What's your name?\n";

  // Используем std::string, поскольку std::cin будет изменять строку
  std::string str{};
  std::cin >> str;

  // Мы переключаемся на std::string_view только в демонстрационных целях.
  // Если вы уже имеете std::string, то нет необходимости переключаться на std::string_view
  std::string_view view{ str };

  std::cout << "Hello " << view << '\n';

```

```
19.   return view;
20. } // str уничтожается и, таким образом, уничтожается и строка, созданная str
21.
22. int main()
23. {
24.   std::string_view view{ askForName() };
25.
26.   // view пытается обратиться к строке, которой уже не существует
27.   std::cout << "Your name is " << view << '\n'; // неопределенное поведение
28.
29.   return 0;
30. }
```

Результат выполнения программы:

```
What's your name?
nascardriver
Hello nascardriver
Your name is P@P@
```

Когда мы объявили переменную `str` и с помощью std::cin присвоили ей определенное значение, то данная переменная создала внутри себя строку, разместив её в динамической области памяти. После того, как переменная `str` вышла за пределы области видимости в конце функции askForName(), внутренняя строка вслед за этим прекратила свое существование. При этом объект класса std::string_view не знает, что строки больше не существует, и все также позволяет нам к ней обратиться. Попытка доступа к такой строке через её представление в функции main() приводит к неопределенному поведению, в результате чего мы получаем кракозябры.

Такая же ситуация может произойти и тогда, когда мы создаем объект std::string_view из объекта std::string, а затем модифицируем первоначальный объект std::string. Изменение объекта std::string может привести к созданию в другом месте новой внутренней строки и последующему уничтожению старой. При этом std::string_view продолжит "смотреть" в то место, где была старая строка, но её там уже не будет.

> ***Предупреждение*:** Следите за тем, чтобы исходная строка, на которую ссылается объект std::string_view, не выходила за пределы области видимости и не изменялась до тех пор, пока используется ссылающийся на нее объект std::string_view.

Конвертация std::string_view в std::string

Объекты класса std::string_view не конвертируются неявным образом в объекты класса std::string, но конвертируются при явном преобразовании:

```
#include <iostream>
#include <string>
#include <string_view>

void print(std::string s)
{
  std::cout << s << '\n';
}

int main()
{
  std::string_view sv{ "balloon" };

  sv.remove_suffix(3);

  // print(sv); // ошибка компиляции: неявная конвертация запрещена

  std::string str{ sv }; // явное преобразование

  print(str); // ок

  print(static_cast<std::string>(sv)); // ок

  return 0;
}
```

Результат выполнения программы:

```
ball
ball
```

Конвертация std::string_view в строку C-style

Некоторые старые функции (такие как `strlen()`) работают только со строками C-style. Для того чтобы преобразовать объект класса std::string_view в строку C-style, мы сначала должны конвертировать его в объект класса std::string:

```
#include <cstring>
#include <iostream>
#include <string>
#include <string_view>

int main()
{
  std::string_view sv{ "balloon" };

  sv.remove_suffix(3);

  // Создание объекта std::string из объекта std::string_view
  std::string str{ sv };
```

```
14.
15.   // Получаем строку C-style с нуль-терминатором
16.   auto szNullTerminated{ str.c_str() };
17.
18.   // Передаем строку с нуль-
    терминатором в функцию, которую мы хотим использовать
19.   std::cout << str << " has " << std::strlen(szNullTerminated) << " letter(s)\n
    ";
20.
21.   return 0;
22. }
```

Результат выполнения программы:

```
ball has 4 letter(s)
```

Однако стоит учитывать, что создание объекта класса std::string всякий раз, когда мы хотим преобразовать объект std::string_view в строку C-style, является дорогостоящей операцией, поэтому мы должны по возможности избегать подобных ситуаций.

Функция data()

Доступ к исходной строке объекта std::string_view можно получить при помощи **функции data()**, которая возвращает строку C-style. При этом обеспечивается быстрый доступ к представляемой строке (как к строке C-style). Но это следует использовать только тогда, когда объект std::string_view не был изменен (например, при помощи функций `remove_prefix()` или `remove_suffix()`) и связанная с ним строка имеет нуль-терминатор (так как это строка C-style).

В следующем примере функция std::strlen() ничего не знает о std::string_view, поэтому мы передаем ей функцию `str.data()`:

```
1.  #include <cstring> // для функции std::strlen()
2.  #include <iostream>
3.  #include <string_view>
4.
5.  int main()
6.  {
7.    std::string_view str{ "balloon" };
8.
9.    std::cout << str << '\n';
10.
11.   // Для простоты мы воспользуемся функцией std::strlen(). Вместо нее можно
    было бы использовать любую другую функцию, которая работает со строкой с нуль-
    терминатором в конце.
12.   // Здесь мы можем использовать функцию data(), так как мы не изменяли
    представление и строка имеет нуль-терминатор
13.   std::cout << std::strlen(str.data()) << '\n';
14.
15.   return 0;
16. }
```

Результат выполнения программы:

```
balloon
7
```

Когда мы пытаемся обратиться к объекту класса std::string_view, который был изменен, функция data() может вернуть совсем не тот результат, который мы ожидали от нее получить. В следующем примере показано, что происходит, когда мы обращаемся к функции data() после изменения представления строки:

```
#include <cstring>
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "balloon" };

  // Удаляем символ "b"
  str.remove_prefix(1);

  // Удаляем часть "oon"
  str.remove_suffix(3);

  // Помните, что предыдущие 2 команды не изменяют исходную строку, они работают лишь с её представлением
  std::cout << str << " has " << std::strlen(str.data()) << " letter(s)\n";
  std::cout << "str.data() is " << str.data() << '\n';
  std::cout << "str is " << str << '\n';

  return 0;
}
```

Результат выполнения программы:

```
all has 6 letter(s)
str.data() is alloon
str is all
```

Очевидно, что данный результат — это не то, что мы планировали увидеть, и он является следствием попытки функции data() получить доступ к данным представления std::string_view, которое было изменено. Информация о длине строки теряется при обращении к ней через функцию data(). std::strlen и std::cout продолжают считывать символы из исходной строки до тех пор, пока не встретят нуль-терминатор, который находится в конце строки `baloon`.

> ***Предупреждение***: Используйте std::string_view::data() только в том случае, если представление std::string_view не было изменено и отображаемая строка содержит завершающий нулевой символ (нуль-терминатор). Использование

> функции std::string_view::data() со строкой без нуль-терминатора чревато возникновением ошибок.

Нюансы std::string_view

Будучи относительно недавним нововведением, класс std::string_view реализован не так уж и идеально, как хотелось бы:

```
1. std::string s{ "hello" };
2. std::string_view v{ "world" };
3.
4. // Не работает
5. std::cout << (s + v) << '\n';
6. std::cout << (v + s) << '\n';
7.
8. // Потенциально небезопасно или не то, что мы хотим получить,
9. // поскольку мы пытаемся использовать объект std::string_view в качестве
   строки C-style
10. std::cout << (s + v.data()) << '\n';
11. std::cout << (v.data() + s) << '\n';
12.
13. // Приемлемо, т.к. нам нужно создать новый объект std::string, но некрасиво и
   нерационально
14. std::cout << (s + std::string{ v }) << '\n';
15. std::cout << (std::string{ v } + s) << '\n';
16. std::cout << (s + static_cast<std::string>(v)) << '\n';
17. std::cout << (static_cast<std::string>(v) + s) << '\n';
```

Нет никаких причин для неработоспособности строк №5-6, но тем не менее они не работают. Вероятно, полная поддержка данного функционала будет реализована в следующих версиях стандарта C++.

Урок №84. Указатели

При выполнении инициализации переменной, ей автоматически присваивается свободный адрес памяти, и, любое значение, которое мы присваиваем переменной, сохраняется по этому адресу в памяти. Например:

```
1. int b;
```

При выполнении этого стейтмента процессором, выделяется часть оперативной памяти. В качестве примера предположим, что переменной `b` присваивается ячейка памяти под номером 150. Всякий раз, когда программа встречает переменную `b` в выражении или в стейтменте, она понимает, что для того, чтобы получить значение — ей нужно заглянуть в ячейку памяти под номером 150.

Хорошая новость: нам не нужно беспокоиться о том, какие конкретно адреса памяти выделены для определенных переменных. Мы просто ссылаемся на переменную через присвоенный ей идентификатор, а компилятор конвертирует это имя в соответствующий адрес памяти. Однако этот подход имеет некоторые ограничения, которые мы обсудим на этом и следующих уроках.

Оператор адреса `&` позволяет узнать, какой адрес памяти присвоен определенной переменной. Всё довольно просто:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int a = 7;
6.     std::cout << a << '\n'; // выводим значение переменной a
7.     std::cout << &a << '\n'; // выводим адрес памяти переменной a
8.
9.     return 0;
10. }
```

Результат на моем компьютере:

```
7
0046FCF0
```

> ***Примечание:*** Хотя оператор адреса выглядит так же, как оператор побитового И, отличить их можно по тому, что оператор адреса является унарным оператором, а оператор побитового И — бинарным оператором.

Оператор разыменования *

Оператор разыменования `*` позволяет получить значение по указанному адресу:

```
#include <iostream>

int main()
{
    int a = 7;
    std::cout << a << '\n'; // выводим значение переменной a
    std::cout << &a << '\n'; // выводим адрес переменной a
    std::cout << *&a << '\n'; /// выводим значение ячейки памяти переменной a

    return 0;
}
```

Результат на моем компьютере:

```
7
0046FCF0
7
```

> ***Примечание:*** Хотя оператор разыменования выглядит так же, как и оператор умножения, отличить их можно по тому, что оператор разыменования - унарный, а оператор умножения - бинарный.

Указатели

Теперь, когда мы уже знаем об операторах адреса и разыменования, мы можем поговорить об указателях.

Указатель - это переменная, значением которой является адрес ячейки памяти. Указатели объявляются точно так же, как и обычные переменные, только со звёздочкой между типом данных и идентификатором:

```
int *iPtr; // указатель на значение типа int
double *dPtr; // указатель на значение типа double

int* iPtr3; // корректный синтаксис (допустимый, но не желательный)
int * iPtr4; // корректный синтаксис (не делайте так)

int *iPtr5, *iPtr6; // объявляем два указателя для переменных типа int
```

Синтаксически язык C++ принимает объявление указателя, когда звёздочка находится рядом с типом данных, с идентификатором или даже посередине. Обратите внимание, эта звёздочка не является оператором разыменования. Это всего лишь часть синтаксиса объявления указателя.

Однако, при объявлении нескольких указателей, звёздочка должна находиться возле каждого идентификатора. Это легко забыть, если вы привыкли указывать звёздочку возле типа данных, а не возле имени переменной. Например:

```
1. int* iPtr3, iPtr4; // iPtr3 - это указатель на значение типа int, а iPtr4 - это
    обычная переменная типа int!
```

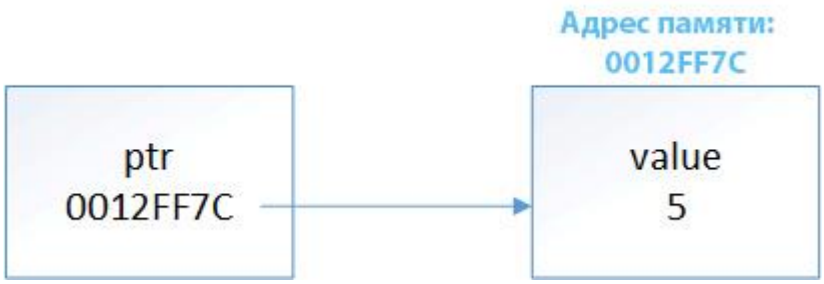
По этой причине, при объявлении указателя, рекомендуется указывать звёздочку возле имени переменной. Как и обычные переменные, указатели не инициализируются при объявлении. Содержимым неинициализированного указателя является обычный мусор.

Присваивание значений указателю

Поскольку указатели содержат только адреса, то при присваивании указателю значения - это значение должно быть адресом. Для получения адреса переменной используется оператор адреса:

```
1. int value = 5;
2. int *ptr = &value; // инициализируем ptr адресом значения переменной
```

Приведенное выше можно проиллюстрировать следующим образом:



Вот почему указатели имеют такое имя: `ptr` содержит адрес значения переменной `value`, и, можно сказать, `ptr` *указывает* на это значение.

Еще очень часто можно увидеть следующее:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int value = 5;
6.     int *ptr = &value; // инициализируем ptr адресом значения переменной
7.
8.     std::cout << &value << '\n'; // выводим адрес значения переменной value
9.     std::cout << ptr << '\n'; // выводим адрес, который хранит ptr
10.
11.    return 0;
12. }
```

Результат на моем компьютере:

```
003AFCD4
003AFCD4
```

Тип указателя должен соответствовать типу переменной, на которую он указывает:

```
1. int iValue = 7;
2. double dValue = 9.0;
3.
4. int *iPtr = &iValue; // ок
5. double *dPtr = &dValue; // ок
6. iPtr = &dValue; // неправильно: указатель типа int не может указывать на
   адрес переменной типа double
7. dPtr = &iValue; // неправильно: указатель типа double не может указывать на
   адрес переменной типа int
```

Следующее не является допустимым:

```
1. int *ptr = 7;
```

Это связано с тем, что указатели могут содержать только адреса, а целочисленный литерал `7` не имеет адреса памяти. Если вы все же сделаете это, то компилятор сообщит вам, что он не может преобразовать целочисленное значение в целочисленный указатель.

Язык C++ также не позволит вам напрямую присваивать адреса памяти указателю:

```
1. double *dPtr = 0x0012FF7C; // не ок: рассматривается как присваивание
   целочисленного литерала
```

Оператор адреса возвращает указатель

Стоит отметить, что оператор адреса `&` не возвращает адрес своего операнда в качестве литерала. Вместо этого он возвращает указатель, содержащий адрес операнда, тип которого получен из аргумента (например, адрес переменной типа int передается как адрес указателя на значение типа int):

```
1. #include <iostream>
2. #include <typeinfo>
3.
4. int main()
5. {
6.     int x(4);
7.     std::cout << typeid(&x).name();
8.
9.     return 0;
10. }
```

Результат выполнения программы:

```
int *
```

Разыменование указателей

Как только у нас есть указатель, указывающий на что-либо, мы можем его разыменовать, чтобы получить значение, на которое он указывает. Разыменованный указатель - это содержимое ячейки памяти, на которую он указывает:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int value = 5;
6.     std::cout << &value << std::endl; // выводим адрес value
7.     std::cout << value << std::endl; // выводим содержимое value
8.
9.     int *ptr = &value; // ptr указывает на value
10.    std::cout << ptr << std::endl; // выводим адрес, который хранится в ptr, т.е. &value
11.    std::cout << *ptr << std::endl; // разыменовываем ptr (получаем значение на которое указывает ptr)
12.
13.    return 0;
14. }
```

Результат:

```
0034FD90
5
0034FD90
5
```

Вот почему указатели должны иметь тип данных. Без типа указатель не знал бы, как интерпретировать содержимое, на которое он указывает (при разыменовании). Также, поэтому и должны совпадать тип указателя с типом переменной. Если они не совпадают, то указатель при разыменовании может неправильно интерпретировать биты (например, вместо типа double использовать тип int).

Одному указателю можно присваивать разные значения:

```
1. int value1 = 5;
2. int value2 = 7;
3.
4. int *ptr;
5.
6. ptr = &value1; // ptr указывает на value1
7. std::cout << *ptr; // выведется 5
8.
```

```
ptr = &value2; // ptr теперь указывает на value2
std::cout << *ptr; // выведется 7
```

Когда адрес значения переменной присвоен указателю, то выполняется следующее:

- `ptr` - это то же самое, что и `&value`;
- `*ptr` обрабатывается так же, как и `value`.

Поскольку `*ptr` обрабатывается так же, как и `value`, то мы можем присваивать ему значения так, как если бы это была обычная переменная. Например:

```
int value = 5;
int *ptr = &value; // ptr указывает на value

*ptr = 7; // *ptr - это то же самое, что и value, которому мы присвоили значение 7
std::cout << value; // выведется 7
```

Разыменование некорректных указателей

Указатели в языке C++ по своей природе являются небезопасными, а их неправильное использование - один из лучших способов получить сбой программы.

При разыменовании указателя, программа пытается перейти в ячейку памяти, которая хранится в указателе и извлечь содержимое этой ячейки. По соображениям безопасности современные операционные системы (ОС) запускают программы в песочнице для предотвращения их неправильного взаимодействия с другими программами и для защиты стабильности самой операционной системы. Если программа попытается получить доступ к ячейке памяти, не выделенной для нее операционной системой, то ОС сразу завершит выполнение этой программы.

Следующая программа хорошо иллюстрирует вышесказанное. При запуске вы получите сбой (попробуйте, ничего страшного с вашим компьютером не произойдет):

```
#include <iostream>

void foo(int *&p)
{
}

int main()
{
    int *p; // создаем неинициализированный указатель (содержимым которого является мусор)
    foo(p); // вводим компилятор в заблуждение, будто бы собираемся присвоить указателю корректное значение

    std::cout << *p; // разыменовываем указатель с мусором

```

```
14.     return 0;
15. }
```

Размер указателей

Размер указателя зависит от архитектуры, на которой скомпилирован исполняемый файл: 32-битный исполняемый файл использует 32-битные адреса памяти. Следовательно, указатель на 32-битном устройстве занимает 32 бита (4 байта). С 64-битным исполняемым файлом указатель будет занимать 64 бита (8 байт). И это вне зависимости от того, на что указывает указатель:

```
1. char *chPtr; // тип char занимает 1 байт
2. int *iPtr; // тип int занимает 4 байта
3.
4. struct Something
5. {
6.     int nX, nY, nZ;
7. };
8.
9. Something *somethingPtr;
10.
11. std::cout << sizeof(chPtr) << '\n'; // выведется 4
12. std::cout << sizeof(iPtr) << '\n'; // выведется 4
13. std::cout << sizeof(somethingPtr) << '\n'; // выведется 4
```

Как вы можете видеть, размер указателя всегда один и тот же. Это связано с тем, что указатель - это всего лишь адрес памяти, а количество бит, необходимое для доступа к адресу памяти на определенном устройстве, — всегда постоянное.

В чём польза указателей?

Сейчас вы можете подумать, что указатели являются непрактичными и вообще ненужными. Зачем использовать указатель, если мы можем использовать исходную переменную?

Однако, оказывается, **указатели полезны в следующих случаях**:

- ***Случай №1:*** **Массивы реализованы с помощью указателей**. Указатели могут использоваться для итерации по массиву.
- ***Случай №2:*** **Они являются единственным способом динамического выделения памяти в C++**. Это, безусловно, самый распространенный вариант использования указателей.
- ***Случай №3:*** **Они могут использоваться для передачи большого количества данных в функцию без копирования этих данных**.
- ***Случай №4:*** **Они могут использоваться для передачи одной функции в качестве параметра другой функции**.

- ***Случай №5:*** **Они используются для достижения полиморфизма при работе с наследованием**.
- ***Случай №6:*** **Они могут использоваться для представления одной структуры/класса в другой структуре/классе, формируя, таким образом, целые цепочки**.

Указатели применяются во многих случаях. Не волнуйтесь, если вы многого не понимаете из вышесказанного. Теперь, когда мы разобрались с указателями на базовом уровне, мы можем начать углубляться в отдельные случаи, в которых они полезны, что мы и сделаем на последующих уроках.

Заключение

Указатели - это переменные, которые содержат адреса памяти. Их можно разыменовать с помощью оператора разыменования `*` для извлечения значений, хранимых по адресу памяти. Разыменование указателя, значением которого является мусор, приведет к сбою в вашей программе.

> ***Совет*:** При объявлении указателя указывайте звёздочку возле имени переменной.

Тест

Задание №1

Какие значения мы получим в результате выполнения следующей программы (предположим, что это 32-битное устройство, и тип short занимает 2 байта):

```
1. short value = 7; // &value = 0012FF60
2. short otherValue = 3; // &otherValue = 0012FF54
3.
4. short *ptr = &value;
5.
6. std::cout << &value << '\n';
7. std::cout << value << '\n';
8. std::cout << ptr << '\n';
9. std::cout << *ptr << '\n';
10.std::cout << '\n';
11.
12.*ptr = 9;
13.
14.std::cout << &value << '\n';
15.std::cout << value << '\n';
16.std::cout << ptr << '\n';
17.std::cout << *ptr  << '\n';
18.std::cout << '\n';
19.
20.ptr = &otherValue;
21.
```

```
22. std::cout << &otherValue << '\n';
23. std::cout << otherValue << '\n';
24. std::cout << ptr << '\n';
25. std::cout << *ptr << '\n';
26. std::cout << '\n';
27.
28. std::cout << sizeof(ptr) << '\n';
29. std::cout << sizeof(*ptr) << '\n';
```

Задание №2

Что не так со следующим фрагментом кода:

```
1. int value = 45;
2. int *ptr = &value; // объявляем указатель и инициализируем его адресом переменн
   ой value
3. *ptr = &value; // присваиваем адрес value для ptr
```

Урок №85. Нулевые указатели

Как и в случае с обычными переменными, указатели не инициализируются при создании. Если значение не было присвоено, то указатель по умолчанию будет указывать на любой адрес, содержимым которого является мусор.

Нулевое значение и нулевые указатели

Помимо адресов памяти, есть еще одно значение, которое указатель может хранить: значение null. **Нулевое значение** (или ***"значение null"***) - это специальное значение, которое означает, что указатель ни на что не указывает. Указатель, содержащий значение null, называется **нулевым указателем**.

В языке C++ мы можем присвоить указателю нулевое значение, инициализируя его/присваивая ему литерал `0`:

```
int *ptr(0);  // ptr теперь нулевой указатель

int *ptr1; // ptr1 не инициализирован
ptr1 = 0; // ptr1 теперь нулевой указатель
```

Поскольку значением нулевого указателя является нуль, то это можно использовать внутри условного ветвления для проверки того, является ли указатель нулевым или нет:

```
#include <iostream>

int main()
{
    double *ptr(0);

    if (ptr)
        std::cout << "ptr is pointing to a double value.";
    else
        std::cout << "ptr is a null pointer.";

    return 0;
}
```

> ***Совет:*** Инициализируйте указатели нулевым значением, если не собираетесь присваивать им другие значения.

Разыменование нулевых указателей

Как мы уже знаем из предыдущего урока, разыменование указателей с мусором приведет к неожиданным результатам. С разыменованием нулевого указателя дела обстоят так же. В большинстве случаев вы получите сбой в программе.

В этом есть смысл, ведь разыменование указателя означает, что нужно «перейти к адресу, на который указывает указатель, и достать из этого адреса значение». Нулевой указатель не имеет адреса, поэтому и такой результат.

Макрос NULL

В языке Си (но не в C++) есть специальный макрос препроцессора с именем NULL, который определен как значение `0`. Хоть он и не является частью языка C++, его использование достаточно распространено, и должно работать в каждом компиляторе C++:

```
1. int *ptr(NULL); // присваиваем адрес 0 указателю ptr
```

Однако, поскольку NULL является макросом препроцессора и, технически, не является частью C++, то его не рекомендуется использовать.

Ключевое слово nullptr в C++11

Обратите внимание, значение `0` не является типом указателя, и присваивание указателю значения `0` для обозначения того, что он является нулевым - немного противоречиво, вам не кажется? В редких случаях, использование `0` в качестве аргумента-литерала может привести к проблемам, так как компилятор не сможет определить, используется ли нулевой указатель или целое число `0`:

```
1. doAnything(0); // является ли 0 аргументом-значением или аргументом-
   указателем? (компилятор определит его как целочисленное значение)
```

Для решения этой проблемы в C++11 ввели новое **ключевое слово nullptr**, которое также является константой r-value.

Начиная с C++11, при работе с нулевыми указателями, использование nullptr является более предпочтительным вариантом, нежели использование `0`:

```
1. int *ptr = nullptr; // примечание: ptr по-
   прежнему остается указателем типа int, просто со значением null (0)
```

Язык C++ неявно преобразует nullptr в соответствующий тип указателя. Таким образом, в вышеприведенном примере, nullptr неявно преобразуется в указатель типа int, а затем значение nullptr присваивается `ptr`.

nullptr также может использоваться для вызова функции (в качестве аргумента-литерала):

```
1. #include <iostream>
2.
```

```
3.  void doAnything(int *ptr)
4.  {
5.      if (ptr)
6.          std::cout << "You passed in " << *ptr << '\n';
7.      else
8.          std::cout << "You passed in a null pointer\n";
9.  }
10.
11. int main()
12. {
13.     doAnything(nullptr); // теперь аргумент является точно нулевым указателем,
    а не целочисленным значением
14.
15.     return 0;
16. }
```

> ***Совет:*** В C++11 используйте nullptr для инициализации нулевых указателей.

Тип данных std::nullptr_t в C++11

В C++11 добавили новый **тип данных std::nullptr_t**, который находится в заголовочном файле cstddef. std::nullptr_t может иметь только одно значение — `nullptr`! Хотя это может показаться немного глупым, но это полезно в одном случае. Если вам нужно написать функцию, которая принимает аргумент nullptr, то какой тип параметра нужно использовать? Правильно! std::nullptr_t. Например:

```
1.  #include <iostream>
2.  #include <cstddef> // для std::nullptr_t
3.
4.  void doAnything(std::nullptr_t ptr)
5.  {
6.      std::cout << "in doAnything()\n";
7.  }
8.
9.  int main()
10. {
11.     doAnything(nullptr); // вызов функции doAnything() с аргументом типа
    std::nullptr_t
12.
13.     return 0;
14. }
```

Вам, вероятно, никогда это не придется использовать, но знать об этом стоит (на всякий пожарный).

Урок №86. Указатели и массивы

В языке C++ указатели и массивы тесно связаны между собой.

Сходства между указателями и массивами

Фиксированный массив определяется следующим образом:

```
1. int array[4] = { 5, 8, 6, 4 }; // определяем фиксированный массив из 4-
   х целых чисел
```

Для нас это массив из 4-х целых чисел, но для компилятора `array` является переменной типа `int[4]`. Мы знаем что `array[0] = 5`, `array[1] = 8`, `array[2] = 6` и `array[3] = 4`. Но какое значение имеет сам `array`?

Переменная `array` содержит адрес первого элемента массива, как если бы это был указатель! Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int array[4] = { 5, 8, 6, 4 };
6.
7.     // Выводим значение массива (переменной array)
8.     std::cout << "The array has address: " << array << '\n';
9.
10.    // Выводим адрес элемента массива
11.    std::cout << "Element 0 has address: " << &array[0] << '\n';
12.
13.    return 0;
14. }
```

Результат на моем компьютере:

```
The array has address: 004BF968
Element 0 has address: 004BF968
```

Обратите внимание, адрес, хранящийся в переменной `array`, является адресом первого элемента массива.

Распространенная ошибка думать, что переменная `array` и указатель на `array` являются одним и тем же объектом. Это не так. Хотя оба указывают на первый элемент массива, информация о типе данных у них разная. В вышеприведенном примере типом переменной `array` является `int[4]`, тогда как типом указателя на массив является `int *`.

Путаница вызвана тем, что во многих случаях, при вычислении, фиксированный массив **распадается** (неявно преобразовывается) в указатель на первый элемент массива. Доступ к элементам по-прежнему осуществляется через указатель, но информация, полученная из типа массива (например, его размер), не может быть доступна из типа указателя.

Однако и это не является столь весомым аргументом, чтобы рассматривать фиксированные массивы и указатели как разные значения. Например, мы можем разыменовать массив, чтобы получить значение первого элемента:

```
1. int array[4] = { 5, 8, 6, 4 };
2.
3. // Разыменование массива (переменной array) приведет к возврату первого
   элемента массива (элемента под номером 0)
4. std::cout << *array; // выведется 5!
5.
6. char name[] = "John"; // строка C-style (также массив)
7. std::cout << *name; // выведется 'J'
```

Обратите внимание, мы не разыменовываем фактический массив. Массив (типа `int[4]`) неявно конвертируется в указатель (типа `int *`), и мы разыменовываем указатель, который указывает на значение первого элемента массива.

Также мы можем создать указатель и присвоить ему `array`:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int array[4] = { 5, 8, 6, 4 };
6.     std::cout << *array; // выведется 5
7.
8.     int *ptr = array;
9.     std::cout << *ptr; // выведется  5
10.
11.    return 0;
12.}
```

Это работает из-за того, что переменная `array` распадается в указатель типа `int *`, а тип нашего указателя такой же (т.е. `int *`).

Различия между указателями и массивами

Однако есть случаи, когда разница между фиксированными массивами и указателями имеет значение. Основное различие возникает при использовании оператора sizeof. При использовании в фиксированном массиве, оператор sizeof возвращает размер всего массива (`длина массива * размер элемента`). При

использовании с указателем, оператор sizeof возвращает размер адреса памяти (в байтах). Например:

```
#include <iostream>

int main()
{
    int array[4] = { 5, 8, 6, 4 };

    std::cout << sizeof(array) << '\n'; // выведется sizeof(int) * длина array

        int *ptr = array;
        std::cout << sizeof(ptr) << '\n'; // выведется размер указателя

    return 0;
}
```

Результат выполнения программы:

```
16
4
```

Фиксированный массив знает свою длину, а указатель на массив - нет.

Второе различие возникает при использовании оператора адреса `&`. Используя адрес указателя, мы получаем адрес памяти переменной указателя. Используя адрес массива, возвращается указатель на целый массив. Этот указатель также указывает на первый элемент массива, но информация о типе отличается. Вряд ли вам когда-нибудь понадобится это использовать.

Передача массивов в функции

На предыдущих уроках мы говорили, что, из-за того, что копирование больших массивов при передаче в функцию является очень затратной операцией, C++ не копирует массив. При передаче массива в качестве аргумента в функцию, массив распадается в указатель на массив и этот указатель передается в функцию:

```
#include <iostream>

void printSize(int *array)
{
    // Здесь массив рассматривается как указатель
    std::cout << sizeof(array) << '\n'; // выведется размер указателя, а не длина массива!
}

int main()
{
    int array[] = { 1, 2, 3, 4, 4, 9, 15, 25 };
    std::cout << sizeof(array) << '\n'; // выведется sizeof(int) * длина массива
```

```
13.
14.     printSize(array); // здесь аргумент array распадается на указатель
15.
16.     return 0;
17. }
```

Результат выполнения программы:

```
32
4
```

Обратите внимание, результат будет таким же, даже если параметром будет фиксированный массив:

```
1.  #include <iostream>
2.
3.  // C++ неявно конвертирует параметр array[] в *array
4.  void printSize(int array[])
5.  {
6.      // Здесь массив рассматривается как указатель, а не как фиксированный
    массив
7.      std::cout << sizeof(array) << '\n'; // выведется размер указателя, а не
    размер массива!
8.  }
9.
10. int main()
11. {
12.     int array[] = { 1, 2, 3, 4, 4, 9, 15, 25 };
13.     std::cout << sizeof(array) << '\n'; // выведется sizeof(int) * длина
    массива array
14.
15.     printSize(array); // здесь аргумент array распадается на указатель
16.
17.     return 0;
18. }
```

Результат выполнения программы:

```
32
4
```

В примере, приведенном выше, C++ неявно конвертирует параметр из синтаксиса массива (`[]`) в синтаксис указателя (`*`). Это означает, что следующие два объявления функции идентичны:

```
1.  void printSize(int array[]);
2.  void printSize(int *array);
```

Некоторые программисты предпочитают использовать синтаксис `[]`, так как он позволяет понять, что функция ожидает массив, а не указатель на значение. Однако, в большинстве случаев, поскольку указатель не знает, насколько велик массив, вам

придется передавать размер массива в качестве отдельного параметра (строки являются исключением, так как они нуль-терминированные).

Рекомендуется использовать синтаксис указателя, поскольку он позволяет понять, что параметр будет обработан как указатель, а не как фиксированный массив, и определенные операции, такие как в случае с оператором sizeof, будут выполняться с параметром-указателем (а не с параметром-массивом).

> ***Совет:*** Используйте синтаксис указателя (`*`) вместо синтаксиса массива (`[]`) при передаче массивов в качестве параметров в функции.

Передача по адресу

Тот факт, что массивы распадаются на указатели при передаче в функции, объясняет основную причину, по которой изменение массива в функции приведет к изменению фактического массива. Рассмотрим следующий пример:

```
1. #include <iostream>
2.
3. // Параметр ptr содержит копию адреса массива
4. void changeArray(int *ptr)
5. {
6.     *ptr = 5; // поэтому изменение элемента массива приведет к изменению
   фактического массива
7. }
8.
9. int main()
10.{
11.    int array[] = { 1, 2, 3, 4, 4, 9, 15, 25 };
12.    std::cout << "Element 0 has value: " << array[0] << '\n';
13.
14.    changeArray(array);
15.
16.    std::cout << "Element 0 has value: " << array[0] << '\n';
17.
18.     return 0;
19.}
```

Результат выполнения программы:

```
Element 0 has value: 1
Element 0 has value: 5
```

При вызове функции changeArray(), массив распадается на указатель, а значение этого указателя (адрес памяти первого элемента массива) копируется в параметр `ptr` функции changeArray(). Хотя значение `ptr` в функции является копией адреса массива, `ptr` все равно указывает на фактический массив (а не на копию!). Следовательно, при разыменовании `ptr`, разыменовывается и фактический массив!

Этот феномен работает так же и с указателями на значения не из массива. Более детально эту тему мы рассмотрим на соответствующем уроке.

Массивы в структурах и классах

Стоит упомянуть, что массивы, которые являются частью структур или классов, не распадаются, когда вся структура или класс передается в функцию.

На следующем уроке мы рассмотрим адресную арифметику и поговорим о том, как на самом деле работает индексация массива.

Урок №87. Адресная арифметика и индексация массивов

Язык C++ позволяет выполнять целочисленные операции сложения/вычитания с указателями. Если `ptr` указывает на целое число, то `ptr + 1` является адресом следующего целочисленного значения в памяти после `ptr`. `ptr - 1` — это адрес предыдущего целочисленного значения (перед `ptr`).

Адресная арифметика

Обратите внимание, `ptr + 1` не возвращает следующий *любой адрес памяти*, который находится сразу после `ptr`, но он возвращает *адрес памяти следующего объекта, тип которого совпадает* с типом значения, на которое указывает `ptr`. Если `ptr` указывает на адрес памяти целочисленного значения (размер которого 4 байта), то `ptr + 3` будет возвращать адрес памяти третьего целочисленного значения после `ptr`. Если `ptr` указывает на адрес памяти значения **типа char**, то `ptr + 3` будет возвращать адрес памяти третьего значения типа char после `ptr`.

При вычислении результата выражения **адресной арифметики** (или ***"арифметики с указателями"***) компилятор всегда умножает целочисленный операнд на размер объекта, на который указывает указатель. Например:

```
#include <iostream>

int main()
{
    int value = 8;
    int *ptr = &value;

    std::cout << ptr << '\n';
    std::cout << ptr+1 << '\n';
    std::cout << ptr+2 << '\n';
    std::cout << ptr+3 << '\n';

    return 0;
}
```

Результат на моем компьютере:

```
002CF9A4
002CF9A8
002CF9AC
002CF9B0
```

Как вы можете видеть, каждый последующий адрес увеличивается на 4. Это связано с тем, что размер типа int на моем компьютере составляет 4 байта.

Та же программа, но с использованием типа short вместо типа int:

```
#include <iostream>

int main()
{
    short value = 8;
    short *ptr = &value;

    std::cout << ptr << '\n';
    std::cout << ptr+1 << '\n';
    std::cout << ptr+2 << '\n';
    std::cout << ptr+3 << '\n';

    return 0;
}
```

Результат:

```
002BFA20
002BFA22
002BFA24
002BFA26
```

Поскольку тип short занимает 2 байта, то каждый следующий адрес больше предыдущего на 2.

Расположение элементов массива в памяти

Используя оператор адреса `&`, мы можем легко определить, что элементы массива расположены в памяти последовательно. То есть, элементы 0, 1, 2 и т.д. размещены рядом (друг за другом):

```
#include <iostream>

int main()
{
    int array[] = { 7, 8, 2, 4, 5 };

    std::cout << "Element 0 is at address: " << &array[0] << '\n';
    std::cout << "Element 1 is at address: " << &array[1] << '\n';
    std::cout << "Element 2 is at address: " << &array[2] << '\n';
    std::cout << "Element 3 is at address: " << &array[3] << '\n';

    return 0;
}
```

Результат на моем компьютере:

```
Element 0 is at address: 002CF6F4
Element 1 is at address: 002CF6F8
```

```
Element 2 is at address: 002CF6FC
Element 3 is at address: 002CF700
```

Обратите внимание, каждый из этих адресов по отдельности занимает 4 байта, как и размер типа int на моем компьютере.

Индексация массивов

Мы уже знаем, что элементы массива расположены в памяти последовательно и то, что **фиксированный массив** может распадаться на указатель, который указывает на первый элемент (элемент под индексом 0) массива.

Также мы уже знаем, что добавление единицы к указателю возвращает адрес памяти следующего объекта этого же типа данных.

Следовательно, можно предположить, что добавление единицы к идентификатору массива приведет к возврату адреса памяти второго элемента (элемента под индексом 1) массива. Проверим на практике:

```
1. #include <iostream>
2.
3. int main()
4. {
5.      int array [5] = { 7, 8, 2, 4, 5 };
6.
7.      std::cout << &array[1] << '\n'; // выведется адрес памяти элемента под индексом 1
8.      std::cout << array+1 << '\n'; // выведется адрес памяти указателя на массив + 1
9.
10.     std::cout << array[1] << '\n'; // выведется 8
11.     std::cout << *(array+1) << '\n'; // выведется 8 (обратите внимание на скобки, они здесь обязательны)
12.
13.    return 0;
14. }
```

При разыменовании результата выражения адресной арифметики скобки необходимы для соблюдения приоритета операций, поскольку оператор `*` имеет более высокий приоритет, чем оператор `+`.

Результат выполнения программы на моем компьютере:

```
001AFE74
001AFE74
8
8
```

Оказывается, когда компилятор видит оператор индекса `[]`, он, на самом деле, конвертирует его в указатель с операцией сложения и разыменования! То есть, `array[n]` — это то же самое, что и `*(array + n)`, где `n` является целочисленным значением. Оператор индекса `[]` используется в целях удобства, чтобы не нужно было всегда помнить о скобках.

Использование указателя для итерации по массиву

Мы можем использовать указатели и адресную арифметику для выполнения итераций по массиву. Хотя обычно это не делается (использование оператора индекса, как правило, читабельнее и менее подвержено ошибкам), следующий пример показывает, что это возможно:

```
#include <iostream>

int main()
{
    const int arrayLength = 9;
    char name[arrayLength] = "Jonathan";
    int numVowels(0);
    for (char *ptr = name; ptr < name + arrayLength; ++ptr)
    {
        switch (*ptr)
        {
        case 'A':
        case 'a':
        case 'E':
        case 'e':
        case 'I':
        case 'i':
        case 'O':
        case 'o':
        case 'U':
        case 'u':
            ++numVowels;
        }
    }

    std::cout << name << " has " << numVowels << " vowels.\n";

    return 0;
}
```

Как это работает? Программа использует указатель для *прогона* каждого элемента массива поочередно. Помните, что массив распадается в указатель на первый элемент массива? Поэтому, присвоив `name` для `ptr`, сам `ptr` стал указывать на первый элемент массива. Каждый элемент разыменовывается с помощью выражения switch, и, если текущий элемент массива является гласной буквой, то `numVowels` увеличивается. Для перемещения указателя на следующий символ (элемент) массива в цикле for используется оператор `++`. Работа цикла завершится, когда все символы будут проверены.

Результат выполнения программы:

```
Jonathan has 3 vowels.
```

Урок №88. Символьные константы строк C-style

Мы уже знаем, как создать и инициализировать строку C-style:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     char myName[] = "John";
6.     std::cout << myName;
7.
8.     return 0;
9. }
```

Язык C++ поддерживает еще один способ создания символьных констант строк C-style — через указатели:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     const char *myName = "John";
6.     std::cout << myName;
7.
8.     return 0;
9. }
```

Хотя обе эти программы работают и выдают одинаковые результаты, выделение памяти в них выполняется по-разному.

В первом случае в программе выделяется память для фиксированного массива длиной 5 и инициализируется эта память строкой `John\0`. Поскольку память была специально выделена для массива, то мы можем изменять её содержимое. Сам массив рассматривается как обычная локальная переменная, поэтому, когда он выходит из области видимости, память, используемая им, освобождается для других объектов.

Что происходит в случае с символьной константой? Компилятор помещает строку `John\0` в память типа read-only (только чтение), а затем создает указатель, который указывает на эту строку. Несколько строковых литералов с одним и тем же содержимым могут указывать на один и тот же адрес. Поскольку эта память доступна только для чтения, а также потому, что внесение изменений в строковый литерал может повлиять на дальнейшее его использование, лучше всего перестраховаться, объявив строку константой (типа const). Также, поскольку строки, объявленные таким образом, существуют на протяжении всей жизни программы (они имеют статическую продолжительность, а не автоматическую, как большинство других локально определенных литералов), нам не нужно

беспокоиться о проблемах, связанных с областью видимости. Поэтому следующее в порядке вещей:

```
const char* getName()
{
    return "John";
}
```

В фрагменте, приведенном выше, функция getName() возвращает указатель на строку C-style `John`. Всё хорошо, так как `John` не выходит из области видимости, когда getName() завершает свое выполнение, поэтому вызывающий объект все равно имеет доступ к строке.

std::cout и указатели типа char

На этом этапе вы, возможно, уже успели заметить то, как std::cout обрабатывает указатели разных типов. Рассмотрим следующий пример:

```
#include <iostream>

int main()
{
    int nArray[5] = { 9, 7, 5, 3, 1 };
    char cArray[] = "Hello!";
    const char *name = "John";

    std::cout << nArray << '\n'; // nArray распадается в указатель типа int
    std::cout << cArray << '\n'; // cArray распадается в указатель типа char
    std::cout << name << '\n'; // name уже и так является указателем типа char

    return 0;
}
```

Результат выполнения программы на моем компьютере:

```
0046FAE8
Hello!
John
```

Почему в массиве типа int выводится адрес, а в массивах типа char - строки?

Дело в том, что при передаче указателя не типа char, в результате выводится просто содержимое этого указателя (адрес памяти). Однако, если вы передадите объект типа char* или const char*, то std::cout предположит, что вы намереваетесь вывести строку. Следовательно, вместо вывода значения указателя — выведется строка, на которую тот указывает!

Хотя это всё замечательно в 99% случаев, но это может привести и к неожиданным результатам, например:

```
#include <iostream>

int main()
{
    char a = 'R';
    std::cout << &a;

    return 0;
}
```

Здесь мы намереваемся вывести адрес переменной `a`. Тем не менее, `&a` имеет тип `char*`, поэтому std::cout выведет это как строку!

Результат выполнения программы на моем компьютере:

`R╠╠╠╠╣4;¿■A`

Почему так? std::cout предположил, что `&a` (типа `char*`) является строкой. Поэтому сначала вывелось `R`, а затем вывод продолжился. Следующим в памяти был мусор. В конце концов, std::cout столкнулся с ячейкой памяти, имеющей значение `0`, которое он интерпретировал как нуль-терминатор, и, соответственно, прекратил вывод. То, что вы видите в результате, может отличаться, в зависимости от того, что находится в памяти после переменной `a`.

Подобное вряд ли случится с вами на практике (так как вы вряд ли захотите выводить адреса памяти), но это хорошая демонстрация того, как всё работает "под капотом" и как программы могут *случайно* "сойти с рельсов".

Урок №89. Динамическое выделение памяти

Язык C++ поддерживает три основных типа **выделения** (или ***"распределения"***) **памяти**, с двумя из которых, мы уже знакомы:

- **Статическое выделение памяти** выполняется для статических и глобальных переменных. Память выделяется один раз (при запуске программы) и сохраняется на протяжении работы всей программы.
- **Автоматическое выделение памяти** выполняется для параметров функции и локальных переменных. Память выделяется при входе в блок, в котором находятся эти переменные, и удаляется при выходе из него.
- **Динамическое выделение памяти** является темой этого урока.

Динамическое выделение переменных

Как статическое, так и автоматическое распределение памяти имеют два общих свойства:

- Размер переменной/массива должен быть известен во время компиляции.
- Выделение и освобождение памяти происходит автоматически (когда переменная создается/уничтожается).

В большинстве случаев с этим всё ОК. Однако, когда дело доходит до работы с пользовательским вводом, то эти ограничения могут привести к проблемам.

Например, при использовании строки для хранения имени пользователя, мы не знаем наперед насколько длинным оно будет, пока пользователь его не введет. Или нам нужно создать игру с непостоянным количеством монстров (во время игры одни монстры умирают, другие появляются, пытаясь, таким образом, убить игрока).

Если нам нужно объявить размер всех переменных во время компиляции, то самое лучшее, что мы можем сделать — это попытаться угадать их максимальный размер, надеясь, что этого будет достаточно:

```
1. char name[30]; // будем надеяться, что пользователь введет имя длиной менее 30 символов!
2. Monster monster[30]; // 30 монстров максимум
3. Polygon rendering[40000]; // этому 3D-рендерингу лучше состоять из менее чем 40000 полигонов!
```

Это плохое решение, по крайней мере, по трем причинам:

Во-первых, теряется память, если переменные фактически не используются или используются, но не все. Например, если мы выделим 30 символов для каждого имени, но имена в среднем будут занимать по 15 символов, то потребление памяти получится в два раза больше, чем нам нужно на самом деле. Или рассмотрим массив `rendering`: если он использует только 20 000 полигонов, то память для других 20 000 полигонов фактически тратится впустую (т.е. не используется)!

Во-вторых, память для большинства обычных переменных (включая фиксированные массивы) выделяется из специального резервуара памяти - **стека**. Объем памяти стека в программе, как правило, невелик: в Visual Studio он по умолчанию равен 1МБ. Если вы превысите это значение, то произойдет *переполнение стека*, и операционная система автоматически завершит выполнение вашей программы.

В Visual Studio это можно проверить, запустив следующий фрагмент кода:

```
1. int main()
2. {
3.     int array[1000000000]; // выделяем 1 миллиард целочисленных значений
4. }
```

Лимит в 1МБ памяти может быть проблематичным для многих программ, особенно где используется графика.

В-третьих, и самое главное, это может привести к искусственным ограничениям и/или переполнению массива. Что произойдет, если пользователь попытается прочесть 500 записей с диска, но мы выделили память максимум для 400? Либо мы выведем пользователю ошибку, что максимальное количество записей - 400, либо (в худшем случае) выполнится переполнение массива и затем что-то очень нехорошее.

К счастью, эти проблемы легко устраняются с помощью динамического выделения памяти. **Динамическое выделение памяти** - это способ запроса памяти из операционной системы запущенными программами по мере необходимости. Эта память не выделяется из ограниченной памяти стека программы, а выделяется из гораздо большего хранилища, управляемого операционной системой - **кучи**. На современных компьютерах размер кучи может составлять гигабайты памяти.

Для динамического выделения памяти одной переменной используется **оператор new**:

```
1. new int; // динамически выделяем целочисленную переменную и сразу же
   отбрасываем результат (так как нигде его не сохраняем)
```

В примере, приведенном выше, мы запрашиваем выделение памяти для целочисленной переменной из операционной системы. Оператор new возвращает указатель, содержащий адрес выделенной памяти.

Для доступа к выделенной памяти создается указатель:

```
1. int *ptr = new int; // динамически выделяем целочисленную переменную и
   присваиваем её адрес ptr, чтобы затем иметь доступ к ней
```

Затем мы можем разыменовать указатель для получения значения:

```
1. *ptr = 8; // присваиваем значение 8 только что выделенной памяти
```

Вот один из случаев, когда указатели полезны. Без указателя с адресом на только что выделенную память у нас не было бы способа получить доступ к ней.

Как работает динамическое выделение памяти?

На вашем компьютере имеется память (возможно, большая её часть), которая доступна для использования программами. При запуске программы ваша операционная система загружает эту программу в некоторую часть этой памяти. И эта память, используемая вашей программой, разделена на несколько частей, каждая из которых выполняет определенную задачу. Одна часть содержит ваш код, другая используется для выполнения обычных операций (отслеживание вызываемых функций, создание и уничтожение глобальных и локальных переменных и т.д.). Мы поговорим об этом чуть позже. Тем не менее, большая часть доступной памяти компьютера просто находится в ожидании запросов на выделение от программ.

Когда вы динамически выделяете память, то вы просите операционную систему зарезервировать часть этой памяти для использования вашей программой. Если ОС может выполнить этот запрос, то возвращается адрес этой памяти обратно в вашу программу. С этого момента и в дальнейшем ваша программа сможет использовать эту память, как только пожелает. Когда вы уже выполнили с этой памятью всё, что было необходимо, то её нужно вернуть обратно в операционную систему, для распределения между другими запросами.

В отличие от статического или автоматического выделения памяти, программа самостоятельно отвечает за запрос и обратный возврат динамически выделенной памяти.

Освобождение памяти

Когда вы динамически выделяете переменную, то вы также можете её инициализировать посредством прямой инициализации или uniform-инициализации (в C++11):

```
1. int *ptr1 = new int (7); // используем прямую инициализацию
2. int *ptr2 = new int { 8 }; // используем uniform-инициализацию
```

Когда уже всё, что требовалось, выполнено с динамически выделенной переменной — нужно явно указать для C++ освободить эту память. Для переменных это выполняется с помощью **оператора delete**:

```
1. // Предположим, что ptr ранее уже был выделен с помощью оператора new
2. delete ptr; // возвращаем память, на которую указывал ptr, обратно в
   операционную систему
3. ptr = 0; // делаем ptr нулевым указателем (используйте nullptr вместо 0 в
   C++11)
```

Оператор delete на самом деле ничего не удаляет. Он просто возвращает память, которая была выделена ранее, обратно в операционную систему. Затем операционная система может переназначить эту память другому приложению (или этому же снова).

Хотя может показаться, что мы удаляем *переменную*, но это не так! Переменная-указатель по-прежнему имеет ту же область видимости, что и раньше, и ей можно присвоить новое значение, как и любой другой переменной.

Обратите внимание, удаление указателя, не указывающего на динамически выделенную память, может привести к проблемам.

Висячие указатели

Язык C++ не предоставляет никаких гарантий относительно того, что произойдет с содержимым освобожденной памяти или со значением удаляемого указателя. В большинстве случаев, память, возвращаемая операционной системе, будет содержать те же значения, которые были у нее до *освобождения*, а указатель так и останется указывать на только что освобожденную (удаленную) память.

Указатель, указывающий на освобожденную память, называется **висячим указателем**. Разыменование или удаление висячего указателя приведет к неожиданным результатам. Рассмотрим следующую программу:

```
1. #include <iostream>
2.
```

```
3. int main()
4. {
5.     int *ptr = new int; // динамически выделяем целочисленную переменную
6.     *ptr = 8; // помещаем значение в выделенную ячейку памяти
7.
8.     delete ptr; // возвращаем память обратно в операционную систему, ptr
   теперь является висячим указателем
9.
10.    std::cout << *ptr; // разыменование висячего указателя приведет к
   неожиданным результатам
11.    delete ptr; // попытка освободить память снова приведет к неожиданным
   результатам также
12.
13.    return 0;
14. }
```

В программе, приведенной выше, значение `8`, которое ранее было присвоено динамической переменной, после освобождения может и далее находиться там, а может и нет. Также возможно, что освобожденная память уже могла быть выделена другому приложению (или для собственного использования операционной системы), и попытка доступа к ней приведет к тому, что операционная система автоматически прекратит выполнение вашей программы.

Процесс освобождения памяти может также привести и к созданию *нескольких* висячих указателей. Рассмотрим следующий пример:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int *ptr = new int; // динамически выделяем целочисленную переменную
6.     int *otherPtr = ptr; // otherPtr теперь указывает на ту же самую
   выделенную память, что и ptr
7.
8.     delete ptr; // возвращаем память обратно в операционную систему. ptr и
   otherPtr теперь висячие указатели
9.     ptr = 0; // ptr теперь уже nullptr
10.
11.    // Однако, otherPtr по-прежнему является висячим указателем!
12.
13.    return 0;
14. }
```

Есть несколько рекомендаций, которые могут здесь помочь:

- Во-первых, старайтесь избегать ситуаций, когда несколько указателей указывают на одну и ту же часть выделенной памяти. Если это невозможно, то выясните, какой указатель из всех «владеет» памятью (и отвечает за её удаление), а какие указатели просто получают доступ к ней.
- Во-вторых, когда вы удаляете указатель, и, если он не выходит из области видимости сразу же после удаления, то его нужно сделать нулевым, т.е. присвоить значение `0` (или nullptr в C++11). Под "выходом из области

видимости сразу же после удаления" имеется в виду, что вы удаляете указатель в самом конце блока, в котором он объявлен.

Правило: Присваивайте удаленным указателям значение 0 (или nullptr в C++11), если они не выходят из области видимости сразу же после удаления.

Оператор new

При запросе памяти из операционной системы в редких случаях она может быть не выделена (т.е. её может и не быть в наличии).

По умолчанию, если оператор new не сработал, память не выделилась, то генерируется *исключение* `bad_alloc`. Если это исключение будет неправильно обработано (а именно так и будет, поскольку мы еще не рассматривали исключения и их обработку), то программа просто прекратит свое выполнение (произойдет сбой) с ошибкой необработанного исключения.

Во многих случаях процесс генерации исключения оператором new (как и сбой программы) нежелателен, поэтому есть альтернативная форма оператора new, которая возвращает нулевой указатель, если память не может быть выделена. Нужно просто добавить **константу std::nothrow** между ключевым словом new и типом данных:

```
1. int *value = new (std::nothrow) int; // указатель value станет нулевым, если
   динамическое выделение целочисленной переменной не выполнится
```

В примере, приведенном выше, если оператор new не возвратит указатель с динамически выделенной памятью, то возвратится нулевой указатель.

Разыменовывать его также не рекомендуется, так как это приведет к неожиданным результатам (скорее всего, к сбою в программе). Поэтому наилучшей практикой является проверка всех запросов на выделение памяти для обеспечения того, что эти запросы будут выполнены успешно и память выделится:

```
1. int *value = new (std::nothrow) int; // запрос на выделение динамической
   памяти для целочисленного значения
2. if (!value) // обрабатываем случай, когда new возвращает null (т.е. память не
   выделяется)
3. {
4.     // Обработка этого случая
5.     std::cout << "Could not allocate memory";
6. }
```

Поскольку не выделение памяти оператором new происходит крайне редко, то обычно программисты забывают выполнять эту проверку!

Нулевые указатели и динамическое выделение памяти

Нулевые указатели (указатели со значением `0` или `nullptr`) особенно полезны в процессе динамического выделения памяти. Их наличие как бы сообщаем нам: «Этому указателю не выделено никакой памяти». А это, в свою очередь, можно использовать для выполнения условного выделения памяти:

```
1. // Если для ptr до сих пор не выделено памяти, то выделяем её
2. if (!ptr)
3.     ptr = new int;
```

Удаление нулевого указателя ни на что не влияет. Таким образом, в следующем нет необходимости:

```
1. if (ptr)
2.     delete ptr;
```

Вместо этого вы можете просто написать:

```
1. delete ptr;
```

Если `ptr` не является нулевым, то динамически выделенная переменная будет удалена. Если значением указателя является нуль, то ничего не произойдет.

Утечка памяти

Динамически выделенная память не имеет области видимости, т.е. она остается выделенной до тех пор, пока не будет явно освобождена или пока ваша программа не завершит свое выполнение (и операционная система очистит все буфера памяти самостоятельно). Однако указатели, используемые для хранения динамически выделенных адресов памяти, следуют правилам области видимости обычных переменных. Это несоответствие может вызвать интересное поведение, например:

```
1. void doSomething()
2. {
3.     int *ptr = new int;
4. }
```

Здесь мы динамически выделяем целочисленную переменную, но никогда не освобождаем память через использование оператора delete. Поскольку указатели следуют всем тем же правилам, что и обычные переменные, то, когда функция завершит свое выполнение, `ptr` выйдет из области видимости. Поскольку `ptr` — это единственная переменная, хранящая адрес динамически выделенной целочисленной переменной, то, когда `ptr` уничтожится, больше не останется указателей на динамически выделенную память. Это означает, что программа

«потеряет» адрес динамически выделенной памяти. И в результате эту динамически выделенную целочисленную переменную нельзя будет удалить.

Это называется **утечкой памяти**. Утечка памяти происходит, когда ваша программа теряет адрес некоторой динамически выделенной части памяти (например, переменной или массива), прежде чем вернуть её обратно в операционную систему. Когда это происходит, то программа уже не может удалить эту динамически выделенную память, поскольку она больше не знает, где она находится. Операционная система также не может использовать эту память, поскольку считается, что она по-прежнему используется вашей программой.

Утечки памяти "съедают" свободную память во время выполнения программы, уменьшая количество доступной памяти не только для этой программы, но и для других программ также. Программы с серьезными проблемами с утечкой памяти могут "съесть" всю доступную память, в результате чего ваш компьютер будет медленнее работать или даже произойдет сбой. Только после того, как выполнение вашей программы завершится, операционная система сможет очистить и *вернуть* всю память, которая "утекла".

Хотя утечка памяти может возникнуть и из-за того, что указатель выходит из области видимости, возможны и другие способы, которые могут привести к утечкам памяти. Например, если указателю, хранящему адрес динамически выделенной памяти, присвоить другое значение:

```
1. int value = 7;
2. int *ptr = new int; // выделяем память
3. ptr = &value; // старый адрес утерян - произойдет утечка памяти
```

Это легко решается удалением указателя перед операцией переприсваивания:

```
1. int value = 7;
2. int *ptr = new int; // выделяем память
3. delete ptr; // возвращаем память обратно в операционную систему
4. ptr = &value; // переприсваиваем указателю адрес value
```

Кроме того, утечка памяти также может произойти и через двойное выделение памяти:

```
1. int *ptr = new int;
2. ptr = new int; // старый адрес утерян - произойдёт утечка памяти
```

Адрес, возвращаемый из второго выделения памяти, перезаписывает адрес из первого выделения. Следовательно, первое динамическое выделение становится утечкой памяти!

Точно так же этого можно избежать удалением указателя перед операцией переприсваивания.

Заключение

С помощью операторов new и delete можно динамически выделять отдельные переменные в программе. Динамически выделенная память не имеет области видимости и остается выделенной до тех пор, пока не произойдет её освобождение или пока программа не завершит свое выполнение. Будьте осторожны, не разыменовывайте висячие или нулевые указатели.

На следующем уроке мы рассмотрим использование операторов new и delete для выделения и удаления динамически выделенных массивов.

Урок №90. Динамические массивы

Помимо динамического выделения переменных мы также можем динамически выделять и массивы. В отличие от фиксированного массива, где его размер должен быть известен во время компиляции, динамическое выделение массива в языке C++ позволяет нам устанавливать его длину во время выполнения программы.

Динамические массивы

Для выделения динамического массива и работы с ним используются отдельные формы операторов new и delete: `new[]` и `delete[]`.

```
#include <iostream>

int main()
{
    std::cout << "Enter a positive integer: ";
    int length;
    std::cin >> length;

    int *array = new int[length]; // используем оператор new[] для выделения массива. Обратите внимание, переменная length не обязательно должна быть константой!

    std::cout << "I just allocated an array of integers of length " << length << '\n';

    array[0] = 7; // присваиваем элементу под индексом 0 значение 7

    delete[] array; // используем оператор delete[] для освобождения выделенной массиву памяти
    array = 0; // используйте nullptr вместо 0 в C++11

    return 0;
}
```

Поскольку мы выделяем массив, то C++ понимает, что он должен использовать другую форму оператора new — форму для массива, а не для переменной. По факту вызывается оператор `new[]`, даже если мы и не указываем `[]` сразу после ключевого слова `new`.

Обратите внимание, поскольку память для динамических и фиксированных массивов выделяется из разных "резервуаров", то размер динамического массива может быть довольно большим. Вы можете запустить программу, приведенную выше, но уже выделить массив длиной 1 000 000 (или, возможно, даже 100 000 000) без проблем. Попробуйте!

Удаление динамического массива

При удалении динамических массивов также используется форма оператора delete для массивов — `delete[]`. Таким образом, мы сообщаем процессору, что ему нужно очистить память от нескольких переменных вместо одной. Самая распространенная ошибка, которую совершают новички при работе с динамическим выделением памяти, является использование delete вместо `delete[]` для удаления динамических массивов. Использование формы оператора delete для переменных при удалении массива приведет к таким неожиданным результатам, как повреждение данных, утечка памяти, сбой или другие проблемы.

Инициализация динамических массивов

Если вы хотите инициализировать динамический массив значением `0`, то всё довольно просто:

```
1. int *array = new int[length]();
```

До C++11 не было простого способа инициализировать динамический массив ненулевыми значениями (список инициализаторов работал только с фиксированными массивами). А это означает, что нужно перебрать каждый элемент массива и присвоить ему значение явным указанием:

```
1. int *array = new int[5];
2. array[0] = 9;
3. array[1] = 7;
4. array[2] = 5;
5. array[3] = 3;
6. array[4] = 1;
```

Немного утомляет, не правда ли?

Однако, начиная с C++11, появилась возможность инициализации динамических массивов через списки инициализаторов:

```
1. int fixedArray[5] = { 9, 7, 5, 3, 1 }; // инициализируем фиксированный массив
2. int *array = new int[5] { 9, 7, 5, 3, 1 }; // инициализируем динамический
   массив
```

Обратите внимание, в синтаксисе динамического массива между длиной массива и списком инициализаторов оператора присваивания (`=`) нет.

В C++11 фиксированные массивы также могут быть инициализированы с использованием uniform-инициализации:

```
1. int fixedArray[5] { 9, 7, 5, 3, 1 }; // инициализируем фиксированный массив в C++11
2. char fixedArray[14] { "Hello, world!" }; // инициализируем фиксированный массив в C++11
```

Однако, будьте осторожны, так как в C++11 вы не можете инициализировать динамический массив символов строкой C-style:

```
1. char *array = new char[14] { "Hello, world!" }; // не работает в C++11
```

Вместо этого вы можете динамически выделить std::string (или выделить динамический массив символов, а затем с помощью функции strcpy_s() скопировать содержимое нужной строки в этот массив).

Также обратите внимание на то, что динамические массивы должны быть объявлены с явным указанием их длины:

```
1. int fixedArray[] {1, 2, 3}; // ок: неявное указание длины фиксированного массива
2.
3. int *dynamicArray1 = new int[] {1, 2, 3}; // не ок: неявное указание длины динамического массива
4.
5. int *dynamicArray2 = new int[3] {1, 2, 3}; // ок: явное указание длины динамического массива
```

Изменение длины массивов

Динамическое выделение массивов позволяет задавать их длину во время выделения. Однако C++ не предоставляет встроенный способ изменения длины массива, который уже был выделен. Но и это ограничение можно обойти, динамически выделив новый массив, скопировав все элементы из старого массива, а затем удалив старый массив. Однако этот способ подвержен ошибкам (об этом чуть позже).

К счастью, в языке C++ есть массивы, размер которых можно изменять, и называются они векторами (std::vector). О них мы поговорим на соответствующем уроке.

Тест

Напишите программу, которая:

- спрашивает у пользователя, сколько имен он хочет ввести;

- просит пользователя ввести каждое имя;
- вызывает функцию для сортировки имен в алфавитном порядке;
- выводит отсортированный список имен.

> ***Подсказки:***
>
> - Используйте динамическое выделение std::string для хранения имен.
> - std::string поддерживает сравнение строк с помощью операторов сравнения `<` и `>`.

Пример результата выполнения вашей программы:

```
How many names would you like to enter? 5
Enter name #1: Jason
Enter name #2: Mark
Enter name #3: Alex
Enter name #4: Chris
Enter name #5: John

Here is your sorted list:
Name #1: Alex
Name #2: Chris
Name #3: Jason
Name #4: John
Name #5: Mark
```

Урок №91. Указатели и const

До этого момента все указатели, которые мы рассматривали, были неконстантными указателями на неконстантные значения:

```
1. int value = 7;
2. int *ptr = &value;
3. *ptr = 8; // изменяем значение value на 8
```

Однако, что произойдет, если указатель будет указывать на константную переменную?

```
1. const int value = 7; // value - это константа
2. int *ptr = &value; // ошибка компиляции: невозможно конвертировать const int*
   в int*
3. *ptr = 8; // изменяем значение value на 8
```

Фрагмент кода, приведенный выше, не скомпилируется: мы не можем присвоить неконстантному указателю константную переменную. Здесь есть смысл, ведь на то она и константа, что её значение нельзя изменить. Гипотетически, если бы мы могли присвоить константное значение неконстантному указателю, то тогда мы могли бы разыменовать неконстантный указатель и изменить значение этой же константы. А это уже является нарушением самого понятия «константа».

Указатели на константные значения

Указатель на константное значение - это неконстантный указатель, который указывает на неизменное значение. Для объявления указателя на константное значение, используется **ключевое слово const перед типом данных**:

```
1. const int value = 7;
2. const int *ptr = &value; // здесь всё ок: ptr - это неконстантный указатель,
   который указывает на "const int"
3. *ptr = 8; // нельзя, мы не можем изменить константное значение
```

В примере, приведенном выше, `ptr` указывает на константный целочисленный тип данных.

Пока что всё хорошо. Рассмотрим следующий пример:

```
1. int value = 7; // value - это не константа
2. const int *ptr = &value; // всё хорошо
```

Указатель на константную переменную может указывать и на неконстантную переменную (как в случае с переменной `value` в примере, приведенном выше). Подумайте об этом так: указатель на константную переменную обрабатывает

переменную как константу при получении доступа к ней независимо от того, была ли эта переменная изначально определена как const или нет. Таким образом, следующее в порядке вещей:

```
1. int value = 7;
2. const int *ptr = &value; // ptr указывает на "const int"
3. value = 8; // переменная value уже не константа, если к ней получают доступ
   через неконстантный идентификатор
```

Но не следующее:

```
1. int value = 7;
2. const int *ptr = &value; // ptr указывает на "const int"
3. *ptr = 8; // ptr обрабатывает value как константу, поэтому изменение значения
   переменной value через ptr не допускается
```

Указателю на константное значение, который сам при этом не является константным (он просто указывает на константное значение), можно присвоить и другое значение:

```
1. int value1 = 7;
2. const int *ptr = &value1; // ptr указывает на const int
3.
4. int value2 = 8;
5. ptr = &value2; // хорошо, ptr теперь указывает на другой const int
```

Константные указатели

Мы также можем сделать указатель константным. **Константный указатель** - это указатель, значение которого не может быть изменено после инициализации. Для объявления константного указателя используется **ключевое слово const между звёздочкой и именем указателя**:

```
1. int value = 7;
2. int *const ptr = &value;
```

Подобно обычным константным переменным, константный указатель должен быть инициализирован значением при объявлении. Это означает, что он всегда будет указывать на один и тот же адрес. В вышеприведенном примере `ptr` всегда будет указывать на адрес `value` (до тех пор, пока указатель не выйдет из области видимости и не уничтожится):

```
1. int value1 = 7;
2. int value2 = 8;
3.
4. int * const ptr = &value1; // ок: константный указатель инициализирован
   адресом value1
5. ptr = &value2; // не ок: после инициализации константный указатель не может
   быть изменен
```

Однако, поскольку переменная `value`, на которую указывает указатель, не является константой, то её значение можно изменить путем разыменования константного указателя:

```
1. int value = 7;
2. int *const ptr = &value; // ptr всегда будет указывать на value
3. *ptr = 8; // ок, так как ptr указывает на тип данных (неконстантный int)
```

Константные указатели на константные значения

Наконец, можно объявить константный указатель на константное значение, используя **ключевое слово const как перед типом данных, так и перед именем указателя**:

```
1. int value = 7;
2. const int *const ptr = &value;
```

Константный указатель на константное значение нельзя перенаправить указывать на другое значение также, как и значение, на которое он указывает, — нельзя изменить.

Заключение

Подводя итоги, вам нужно запомнить всего лишь 4 правила:

- Неконстантный указатель можно перенаправить указывать на любой другой адрес.
- С помощью указателя на неконстантное значение можно изменить это же значение (на которое он указывает).
- Константный указатель всегда указывает на один и тот же адрес, и этот адрес не может быть изменен.
- Указатель на константное значение обрабатывает значение как константное (даже если оно таковым не является) и, следовательно, это значение через указатель изменить нельзя.

А вот с синтаксисом может быть немного труднее. Просто помните, что тип значения, на который указывает указатель, всегда находится слева (в самом начале):

```
1. int value = 7;
2. const int *ptr1 = &value; // ptr1 указывает на "const int", поэтому это
   указатель на константное значение
3. int *const ptr2 = &value; // ptr2 указывает на "int", поэтому это константный
   указатель на неконстантное значение
4. const int *const ptr3 = &value; // ptr3 указывает на "const int", поэтому это
   константный указатель на константное значение
```

Указатели на константные значения в основном используются в параметрах функций (например, при передаче массива) для гарантии того, что функция случайно не изменит значение(я) переданного ей аргумента.

Урок №92. Ссылки

До этого момента мы успели рассмотреть 2 основных типа переменных:

- обычные переменные, которые хранят значения напрямую;
- указатели, которые хранят адрес другого значения (или null), для доступа к которым выполняется операция разыменования указателя.

Ссылки - это третий базовый тип переменных в языке C++.

Ссылки

Ссылка - это тип переменной в языке C++, который работает как псевдоним другого объекта или значения. **Язык C++ поддерживает 3 типа ссылок:**

- **Ссылки на неконстантные значения** (обычно их называют просто *«ссылки»* или *«неконстантные ссылки»*), которые мы обсудим на этом уроке.
- **Ссылки на константные значения** (обычно их называют *«константные ссылки»*), которые мы обсудим на следующем уроке.
- В C++11 добавлены **ссылки r-value**, о которых мы поговорим чуть позже.

Ссылка (на неконстантное значение) объявляется с использованием амперсанда (`&`) между типом данных и именем ссылки:

```
1. int value = 7; // обычная переменная
2. int &ref = value; // ссылка на переменную value
```

В этом контексте амперсанд не означает «оператор адреса», он означает «ссылка на».

Ссылки в качестве псевдонимов

Ссылки обычно ведут себя идентично значениям, на которые они ссылаются. В этом смысле ссылка работает как псевдоним объекта, на который она ссылается, например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int value = 7; // обычная переменная
6.     int &ref = value; // ссылка на переменную value
7.
8.     value = 8; // value теперь 8
9.     ref = 9; // value теперь 9
```

```
10.
11.     std::cout << value << std::endl; // выведется 9
12.     ++ref;
13.     std::cout << value << std::endl; // выведется 10
14.
15.     return 0;
16. }
```

Результат выполнения программы:

```
9
10
```

В примере, приведенном выше, объекты `ref` и `value` обрабатываются как одно целое. Использование оператора адреса с ссылкой приведет к возврату адреса значения, на которое ссылается ссылка:

```
1. std::cout << &value; // выведется 0035FE58
2. std::cout << &ref; // выведется 0035FE58
```

Краткий обзор l-value и r-value

l-value - это объект, который имеет определенный адрес памяти (например, переменная `x`) и сохраняется за пределами одного выражения. r-value - это временное значение без определенного адреса памяти и с областью видимости выражения (т.е. сохраняется в пределах одного выражения). В качестве r-values могут быть как результаты выражения (например, `2 + 3`), так и литералы.

Инициализация ссылок

Ссылки должны быть инициализированы при создании:

```
1. int value = 7;
2. int &ref = value; // корректная ссылка: инициализирована переменной value
3.
4. int &invalidRef; // некорректная ссылка: ссылка должна ссылаться на что-либо
```

В отличие от указателей, которые могут содержать нулевое значение, ссылки нулевыми быть не могут.

Ссылки на неконстантные значения могут быть инициализированы только неконстантными l-values. Они не могут быть инициализированы константными l-values или r-values:

```
1. int a = 7;
2. int &ref1 = a; // ок: a - это неконстантное l-value
3.
4. const int b = 8;
5. int &ref2 = b; // не ок: b - это константное l-value
```

```
6.
7.  int &ref3 = 4; // не ок: 4 - это r-value
```

Обратите внимание, во втором случае вы не можете инициализировать неконстантную ссылку константным объектом. В противном случае, вы бы могли изменить значение константного объекта через ссылку, что уже является нарушением понятия «константа».

После инициализации изменить объект, на который указывает ссылка — нельзя. Рассмотрим следующий фрагмент кода:

```
1.  int value1 = 7;
2.  int value2 = 8;
3.
4.  int &ref = value1; // ок: ref - теперь псевдоним для value1
5.  ref = value2; // присваиваем 8 (значение переменной value2) переменной value1.
    Здесь НЕ изменяется объект, на который ссылается ссылка!
```

Обратите внимание, во втором стейтменте (`ref = value2;`) выполняется не то, что вы могли бы ожидать! Вместо переприсваивания `ref` (ссылаться на переменную `value2`), значение из `value2` присваивается переменной `value1` (на которое и ссылается `ref`).

Ссылки в качестве параметров в функциях

Ссылки чаще всего используются в качестве параметров в функциях. В этом контексте ссылка-параметр работает как псевдоним аргумента, а сам аргумент не копируется при передаче в параметр. Это в свою очередь улучшает производительность, если аргумент слишком большой или затратный для копирования.

Ранее мы говорили о том, что передача аргумента-указателя в функцию позволяет функции при разыменовании этого указателя *напрямую* изменять значение аргумента.

Ссылки работают аналогично. Поскольку ссылка-параметр - это псевдоним аргумента, то функция, использующая ссылку-параметр, может изменять аргумент, переданный ей, также *напрямую*:

```
1.  #include <iostream>
2.
3.  // ref - это ссылка на переданный аргумент, а не копия аргумента
4.  void changeN(int &ref)
5.  {
6.      ref = 8;
7.  }
8.
9.  int main()
```

```
{
    int x = 7;

    std::cout << x << '\n';

    changeN(x); // обратите внимание, этот аргумент не обязательно должен быть ссылкой

    std::cout << x << '\n';
    return 0;
}
```

Результат выполнения программы:

```
7
8
```

Когда аргумент `x` передается в функцию, то параметр функции `ref` становится ссылкой на аргумент `x`. Это позволяет функции изменять значение `x` непосредственно через `ref`! Обратите внимание, переменная `x` не обязательно должна быть ссылкой.

> ***Совет:*** Передавайте аргументы в функцию через неконстантные ссылки-параметры, если они должны быть изменены функцией в дальнейшем.

Основным недостатком использования неконстантных ссылок в качестве параметров в функциях является то, что аргумент должен быть неконстантным l-value (т.е. константой или литералом он быть не может). Мы поговорим об этом подробнее (и о том, как это обойти) на следующем уроке.

Ссылки как более легкий способ доступа к данным

Второе (гораздо менее используемое) применение ссылок заключается в более легком способе доступа к вложенным данным. Рассмотрим следующую структуру:

```
struct Something
{
    int value1;
    float value2;
};

struct Other
{
    Something something;
    int otherValue;
};

Other other;
```

Предположим, что нам нужно работать с полем `value1` структуры `Something` переменной `other` структуры `Other` (звучит сложно, но такое также встречается на практике). Обычно, доступ к этому полю осуществлялся бы через `other.something.value1`. А что, если нам нужно неоднократно получать доступ к этому члену? В этом случае код становится громоздким и беспорядочным. Ссылки же предоставляют более легкий способ доступа к данным:

```
1. int &ref = other.something.value1;
2. // ref теперь может использоваться вместо other.something.value1
```

Таким образом, следующие два стейтмента идентичны:

```
1. other.something.value1 = 7;
2. ref = 7;
```

Ссылки позволяют сделать ваш код более чистым и понятным.

Ссылки vs. Указатели

Ссылка — это тот же указатель, который неявно разыменовывается при доступе к значению, на которое он указывает ("под капотом" ссылки реализованы с помощью указателей). Таким образом, в следующем коде:

```
1. int value = 7;
2. int *const ptr = &value;
3. int &ref = value;
```

`*ptr` и `ref` обрабатываются одинаково. Т.е. это одно и то же:

```
1. *ptr = 7;
2. ref = 7;
```

Поскольку ссылки должны быть инициализированы корректными объектами (они не могут быть нулевыми) и не могут быть изменены позже, то они, как правило, безопаснее указателей (так как риск разыменования нулевого указателя отпадает). Однако они немного ограничены в функциональности по сравнению с указателями.

Если определенное задание может быть решено с помощью как ссылок, так и указателей, то лучше использовать ссылки. Указатели следует использовать только в тех ситуациях, когда ссылки являются недостаточно эффективными (например, при динамическом выделении памяти).

Заключение

Ссылки позволяют определять псевдонимы для других объектов или значений. Ссылки на неконстантные значения могут быть инициализированы только

неконстантными l-values. Они не могут быть переприсвоены после инициализации. Ссылки чаще всего используются в качестве параметров в функциях, когда мы хотим изменить значение аргумента или хотим избежать его затратного копирования.

Урок №93. Ссылки и const

Так же, как можно объявить указатель на константное значение, так же можно объявить и ссылку на константное значение в языке C++.

Ссылки на константные значения

Объявить ссылку на константное значение можно путем добавления **ключевого слова const перед типом данных**:

```
1. const int value = 7;
2. const int &ref = value; // ref - это ссылка на константную переменную value
```

Ссылки на константные значения часто называют просто ***«ссылки на константы»*** или ***«константные ссылки»***.

Инициализация ссылок на константы

В отличие от ссылок на неконстантные значения, которые могут быть инициализированы только неконстантными l-values, ссылки на константные значения могут быть инициализированы неконстантными l-values, константными l-values и r-values:

```
1. int a = 7;
2. const int &ref1 = a; // ок: a - это неконстантное l-value
3.
4. const int b = 9;
5. const int &ref2 = b; // ок: b - это константное l-value
6.
7. const int &ref3 = 5; // ок: 5 - это r-value
```

Как и в случае с указателями, константные ссылки также могут ссылаться и на неконстантные переменные. При доступе к значению через константную ссылку, это значение автоматически считается const, даже если исходная переменная таковой не является:

```
1. int value = 7;
2. const int &ref = value; // создаем константную ссылку на переменную value
3.
4. value = 8; // ок: value - это не константа
5. ref = 9; // нельзя: ref - это константа
```

Ссылки r-values

Обычно r-values имеют область видимости выражения, что означает, что они уничтожаются в конце выражения, в котором созданы:

```
1. std::cout << 3 + 4; // 3 + 4 вычисляется в r-value 7, которое уничтожается в конце этого стейтмента
```

Однако, когда константная ссылка инициализируется значением r-value, время жизни r-value продлевается в соответствии со временем жизни ссылки:

```
1. int somefcn()
2. {
3.     const int &ref = 3 + 4; // обычно результат 3 + 4 имеет область видимости выражения и уничтожился бы в конце этого стейтмента, но, поскольку результат выражения сейчас привязан к ссылке на константное значение,
4.     std::cout << ref; // мы можем использовать его здесь
5. } // и время жизни r-value продлевается до этой точки, когда константная ссылка уничтожается
```

Константные ссылки в качестве параметров функции

Ссылки, используемые в качестве параметров функции, также могут быть константными. Это позволяет получить доступ к аргументу без его копирования, гарантируя, что функция не изменит значение, на которое ссылается ссылка:

```
1. // ref - это константная ссылка на переданный аргумент, а не копия аргумента
2. void changeN(const int &ref)
3. {
4.     ref = 8; // нельзя: ref - это константа
5. }
```

Ссылки на константные значения особенно полезны в качестве параметров функции из-за их универсальности. Константная ссылка в качестве параметра позволяет передавать неконстантный аргумент l-value, константный аргумент l-value, литерал или результат выражения:

```
1. #include <iostream>
2.
3. void printIt(const int &a)
4. {
5.     std::cout << a;
6. }
7.
8. int main()
9. {
10.     int x = 3;
11.     printIt(x); // неконстантное l-value
12.
13.     const int y = 4;
14.     printIt(y); // константное l-value
15.
```

```
16.     printIt(5); // литерал в качестве r-value
17.
18.     printIt(3+y); // выражение в качестве r-value
19.
20.     return 0;
21. }
```

Результат выполнения программы:

```
3457
```

Во избежание ненужного, слишком затратного копирования аргументов, переменные, которые не являются фундаментальных типов данных (типов int, double и т.д.) или указателями, — должны передаваться по (константной) ссылке в функцию. Фундаментальные типы данных должны передаваться по значению в случае, если функция не будет изменять их значений.

Правило: Переменные не фундаментальных типов данных и которые не являются указателями, передавайте в функцию по (константной) ссылке.

Урок №94. Оператор доступа к членам через указатель

Обычно есть либо указатель, либо ссылка на структуру/класс. Как мы уже знаем из предыдущих уроков, доступ к члену структуры осуществляется через **оператор выбора члена** (`.`) (или ***«оператор доступа к члену»***):

```
1. struct Man
2. {
3.     int weight;
4.     double height;
5. };
6. Man man;
7.
8. // Доступ к члену осуществляется через использование фактической переменной структуры Man
9. man.weight = 60;
```

Этот синтаксис также работает и со ссылками:

```
1. struct Man
2. {
3.     int weight;
4.     double height;
5. };
6. Man man; // определяем переменную структуры Man
7.
8. // Доступ к члену осуществляется через ссылку на переменную структуры Man
9. Man &ref = man;
10. ref.weight = 60;
```

Однако, в случае с указателем, вам нужно его сначала разыменовать:

```
1. struct Man
2. {
3.     int weight;
4.     double height;
5. };
6. Man man;
7.
8. // Доступ к члену осуществляется через указатель на переменную структуры Man
9. Man *ptr = &man;
10. (*ptr).weight = 60;
```

Обратите внимание, разыменование указателя должно находиться в круглых скобках, поскольку оператор выбора члена имеет более высокий приоритет, чем оператор разыменования.

Поскольку синтаксис доступа к членам структур/классов с помощью указателя несколько неудобен, то C++ предоставляет второй **оператор выбора членов** (`->`) для осуществления доступа к членам через указатель.

Следующие две строки идентичны:

```
1. (*ptr).weight = 60;
2. ptr->weight = 60;
```

Это не только легче писать, но и этот способ так же менее подвержен ошибкам, поскольку здесь разыменование выполняется неявно, поэтому нет проблем с приоритетом, о котором нужно помнить. Следовательно, при доступе к членам структур или классов через указатель, всегда используйте оператор `->` вместо оператора `.`.

Правило: При использовании указателя для доступа к значению члена структуры или класса используйте оператор «->» вместо оператора «.».

Урок №95. Цикл foreach

На предыдущих уроках мы рассматривали примеры использования цикла for для осуществления итерации по каждому элементу массива. Например:

```
#include <iostream>

int main()
{
    const int numStudents = 7;
    int scores[numStudents] = { 45, 87, 55, 68, 80, 90, 58 };
    int maxScore = 0; // отслеживаем наивысший балл
    for (int student = 0; student < numStudents; ++student)
        if (scores[student] > maxScore)
            maxScore = scores[student];

    std::cout << "The best score was " << maxScore << '\n';

    return 0;
}
```

В то время как циклы for предоставляют удобный и гибкий способ итерации по массиву, в них так же легко можно запутаться и наделать «ошибок неучтенных единиц».

Поэтому в C++11 добавили новый тип цикла - **foreach** (или *«цикл, основанный на диапазоне»*), который предоставляет более простой и безопасный способ итерации по массиву (или по любой другой структуре типа списка).

Синтаксис цикла foreach следующий:

```
for (объявление_элемента : массив)
   стейтмент;
```

Выполняется итерация по каждому элементу массива, присваивая значение текущего элемента массива переменной, объявленной как **элемент** (`объявление_элемента`). В целях улучшения производительности объявляемый элемент должен быть того же типа, что и элементы массива, иначе произойдет неявное преобразование. Рассмотрим простой пример использования цикла foreach для вывода всех элементов массива `math`:

```
#include <iostream>

int main()
{
    int math[] = { 0, 1, 4, 5, 7, 8, 10, 12, 15, 17, 30, 41};
    for (int number : math) // итерация по массиву math
       std::cout << number << ' '; // получаем доступ к элементу массива в этой итерации через переменную number
```

```
8.
9.      return 0;
10. }
```

Результат выполнения программы:

```
0 1 4 5 7 8 10 12 15 17 30 41
```

Рассмотрим детально, как это всё работает. При выполнении цикла foreach переменной `number` присваивается значение первого элемента (т.е. значение `0`). Дальше программа выполняет стейтмент вывода значения переменной `number`, т.е. нуля. Затем цикл выполняется снова, и значением переменной `number` уже является `1` (второй элемент массива). Вывод значения `number` выполняется снова. Цикл продолжает свое выполнение до тех пор, пока в массиве не останется непройденных элементов. В конце выполнения программа возвращает `0` обратно в операционную систему с помощью оператора return.

Обратите внимание, переменная `number` не является индексом массива. Ей просто присваивается значение элемента массива в текущей итерации цикла.

Цикл foreach и ключевое слово auto

Поскольку объявляемый элемент цикла foreach должен быть того же типа, что и элементы массива, то это идеальный случай для использования ключевого слова auto, когда мы позволяем C++ вычислить тип данных элементов массива вместо нас. Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int math[] = { 0, 1, 4, 5, 7, 8, 10, 12, 15, 17, 30, 41 };
6.     for (auto number : math) // тип number определяется автоматически исходя
   из типа элементов массива math
7.         std::cout << number << ' ';
8.
9.     return 0;
10. }
```

Цикл foreach и ссылки

В примерах, приведенных выше, объявляемый элемент всегда является переменной:

```
1. int array[7] = { 10, 8, 6, 5, 4, 3, 1 };
2. for (auto element: array) // element будет копией текущего элемента массива
3.     std::cout << element << ' ';
```

То есть каждый обработанный элемент массива копируется в переменную `element`. А это копирование может оказаться затратным, в большинстве случаев мы можем просто ссылаться на исходный элемент с помощью ссылки:

```
1. int array[7] = { 10, 8, 6, 5, 4, 3, 1 };
2. for (auto &element: array) // символ амперсанда делает element ссылкой на текущий элемент массива, предотвращая копирование
3.     std::cout << element << ' ';
```

В примере, приведенном выше, в качестве объявляемого элемента цикла foreach используется ссылка на текущий элемент массива, при этом копирования этого элемента не происходит. Но, с указанием обычной ссылки, любые изменения элемента будут влиять на сам массив, что не всегда может быть желательно.

Конечно же, хорошей идеей будет сделать объявляемый элемент константным, тогда вы сможете его использовать в режиме «*только для чтения*»:

```
1. int array[7] = { 10, 8, 6, 5, 4, 3, 1 };
2. for (const auto &element: array) // element - это константная ссылка на текущий элемент массива в итерации
3.     std::cout << element << ' ';
```

Правило: Используйте обычные ссылки или константные ссылки в качестве объявляемого элемента в цикле foreach (в целях улучшения производительности).

Еще один пример

Вот пример первой программы из начала этого урока, но уже с использованием цикла foreach:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     const int numStudents = 7;
6.     int scores[numStudents] = { 45, 87, 55, 68, 80, 90, 58};
7.     int maxScore = 0; // отслеживаем индекс наибольшего score (значения)
8.     for (const auto &score: scores) // итерация по массиву, присваиваем каждое значение массива поочередно переменной score
9.         if (score > maxScore)
10.            maxScore = score;
11.
12.    std::cout << "The best score was " << maxScore << '\n';
13.
14.    return 0;
15. }
```

Обратите внимание, здесь нам уже не нужно вручную прописывать индексацию массива. Мы можем получить доступ к каждому элементу массива непосредственно через переменную `score`.

Цикл foreach и не массивы

Циклы foreach работают не только с фиксированными массивами, но также и со многими другими структурами типа списка такими, как векторы (например, std::vector), связанные списки, деревья. Не беспокойтесь, если вы не знаете, что это такое (мы всё это рассмотрим чуть позже). Просто помните, что циклы foreach обеспечивают гибкий и удобный способ итерации не только по массивам:

```
1. #include <vector>
2. #include <iostream>
3.
4. int main()
5. {
6.     std::vector<int> math = { 0, 1, 4, 5, 7, 8, 10, 12, 15, 17, 30, 41}; // обратите внимание здесь на использование std::vector вместо фиксированного массива
7.     for (const auto &number : math)
8.         std::cout << number << ' ';
9.
10.    return 0;
11. }
```

Цикл foreach не работает с указателями на массив

Для итерации по массиву, цикл foreach должен знать длину массива. Поскольку массивы, которые распадаются в указатель, не знают своей длины, то циклы foreach с ними работать не могут!

```
1. #include <iostream>
2.
3. int sumArray(int array[]) // array - это указатель
4. {
5.     int sum = 0;
6.     for (const auto &number : array) // ошибка компиляции, размер массива неизвестен
7.         sum += number;
8.
9.     return sum;
10. }
11.
12. int main()
13. {
14.     int array[7] = { 10, 8, 6, 5, 4, 3, 1 };
15.     std::cout << sumArray(array); // array распадается в указатель здесь
16.     return 0;
17. }
```

По этой же причине циклы foreach не работают с динамическими массивами.

Могу ли я получить индекс текущего элемента?

Циклы foreach не предоставляют прямой способ получения индекса текущего элемента массива. Это связано с тем, что большинство структур, с которыми могут использоваться циклы foreach (например, связанные списки), напрямую не индексируются!

Заключение

Циклы foreach обеспечивают лучший синтаксис для итерации по массиву, когда нам нужно получить доступ ко всем элементам массива в последовательном порядке. Эти циклы предпочтительнее использовать вместо стандартных циклов for в случаях, когда они могут использоваться. Для предотвращения создания копий каждого элемента в качестве объявляемого элемента следует использовать ссылку.

Тест

Это должно быть легко!

Объявите фиксированный массив со следующими именами: `Sasha`, `Ivan`, `John`, `Orlando`, `Leonardo`, `Nina`, `Anton` и `Molly`. Попросите пользователя ввести имя. Используйте цикл foreach для проверки того, не находится ли имя, введенное пользователем, уже в массиве.

Пример результата выполнения программы:

```
Enter a name: Sasha
Sasha was found.

Enter a name: Maruna
Maruna was not found.
```

> ***Подсказка:*** Используйте std::string в качестве типа массива.

Урок №96. Указатели типа void

Указатель типа void (или ***«общий указатель»***) - это специальный тип указателя, который может указывать на объекты любого типа данных! Объявляется он как обычный указатель, только вместо типа данных используется **ключевое слово void**:

```
1. void *ptr; // ptr - это указатель типа void
```

Указатель типа void может указывать на объекты любого типа данных:

```
1. int nResult;
2. float fResult;
3.
4. struct Something
5. {
6.     int n;
7.     float f;
8. };
9.
10. Something sResult;
11.
12. void *ptr;
13. ptr = &nResult; // допустимо
14. ptr = &fResult; // допустимо
15. ptr = &sResult; // допустимо
```

Однако, поскольку указатель типа void сам не знает, на объект какого типа он будет указывать, разыменовать его напрямую не получится! Вам сначала нужно будет явно преобразовать указатель типа void с помощью оператора static_cast в другой тип данных, а затем уже его разыменовать:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int value = 7;
6.     void *voidPtr = &value;
7.
8.     //std::cout << *voidPtr << std::endl; // запрещено: нельзя разыменовать
   указатель типа void
9.
10.    int *intPtr = static_cast<int*>(voidPtr); // однако, если мы конвертируем
   наш указатель типа void в указатель типа int,
11.    std::cout << *intPtr << std::endl; // то мы сможем его разыменовать,
   будто бы это обычный указатель
12.
13.    return 0;
14. }
```

Результат выполнения программы:

```
7
```

Возникает вопрос: «Если указатель типа void сам не знает, на что он указывает, то как мы тогда можем знать, в какой тип данных его следует явно конвертировать с помощью оператора static_cast?». Никак, это уже на ваше усмотрение, вам самим придется выбрать нужный тип. Например:

```
#include <iostream>

enum Type
{
    INT,
    DOUBLE,
    CSTRING
};

void printValue(void *ptr, Type type)
{
    switch (type)
    {
        case INT:
            std::cout << *static_cast<int*>(ptr) << '\n'; // конвертируем в указатель типа int и выполняем разыменование
            break;
        case DOUBLE:
            std::cout << *static_cast<double*>(ptr) << '\n'; // конвертируем в указатель типа double и выполняем разыменование
            break;
        case CSTRING:
            std::cout << static_cast<char*>(ptr) << '\n'; // конвертируем в указатель типа char (без разыменования)
            // std::cout знает, что char* следует обрабатывать как строку C-style.
            // Если бы мы разыменовали результат (целое выражение), то тогда бы вывелся просто первый символ из массива букв, на который указывает ptr
            break;
    }
}

int main()
{
    int nValue = 7;
    double dValue = 9.3;
    char szValue[] = "Jackie";

    printValue(&nValue, INT);
    printValue(&dValue, DOUBLE);
    printValue(szValue, CSTRING);

    return 0;
}
```

Результат выполнения программы:

```
7
9.3
Jackie
```

Указателям типа void можно присвоить нулевое значение:

```
1. void *ptr = 0; // ptr - это указатель типа void, который сейчас является
   нулевым
```

Хотя некоторые компиляторы позволяют удалять указатели типа void, которые указывают на динамически выделенную память, делать это не рекомендуется, так как результаты могут быть неожиданными.

Также не получится выполнить адресную арифметику с указателями типа void, так как для этого требуется, чтобы указатель знал размер объекта, на который он указывает (для выполнения корректного инкремента/декремента). Также нет такого понятия, как ссылка на void.

Заключение

В общем, использовать указатели типа void рекомендуется только в самых крайних случаях, когда без этого не обойтись, так как с их использованием проверку типов данных ни вам, ни компилятору выполнить не удастся. А это, в свою очередь, позволит вам случайно сделать то, что не имеет смысла, и компилятор на это жаловаться не будет. Например:

```
1. int nResult = 7;
2. printResult(&nResult, CSTRING);
```

Здесь компилятор промолчит. Но что будет в результате? Непонятно!

Хотя код, приведенный выше, кажется аккуратным способом заставить одну функцию обрабатывать несколько типов данных, в языке C++ есть гораздо лучший способ сделать то же самое (через перегрузку функций), в котором сохраняется проверка типов для предотвращения неправильного использования. Также для обработки нескольких типов данных можно использовать шаблоны, которые обеспечивают хорошую проверку типов (но об этом уже на следующих уроках).

Если вам все же придется использовать указатель типа void, то убедитесь, что нет лучшего (более безопасного) способа сделать то же самое, но с использованием других механизмов языка C++!

Тест

В чём разница между нулевым указателем и указателем типа void?

Урок №97. Указатели на указатели

Указатель на указатель — это именно то, что вы подумали: указатель, который содержит адрес другого указателя.

Указатели на указатели

Обычный указатель типа int объявляется с использованием одной звёздочки:

```
int *ptr; // указатель типа int, одна звёздочка
```

Указатель на указатель типа int объявляется с использованием 2-х звёздочек:

```
int **ptrptr; // указатель на указатель типа int (две звёздочки)
```

Указатель на указатель работает подобно обычному указателю: вы можете его разыменовать для получения значения, на которое он указывает. И, поскольку этим значением является другой указатель, для получения исходного значения вам потребуется выполнить разыменование еще раз. Их следует выполнять последовательно:

```
#include <iostream>

int main()
{
    int value = 7;

    int *ptr = &value;
    std::cout << *ptr << std::endl; // разыменовываем указатель, чтобы получить значение типа int

    int **ptrptr = &ptr;
    std::cout << **ptrptr << std::endl;

    return 0;
}
```

Результат выполнения программы:

```
7
7
```

Обратите внимание, вы не можете инициализировать указатель на указатель напрямую значением:

```
int value = 7;
int **ptrptr = &&value; // нельзя
```

Это связано с тем, что оператор адреса (`&`) требует l-value, но `&value` — это r-value. Однако указателю на указатель можно задать значение null:

```
1. int **ptrptr = nullptr; // используйте 0, если не поддерживается C++11
```

Массивы указателей

Указатели на указатели имеют несколько применений. Наиболее используемым является динамическое выделение массива указателей:

```
1. int **array = new int*[20]; // выделяем массив из 20 указателей типа int
```

Это тот же обычный динамически выделенный массив, за исключением того, что элементами являются указатели на тип int, а не значения типа int.

Двумерные динамически выделенные массивы

Другим распространенным применением указателей на указатели является динамическое выделение многомерных массивов. В отличие от двумерного фиксированного массива, который можно легко объявить следующим образом:

```
1. int array[15][7];
```

Динамическое выделение двумерного массива немного отличается. У вас может возникнуть соблазн написать что-то вроде следующего:

```
1. int **array = new int[15][7]; // не будет работать!
```

Здесь вы получите ошибку. Есть два возможных решения. Если правый индекс является константой типа compile-time, то вы можете сделать следующее:

```
1. int (*array)[7] = new int[15][7];
```

Скобки здесь потребуются для соблюдения приоритета. В C++11 хорошей идеей будет использовать ключевое слово auto для автоматического определения типа данных:

```
1. auto array = new int[15][7]; // намного проще!
```

К сожалению, это относительно простое решение не работает, если правый индекс не является константой типа compile-time. В таком случае всё немного усложняется. Сначала мы выделяем массив указателей (как в примере, приведенном выше), а затем перебираем каждый элемент массива указателей и выделяем динамический массив для каждого элемента этого массива. Итого, наш динамический двумерный

массив - это динамический одномерный массив динамических одномерных массивов!

```
1. int **array = new int*[15]; // выделяем массив из 15 указателей типа int — это наши строки
2. for (int count = 0; count < 15; ++count)
3.     array[count] = new int[7]; // а это наши столбцы
```

Доступ к элементам массива выполняется как обычно:

```
1. array[8][3] = 4; // это то же самое, что и (array[8])[3] = 4;
```

Этим методом, поскольку каждый столбец массива динамически выделяется независимо, можно сделать динамически выделенные двумерные массивы, которые не являются прямоугольными. Например, мы можем создать массив треугольной формы:

```
1. int **array = new int*[15]; // выделяем массив из 15 указателей типа int — это наши строки
2. for (int count = 0; count < 15; ++count)
3.     array[count] = new int[count+1]; // а это наши столбцы
```

В примере, приведенном выше, `array[0]` - это массив длиной 1, а `array[1]` - массив длиной 2 и т.д.

Для освобождения памяти динамически выделенного двумерного массива (который создавался с помощью этого способа) также потребуется цикл:

```
1. for (int count = 0; count < 15; ++count)
2.     delete[] array[count];
3. delete[] array; // это следует выполнять в конце
```

Обратите внимание, мы удаляем массив в порядке, противоположном его созданию. Если мы удалим массив перед удалением элементов массива, то нам придется получать доступ к освобожденной памяти для удаления элементов массива. А это, в свою очередь, приведет к неожиданным результатам.

Поскольку процесс выделения и освобождения двумерных массивов является несколько запутанным (можно легко наделать ошибок), то часто проще «сплющить» двумерный массив в одномерный массив:

```
1. // Вместо следующего:
2. int **array = new int*[15]; // выделяем массив из 15 указателей типа int — это наши строки
3. for (int count = 0; count < 15; ++count)
4.     array[count] = new int[7]; // а это наши столбцы
5.
6. // Делаем следующее:
7. int *array = new int[105]; // двумерный массив 15x7 "сплющенный" в одномерный массив
```

Простая математика используется для конвертации индексов строки и столбца прямоугольного двумерного массива в один индекс одномерного массива:

```
1. int getSingleIndex(int row, int col, int numberOfColumnsInArray)
2. {
3.     return (row * numberOfColumnsInArray) + col;
4. }
5.
6. // Присваиваем array[9,4] значение 3, используя наш "сплющенный" массив
7. array[getSingleIndex(9, 4, 5)] = 3;
```

Указатель на указатель на указатель на указатель и т.д.

Также можно объявить указатель на указатель на указатель:

```
1. int ***ptrx3;
```

Они могут использоваться для динамического выделения трехмерного массива. Тем не менее, для этого потребуется цикл внутри цикла и чрезвычайная аккуратность и осторожность, чтобы не наделать ошибок. Вы даже можете объявить указатель на указатель на указатель на указатель:

```
1. int ****ptrx4;
```

Или сделать еще большую вложенность, если захотите. Однако на практике такие указатели редко используются.

Заключение

Рекомендуется применять указатели на указатели только в самых крайних случаях, так как они сложны в использовании и потенциально опасны. Достаточно легко разыменовать нулевой или "висячий" указатель в ситуациях с использованием обычных указателей, вдвое легче это сделать в ситуациях с указателем на указатель, поскольку для получения исходного значения потребуется выполнить двойное разыменование!

Урок №98. Введение в std::array

На предыдущих уроках мы подробно говорили о фиксированных и динамических массивах. Хотя они очень полезны и активно используются в языке C++, у них также есть свои недостатки: фиксированные массивы распадаются в указатели, теряя информацию о своей длине; в динамических массивах проблемы могут возникнуть с освобождением памяти и с попытками изменить их длину после выделения.

Поэтому в Стандартную библиотеку C++ добавили функционал, который упрощает процесс управления массивами: std::array и std::vector. На этом уроке мы рассмотрим std::array, а на следующем — std::vector.

Введение в std::array

Представленный в C++11, **std::array** - это фиксированный массив, который не распадается в указатель при передаче в функцию. std::array определяется в заголовочном файле array, внутри пространства имен std. Объявление переменной std::array следующее:

```
1. #include <array>
2.
3. std::array<int, 4> myarray; // объявляем массив типа int длиной 4
```

Подобно обычным фиксированным массивам, длина std::array должна быть установлена во время компиляции. std::array можно инициализировать с использованием списка инициализаторов или uniform-инициализации:

```
1. std::array<int, 4> myarray = { 8, 6, 4, 1 }; // список инициализаторов
2. std::array<int, 4> myarray2 { 8, 6, 4, 1 }; // uniform-инициализация
```

В отличие от стандартных фиксированных массивов, в std::array вы не можете пропустить (не указывать) длину массива:

```
1. std::array<int, > myarray = { 8, 6, 4, 1 }; // нельзя, должна быть указана
   длина массива
```

Также можно присваивать значения массиву с помощью списка инициализаторов:

```
1. std::array<int, 4> myarray;
2. myarray = { 0, 1, 2, 3 }; // ок
3. myarray = { 8, 6 }; // ок, элементам 2 и 3 присвоен нуль!
4. myarray = { 0, 1, 3, 5, 7, 9 }; // нельзя, слишком много элементов в списке
   инициализаторов!
```

Доступ к значениям массива через оператор индекса осуществляется как обычно:

```
std::cout << myarray[1];
myarray[2] = 7;
```

Так же, как и в стандартных фиксированных массивах, оператор индекса не выполняет никаких проверок на диапазон. Если указан недопустимый индекс, то произойдут плохие вещи.

std::array поддерживает вторую форму доступа к элементам массива — **функция at()**, которая осуществляет проверку диапазона:

```
std::array<int, 4> myarray { 8, 6, 4, 1 };
myarray.at(1) = 7; // элемент массива под номером 1 - корректный, присваиваем ему значение 7
myarray.at(8) = 15; // элемент массива под номером 8 - некорректный, получим ошибку
```

В примере, приведенном выше, вызов `myarray.at(1)` проверяет, есть ли элемент массива под номером 1, и, поскольку он есть, возвращается ссылка на этот элемент. Затем мы присваиваем ему значение `7`. Однако, вызов `myarray.at(8)` не срабатывает, так как элемента под номером 8 в массиве нет. Вместо возвращения ссылки, функция at() выдает ошибку, которая завершает работу программы (на самом деле выбрасывается исключение типа `std::out_of_range`. Об исключениях мы поговорим на соответствующих уроках). Поскольку проверка диапазона выполняется, то функция at() работает медленнее (но безопаснее), чем оператор `[]`.

std::array автоматически делает все очистки после себя, когда выходит из области видимости, поэтому нет необходимости прописывать это вручную.

Размер и сортировка

С помощью **функции size()** можно узнать длину массива:

```
#include <iostream>
#include <array>

int main()
{
    std::array<double, 4> myarray{ 8.0, 6.4, 4.3, 1.9 };
    std::cout << "length: " << myarray.size();

    return 0;
}
```

Результат:

```
length: 4
```

Поскольку std::array не распадается в указатель при передаче в функцию, то функция size() будет работать, даже если её вызвать из другой функции:

```
#include <iostream>
#include <array>

void printLength(const std::array<double, 4> &myarray)
{
    std::cout << "length: " << myarray.size();
}

int main()
{
    std::array<double, 4> myarray { 8.0, 6.4, 4.3, 1.9 };

    printLength(myarray);

    return 0;
}
```

Результат тот же:

```
length: 4
```

Обратите внимание, Стандартная библиотека C++ использует термин «размер» для обозначения длины массива - не путайте это с результатами выполнения оператора sizeof с обычным фиксированным массивом, когда возвращается фактический размер массива в памяти (`размер элемент * длина массива`).

Также обратите внимание на то, что мы передаем std::array по ссылке (константной). Это делается по соображениям производительности для того, чтобы компилятор не выполнял копирование массива при передаче в функцию.

Правило: Всегда передавайте std::array в функции по обычной или по константной ссылке.

Поскольку длина массива всегда известна, то циклы foreach также можно использовать с std::array:

```
std::array<int, 4> myarray { 8, 6, 4, 1 };

for (auto &element : myarray)
    std::cout << element << ' ';
```

Вы можете отсортировать std::array, используя **функцию std::sort()**, которая находится в заголовочном файле algorithm:

```
#include <iostream>
#include <array>
#include <algorithm> // для std::sort

int main()
{
    std::array<int, 5> myarray { 8, 4, 2, 7, 1 };
    std::sort(myarray.begin(), myarray.end()); // сортировка массива по возрастанию
//    std::sort(myarray.rbegin(), myarray.rend()); // сортировка массива по убыванию

    for (const auto &element : myarray)
        std::cout << element << ' ';

    return 0;
}
```

Результат:

```
1 2 4 7 8
```

Функция сортировки использует итераторы, которые мы еще не рассматривали. О них мы поговорим несколько позже.

Заключение

std::array — это отличная замена стандартных фиксированных массивов. Массивы, созданные с помощью std::array, более эффективны, так как используют меньше памяти. Единственными недостатками std::array по сравнению со стандартными фиксированными массивами являются немного неудобный синтаксис и то, что нужно явно указывать длину массива (компилятор не будет вычислять её за нас). Но это сравнительно незначительные нюансы. Рекомендуется использовать std::array вместо стандартных фиксированных массивов в любых нетривиальных задачах.

Урок №99. Введение в std::vector

На предыдущем уроке мы рассматривали std::array, который является более безопасной и удобной формой обычных фиксированных массивов в языке C++. Аналогично, в Стандартной библиотеке C++ есть и улучшенная версия динамических массивов (более безопасная и удобная) - std::vector.

В отличие от std::array, который недалеко отходит от базового функционала обычных фиксированных массивов, std::vector идет в комплекте с дополнительными возможностями, которые делают его одним из самых полезных и универсальных инструментов в языке C++.

Векторы

Представленный в C++03, **std::vector** (или просто ***"вектор"***) — это тот же динамический массив, но который может сам управлять выделенной себе памятью. Это означает, что вы можете создавать массивы, длина которых задается во время выполнения, без использования операторов new и delete (явного указания выделения и освобождения памяти). std::vector находится в заголовочном файле vector. Объявление std::vector следующее:

```
1. #include <vector>
2.
3. // Нет необходимости указывать длину при инициализации
4. std::vector<int> array;
5. std::vector<int> array2 = { 10, 8, 6, 4, 2, 1 }; // используется список
   инициализаторов для инициализации массива
6. std::vector<int> array3 { 10, 8, 6, 4, 2, 1 }; // используется uniform-
   инициализация для инициализации массива (начиная с C++11)
```

Обратите внимание, что в неинициализированном, что в инициализированном случаях вам не нужно явно указывать длину массивов. Это связано с тем, что std::vector динамически выделяет память для своего содержимого по запросу.

Подобно std::array, доступ к элементам массива может выполняться как через оператор `[]` (который не выполняет проверку диапазона), так и через функцию at() (которая выполняет проверку диапазона):

```
1. array[7] = 3; // без проверки диапазона
2. array.at(8) = 4; // с проверкой диапазона
```

В любом случае, если вы будете запрашивать элемент, который находится вне диапазона `array`, длина вектора автоматически изменяться не будет. Начиная с

C++11, вы также можете присваивать значения для std::vector, используя список инициализаторов:

```
array = { 0, 2, 4, 5, 7 }; // ок, длина array теперь 5
array = { 11, 9, 5 }; // ок, длина array теперь 3
```

В таком случае вектор будет самостоятельно изменять свою длину, чтобы соответствовать количеству предоставленных элементов.

Нет утечкам памяти!

Когда переменная-вектор выходит из области видимости, то она автоматически освобождает память, которую контролировала (занимала). Это не только удобно (так как вам не нужно это делать вручную), но также помогает предотвратить утечки памяти. Рассмотрим следующий фрагмент:

```
void doSomething(bool value)
{
    int *array = new int[7] { 12, 10, 8, 6, 4, 2, 1 };

    if (value)
        return;

   // Делаем что-нибудь

    delete[] array; // если value == true, то этот стейтмент никогда не выполнится
}
```

Если переменной `value` присвоить значение true, то `array` никогда не будет удален, память никогда не будет освобождена и произойдет утечка памяти.

Однако, если бы `array` был вектором, то подобное никогда бы и не произошло, так как память освобождалась бы автоматически при выходе `array` из области видимости (независимо от того, выйдет ли функция раньше из области видимости или нет). Именно из-за этого использование std::vector является более безопасным, чем динамическое выделение памяти через оператор new.

Длина векторов

В отличие от стандартных динамических массивов, которые не знают свою длину, std::vector свою длину запоминает. Чтобы её узнать, нужно использовать **функцию size()**:

```
#include <vector>
#include <iostream>

int main()
```

```
{
    std::vector<int> array { 12, 10, 8, 6, 4, 2, 1 };
    std::cout << "The length is: " << array.size() << '\n';

    return 0;
}
```

Результат:

```
The length is: 7
```

Изменить длину стандартного динамически выделенного массива довольно проблематично и сложно. Изменить длину std::vector так же просто, как вызвать **функцию resize()**:

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> array { 0, 1, 2 };
    array.resize(7); // изменяем длину array на 7

    std::cout << "The length is: " << array.size() << '\n';

    for (auto const &element: array)
        std::cout << element << ' ';

    return 0;
}
```

Результат:

```
The length is: 7
0 1 2 0 0 0 0
```

Здесь есть две вещи, на которые следует обратить внимание. Во-первых, когда мы изменили длину `array`, существующие значения элементов сохранились! Во-вторых, новые элементы были инициализированы значением по умолчанию в соответствие с определенным типом данных (значением `0` для типа int).

Длину вектора также можно изменить и в обратную сторону (обрезать):

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> array { 0, 1, 4, 7, 9, 11 };
    array.resize(4); // изменяем длину array на 4

    std::cout << "The length is: " << array.size() << '\n';

```

```
    for (auto const &element: array)
        std::cout << element << ' ';

    return 0;
}
```

Результат:

```
The length is: 4
0 1 4 7
```

Изменение длины вектора является затратной операцией, поэтому вы должны стремиться минимизировать количество подобных выполняемых операций.

Заключение

Это вводная статья, предназначенная для ознакомления с основами std::vector. На следующих уроках мы детально рассмотрим std::vector, в том числе и разницу между длиной и ёмкостью вектора, и то, как в std::vector выполняется выделение памяти.

Поскольку переменные типа std::vector могут сами управлять выделенной себе памятью (что помогает предотвратить утечку памяти), отслеживают свою длину и легко её изменяют, то рекомендуется использовать std::vector вместо стандартных динамических массивов.

Урок №100. Введение в итераторы

Итерация/перемещение по элементам массива (или какой-нибудь другой структуры) является довольно распространенным действием в программировании. Мы уже рассматривали множество различных способов выполнения данной задачи, а именно: с использованием циклов и индексов (циклы for и while), с помощью указателей и арифметики указателей, а также с помощью циклов for с явным указанием диапазона:

```
1. #include <array>
2. #include <iostream>
3.
4. int main()
5. {
6.   // Автоматическое определение типа как std::array<int, 7> (требуется поддержка стандарта C++17).
7.   std::array data{ 0, 1, 2, 3, 4, 5, 6 };  // используйте std::array<int, 7>, если ваш компилятор не поддерживает C++17
8.   std::size_t length{ std::size(data) };
9.
10.  // Цикл while с использованием явного индекса
11.  std::size_t index{ 0 };
12.  while (index != length)
13.  {
14.    std::cout << data[index] << ' ';
15.    ++index;
16.  }
17.  std::cout << '\n';
18.
19.  // Цикл for с использованием явного индекса
20.  for (index = 0; index < length; ++index)
21.  {
22.    std::cout << data[index] << ' ';
23.  }
24.  std::cout << '\n';
25.
26.  // Цикл for с использованием указателей (обратите внимание, ptr не может быть const, так как позже мы выполняем с ним операцию инкремента)
27.  for (auto ptr{ &data[0] }; ptr != (&data[0] + length); ++ptr)
28.  {
29.    std::cout << *ptr << ' ';
30.  }
31.  std::cout << '\n';
32.
33.  // Цикл for с явным указанием диапазона
34.  for (int i : data)
35.  {
36.    std::cout << i << ' ';
37.  }
38.  std::cout << '\n';
39.
40.  return 0;
41. }
```

Использование циклов с индексами в ситуациях, когда мы используем их только для доступа к элементам, требует написания большего количества кода, нежели могло бы быть.

При этом данный способ работает только в том случае, если контейнер (например, массив), содержащий данные, дает возможность прямого доступа к своим элементам (что делают массивы, но не делают некоторые другие типы контейнеров, например, списки).

Использование циклов с указателями и арифметикой указателей требует довольно большого объёма теоретических знаний и может сбить с толку читателей, которые не знакомы с арифметикой указателей и не знают её правил. Да и сама арифметика указателей применима лишь том случае, если элементы структуры данных расположены в памяти последовательно (что опять же верно для массивов, но не всегда выполняется для других типов данных, таких как списки, деревья, карты).

> ***Примечание для продвинутых читателей*:** Указатели (без арифметики указателей) могут использоваться для перебора/итерации некоторых структур данных с непоследовательным расположением элементов. Например, в связном списке каждый элемент соединен указателем с предыдущим элементом, поэтому мы можем перебирать список, следуя по цепочке указателей.

Циклы for с явным указанием диапазона чуть более интересны, поскольку у них скрыт механизм перебора нашего контейнера, но при всем этом они все равно могут быть применены к различным структурам данных (массивы, списки, деревья, карты и т.д.). «Как же они работают?» — спросите вы. Они используют итераторы.

Итераторы в C++

Итератор — это объект, разработанный специально для перебора элементов контейнера (например, значений массива или символов в строке), обеспечивающий во время перемещения по элементам доступ к каждому из них.

Контейнер может предоставлять различные типы итераторов. Например, контейнер на основе массива может предлагать прямой итератор, который проходится по массиву в прямом порядке, и реверсивный итератор, который проходится по массиву в обратном порядке.

После того, как итератор соответствующего типа создан, программист может использовать интерфейс, предоставляемый данным итератором, для перемещения по элементам контейнера или доступа к его элементам, не беспокоясь при этом о

том, какой тип перебора элементов задействован или каким образом в контейнере хранятся данные. И, поскольку итераторы в языке C++ обычно используют один и тот же интерфейс как для перемещения по элементам контейнера (оператор `++` для перехода к следующему элементу), так и для доступа (оператор `*` для доступа к текущему элементу) к ним, итерации можно выполнять по разнообразным типам контейнеров, используя последовательный метод.

Указатели в качестве итераторов

Простейший пример итератора — это указатель, который (используя арифметику указателей) работает с последовательно расположенными элементами данных. Давайте снова рассмотрим пример перемещения по элементам массива, используя указатель и арифметику указателей:

```
#include <array>
#include <iostream>

int main()
{
  std::array data{ 0, 1, 2, 3, 4, 5, 6 };

  auto begin{ &data[0] };
  // Обратите внимание, что здесь мы указываем на место, следующее за последним элементом
  auto end{ begin + std::size(data) };

  // Цикл for с использованием указателя
  for (auto ptr{ begin }; ptr != end; ++ptr) // выполняем инкремент для перехода к следующему элементу
  {
    std::cout << *ptr << ' '; // разыменовываем указатель для получения текущего значения элемента
  }
  std::cout << '\n';

  return 0;
}
```

Результат выполнения программы:

```
0 1 2 3 4 5 6
```

В примере, приведенном выше, мы определили две переменные: `begin` (которая указывает на начало нашего контейнера) и `end` (которая указывает на конец нашего контейнера). Для массивов конечным маркером обычно является место в памяти, где мог находиться последний элемент, если бы контейнер содержал на один элемент больше.

Затем указатель перемещается между `begin` и `end`, при этом доступ к текущему элементу можно получить с помощью оператора разыменовывания.

> ***Предупреждение***: У вас может появиться соблазн вычислить конечную точку, используя оператор адреса (`&`) следующим образом:
>
> ```
> int* end{ &array[std::size(array)] };
> ```
>
> Но это приведет к неопределенному поведению, потому что `array[std::size(array)]` обращается к элементу, который находится за пределами массива.
>
> Вместо этого следует использовать:
>
> ```
> int* end{ &array[0] + std::size(array) };
> ```

Итераторы Стандартной библиотеки C++

Выполнение итераций является настолько распространенным действием, что все Стандартные библиотеки контейнеров обеспечивают прямую поддержку итераций. Вместо вычисления начальной и конечной точек вручную, мы можем просто попросить контейнер сделать это за нас, обратившись к функциям `begin()` и `end()`:

```
#include <iostream>
#include <array>

int main()
{
  std::array array{ 1, 2, 3 };

  // Просим наш массив указать нам начальную и конечную точки при помощи функций begin() и end()
  auto begin{ array.begin() };
  auto end{ array.end() };

  for (auto p{ begin }; p != end; ++p) // выполняем инкремент для перехода к следующему элементу
  {
    std::cout << *p << ' '; // разыменовываем указатель для получения текущего значения элемента
  }
  std::cout << '\n';

  return 0;
}
```

Результат выполнения программы:

```
1 2 3
```

В заголовочном файле iterator также содержатся две обобщенные функции (`std::begin()` и `std::end()`):

```
#include <array>
#include <iostream>
#include <iterator> // для std::begin() и std::end()

int main()
{
  std::array array{ 1, 2, 3 };

  // Используем std::begin() и std::end() для получения начальной и конечной точек array
  auto begin{ std::begin(array) };
  auto end{ std::end(array) };

  for (auto p{ begin }; p != end; ++p) // выполняем инкремент для перехода к следующему элементу
  {
    std::cout << *p << ' '; // разыменовываем указатель для получения текущего значения элемента
  }
  std::cout << '\n';

  return 0;
}
```

Результат выполнения аналогичен предыдущему результату:

```
1 2 3
```

Не стоит сейчас беспокоиться о типах итераторов. Сейчас важно понять лишь то, что всю работу по перемещению по контейнеру итератор берет на себя. Нам не обязательно знать детали того, как это происходит. Нам нужно знать следующие 4 вещи:

- начальная точка;
- конечная точка;
- оператор `++` для перемещения итератора к следующему элементу (или к концу);
- оператор `*` для получения значения текущего элемента.

Итераторы и циклы for с явным указанием диапазона

Все типы данных, которые имеют методы begin() и end() или используются с std::begin() и std::end(), могут быть задействованы в циклах for с явным указанием диапазона:

```
#include <array>
#include <iostream>
```

```
3.
4.  int main()
5.  {
6.    std::array array{ 1, 2, 3 };
7.
8.    // Данный цикл работает аналогично циклу, приведенному выше
9.    for (int i : array)
10.   {
11.     std::cout << i << ' ';
12.   }
13.   std::cout << '\n';
14.
15.   return 0;
16. }
```

На самом деле, циклы for с явным указанием диапазона для осуществления итерации незаметно обращаются к вызовам функций begin() и end(). Тип данных std::array также имеет в своем арсенале методы begin() и end(), а значит и его мы можем использовать в циклах for с явным указанием диапазона. Массивы C-style с фиксированным размером также можно использовать с функциями std::begin() и std::end(). Однако с динамическими массивами данный способ не работает, так как для них не существует функции std::end() (из-за того, что отсутствует информация о длине массива).

Позже вы узнаете, как добавлять функционал к вашим типам данных так, чтобы их можно было использовать и с циклами for с явным указанием диапазона.

Циклы for с явным указанием диапазона используются не только при работе с итераторами. Они также могут быть задействованы вместе с std::sort и другими алгоритмами. Теперь, когда вы знаете, что это такое, вы можете заметить, что они довольно часто используются в Стандартной библиотеке C++.

"Висячие" итераторы

Подобно указателям и ссылкам, итераторы также могут стать "висячими", если элементы, по которым выполняется итерация, изменяют свой адрес или уничтожаются. Когда такое происходит, то говорят, что итератор был **недействительным** (или произошла ***"инвалидация итератора"***). Обращение к недействительному итератору порождает ошибку неопределенного поведения.

Некоторые операции, которые изменяют контейнеры (например, добавление элемента в std::vector), могут иметь побочный эффект, приводя к изменению адресов элементов контейнера. Когда такое происходит, текущие итераторы для этих элементов считаются недействительными. Хорошая справочная документация по C++ обязательно должна иметь информацию о том, какие операции с

контейнерами могут привести или приведут к инвалидации итераторов (в качестве примера вот справочная информация по **"инвалидации итераторов" в std::vector**).

Вот пример подобной ситуации:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector v { 1, 2, 3, 4, 5, 6, 7 };

    auto it { v.begin() };

    ++it; // двигаемся ко второму элементу
    std::cout << *it << '\n'; // ок: выводится "2"

    v.erase(it); // удаляем элемент, на который в данный момент указывает итератор

    // erase() инвалидирует итераторы для стираемого элемента (и последующих элементов тоже),
    // поэтому теперь итератор "it" является недействительным

    ++it; // неопределенное поведение
    std::cout << *it << '\n'; // неопределенное поведение

    return 0;
}
```

Урок №101. Алгоритмы в Стандартной библиотеке C++

Новички обычно тратят довольно много времени на написание пользовательских циклов для выполнения относительно простых задач, таких как: сортировка, поиск или подсчет элементов массивов. Эти циклы могут стать проблематичными как с точки зрения того, насколько легко в них можно сделать ошибку, так и с точки зрения общей надежности и удобства использования, т.к. данные циклы могут быть трудны для понимания.

Поскольку поиск, подсчет и сортировка являются очень распространенными операциями в программировании, то в состав Стандартной библиотеки C++ изначально уже включен большой набор функций, которые выполняют данные задачи всего в несколько строчек кода. В дополнение к этому, эти функции уже предварительно протестированные, эффективные и имеют поддержку множества различных типов контейнеров. А некоторые из этих функций поддерживают и **распараллеливание** — возможность выделять несколько потоков ЦП для одной и той же задачи, чтобы выполнить её быстрее.

Функционал, предоставляемый библиотекой алгоритмов, обычно относится к одной из 3-х категорий:

- **Инспекторы** — используются для просмотра (без изменений) данных в контейнере (например, операции поиска или подсчета элементов).
- **Мутаторы** — используются для изменения данных в контейнере (например, операции сортировки или перестановки элементов).
- **Фасилитаторы** — используются для генерации результата на основе значений элементов данных (например, объекты, которые умножают значения, либо объекты, которые определяют, в каком порядке пары элементов должны быть отсортированы).

Данные алгоритмы расположены в библиотеке алгоритмов (заголовочный файл algorithm). На этом уроке мы рассмотрим некоторые из наиболее распространенных алгоритмов.

> ***Примечание*:** Все эти алгоритмы используют итераторы.

Алгоритм std::find() и поиск элемента по значению

Функция `std::find()` выполняет поиск первого вхождения заданного значения в контейнере.

В качестве аргументов `std::find()` принимает 3 параметра:

- итератор для начального элемента в последовательности;
- итератор для конечного элемента в последовательности;
- значение для поиска.

В результате будет возвращен итератор, указывающий на элемент с искомым значением (если он найден) или конец контейнера (если такой элемент не найден). Например:

```
#include <algorithm>
#include <array>
#include <iostream>

int main()
{
  std::array<int, 6> arr{ 13, 90, 99, 5, 40, 80 };

  std::cout << "Enter a value to search for and replace with: ";
  int search{};
  int replace{};
  std::cin >> search >> replace;

  // Проверка пользовательского ввода должна быть здесь

  // std::find() возвращает итератор, указывающий на найденный элемент (или на конец контейнера).
  // Мы сохраним его в переменной, используя автоматический вывод типа итератора
  auto found{ std::find(arr.begin(), arr.end(), search) };

  // Алгоритмы, которые не нашли то, что искали, возвращают итератор, указывающий на конец контейнера.
  // Мы можем получить доступ к этому итератору, используя метод end()
  if (found == arr.end())
  {
    std::cout << "Could not find " << search << '\n';
  }
  else
  {
    // Перезаписываем найденный элемент
    *found = replace;
  }

  for (int i : arr)
  {
    std::cout << i << ' ';
  }

  std::cout << '\n';

  return 0;
}
```

> ***Примечание:*** Для корректной работы всех примеров данного урока ваш компилятор должен поддерживать стандарт C++17. Детально о том, как использовать функционал C++17 вашей IDE, мы говорили на уроке №9.

Пример, в котором элемент найден:

```
Enter a value to search for and replace with: 5 234
13 90 99 234 40 80
```

Пример, в котором элемент не найден:

```
Enter a value to search for and replace with: 0 234
Could not find 0
13 90 99 5 40 80
```

Алгоритм std::find_if() и поиск элемента с условием

Иногда мы хотим увидеть, есть ли в контейнере значение, которое соответствует некоторому условию (например, строка, содержащая заданную подстроку).

В таких случаях функция `std::find_if()` будет идеальным помощником. Она работает аналогично функции `std::find()`, но вместо того, чтобы передавать значение для поиска, мы передаем вызываемый объект, например, указатель на функцию (или лямбду — об этом чуть позже), который проверяет, найдено ли совпадение. Функция `std::find_if()` будет вызывать этот объект для каждого элемента, пока не найдет искомый элемент (или в контейнере больше не останется элементов для проверки).

Вот пример, где мы используем функцию `std::find_if()`, чтобы проверить, содержат ли какие-либо элементы подстроку `"nut"`:

```
1. #include <algorithm>
2. #include <array>
3. #include <iostream>
4. #include <string_view>
5.
6. // Наша функция возвратит true, если элемент найден
7. bool containsNut(std::string_view str)
8. {
9.   // std::string_view::find возвращает std::string_view::npos, если он не
   нашел подстроку.
10.  // В противном случае, он возвращает индекс, где происходит вхождение
   подстроки в строку str
11.  return (str.find("nut") != std::string_view::npos);
12.}
13.
14.int main()
15.{
```

```
16.   std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };
17.
18.   // Сканируем наш массив, чтобы посмотреть, содержат ли какие-либо элементы подстроку "nut"
19.   auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };
20.
21.   if (found == arr.end())
22.   {
23.     std::cout << "No nuts\n";
24.   }
25.   else
26.   {
27.     std::cout << "Found " << *found << '\n';
28.   }
29.
30.   return 0;
31. }
```

Результат выполнения программы:

```
Found walnut
```

Если бы мы решали задачу, приведенную выше, обычным стандартным способом, то нам бы понадобилось, по крайней мере, два цикла (один для циклического перебора массива и один для сравнения подстроки). Функции Стандартной библиотеки C++ позволяют сделать то же самое всего в несколько строчек кода!

Алгоритмы std::count()/std::count_if() и подсчет вхождений элемента

Функции `std::count()` и `std::count_if()` ищут все вхождения элемента или элемент, соответствующий заданным критериям.

В следующем примере мы посчитаем, сколько элементов содержит подстроку `"nut"`:

```
1.  #include <algorithm>
2.  #include <array>
3.  #include <iostream>
4.  #include <string_view>
5.
6.  bool containsNut(std::string_view str)
7.  {
8.    return (str.find("nut") != std::string_view::npos);
9.  }
10.
11. int main()
12. {
13.   std::array<std::string_view, 5> arr{ "apple", "banana", "walnut", "lemon", "peanut" };
14.
15.   auto nuts{ std::count_if(arr.begin(), arr.end(), containsNut) };
16.
17.   std::cout << "Counted " << nuts << " nut(s)\n";
```

```
18.
19.   return 0;
20. }
```

Результат выполнения программы:

```
Counted 2 nut(s)
```

Алгоритм std::sort() и пользовательская сортировка

Ранее мы использовали `std::sort()` для сортировки массива в порядке возрастания, но возможности `std::sort()` этим не ограничиваются. Есть версия `std::sort()`, которая принимает вспомогательную функцию в качестве третьего параметра, что позволяет выполнять сортировку так, как нам это захочется. Данная вспомогательная функция принимает два параметра для сравнения и возвращает `true`, если первый аргумент должен быть упорядочен перед вторым. По умолчанию, `std::sort()` сортирует элементы в порядке возрастания.

Давайте попробуем использовать `std::sort()` для сортировки массива в обратном порядке с помощью вспомогательной пользовательской функции для сравнения `greater()`:

```
1. #include <algorithm>
2. #include <array>
3. #include <iostream>
4.
5. bool greater(int a, int b)
6. {
7.   // Размещаем a перед b, если a больше, чем b
8.   return (a > b);
9. }
10.
11. int main()
12. {
13.   std::array arr{ 13, 90, 99, 5, 40, 80 };
14.
15.   // Передаем greater в качестве аргумента в функцию std::sort()
16.   std::sort(arr.begin(), arr.end(), greater);
17.
18.   for (int i : arr)
19.   {
20.     std::cout << i << ' ';
21.   }
22.
23.   std::cout << '\n';
24.
25.   return 0;
26. }
```

Результат выполнения программы:

```
99 90 80 40 13 5
```

Опять же, вместо того, чтобы самостоятельно писать с нуля свои циклы/функции, мы можем отсортировать наш массив так, как нам нравится, с использованием всего нескольких строчек кода!

> ***Совет:*** Поскольку сортировка в порядке убывания также очень распространена, то C++ предоставляет пользовательский тип `std::greater{}` для этой задачи (который находится в заголовочном файле functional). В примере, приведенном выше, мы можем заменить:
>
> ```
> std::sort(arr.begin(), arr.end(), greater); // вызов нашей функции greater
> ```
>
> на
>
> ```
> std::sort(arr.begin(), arr.end(), std::greater{}); // используем greater из Стандартной библиотеки C++
> ```
>
> Обратите внимание, что `std::greater{}` нуждается в фигурных скобках, потому что это не вызываемая функция, а тип данных, и для его использования нам нужно создать экземпляр данного типа. Фигурные скобки создают анонимный объект данного типа (который затем передается в качестве аргумента в функцию `std::sort()`).

Алгоритм std::for_each() и все элементы контейнера

Функция `std::for_each()` принимает список в качестве входных данных и применяет пользовательскую функцию к каждому элементу этого списка. Это полезно, когда нам нужно выполнить одну и ту же операцию со всеми элементами списка.

Вот пример, где мы используем `std::for_each()` для удвоения всех чисел в массиве:

```
1. #include <algorithm>
2. #include <array>
3. #include <iostream>
4.
5. void doubleNumber(int &i)
6. {
7.   i *= 2;
8. }
9.
10.int main()
11.{
12.  std::array arr{ 1, 2, 3, 4 };
13.
14.  std::for_each(arr.begin(), arr.end(), doubleNumber);
15.
16.  for (int i : arr)
17.  {
18.    std::cout << i << ' ';
19.  }
```

```
20.
21.   std::cout << '\n';
22.
23.   return 0;
24. }
```

Результат выполнения программы:

```
2  4  6  8
```

Новичкам данный способ может показаться ненужным алгоритмом, потому что эквивалентный код с использованием цикла for с явным указанием диапазона будет короче и проще. Но плюс `std::for_each()` состоит в том, что у нас есть возможность повторного использования тела цикла и применения распараллеливания при его обработке, что делает `std::for_each()` более подходящим инструментом для больших проектов с большим объемом данных.

Порядок выполнения

Обратите внимание, что большинство алгоритмов в библиотеке алгоритмов не гарантируют определенного порядка выполнения. Для использования таких алгоритмов вам нужно позаботиться о том, чтобы любые передаваемые функции не предполагали заданного порядка выполнения, так как порядок вызова этих функций может быть различным в зависимости от используемого компилятора.

Следующие алгоритмы гарантируют последовательное выполнение:

- `std::for_each()`
- `std::copy()`
- `std::copy_backward()`
- `std::move()`
- `std::move_backward()`

> ***Совет:*** Если не указано иное, считайте, что для алгоритмов из Стандартной библиотеки C++ порядок выполнения является неопределенным. Алгоритмы, приведенные выше, дают гарантию последовательного выполнения.

Диапазоны в C++20

Необходимость для каждого алгоритма явно передавать `arr.begin()` и `arr.end()` может немного раздражать. Но в стандарте C++20 добавлен такой инструмент, как **диапазоны**, который позволит нам просто передавать `arr`. Благодаря этому мы сможем сделать наш код еще короче и читабельнее.

Заключение

Библиотека алгоритмов имеет массу полезных функций, которые могут сделать ваш код проще и надежнее. На этом уроке мы рассмотрели лишь небольшую часть алгоритмов, но, поскольку большинство из них работают схожим образом, как только вы разберетесь с некоторыми из них, вы сможете без больших трудностей использовать и оставшиеся функции.

> ***Совет:*** Отдавайте предпочтение использованию функций из библиотеки алгоритмов, нежели самостоятельному написанию своего собственного функционала для выполнения данных задач.

Глава №6. Итоговый тест

Поздравляем вас с преодолением самой длинной главы этого туториала. Если у вас не было предыдущего опыта в программировании, то эта глава, скорее всего, была для вас наиболее сложной из всех предыдущих. Однако, если вы дошли до этого момента, то всё хорошо — вы справились! Так держать!

Хорошая новость заключается в том, что следующая глава будет легче этой, и очень скоро мы доберемся до самого сердца этого туториала - объектно-ориентированного программирования!

Теория

Массивы позволяют хранить и получать доступ ко многим переменным одного и того же типа данных через один идентификатор. Доступ к элементам массива осуществляется с помощью оператора индекса `[]`. Будьте осторожны с диапазоном массива, не допускайте индексации элементов вне диапазона. Массивы можно инициализировать с помощью списка инициализаторов или uniform-инициализации.

Фиксированные массивы должны иметь длину, установленную во время компиляции. Фиксированные массивы распадаются в указатели при передаче в функцию.

Циклы используются для итераций по массиву. Остерегайтесь ошибок «неучтенных единиц». **Циклы foreach** полезны, когда массив не распадается в указатель.

Массивы можно сделать **многомерными**, используя сразу несколько индексов.

Массивы используются в создании **строк C-style**. Избегайте использования строк C-style, вместо них используйте std::string.

Указатели - это переменные, которые хранят адреса памяти (указывают на) определенных переменных. **Оператор адреса** (`&`) используется для получения адреса переменной. **Оператор разыменования** (`*`) используется для получения значения, на которое указывает указатель.

Нулевой указатель - это указатель, который ни на что не указывает. Указатель можно сделать нулевым, инициализировав или присвоив ему значение `0` (или `nullptr` в C++11). Избегайте использования макроса NULL. Разыменование

нулевого указателя может привести к неожиданным результатам (сбоям). При удалении нулевого указателя ничего плохого не случится.

Указатель на массив не знает длину массива, на который он указывает. Это означает, что оператор sizeof и циклы foreach работать с ним не могут.

Операторы new и delete используются для динамического выделения памяти для указателя, переменной или массива и освобождения этой памяти. Хотя подобное случается крайне редко, оператор new может потерпеть крах, если в операционной системе не останется свободной памяти, поэтому не забывайте выполнять проверку того, возвращает ли оператор new нулевой указатель.

Обязательно используйте оператор **delete[]** для удаления динамически выделенного массива. Указатели, указывающие на освобожденную память, называются **висячими указателями**. Разыменование висячего указателя не приведет ни к чему хорошему.

Невозможность удалить динамически выделенную память приведет к **утечке памяти**, когда указатель, указывающий на эту память, выйдет из области видимости.

Для обычных переменных память выделяется из ограниченного резервуара - **стека**. Память для динамически выделенных переменных выделяется из общего резервуара памяти — **кучи**.

Указатель на константное значение обрабатывает значение, на которое он указывает, как константное:

```
1. int value = 7;
2. const int *ptr = &value; // всё нормально, ptr указывает на "const int"
```

Константный указатель - это указатель, значение которого не может быть изменено после инициализации:

```
1. int value = 7;
2. int *const ptr = &value;
```

Ссылка - это псевдоним для определенной переменной. Ссылки объявляются с использованием амперсанда `&` (в этом контексте это не оператор адреса). Для **константных ссылок** изменить их значения после инициализации нельзя. Ссылки используются для предотвращения копирования данных при их передаче в функцию или из функции.

Оператор выбора элемента (`->`) может использоваться для выбора члена через указатель на структуру. Он сочетает в себе как операцию разыменования, так и обычный доступ к элементам (`.`).

Указатели типа void - это указатели, которые могут указывать на любой тип данных. Они не могут быть разыменованы напрямую. Вы можете использовать оператор static_cast для преобразования их обратно в исходный тип указателя. Какой уже это будет тип — решать вам.

Указатели на указатели позволяют создать указатель, указывающий на другой указатель.

std::array предоставляет весь функционал стандартных обычных фиксированных массивов в языке C++ в форме, которая не будет распадаться в указатель при передаче. Рекомендуется использовать std::array вместо стандартных фиксированных массивов.

std::vector предоставляет весь функционал динамических массивов, но которые при этом могут самостоятельно управлять выделенной себе памятью и запоминают свою длину. Рекомендуется использовать std::vector вместо стандартных динамических массивов.

Тест

Задание №1

Представьте, что вы пишете игру, в которой игрок может иметь 3 типа предметов: зелья здоровья, факелы и стрелы. Создайте перечисление с этими типами предметов и фиксированный массив для хранения количества каждого типа предметов, которое имеет при себе игрок (используйте стандартные фиксированные массивы, а не std::array). У вашего игрока должны быть при себе 3 зелья здоровья, 6 факелов и 12 стрел. Напишите функцию countTotalItems(), которая возвращает общее количество предметов, которые есть у игрока. В функции main() выведите результат работы функции countTotalItems().

Задание №2

Создайте структуру, содержащую имя и оценку учащегося (по шкале от 0 до 100). Спросите у пользователя, сколько учеников он хочет ввести. Динамически выделите массив для хранения всех студентов. Затем попросите пользователя ввести для каждого студента его имя и оценку. Как только пользователь ввел все имена и

оценки, отсортируйте список оценок студентов по убыванию (сначала самый высокий бал). Затем выведите все имена и оценки в отсортированном виде.

Для следующего ввода:

```
Andre
74
Max
85
Anton
12
Josh
17
Sasha
90
```

Вывод должен быть следующим:

```
Sasha got a grade of 90
Max got a grade of 85
Andre got a grade of 74
Josh got a grade of 17
Anton got a grade of 12
```

> ***Подсказка***: Вы можете изменить алгоритм сортировки массива методом выбора из урока №80 для сортировки вашего динамического массива. Если вы напишете сортировку массива отдельной функцией, то массив должен передаваться по адресу (как указатель).

Задание №3

Напишите свою функцию, которая меняет местами значения двух целочисленных переменных. Проверку осуществляйте в функции main().

> ***Подсказка***: Используйте ссылки в качестве параметров.

Задание №4

Напишите функцию для вывода строки C-style символ за символом. Используйте указатель для перехода и вывода каждого символа поочерёдно. Остановите вывод при столкновении с нуль-терминатором. В функции main() протестируйте строку `Hello, world!`.

> ***Подсказка***: Используйте оператор `++` для перевода указателя на следующий символ.

Задание №5

Что не так с каждым из следующих фрагментов кода, и как бы вы их исправили?

a)

```
#include <iostream>

int main()
{
    int array[6] { 0, 2, 4, 7, 9 };
    for (int count = 0; count <= 6; ++count)
        std::cout << array[count] << " ";

    return 0;
}
```

b)

```
#include <iostream>

int main()
{
    int a = 4;
    int b = 6;

    const int *ptr = &a;
    std::cout << *ptr;
    *ptr = 7;
    std::cout << *ptr;
    ptr = &b;
    std::cout << *ptr;

    return 0;
}
```

c)

```
#include <iostream>

void printArray(int array[])
{
    for (const int &element : array)
        std::cout << element << ' ';
}


int main()
{
    int array[] { 8, 6, 4, 2, 0 };
    printArray(array);

    return 0;
}
```

d)

```
#include <iostream>

int main()
{
    double d(4.7);
    int *ptr = &d;
    std::cout << ptr;

    return 0;
}
```

Задание №6

Предположим, что мы хотим написать карточную игру.

a) В колоде карт находятся 52 уникальные карты: 13 достоинств (2, 3, 4, 5, 6, 7, 8, 9, 10, Валет, Дама, Король, Туз) и 4 масти (трефы, бубны, червы, пики). Создайте два перечисления: первое для масти, второе для достоинств карт.

> ***Подсказка***: Добавьте в каждое перечисление еще по одному элементу, который будет обозначать длину этого перечисления.

b) Каждая карта должна быть представлена структурой `Card`, в которой хранится информация о достоинстве и масти карты (например, 4 бубны, король трефы). Создайте эту структуру.

c) Создайте функцию printCard(), параметром которой будет константная ссылка типа структуры `Card`, которая будет выводить значения достоинства и масти определенной карты в виде 2-буквенного кода (например, валет пики будет выводиться как `VP`).

d) Для представления целой колоды карт (52 карты) создайте массив `deck` (используя std::array) и инициализируйте каждый элемент определенной картой.

> ***Подсказка***: Используйте оператор static_cast для конвертации целочисленной переменной в тип перечисления.

e) Напишите функцию printDeck(), которая в качестве параметра принимает константную ссылку на массив `deck` и выводит все значения (карты). Используйте цикл foreach.

f) Напишите функцию swapCard(), которая принимает две карты и меняет местами их значения.

g) Напишите функцию shuffleDeck() для перетасовки колоды карт. Для этого используйте цикл for с итерацией по массиву. Перетасовка карт должна произойти 52 раза. В цикле for выберите случайное число от 1 до 52 и вызовите swapCard(), параметрами которой будут текущая карта и карта, выбранная случайным образом. Добавьте в функцию main() возможность перетасовки и вывода уже обновленной (перетасованной) колоды карт.

> ***Подсказки***:
>
> - Для генерации случайных чисел смотрите урок №74.
> - Не забудьте в начале функции main() вызвать функцию srand().
> - Если вы используете Visual Studio, то не забудьте перед генерацией случайного числа вызвать один раз функцию rand().

h) Напишите функцию getCardValue(), которая возвращает значение карты (например, 2 значит 2, 3 значит 3 и т.д., 10, валет, королева или король — это 10, туз - это 11).

Задание №7

Хорошо, настало время для серьезной игры! Давайте напишем упрощенную версию известной карточной игры Blackjack (русский аналог «Очко» или «21 очко»). Если вы не знакомы с этой игрой и её правилами, то вот ссылка на статью в Википедии о **Блэкджеке**.

Правила нашей версии Blackjack следующие:

- вначале дилер получает одну карту (в реальной жизни, дилер получает две карты, но одна лицевой стороной вниз, поэтому на данном этапе это не имеет значения);
- затем игрок получает две карты;
- игрок начинает;
- игрок может либо "взять" (`hit`), либо "удержаться" (`stand`);
- если игрок "удержался", то его ход завершен, и его результат подсчитывается на основе карт, которые у него есть;
- если игрок "берет", то он получает вторую карту, и значение этой карты добавляется к его уже имеющемуся результату;
- туз обычно считается как 1 или как 11. Чтобы было проще, мы будем считать его как 11;
- если у игрока в результате получается больше 21, то он проиграл;

- ход дилера выполняется после хода игрока;
- дилер берет карты до тех пор, пока его общий результат не достигнет 17 или более очков. Как только этот предел достигнут — дилер карт уже не берет;
- если у дилера больше 21-го, то дилер проиграл, а игрок победил;
- если же у дилера и у игрока до 21 очка, то выигрывает тот, у кого результат больше.

В нашей упрощенной версии Blackjack мы не будем отслеживать, какие конкретно карты были у игрока, а какие у дилера. Мы будем отслеживать только сумму значений карт, которые они получили. Так будет проще.

Начнем с кода, который у нас получился в задании №6. Создайте функцию playBlackjack(), которая возвращает `true`, если игрок побеждает, и `false` — если он проигрывает. Эта функция должна:

- Принимать перетасованную колоду карт (`deck`) в качестве параметра.
- Инициализировать указатель на первую карту (имя указателя — `cardPtr`). Это будет использоваться для раздачи карт из колоды.
- Иметь две целочисленные переменные для хранения результата игрока и дилера.
- Соответствовать правилам, приведенным выше.

> ***Подсказка:*** Самый простой способ раздачи карт из колоды — это заставить указатель указывать на следующую карту в колоде (которая будет раздаваться). Всякий раз, когда нам нужно будет раздать карту, мы получаем значение текущей карты, а затем заставляем указатель указывать на следующую карту. Это можно сделать в одной строке кода:
>
> ```
> getCardValue(*cardPtr++);
> ```
>
> Здесь возвращается значение текущей карты (которое затем может быть добавлено к общему результату игрока или дилера) и указатель cardPtr переходит на следующую карту.

Протестируйте выполнение одиночной игры «Блэкджек» в функции main().

Дополнительные задания

а) Время для критического мышления. Опишите, как бы вы могли модифицировать программу, приведенную выше, для обработки случаев, когда стоимость тузов может равняться 1 очку или 11 очкам.

b) В реальном Блэкджеке, если у игрока и дилера равное количество очков, то результатом является ничья, и ни один из них не выигрывает. Опишите, как бы вы изменили программу, приведенную выше, для учета такого исхода игры.

Урок №102. Параметры и аргументы функций

В первой главе этого туториала мы рассматривали функции на следующих уроках:

- **Урок №15. Функции и оператор возврата return**
- **Урок №16. Параметры и аргументы функций**
- **Урок №22. Прототип функции и Предварительное объявление**
- **Урок №23. Многофайловые программы**
- **Урок №24. Заголовочные файлы**

Перед тем как продолжить, вы должны быть знакомы с концепциями, обсуждаемыми на этих уроках.

Параметры vs. Аргументы

На следующих 3-х уроках мы поговорим о параметрах и аргументах, поэтому давайте вкратце вспомним их определения.

Параметр функции (или ***"формальный параметр"***) - это переменная, создаваемая в объявлении функции:

```
1. void boo(int x); // объявление (прототип функции). x - это параметр
2.
3. void boo(int x) // определение (также объявление). x - это параметр
4. {
5. }
```

Аргумент (или ***"фактический параметр"***) - это значение, которое передает в функцию вызывающий объект (caller):

```
1. boo(7); // 7 - это аргумент, который передается в параметр x
2. boo(y+1); // выражение y+1 - это аргумент, который передается в параметр x
```

Когда функция вызывается, все параметры функции создаются как переменные, а значения аргументов копируются в параметры. Например:

```
1. void boo(int x, int y)
2. {
3. }
4.
5. boo(4, 5);
```

При вызове функции boo() с аргументами `4` и `5`, создаются параметры `x` и `y` функции boo() и им присваиваются соответствующие значения: `4` и `5`. Результатом будет `x = 4` и `y = 5`.

> ***Примечание***: В примере, приведенном выше, порядок обработки параметров в функции boo() будет справа налево, т.е. сначала создастся переменная `y` и ей присвоится значение `5`, а затем уже создастся переменная `x` и ей присвоится значение `4`. Порядок, в котором инициализируются параметры в круглых скобках функции, определяет каждый компилятор отдельно, так как C++ явно не указывает этот порядок обработки. С параметрами-переменными это не столь важно и критично, но если вы будете использовать в качестве параметров функции вызовы других функций (что является плохой практикой и не рекомендуется к использованию), то результат может быть неожиданным.

Рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int prinX()
4. {
5.     std::cout << "x = 4\n";
6.     return 0;
7. }
8.
9. int prinY()
10.{
11.    std::cout << "y = 5\n";
12.    return 0;
13.}
14.
15.void prinAll(int a, int b) {}
16.
17.int main() {
18.
19.    prinAll(prinX(), prinY());  // в качестве параметров функции используются
   вызовы функций X() и Y()
20.    return 0;
21.}
```

Результат выполнения программы:

```
y = 5
x = 4
```

Хотя параметры не объявлены внутри блока функции, они имеют локальную область видимости. Это означает, что они создаются при вызове функции и уничтожаются, когда блок функции завершается:

```
1. void boo(int x, int y) // x и y создаются здесь
2. {
3. } // x и y уничтожаются здесь
```

Существует 3 основных способа передачи аргументов в функцию:

- передача по значению;

- передача по ссылке;
- передача по адресу.

Мы рассмотрим каждый из этих способов по порядку.

Урок №103. Передача по значению

По умолчанию, аргументы в C++ передаются по значению. Когда аргумент **передается по значению**, то его значение копируется в параметр функции. Например:

```
#include <iostream>

void boo(int y)
{
    std::cout << "y = " << y << std::endl;
}

int main()
{
    boo(7); // 1-й вызов

    int x = 8;
    boo(x); // 2-й вызов
    boo(x + 2); // 3-й вызов

    return 0;
}
```

В первом вызове функции boo() аргументом является литерал `7`. При вызове boo() создается переменная `y`, в которую копируется значение `7`. Затем, когда boo() завершает свое выполнение, переменная `y` уничтожается.

Во втором вызове функции boo() аргументом является уже переменная `x = 8`. Когда boo() вызывается во второй раз, переменная `y` создается снова и значение `8` копируется в `y`. Затем, когда boo() завершает свое выполнение, переменная `y` снова уничтожается.

В третьем вызове функции boo() аргументом является выражение `x + 2`, которое вычисляется в значение `10`. Затем это значение передается в переменную `y`. При завершении выполнения функции boo() переменная `y` вновь уничтожается.

Таким образом, результат выполнения программы:

```
y = 7
y = 8
y = 10
```

Поскольку в функцию передается копия аргумента, то исходное значение не может быть изменено функцией.

Это хорошо проиллюстрировано в следующем примере:

```
#include <iostream>

void boo(int y)
{
    std::cout << "y = " << y << '\n';

    y = 8;

    std::cout << "y = " << y << '\n';
} // y уничтожается здесь

int main()
{
    int x = 7;
    std::cout << "x = " << x << '\n';

    boo(x);

    std::cout << "x = " << x << '\n';
    return 0;
}
```

Результат:

```
x = 7
y = 7
y = 8
x = 7
```

В начале функции main() `x` равно `7`. При вызове boo() значение `x` (`7`) передается в параметр `y` функции boo(). Внутри boo() переменной `y` сначала присваивается значение `8`, а затем `y` уничтожается. Значение `x` не изменяется, даже если изменить `y`.

Параметры функции, переданные по значению, также могут быть const. Тогда уже будет 100% гарантия того, что функция не изменит значение параметра.

Плюсы и минусы передачи по значению

Плюсы передачи по значению:

- Аргументы, переданные по значению, могут быть переменными (например, `x`), литералами (например, `8`), выражениями (например, `x + 2`), структурами, классами или перечислителями (т.е. почти всем, чем угодно).
- Аргументы никогда не изменяются функцией, в которую передаются, что предотвращает возникновение побочных эффектов.

Минусы передачи по значению:

- Копирование структур и классов может привести к значительному снижению производительности (особенно, когда функция вызывается много раз).

Когда использовать передачу по значению:

- При передаче фундаментальных типов данных и перечислителей, когда предполагается, что функция не должна изменять аргумент.

Когда не использовать передачу по значению:

- При передаче массивов, структур и классов.

В большинстве случаев, передача по значению - это наилучший способ передачи аргументов фундаментальных типов данных, когда функция не должна изменять исходные значения. Передача по значению является гибкой и безопасной, а в случае фундаментальных типов данных еще и эффективной.

Урок №104. Передача по ссылке

Хотя передача по значению является хорошим вариантом во многих случаях, она имеет несколько ограничений. Во-первых, при передаче по значению большой структуры или класса в функцию, создается копия аргумента и уже эта копия передается в параметр функции. В большинстве своем это бесполезная трата ресурсов, которая снижает производительность.

Во-вторых, при передаче аргументов по значению, единственный способ вернуть значение обратно в вызывающий объект — это использовать возвращаемое значение функции. Но иногда случаются ситуации, когда нужно, чтобы функция изменила значение переданного аргумента. Передача по ссылке решает все эти проблемы.

Передача по ссылке

При **передаче переменной по ссылке** нужно просто объявить параметры функции как ссылки, а не как обычные переменные:

```
1. void func(int &x) // x - это переменная-ссылка
2. {
3.     x = x + 1;
4. }
```

При вызове функции переменная `x` станет ссылкой на аргумент. Поскольку ссылка на переменную обрабатывается точно так же, как и сама переменная, то любые изменения, внесенные в ссылку, приведут к изменениям исходного значения аргумента! В следующем примере это хорошо проиллюстрировано:

```
1. #include <iostream>
2.
3. void boo(int &value)
4. {
5.     value = 7;
6. }
7.
8. int main()
9. {
10.     int value = 6;
11.
12.     std::cout << "value = " << value << '\n';
13.     boo(value);
14.     std::cout << "value = " << value << '\n';
15.     return 0;
16. }
```

Эта программа точно такая же, как и программа из предыдущего урока, за исключением того, что параметром функции boo() теперь является ссылка вместо обычной переменной.

Результат выполнения программы:

```
value = 6
value = 7
```

Как вы можете видеть, функция изменила значение аргумента с `6` на `7`!

Вот еще один пример:

```
1. #include <iostream>
2.
3. void addOne(int &x) // x - это переменная-ссылка
4. {
5.     x = x + 1;
6. } // x уничтожается здесь
7.
8. int main()
9. {
10.     int a = 7;
11.     std::cout << "a = " << a << '\n';
12.     addOne(a);
13.     std::cout << "a = " << a << '\n';
14.     return 0;
15.}
```

Результат выполнения программы:

```
a = 7
a = 8
```

Обратите внимание, значение аргумента `a` было изменено функцией.

Возврат сразу нескольких значений

Иногда нам может понадобиться, чтобы функция возвращала сразу несколько значений. Однако оператор return позволяет функции иметь только одно возвращаемое значение. Одним из способов возврата сразу нескольких значений является использование ссылок в качестве параметров:

```
1. #include <iostream>
2. #include <math.h> // для sin() и cos()
3.
4. void getSinCos(double degrees, double &sinOut, double &cosOut)
5. {
6.     // sin() и cos() принимают радианы, а не градусы, поэтому необходима конвер
   тация
7.     const double pi = 3.14159265358979323846; // значение Пи
```

```
8.      double radians = degrees * pi / 180.0;
9.      sinOut = sin(radians);
10.     cosOut = cos(radians);
11. }
12.
13. int main()
14. {
15.     double sin(0.0);
16.     double cos(0.0);
17.
18.     // Функция getSinCos() возвратит sin и cos в переменные sin и cos
19.     getSinCos(30.0, sin, cos);
20.
21.     std::cout << "The sin is " << sin << '\n';
22.     std::cout << "The cos is " << cos << '\n';
23.     return 0;
24. }
```

Эта функция принимает один параметр (передача по значению) в качестве входных данных и «возвращает» два параметра (передача по ссылке) в качестве выходных данных. Параметры, которые используются только для возврата значений обратно в caller, называются **параметрами вывода**. Они дают понять caller-у, что значения исходных переменных, переданных в функцию, не столь значительны, так как мы ожидаем, что эти переменные будут перезаписаны.

Давайте рассмотрим это детально. Во-первых, в функции main() мы создаем локальные переменные `sin` и `cos`. Они передаются в функцию getSinCos() по ссылке (а не по значению). Это означает, что функция getSinCos() имеет прямой доступ к исходным значениям переменных `sin` и `cos`, а не к их копиям. Функция getSinCos(), соответственно, присваивает новые значения переменным `sin` и `cos` (через ссылки `sinOut` и `cosOut`), перезаписывая их старые значения. Затем main() выводит эти обновленные значения.

Если бы `sin` и `cos` были переданы по значению, а не по ссылке, то функция getSinCos() изменила бы копии `sin` и `cos`, а не исходные значения и эти изменения уничтожились бы в конце функции — переменные вышли бы из локальной области видимости. Но, поскольку `sin` и `cos` передавались по ссылке, любые изменения, внесенные в `sin` или `cos` (через ссылки), сохраняются и за пределами функции getSinCos(). Таким образом, мы можем использовать этот механизм для возврата сразу нескольких значений обратно в caller.

Хотя этот способ хорош, но он также имеет свои нюансы. Во-первых, синтаксис немного непривычен, так как параметры ввода и вывода указываются вместе с вызовом функции. Во-вторых, в caller-е не очевидно, что `sin` и `cos` являются параметрами вывода, и они будут изменены функцией. Это, вероятно, самая опасная часть данного способа передачи (так как может привести к ошибкам). Некоторые программисты считают, что это достаточно большая проблема, и не

советуют передавать аргументы по ссылке, отдав предпочтение передаче по адресу, не смешивая при этом параметры ввода и вывода.

Лично я не рекомендую смешивать параметры ввода и вывода именно по этой причине, но если вы это делаете, то обязательно добавляйте комментарии к коду, описывая, что вы делаете и как это делаете.

Неконстантные ссылки могут ссылаться только на неконстантные l-values (например, на неконстантные переменные), поэтому параметр-ссылка не может принять аргумент, который является константным l-value или r-value (например, литералом или результатом выражения).

Передача по константной ссылке

Одним из самых главных недостатков передачи по значению является то, что все аргументы, переданные по значению, *копируются* в параметры функции. Когда аргументами являются большие структуры или классы, то этот процесс может занять много времени. В случае с передачей по ссылке эта проблема легко решается. Когда аргумент передается по ссылке, то создается ссылка на фактический аргумент (что занимает минимальное количество времени на выполнение), и никакого копирования значений не происходит. Это позволяет передавать большие структуры или классы с минимальной затратой ресурсов.

Однако здесь также могут возникнуть потенциальные проблемы. Ссылки позволяют функции изменять значения аргументов напрямую, что нежелательно, если мы хотим, чтобы аргумент был доступен только для чтения. Когда мы знаем, что функция не должна изменять значение аргумента, но не хотим использовать передачу по значению, то лучшим решением будет использовать **передачу по константной ссылке**.

Вы уже знаете, что константная ссылка - это ссылка на переменную, значение которой изменить через эту же ссылку не получится никак. Следовательно, если мы используем константную ссылку в качестве параметра, то получаем 100% гарантию того, что функция не изменит аргумент!

Запустив следующий фрагмент кода, мы получим ошибку компиляции:

```
1. void boo(const int &y) // y - это константная ссылка
2. {
3.     y = 8; // ошибка компиляции: константная ссылка не может изменить свое же значение!
4. }
```

Использование const полезно по нескольким причинам:

- Мы получаем гарантию от компилятора, что значения, которые не должны быть изменены — не изменятся (компилятор выдаст ошибку, если мы попытаемся сделать нечто подобное тому, что было в вышеприведенном примере).
- Программист, видя const, понимает, что функция не изменит значение аргумента. Это может помочь при отладке программы.
- Мы не можем передать константный аргумент в неконстантную ссылку-параметр. Использование константного параметра гарантирует, что мы сможем передавать как неконстантные, так и константные аргументы в функцию.
- Константные ссылки могут принимать любые типы аргументов, включая l-values, константные l-values и r-values.

Правило: При передаче аргументов по ссылке всегда используйте константные ссылки, если вам не нужно, чтобы функция изменяла значения аргументов.

Плюсы и минусы передачи по ссылке

Плюсы передачи по ссылке:

- Ссылки позволяют функции изменять значение аргумента, что иногда полезно. В противном случае, для гарантии того, что функция не изменит значение аргумента, нужно использовать константные ссылки.
- Поскольку при передаче по ссылке копирования аргументов не происходит, то этот способ гораздо эффективнее и быстрее передачи по значению, особенно при работе с большими структурами или классами.
- Ссылки могут использоваться для возврата сразу нескольких значений из функции (через параметры вывода).

Минусы передачи по ссылке:

- Трудно определить, является ли параметр, переданный по неконстантной ссылке, параметром ввода, вывода или того и другого одновременно. Разумное использование const и суффикса `Out` для внешних переменных решает эту проблему.
- По вызову функции невозможно определить, будет аргумент изменен функцией или нет. Аргумент, переданный по значению или по ссылке, выглядит одинаково. Мы можем определить способ передачи аргумента только просмотрев объявление функции. Это может привести к ситуации,

когда программист не сразу поймет, что функция изменяет значение аргумента.

Когда использовать передачу по ссылке:

- при передаче структур или классов (используйте const, если нужно только для чтения);
- когда нужно, чтобы функция изменяла значение аргумента.

Когда не использовать передачу по ссылке:

- при передаче фундаментальных типов данных (используйте передачу по значению);
- при передаче обычных массивов (используйте передачу по адресу).

Урок №105. Передача по адресу

Есть еще один способ передачи переменных в функцию в языке C++ - по адресу.

Передача по адресу

Передача аргументов по адресу - это передача адреса переменной-аргумента (а не исходной переменной). Поскольку аргумент является адресом, то параметром функции должен быть указатель. Затем функция сможет разыменовать этот указатель для доступа или изменения исходного значения. Вот пример функции, которая принимает параметр, передаваемый по адресу:

```
1. #include <iostream>
2.
3. void boo(int *ptr)
4. {
5.     *ptr = 7;
6. }
7.
8. int main()
9. {
10.     int value = 4;
11.
12.     std::cout << "value = " << value << '\n';
13.     boo(&value);
14.     std::cout << "value = " << value << '\n';
15.     return 0;
16. }
```

Результат выполнения программы:

```
value = 4
value = 7
```

Как вы можете видеть, функция boo() изменила значение аргумента (переменную `value`) через параметр-указатель `ptr`. Передачу по адресу обычно используют с указателями на обычные массивы. Например, следующая функция выведет все значения массива:

```
1. void printArray(int *array, int length)
2. {
3.     for (int index=0; index < length; ++index)
4.         std::cout << array[index] << ' ';
5. }
```

Вот пример программы, которая вызывает эту функцию:

```
1. int main()
2. {
```

```
3.      int array[7] = { 9, 8, 6, 4, 3, 2, 1 }; // помните, что массивы
    распадаются в указатели при передаче
4.      printArray(array, 7); // поэтому здесь array - это указатель на первый
    элемент массива (в использовании оператора & нет необходимости)
5.  }
```

Результат:

```
9 8 6 4 3 2 1
```

Помните, что фиксированные массивы распадаются в указатели при передаче в функцию, поэтому их длину нужно передавать в виде отдельного параметра. Перед разыменованием параметров, передаваемых по адресу, не лишним будет проверить — не являются ли они нулевыми указателями. Разыменование нулевого указателя приведет к сбою в программе. Вот функция printArray() с проверкой (обнаружением) нулевых указателей:

```
1.  #include <iostream>
2.
3.  void printArray(int *array, int length)
4.  {
5.      // Если пользователь передал нулевой указатель в качестве array
6.      if (!array)
7.          return;
8.
9.      for (int index=0; index < length; ++index)
10.         std::cout << array[index] << ' ';
11. }
12.
13. int main()
14. {
15.     int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
16.     printArray(array, 7);
17. }
```

Передача по константному адресу

Поскольку printArray() все равно не изменяет значения получаемых аргументов, то хорошей идеей будет сделать параметр `array` константным:

```
1.  #include <iostream>
2.
3.  void printArray(const int *array, int length)
4.  {
5.      // Если пользователь передал нулевой указатель в качестве array
6.      if (!array)
7.          return;
8.
9.      for (int index=0; index < length; ++index)
10.         std::cout << array[index] << ' ';
11. }
12.
13. int main()
14. {
```

```
    int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
    printArray(array, 7);
}
```

Так мы видим сразу, что printArray() не изменит переданный аргумент `array`. Когда вы передаете указатель в функцию по адресу, то значение этого указателя (адрес, на который он указывает) копируется из аргумента в параметр функции. Другими словами, он передается по значению! Если изменить значение параметра функции, то изменится только копия, исходный указатель-аргумент не будет изменен. Например:

```
#include <iostream>

void setToNull(int *tempPtr)
{
    // Мы присваиваем tempPtr другое значение (мы не изменяем значение, на которое указывает tempPtr)
    tempPtr = nullptr; // используйте 0, если не поддерживается C++11
}

int main()
{
    // Сначала мы присваиваем ptr адрес six, т.е. *ptr = 6
    int six = 6;
    int *ptr = &six;

    // Здесь выведется 6
    std::cout << *ptr << "\n";

    // tempPtr получит копию ptr
    setToNull(ptr);

    // ptr до сих пор указывает на переменную six!

    // Здесь выведется 6
    if (ptr)
        std::cout << *ptr << "\n";
    else
        std::cout << " ptr is null";

    return 0;
}
```

В `tempPtr` копируется адрес указателя `ptr`. Несмотря на то, что мы изменили `tempPtr` на нулевой указатель (присвоили ему `nullptr`), это никак не повлияло на значение, на которое указывает `ptr`. Следовательно, результат выполнения программы:

```
6
6
```

Обратите внимание, хотя сам адрес передается по значению, вы все равно можете разыменовать его для изменения значения исходного аргумента. Запутано? Давайте проясним:

- При передаче аргумента по адресу в переменную-параметр функции копируется адрес из аргумента. В этот момент параметр функции и аргумент указывают на одно и то же значение.
- Если параметр функции затем разыменовать для изменения исходного значения, то это приведет к изменению значения, на которое указывает аргумент, поскольку параметр функции и аргумент указывают на одно и то же значение!
- Если параметру функции присвоить другой адрес, то это никак не повлияет на аргумент, поскольку параметр функции является копией, а изменение копии не приводит к изменению оригинала. После изменения адреса параметра функции, параметр функции и аргумент будут указывать на разные значения, поэтому разыменование параметра и дальнейшее его изменение никак не повлияют на значение, на которое указывает аргумент.

В следующей программе это всё хорошо проиллюстрировано:

```
#include <iostream>

void setToSeven(int *tempPtr)
{
    *tempPtr = 7; // мы изменяем значение, на которое указывает tempPtr (и ptr тоже)
}

int main()
{
    // Сначала мы присваиваем ptr адрес six, т.е. *ptr = 6
    int six = 6;
    int *ptr = &six;

    // Здесь выведется 6
    std::cout << *ptr << "\n";

    // tempPtr получит копию ptr
    setToSeven(ptr);

    // tempPtr изменил значение, на которое указывал, на 7

    // Здесь выведется 7
    if (ptr)
        std::cout << *ptr << "\n";
    else
        std::cout << " ptr is null";

    return 0;
}
```

Результат выполнения программы:

```
6
7
```

Передача адресов по ссылке

Следует вопрос: «А что, если мы хотим изменить адрес, на который указывает аргумент, внутри функции?». Оказывается, это можно сделать очень легко. Вы можете просто передать адрес по ссылке. Синтаксис ссылки на указатель может показаться немного странным, но все же:

```
1. #include <iostream>
2.
3. // tempPtr теперь является ссылкой на указатель, поэтому любые изменения
   tempPtr приведут и к изменениям исходного аргумента!
4. void setToNull(int *&tempPtr)
5. {
6.     tempPtr = nullptr; // используйте 0, если не поддерживается C++11
7. }
8.
9. int main()
10.{
11.    // Сначала мы присваиваем ptr адрес six, т.е. *ptr = 6
12.    int six = 6;
13.    int *ptr = &six;
14.
15.    // Здесь выведется 6
16.    std::cout << *ptr;
17.
18.    // tempPtr является ссылкой на ptr
19.    setToNull(ptr);
20.
21.    // ptr было присвоено значение nullptr!
22.
23.    if (ptr)
24.        std::cout << *ptr;
25.    else
26.        std::cout << " ptr is null";
27.
28.    return 0;
29.}
```

Результат выполнения программы:

```
6 ptr is null
```

Наконец, наша функция setToNull() действительно изменила значение `ptr` с `&six` на `nullptr`!

Существует только передача по значению

Теперь, когда вы понимаете основные различия между передачей по ссылке, по адресу и по значению, давайте немного поговорим о том, что находится "под капотом".

На уроке о ссылках мы упоминали, что ссылки на самом деле реализуются с помощью указателей. Это означает, что передача по ссылке является просто передачей по адресу. И чуть выше мы говорили, что передача по адресу на самом деле является передачей адреса по значению! Из этого следует, что C++ действительно передает всё по значению!

Плюсы и минусы передачи по адресу

Плюсы передачи по адресу:

- Передача по адресу позволяет функции изменить значение аргумента, что иногда полезно. В противном случае, используем const для гарантии того, что функция не изменит аргумент.
- Поскольку копирования аргументов не происходит, то скорость передачи по адресу достаточно высокая, даже если передавать большие структуры или классы.
- Мы можем вернуть сразу несколько значений из функции, используя параметры вывода.

Минусы передачи по адресу:

- Все указатели нужно проверять, не являются ли они нулевыми. Попытка разыменовать нулевой указатель приведет к сбою в программе.
- Поскольку разыменование указателя выполняется медленнее, чем доступ к значению напрямую, то доступ к аргументам, переданным по адресу, выполняется также медленнее, чем доступ к аргументам, переданным по значению.

Когда использовать передачу по адресу:

- при передаче обычных массивов (если нет никаких проблем с тем, что массивы распадаются в указатели при передаче).

Когда не использовать передачу по адресу:

- при передаче структур или классов (используйте передачу по ссылке);

- при передаче фундаментальных типов данных (используйте передачу по значению).

Как вы можете видеть сами, передача по адресу и по ссылке имеют почти одинаковые преимущества и недостатки. Поскольку передача по ссылке обычно безопаснее, чем передача по адресу, то в большинстве случаев предпочтительнее использовать передачу по ссылке.

Правило: Используйте передачу по ссылке, вместо передачи по адресу, когда это возможно.

Урок №106. Возврат значений по ссылке, по адресу и по значению

На 3-х предыдущих уроках мы узнали о передаче аргументов в функции по значению, по ссылке и по адресу. На этом уроке мы рассмотрим возврат значений обратно из функции в вызывающий объект всеми этими тремя способами.

Возврат значений (с помощью оператора return) работает почти так же, как и передача значений в функцию. Все те же плюсы и минусы. Основное отличие состоит в том, что поток данных двигается уже в противоположную сторону. Однако здесь есть еще один нюанс — локальные переменные, которые выходят из области видимости и уничтожаются, когда функция завершает свое выполнение.

Возврат по значению

Возврат по значению - это самый простой и безопасный тип возврата. При возврате по значению, копия возвращаемого значения передается обратно в caller. Как и в случае с передачей по значению, вы можете возвращать литералы (например, `7`), переменные (например, `x`) или выражения (например, `x + 2`), что делает этот способ очень гибким.

Еще одним преимуществом является то, что вы можете возвращать переменные (или выражения), в вычислении которых задействованы и локальные переменные, объявленные в теле самой функции. При этом, можно не беспокоиться о проблемах, которые могут возникнуть с областью видимости. Поскольку переменные вычисляются до того, как функция производит возврат значения, то здесь не должно быть никаких проблем с областью видимости этих переменных, когда заканчивается блок, в котором они объявлены. Например:

```
1. int doubleValue(int a)
2. {
3.     int value = a * 3;
4.     return value; // копия value возвращается здесь
5. } // value выходит из области видимости здесь
```

Возврат по значению идеально подходит для возврата переменных, которые были объявлены внутри функции, или для возврата аргументов функции, которые были переданы по значению. Однако, подобно передаче по значению, возврат по значению медленный при работе со структурами и классами.

Когда использовать возврат по значению:

- при возврате переменных, которые были объявлены внутри функции;
- при возврате аргументов функции, которые были переданы в функцию по значению.

Когда не использовать возврат по значению:

- при возврате стандартных массивов или указателей (используйте возврат по адресу);
- при возврате больших структур или классов (используйте возврат по ссылке).

Возврат по адресу

Возврат по адресу — это возврат адреса переменной обратно в caller. Подобно передаче по адресу, возврат по адресу может возвращать только адрес переменной. Литералы и выражения возвращать нельзя, так как они не имеют адресов. Поскольку при возврате по адресу просто копируется адрес из функции в caller, то этот процесс также очень быстрый.

Тем не менее, этот способ имеет один недостаток, который отсутствует при возврате по значению: если вы попытаетесь возвратить адрес локальной переменной, то получите неожиданные результаты. Например:

```
1. int* doubleValue(int a)
2. {
3.     int value = a * 3;
4.     return &value; // value возвращается по адресу здесь
5. } // value уничтожается здесь
```

Как вы можете видеть, `value` уничтожается сразу после того, как её адрес возвращается в caller. Конечным результатом будет то, что caller получит адрес освобожденной памяти (висячий указатель), что, несомненно, вызовет проблемы. Это одна из самых распространенных ошибок, которую делают новички. Большинство современных компиляторов выдадут предупреждение (а не ошибку), если программист попытается вернуть локальную переменную по адресу. Однако есть несколько способов обмануть компилятор, чтобы сделать что-то "плохое", не генерируя при этом предупреждения, поэтому вся ответственность лежит на программисте, который должен гарантировать, что возвращаемый адрес будет корректен.

Возврат по адресу часто используется для возврата динамически выделенной памяти обратно в caller:

```
int* allocateArray(int size)
{
    return new int[size];
}

int main()
{
    int *array = allocateArray(20);

    // Делаем что-нибудь с array

    delete[] array;
    return 0;
}
```

Здесь не возникнет никаких проблем, так как динамически выделенная память не выходит из области видимости в конце блока, в котором объявлена, и все еще будет существовать, когда адрес будет возвращаться в caller.

Когда использовать возврат по адресу:

- при возврате динамически выделенной памяти;
- при возврате аргументов функции, которые были переданы по адресу.

Когда не использовать возврат по адресу:

- при возврате переменных, которые были объявлены внутри функции (используйте возврат по значению);
- при возврате большой структуры или класса, который был передан по ссылке (используйте возврат по ссылке).

Возврат по ссылке

Подобно передаче по ссылке, значения, возвращаемые по ссылке, должны быть переменными (вы не сможете вернуть ссылку на литерал или выражение). При **возврате по ссылке** в caller возвращается ссылка на переменную. Затем caller может её использовать для продолжения изменения переменной, что может быть иногда полезно. Этот способ также очень быстрый и при возврате больших структур или классов.

Однако, как и при возврате по адресу, вы не должны возвращать локальные переменные по ссылке. Рассмотрим следующий фрагмент кода:

```
int& doubleValue(int a)
```

```
2. {
3.     int value = a * 3;
4.     return value; // value возвращается по ссылке здесь
5. } // value уничтожается здесь
```

В программе, приведенной выше, возвращается ссылка на переменную `value`, которая уничтожится, когда функция завершит свое выполнение. Это означает, что caller получит ссылку на мусор. К счастью, ваш компилятор, вероятнее всего, выдаст предупреждение или ошибку, если вы попытаетесь это сделать.

Возврат по ссылке обычно используется для возврата аргументов, переданных в функцию по ссылке. В следующем примере мы возвращаем (по ссылке) элемент массива, который был передан в функцию по ссылке:

```
1. #include <iostream>
2. #include <array>
3.
4. // Возвращаем ссылку на элемент массива по индексу index
5. int& getElement(std::array<int, 20> &array, int index)
6. {
7.     // Мы знаем, что array[index] не уничтожится, когда мы будем возвращать
   данные в caller (так как caller сам передал этот array в функцию!)
8.     // Так что здесь не должно быть никаких проблем с возвратом по ссылке
9.     return array[index];
10. }
11.
12. int main()
13. {
14.     std::array<int, 20> array;
15.
16.     // Присваиваем элементу массива под индексом 15 значение 7
17.     getElement(array, 15) = 7;
18.
19.     std::cout << array[15] << '\n';
20.
21.     return 0;
22. }
```

Результат выполнения программы:

```
7
```

Когда мы вызываем `getElement(array, 15)`, то getElement() возвращает ссылку на элемент массива под индексом 15, а затем main() использует эту ссылку для присваивания этому элементу значения `7`.

Хотя этот пример непрактичен, так как мы можем напрямую обратиться к 15 элементу массива, но как только мы будем рассматривать классы, то вы обнаружите гораздо больше применений для возврата значений по ссылке.

Когда использовать возврат по ссылке:

- при возврате ссылки-параметра;
- при возврате элемента массива, который был передан в функцию;
- при возврате большой структуры или класса, который не уничтожается в конце функции (например, тот, который был передан в функцию).

Когда не использовать возврат по ссылке:

- при возврате переменных, которые были объявлены внутри функции (используйте возврат по значению);
- при возврате стандартного массива или значения указателя (используйте возврат по адресу).

Смешивание возвращаемых значений и ссылок

Хотя функция может возвращать как значение, так и ссылку, caller может неправильно это интерпретировать. Посмотрим, что произойдет при смешивании возвращаемых значений и ссылок на значения:

```
1. int returnByValue()
2. {
3.     return 7;
4. }
5.
6. int& returnByReference()
7. {
8.      static int y = 7; // static гарантирует то, что переменная y не
   уничтожится, когда выйдет из локальной области видимости
9.      return y;
10.}
11.
12.int main()
13.{
14.    int value = returnByReference(); // случай A: всё хорошо, обрабатывается
   как возврат по значению
15.    int &ref = returnByValue(); // случай B: ошибка компилятора, так как 7 -
   это r-value, а r-value не может быть привязано к неконстантной ссылке
16.    const int &cref = returnByValue(); // случай C: всё хорошо, время жизни
   возвращаемого значения продлевается в соответствии со временем жизни cref
17.}
```

В случае A мы присваиваем ссылку возвращаемого значения переменной, которая сама не является ссылкой. Поскольку `value` не является ссылкой, то возвращаемое значение просто копируется в `value` так, как если бы returnByReference() был возвратом по значению.

В случае B мы пытаемся инициализировать ссылку `ref` копией возвращаемого значения функции returnByValue(). Однако, поскольку возвращаемое значение не имеет адреса (это r-value), мы получим ошибку компиляции.

В случае C мы пытаемся инициализировать константную ссылку `cref` копией возвращаемого значения функции returnByValue(). Поскольку константные ссылки могут быть инициализированы с помощью r-values, то здесь не должно быть никаких проблем. Обычно r-values уничтожаются в конце выражения, в котором они созданы, однако, при привязке к константной ссылке, время жизни r-value (в данном случае, возвращаемого значения функции) продлевается в соответствии со временем жизни ссылки (в данном случае, `cref`).

Заключение

В большинстве случаев идеальным вариантом для использования является возврат по значению. Это также самый гибкий и безопасный способ возврата данных обратно в вызывающий объект. Однако возврат по ссылке или по адресу также может быть полезен при работе с динамически выделенной памятью. При использовании возврата по ссылке или по адресу убедитесь, что вы не возвращаете ссылку или адрес локальной переменной, которая выйдет из области видимости, когда функция завершит свое выполнение!

Тест

Напишите прототипы для каждой из следующих функций. Используйте наиболее подходящие параметры и типы возврата (по значению, по адресу или по ссылке). Используйте const, когда это необходимо.

Задание №1

Функция sumTo(), которая принимает целочисленный параметр, а возвращает сумму всех чисел между `1` и числом, которое ввел пользователь.

Задание №2

Функция printAnimalName(), которая принимает структуру `Animal` в качестве параметра.

Задание №3

Функция minmax(), которая принимает два целых числа в качестве входных данных, а возвращает наименьшее и наибольшее числа в качестве отдельных параметров.

> ***Подсказка*:** Используйте параметры вывода.

Задание №4

Функция getIndexOfLargestValue(), которая принимает целочисленный массив (как указатель) и его размер, а возвращает индекс наибольшего элемента массива.

Задание №5

Функция getElement(), которая принимает целочисленный массив (как указатель) и индекс, а возвращает элемент массива по этому индексу (не копию элемента). Предполагается, что индекс корректен, а возвращаемое значение - константное.

Урок №107. Встроенные функции

Использование функций имеет много преимуществ, в том числе:

- Код, находящийся внутри функции, может быть повторно использован.
- Гораздо проще изменить или обновить код в функции (что делается один раз), нежели искать и изменять все части кода в функции main() "на месте". Дублирование кода - хороший рецепт для ошибок и ухудшения производительности.
- Упрощение чтения и понимания кода, так как вам не нужно знать реализацию функции, чтобы её использовать (предполагается наличие информативного названия функции и комментариев).
- В функциях поддерживается проверка типов данных для гарантии того, что передаваемые аргументы соответствуют параметрам функции.
- Функции упрощают отладку вашей программы.

Однако, одним из главных недостатков использования функций является то, что при каждом её вызове происходит расход ресурсов, что влияет на производительность программы. Это связано с тем, что ЦП должен хранить адрес текущей команды (инструкции или стейтмента), которую он выполняет (чтобы знать, куда нужно будет вернуться позже), вместе с другими данными. Затем точка выполнения перемещается в другое место программы. Дальше все параметры функции должны быть созданы и им должны быть присвоены значения. И только потом, после выполнения функции, точка выполнения возвращается обратно. Код, написанный "на месте", выполняется значительно быстрее.

Для функций, которые являются большими и/или выполняют сложные задачи, расходы на вызов обычно незначительны по сравнению с количеством времени, которое отводится на выполнение кода этой функции. Однако для небольших, часто используемых функций, время, необходимое для выполнения вызова, часто превышает время, необходимое для фактического выполнения кода этой функции. А это, в свою очередь, может привести к существенному снижению производительности.

Язык C++ предлагает возможность совместить все преимущества функций вместе с высокой производительностью кода, написанного "на месте". Речь идет о встроенных функциях. **Ключевое слово inline** используется для запроса, чтобы компилятор рассматривал вашу функцию как встроенную. При компиляции вашего кода, все **встроенные функции** (англ. *"inline functions"*) раскрываются "на месте", то есть вызов функции заменяется копией содержимого самой функции, и ресурсы,

которые могли бы быть потрачены на вызов этой функции, сохраняются! Минусом является лишь увеличение компилируемого кода за счет того, что встроенная функция раскрывается в коде при каждом вызове (особенно если она длинная и/или её вызывают много раз). Рассмотрим следующий фрагмент кода:

```
#include <iostream>

int max(int a, int b)
{
    return a < b ? b : a;
}

int main()
{
    std::cout << max(7, 8) << '\n';
    std::cout << max(5, 4) << '\n';
    return 0;
}
```

Эта программа дважды вызывает функцию max(), т.е. дважды расходуются ресурсы на вызов функции. Поскольку max() является довольно таки короткой функцией, то это идеальный вариант для её конвертации во встроенную функцию:

```
inline int max(int a, int b)
{
    return a < b ? b : a;
}
```

Теперь, при компиляции функции main(), ЦП будет читать код следующим образом:

```
int main()
{
    std::cout << (7 < 8 ? 8 : 7) << '\n';
    std::cout << (5 < 4 ? 4 : 5) << '\n';
    return 0;
}
```

Такой код выполнится быстрее ценой несколько увеличенного объема.

Из-за возможности подобного «раздувания», встроенные функции лучше всего использовать только для коротких функций (не более нескольких строк), которые обычно вызываются внутри циклов и не имеют ветвлений. Также, обратите внимание, ключевое слово inline является лишь рекомендацией - компилятор может игнорировать ваш запрос на встроенную функцию. Подобное произойдет, если вы попытаетесь сделать встроенной длинную функцию!

Наконец, современные компиляторы автоматически конвертируют соответствующие функции во встроенные — этот процесс автоматизирован настолько, что даже лучше ручной выборочной конвертации, проведенной программистом. Даже если вы не пометите функцию как встроенную, компилятор

автоматически выполнит её как таковую, если посчитает, что это способствует улучшению производительности. Таким образом, в большинстве случаев нет особой необходимости использовать ключевое слово inline. Компилятор всё сделает сам.

Правило: Если вы используете современный компилятор, то нет необходимости использовать ключевое слово inline.

Встроенные функции освобождаются от "правила одного определения"

На предыдущих уроках мы не раз говорили, что вы не должны определять функции в заголовочных файлах, так как если вы подключаете один заголовок с определением функции в несколько файлов .cpp, то определение функции также будет скопировано несколько раз. Затем, на этапе линкинга, линкер выдаст ошибку, что вы определяете одну и ту же функцию больше одного раза.

Однако встроенные функции освобождаются от этого правила, так как дублирования в исходном коде не происходит - определение функции одно, и никакого конфликта при соединении линкером файлов .cpp возникнуть не должно.

Сейчас это всё может показаться неинтересными пустяками, но чуть позже, когда мы будем рассматривать новый тип функций (дружественные функции), эти *пустяки* пригодятся. И помните, что даже с использованием встроенных функций, вы НЕ должны определять глобальные функции в заголовочных файлах.

Урок №108. Перегрузка функций

Перегрузка функций - это возможность определять несколько функций с одним и тем же именем, но с разными параметрами. Например:

```
1. int subtract(int a, int b)
2. {
3.     return a - b;
4. }
```

Здесь мы выполняем операцию вычитания с целыми числами. Однако, что, если нам нужно использовать числа типа с плавающей запятой? Эта функция совсем не подходит, так как любые параметры типа double будут конвертироваться в тип int, в результате чего будет теряться дробная часть значений.

Одним из способов решения этой проблемы является определение двух функций с разными именами и параметрами:

```
1. int subtractInteger(int a, int b)
2. {
3.     return a - b;
4. }
5.
6. double subtractDouble(double a, double b)
7. {
8.     return a - b;
9. }
```

Но есть и лучшее решение — перегрузка функции. Мы можем просто объявить еще одну функцию subtract(), которая принимает параметры типа double:

```
1. double subtract(double a, double b)
2. {
3.     return a - b;
4. }
```

Теперь у нас есть две версии функции subtract():

```
1. int subtract(int a, int b); // целочисленная версия
2. double subtract(double a, double b); // версия типа с плавающей запятой
```

Может показаться, что произойдет конфликт имен, но это не так. Компилятор может определить сам, какую версию subtract() следует вызывать на основе аргументов, используемых в вызове функции. Если параметрами будут переменные типа int, то C++ понимает, что мы хотим вызвать `subtract(int, int)`. Если же мы предоставим два значения типа с плавающей запятой, то C++ поймет, что мы хотим вызвать `subtract(double, double)`. Фактически, мы можем определить

столько перегруженных функций subtract(), сколько хотим, до тех пор, пока каждая из них будет иметь свои (уникальные) параметры.

Следовательно, можно определить функцию subtract() и с большим количеством параметров:

```
1. int subtract(int a, int b, int c)
2. {
3.     return a - b - c;
4. }
```

Хотя здесь subtract() имеет 3 параметра вместо 2-х, это не является ошибкой, поскольку эти параметры отличаются от параметров других версий subtract().

Типы возврата в перегрузке функций

Обратите внимание, тип возврата функции НЕ учитывается при перегрузке функции. Предположим, что вы хотите написать функцию, которая возвращает рандомное число, но вам нужна одна версия, которая возвращает значение типа int, и вторая — которая возвращает значение типа double. У вас может возникнуть соблазн сделать следующее:

```
1. int getRandomValue();
2. double getRandomValue();
```

Компилятор выдаст ошибку. Эти две функции имеют одинаковые параметры (точнее, они отсутствуют), и, следовательно, второй вызов функции getRandomValue() будет рассматриваться как ошибочное переопределение первого вызова. Имена функций нужно будет изменить.

Псевдонимы типов в перегрузке функций

Поскольку объявление typedef (псевдонима типа) не создает новый тип данных, то следующие два объявления функции print() считаются идентичными:

```
1. typedef char *string;
2. void print(string value);
3. void print(char *value);
```

Вызовы функций

Выполнение вызова перегруженной функции приводит к одному из 3-х возможных результатов:

- **Совпадение найдено.** Вызов разрешен для соответствующей перегруженной функции.

- **Совпадение не найдено.** Аргументы не соответствуют любой из перегруженных функций.
- **Найдены несколько совпадений.** Аргументы соответствуют более чем одной перегруженной функции.

При компиляции перегруженной функции, C++ выполняет следующие шаги для определения того, какую версию функции следует вызывать:

***Шаг №1:* C++ пытается найти точное совпадение**. Это тот случай, когда фактический аргумент точно соответствует типу параметра одной из перегруженных функций. Например:

```
1. void print(char *value);
2. void print(int value);
3.
4. print(0); // точное совпадение с print(int)
```

Хотя `0` может технически соответствовать и `print(char *)` (как нулевой указатель), но он точно соответствует `print(int)`. Таким образом, `print(int)` является лучшим (точным) совпадением.

***Шаг №2:* Если точного совпадения не найдено, то C++ пытается найти совпадение путем дальнейшего неявного преобразования типов**. На предыдущих уроках мы говорили о том, как определенные типы данных могут автоматически конвертироваться в другие типы данных. Если вкратце, то:

- char, unsigned char и short конвертируются в int;
- unsigned short может конвертироваться в int или unsigned int (в зависимости от размера int);
- float конвертируется в double;
- enum конвертируется в int.

Например:

```
1. void print(char *value);
2. void print(int value);
3.
4. print('b'); // совпадение с print(int) после неявного преобразования
```

В этом случае, поскольку нет `print(char)`, символ `b` конвертируется в тип int, который уже соответствует `print(int)`.

***Шаг №3:* Если неявное преобразование невозможно, то C++ пытается найти соответствие посредством стандартного преобразования**. В стандартном преобразовании:

- Любой числовой тип будет соответствовать любому другому числовому типу, включая unsigned (например, int равно float).
- enum соответствует формальному типу числового типа данных (например, enum равно float).
- Ноль соответствует типу указателя и числовому типу (например, `0` как `char *` или `0` как `float`).
- Указатель соответствует указателю типа void.

Например:

```
1. struct Employee; // определение упустим
2. void print(float value);
3. void print(Employee value);
4.
5. print('b'); // 'b' конвертируется в соответствие версии print(float)
```

В этом случае, поскольку нет `print(char)` (точного совпадения) и нет `print(int)` (совпадения путем неявного преобразования), символ `b` конвертируется в тип float и сопоставляется с `print(float)`.

Обратите внимание, все стандартные преобразования считаются равными. Ни одно из них не считается выше остальных по приоритету.

***Шаг №4:* C++ пытается найти соответствие путем пользовательского преобразования**. Хотя мы еще не рассматривали классы, но они могут определять преобразования в другие типы данных, которые могут быть неявно применены к объектам этих классов. Например, мы можем создать класс `W` и в нем определить пользовательское преобразование в тип int:

```
1. class W; // с пользовательским преобразованием в тип int
2.
3. void print(float value);
4. void print(int value);
5.
6. W value; // объявляем переменную value типа класса W
7. print(value); // value конвертируется в int и, следовательно, соответствует
   print(int)
```

Хотя `value` относится к типу класса `W`, но, поскольку тот имеет пользовательское преобразование в тип int, вызов `print(value)` соответствует версии `print(int)`.

То, как делать пользовательские преобразования в классах, мы рассмотрим на соответствующих уроках.

Несколько совпадений

Если каждая из перегруженных функций должна иметь уникальные параметры, то как могут быть возможны несколько совпадений? Поскольку все стандартные и пользовательские преобразования считаются равными, то, если вызов функции соответствует нескольким кандидатам посредством стандартного или пользовательского преобразования, результатом будет **неоднозначное совпадение** (т.е. несколько совпадений). Например:

```
1. void print(unsigned int value);
2. void print(float value);
3.
4. print('b');
5. print(0);
6. print(3.14159);
```

В случае с `print('b')` C++ не может найти точного совпадения. Он пытается преобразовать `b` в тип int, но версии `print(int)` тоже нет. Используя стандартное преобразование, C++ может преобразовать `b` как в unsigned int, так и во float. Поскольку все стандартные преобразования считаются равными, то получается два совпадения.

С `print(0)` всё аналогично. `0` - это int, а версии `print(int)` нет. Путем стандартного преобразования мы опять получаем два совпадения.

А вот с `print(3.14159)` всё несколько запутаннее: большинство программистов отнесут его однозначно к `print(float)`. Однако, помните, что по умолчанию все значения-литералы типа с плавающей запятой относятся к типу double, если у них нет окончания `f`. `3.14159` — это значение типа double, а версии `print(double)` нет. Следовательно, мы получаем ту же ситуацию, что и в предыдущих случаях — неоднозначное совпадение (два варианта).

Неоднозначное совпадение считается ошибкой типа compile-time. Следовательно, оно должно быть устранено до того, как ваша программа скомпилируется. Есть два решения этой проблемы:

Решение №1: Просто определить новую перегруженную функцию, которая принимает параметры именно того типа данных, который вы используете в вызове функции. Тогда C++ сможет найти точное совпадение.

Решение №2: Явно преобразовать с помощью операторов явного преобразования неоднозначный параметр(ы) в соответствии с типом функции, которую вы хотите вызвать. Например, чтобы вызов `print(0)` соответствовал `print(unsigned int)`, вам нужно сделать следующее:

```
1. print(static_cast<unsigned int>(0)); // произойдет вызов print(unsigned int)
```

Заключение

Перегрузка функций может значительно снизить сложность программы, в то же время создавая небольшой дополнительный риск. Хотя этот урок несколько долгий и может показаться сложным, но, на самом деле, перегрузка функций обычно работает прозрачно и без каких-либо проблем. Все неоднозначные случаи компилятор будет отмечать, и их можно будет легко исправить.

Правило: Используйте перегрузку функций для упрощения ваших программ.

Урок №109. Параметры по умолчанию

Параметр по умолчанию (или ***«необязательный параметр»***) - это параметр функции, который имеет определенное (по умолчанию) значение. Если пользователь не передает в функцию значение для параметра, то используется значение по умолчанию. Если же пользователь передает значение, то это значение используется вместо значения по умолчанию. Например:

```
1. #include <iostream>
2.
3. void printValues(int a, int b=5)
4. {
5.     std::cout << "a: " << a << '\n';
6.     std::cout << "b: " << b << '\n';
7. }
8.
9. int main()
10.{
11.    printValues(1); // в качестве b будет использоваться значение по
   умолчанию - 5
12.    printValues(6, 7); // в качестве b будет использоваться значение,
   предоставляемое пользователем - 7
13.}
```

Результат выполнение программы:

```
a: 1
b: 5
a: 6
b: 7
```

В первом вызове функции мы не передаем аргумент для `b`, поэтому функция использует значение по умолчанию - `5`. Во втором вызове мы передаем значение для `b`, поэтому оно используется вместо параметра по умолчанию.

Параметр по умолчанию — это отличный вариант, когда функция нуждается в значении, которое пользователь может переопределить, а может и не переопределить. Например, вот несколько прототипов функций, для которых могут использоваться параметры по умолчанию:

```
1. void openLogFile(std::string filename="default.log");
2. int rollDie(int sides=6);
3. void printStringInColor(std::string str, Color color=COLOR_RED); // Color - это
    перечисление
```

Несколько параметров по умолчанию

Функция может иметь несколько параметров по умолчанию:

```
1. void printValues(int a=10, int b=11, int c=12)
2. {
3.     std::cout << "Values: " << a << " " << b << " " << c << '\n';
4. }
```

При следующих вызовах функции:

```
1. printValues(3, 4, 5);
2. printValues(3, 4);
3. printValues(3);
4. printValues();
```

Результат следующий:

```
Values: 3 4 5
Values: 3 4 12
Values: 3 11 12
Values: 10 11 12
```

Обратите внимание, предоставить аргумент для параметра `c`, не предоставляя при этом аргументы для параметров `a` и `b` - нельзя (перепрыгивать через параметры не разрешается). Это связано с тем, что язык C++ не поддерживает следующий синтаксис вызова функции: `printValues(,,5)`. Из этого вытекают следующие два правила:

Правило №1: **Все параметры по умолчанию в прототипе или в определении функции должны находиться справа.** Следующее вызовет ошибку:

```
1. void printValue(int a=5, int b); // не разрешается
```

Правильно:

```
1. void printValue(int a, int b=5);
```

Правило №2: **Если имеется более одного параметра по умолчанию, то самым левым параметром по умолчанию должен быть тот, который с наибольшей вероятностью (среди всех остальных параметров) будет явно переопределен пользователем**.

Объявление параметров по умолчанию

Как только параметр по умолчанию объявлен, повторно объявить его уже нельзя. Это значит, что для функции с предварительным объявлением и определением,

параметр по умолчанию объявить можно либо в предварительном объявлении, либо в определении функции, но не в обоих местах сразу. Например:

```
1. void printValues(int a, int b=15);
2.
3. void printValues(int a, int b=15) // ошибка: переопределение параметра по
   умолчанию
4. {
5.     std::cout << "a: " << a << '\n';
6.     std::cout << "b: " << b << '\n';
7. }
```

Рекомендуется объявлять параметры по умолчанию в предварительном объявлении, а не в определении функции, так как предварительные объявления можно использовать в нескольких файлах — при таком раскладе параметры по умолчанию будет легче увидеть (особенно, если предварительное объявление находится в заголовочном файле). Например:

boo.h:

```
1. #ifndef BOO_H
2. #define BOO_H
3. void printValues(int a, int b=15);
4. #endif
```

main.cpp:

```
1. #include "boo.h"
2. #include <iostream>
3.
4. void printValues(int a, int b)
5. {
6.     std::cout << "a: " << a << '\n';
7.     std::cout << "b: " << b << '\n';
8. }
9.
10. int main()
11. {
12.     printValues(7);
13.
14.     return 0;
15. }
```

Обратите внимание, в примере, приведенном выше, используется параметр по умолчанию `b` для функции printValues(), так как main.cpp подключает boo.h, который имеет предварительное объявление функции printValues() с объявленным параметром по умолчанию.

Правило: Объявляйте параметры по умолчанию в предварительном объявлении функции, в противном случае (если функция не имеет предварительного объявления) — объявляйте в определении функции.

Параметры по умолчанию и перегрузка функций

Функции с параметрами по умолчанию могут быть перегружены. Например:

```
1. void print(std::string string);
2. void print(char ch=' ');
```

Если пользователь вызовет просто `print()` (без параметров), то выведется пробел, что будет результатом выполнения `print(' ')`.

Однако, стоит отметить, что параметры по умолчанию НЕ относятся к параметрам, которые учитываются при определении уникальности функции. Следовательно, следующее не допускается:

```
1. void printValues(int a);
2. void printValues(int a, int b=15);
```

При вызове `printValues(10)` компилятор не сможет определить, хотите ли вы вызвать `printValues(int)` или `printValues(int, 15)` (со значением по умолчанию).

Заключение

Параметры по умолчанию - это полезный механизм указания параметров, при котором пользователь может переопределять значения по умолчанию, либо не переопределять их вообще. Они часто используются в языке C++, и их применение вы увидите уже на следующих уроках.

Урок №110. Указатели на функции

Из предыдущих уроков мы узнали, что указатель - это переменная, которая содержит адрес другой переменной. Указатели на функции аналогичны, за исключением того, что вместо обычных переменных, они указывают на функции!

Указатели на функции

Рассмотрим следующий фрагмент кода:

```
1. int boo()
2. {
3.     return 7;
4. }
```

Идентификатор `boo` - это имя функции. Но какой её тип? Функции имеют свой собственный l-value тип. В этом случае это тип функции, который возвращает целочисленное значение и не принимает никаких параметров. Подобно переменным, функции также имеют свой адрес в памяти.

Когда функция вызывается (с помощью оператора `()`), точка выполнения переходит к адресу вызываемой функции:

```
1. int boo() // код функции boo() находится в ячейке памяти 002B1050
2. {
3.     return 7;
4. }
5.
6. int main()
7. {
8.     boo(); // переходим к адресу 002B1050
9.
10.    return 0;
11. }
```

Одной из распространенных ошибок новичков является:

```
1. #include <iostream>
2.
3. int boo() // код функции boo() находится в ячейке памяти 002B1050
4. {
5.     return 7;
6. }
7.
8. int main()
9. {
10.    std::cout << boo; // мы хотим вызвать boo(), но вместо этого мы просто
   выводим boo!
11.
12.    return 0;
13. }
```

Вместо вызова функции boo() и вывода возвращаемого значения мы, совершенно случайно, отправили указатель на функцию boo() непосредственно в std::cout. Что произойдет в этом случае?

Результат на моем компьютере:

```
002B1050
```

У вас может быть и другое значение, в зависимости от того, в какой тип данных ваш компилятор решит конвертировать указатель на функцию. Если ваш компьютер не вывел адрес функции, то вы можете заставить его это сделать, конвертируя `boo` в указатель типа void и отправляя его на вывод:

```
1. #include <iostream>
2.
3. int boo() // код функции boo() находится в ячейке памяти 002B1050
4. {
5.     return 7;
6. }
7.
8. int main()
9. {
10.     std::cout << reinterpret_cast<void*>(boo); // указываем C++ конвертировать функцию boo() в указатель типа void
11.
12.     return 0;
13. }
```

Так же, как можно объявить неконстантный указатель на обычную переменную, можно объявить и неконстантный указатель на функцию. Синтаксис создания неконстантного указателя на функцию, пожалуй, один из самых "уродливых" в языке C++:

```
1. // fcnPtr - это указатель на функцию, которая не принимает никаких аргументов и возвращает целочисленное значение
2. int (*fcnPtr)();
```

В примере, приведенном выше, `fcnPtr` - это указатель на функцию, которая не имеет параметров и возвращает целочисленное значение. `fcnPtr` может указывать на любую другую функцию, соответствующую этому типу.

Скобки вокруг `*fcnPtr` необходимы для соблюдения приоритета операций, в противном случае `int *fcnPtr()` будет интерпретироваться как предварительное объявление функции `fcnPtr`, которая не имеет параметров и возвращает указатель на целочисленное значение.

Для создания константного указателя на функцию используйте const после звёздочки:

```
int (*const fcnPtr)();
```

Если вы поместите const перед int, это будет означать, что функция, на которую указывает указатель, возвращает const int.

Присваивание функции указателю на функцию

Указатель на функцию может быть инициализирован функцией (и неконстантному указателю на функцию тоже можно присвоить функцию):

```
int boo()
{
    return 7;
}

int doo()
{
    return 8;
}

int main()
{
    int (*fcnPtr)() = boo; // fcnPtr указывает на функцию boo()
    fcnPtr = doo; // fcnPtr теперь указывает на функцию doo()

    return 0;
}
```

Одна из распространенных ошибок, которую совершают новички:

```
fcnPtr = doo();
```

Здесь мы фактически присваиваем возвращаемое значение из вызова функции doo() указателю `fcnPtr`, чего мы не хотим делать. Мы хотим, чтобы `fcnPtr` содержал адрес функции doo(), а не возвращаемое значение из doo(). Поэтому скобки здесь не нужны.

Обратите внимание, у указателя на функцию и самой функции должны совпадать тип, параметры и тип возвращаемого значения. Например:

```
// Прототипы функций
int boo();
double doo();
int moo(int a);

// Присваивание значений указателям на функции
int (*fcnPtr1)() = boo; // ок
int (*fcnPtr2)() = doo; // не ок: тип указателя и тип возврата функции
не совпадают!
```

```
9. double (*fcnPtr4)() = doo; // ок
10. fcnPtr1 = moo; // не ок: fcnPtr1 не имеет параметров, но moo() имеет
11. int (*fcnPtr3)(int) = moo; // ок
```

В отличие от фундаментальных типов данных, язык C++ неявно конвертирует функцию в указатель на функцию, если это необходимо (поэтому вам не нужно использовать оператор адреса `&` для получения адреса функции). Однако, язык C++ не будет неявно конвертировать указатель на функцию в указатель типа void или наоборот.

Вызов функции через указатель на функцию

Вы также можете использовать указатель на функцию для вызова самой функции. Есть два способа сделать это. Первый - через явное разыменование:

```
1. int boo(int a)
2. {
3.     return a;
4. }
5.
6. int main()
7. {
8.     int (*fcnPtr)(int) = boo; // присваиваем функцию boo() указателю fcnPtr
9.     (*fcnPtr)(7); // вызываем функцию boo(7), используя fcnPtr
10.
11.    return 0;
12. }
```

Второй - через неявное разыменование:

```
1. int boo(int a)
2. {
3.     return a;
4. }
5.
6. int main()
7. {
8.     int (*fcnPtr)(int) = boo; // присваиваем функцию boo() указателю fcnPtr
9.     fcnPtr(7); // вызываем функцию boo(7), используя fcnPtr
10.
11.    return 0;
12. }
```

Как вы можете видеть, способ неявного разыменования выглядит так же, как и обычный вызов функции, так как обычные имена функций являются указателями на функции!

> ***Примечание***: Параметры по умолчанию не будут работать с функциями, вызванными через указатели на функции. Параметры по умолчанию обрабатываются во время компиляции (т.е. вам нужно предоставить аргумент для параметра по умолчанию во время компиляции). Однако указатели на функции

> обрабатываются во время выполнения. Следовательно, параметры по умолчанию не могут обрабатываться при вызове функции через указатель на функцию. В этом случае вам нужно будет явно передать значения для параметров по умолчанию.

Передача функций в качестве аргументов другим функциям

Одна из самых полезных вещей, которую вы можете сделать с указателями на функции — это передать функцию в качестве аргумента другой функции. Функции, используемые в качестве аргументов для других функций, называются **функциями обратного вызова**.

Предположим, что вы пишете функцию для выполнения определенного задания (например, сортировки массива), но вы хотите, чтобы пользователь мог определить, каким образом выполнять эту сортировку (например, по возрастанию или по убыванию). Рассмотрим более подробно этот случай.

Все алгоритмы сортировки работают по одинаковой схеме: алгоритм выполняет итерацию по списку чисел, сравнивает пары чисел и меняет их местами, исходя из результатов этих сравнений. Следовательно, изменяя алгоритм сравнения чисел, мы можем изменить способ сортировки, не затрагивая остальные части кода.

Вот наша сортировка методом выбора, рассмотренная на соответствующем уроке:

```
1. #include <algorithm> // для std::swap() (используйте <utility>, если поддерживается C++11)
2. 
3. void SelectionSort(int *array, int size)
4. {
5.     // Перебираем каждый элемент массива
6.     for (int startIndex = 0; startIndex < size; ++startIndex)
7.     {
8.         // smallestIndex - это индекс наименьшего элемента, который мы обнаружили до этого момента
9.         int smallestIndex = startIndex;
10.
11.        // Ищем наименьший элемент среди оставшихся в массиве (начинаем со startIndex+1)
12.        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
13.        {
14.            // Если текущий элемент меньше нашего предыдущего найденного наименьшего элемента,
15.            if (array[smallestIndex] > array[currentIndex]) // СРАВНЕНИЕ ВЫПОЛНЯЕТСЯ ЗДЕСЬ
16.                // то это наш новый наименьший элемент в этой итерации
17.                smallestIndex = currentIndex;
18.        }
19.
20.        // Меняем местами наш стартовый элемент с найденным наименьшим элементом
21.        std::swap(array[startIndex], array[smallestIndex]);
```

```
    }
}
```

Давайте заменим сравнение чисел на функцию сравнения. Поскольку наша функция сравнения будет сравнивать два целых числа и возвращать логическое значение для указания того, следует ли выполнять замену, то она будет выглядеть следующим образом:

```
bool ascending(int a, int b)
{
    return a > b; // условие, при котором меняются местами элементы массива
}
```

А вот уже сортировка методом выбора с функцией ascending() для сравнения чисел:

```
#include <algorithm> // для std::swap() (используйте <utility>, если поддерживается C++11)

void SelectionSort(int *array, int size)
{
    // Перебираем каждый элемент массива
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // smallestIndex - это индекс наименьшего элемента, который мы обнаружили до этого момента
        int smallestIndex = startIndex;

        // Ищем наименьший элемент среди оставшихся в массиве (начинаем со startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // Если текущий элемент меньше нашего предыдущего найденного наименьшего элемента,
            if (ascending(array[smallestIndex], array[currentIndex])) // СРАВНЕНИЕ ВЫПОЛНЯЕТСЯ ЗДЕСЬ
                // то это наш новый наименьший элемент в этой итерации
                smallestIndex = currentIndex;
        }

        // Меняем местами наш стартовый элемент с найденным наименьшим элементом
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

Теперь, чтобы позволить caller-у решить, каким образом будет выполняться сортировка, вместо использования нашей функции сравнения, мы разрешаем caller-у предоставить свою собственную функцию сравнения! Это делается с помощью указателя на функцию.

Поскольку функция сравнения caller-а будет сравнивать два целых числа и возвращать логическое значение, то указатель на эту функцию будет выглядеть следующим образом:

```
bool (*comparisonFcn)(int, int);
```

Мы разрешаем caller-у передавать способ сортировки массива с помощью указателя на функцию в качестве третьего параметра в нашу функцию сортировки.

Вот готовый код сортировки методом выбора с выбором способа сортировки в caller-е (т.е. в функции main()):

```
#include <iostream>
#include <algorithm> // для std::swap() (используйте <utility>, если поддерживается C++11)

// Обратите внимание, третьим параметром является пользовательский выбор выполнения сортировки
void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
{
    // Перебираем каждый элемент массива
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // bestIndex - это индекс наименьшего/наибольшего элемента, который мы обнаружили до этого момента
        int bestIndex = startIndex;

        // Ищем наименьший/наибольший элемент среди оставшихся в массиве (начинаем со startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // Если текущий элемент меньше/больше нашего предыдущего найденного наименьшего/наибольшего элемента,
            if (comparisonFcn(array[bestIndex], array[currentIndex])) // СРАВНЕНИЕ ВЫПОЛНЯЕТСЯ ЗДЕСЬ
                // то это наш новый наименьший/наибольший элемент в этой итерации
                bestIndex = currentIndex;
        }

        // Меняем местами наш стартовый элемент с найденным наименьшим/наибольшим элементом
        std::swap(array[startIndex], array[bestIndex]);
    }
}

// Вот функция сравнения, которая выполняет сортировку в порядке возрастания (обратите внимание, это та же функция ascending(), что и в примере, приведенном выше)
bool ascending(int a, int b)
{
    return a > b; // меняем местами, если первый элемент больше второго
}

// Вот функция сравнения, которая выполняет сортировку в порядке убывания
bool descending(int a, int b)
```

```
{
    return a < b; // меняем местами, если второй элемент больше первого
}

// Эта функция выводит значения массива
void printArray(int *array, int size)
{
    for (int index=0; index < size; ++index)
        std::cout << array[index] << " ";
    std::cout << '\n';
}

int main()
{
    int array[8] = { 4, 8, 5, 6, 2, 3, 1, 7 };

    // Сортируем массив в порядке убывания, используя функцию descending()
    selectionSort(array, 8, descending);
    printArray(array, 8);

    // Сортируем массив в порядке возрастания, используя функцию ascending()
    selectionSort(array, 8, ascending);
    printArray(array, 8);

    return 0;
}
```

Результат выполнения программы:

```
8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8
```

Прикольно, правда? Мы предоставили caller-у возможность контролировать процесс сортировки чисел (caller может определить любые другие функции сравнения):

```
bool evensFirst(int a, int b)
{
    // Если a - чётное число, а b - нечётное число, то a идет первым (никакого обмена местами не требуется)
    if ((a % 2 == 0) && !(b % 2 == 0))
        return false;

    // Если a - нечётное число, а b - чётное число, то b идет первым (здесь уже требуется обмен местами)
    if (!(a % 2 == 0) && (b % 2 == 0))
        return true;

    // В противном случае, сортируем в порядке возрастания
    return ascending(a, b);
}

int main()
{
    int array[8] = { 4, 8, 6, 3, 1, 2, 5, 7 };

    selectionSort(array, 8, evensFirst);
    printArray(array, 8);
```

```
21.
22.     return 0;
23. }
```

Результат выполнения программы:

```
2 4 6 8 1 3 5 7
```

Как вы можете видеть, использование указателя на функцию позволяет caller-у «подключить» свой собственный функционал к чему-то, что мы писали и тестировали ранее, что способствует повторному использованию кода! Раньше, если вы хотели отсортировать один массив в порядке убывания, а другой — в порядке возрастания, вам понадобилось бы написать несколько версий сортировки массива. Теперь же у вас может быть одна версия, которая будет выполнять сортировку любым способом, каким вы только захотите!

Параметры по умолчанию в функциях

Если вы позволите caller-у передавать функцию в качестве параметра, то полезным будет предоставить и некоторые стандартные функции для удобства caller-а. Например, в вышеприведенном примере с сортировкой методом выбора, было бы проще установить дефолтный (по умолчанию) способ сравнения чисел. Например:

```
1. // Сортировка по умолчанию выполняется в порядке возрастания
2. void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int) =
   ascending);
```

В этом случае, до тех пор, пока пользователь вызывает selectionSort() обычно (а не через указатель на функцию), параметр `comparisonFcn` будет по умолчанию соответствовать функции ascending().

Указатели на функции и псевдонимы типов

Посмотрим правде в глаза - синтаксис указателей на функции уродлив. Тем не менее, с помощью typedefs мы можем исправить эту ситуацию:

```
1. typedef bool (*validateFcn)(int, int);
```

Здесь мы определили псевдоним типа под названием `validateFcn`, который является указателем на функцию, которая принимает два значения типа int и возвращает значение типа bool.

Теперь вместо написания следующего:

```
1. bool validate(int a, int b, bool (*fcnPtr)(int, int)); // фу, какой синтаксис
```

Мы можем написать следующее:

```
1. bool validate(int a, int b, validateFcn pfcn) // вот это другое дело
```

Так гораздо лучше, не правда ли? Однако синтаксис определения самого typedef может быть несколько трудным для запоминания. В C++11 вместо typedef вы можете использовать type alias для создания псевдонима типа указателя на функцию:

```
1. using validateFcn = bool(*)(int, int); // type alias
```

Это уже читабельнее, чем с typedef, так как имя псевдонима и его определение расположены на противоположных сторонах от оператора `=`.

Использование type alias идентично использованию typedef:

```
1. bool validate(int a, int b, validateFcn pfcn) // круто, не так ли?
```

Использование std::function в C++11

В C++11 ввели альтернативный способ определения и хранения указателей на функции, который выполняется с использованием **std::function**. std::function является частью заголовочного файла functional Стандартной библиотеки C++. Для определения указателя на функцию с помощью этого способа вам нужно объявить объект std::function следующим образом:

```
1. #include <functional>
2.
3. bool validate(int a, int b, std::function<bool(int, int)> fcn); // указываем
   указатель на функцию с помощью std::function, которая возвращает bool и
   принимает два int-а
```

Как вы можете видеть, тип возврата и параметры находятся в угловых скобках, а параметры еще и внутри круглых скобок. Если параметров нет, то внутренние скобки можно оставить пустыми. Здесь уже более понятно, какой тип возвращаемого значения и какие ожидаемые параметры функции.

Обновим наш предыдущий пример из раздела «Присваивание функции указателю на функцию» текущего урока, но уже с использованием std::function:

```
1. #include <iostream>
2. #include <functional>
3.
4. int boo()
5. {
6.     return 7;
7. }
8.
```

```
int doo()
{
    return 8;
}

int main()
{
    std::function<int()> fcnPtr; // объявляем указатель на функцию, который возвращает int и не принимает никаких параметров
    fcnPtr = doo; // fcnPtr теперь указывает на функцию doo()
    std::cout << fcnPtr(); // вызываем функцию как обычно

    return 0;
}
```

Заключение

Указатели на функции полезны, прежде всего, когда вы хотите хранить функции в массиве (или в структуре) или когда вам нужно передать одну функцию в качестве аргумента другой функции. Поскольку синтаксис объявления указателей на функции является несколько уродливым и подвержен ошибкам, то рекомендуется использовать type alias (или std::function в C++11).

Тест

Задание №1

В этот раз мы попытаемся написать версию базового калькулятора с помощью указателей на функции.

a) Напишите короткую программу, которая просит пользователя ввести два целых числа и выбрать математическую операцию: `+`, `-`, `*` или `/`. Убедитесь, что пользователь ввел корректный символ математической операции (используйте проверку).

b) Напишите функции add(), subtract(), multiply() и divide(). Они должны принимать два целочисленных параметра и возвращать целочисленное значение.

c) Создайте typedef с именем `arithmeticFcn` для указателя на функцию, которая принимает два целочисленных параметра и возвращает целочисленное значение.

d) Напишите функцию с именем getArithmeticFcn(), которая принимает символ выбранного математического оператора и возвращает соответствующую функцию в качестве указателя на функцию.

e) Добавьте в функцию main() вызов функции getArithmeticFcn().

f) Соедините все части вместе.

Задание №2

Теперь давайте изменим программу, которую мы написали в 1-м задании, чтобы переместить логику из getArithmeticFcn в массив.

a) Создайте структуру с именем `arithmeticStruct`, которая имеет два члена: математический оператор типа char и указатель на функцию `arithmeticFcn`.

b) Создайте статический глобальный массив `arithmeticArray`, используя структуру `arithmeticStruct`, который будет инициализирован каждой из 4-х математических операций.

c) Измените `getArithmeticFcn` для выполнения цикла по массиву и возврата соответствующего указателя на функцию.

> ***Подсказка*:** Используйте цикл foreach.

d) Соедините все части вместе.

Урок №111. Стек и Куча

Память, которую используют программы, состоит из нескольких частей - **сегментов**:

- **Сегмент кода** (или *«текстовый сегмент»*), где находится скомпилированная программа. Обычно доступен только для чтения.
- **Сегмент bss** (или *«неинициализированный сегмент данных»*), где хранятся глобальные и статические переменные, инициализированные нулем.
- **Сегмент данных** (или *«сегмент инициализированных данных»*), где хранятся инициализированные глобальные и статические переменные.
- **Куча**, откуда выделяются динамические переменные.
- **Стек вызовов**, где хранятся параметры функции, локальные переменные и другая информация, связанная с функциями.

Куча

Сегмент кучи (или просто ***«куча»***) отслеживает память, используемую для динамического выделения. Мы уже немного поговорили о куче на уроке о динамическом выделении памяти.

В языке C++ при использовании оператора new динамическая память выделяется из сегмента кучи самой программы:

```
1. int *ptr = new int; // для ptr выделяется 4 байта из кучи
2. int *array = new int[10]; // для array выделяется 40 байт из кучи
```

Адрес выделяемой памяти передается обратно оператором new и затем он может быть сохранен в указателе. О механизме хранения и выделения свободной памяти нам сейчас беспокоиться незачем. Однако стоит знать, что последовательные запросы памяти не всегда приводят к выделению последовательных адресов памяти!

```
1. int *ptr1 = new int;
2. int *ptr2 = new int;
3. // ptr1 и ptr2 могут не иметь последовательных адресов памяти
```

При удалении динамически выделенной переменной, память возвращается обратно в кучу и затем может быть переназначена (исходя из последующих запросов). Помните, что удаление указателя не удаляет переменную, а просто приводит к возврату памяти по этому адресу обратно в операционную систему.

Куча имеет свои преимущества и недостатки:

- Выделение памяти в куче сравнительно медленное.
- Выделенная память остается выделенной до тех пор, пока не будет освобождена (остерегайтесь утечек памяти) или пока программа не завершит свое выполнение.
- Доступ к динамически выделенной памяти осуществляется только через указатель. Разыменование указателя происходит медленнее, чем доступ к переменной напрямую.
- Поскольку куча представляет собой большой резервуар памяти, то именно она используется для выделения больших массивов, структур или классов.

Стек вызовов

Стек вызовов (или просто ***«стек»***) отслеживает все активные функции (те, которые были вызваны, но еще не завершены) от начала программы и до текущей точки выполнения, и обрабатывает выделение всех параметров функции и локальных переменных.

Стек вызовов реализуется как структура данных «Стек». Поэтому, прежде чем мы поговорим о том, как работает стек вызовов, нам нужно понять, что такое стек как структура данных.

Стек как структура данных

Структура данных в программировании — это механизм организации данных для их эффективного использования. Вы уже видели несколько типов структур данных, например, массивы или структуры. Существует множество других структур данных, которые используются в программировании. Некоторые из них реализованы в Стандартной библиотеке C++, и стек как раз является одним из таковых.

Например, рассмотрим стопку (аналогия стеку) тарелок на столе. Поскольку каждая тарелка тяжелая, а они еще и сложены друг на друге, то вы можете сделать лишь что-то одно из следующего:

- Посмотреть на поверхность первой тарелки (которая находится на самом верху).
- Взять верхнюю тарелку из стопки (обнажая таким образом следующую тарелку, которая находится под верхней, если она вообще существует).
- Положить новую тарелку поверх стопки (спрятав под ней самую верхнюю тарелку, если она вообще была).

В компьютерном программировании стек представляет собой контейнер (как структуру данных), который содержит несколько переменных (подобно массиву). Однако, в то время как массив позволяет получить доступ и изменять элементы в любом порядке (так называемый ***«произвольный доступ»***), стек более ограничен.

В стеке вы можете:

- Посмотреть на верхний элемент стека (используя функцию `top()` или `peek()`).
- Вытянуть верхний элемент стека (используя функцию `pop()`).
- Добавить новый элемент поверх стека (используя функцию `push()`).

Стек — это структура данных типа **LIFO** (англ. *"**L**ast **I**n, **F**irst **O**ut" = "Последним пришел, первым ушел"*). Последний элемент, который находится на вершине стека, первым и уйдет из него. Если положить новую тарелку поверх других тарелок, то именно эту тарелку вы первой и возьмете. По мере того, как элементы помещаются в стек — стек растет, по мере того, как элементы удаляются из стека — стек уменьшается.

Например, рассмотрим короткую последовательность, показывающую, как работает добавление и удаление в стеке:

```
Stack: empty
Push 1
Stack: 1
Push 2
Stack: 1 2
Push 3
Stack: 1 2 3
Push 4
Stack: 1 2 3 4
Pop
Stack: 1 2 3
Pop
Stack: 1 2
Pop
Stack: 1
```

Стопка тарелок довольно-таки хорошая аналогия работы стека, но есть лучшая аналогия. Например, рассмотрим несколько почтовых ящиков, которые расположены друг на друге. Каждый почтовый ящик может содержать только один элемент, и все почтовые ящики изначально пустые. Кроме того, каждый почтовый ящик прибивается гвоздем к почтовому ящику снизу, поэтому количество почтовых

ящиков не может быть изменено. Если мы не можем изменить количество почтовых ящиков, то как мы получим поведение, подобное стеку?

Во-первых, мы используем наклейку для обозначения того, где находится самый нижний пустой почтовый ящик. Вначале это будет первый почтовый ящик, который находится на полу. Когда мы добавим элемент в наш стек почтовых ящиков, то мы поместим этот элемент в почтовый ящик, на котором будет наклейка (т.е. в самый первый пустой почтовый ящик на полу), а затем переместим наклейку на один почтовый ящик выше. Когда мы вытаскиваем элемент из стека, то мы перемещаем наклейку на один почтовый ящик ниже и удаляем элемент из почтового ящика. Всё, что находится ниже наклейки - находится в стеке. Всё, что находится в ящике с наклейкой и выше — находится вне стека.

Сегмент стека вызовов

Сегмент стека вызовов содержит память, используемую для стека вызовов. При запуске программы, функция main() помещается в стек вызовов операционной системой. Затем программа начинает свое выполнение.

Когда программа встречает вызов функции, то эта функция помещается в стек вызовов. При завершении выполнения функции, она удаляется из стека вызовов. Таким образом, просматривая функции, добавленные в стек, мы можем видеть все функции, которые были вызваны до текущей точки выполнения.

Наша аналогия с почтовыми ящиками — это действительно то, как работает стек вызовов. Стек вызовов имеет фиксированное количество адресов памяти (фиксированный размер). Почтовые ящики являются адресами памяти, а «элементы», которые мы добавляем или вытягиваем из стека, называются **фреймами** (или ***«кадрами»***) **стека**. Кадр стека отслеживает все данные, связанные с одним вызовом функции. «Наклейка» - это регистр (небольшая часть памяти в ЦП), который является указателем стека. **Указатель стека** отслеживает вершину стека вызовов.

Единственное отличие фактического стека вызовов от нашего гипотетического стека почтовых ящиков заключается в том, что, когда мы вытягиваем элемент из стека вызовов, нам не нужно очищать память (т.е. вынимать всё содержимое из почтового ящика). Мы можем просто оставить эту память для следующего элемента, который и перезапишет её. Поскольку указатель стека будет ниже этого адреса памяти, то, как мы уже знаем, эта ячейка памяти не будет находиться в стеке.

Стек вызовов на практике

Давайте рассмотрим детально, как работает стек вызовов. Ниже приведена **последовательность шагов, выполняемых при вызове функции**:

- Программа сталкивается с вызовом функции.
- Создается фрейм стека, который помещается в стек. Он состоит из:
  - адреса инструкции, который находится за вызовом функции (так называемый ***«обратный адрес»***). Так процессор запоминает, куда ему возвращаться после выполнения функции;
  - аргументов функции;
  - памяти для локальных переменных;
  - сохраненных копий всех регистров, модифицированных функцией, которые необходимо будет восстановить после того, как функция завершит свое выполнение.
- Процессор переходит к точке начала выполнения функции.
- Инструкции внутри функции начинают выполняться.

После завершения функции, выполняются следующие шаги:

- Регистры восстанавливаются из стека вызовов.
- Фрейм стека вытягивается из стека. Освобождается память, которая была выделена для всех локальных переменных и аргументов.
- Обрабатывается возвращаемое значение.
- ЦП возобновляет выполнение кода (исходя из обратного адреса).

Возвращаемые значения могут обрабатываться разными способами, в зависимости от архитектуры компьютера. Некоторые архитектуры считают возвращаемое значение частью фрейма стека, другие используют регистры процессора.

Знать все детали работы стека вызовов не так уж и важно. Однако понимание того, что функции при вызове добавляются в стек, а при завершении выполнения — удаляются из стека, дает основы, необходимые для понимания рекурсии, а также некоторых других концепций, которые полезны при отладке программ.

Пример стека вызовов

Рассмотрим следующий фрагмент кода:

```
1. int boo(int b)
2. {
3.     // b
```

```
4.      return b;
5.  } // функция boo() вытягивается из стека вызовов здесь
6.
7.  int main()
8.  {
9.      // a
10.     boo(7); // функция boo() добавляется в стек вызовов здесь
11.     // c
12.
13.     return 0;
14. }
```

Стек вызовов этой программы выглядит следующим образом:

a:

`main()`

b:

`boo()` (включая параметр `b`)
`main()`

c:

`main()`

Переполнение стека

Стек имеет ограниченный размер и, следовательно, может содержать только ограниченный объем информации. В операционной системе Windows размер стека по умолчанию составляет 1МБ. На некоторых Unix-системах этот размер может достигать и 8МБ. Если программа пытается поместить в стек слишком много информации, то это приведет к переполнению стека. **Переполнение стека** (англ. ***"stack overflow"***) происходит, когда запрашиваемой памяти нет в наличии (вся память уже занята).

Переполнение стека является результатом добавления слишком большого количества переменных в стек и/или создания слишком большого количества вложенных вызовов функций (например, когда функция A() вызывает функцию B(), которая вызывает функцию C(), а та, в свою очередь, вызывает функцию D() и т.д.). Переполнение стека обычно приводит к сбою в программе, например:

```
1.  int main()
2.  {
3.      int stack[1000000000];
4.      return 0;
5.  }
```

Эта программа пытается добавить огромный массив в стек вызовов. Поскольку размера стека недостаточно для обработки такого массива, то операция его добавления переходит и на другие части памяти, которые программа использовать не может. Следовательно, получаем сбой.

Вот еще одна программа, которая вызовет переполнение стека, но уже по другой причине:

```
1. void boo()
2. {
3.     boo();
4. }
5.
6. int main()
7. {
8.     boo();
9.
10.    return 0;
11. }
```

В программе, приведенной выше, фрейм стека добавляется в стек каждый раз, когда вызывается функция boo(). Поскольку функция boo() вызывает сама себя бесконечное количество раз, то в конечном итоге в стеке не хватит памяти, что приведет к переполнению стека.

Стек имеет свои преимущества и недостатки:

- Выделение памяти в стеке происходит сравнительно быстро.
- Память, выделенная в стеке, остается в области видимости до тех пор, пока находится в стеке. Она уничтожается при выходе из стека.
- Вся память, выделенная в стеке, обрабатывается во время компиляции, следовательно, доступ к этой памяти осуществляется напрямую через переменные.
- Поскольку размер стека является относительно небольшим, то не рекомендуется делать что-либо, что съест много памяти стека (например, передача по значению или создание локальных переменных больших массивов или других *затратных* структур данных).

Урок №112. Ёмкость вектора

Мы уже знаем, что такое std::vector в языке C++ и как его можно использовать в качестве динамического массива, который запоминает свою длину и длина которого может быть динамически изменена по мере необходимости. Хотя использование std::vector в качестве динамического массива — это самая полезная и наиболее часто применяемая его особенность, но он также имеет и некоторые другие способности, которые также могут быть полезными.

Длина vs. Ёмкость

Рассмотрим следующий пример:

```
1. int *array = new int[12] { 1, 2, 3, 4, 5, 6, 7 };
```

Мы можем сказать, что длина массива равна 12, но используется только 7 элементов (которые мы, собственно, выделили).

А что, если мы хотим выполнять итерации только с элементами, которые мы инициализировали, оставляя в резерве неиспользованные элементы для будущего применения? В таком случае нам потребуется отдельно отслеживать, сколько элементов было «использовано» из общего количества выделенных элементов. В отличие от фиксированного массива или std::array, которые запоминают только свою длину, std::vector имеет два отдельных свойства:

- **Длина в std::vector** — это количество фактически используемых элементов.
- **Ёмкость** (или ***"вместимость"***) **в std::vector** — это количество выделенных элементов.

Рассмотрим пример из урока о std::vector:

```
1. #include <iostream>
2. #include <vector>
3.
4. int main()
5. {
6.     std::vector<int> array { 0, 1, 2, 3 };
7.     array.resize(6); // устанавливаем длину, равную 6
8.
9.     std::cout << "The length is: " << array.size() << '\n';
10.
11.    for (auto const &element: array)
12.        std::cout << element << ' ';
13.
14.    return 0;
15. }
```

Результат выполнения программы:

```
The length is: 6
0 1 2 3 0 0
```

В примере, приведенном выше, мы использовали функцию resize() для изменения длины вектора до 6 элементов. Это сообщает массиву, что мы намереваемся использовать только первые 6 элементов, поэтому он должен их учитывать, как активные (те, которые фактически используются). Следует вопрос: "Какова ёмкость этого массива?".

Мы можем спросить std::vector о его ёмкости, используя **функцию capacity()**:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> array { 0, 1, 2, 3};
    array.resize(6); // устанавливаем длину, равную 6

    std::cout << "The length is: " << array.size() << '\n';
    std::cout << "The capacity is: " << array.capacity() << '\n';
}
```

Результат на моем компьютере:

```
The length is: 6
The capacity is: 6
```

В этом случае функция resize() заставила std::vector изменить как свою длину, так и ёмкость. Обратите внимание, ёмкость всегда должна быть не меньше длины массива (но может быть и больше), иначе доступ к элементам в конце массива будет за пределами выделенной памяти!

Зачем вообще нужны длина и ёмкость? std::vector может перераспределить свою память, если это необходимо, но он бы предпочел этого не делать, так как изменение размера массива является несколько затратной операцией. Например:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> array;
    array = { 0, 1, 2, 3, 4, 5 }; // ок, длина array равна 6
    std::cout << "length: " << array.size() << "  capacity: " << array.capacity() << '\n';

    array = { 8, 7, 6, 5 }; // ок, длина array теперь равна 4!
```

```
11.     std::cout << "length: " << array.size() << "  capacity: " << array.capacity
    () << '\n';
12.
13.     return 0;
14. }
```

Результат выполнения программы:

```
length: 6 capacity: 6
length: 4 capacity: 6
```

Обратите внимание, хотя мы присвоили меньшее количество элементов массиву во второй раз - он не перераспределил свою память, ёмкость по-прежнему составляет 6 элементов. Он просто изменил свою длину. Таким образом, он понимает, что в настоящий момент активны только первые 4 элемента.

Оператор индекса и функция at()

Диапазон для оператора индекса `[]` и функции at() основан на длине вектора, а не на его ёмкости. Рассмотрим массив из вышеприведенного примера, длина которого равна 4, а ёмкость равна 6. Что произойдет, если мы попытаемся получить доступ к элементу массива под индексом 5? Ничего, поскольку индекс 5 находится за пределами длины массива.

Обратите внимание, вектор не будет изменять свой размер из-за вызова оператора индекса или функции at()!

std::vector в качестве стека

Если оператор индекса и функция at() основаны на длине массива, а его ёмкость всегда не меньше, чем его длина, то зачем беспокоиться о ёмкости вообще? Хотя std::vector может использоваться как динамический массив, его также можно использовать в качестве стека. Мы можем использовать **3 ключевые функции вектора**, которые соответствуют 3-м ключевым операциям стека:

- **функция push_back()** добавляет элемент в стек.
- **функция back()** возвращает значение верхнего элемента стека.
- **функция pop_back()** вытягивает элемент из стека.

Например:

```
1. #include <iostream>
2. #include <vector>
3.
4. void printStack(const std::vector<int> &stack)
5. {
```

```
    for (const auto &element : stack)
        std::cout << element << ' ';
    std::cout << "(cap " << stack.capacity() << " length " << stack.size() << " )\n";
}

int main()
{
    std::vector<int> stack;

    printStack(stack);

    stack.push_back(7); // функция push_back() добавляет элемент в стек
    printStack(stack);

    stack.push_back(4);
    printStack(stack);

    stack.push_back(1);
    printStack(stack);

    std::cout << "top: " << stack.back() << '\n'; // функция back() возвращает последний элемент

    stack.pop_back(); // функция pop_back() вытягивает элемент из стека
    printStack(stack);

    stack.pop_back();
    printStack(stack);

    stack.pop_back();
    printStack(stack);

    return 0;
}
```

Результат выполнения программы:

```
(cap 0 length 0)
7 (cap 1 length 1)
7 4 (cap 2 length 2)
7 4 1 (cap 3 length 3)
top: 1
7 4 (cap 3 length 2)
7 (cap 3 length 1)
(cap 3 length 0)
```

В отличие от оператора индекса и функции at(), функции вектора-стека *изменяют* размер std::vector (выполняется функция resize()), если это необходимо. В примере, приведенном выше, вектор изменяет свой размер 3 раза (3 раза выполняется функция resize(): от ёмкости 0 до ёмкости 1, от 1 до 2 и от 2 до 3).

Поскольку изменение размера вектора является затратной операцией, то мы можем сообщить вектору выделить заранее заданный объем ёмкости, используя **функцию reserve()**:

```
#include <iostream>
#include <vector>

void printStack(const std::vector<int> &stack)
{
    for (const auto &element : stack)
        std::cout << element << ' ';
    std::cout << "(cap " << stack.capacity() << " length " << stack.size() << " )\n";
}

int main()
{
    std::vector<int> stack;

    stack.reserve(7); // устанавливаем ёмкость (как минимум), равную 7

    printStack(stack);

    stack.push_back(7);
    printStack(stack);

    stack.push_back(4);
    printStack(stack);

    stack.push_back(1);
    printStack(stack);

    std::cout << "top: " << stack.back() << '\n';

    stack.pop_back();
    printStack(stack);

    stack.pop_back();
    printStack(stack);

    stack.pop_back();
    printStack(stack);

    return 0;
}
```

Результат выполнения программы:

```
(cap 7 length 0)
7 (cap 7 length 1)
7 4 (cap 7 length 2)
7 4 1 (cap 7 length 3)
top: 1
7 4 (cap 7 length 2)
7 (cap 7 length 1)
(cap 7 length 0)
```

Ёмкость вектора была заранее предустановлена (значением 7) и не изменялась в течение всего времени выполнения программы.

Дополнительная ёмкость

При изменении вектором своего размера, он может выделить больше ёмкости, чем требуется. Это делается для обеспечения некоего резерва для дополнительных элементов, чтобы свести к минимуму количество операций изменения размера. Например:

```
1. #include <iostream>
2. #include <vector>
3.
4. int main()
5. {
6.     std::vector<int> vect = { 0, 1, 2, 3, 4, 5 };
7.     std::cout << "size: " << vect.size() << "  cap: " << vect.capacity() <<
   '\n';
8.
9.     vect.push_back(6); // добавляем другой элемент
10.    std::cout << "size: " << vect.size() << "  cap: " << vect.capacity() <<
   '\n';
11.
12.    return 0;
13.}
```

Результат на моем компьютере:

```
size: 6 cap: 6
size: 7 cap: 9
```

Когда мы использовали функцию push_back() для добавления нового элемента, то нашему вектору потребовалось выделить комнату только для 7 элементов, но он выделил комнату для 9 элементов. Это было сделано для того, чтобы при использовании функции push_back() в случае добавления еще одного элемента, вектору не пришлось опять выполнять операцию изменения своего размера (экономя, таким образом, ресурсы).

Как, когда и сколько выделяется дополнительной ёмкости — зависит от каждого компилятора отдельно.

Урок №113. Рекурсия и Числа Фибоначчи

Рекурсивная функция (или просто ***"рекурсия"***) в языке C++ - это функция, которая вызывает сама себя. Например:

```
1. #include <iostream>
2.
3. void countOut(int count)
4. {
5.     std::cout << "push " << count << '\n';
6.     countOut(count-1); // функция countOut() вызывает рекурсивно сама себя
7. }
8.
9. int main()
10.{
11.    countOut(4);
12.
13.    return 0;
14.}
```

При вызове функции countOut(4) на экран выведется `push 4`, а затем вызывается countOut(3). countOut(3) выведет `push 3` и вызывает countOut(2). Последовательность вызова countOut(n) других функций countOut(n-1) повторяется бесконечное количество раз (аналог бесконечного цикла). Попробуйте запустить у себя.

На уроке о стеке и куче мы узнали, что при каждом вызове функции, определенные данные помещаются в стек вызовов. Поскольку функция countOut() никогда ничего не возвращает (она просто снова вызывает countOut()), то данные этой функции никогда не вытягиваются из стека! Следовательно, в какой-то момент, память стека закончится и произойдет переполнение стека.

Условие завершения рекурсии

Рекурсивные вызовы функций работают точно так же, как и обычные вызовы функций. Однако, программа, приведенная выше, иллюстрирует наиболее важное отличие простых функций от рекурсивных: вы должны указать условие завершения рекурсии, в противном случае — функция будет выполняться «бесконечно» (фактически до тех пор, пока не закончится память в стеке вызовов).

Условие завершения рекурсии — это условие, которое, при его выполнении, остановит вызов рекурсивной функции самой себя. В этом условии обычно используется оператор if.

Вот пример функции, приведенной выше, но уже с условием завершения рекурсии (и еще с одним дополнительным выводом текста):

```
#include <iostream>

void countOut(int count)
{
    std::cout << "push " << count << '\n';

    if (count > 1) // условие завершения
        countOut(count-1);

    std::cout << "pop " << count << '\n';
}

int main()
{
    countOut(4);
    return 0;
}
```

Когда мы запустим эту программу, то countOut() начнет выводить:

```
push 4
push 3
push 2
push 1
```

Если сейчас посмотреть на стек вызовов, то увидим следующее:

```
countOut(1)
countOut(2)
countOut(3)
countOut(4)
main()
```

Из-за условия завершения, countOut(1) не вызовет countOut(0): условие if не выполнится, и поэтому выведется `pop 1` и countOut(1) завершит свое выполнение. На этом этапе countOut(1) вытягивается из стека, и управление возвращается к countOut(2). countOut(2) возобновляет выполнение в точке после вызова countOut(1), и поэтому выведется `pop 2`, а затем countOut(2) завершится. Рекурсивные вызовы функций countOut() постепенно вытягиваются из стека до тех пор, пока не будут удалены все экземпляры countOut().

Таким образом, результат выполнения программы, приведенной выше:

```
push 4
push 3
push 2
```

```
push 1
pop 1
pop 2
pop 3
pop 4
```

Стоит отметить, что `push` выводится в порядке убывания, а `pop` — в порядке возрастания. Дело в том, что `push` выводится до вызова рекурсивной функции, а `pop` выполняется (выводится) после вызова рекурсивной функции, когда все экземпляры countOut() вытягиваются из стека (это происходит в порядке, обратном тому, в котором эти экземпляры были введены в стек).

Теперь, когда мы обсудили основной механизм вызова рекурсивных функций, давайте взглянем на несколько другой тип рекурсии, который более распространен:

```
1. // Возвращаем сумму всех чисел между 1 и value
2. int sumCount(int value)
3. {
4.     if (value <= 0)
5.         return 0; // базовый случай (условие завершения)
6.     else if (value == 1)
7.         return 1; // базовый случай (условие завершения)
8.     else
9.         return sumCount(value - 1) + value; // рекурсивный вызов функции
10.}
```

Рассмотреть рекурсию с первого взгляда на код не так уж и легко. Лучшим вариантом будет посмотреть, что произойдет при вызове рекурсивной функции с определенным значением. Например, посмотрим, что произойдет при вызове вышеприведенной функции с `value = 4`:

```
sumCount(4). 4 > 1, поэтому возвращается sumCount(3) + 4
sumCount(3). 3 > 1, поэтому возвращается sumCount(2) + 3
sumCount(2). 2 > 1, поэтому возвращается sumCount(1) + 2
sumCount(1). 1 = 1, поэтому возвращается 1. Это условие
завершения рекурсии
```

Теперь посмотрим на стек вызовов:

```
sumCount(1) возвращает 1
sumCount(2) возвращает sumCount(1) + 2, т.е. 1 + 2 = 3
sumCount(3) возвращает sumCount(2) + 3, т.е. 3 + 3 = 6
sumCount(4) возвращает sumCount(3) + 4, т.е. 6 + 4 = 10
```

На этом этапе уже легче увидеть, что мы просто добавляем числа между `1` и значением, которое предоставил caller. На практике рекомендуется указывать

комментарии возле рекурсивных функций, дабы облегчить жизнь не только себе, но, возможно, и другим людям, которые будут смотреть ваш код.

Рекурсивные алгоритмы

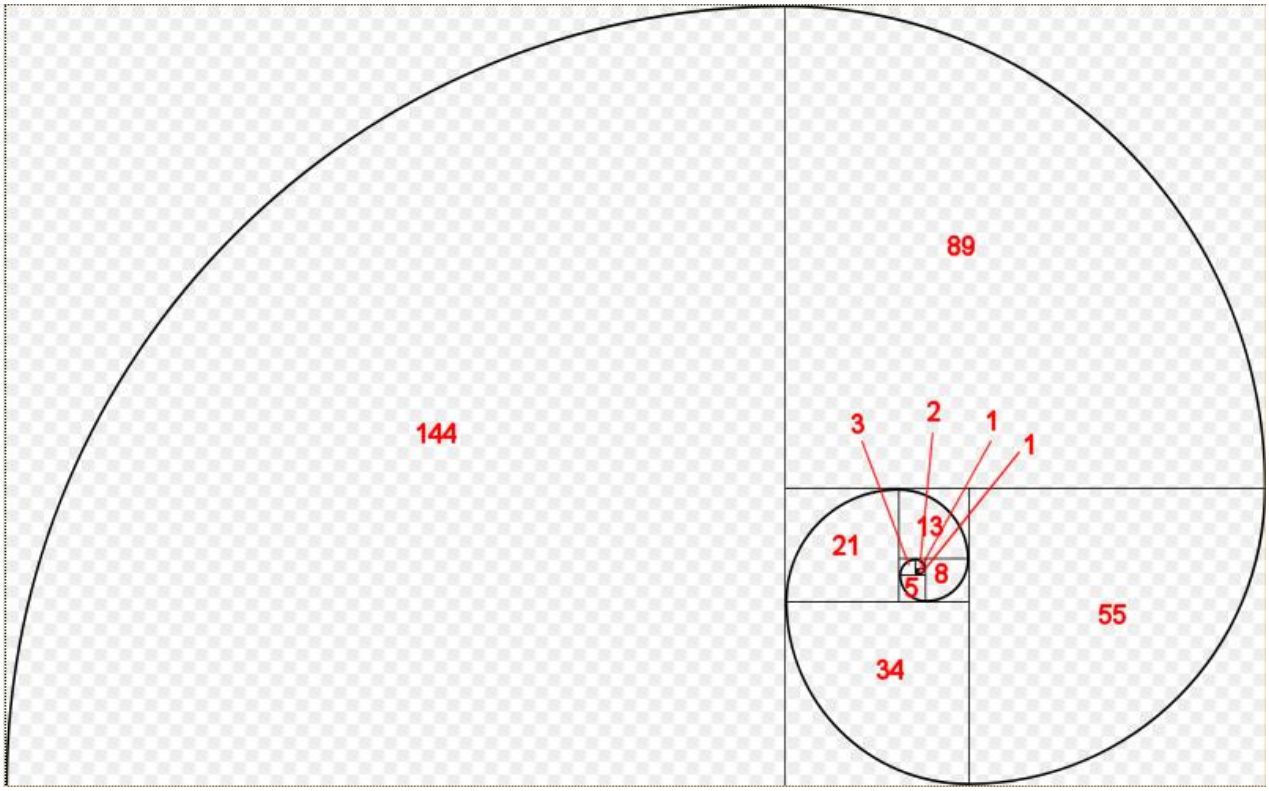
Рекурсивные функции обычно решают проблему, сначала найдя решение для подмножеств проблемы (рекурсивно), а затем модифицируя это «подрешение», дабы добраться уже до верного решения. В вышеприведенном примере, алгоритм sumCount(value) сначала решает sumCount(value-1), а затем добавляет значение `value`, чтобы найти решение для sumCount(value).

Во многих рекурсивных алгоритмах некоторые данные ввода производят предсказуемые данные вывода. Например, sumCount(1) имеет предсказуемый вывод `1` (вы можете легко это вычислить и проверить самостоятельно). Случай, когда алгоритм при определенных данных ввода производит предсказуемые данные вывода, называется **базовым случаем**. Базовые случаи работают как условия для завершения выполнения алгоритма. Их часто можно идентифицировать, рассматривая результаты вывода для следующих значений ввода: `0`, `1`, « » или `null`.

Числа Фибоначчи

Одним из наиболее известных математических рекурсивных алгоритмов является последовательность Фибоначчи. Последовательность Фибоначчи можно увидеть даже в природе: ветвление деревьев, спираль раковин, плоды ананаса, разворачивающийся папоротник и т.д.

Спираль Фибоначчи выглядит следующим образом:



Каждое из чисел Фибоначчи — это длина горизонтальной стороны квадрата, в которой находится данное число. Математически числа Фибоначчи определяются следующим образом:

```
F(n) = 0, если n = 0
1, если n = 1
f(n-1) + f(n-2), если n > 1
```

Следовательно, довольно просто написать рекурсивную функцию для вычисления n-го числа Фибоначчи:

```
1. #include <iostream>
2.
3. int fibonacci(int number)
4. {
5.     if (number == 0)
6.         return 0; // базовый случай (условие завершения)
7.     if (number == 1)
8.         return 1; // базовый случай (условие завершения)
9.     return fibonacci(number-1) + fibonacci(number-2);
10.}
11.
12.// Выводим первые 13 чисел Фибоначчи
13.int main()
14.{
15.    for (int count=0; count < 13; ++count)
16.        std:: cout << fibonacci(count) << " ";
17.
```

```
    return 0;
}
```

Результат выполнения программы:

```
0 1 1 2 3 5 8 13 21 34 55 89 144
```

Заметили? Это те же числа, что и в спирали Фибоначчи.

Рекурсия vs. Итерации

Наиболее популярный вопрос, который задают о рекурсивных функциях: «Зачем использовать рекурсивную функцию, если задание можно выполнить и с помощью итераций (используя цикл for или цикл while)?». Оказывается, вы всегда можете решить рекурсивную проблему итеративно. Однако, для нетривиальных случаев, рекурсивная версия часто бывает намного проще как для написания, так и для чтения. Например, функцию вычисления n-го числа Фибоначчи можно написать и с помощью итераций, но это будет сложнее! (Попробуйте!)

Итеративные функции (те, которые используют циклы for или while) почти всегда более эффективны, чем их рекурсивные аналоги. Это связано с тем, что каждый раз, при вызове функции, расходуется определенное количество ресурсов, которое тратится на добавление и вытягивание фреймов из стека. Итеративные функции расходуют намного меньше этих ресурсов.

Это не значит, что итеративные функции всегда являются лучшим вариантом. Иногда рекурсивная реализация может быть чище и проще, а некоторые дополнительные расходы могут быть более чем оправданы, сведя к минимуму трудности при будущей поддержке кода, особенно, если алгоритм не требует слишком много времени для поиска решения.

В общем, рекурсия является хорошим выбором, если выполняется большинство из следующих утверждений:

- рекурсивный код намного проще реализовать;
- глубина рекурсии может быть ограничена;
- итеративная версия алгоритма требует управления стеком данных;
- это не критическая часть кода, которая напрямую влияет на производительность программы.

> ***Совет***: Если рекурсивный алгоритм проще реализовать, то имеет смысл начать с рекурсии, а затем уже оптимизировать код в итеративный алгоритм.

Правило: Рекомендуется использовать итерацию, вместо рекурсии, но в тех случаях, когда это действительно практичнее.

Тест

Задание №1

Факториал целого числа N определяется как умножение всех чисел между 1 и N (0! = 1). Напишите рекурсивную функцию factorial(), которая возвращает факториал ввода. Протестируйте её с помощью первых 8 чисел.

> ***Подсказка***: Помните, что `x * y = y * x`, поэтому умножение всех чисел между 1 и N — это то же самое, что и умножение всех чисел между N и 1.

Задание №2

Напишите рекурсивную функцию, которая принимает целое число в качестве входных данных и возвращает сумму всех чисел этого значения (например, `482 = 4 + 8 + 2 = 14`). Протестируйте вашу программу, используя число `83569` (результатом должно быть `31`).

Задание №3

Это уже немного сложнее. Напишите программу, которая просит пользователя ввести целое число, а затем использует рекурсивную функцию для вывода бинарного представления этого числа. Предполагается, что число, которое введет пользователь, является положительным.

> ***Подсказка***: Используя способ №1 для конвертации чисел из десятичной системы в двоичную, вам нужно будет выводить биты «снизу вверх» (т.е. в обратном порядке), для этого ваш стейтмент вывода должен находиться *после* вызова рекурсии.

Задание №4

Используя программу из задания №3, обработайте случай, когда пользователь ввел `0` или отрицательное число, например:

```
Enter an integer: -14
11111111111111111111111111110010
```

> ***Подсказка*:** Вы можете конвертировать отрицательное целое число в положительное, используя оператор static_cast для конвертации в unsigned int.

Урок №114. Обработка ошибок, cerr и exit()

При написании программ возникновение ошибок почти неизбежно. Ошибки в языке C++ делятся на две категории: синтаксические и семантические.

Синтаксические ошибки

Синтаксическая ошибка возникает при нарушении правил грамматики языка C++. Например:

```
если 7 не равно 8, то пишем "not equal";
```

Хотя этот стейтмент нам (людям) понятен, компьютер не сможет его корректно обработать. В соответствии с правилами грамматики языка C++, корректно будет:

```
1. if (7 != 8)
2.     std::cout << "not equal";
```

Синтаксические ошибки почти всегда улавливаются компилятором и их обычно легко исправить. Следовательно, о них слишком беспокоиться не стоит.

Семантические ошибки

Семантическая (или ***"смысловая"***) **ошибка** возникает, когда код синтаксически правильный, но выполняет не то, что нужно программисту. Например:

```
1. for (int count=0; count <= 4; ++count)
2.     std::cout << count << " ";
```

Возможно, программист хотел, чтобы вывелось `0 1 2 3`, но на самом деле выведется `0 1 2 3 4`.

Семантические ошибки не улавливаются компилятором и могут иметь разное влияние: некоторые могут вообще не отображаться, что приведет к неверным результатам, к повреждению данных или вообще к сбою программы. Поэтому о семантических ошибках беспокоиться уже придется.

Они могут возникать несколькими способами. Одной из наиболее распространенных семантических ошибок является логическая ошибка. **Логическая ошибка** возникает, когда программист неправильно программирует логику выполнения кода. Например, вышеприведенный фрагмент кода имеет логическую ошибку. Вот еще один пример:

```
1. if (x >= 4)
```

```
2.      std::cout << "x is greater than 4";
```

Что произойдет, если `x` будет равен `4`? Условие выполнится как `true`, а программа выведет `x is greater than 4`. Логические ошибки иногда бывает довольно-таки трудно обнаружить.

Другой распространенной семантической ошибкой является ложное предположение. **Ложное предположение** возникает, когда программист предполагает, что что-то будет истинным или ложным, а оказывается наоборот. Например:

```
1. std::string hello = "Hello, world!";
2. std::cout << "Enter an index: ";
3.
4. int index;
5. std::cin >> index;
6.
7. std::cout << "Letter #" << index << " is " << hello[index] << std::endl;
```

Заметили потенциальную проблему здесь? Предполагается, что пользователь введет значение между `0` и длиной строки `Hello, world!`. Если же пользователь введет отрицательное число или число, которое больше длины указанной строки, то `index` окажется за пределами диапазона массива. В этом случае, поскольку мы просто выводим значение по индексу, результатом будет вывод мусора (при условии, что пользователь введет число вне диапазона). Но в других случаях ложное предположение может привести и к изменениям значений переменных, и к сбою в программе.

Безопасное программирование - это методика разработки программ, которая включает анализ областей, где могут быть допущены ложные предположения, и написание кода, который обнаруживает и обрабатывает любой случай такого нарушения, чтобы свести к минимуму риск возникновения сбоя или повреждения программы.

Определение ложных предположений

Оказывается, мы можем найти почти все предположения, которые необходимо проверить в одном из следующих 3-х мест:

- При вызове функции, когда caller может передать некорректные или семантически бессмысленные аргументы.
- При возврате значения функцией, когда возвращаемое значение может быть индикатором выполнения (произошла ли ошибка или нет).

- При обработке данных ввода (либо от пользователя, либо из файла), когда эти данные могут быть не того типа, что нужно.

Поэтому, придерживаясь безопасного программирования, нужно следовать следующим 3-м правилам:

- В верхней части каждой функции убедитесь, что все параметры имеют соответствующие значения.
- После возврата функцией значения, проверьте возвращаемое значение (если оно есть) и любые другие механизмы сообщения об ошибках на предмет того, произошла ли ошибка.
- Проверяйте данные ввода на соответствие ожидаемому типу данных и его диапазону.

Рассмотрим примеры проблем:

***Проблема №1**:* **При вызове функции caller может передать некорректные или семантически бессмысленные аргументы:**

```
1. void printString(const char *cstring)
2. {
3.     std::cout << cstring;
4. }
```

Можете ли вы определить потенциальную проблему здесь? Дело в том, что caller может передать нулевой указатель вместо допустимой строки C-style. Если это произойдет, то в программе будет сбой. Вот как правильно (с проверкой параметра функции на то, не является ли он нулевым):

```
1. void printString(const char *cstring)
2. {
3.     // Выводим cstring при условии, что он не нулевой
4.     if (cstring)
5.         std::cout << cstring;
6. }
```

***Проблема №2**:* **Возвращаемое значение может указывать на возникшую ошибку:**

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string hello = "Hello, world!";
7.     std::cout << "Enter a letter: ";
8.
9.     char ch;
10.    std::cin >> ch;
11.
```

```
12.     int index = hello.find(ch);
13.     std::cout << ch << " was found at index " << index << '\n';
14.
15.     return 0;
16. }
```

Можете ли вы определить потенциальную проблему здесь? Пользователь может ввести символ, который не находится в строке `hello`. Если это произойдет, то функция find() возвратит индекс -1, который и выведется. Правильно:

```
1.  #include <iostream>
2.  #include <string>
3.
4.  int main()
5.  {
6.      std::string hello = "Hello, world!";
7.      std::cout << "Enter a letter: ";
8.
9.      char ch;
10.     std::cin >> ch;
11.
12.     int index = hello.find(ch);
13.     if (index != -
    1) // обрабатываем случай, когда функция find() не нашла символ в строке hello
14.         std::cout << ch << " was found at index " << index << '\n';
15.     else
16.         std::cout << ch << " wasn't found" << '\n';
17.
18.     return 0;
19. }
```

Проблема №3: **При обработке данных ввода (либо от пользователя, либо из файла), эти данные могут быть не того типа и диапазона, что нужно.** Разберем программу из предыдущего примера: данный код позволяет проиллюстрировать ситуацию с обработкой ввода.

```
1.  #include <iostream>
2.  #include <string>
3.
4.  int main()
5.  {
6.      std::string hello = "Hello, world!";
7.      std::cout << "Enter an index: ";
8.
9.      int index;
10.     std::cin >> index;
11.
12.     std::cout << "Letter #" << index << " is " << hello [index] << std::endl;
13.
14.     return 0;
15. }
```

Вот как правильно (с проверкой пользовательского ввода):

```
1.  #include <iostream>
2.  #include <string>
```

```
3.
4.  int main()
5.  {
6.      std::string hello = "Hello, world!";
7.      int index;
8.
9.      do
10.     {
11.         std::cout << "Enter an index: ";
12.         std::cin >> index;
13.
14.         // Обрабатываем случай, когда пользователь ввел нецелочисленное
    значение
15.         if (std::cin.fail())
16.         {
17.             std::cin.clear();
18.             std::cin.ignore(32767, '\n');
19.             index = -1; // убеждаемся, что index имеет недопустимое значение,
    чтобы цикл продолжался
20.             continue; // этот continue может показаться здесь лишним, но он
    явно указывает на готовность прекратить выполнение этой итерации цикла
21.         }
22.
23.     } while (index < 0 || index >= hello.size()); // обрабатываем случай, когда
     пользователь ввел значение вне диапазона
24.
25.     std::cout << "Letter #" << index << " is " << hello [index] << std::endl;
26.
27.     return 0;
28. }
```

Обратите внимание, здесь проверка двухуровневая:

- Во-первых, мы должны убедиться, что пользователь введет значение того типа данных, который мы используем.
- Во-вторых, это значение должно находиться в диапазоне массива.

Обработка ложных предположений

Теперь, когда вы знаете, где обычно возникают ложные предположения, давайте поговорим о способах, позволяющих избежать их. Одного универсального способа исправления всех ошибок нет, всё зависит от характера проблемы.

Но все же **есть несколько способов обработки ложных предположений**:

***Способ №1*: Пропустите код, который зависит напрямую от правильности предположения:**

```
1.  void printString(const char *cstring)
2.  {
3.      // Выводим cstring только при условии, что он не нулевой
4.      if (cstring)
5.          std::cout << cstring;
6.  }
```

В примере, приведенном выше, если `cstring` окажется `NULL`, то мы ничего не будем выводить. Мы пропустили тот код, который напрямую зависит от значения `cstring` и который с ним работает (в коде мы просто выводим этот `cstring`). Это может быть хорошим вариантом, если пропущенный стейтмент не является критическим и не влияет на логику программы. Основной недостаток при этом заключается в том, что caller или пользователь не имеет возможности определить, что что-то пошло не так.

***Способ №2*: Из функции возвращайте код ошибки обратно в caller и позволяйте caller-у обработать эту ошибку:**

```
1. int getArrayValue(const std::array &array, int index)
2. {
3.     // Используем условие if для обнаружения ложного предположения
4.     if (index < 0 || index >= array.size())
5.         return -1; // возвращаем код ошибки обратно в caller
6.
7.     return array[index];
8. }
```

Здесь функция возвратит `-1`, если caller передаст некорректный `index`. Возврат перечислителя в качестве кода ошибки будет еще лучшим вариантом.

***Способ №3*: Если нужно немедленно завершить программу, то используйте функцию exit(), которая находится в заголовочном файле cstdlib, для возврата кода ошибки обратно в операционную систему:**

```
1. #include <cstdlib> // for exit()
2.
3. int getArrayValue(const std::array &array, int index)
4. {
5.     // Используем условие if для обнаружения ложного предположения
6.     if (index < 0 || index >= array.size())
7.         exit(2); // завершаем программу и возвращаем код ошибки 2 обратно в ОС
8.
9.     return array[index];
10. }
```

Если caller передаст некорректный `index`, то программа немедленно завершит свое выполнение и передаст код ошибки `2` обратно в операционную систему.

***Способ №4*: Если пользователь ввел данные не того типа, что нужно - попросите пользователя ввести данные еще раз:**

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string hello = "Hello, world!";
```

```
7.      int index;
8.
9.      do
10.     {
11.         std::cout << "Enter an index: ";
12.         std::cin >> index;
13.
14.         // Обрабатываем случай, когда пользователь ввел нецелочисленное
    значение
15.         if (std::cin.fail())
16.         {
17.             std::cin.clear();
18.             std::cin.ignore(32767, '\n');
19.             index = -1; // убеждаемся, что index имеет недопустимое
    значение, чтобы цикл продолжался
20.             continue; // этот continue может показаться здесь лишним, но он
    явно указывает на готовность прекратить выполнение этой итерации цикла
21.         }
22.
23.     } while (index < 0 || index >= hello.size()); // обрабатываем случай,
    когда пользователь ввел значение вне диапазона
24.
25.     std::cout << "Letter #" << index << " is " << hello [index] << std::endl;
26.
27.     return 0;
28. }
```

***Способ №5**:* **Используйте cerr. cerr** — это объект вывода (как и cout), который находится в заголовочном файле iostream и выводит сообщения об ошибках в консоль (как и cout), но только эти сообщения можно еще и перенаправить в отдельный файл с ошибками. Т.е. основное отличие cerr от cout заключается в том, что cerr целенаправленно используется для вывода сообщений об ошибках, тогда как cout — для вывода всего остального. Например:

```
1.  void printString(const char *cstring)
2.  {
3.      // Выводим cstring при условии, что он не нулевой
4.      if (cstring)
5.          std::cout << cstring;
6.      else
7.          std::cerr << "function printString() received a null parameter";
8.  }
```

В примере, приведенном выше, мы не только пропускаем код, который напрямую зависит от правильности предположения, но также регистрируем ошибку, чтобы пользователь мог позже определить, почему программа выполняется не так, как нужно.

***Способ №6**:* **Если вы работаете в какой-то графической среде, то распространенной практикой является вывод всплывающего окна с кодом ошибки, а затем немедленное завершение программы.** То, как это сделать, зависит от конкретной среды разработки.

Урок №115. assert и static_assert

Использование операторов условного ветвления для обнаружения ложного предположения, а также вывода сообщения об ошибке и завершения выполнения программы является настолько распространенным решением возникающих проблем, что C++ решил это дело упростить. И упростил он его с помощью **assert**.

Стейтмент assert

Стейтмент assert (или ***"оператор проверочного утверждения"***) в языке C++ - это макрос препроцессора, который обрабатывает условное выражение во время выполнения. Если условное выражение истинно, то стейтмент assert ничего не делает. Если же оно ложное, то выводится сообщение об ошибке, и программа завершается. Это сообщение об ошибке содержит ложное условное выражение, а также имя файла с кодом и номером строки с assert. Таким образом, можно легко найти и идентифицировать проблему, что очень помогает при отладке программ.

Сам assert реализован в заголовочном файле cassert и часто используется как для проверки корректности переданных параметров функции, так и для проверки возвращаемого значения функции:

```
1. #include <cassert> // для assert()
2. 
3. int getArrayValue(const std::array<int, 10> &array, int index)
4. {
5.     // Предполагается, что значение index-а находится между 0 и 8
6.     assert(index >= 0 && index <= 8); // это строка 6 в Program.cpp
7. 
8.     return array[index];
9. }
```

Если в вышеприведенной программе вызвать `getArrayValue(array, -3);`, то программа выведет следующее сообщение:

```
Assertion failed: index >= 0 && index <=8, file
C:\\VCProjects\\Program.cpp, line 6
```

Рекомендуется использовать стейтменты assert. Иногда утверждения assert бывают не очень описательными, например:

```
1. assert(found);
```

Если этот assert сработает, то получим:

```
Assertion failed: found, file C:\\VCProjects\\Program.cpp, line 42
```

Но что это нам сообщает? Очевидно, что что-то не было найдено, но что именно? Вам нужно будет самому пройтись по коду, чтобы это определить.

К счастью, есть небольшой трюк, который можно использовать для исправления этой ситуации. Просто добавьте сообщение в качестве строки C-style вместе с логическим оператором И:

```
1. assert(found && "Animal could not be found in database");
```

Как это работает? Строка C-style всегда принимает значение `true`. Поэтому, если `found` примет значение `false`, то `false && true = false`. Если же `found` примет значение `true`, то `true && true = true`. Таким образом, строка C-style вообще не влияет на обработку утверждения.

Однако, если assert сработает, то строка C-style будет включена в сообщение assert:

```
Assertion failed: found && "Animal could not be found in database", file C:\\VCProjects\\Program.cpp, line 42
```

Это даст дополнительное объяснение того, что пошло не так.

NDEBUG

Функция assert() тратит мало ресурсов на проверку условия. Кроме того, стейтменты assert (в идеале) никогда не должны встречаться в релизном коде (потому что ваш код к этому моменту уже должен быть тщательно протестирован). Следовательно, многие разработчики предпочитают использовать assert только в конфигурации Debug. В языке C++ есть возможность отключить все assert-ы в релизном коде - использовать директиву `#define NDEBUG`:

```
1. #define NDEBUG
2.
3. // Все стейтменты assert будут проигнорированы аж до самого конца этого файла
```

Некоторые IDE устанавливают `NDEBUG` по умолчанию, как часть параметров проекта в конфигурации Release.

Обратите внимание, функция exit() и assert (если он срабатывает) немедленно прекращают выполнение программы, без возможности выполнить дальнейшую

любую очистку (например, закрыть файл или базу данных). Из-за этого их следует использовать аккуратно.

static_assert

В C++11 добавили еще один тип assert-а - **static_assert**. В отличие от assert, который срабатывает во время выполнения программы, static_assert срабатывает во время компиляции, вызывая ошибку компилятора, если условие не является истинным. Если условие ложное, то выводится диагностическое сообщение.

Вот пример использования static_assert для проверки размеров определенных типов данных:

```
1. static_assert(sizeof(long) == 8, "long must be 8 bytes");
2. static_assert(sizeof(int) == 4, "int must be 4 bytes");
3.
4. int main()
5. {
6.     return 0;
7. }
```

Результат на моем компьютере:

```
1>C:\ConsoleApplication1\main.cpp(19): error C2338: long must be 8 bytes
```

Поскольку static_assert обрабатывается компилятором, то условная часть static_assert также должна обрабатываться во время компиляции. Поскольку static_assert не обрабатывается во время выполнения программы, то стейтменты static_assert могут быть размещены в любом месте кода (даже в глобальном пространстве).

В C++11 диагностическое сообщение должно быть обязательно предоставлено в качестве второго параметра. В C++17 предоставление диагностического сообщения является необязательным.

Урок №116. Аргументы командной строки

Из предыдущих уроков мы уже знаем, что при компиляции и линкинге, компилятор создает исполняемый файл. Когда программа запускается, выполнение начинается с первой строки функции main(). До этого урока мы объявляли main() следующим образом:

```
1. int main()
```

Обратите внимание, в этой версии функции main() никаких параметров нет. Тем не менее, многие программы нуждаются в некоторых входных данных. Например, предположим, что вы пишете программу под названием Picture, которая принимает изображение в качестве входных данных, а затем делает из этого изображения миниатюру (уменьшенная версия изображения). Как функция picture() узнает, какое изображение нужно принять и обработать? Пользователь должен сообщить программе, какой файл следует открыть. Это можно сделать следующим образом:

```
1. // Программа: Picture
2. #include <iostream>
3. #include <string>
4.
5. int main()
6. {
7.     std::cout << "Enter name of image-file to create a thumbnail for: ";
8.     std::string filename;
9.     std::cin >> filename;
10.
11.    // Открываем файл-изображение
12.    // Создаем миниатюру
13.    // Выводим миниатюру
14.}
```

Тем не менее, здесь есть потенциальная проблема. Каждый раз при запуске программа будет ожидать пользовательский ввод. Это не проблема, если вы вручную запускаете программу из командной строки один раз для одного изображения. Но это уже проблема, если вы хотите работать с большим количеством файлов или чтобы другая программа имела возможность запустить эту программу.

Рассмотрим это детально. Например, вы хотите создать миниатюры для всех файлов-изображений, которые находятся в определенном каталоге. Как это сделать? Вы можете запускать эту программу столько раз, сколько есть изображений в каталоге, введя каждое имя файла вручную. Однако, если есть сотни изображений, такой подход будет, мягко говоря, не очень эффективным! Решением здесь будет написать программу, которая перебирала бы каждое имя файла в каталоге, вызывая каждый раз функцию picture() для каждого файла.

Теперь рассмотрим случай, когда у вас есть веб-сайт, и вы хотите, чтобы он создавал миниатюру каждый раз, когда пользователь загружает изображение на сайт. Эта программа не может принимать входные данные из Интернета и следует логический вопрос: "Как тогда вводить имя файла?". Выходом является вызов веб-сервером функции picture() автоматически каждый раз после загрузки файла.

В обоих случаях нам нужно, чтобы внешняя программа передавала имя файла в качестве входных данных в нашу программу при её запуске, вместо того, чтобы функция picture() сама дожидалась, пока пользователь вручную введет имя файла.

Аргументы командной строки - это необязательные строковые аргументы, передаваемые операционной системой в программу при её запуске. Программа может их использовать в качестве входных данных, либо игнорировать. Подобно тому, как параметры одной функции предоставляют данные для параметров другой функции, так и аргументы командной строки предоставляют возможность людям или программам предоставлять входные данные для программы.

Передача аргументов командной строки

Исполняемые программы могут запускаться в командной строке через вызов. Например, для запуска исполняемого файла `MyProgram`, который находится в корневом каталоге диска C в ОС Windows, вам нужно ввести:

```
C:\>MyProgram
```

Чтобы передать аргументы командной строки в `MyProgram`, вам нужно будет их просто перечислить после имени исполняемого файла:

```
C:\>MyProgram SomeContent.txt
```

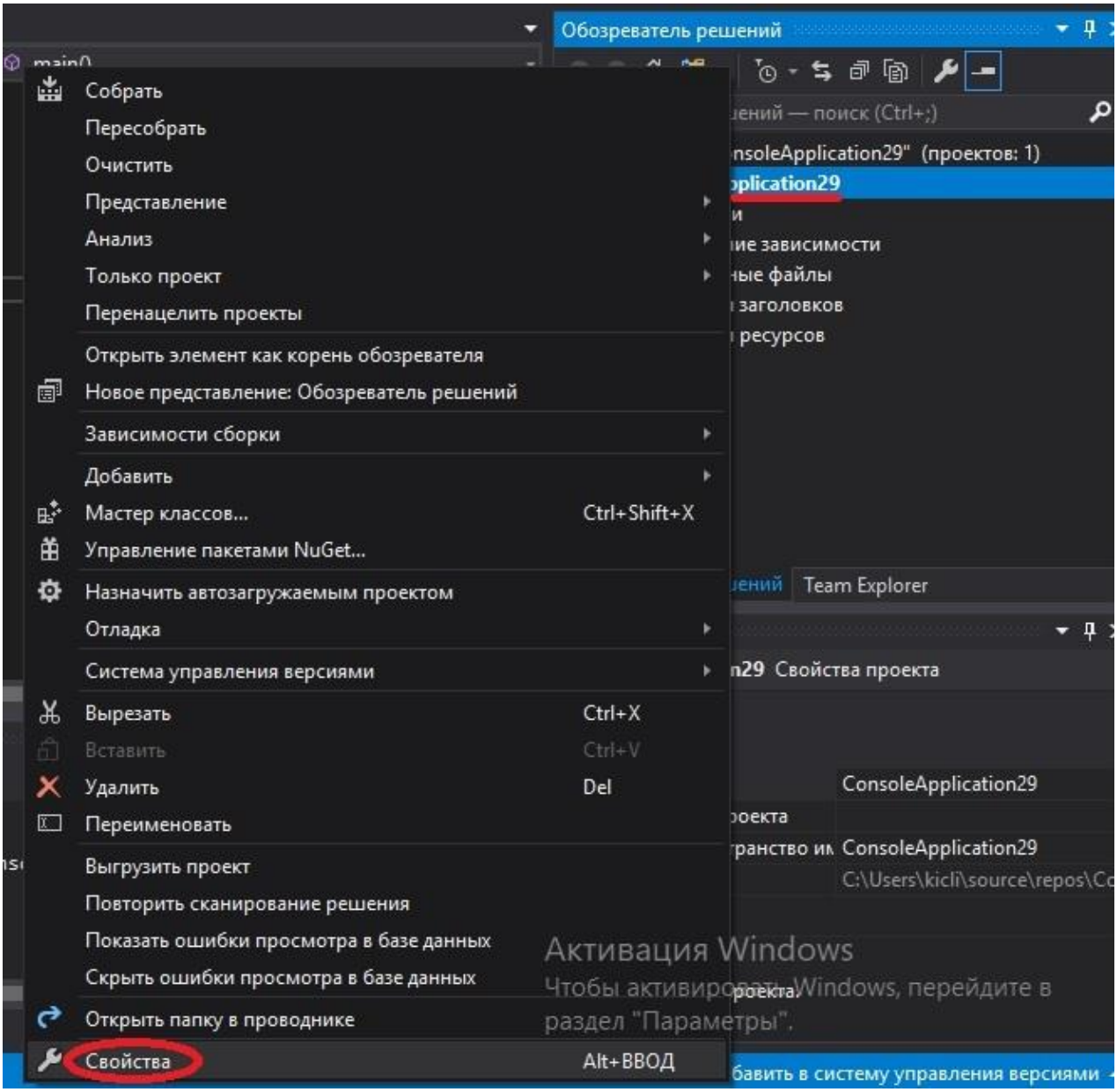
Теперь, при запуске `MyProgram`, `SomeContent.txt` будет предоставлен в качестве аргумента командной строки. Программа может иметь несколько аргументов командной строки, разделенных пробелами:

```
C:\>MyProgram SomeContent.txt SomeOtherContent.txt
```

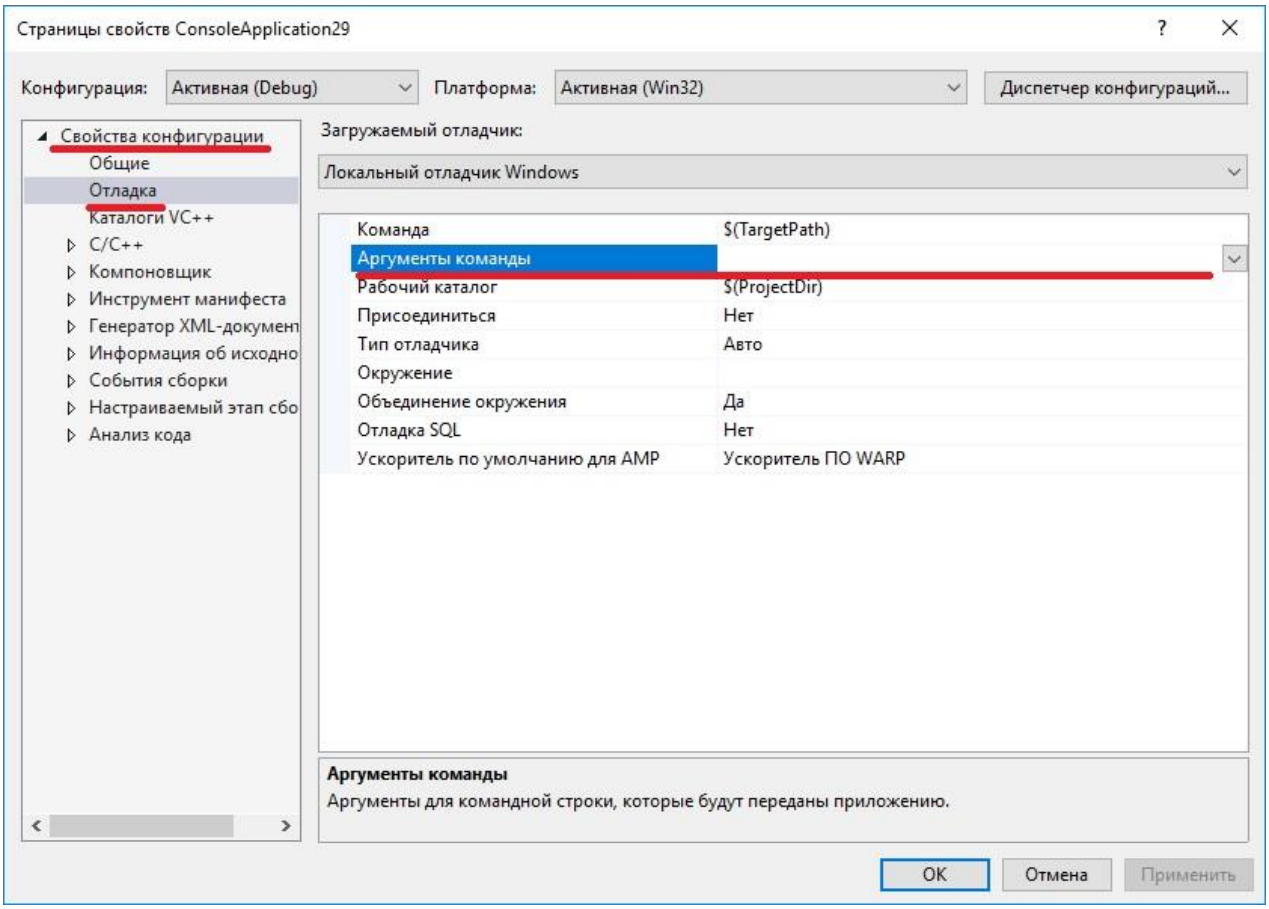
Это также работает и с ОС Linux (хотя структура каталогов будет отличаться от структуры каталогов в ОС Windows).

Если вы запускаете свою программу из среды IDE, то ваша IDE должна предоставить способ ввода аргументов командной строки.

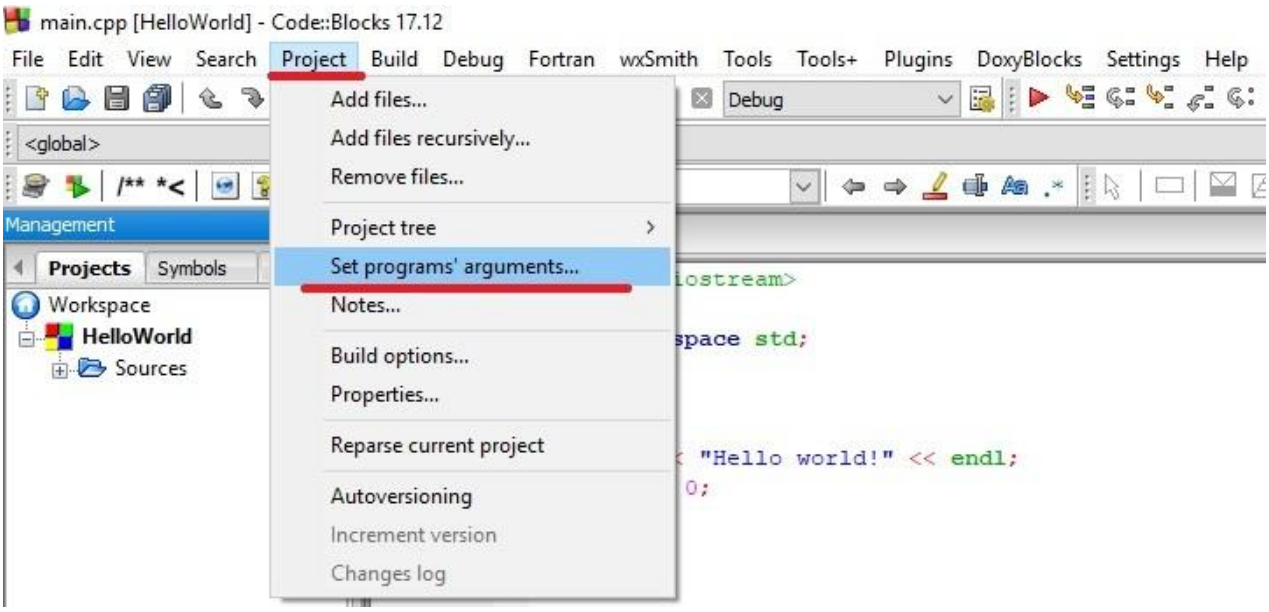
Для пользователей Visual Studio: Щелкните правой кнопкой мыши по нужному проекту в меню `"Обозреватель Решений" > "Свойства"`:



Затем выберите `"Свойства конфигурации" > "Отладка"`. На правой панели будет строка `"Аргументы команды"`. Вы сможете здесь ввести аргументы командной строки, и они будут автоматически переданы вашей программе при её запуске:



Пользователям Code::Blocks: Выберите `"Project" > "Set program`s arguments"`:



Использование аргументов командной строки

Теперь, когда вы знаете, как передавать аргументы командной строки в программу, следующим шагом будет доступ к ним из программы. Для этого используется уже другая форма функции main(), которая принимает два аргумента (`argc` и `argv`) следующим образом:

```
1. int main(int argc, char *argv[])
```

Также вы можете увидеть и такой вариант:

```
1. int main(int argc, char** argv)
```

Хоть оба эти варианта идентичны по своей сути, но рекомендуется использовать первый, так как он интуитивно понятнее.

`argc` (англ. *"**arg**ument **c**ount" = "количество аргументов"*) - это целочисленный параметр, содержащий количество аргументов, переданных в программу. `argc` всегда будет как минимум один, так как первым аргументом всегда является имя самой программы. Каждый аргумент командной строки, который предоставляет пользователь, заставит `argc` увеличиться на единицу.

`argv` (англ. *"**arg**ument **v**alues" = "значения аргументов"*) - это место, где хранятся фактические значения аргументов. Хотя объявление `argv` выглядит немного пугающе, но это всего лишь массив строк C-style. Длиной этого массива является `argc`.

Давайте напишем короткую программу `MyArguments`, которая будет выводить значения всех аргументов командной строки:

```
1. // Программа MyArguments
2. #include <iostream>
3.
4. int main(int argc, char *argv[])
5. {
6.     std::cout << "There are " << argc << " arguments:\n";
7.
8.     // Перебираем каждый аргумент и выводим его порядковый номер и значение
9.     for (int count=0; count < argc; ++count)
10.        std::cout << count << " " << argv[count] << '\n';
11.
12.    return 0;
13. }
```

Теперь, при вызове `MyArguments` с аргументами командной строки `SomeContent.txt` и `200`, вывод будет следующим:

```
There are 3 arguments:
0 C:\MyArguments
1 SomeContent.txt
2 200
```

Нулевой параметр - это путь и имя текущей программы. Первый и второй параметры здесь являются аргументами командной строки, которые мы передали.

Обработка числовых аргументов

Аргументы командной строки всегда передаются в качестве строк, даже если предоставленное значение является числовым. Чтобы использовать аргумент командной строки в виде числа, вам нужно будет конвертировать его из строки в число. К сожалению, в языке C++ это делается немного сложнее, чем должно быть:

```
#include <iostream>
#include <string>
#include <sstream> // для std::stringstream
#include <cstdlib> // для exit()

int main(int argc, char *argv[])
{
    if (argc <= 1)
    {
        // В некоторых операционных системах argv[0] может быть просто пустой строкой, без имени программы

        // Обрабатываем случай, когда argv[0] может быть пустым или не пустым
        if (argv[0])
            std::cout << "Usage: " << argv[0] << " <number>" << '\n';
        else
            std::cout << "Usage: <program name> <number>" << '\n';

        exit(1);
    }

    std::stringstream convert(argv[1]); // создаем переменную stringstream с именем convert, инициализируя её значением argv[1]

    int myint;
    if (!(convert >> myint)) // выполняем конвертацию
        myint = 0; // если конвертация терпит неудачу, то присваиваем myint значение по умолчанию

    std::cout << "Got integer: " << myint << '\n';

    return 0;
}
```

Если мы запустим эту программу с аргументом командной строки `843`, то результатом будет:

```
Got integer: 843
```

std::stringstream работает почти так же, как и std::cin. Здесь мы инициализируем переменную std::stringstream значением `argv[1]`, так что мы можем использовать оператор `>>` для извлечения значения в переменную типа int.

Анализ аргументов командной строки

Когда вы пишете что-то в командной строке (или запускаете свою программу из среды IDE), то операционная система ответственна за то, чтобы ваш запрос проделал правильный путь. Это связано не только с запуском исполняемого файла, но и с анализом любых аргументов для определения того, как их следует обрабатывать и передавать в программу.

Операционные системы имеют обязательные правила обработки специальных символов (двойные кавычки, бэкслешы и т.д.).

Например:

```
MyArguments Hello world!
```

Результат:

```
There are 3 arguments:
0 C:\MyArguments
1 Hello
2 world!
```

Строки, переданные в двойных кавычках, считаются частью одной и той же строки:

```
MyArguments "Hello world!"
```

Результат:

```
There are 2 arguments:
0 C:\MyArguments
1 Hello world!
```

Для того, чтобы вывести каждое слово отдельно на строке, используйте бэкслешы:

```
MyArguments \"Hello world!\"
```

Результат:

```
There are 3 arguments:
0 C:\MyArguments
1 "Hello
2 world!"
```

Заключение

Аргументы командной строки предоставляют отличный способ для пользователей или других программ передавать входные данные в программу при её запуске. Используйте любые входные данные, необходимые программе при запуске, в качестве аргументов командной строки. Если командная строка не передана, то вы всегда сможете это обнаружить и попросить пользователя ввести данные вручную. Таким образом, ваша программа будет работать в любом случае.

Урок №117. Эллипсис

Во всех функциях, которые мы рассматривали до этого момента, количество их параметров должно было быть известно заранее (даже если это параметры по умолчанию). Однако есть несколько случаев, в которых полезно передать переменную, указывающую на количество параметров, в функцию. В языке C++ есть специальный объект, который позволяет это сделать — эллипсис.

Эллипсис

Функции, использующие эллипсис, выглядят следующим образом:

```
тип_возврата имя_функции(список_аргументов, ...)
```

`список_аргументов` - это один или несколько обычных параметров функции. Обратите внимание, функции, которые используют эллипсис, должны иметь по крайней мере один параметр, который не является эллипсисом.

Эллипсис (англ. ***"ellipsis"***), который представлен в виде многоточия `...` в языке C++, всегда должен быть последним параметром в функции. О нем можно думать, как о массиве, который содержит любые другие параметры, кроме тех, которые указаны в `список_аргументов`.

Рассмотрим пример с использованием эллипсиса. Предположим, что нам нужно написать функцию, которая вычисляет среднее арифметическое переданных аргументов:

```
1. #include <iostream>
2. #include <cstdarg> // требуется для использования эллипсиса
3.
4. // Эллипсис должен быть последним параметром.
5. // Переменная count - это количество переданных аргументов
6. double findAverage(int count, ...)
7. {
8.     double sum = 0;
9.
10.    // Мы получаем доступ к эллипсису через va_list, поэтому объявляем
   переменную этого типа
11.    va_list list;
12.
13.    // Инициализируем va_list, используя va_start. Первый параметр – это
   список, который нужно инициализировать.
14.    // Второй параметр - это последний параметр, который не является эллипсисом
15.    va_start(list, count);
16.
17.    // Перебираем каждый из аргументов эллипсиса
18.    for (int arg=0; arg < count; ++arg)
19.         // Используем va_arg для получения параметров из эллипсиса.
```

```
20.         // Первый параметр - это va_list, который мы используем.
21.         // Второй параметр - это ожидаемый тип параметров
22.         sum += va_arg(list, int);
23.
24.     // Выполняем очистку va_list, когда уже сделали всё необходимое
25.     va_end(list);
26.
27.     return sum / count;
28. }
29.
30. int main()
31. {
32.     std::cout << findAverage(4, 1, 2, 3, 4) << '\n';
33.     std::cout << findAverage(5, 1, 2, 3, 4, 5) << '\n';
34. }
```

Результат выполнения программы:

```
2.5
3
```

Как вы можете видеть, функция findAverage() принимает переменную `count`, которая указывает на количество передаваемых аргументов. Рассмотрим другие компоненты этого примера.

Во-первых, мы должны подключить заголовочный файл cstdarg. Этот заголовок определяет `va_list`, `va_start` и `va_end` - макросы, необходимые для доступа к параметрам, которые являются частью эллипсиса.

Затем мы объявляем функцию, которая использует эллипсис. Помните, что `список_аргументов` должен быть представлен одним или несколькими фиксированными параметрами. Здесь мы передаем одно целочисленное значение, которое сообщает функции, сколько будет параметров.

Обратите внимание, в эллипсисе нет никаких имен переменных! Вместо этого мы получаем доступ к значениям через специальный тип — `va_list`. О `va_list` можно думать, как об указателе, который указывает на массив с эллипсисом. Сначала мы объявляем переменную `va_list`, которую называем просто `list` для удобства использования.

Затем нам нужно, чтобы `list` указывал на параметры эллипсиса. Делается это с помощью `va_start()`, который имеет два параметра: `va_list` и имя последнего параметра, который не является эллипсисом. После того, как `va_start()` был вызван, `va_list` указывает на первый параметр из списка передаваемых аргументов.

Чтобы получить значение параметра, на который указывает `va_list`, нужно использовать `va_arg()`, который также имеет два параметра: `va_list` и тип данных параметра, к которому мы пытаемся получить доступ. Обратите внимание, с помощью `va_arg()` мы также переходим к следующему параметру `va_list`!

Наконец, когда мы уже всё сделали, нужно выполнить очистку: `va_end()` с параметром `va_list`.

Почему эллипсис небезопасен?

Эллипсис предоставляет программисту большую гибкость для реализации функций, которые принимают переменную, указывающую на общее количество параметров. Однако эта гибкость имеет свои недостатки.

С обычными параметрами функции компилятор использует проверку типов для гарантирования того, что типы аргументов функции соответствуют типам параметров функции (или аргументы могут быть неявно преобразованы для дальнейшего соответствия). Это делается с целью предотвращения случаев, когда вы передаете в функцию целочисленное значение, тогда как она ожидает строку (или наоборот). Обратите внимание, параметры эллипсиса не имеют объявлений типа данных. При их использовании компилятор полностью пропускает проверку типов данных. Это означает, что можно отправить аргументы любого типа в многоточии, и компилятор не сможет предупредить вас, что это произошло. В конечном итоге, мы получим сбой или неверные результаты. При использовании эллипсиса вся ответственность ложится на caller, и от него зависит корректность переданных аргументов в функцию. Очевидно, что это является хорошей лазейкой для возникновения ошибок. Рассмотрим пример такой ошибки:

```
1. std::cout << findAverage(6, 1.0, 2, 3, 4, 5, 6) << '\n';
```

Хотя на первый взгляд всё может показаться достаточно безвредным, но посмотрите на второй аргумент типа double — он должен быть типа int. Хотя всё скомпилируется без ошибок, но результат следующий:

```
1.78782e+08
```

Число не маленькое. Как это произошло?

Как мы уже знаем из предыдущих уроков, компьютер сохраняет все данные в виде последовательности бит. Тип переменной указывает компьютеру, как перевести эту последовательность бит в определенное (читабельное) значение. Однако в эллипсисе тип переменной отбрасывается. Следовательно, единственный способ

получить нормальное значение обратно из эллипсиса - вручную указать `va_arg()`, каков ожидаемый тип параметра. Это то, что делает второй параметр в `va_arg()`. Если фактический тип параметра не соответствует ожидаемому типу параметра, то происходят плохие вещи.

В программе, приведенной выше, с помощью `va_arg()`, мы указали, что все параметры должны быть типа int. Следовательно, каждый вызов `va_arg()` будет возвращать последовательность бит, которая будет конвертирована в тип int.

В этом случае проблема заключается в том, что значение типа double, которое мы передали в качестве первого аргумента эллипсиса, занимает 8 байт, тогда как `va_arg(list, int)` возвращает только 4 байта данных при каждом вызове (тип int занимает 4 байта). Следовательно, первый вызов `va_arg` возвращает первую часть типа double (4 байта), а второй вызов `va_arg` возвращает вторую часть типа double (еще 4 байта). Итого, в результате получаем мусор.

Поскольку проверка типов пропущена, то компилятор даже не будет жаловаться, если мы сделаем что-то вообще дикое, например:

```
1. int value = 8;
2. std::cout << findAverage(7, 1.0, 3, "Hello, world!", 'G', &value, &findAverage)
    << '\n';
```

Верите или нет, но это действительно скомпилировалось без ошибок и выдало следующий результат на моем компьютере:

```
1.56805e+08
```

Этот результат подтверждает фразу: «Мусор на входе, мусор на выходе».

Мало того, что эллипсис отбрасывает тип параметров, он также отбрасывает и количество этих параметров. Это означает, что нам нужно будет самим разработать решение для отслеживания количества параметров, передаваемых в эллипсис. Как правило, это делается одним из следующих 3-х способов:

Способ №1: **Передать параметр-длину**. Нужно, чтобы один из фиксированных параметров, не входящих в эллипсис, отображал количество переданных параметров. Это решение использовалось в программе, приведенной выше. Однако даже здесь мы столкнемся с проблемами, например:

```
1. std::cout << findAverage(6, 1, 2, 3, 4, 5) << '\n';
```

Результат на моем компьютере:

```
4.16667
```

Что случилось? Мы сообщили findAverage(), что собираемся передать 6 значений, но фактически передали только 5. Следовательно, с первыми пятью значениями, возвращаемыми `va_arg()` — всё ок. Но вот 6-е значение, которое возвращает `va_arg()` — это просто мусор из стека, так как мы его не передавали. Следовательно, таков и результат. По крайней мере, здесь очевидно, что это значение является мусором. А вот рассмотрим более коварный случай:

```
1. std::cout << findAverage(6, 1, 2, 3, 4, 5, 6, 7) << '\n';
```

Результат:

```
3.5
```

На первый взгляд всё корректно, но последнее число (`7`) в списке аргументов игнорируется, так как мы сообщили, что собираемся предоставить 6 параметров (а предоставили 7). Такие ошибки бывает довольно-таки трудно обнаружить.

***Способ №2:* Использовать контрольное значение**. Контрольное значение — это специальное значение, которое используется для завершения цикла при его обнаружении. Например, нуль-терминатор используется в строках для обозначения конца строки. В эллипсисе контрольное значение передается последним из аргументов. Вот программа, приведенная выше, но уже с использованием контрольного значения `-1`:

```
1. #include <iostream>
2. #include <cstdarg> // требуется для использования эллипсиса
3.
4. // Эллипсис должен быть последним параметром
5. double findAverage(int first, ...)
6. {
7.     // Обработка первого значения
8.     double sum = first;
9.
10.    // Мы получаем доступ к эллипсису через va_list, поэтому объявляем переменную этого типа
11.    va_list list;
12.
13.    // Инициализируем va_list, используя va_start. Первый параметр - это список, который нужно инициализировать.
14.    // Второй параметр - это последний параметр, который не является эллипсисом
15.    va_start(list, first);
16.
17.    int count = 1;
18.    // Бесконечный цикл
19.    while (1)
20.    {
```

```
21.         // Используем va_arg для получения параметров из эллипсиса.
22.         // Первый параметр - это va_list, который мы используем.
23.         // Второй параметр - это ожидаемый тип параметров
24.         int arg = va_arg(list, int);
25.
26.         // Если текущий параметр является контрольным значением, то прекращаем
    выполнение цикла
27.         if (arg == -1)
28.             break;
29.
30.         sum += arg;
31.         count++;
32.     }
33.
34.     // Выполняем очистку va_list, когда уже сделали всё необходимое
35.     va_end(list);
36.
37.     return sum / count;
38. }
39.
40. int main()
41. {
42.     std::cout << findAverage(1, 2, 3, 4, -1) << '\n';
43.     std::cout << findAverage(1, 2, 3, 4, 5, -1) << '\n';
44. }
```

Обратите внимание, нам уже не нужно явно передавать длину в качестве первого параметра. Вместо этого мы передаем контрольное значение в качестве последнего параметра.

Однако здесь также есть нюансы. Во-первых, язык C++ требует, чтобы мы передавали хотя бы один фиксированный параметр. В предыдущем примере для этого использовалась переменная `count`. В этом примере первое значение является частью чисел, используемых в вычислении. Поэтому, вместо обработки первого значения в паре с другими параметрами эллипсиса, мы явно объявляем его как обычный параметр. Затем нам нужно это обработать внутри функции (мы присваиваем переменной `sum` значение `first`, а не `0`, как в предыдущей программе).

Во-вторых, требуется, чтобы пользователь передал контрольное значение последним в списке. Если пользователь забудет передать контрольное значение (или передаст неправильное), то функция будет циклически работать до тех пор, пока не дойдет до значения, которое будет соответствовать контрольному (которое не было указано), т.е. мусору (или произойдет сбой).

***Способ №3:* Использовать строку-декодер.** Передайте строку-декодер в функцию, чтобы сообщить, как правильно интерпретировать параметры:

```
1. #include <iostream>
2. #include <string>
3. #include <cstdarg> // требуется для использования эллипсиса
```

```
4.
5.  // Эллипсис должен быть последним параметром
6.  double findAverage(std::string decoder, ...)
7.  {
8.      double sum = 0;
9.
10.     // Мы получаем доступ к эллипсису через va_list, поэтому объявляем
    переменную этого типа
11.     va_list list;
12.
13.     // Инициализируем va_list, используя va_start. Первый параметр – это
    список, который необходимо инициализировать.
14.     // Второй параметр - это последний параметр, который не является эллипсисом
15.     va_start(list, decoder);
16.
17.     int count = 0;
18.     // Бесконечный цикл
19.     while (1)
20.     {
21.         char codetype = decoder[count];
22.         switch (codetype)
23.         {
24.         default:
25.         case '\0':
26.             // Выполняем очистку va_list, когда уже сделали всё необходимое
27.             va_end(list);
28.             return sum / count;
29.
30.         case 'i':
31.             sum += va_arg(list, int);
32.             count++;
33.             break;
34.
35.         case 'd':
36.             sum += va_arg(list, double);
37.             count++;
38.             break;
39.         }
40.     }
41. }
42.
43. int main()
44. {
45.     std::cout << findAverage("iiii", 1, 2, 3, 4) << '\n';
46.     std::cout << findAverage("iiiii", 1, 2, 3, 4, 5) << '\n';
47.     std::cout << findAverage("ididdi", 1, 2.2, 3, 3.5, 4.5, 5) << '\n';
48. }
```

В этом примере мы передаем строку, в которой указывается как количество передаваемых аргументов, так и их типы (`i = int`, `d = double`). Таким образом, мы можем работать с параметрами разных типов. Однако следует помнить, что если число или типы передаваемых параметров не будут в точности соответствовать тому, что указано в строке-декодере, то могут произойти плохие вещи.

Рекомендации по безопасному использованию эллипсиса

Во-первых, если это возможно, не используйте эллипсис вообще! Часто доступны другие разумные решения, даже если они требуют немного больше работы и времени. Например, в функции findAverage() в вышеприведенной программе мы могли бы передать динамически выделенный массив целых чисел, вместо использования эллипсиса. Это бы обеспечило проверку типов (гарантируя, что caller не попытается сделать что-то бессмысленное), сохраняя при этом возможность передавать переменную-длину, которая бы указывала на количество всех передаваемых значений.

Во-вторых, если вы используете эллипсис, не смешивайте разные типы аргументов в пределах вашего эллипсиса, если это возможно. Это уменьшит вероятность того, что caller случайно передаст данные не того типа, а `va_arg()` произведет результат-мусор.

В-третьих, использование параметра `count` или строки-декодера в качестве `список_аргументов` обычно безопаснее, чем использование контрольного значения. Это гарантирует, что цикл эллипсиса будет завершен после четко определенного количества итераций.

Урок №118. Лямбда-выражения (анонимные функции)

Рассмотрим следующий пример:

```
1. #include <algorithm>
2. #include <array>
3. #include <iostream>
4. #include <string_view>
5.
6. static bool containsNut(std::string_view str) // static в данном контексте
   означает внутреннее связывание
7. {
8.   // Функция std::string_view::find() возвращает std::string_view::npos, если
   она не нашла подстроку.
9.   // В противном случае, она возвращает индекс, где происходит вхождение
   подстроки в строку str
10.  return (str.find("nut") != std::string_view::npos);
11.}
12.
13.int main()
14.{
15.  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

16.
17.  // std::find_if() принимает указатель на функцию
18.  auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };
19.
20.  if (found == arr.end())
21.  {
22.    std::cout << "No nuts\n";
23.  }
24.  else
25.  {
26.    std::cout << "Found " << *found << '\n';
27.  }
28.
29.  return 0;
30.}
```

Этот код перебирает массив строк в поисках первого попавшегося элемента, который содержит подстроку `nut`. Таким образом, результат выполнения программы:

```
Found walnut
```

Хотя это и рабочий код, но мы можем его улучшить.

Проблема кроется в том, что функция std::find_if() требует указатель на функцию в качестве аргумента. Из-за этого мы вынуждены определить новую функцию, которая будет использована только один раз, дать ей имя и поместить её в глобальную область видимости (т.к. функции не могут быть вложенными!). При этом она будет настолько короткой, что быстрее и проще понять её смысл,

посмотрев лишь на одну строку кода, нежели изучать описание этой функции и её имя.

Введение в лямбда-выражения

Лямбда-выражение (или просто ***«лямбда»***) **в программировании позволяет определить анонимную функцию внутри другой функции**. Возможность сделать функцию вложенной является очень важным преимуществом, так как позволяет избегать как захламления пространства имен лишними объектами, так и определить функцию как можно ближе к месту её первого использования.

Синтаксис лямбда-выражений является одним из самых странных в языке C++, и вам может потребоваться некоторое время, чтобы к нему привыкнуть.

Лямбда-выражения имеют следующий синтаксис:

```
[ captureClause ] ( параметры ) -> возвращаемыйТип
{
стейтменты;
}
```

Поля `captureClause` и `параметры` могут быть пустыми, если они не требуются программисту.

Поле `возвращаемыйТип` является опциональным, и, если его нет, то будет использоваться вывод типа с помощью ключевого слова auto. Хотя мы ранее уже отмечали, что следует избегать использования вывода типа для возвращаемых значений функций, в данном контексте подобное использование допускается (поскольку обычно такие функции являются тривиальными).

Также обратите внимание, что лямбда-выражения не имеют имен, поэтому нам и не нужно будет их предоставлять. Из этого факта следует, что тривиальное определение лямбды может иметь следующий вид:

```
1. #include <iostream>
2.
3. int main()
4. {
5.   []() {}; // определяем лямбда-
   выражение без captureClause, параметров и возвращаемого типа
6.
7.   return 0;
8. }
```

Давайте перепишем предыдущий пример, но уже с использованием лямбда-выражений:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  // Определяем функцию непосредственно в том месте, где собираемся её использовать
  auto found{ std::find_if(arr.begin(), arr.end(),
                           [](std::string_view str) // вот наша лямбда, без поля captureClause
                           {
                             return (str.find("nut") != std::string_view::npos);
                           }) };

  if (found == arr.end())
  {
    std::cout << "No nuts\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
}
```

При этом всё работает точно так же, как и в случае с указателем на функцию. Результат выполнения программы аналогичен:

```
Found walnut
```

Обратите внимание, насколько наша лямбда похожа на функцию containsNut(). Они обе имеют одинаковые параметры и тела функций. Отметим, что у лямбды отсутствует поле `captureClause` (детально о `captureClause` мы поговорим на следующем уроке), т.к. оно не нужно. Также для краткости мы пропустили синтаксис типа возвращаемого значения trailing, но из-за того, что `operator!=` возвращает значение типа bool, наша лямбда также будет возвращать логическое значение.

Тип лямбда-выражений

В примере, приведенном выше, мы определили лямбду прямо в том месте, где она была нам нужна. Такое использование лямбда-выражения иногда еще называют **функциональным литералом**.

Однако написание лямбды в той же строке, где она используется, иногда может затруднить чтение кода. Подобно тому, как мы можем инициализировать переменную с помощью литерала (или указателя на функцию) для использования в дальнейшем, так же мы можем инициализировать и лямбда-переменную с помощью лямбда-определения для её дальнейшего использования. Именованная лямбда вместе с удачным именем функции может облегчить чтение кода.

Например, в следующем фрагменте кода мы используем функцию std::all_of() для того, чтобы проверить, являются ли все элементы массива чётными:

```
1. // Плохо: Мы должны прочитать лямбду, чтобы понять, что происходит
2. return std::all_of(array.begin(), array.end(), [](int i){ return ((i % 2) == 0)
   ; });
```

Мы можем улучшить читабельность кода следующим образом:

```
1. // Хорошо: Мы можем хранить лямбду в именованной переменной и передавать её в
   функцию в качестве параметра
2. auto isEven{
3.   [](int i)
4.   {
5.     return ((i % 2) == 0);
6.   }
7. };
8.
9. return std::all_of(array.begin(), array.end(), isEven);
```

Обратите внимание, как просто читается последняя строка кода: "… возвращаем все элементы массива, которые являются чётными ...".

Но какого типа является лямбда в `isEven`?

Оказывается, у лямбд нет типа, который мы могли бы явно использовать. Когда мы пишем лямбду, компилятор генерирует уникальный тип лямбды, который нам не виден.

> ***Для продвинутых читателей**:* На самом деле, лямбды не являются функциями (что и помогает им избегать ограничений C++, которые накладываются на использование вложенных функций). Лямбды являются особым типом объектов, который называется функтором. **Функторы** — это объекты, содержащие перегруженный operator(), который и делает их вызываемыми подобно обычным функциям.

Хотя мы не знаем тип лямбды, есть несколько способов её хранения для использования после определения. Если лямбда ничего не захватывает, то мы можем использовать обычный указатель на функцию. Как только лямбда что-либо

захватывает, указатель на функцию больше не будет работать. Однако `std::function` может использоваться для лямбд, даже если они что-то захватывают:

```
#include <functional>

int main()
{
  // Обычный указатель на функцию. Лямбда не может ничего захватить
  double (*addNumbers1)(double, double){
    [](double a, double b) {
      return (a + b);
    }
  };

  addNumbers1(1, 2);

  // Используем std::function. Лямбда может захватывать переменные.
  std::function addNumbers2{ // примечание: Если у вас не поддерживается C++17 и выше, используйте std::function<double(double, double)>
    [](double a, double b) {
      return (a + b);
    }
  };

  addNumbers2(3, 4);

  // Используем auto. Храним лямбду с её реальным типом
  auto addNumbers3{
    [](double a, double b) {
      return (a + b);
    }
  };

  addNumbers3(5, 6);

  return 0;
}
```

С помощью auto мы можем использовать фактический тип лямбды. При этом мы можем получить преимущество в виде отсутствия накладных расходов в сравнении с использованием `std::function`.

К сожалению, мы не всегда можем использовать auto. В тех случаях, когда фактический тип лямбды неизвестен (например, из-за того, что мы передаем лямбду в функцию в качестве параметра, и вызывающий объект сам определяет какого типа лямбда будет передана), мы не можем использовать auto. В таких случаях следует использовать `std::function`:

```
#include <functional>
#include <iostream>

// Мы не знаем, чем будет fn. std::function работает с обычными функциями и лямбдами
void repeat(int repetitions, const std::function<void(int)>& fn)
```

```
{
  for (int i{ 0 }; i < repetitions; ++i)
  {
    fn(i);
  }
}

int main()
{
  repeat(3, [](int i) {
    std::cout << i << '\n';
  });

  return 0;
}
```

Результат выполнения программы:

```
0
1
2
```

Правило: Используйте auto при инициализации переменных с помощью лямбд и std::function, если вы не можете инициализировать переменную с помощью лямбд.

Общие/Обобщённые лямбды

По большей части лямбда-параметры работают так же, как и обычные параметры функций.

Одним примечательным исключением является то, что, начиная с C++14, нам разрешено использовать auto с параметрами функций.

> ***Примечание***: В C++20 обычные функции также могут использовать auto с параметрами.

Если у лямбды есть один или несколько параметров auto, то компилятор определит необходимые типы параметров из вызовов лямбд-выражений.

Поскольку лямбды с одним или несколькими параметрами типа auto потенциально могут работать с большим количество типов данных, то они называются **общими** (или ***«обобщёнными»*** от англ. *"generic lambdas"*) **лямбдами**.

Рассмотрим использование общей лямбды на практике:

```
#include <algorithm>
#include <array>
#include <iostream>
```

```
#include <string_view>

int main()
{
  std::array months{ // если у вас не поддерживается C++17, то используйте std::array<std::string_view, 12>
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
  };

  // Поиск двух последовательных месяцев, которые начинаются с одинаковой буквы
  auto sameLetter{ std::adjacent_find(months.begin(), months.end(),
                                      [](const auto& a, const auto& b) {
                                        return (a[0] == b[0]);
                                      }) };

  // Убеждаемся, что эти два месяца были найдены
  if (sameLetter != months.end())
  {
    std::cout << *sameLetter << " and " << *std::next(sameLetter)
              << " start with the same letter\n";
  }

  return 0;
}
```

Результат выполнения программы:

```
June and July start with the same letter
```

В примере, приведенном выше, мы использовали auto-параметры для захвата наших строк с использованием константной ссылки. Т.к. все строковые типы предоставляют доступ к своим отдельным символам через оператор `[]`, то нам не нужно волноваться о том, передает ли пользователь в качестве параметра std::string, строку C-style или что-то другое. Это позволяет нам написать лямбду, которая могла бы принять любой из этих объектов, то есть, если позже мы изменим тип `months`, — нам не придется переписывать лямбду.

Однако auto не всегда является лучшим выбором. Рассмотрим следующую программу:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>
```

```
5.
6.  int main()
7.  {
8.    std::array months{ // если у вас не поддерживается C++17, то используйте
    std::array<const char*, 12>
9.      "January",
10.     "February",
11.     "March",
12.     "April",
13.     "May",
14.     "June",
15.     "July",
16.     "August",
17.     "September",
18.     "October",
19.     "November",
20.     "December"
21.   };
22.
23.   // Подсчитываем количество месяцев с названиями в 5 букв
24.   auto fiveLetterMonths{ std::count_if(months.begin(), months.end(),
25.                                        [](std::string_view str) {
26.                                          return (str.length() == 5);
27.                                        }) };
28.
29.   std::cout << "There are " << fiveLetterMonths << " months with 5 letters\n";

30.
31.   return 0;
32. }
```

Результат выполнения программы:

```
There are 2 months with 5 letters
```

В этом примере использование auto выводит тип `const char*`. Мы знаем, что со строками C-style трудно работать (кроме использования оператора `[]`). Поэтому в данном случае для нас предпочтительнее явно определить тип параметра, как std::string_view, который позволит нам работать с базовыми типами данных намного проще (например, мы можем запросить у представления значение длины строки, даже если пользователь передал в качестве аргумента массив C-style).

Общие лямбды и статические переменные

Следует иметь в виду, что для каждого отдельного типа, выводимого с помощью auto, будет сгенерирована уникальная лямбда. В следующем примере показано, как общая лямбда разделяется на две отдельные:

```
1.  #include <algorithm>
2.  #include <array>
3.  #include <iostream>
4.  #include <string_view>
5.
6.  int main()
```

```
{
  // Выводим значение и подсчитываем, сколько раз будет вызван print
  auto print{
    [](auto value) {
      static int callCount{ 0 };
      std::cout << callCount++ << ": " << value << '\n';
    }
  };

  print("hello"); // 0: hello
  print("world"); // 1: world

  print(1); // 0: 1
  print(2); // 1: 2

  print("ding dong"); // 2: ding dong

  return 0;
}
```

Результат выполнения программы:

```
0: hello
1: world
0: 1
1: 2
2: ding dong
```

В примере, приведенном выше, мы определяем лямбду и затем вызываем её с двумя различными параметрами (строковым литералом и целочисленным типом). При этом генерируются две различные версии лямбды (одна с параметром строкового литерала, а другая — с параметром в виде целочисленного типа).

В большинстве случаев это не существенно. Однако, обратите внимание, что если в общей лямбде используются статические переменные, то эти переменные не являются общими для генерируемых лямбд.

Мы можем видеть это в вышеприведенном примере, где каждый тип (строковые литералы и целые числа) имеет свой собственный уникальный счет! Хотя мы написали лямбду один раз, были сгенерированы две лямбды, и у каждой есть своя версия `callCount`.

Если бы мы хотели, чтобы `callCount` был общим для лямбд, то нам пришлось бы объявить его вне лямбды и захватить его по ссылке, чтобы он мог быть изменен лямбдой.

Вывод возвращаемого типа и возвращаемые типы trailing

Если использовался вывод возвращаемого типа, то возвращаемый тип лямбды выводится из стейтментов return внутри лямбды. Если использовался вывод возвращаемого типа, то все возвращаемые стейтменты внутри лямбды должны возвращать значения одного и того же типа (иначе компилятор не будет знать, какой из них ему следует использовать). Например:

```
#include <iostream>

int main()
{
  auto divide{ [](int x, int y, bool bInteger) { // примечание: Не указан тип возвращаемого значения
    if (bInteger)
      return x / y;
    else
      return static_cast<double>(x) / y; // ОШИБКА: Тип возвращаемого значения не совпадает с предыдущим возвращаемым типом
  } };

  std::cout << divide(3, 2, true) << '\n';
  std::cout << divide(3, 2, false) << '\n';

  return 0;
}
```

Это приведет к ошибке компиляции, так как тип возвращаемого значения первого стейтмента return (`int`) не совпадает с типом возвращаемого значения второго стейтмента return (`double`).

В случае, когда у нас используются разные возвращаемые типы, у нас есть два варианта:

- выполнить явные преобразования в один тип;
- явно указать тип возвращаемого значения для лямбды и позволить компилятору выполнить неявные преобразования.

Второй вариант обычно является более предпочтительным:

```
#include <iostream>

int main()
{
  // Примечание: Явно указываем тип double для возвращаемого значения
  auto divide{ [](int x, int y, bool bInteger) -> double {
    if (bInteger)
      return x / y; // выполнится неявное преобразование в тип double
    else
      return static_cast<double>(x) / y;
  } };

```

```
13.   std::cout << divide(3, 2, true) << '\n';
14.   std::cout << divide(3, 2, false) << '\n';
15.
16.   return 0;
17. }
```

Таким образом, если вы когда-либо решите изменить тип возвращаемого значения, вам (как правило) нужно будет изменить только тип возвращаемого значения лямбды и ничего внутри основной части.

Функциональные объекты Стандартной библиотеки C++

Для основных операций (например, сложения, вычитания или сравнения) вам не нужно писать свои собственные лямбды, потому что Стандартная библиотека C++ поставляется со многими базовыми вызываемыми объектами, которые вы можете использовать. Эти объекты определены в заголовочном файле functional. Например:

```
1. #include <algorithm>
2. #include <array>
3. #include <iostream>
4.
5. bool greater(int a, int b)
6. {
7.   // Размещаем a перед b, если a больше b
8.   return (a > b);
9. }
10.
11.int main()
12.{
13.   std::array arr{ 13, 90, 99, 5, 40, 80 };
14.
15.   // Передаем greater в качестве параметра в std::sort()
16.   std::sort(arr.begin(), arr.end(), greater);
17.
18.   for (int i : arr)
19.   {
20.     std::cout << i << ' ';
21.   }
22.
23.   std::cout << '\n';
24.
25.   return 0;
26.}
```

Результат выполнения программы:

```
99 90 80 40 13 5
```

Вместо преобразования функции greater() в лямбду, мы можем использовать `std::greater`:

```
1. #include <algorithm>
2. #include <array>
3. #include <iostream>
```

```
#include <functional> // для std::greater

int main()
{
  std::array arr{ 13, 90, 99, 5, 40, 80 };

  // Передаем std::greater в качестве параметра в std::sort()
  std::sort(arr.begin(), arr.end(), std::greater{}); // примечание: Требуются фигурные скобки для создания объекта

  for (int i : arr)
  {
    std::cout << i << ' ';
  }

  std::cout << '\n';

  return 0;
}
```

Результат выполнения программы:

```
99 90 80 40 13 5
```

Заключение

Лямбда-выражения и библиотека алгоритмов могут показаться излишне сложными по сравнению с обычными решениями, использующими циклы. Однако эта комбинация позволяет выполнять некоторые очень мощные операции всего в несколько строчек кода и может быть куда читабельнее, чем ваши «самописные» циклы. Кроме того, библиотека алгоритмов обладает мощным и простым в использовании параллелизмом, который вы не получите при использовании циклов. Обновление исходного кода, использующего библиотечные функции, проще, чем обновление кода, использующего циклы.

Лямбды великолепны, но они не заменяют обычные функции для всех случаев. Используйте обычные функции для нетривиальных случаев.

Тест

Задание №1

Создайте структуру `Student`, которая будет хранить имя и баллы студента. Создайте массив студентов и используйте функцию std::max_element() для поиска студента с наибольшими баллами, а затем выведите на экран имя найденного студента. Функция std::max_element() принимает `begin` и `end` списка, и функцию с двумя параметрами, которая возвращает `true`, если первый аргумент меньше второго.

При использовании следующего массива:

```
std::array<Student, 8> arr{
  { { "Albert", 3 },
    { "Ben", 5 },
    { "Christine", 2 },
    { "Dan", 8 }, // Dan имеет больше всего баллов (8).
    { "Enchilada", 4 },
    { "Francis", 1 },
    { "Greg", 3 },
    { "Hagrid", 5 } }
};
```

Результатом выполнения вашей программы должно быть следующее:

```
Dan is the best student
```

Задание №2

Используйте std::sort() и лямбду в следующем коде для сортировки времен года по возрастанию средней температуры:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

struct Season
{
  std::string_view name{};
  double averageTemperature{};
};

int main()
{
  std::array<Season, 4> seasons{
    { { "Spring", 285.0 },
      { "Summer", 296.0 },
      { "Fall", 288.0 },
      { "Winter", 263.0 } }
  };

  /*
   * Используйте std::sort() здесь
   */

  for (const auto& season : seasons)
  {
    std::cout << season.name << '\n';
  }

  return 0;
}
```

Результатом выполнения вашей программы должно быть следующее:

```
Winter
Spring
Fall
Summer
```

Урок №119. Лямбда-захваты

На предыдущем уроке мы рассматривали следующий пример:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  auto found{ std::find_if(arr.begin(), arr.end(),
                           [](std::string_view str)
                           {
                             return (str.find("nut") != std::string_view::npos)
;
                           }) };

  if (found == arr.end())
  {
    std::cout << "No nuts\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
}
```

Давайте изменим его так, чтобы пользователь сам выбирал подстроку для поиска. Это не настолько интуитивно легко, как может показаться на первый взгляд:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>
#include <string>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  // Просим пользователя ввести объект для поиска
  std::cout << "search for: ";

  std::string search{};
  std::cin >> search;

  auto found{ std::find_if(arr.begin(), arr.end(), [](std::string_view str) {
    // Ищем значение переменной search, вместо подстроки "nut"
    return (str.find(search) != std::string_view::npos); // ошибка: Переменная
search недоступна в данной области видимости
  }) };

```

```
22.   if (found == arr.end())
23.   {
24.     std::cout << "Not found\n";
25.   }
26.   else
27.   {
28.     std::cout << "Found " << *found << '\n';
29.   }
30.
31.   return 0;
32. }
```

Данный код не скомпилируется. В отличие от вложенных блоков кода, где любой идентификатор, определенный во внешнем блоке, доступен и во внутреннем, лямбды в языке C++ могут получить доступ только к определенным видам идентификаторов: глобальные идентификаторы, объекты, известные во время компиляции и со статической продолжительностью жизни. Переменная `search` не соответствует ни одному из этих требований, поэтому лямбда не может её увидеть. Вот для этого и существует **лямбда-захват** (англ. ***"capture clause"***).

Введение в лямбда-захваты

Поле `capture clause` используется для того, чтобы предоставить (косвенно) лямбде доступ к переменным из окружающей области видимости, к которым она обычно не имеет доступ. Всё, что нам нужно для этого сделать, так это перечислить в поле `capture clause` объекты, к которым мы хотим получить доступ внутри лямбды. В нашем примере мы хотим предоставить лямбде доступ к значению переменной `search`, поэтому добавляем её в захват:

```
1.  #include <algorithm>
2.  #include <array>
3.  #include <iostream>
4.  #include <string_view>
5.  #include <string>
6.
7.  int main()
8.  {
9.    std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };
10.
11.   std::cout << "search for: ";
12.
13.   std::string search{};
14.   std::cin >> search;
15.
16.   // Захват переменной search
17.   auto found{ std::find_if(arr.begin(), arr.end(), [search](std::string_view
    str) {
18.     return (str.find(search) != std::string_view::npos);
19.   }) };
20.
21.   if (found == arr.end())
22.   {
```

```
23.     std::cout << "Not found\n";
24.   }
25.   else
26.   {
27.     std::cout << "Found " << *found << '\n';
28.   }
29.
30.   return 0;
31. }
```

Теперь пользователь сможет выполнить поиск нужного элемента в массиве:

```
search for: nana
Found banana
```

Суть работы лямбда-захватов

Хотя может показаться, будто в вышеприведенном примере наша лямбда напрямую обращается к значению переменной `search` (относящейся к блоку кода функции main()), но это не так. Да, лямбды могут выглядеть и функционировать как вложенные блоки, но на самом деле они работают немного по-другому, и при этом существует довольно важное отличие.

Когда выполняется лямбда-определение, то для каждой захватываемой переменной внутри лямбды создается клон этой переменной (с идентичным именем). Данные переменные-клоны инициализируются с помощью переменных из внешней области видимости с тем же именем.

Таким образом, в примере, приведенном выше, при создании объекта лямбды, она получает свою собственную переменную-клон с именем `search`. Эта переменная имеет такое же значение, что и переменная `search` из функции main(), поэтому кажется будто мы получаем доступ непосредственно к переменной `search` функции main(), но это не так.

Несмотря на то, что эти переменные-клоны имеют одно и то же имя, их тип может отличаться от типа исходной переменной (об этом чуть позже).

Ключевой момент: Переменные, захваченные лямбдой, являются клонами переменных из внешней области видимости, а не фактическими «внешними» переменными.

> ***Для продвинутых читателей:*** Когда компилятор обнаруживает определение лямбды, он создает для нее определение как для пользовательского объекта. Каждая захваченная переменная становится элементом данных этого объекта. Во время выполнения программы, при обнаружении определения лямбды,

> создается экземпляр объекта лямбды и в этот момент инициализируются члены лямбды.

Захваты переменных и const

По умолчанию переменные захватываются как константные значения. Это означает, что при создании лямбды, она захватывает константную копию переменной из внешней области видимости, что означает, что значения этих переменных лямбда изменить не может. В следующем примере мы захватим переменную `ammo` и попытаемся выполнить декремент:

```
1. #include <iostream>
2.
3. int main()
4. {
5.   int ammo{ 10 };
6.
7.   // Определяем лямбду внутри переменной с именем shoot
8.   auto shoot{
9.     [ammo]() {
10.      // Запрещено, так как переменная ammo была захвачена в виде константной
   копии
11.      --ammo;
12.
13.      std::cout << "Pew! " << ammo << " shot(s) left.\n";
14.    }
15.  };
16.
17.  // Вызов лямбды
18.  shoot();
19.
20.  std::cout << ammo << " shot(s) left\n";
21.
22.  return 0;
23.}
```

В примере, приведенном выше, когда мы захватываем переменную `ammo`, внутри лямбды создается константная переменная с таким же именем и значением. Мы не может изменить её, потому что она имеет спецификатор `const`. Подобная попытка изменения приведет к ошибке компиляции.

Захват по значению

Чтобы разрешить изменения значения переменных, которые были захвачены по значению, мы можем пометить лямбду как `mutable`.

В данном контексте, **ключевое слово mutable** удаляет спецификатор `const` со всех переменных, захваченных по значению:

```
1. #include <iostream>
```

```
2.
3.  int main()
4.  {
5.    int ammo{ 10 };
6.
7.    auto shoot{
8.      // Добавляем ключевое слово mutable после списка параметров
9.      [ammo]() mutable {
10.       // Теперь нам разрешено изменять значение переменной ammo
11.       --ammo;
12.
13.       std::cout << "Pew! " << ammo << " shot(s) left.\n";
14.     }
15.   };
16.
17.   shoot();
18.   shoot();
19.
20.   std::cout << ammo << " shot(s) left\n";
21.
22.   return 0;
23. }
```

Результат выполнения программы:

```
Pew! 9 shot(s) left.
Pew! 8 shot(s) left.
10 shot(s) left
```

Хотя теперь этот код и скомпилируется, но в нем все еще есть логическая ошибка. Какая именно? При вызове лямбда захватила копию переменной `ammo`. Затем, когда лямбда уменьшает значение переменной `ammo` с 10 до 9 и до 8, то, на самом деле, она уменьшает значение копии, а не исходной переменной.

Обратите внимание, что значение переменной `ammo` сохраняется, несмотря на вызовы лямбды.

Захват по ссылке

Подобно тому, как функции могут изменять значения аргументов, передаваемых им по ссылке, мы также можем захватывать переменные по ссылке, чтобы позволить нашей лямбде влиять на значения аргументов.

Чтобы захватить переменную по ссылке, мы должны добавить знак амперсанда (`&`) к имени переменной, которую хотим захватить. В отличие от переменных, которые захватываются по значению, переменные, которые захватываются по ссылке, не являются константными (если только переменная, которую они захватывают, не является изначально `const`). Если вы предпочитаете передавать аргумент функции

по ссылке (например, для типов, не являющихся базовыми), то вместо захвата по значению, предпочтительнее использовать захват по ссылке.

Вот вышеприведенный код, но уже с захватом переменной `ammo` по ссылке:

```
#include <iostream>

int main()
{
  int ammo{ 10 };

  auto shoot{
    // Ключевое слово mutable теперь нам не потребуется
    [&ammo]() { // &ammo означает, что переменная ammo захватывается по ссылке
      // Изменения текущей переменной ammo приведут к изменению переменной ammo из блока main()
      --ammo;

      std::cout << "Pew! " << ammo << " shot(s) left.\n";
    }
  };

  shoot();

  std::cout << ammo << " shot(s) left\n";

  return 0;
}
```

Результат выполнения программы:

```
Pew! 9 shot(s) left.
9 shot(s) left
```

Теперь давайте воспользуемся захватом по ссылке, чтобы подсчитать, сколько сравнений делает алгоритм std::sort() при сортировке массива:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string>

struct Car
{
  std::string make{};
  std::string model{};
};

int main()
{
  std::array<Car, 3> cars{ { { "Volkswagen", "Golf" },
                             { "Toyota", "Corolla" },
                             { "Honda", "Civic" } } };

  int comparisons{ 0 };

  std::sort(cars.begin(), cars.end(),
```

```
    // Захват переменной comparisons по ссылке
    [&comparisons](const auto& a, const auto& b) {
      // Мы захватили переменную comparisons по ссылке, а это означает, что мы можем изменять её без использования спецификатора mutable
      ++comparisons;

      // Сортировка машин по марке
      return (a.make < b.make);
  });

  std::cout << "Comparisons: " << comparisons << '\n';

  for (const auto& car : cars)
  {
    std::cout << car.make << ' ' << car.model << '\n';
  }

  return 0;
}
```

Результат выполнения программы:

```
Comparisons: 2
Honda Civic
Toyota Corolla
Volkswagen Golf
```

Захват нескольких переменных

Мы можем захватить несколько переменных, разделив их запятыми. Мы также можем использовать как захват по значению, так и захват по ссылке:

```
int health{ 33 };
int armor{ 100 };
std::vector<CEnemy> enemies{};

// Захватываем переменные health и armor по значению, а enemies – по ссылке
[health, armor, &enemies](){};
```

Захваты по умолчанию

Необходимость явно перечислять переменные для захвата иногда может быть несколько обременительной. Если вы изменяете свою лямбду, то вы можете забыть добавить или удалить захватываемые переменные. К счастью, есть возможность заручиться помощью компилятора для автоматической генерации списка переменных, которые нам нужно захватить.

Захват по умолчанию захватывает все переменные, упомянутые в лямбде. Если используется захват по умолчанию, то переменные, не упомянутые в лямбде, не будут захвачены.

Чтобы захватить все задействованные переменные по значению, используйте `=` в качестве значения для захвата. Чтобы захватить все задействованные переменные по ссылке, используйте `&` в качестве значения для захвата.

Вот пример использования захвата по умолчанию по значению:

```
#include <array>
#include <iostream>

int main()
{
  std::array areas{ 100, 25, 121, 40, 56 };

  int width{};
  int height{};

  std::cout << "Enter width and height: ";
  std::cin >> width >> height;

  auto found{ std::find_if(areas.begin(), areas.end(),
                           [=](int knownArea) { // выполняется захват по умолчанию по значению переменных width и height
                             return (width * height == knownArea); // потому что они здесь упомянуты
                           }) };

  if (found == areas.end())
  {
    std::cout << "I don't know this area :(\n";
  }
  else
  {
    std::cout << "Area found :)\n";
  }

  return 0;
}
```

Захваты по умолчанию могут быть смешаны с обычными захватами. Вполне допускается захватить некоторые переменные по значению, а другие — по ссылке, но при этом каждая переменная может быть захвачена только один раз:

```
int health{ 33 };
int armor{ 100 };
std::vector<CEnemy> enemies{};

// Захватываем переменные health и armor по значению, а enemies – по ссылке
[health, armor, &enemies](){};

// Захватываем переменную enemies по ссылке, а все остальные – по значению
[=, &enemies](){};

// Захватываем переменную armor по значению, а все остальные – по ссылке
[&, armor](){};

// Запрещено, так как мы уже определили захват по ссылке для всех переменных
[&, &armor](){};
```

```
16.
17. // Запрещено, так как мы уже определили захват по значению для всех переменных
18. [=, armor](){};
19.
20. // Запрещено, так как переменная armor используется дважды
21. [armor, &health, &armor](){};
22.
23. // Запрещено, так как захват по умолчанию должен быть первым элементом в списке захвата
24. [armor, &](){};
```

Определение новых переменных в лямбда-захвате

Допустим, что нам нужно захватить переменную с небольшой модификацией или объявить новую переменную, которая видна только в области видимости лямбды. Мы можем это сделать, определив переменную в лямбда-захвате без указания её типа:

```
1.  #include <array>
2.  #include <iostream>
3.
4.  int main()
5.  {
6.    std::array areas{ 100, 25, 121, 40, 56 };
7.
8.    int width{};
9.    int height{};
10.
11.   std::cout << "Enter width and height: ";
12.   std::cin >> width >> height;
13.
14.   // Мы храним переменную areas, но пользователь ввел width и height.
15.   // Прежде, чем выполнить операцию поиска, мы должны вычислить значение площади (area)
16.   auto found{ std::find_if(areas.begin(), areas.end(),
17.                            // Объявляем новую переменную, которая видима только для лямбды.
18.                            // Тип переменной userArea автоматически выведен как тип int
19.                            [userArea{ width * height }](int knownArea) {
20.                              return (userArea == knownArea);
21.                            }) };
22.
23.   if (found == areas.end())
24.   {
25.     std::cout << "I don't know this area :(\n";
26.   }
27.   else
28.   {
29.     std::cout << "Area found :)\n";
30.   }
31.
32.   return 0;
33. }
```

Переменная `userArea` будет рассчитана только один раз: во время определения лямбды. Вычисляемая площадь хранится в объекте лямбды и одинакова для каждого вызова. Если лямбда имеет модификатор `mutable` и изменяет переменную, которая определена в захвате, то исходное значение переменной будет переопределено.

> ***Совет:*** Инициализируйте переменные в захвате только в том случае, если их значения не являются слишком большими и их тип очевиден. В противном случае, лучше всего определить переменную вне лямбды, а затем захватить её.

Висячие захваченные переменные

Переменные захватываются в точке определения лямбды. Если переменная, захваченная по ссылке, прекращает свое существование до прекращения существования лямбды, то лямбда остается с висячей ссылкой:

```
#include <iostream>
#include <string>

// Функция возвращает лямбду
auto makeWalrus(const std::string& name)
{
  // Захват переменной name по ссылке и возврат лямбды
  return [&]() {
    std::cout << "I am a walrus, my name is " << name << '\n'; // неопределенное поведение
  };
}

int main()
{
  // Создаем новый объект с именем Roofus.
  // sayName является лямбдой, возвращаемой функцией makeWalrus()
  auto sayName{ makeWalrus("Roofus") };

  // Вызов лямбды, которую возвращает функция makeWalrus()
  sayName();

  return 0;
}
```

Вызов функции makeWalrus() создает временный объект std::string из строкового литерала `"Roofus"`. Лямбда в функции makeWalrus() захватывает временную строку по ссылке. Данная строка уничтожается при выполнении возврата makeWalrus(), но при этом лямбда все еще ссылается на нее. Затем, когда мы вызываем sayName(), происходит попытка доступа к висячей ссылке, что чревато неопределенными результатами.

Обратите внимание, это также происходит, если переменная `name` передается в функцию makeWalrus() по значению. Переменная `name` все равно прекратит свое существование в конце работы функции makeWalrus(), и лямбда останется с висячей ссылкой.

> ***Предупреждение***: Будьте особенно осторожны при захвате переменных по ссылке, особенно при указанном захвате по умолчанию по ссылке. Захваченные переменные должны существовать дольше, чем сама лямбда.

Если мы хотим, чтобы захваченная переменная `name` была валидна, когда используется лямбда, то нам нужно захватить данную переменную по значению (либо явно, либо с помощью захвата по умолчанию по значению).

Непреднамеренные копии лямбд

Поскольку лямбды являются объектами, то их можно копировать. В некоторых случаях это может вызвать проблемы. Рассмотрим следующий код:

```
#include <iostream>

int main()
{
  int i{ 0 };

  // Создаем новую лямбду с именем count
  auto count{ [i]() mutable {
    std::cout << ++i << '\n';
  } };

  count(); // обращаемся к count

  auto otherCount{ count }; // создаем копию count

  // Обращаемся к count и к копии count
  count();
  otherCount();

  return 0;
}
```

Результат выполнения программы:

```
1
2
2
```

Вместо вывода `1 2 3` программа дважды выводит число `2`. Создавая объект `otherCount`, как копию объекта `count`, мы копируем его текущее состояние. Значением переменной `i`, принадлежащей объекту `count`, является `1` и значением

переменной `i`, принадлежащей объекту `otherCount`, так же является `1`. Поскольку `otherCount` — это копия `count`, то у каждого объекта имеется своя собственная переменная `i`.

Теперь давайте рассмотрим менее очевидный пример:

```
#include <iostream>
#include <functional>

void invoke(const std::function<void(void)>& fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // Выполняем инкремент и выводим на экран локальную копию переменной i
    auto count{ [i]() mutable {
      std::cout << ++i << '\n';
    } };

    invoke(count);
    invoke(count);
    invoke(count);

    return 0;
}
```

Результат выполнения программы:

```
1
1
1
```

Данный пример демонстрирует возникновение той же проблемы, что и предыдущий пример. Когда с помощью лямбды создается объект `std::function`, то он внутри себя создает копию лямбда-объекта. Таким образом, наш вызов fn() фактически выполняется при использовании копии лямбды, а не самой лямбды.

Если нам нужно передать изменяемую лямбду, и при этом мы хотим избежать непреднамеренного копирования, то есть два варианта решения данной проблемы. Один из них — использовать вместо этого лямбду, не содержащую захватов. В примере, приведеном выше, мы могли бы удалить захват и отслеживать наше состояние, используя статическую локальную переменную. Но статические локальные переменные могут быть трудны для отслеживания и делают наш код менее читабельным. Лучший вариант — это с самого начала не допустить возможности копирования нашей лямбды. Но, поскольку мы не можем повлиять на

реализацию `std::function` (или любой другой функции или объекта из Стандартной библиотеки C++), как мы можем это сделать?

К счастью, C++ предоставляет **тип std::ref** (как часть заголовочного файла functional), который позволяет нам передавать обычный тип, как если бы это была ссылка. Обёртывая нашу лямбду в `std::ref` всякий раз, когда кто-либо пытается сделать копию нашей лямбды, он будет делать копию ссылки, а не фактического объекта.

Вот наш обновленный код с использованием `std::ref`:

```
#include <iostream>
#include <functional>

void invoke(const std::function<void(void)> &fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // Выполняем инкремент и выводим на экран локальную копию переменной i
    auto count{ [i]() mutable {
      std::cout << ++i << '\n';
    } };

    // std::ref(count) гарантирует, что count рассматривается, как ссылка.
    // Таким образом, всё, что пытается скопировать count, фактически является ссылкой, гарантируя тем самым существование только одного объекта count
    invoke(std::ref(count));
    invoke(std::ref(count));
    invoke(std::ref(count));

    return 0;
}
```

Результат выполнения программы:

```
1
2
3
```

Обратите внимание, выходные данные не изменяются, даже если invoke() принимает fn() по значению. std::function не создает копию лямбды, если мы используем `std::ref`.

Правило: Стандартные библиотечные функции могут копировать функциональные объекты (напомним, что лямбды принадлежат к категории функциональных объектов). Если вы хотите использовать лямбду вместе с

изменяемыми захваченными переменными, то передавайте их по ссылке с помощью std::ref.

Тест

Задание №1

Какие из следующих переменных могут использоваться лямбдой в функции main() без их явного захвата?

```
1. int i{};
2. static int j{};
3.
4. int getValue()
5. {
6.   return 0;
7. }
8.
9. int main()
10.{
11.  int a{};
12.  constexpr int b{};
13.  static int c{};
14.  static constexpr int d{};
15.  const int e{};
16.  const int f{ getValue() };
17.  static const int g{};
18.  static const int h{ getValue() };
19.
20.  [](){
21.    // Попробуйте использовать переменные без их явного захвата
22.    a;
23.    b;
24.    c;
25.    d;
26.    e;
27.    f;
28.    g;
29.    h;
30.    i;
31.    j;
32.  }();
33.
34.  return 0;
35.}
```

Задание №2

Что выведет на экран следующая программа? Не запускайте код, а выполните его в уме:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
```

```
{
  std::string favoriteFruit{ "grapes" };

  auto printFavoriteFruit{
    [=]() {
      std::cout << "I like " << favoriteFruit << '\n';
    }
  };

  favoriteFruit = "bananas with chocolate";

  printFavoriteFruit();

  return 0;
}
```

Задание №3

Мы собираемся написать небольшую игру с квадратами чисел.

Суть игры:

- Попросите пользователя ввести 2 числа: первое — стартовое число, которое нужно возвести в квадрат, второе — количество чисел, которые нужно возвести в квадрат.
- Сгенерируйте случайное целое число от `2` до `4` и возведите в квадрат указанное пользователем количество чисел, начиная со стартового.
- Умножьте каждое возведенное в квадрат число на сгенерированное ранее число (от `2` до `4`).
- Пользователь должен вычислить, какие числа были сгенерированы — он указывает свои предположения.
- Программа проверяет, угадал ли пользователь число, и, если угадал — удаляет угаданное число из списка.
- Если пользователь не угадал число, то игра заканчивается, и программа выводит число, которое было ближе всего к окончательному предположению пользователя, но только если последнее предположение не отличалось больше чем на `4` единицы от числа из списка.

Вот первый запуск игры:

```
Start where? 4
How many? 8
I generated 8 square numbers. Do you know what each number is
after multiplying it by 2?
> 32
Nice! 7 number(s) left.
> 72
```

```
Nice! 6 number(s) left.
> 50
Nice! 5 number(s) left.
> 126
126 is wrong! Try 128 next time.
```

Разбираемся:

- Пользователь решил начать с числа `4` и хочет `8` чисел.
- Квадрат каждого числа будет умножен на `2`. Число `2` было выбрано программой случайным образом.
- Программа сгенерировала 8 квадратов чисел, начиная с числа `4`: `16 25 36 49 64 81 100 121`.
- Но при этом каждое число было умножено на `2`, поэтому мы получаем следующие числа: `32 50 72 98 128 162 200 242`.
- Теперь пользователь начинает угадывать. Порядок, в котором вводятся догадки, не имеет значения.
- Число `32` значится в списке.
- Число `72` значится в списке.
- Числа `126` нет в списке, поэтому пользователь проиграл. В списке есть число `128`, которое отличается не более чем на `4` единицы от предположения пользователя, поэтому его мы и выводим в качестве подсказки.

Вот второй запуск игры:

```
Start where? 1
How many? 3
I generated 3 square numbers. Do you know what each number is
after multiplying it by 4?
> 4
Nice! 2 numbers left.
> 16
Nice! 1 numbers left.
> 36
Nice! You found all numbers, good job!
```

Разбираемся:

- Пользователь решил начать с числа `1` и хочет `3` числа.
- Квадрат каждого числа будет умножен на `4`.
- Программа сгенерировала следующие числа: `1 4 9`.
- Умножаем их на `4`: `4 16 36`.
- Пользователь выиграл, угадав все числа.

Вот третий запуск игры:

```
Start where? 2
How many? 2
I generated 2 square numbers. Do you know what each number is after multiplying it by 4?
> 21
21 is wrong!
```

Разбираемся:

- Пользователь решил начать с числа `2` и хочет `2` числа.
- Квадрат каждого числа умножается на `4`.
- Программа сгенерировала следующие числа: `16 36`.
- Пользователь выдвигает предположение — `21`, и проигрывает. `21` не достаточно близко к любому из оставшихся чисел, поэтому число-подсказка не выводится.

> ***Подсказки:***
>
> 1. Используйте `std::find()` для поиска номера в списке.
> 2. Используйте `std::vector::erase()` для удаления элемента, например:
>
> ```
> auto found{ std::find(/* ... */) };
>
> // Убедитесь, что элемент был найден
>
> myVector.erase(found);
> ```
>
> 3. Используйте `std::min_element()` и лямбду, чтобы найти число, наиболее близкое к предположению пользователя. `std::min_element()` работает аналогично `std::max_element()` из теста предыдущего урока.
> 4. Используйте `std::abs()` из заголовочного файла cmath, чтобы вычислить положительную разницу между двумя числами:
>
> ```
> int distance{ std::abs(5 - 3) }; // 2
> ```

Глава №7. Итоговый тест

Еще одна глава пройдена! Впереди самое сердце этого туториала — объектно-ориентированное программирование, мы почти добрались! Осталась всего лишь одна ступенька - итоговый тест.

Теория

Аргументы функций могут передаваться по значению, по ссылке или по адресу. Используйте:

- **передачу по значению** для фундаментальных типов данных и перечислителей;
- **передачу по (константной) ссылке** для структур, классов или в тех случаях, когда нужно, чтобы функция изменяла значение аргумента;
- **передачу по адресу** для указателей или обычных массивов.

В большинстве случаев используется **возврат по значению**, однако **возврат по ссылке** или **по адресу** также может быть полезен при работе с динамически выделенными массивами, структурами или классами. Если используете возврат по ссылке или по адресу, то убедитесь, что не возвращаете чего-то, что выйдет из локальной области видимости.

С помощью **встроенных функций** вызов функции можно заменить на непосредственный код этой функции.

Перегрузка функций позволяет создать несколько функций с одним и тем же именем, но при условии, что параметры этих функций будут разные. Возвращаемое значение не считается параметром.

Параметр по умолчанию - это параметр функции, который имеет значение по умолчанию. Если caller не передает значение для параметра, то будет использоваться значение по умолчанию. У вас может быть несколько параметров со значениями по умолчанию. Они всегда должны находиться справа от обычных параметров. Параметр по умолчанию может быть установлен только в одном месте. Обычно его размещают в предварительном объявлении функции. Если же предварительного объявления нет, то его размещают в определении функции.

Указатели на функции позволяют передать одну функцию в качестве аргумента другой функции.

Динамическая память выделяется из **кучи**.

Стек вызовов отслеживает все активные функции (те, которые были вызваны, но еще не завершены) от начала программы и до текущей точки выполнения. Стек имеет ограниченный размер.

std::vector можно использовать в качестве стека.

Рекурсивная функция - это функция, которая вызывает сама себя. Для всех рекурсивных функций требуется условие завершения.

Синтаксическая ошибка возникает, когда вы пишете код, который нарушает правила грамматики языка C++. Компилятор такие ошибки легко отлавливает.

Семантическая ошибка возникает, когда код синтаксически правильный, но выполняет не то, что нужно программисту. Среди семантических ошибок распространены **логические ошибки** и **ложные предположения**.

Стейтмент assert используется для обнаружения ложных предположений, но его недостаток заключается в том, что при ложном утверждении выполнение программы немедленно прекращается.

Аргументы командной строки позволяют пользователям или другим программам передавать данные в программу при её запуске. Аргументы командной строки всегда являются строками C-style и для использования числовых значений вам нужно будет конвертировать строки в числовые типы данных.

Эллипсис позволяет передать переменную, указывающую на количество всех передаваемых аргументов, в функцию. Однако при таком раскладе игнорируется проверка типов, что крайне нежелательно и может привести к проблемам.

Тест

Задание №1

Напишите прототипы функций для следующих случаев. Используйте const при необходимости.

a) Функция с именем max(), которая принимает два значения типа double и возвращает большее из них.

b) Функция swap(), которая меняет местами два целых числа.

c) Функция getLargestElement(), которая принимает динамически выделенный массив целых чисел и возвращает наибольшее число таким образом, что caller может изменить значение возвращаемого элемента (не забудьте о параметре-длине).

Задание №2

Что не так со следующими программами?

a)

```
1. int& doSomething()
2. {
3.     int array[] = { 1, 3, 5, 7, 9 };
4.     return array[2];
5. }
```

b)

```
1. int sumTo(int value)
2. {
3.     return value + sumTo(value - 1);
4. }
```

c)

```
1. float divide(float a, float b)
2. {
3.     return a / b;
4. }
5.
6. double divide(float a, float b)
7. {
8.     return a / b;
9. }
```

d)

```
1. #include <iostream>
2.
3. int main()
4. {
5.     int array[1000000000];
6.
7.     for (const auto &x: array)
8.         std::cout << x << ' ';
9.
10.    return 0;
11.}
```

e)

```
1. #include <iostream>
```

```
2.
3.  int main(int argc, char *argv[])
4.  {
5.      int times = argv[1];
6.      for (int count = 0; count < times; count++)
7.          std::cout << count << ' ';
8.
9.      return 0;
10. }
```

Задание №3

Лучшим алгоритмом определения того, существует ли значение в отсортированном массиве или нет, является бинарный поиск.

Бинарный поиск работает следующим образом:

- Смотрим на центральный элемент массива.
- Если центральный элемент массива больше элемента, который мы ищем, то всё, что находится справа от центрального элемента — отбрасываем.
- Если центральный элемент меньше элемента, который мы ищем, то отбрасываем всё, что находится слева от центрального элемента.
- Если центральный элемент равен элементу, который мы ищем, то возвращаем индекс этого элемента.
- Если перебрали весь массив и не нашли искомого значения, то возвращаем контрольное значение с выводом `not found`.

Поскольку в каждой итерации мы можем отбрасывать сразу половину массива, то скорость выполнения этого алгоритма достаточно большая. Даже с массивом в миллион элементов для определения того, существует ли конкретное значение в этом массиве или нет, потребуется не более 20 итераций! Однако бинарный поиск работает только в отсортированном массиве.

Изменение массива (например, отбрасывание половины элементов массива) является затратной операцией, поэтому обычно массив не изменяется. Вместо этого используется два целочисленных значения (`min` и `max`) для хранения индексов минимальной и максимальной границ поиска элемента в массиве.

Рассмотрим пример работы этого алгоритма с массивом `{4, 5, 7, 10, 11, 14, 19, 20, 25}` и искомым значением `7`. Сначала `min = 0`, `max = 8`, так как мы перебираем весь массив (всего элементов 9, но индекс последнего элемента равен 8).

- ***Итерация №1**:* Вычисляем среднее значение между `min` (0) и `max` (8), которое равно 4. Элемент №4 имеет значение 11, которое больше нашего искомого значения. Поскольку массив отсортирован, то мы знаем, что все элементы, которые находятся справа от индекса 4 (и индекс 4 тоже) являются больше нашего искомого числа. Поэтому `min` оставляем прежним, а `max` изменяем на 3.
- ***Итерация №2**:* Вычисляем среднее значение между `min` (0) и `max` (3), которое равно 1. Элемент №1 имеет значение 5, которое меньше нашего искомого значения. Поскольку массив отсортирован, то мы знаем, что все элементы, которые находятся слева от индекса 1 (и индекс 1 тоже) — меньше нашего искомого числа. Следовательно, `min` изменяем на 2, а `max` оставляем прежним.
- ***Итерация №3**:* Вычисляем среднее значение между `min` (2) и `max` (3), которое равно 2. Элемент №2 имеет значение 7, которое является нашим искомым значением. Возвращаем элемент №2.

Используя следующий код:

```
1. // array - это массив, в котором мы проводим поиски.
2. // target - это искомое значение.
3. // min - это индекс минимальной границы массива, в котором мы осуществляем
   поиск.
4. // max - это индекс максимальной границы массива, в котором мы осуществляем
   поиск.
5. // Функция binarySearch() должна возвращать индекс искомого значения, если он
   обнаружен. В противном случае, возвращаем -1
6. int binarySearch(int *array, int target, int min, int max)
7. {
8.
9. }
10.
11.int main()
12.{
13.    int array[] = { 4, 7, 9, 13, 15, 19, 22, 24, 28, 33, 37, 41, 43, 47, 50 };
14.
15.    std::cout << "Enter a number: ";
16.    int x;
17.    std::cin >> x;
18.
19.    int index = binarySearch(array, x, 0, 14);
20.
21.    if (array[index] == x)
22.        std::cout << "Good! Your value " << x << " is on position "<< index <<
   " in array!\n";
23.    else
24.        std::cout << "Fail! Your value " << x << " isn't in array!\n";
25.    return 0;
26.}
```

а) Напишите итеративную версию функции binarySearch().

b) Напишите рекурсивную версию функции binarySearch().

Урок №120. Введение в ООП

На уроке №13 мы определили объект в языке C++ как часть памяти, которая используется для хранения значений. Объект с именем называется переменной.

В традиционном программировании (чем мы занимались до этого момента), программа - это набор инструкций для компьютера, которые определяют данные (через объекты), а затем работают с этими данными (через операторы и функции). Объекты и функции, которые работают с этими данными, являются отдельными единицами, которые объединяются для получения программистом желаемого результата. Из-за того, что они являются отдельными единицами, традиционное программирование часто не позволяет использовать интуитивное представление реальности. Это является делом программиста — управлять и соединять свойства (переменные) с поведением (функциями) соответствующим образом, что приводит к созданию следующего кода:

```
1. driveTo(you, work);
```

Так что же тогда является объектно-ориентированным программированием? Для лучшего понимания воспользуемся аналогией. Оглянитесь вокруг, везде находятся объекты: книги, здания, еда и даже вы сами. **Объекты имеют два основных компонента**:

- **свойства** (например, вес, цвет, размер, прочность, форма и т.д.);
- **поведение**, которое они могут проявлять (например, открывать что-либо, делать что-то и т.д.).

Свойства и поведение неотделимы друг от друга.

Объектно-ориентированное программирование (сокр. ***"ООП"***) предоставляет возможность создавать объекты, которые объединяют свойства и поведение в самостоятельный союз, который затем можно многоразово использовать. Это приводит к созданию следующего кода:

```
1. you.driveTo(work);
```

Так не только читабельнее, но и понятнее, кем является объект (`you` - вы) и какое поведение вызывается (`driveTo` - поездка). Вместо того, чтобы сосредоточиться на написании функций, мы концентрируемся на определении объектов, которые имеют четкий набор поведений. Вот почему эта парадигма называется «объектно-ориентированной».

Это позволяет создавать программы модульным способом, что упрощает не только написание и понимание кода, но и обеспечивает более высокий уровень возможности повторного использования этого кода. Объекты также обеспечивают более интуитивный способ работы с данными, позволяя программисту определить, как он будет взаимодействовать с объектами, и как эти объекты будут взаимодействовать с другими объектами.

Обратите внимание, ООП не заменяет традиционные методы программирования. ООП - это дополнительный инструмент управления сложностью.

Объектно-ориентированное программирование также предоставляет несколько других полезных концепций, таких как наследование, инкапсуляция, абстракция и полиморфизм. Мы рассмотрим каждую из этих концепций на соответствующих уроках. Будет много нового материала, но как только вы разберетесь с ООП, вам уже не захочется возвращаться к традиционному программированию.

Обратите внимание, термин «объект» перегружен, он имеет несколько значений, что может вызывать некоторую путаницу. В традиционном программировании, «объект» - это часть памяти для хранения значений. В объектно-ориентированном программировании, «объект» - это тот же объект, что и в традиционном программировании, но который соединяет в себе как свойства, так и способы поведения. С этого момента мы будем использовать термин «объект» в объектно-ориентированном смысле этого слова.

Урок №121. Классы, Объекты и Методы

Хотя язык C++ предоставляет ряд фундаментальных типов данных (например, char, int, long, float, double и т.д.), которых бывает достаточно для решения относительно простых проблем, для решения сложных проблем функционала этих простых типов может не хватать.

Классы

Одной из наиболее полезных особенностей языка C++ является возможность определять собственные типы данных, которые будут лучше соответствовать в решении конкретных проблем. Вы уже видели, как перечисления и структуры могут использоваться для создания собственных пользовательских типов данных. Например, структура для хранения даты:

```
1. struct DateStruct
2. {
3.     int day;
4.     int month;
5.     int year;
6. };
```

Перечисления и структуры - это традиционный (не объектно-ориентированный) мир программирования, в котором мы можем только хранить данные. В C++11 мы можем создать и инициализировать структуру следующим образом:

```
1. DateStruct today { 12, 11, 2018}; // используем uniform-инициализацию
```

Для вывода даты на экран (что может понадобиться выполнить и не раз, и не два) хорошей идеей будет написать отдельную функцию, например:

```
1. #include <iostream>
2.
3. struct DateStruct
4. {
5.     int day;
6.     int month;
7.     int year;
8. };
9.
10. void print(DateStruct &date)
11. {
12.     std::cout << date.day<< "/" << date.month << "/" << date.year;
13. }
14.
15. int main()
16. {
17.     DateStruct today { 12, 11, 2018}; // используем uniform-инициализацию
18.
```

```
    today.day = 18; // используем оператор выбора члена для выбора члена структуры
    print(today);

    return 0;
}
```

Результат выполнения программы:

```
18/11/2018
```

В объектно-ориентированном программировании типы данных могут содержать не только данные, но и функции, которые будут работать с этими данными. Для определения такого типа данных в языке C++ используется **ключевое слово class**. Использование ключевого слова class определяет новый **пользовательский тип данных** - **класс**.

В языке C++ классы очень похожи на структуры, за исключением того, что они обеспечивают гораздо большую мощность и гибкость. Фактически, следующая структура и класс идентичны по функционалу:

```
struct DateStruct
{
    int day;
    int month;
    int year;
};

class DateClass
{
public:
    int m_day;
    int m_month;
    int m_year;
};
```

Единственным существенным отличием здесь является `public` - ключевое слово в классе (о нем мы поговорим детально на соответствующем уроке).

Так же, как и объявление структуры, объявление класса не приводит к выделению какой-либо памяти. Для использования класса нужно объявить переменную этого типа класса:

```
DateClass today { 12, 11, 2018 }; // инициализируем переменную класса DateClass
```

В языке C++ переменная класса называется **экземпляром** (или ***"объектом"***) **класса**. Точно так же, как определение переменной фундаментального типа данных (например, `int x`) приводит к выделению памяти для этой переменной, так же и

создание объекта класса (например, `DateClass today`) приводит к выделению памяти для этого объекта.

Методы классов

Помимо хранения данных, классы могут содержать и функции! Функции, определенные внутри класса, называются **методами**. Методы могут быть определены, как внутри, так и вне класса. Пока что мы будем определять их внутри класса (для простоты), как определить их вне класса — рассмотрим несколько позже.

Класс Date с методом вывода даты:

```
class DateClass
{
public:
    int m_day;
    int m_month;
    int m_year;

    void print() // определяем функцию-член
    {
        std::cout << m_day << "/" << m_month << "/" << m_year;
    }
};
```

Точно так же, как к членам структуры, так и к членам (переменным и функциям) класса доступ осуществляется через **оператор выбора членов** (`.`):

```
#include <iostream>

class DateClass
{
public:
    int m_day;
    int m_month;
    int m_year;

    void print()
    {
        std::cout << m_day << "/" << m_month << "/" << m_year;
    }
};

int main()
{
    DateClass today { 12, 11, 2018 };

    today.m_day = 18; // используем оператор выбора членов для выбора переменной-члена m_day объекта today класса DateClass
    today.print(); // используем оператор выбора членов для вызова метода print() объекта today класса DateClass

    return 0;
```

```
24. }
```

Результат выполнения программы:

```
18/11/2018
```

Обратите внимание, как эта программа похожа на вышеприведенную программу, где используется структура. Однако есть несколько отличий. В версии DateStruct нам нужно было передать переменную структуры непосредственно в функцию print() в качестве параметра. Если бы мы этого не сделали, то функция print() не знала бы, какую переменную структуры DateStruct выводить. Нам тогда бы пришлось явно ссылаться на члены структуры внутри функции.

Методы класса работают несколько иначе: все вызовы методов должны быть связаны с объектом класса. Когда мы вызываем `today.print()`, то мы сообщаем компилятору вызвать метод `print()` объекта `today`.

Рассмотрим определение метода print() еще раз:

```
1. void print() // определяем метод
2. {
3.     std::cout << m_day << "/" << m_month << "/" << m_year;
4. }
```

На что фактически ссылаются `m_day`, `m_month` и `m_year`? Они ссылаются на связанный объект `today` (который определен caller-ом).

Поэтому, при вызове `today.print()`, компилятор интерпретирует:

- `m_day`, как `today.m_day`;
- `m_month`, как `today.m_month`;
- `m_year`, как `today.m_year`.

Если бы мы вызвали `tomorrow.print()`, то `m_day` ссылался бы на `tomorrow.m_day`.

По сути, связанный объект неявно передается методу. По этой причине его часто называют **неявным объектом**.

Детально о том, как передается неявный объект методу, мы поговорим на соответствующем уроке. Ключевым моментом здесь является то, что для работы с функциями, не являющимися членами класса, нам нужно передавать данные в эту функцию явно (в качестве параметров). А для работы с методами у нас всегда есть неявный объект класса!

Использование префикса `m_` (англ. *"**m**" = "**m**embers"*) для переменных-членов помогает различать переменные-члены от параметров функции или локальных переменных внутри методов класса. Это полезно по нескольким причинам:

- во-первых, когда мы видим переменную с префиксом `m_`, то мы понимаем, что работаем с переменной-членом класса;
- во-вторых, в отличие от параметров функции или локальных переменных, объявленных внутри функции, переменные-члены объявляются в определении класса. Следовательно, если мы хотим знать, как объявлена переменная с префиксом `m_`, мы понимаем, что искать нужно в определении класса, а не внутри функции.

Обычно программисты пишут имена классов с заглавной буквы.

Правило: Пишите имена классов с заглавной буквы.

Вот еще один пример программы с использованием класса:

```
#include <iostream>
#include <string>

class Employee
{
public:
    std::string m_name;
    int m_id;
    double m_wage;

    // Метод вывода информации о работнике на экран
    void print()
    {
        std::cout << "Name: " << m_name <<
                "\nId: " << m_id <<
                "\nWage: $" << m_wage << '\n';
    }
};

int main()
{
    // Определяем двух работников
    Employee john { "John", 5, 30.00 };
    Employee max { "Max", 6, 32.75 };

    // Выводим информацию о работниках на экран
    john.print();
    std::cout<<std::endl;
    max.print();

    return 0;
}
```

Результат выполнения программы:

```
Name: John
Id: 5
Wage: $30

Name: Max
Id: 6
Wage: $32.75
```

В отличие от обычных функций, порядок, в котором определены методы класса, не имеет значения!

Примечание о структурах в C++

В языке Си структуры могут только хранить данные и не могут иметь связанных методов. После проектирования классов (используя ключевое слово class) в языке C++, Бьёрн Страуструп размышлял о том, нужно ли, чтобы структуры (которые были унаследованы из языка Си) имели связанные методы. После некоторых размышлений он решил, что нужно. Поэтому в программах, приведенных выше, мы также можем использовать ключевое слово struct, вместо class, и всё будет работать!

Многие разработчики (включая и меня) считают, что это было неправильное решение, поскольку оно может привести к проблемам, например, справедливо предположить, что класс выполнит очистку памяти после себя (например, класс, которому выделена память, освободит её непосредственно перед моментом уничтожения самого класса), но предполагать то же самое при работе со структурами — небезопасно. Следовательно, рекомендуется использовать ключевое слово struct для структур, используемых только для хранения данных, и ключевое слово class для определения объектов, которые требуют объединения как данных, так и функций.

Правило: Используйте ключевое слово struct для структур, используемых только для хранения данных. Используйте ключевое слово class для объектов, объединяющих как данные, так и функции.

Заключение

Оказывается, Стандартная библиотека C++ полна классов, созданных для нашего удобства. std::string, std::vector и std::array — это всё типы классов! Поэтому, когда

вы создаете объект любого из этих типов, вы создаете объект класса. А когда вы вызываете функцию с использованием этих объектов, вы вызываете метод:

```
#include <string>
#include <array>
#include <vector>
#include <iostream>

int main()
{
    std::string s { "Hello, world!" }; // создаем экземпляр класса string
    std::array<int, 3> a { 7, 8, 9 }; // создаем экземпляр класса array
    std::vector<double> v { 1.5, 2.6, 3.7 }; // создаем экземпляр класса vector

    std::cout << "length: " << s.length() << '\n'; // вызываем метод

    return 0;
}
```

Ключевое слово class позволяет создать пользовательский тип данных в языке C++, который может содержать как переменные-члены, так и методы. Классы — это основа объектно-ориентированного программирования!

Тест

Задание №1

Создайте класс Numbers, который содержит два целых числа. Этот класс должен иметь две переменные-члены для хранения этих двух целых чисел. Вы также должны создать два метода:

- метод set(), который позволит присваивать значения переменным;
- метод print(), который будет выводить значения переменных.

Выполнение следующей функции main():

```
int main()
{
    Numbers n1;
    n1.set(3, 3); // инициализируем объект n1 значениями 3 и 3

    Numbers n2{ 4, 4 }; // инициализируем объект n2 значениями 4 и 4

    n1.print();
    n2.print();

    return 0;
}
```

Должно выдавать следующий результат:

```
Numbers(3, 3)
Numbers(4, 4)
```

Задание №2

Почему для Numbers должен использоваться класс, а не структура?

Урок №122. Спецификаторы доступа public и private

Рассмотрим следующую программу:

```
1. struct DateStruct // члены структуры являются открытыми по умолчанию
2. {
3.     int day; // открыто по умолчанию, доступ имеет любой объект
4.     int month; // открыто по умолчанию, доступ имеет любой объект
5.     int year; // открыто по умолчанию, доступ имеет любой объект
6. };
7.
8. int main()
9. {
10.     DateStruct date;
11.     date.day = 12;
12.     date.month = 11;
13.     date.year = 2018;
14.
15.     return 0;
16. }
```

Здесь мы объявляем структуру DateStruct, а затем напрямую обращаемся к её членам для их инициализации. Это работает, так как все члены структуры являются открытыми по умолчанию. **Открытые члены** (или ***"члены public"***) — это члены структуры или класса, к которым можно получить доступ извне этой же структуры или класса. В программе, приведенной выше, функция main() находится вне структуры, но она может напрямую обращаться к членам `day`, `month` и `year`, так как они являются открытыми.

С другой стороны, рассмотрим следующий почти идентичный класс:

```
1. class DateClass // члены класса являются закрытыми по умолчанию
2. {
3.     int m_day; // закрыто по умолчанию, доступ имеют только другие члены класса
4.     int m_month; // закрыто по умолчанию, доступ имеют только другие члены
   класса
5.     int m_year; // закрыто по умолчанию, доступ имеют только другие члены
   класса
6. };
7.
8. int main()
9. {
10.     DateClass date;
11.     date.m_day = 12; // ошибка
12.     date.m_month = 11; // ошибка
13.     date.m_year = 2018; // ошибка
14.
15.     return 0;
16. }
```

Вам бы не удалось скомпилировать эту программу, так как все члены класса являются закрытыми по умолчанию. **Закрытые члены** (или ***"члены private"***) — это

члены класса, доступ к которым имеют только другие члены этого же класса. Поскольку функция main() не является членом DateClass, то она и не имеет доступа к закрытым членам объекта `date`.

Хотя члены класса являются закрытыми по умолчанию, мы можем сделать их открытыми, используя **ключевое слово public**:

```
1. class DateClass
2. {
3. public: // обратите внимание на ключевое слово public и двоеточие
4.     int m_day; // открыто, доступ имеет любой объект
5.     int m_month; // открыто, доступ имеет любой объект
6.     int m_year; // открыто, доступ имеет любой объект
7. };
8.
9. int main()
10.{
11.    DateClass date;
12.    date.m_day = 12; // ок, так как m_day имеет спецификатор доступа public
13.    date.m_month = 11; // ок, так как m_month имеет спецификатор доступа public
14.    date.m_year = 2018; // ок, так как m_year имеет спецификатор доступа public
15.
16.    return 0;
17.}
```

Поскольку теперь члены класса DateClass являются открытыми, то к ним можно получить доступ напрямую из функции main().

Ключевое слово public вместе с двоеточием называется спецификатором доступа. **Спецификатор доступа** определяет, кто имеет доступ к членам этого спецификатора. Каждый из членов «приобретает» уровень доступа в соответствии со спецификатором доступа (или, если он не указан, в соответствии со спецификатором доступа по умолчанию).

В языке C++ есть 3 уровня доступа:

- **спецификатор public** делает члены открытыми;
- **спецификатор private** делает члены закрытыми;
- **спецификатор protected** открывает доступ к членам только для дружественных и дочерних классов (детально об этом на соответствующем уроке).

Использование спецификаторов доступа

Классы могут использовать (и активно используют) сразу несколько спецификаторов доступа для установки уровней доступа для каждого из своих членов. Обычно

переменные-члены являются закрытыми, а методы — открытыми. Почему именно так? Об этом мы поговорим на следующем уроке.

Правило: Устанавливайте спецификатор доступа private переменным-членам класса и спецификатор доступа public — методам класса (если у вас нет веских оснований делать иначе).

Рассмотрим пример класса, который использует спецификаторы доступа private и public:

```
#include <iostream>

class DateClass // члены класса являются закрытыми по умолчанию
{
    int m_day; // закрыто по умолчанию, доступ имеют только другие члены класса
    int m_month; // закрыто по умолчанию, доступ имеют только другие члены класса
    int m_year; // закрыто по умолчанию, доступ имеют только другие члены класса

public:
    void setDate(int day, int month, int year) // открыто, доступ имеет любой объект
    {
        // Метод setDate() имеет доступ к закрытым членам класса, так как сам является членом класса
        m_day = day;
        m_month = month;
        m_year = year;
    }

    void print() // открыто, доступ имеет любой объект
    {
        std::cout << m_day << "/" << m_month << "/" << m_year;
    }
};

int main()
{
    DateClass date;
    date.setDate(12, 11, 2018); // ок, так как setDate() имеет спецификатор доступа public
    date.print(); // ок, так как print() имеет спецификатор доступа public

    return 0;
}
```

Результат выполнения программы:

```
12/11/2018
```

Обратите внимание, хоть мы и не можем получить доступ к переменным-членам объекта `date` напрямую из main() (так как они являются private по умолчанию), мы можем получить доступ к ним через открытые методы setDate() и print()!

Открытые члены классов составляют **открытый** (или ***"public"***) **интерфейс**. Поскольку доступ к открытым членам класса может осуществляться извне класса, то открытый интерфейс и определяет, как программы, использующие класс, будут взаимодействовать с этим же классом.

Некоторые программисты предпочитают сначала перечислить private-члены, а затем уже public-члены. Они руководствуются следующей логикой: public-члены обычно используют private-члены (те же переменные-члены в методах класса), поэтому имеет смысл сначала определять private-члены, а затем уже public-члены. Другие же программисты считают, что сначала нужно указывать public-члены. Здесь уже иная логика: поскольку private-члены закрыты и получить к ним доступ напрямую нельзя, то и выносить их на первое место тоже не нужно. Работать будет и так, и так. Какой способ использовать — выбирайте сами, что вам удобнее.

Рассмотрим следующую программу:

```
#include <iostream>

class DateClass // члены класса являются закрытыми по умолчанию
{
    int m_day; // закрыто по умолчанию, доступ имеют только другие члены класса
    int m_month; // закрыто по умолчанию, доступ имеют только другие члены класса
    int m_year; // закрыто по умолчанию, доступ имеют только другие члены класса

public:
    void setDate(int day, int month, int year)
    {
        m_day = day;
        m_month = month;
        m_year = year;
    }

    void print()
    {
        std::cout << m_day << "/" << m_month << "/" << m_year;
    }

    // Обратите внимание на этот дополнительный метод
    void copyFrom(const DateClass &b)
    {
        // Мы имеем прямой доступ к закрытым членам объекта b
        m_day = b.m_day;
        m_month = b.m_month;
        m_year = b.m_year;
    }
};

int main()
{
    DateClass date;
    date.setDate(12, 11, 2018); // ок, так как setDate() имеет спецификатор доступа public
```

```
36.
37.     DateClass copy;
38.     copy.copyFrom(date); // ок, так как copyFrom() имеет спецификатор доступа
    public
39.     copy.print();
40.
41.     return 0;
42. }
```

Один нюанс в языке C++, который часто игнорируют/забывают/неправильно понимают, заключается в том, что **контроль доступа работает на основе класса**, а не на основе объекта. Это означает, что, когда метод имеет доступ к закрытым членам класса, он может обращаться к закрытым членам любого объекта этого класса.

В примере, приведенном выше, метод copyFrom() является членом класса DateClass, что открывает ему доступ к private-членам класса DateClass. Это означает, что copyFrom() может не только напрямую обращаться к закрытым членам неявного объекта с которым работает (копия объекта), но и имеет прямой доступ к закрытым членам объекта `b` класса DateClass!

Это полезно, когда нужно скопировать элементы из одного объекта класса в другой объект того же класса. Детально об этом мы поговорим на следующих уроках.

Структуры vs. Классы

Теперь, когда мы узнали о спецификаторах доступа, мы можем поговорить о фактических различиях между классом и структурой в языке C++. **Класс по умолчанию устанавливает всем своим членам спецификатор доступа private. Структура же по умолчанию устанавливает всем своим членам спецификатор доступа public**.

Есть еще одно незначительное отличие: **структуры наследуют от других конструкций языка C++ открыто, в то время как классы наследуют закрыто**.

Тест

Задание №1

a) Что такое открытый член?

b) Что такое закрытый член?

c) Что такое спецификатор доступа?

d) Сколько есть спецификаторов доступа в языке C++? Назовите их.

Задание №2

a) Напишите простой класс с именем Numbers. Этот класс должен иметь:

- три закрытые переменные-члены типа double: `m_a`, `m_b` и `m_c`;
- открытый метод с именем setValues(), который позволит устанавливать значения для `m_a`, `m_b` и `m_c`;
- открытый метод с именем print(), который будет выводить объект класса Numbers в следующем формате: `<m_a, m_b, m_c>`.

Следующий код функции main():

```
int main()
{
    Numbers point;
    point.setValues(3.0, 4.0, 5.0);

    point.print();

    return 0;
}
```

Должен выдавать следующий результат:

```
<3, 4, 5>
```

b) Добавьте функцию isEqual() в класс Numbers, чтобы следующий код работал корректно:

```
int main()
{
    Numbers point1;
    point1.setValues(3.0, 4.0, 5.0);

    Numbers point2;
    point2.setValues(3.0, 4.0, 5.0);

    if (point1.isEqual(point2))
        std::cout << "point1 and point2 are equal\n";
    else
        std::cout << "point1 and point2 are not equal\n";

    Numbers point3;
    point3.setValues(7.0, 8.0, 9.0);

    if (point1.isEqual(point3))
        std::cout << "point1 and point3 are equal\n";
    else
        std::cout << "point1 and point3 are not equal\n";

    return 0;
```

```
23. }
```

Задание №3

Теперь попробуем что-то посложнее. Напишите класс, который реализует функционал стека. Класс Stack должен иметь:

- закрытый целочисленный фиксированный массив длиной 10 элементов;
- закрытое целочисленное значение для отслеживания длины стека;
- открытый метод с именем reset(), который будет инициализировать значением `0` длину и все значения элементов;
- открытый метод с именем push(), который будет добавлять значение в стек. Метод push() должен возвращать значение `false`, если массив уже заполнен, в противном случае — `true`;
- открытый метод с именем pop() для возврата значений из стека. Если в стеке нет значений, то должен выводиться стейтмент assert.
- открытый метод с именем print(), который будет выводить все значения стека.

Следующий код функции main():

```
1. int main()
2. {
3.     Stack stack;
4.     stack.reset();
5. 
6.     stack.print();
7. 
8.     stack.push(3);
9.     stack.push(7);
10.    stack.push(5);
11.    stack.print();
12. 
13.    stack.pop();
14.    stack.print();
15. 
16.    stack.pop();
17.    stack.pop();
18. 
19.    stack.print();
20. 
21.    return 0;
22. }
```

Должен выдавать следующий результат:

```
( )
( 3 7 5 )
( 3 7 )
( )
```

Урок №123. Инкапсуляция, Геттеры и Сеттеры

На предыдущем уроке мы узнали, что переменные-члены класса по умолчанию являются закрытыми. Новички, которые изучают объектно-ориентированное программирование, очень часто не понимают, почему всё обстоит именно так.

Зачем делать переменные-члены класса закрытыми?

В качестве ответа, воспользуемся аналогией. В современной жизни мы имеем доступ ко многим электронным устройствам. К телевизору есть пульт дистанционного управления, с помощью которого можно включать/выключать телевизор. Управление автомобилем позволяет в разы быстрее передвигаться между двумя точками. С помощью фотоаппарата можно делать снимки.

Все эти 3 вещи используют общий шаблон: они предоставляют вам простой интерфейс (кнопка, руль и т.д.) для выполнения определенного действия. Однако, то, как эти устройства фактически работают, скрыто от вас (как от пользователей). Для нажатия кнопки на пульте дистанционного управления вам не нужно знать, что выполняется «под капотом» пульта для взаимодействия с телевизором. Когда вы нажимаете на педаль газа в своем автомобиле, вам не нужно знать о том, как двигатель внутреннего сгорания приводит в движение колеса. Когда вы делаете снимок, вам не нужно знать, как датчики собирают свет в пиксельное изображение.

Такое разделение интерфейса и реализации чрезвычайно полезно, поскольку оно позволяет использовать объекты, без необходимости понимания их реализации. Это значительно снижает сложность использования этих устройств и значительно увеличивает их количество (устройства с которыми можно взаимодействовать).

По аналогичным причинам разделение реализации и интерфейса полезно и в программировании.

Инкапсуляция

В объектно-ориентированном программировании **инкапсуляция** (или ***"сокрытие информации"***) - это процесс скрытого хранения деталей реализации объекта. Пользователи обращаются к объекту через открытый интерфейс.

В языке C++ инкапсуляция реализована через спецификаторы доступа. Как правило, все переменные-члены класса являются закрытыми (скрывая детали реализации), а большинство методов являются открытыми (с открытым интерфейсом для пользователя). Хотя требование к пользователям использовать публичный

интерфейс может показаться более обременительным, нежели просто открыть доступ к переменным-членам, но на самом деле это предоставляет большое количество полезных преимуществ, которые улучшают возможность повторного использования кода и его поддержку.

Преимущество №1: **Инкапсулированные классы проще в использовании и уменьшают сложность ваших программ.**

С полностью инкапсулированным классом вам нужно знать только то, какие методы являются доступными для использования, какие аргументы они принимают и какие значения возвращают. Не нужно знать, как класс реализован изнутри. Например, класс, содержащий список имен, может быть реализован с использованием динамического массива, строк C-style, std::array, std::vector, std::map, std::list или любой другой структуры данных. Для использования этого класса, вам не нужно знать детали его реализации. Это значительно снижает сложность ваших программ, а также уменьшает количество возможных ошибок. Это является ключевым преимуществом инкапсуляции.

Все классы Стандартной библиотеки C++ инкапсулированы. Представьте, насколько сложнее был бы процесс изучения языка C++, если бы вам нужно было знать реализацию std::string, std::vector или std::cout (и других объектов) для того, чтобы их использовать!

Преимущество №2: **Инкапсулированные классы помогают защитить ваши данные и предотвращают их неправильное использование.**

Глобальные переменные опасны, так как нет строгого контроля над тем, кто имеет к ним доступ и как их используют. Классы с открытыми членами имеют ту же проблему, только в меньших масштабах. Например, допустим, что нам нужно написать строковый класс. Мы могли бы начать со следующего:

```
1. class MyString
2. {
3.     char *m_string; // динамически выделяем строку
4.     int m_length; // используем переменную для отслеживания длины строки
5. };
```

Эти два члена связаны: `m_length` всегда должен соответствовать длине строки, удерживаемой `m_string`. Если бы `m_length` был открытым, то любой мог бы изменить длину строки без изменения `m_string` (или наоборот). Это точно привело бы к проблемам. Делая как `m_length`, так и `m_string` закрытыми, пользователи вынуждены использовать методы для взаимодействия с классом.

Мы также можем сами улучшить защиту нашего класса от неправильного использования. Например, рассмотрим класс с открытой переменной-членом в виде массива:

```
1. class IntArray
2. {
3. public:
4.     int m_array[10];
5. };
```

Если бы пользователи могли напрямую обращаться к массиву, то они могли бы использовать недопустимый индекс:

```
1. int main()
2. {
3.     IntArray array;
4.     array.m_array[16] = 2; // некорректный индекс, вследствие чего
   перезаписываем память, которой мы не владеем
5. }
```

Однако, если мы сделаем массив закрытым, то сможем заставить пользователя использовать функцию, которая первым делом проверяет корректность индекса:

```
1. class IntArray
2. {
3. private:
4.     int m_array[10]; // пользователь не имеет прямого доступа к этому члену
5.
6. public:
7.     void setValue(int index, int value)
8.     {
9.         // Если индекс недействителен, то не делаем ничего
10.        if (index < 0 || index >= 10)
11.            return;
12.
13.        m_array[index] = value;
14.    }
15. };
```

Таким образом, мы защитим целостность нашей программы.

> ***Примечание***: Функция at() в std::array и std::vector делает что-то похожее!

***Преимущество №3:* Инкапсулированные классы легче изменить.**

Рассмотрим следующий простой пример:

```
1. #include <iostream>
2.
3. class Values
4. {
5. public:
6.     int m_number1;
```

```
7.      int m_number2;
8.      int m_number3;
9.  };
10.
11. int main()
12. {
13.     Values value;
14.     value.m_number1 = 7;
15.     std::cout << value.m_number1 << '\n';
16. }
```

Хотя эта программа работает нормально, но что произойдет, если мы решим переименовать `m_number1` или изменить тип этой переменной? Мы бы сломали не только эту программу, но и большую часть программ, которые используют класс Values!

Инкапсуляция предоставляет возможность изменения способа реализации классов, не нарушая при этом работу всех программ, которые их используют. Вот инкапсулированная версия класса, приведенного выше, но которая использует методы для доступа к `m_number1`:

```
1.  #include <iostream>
2.
3.  class Values
4.  {
5.  private:
6.      int m_number1;
7.      int m_number2;
8.      int m_number3;
9.
10. public:
11.     void setNumber1(int number) { m_number1 = number; }
12.     int getNumber1() { return m_number1; }
13. };
14.
15. int main()
16. {
17.     Values value;
18.     value.setNumber1(7);
19.     std::cout << value.getNumber1() << '\n';
20. }
```

Теперь давайте изменим реализацию класса:

```
1.  #include <iostream>
2.
3.  class Values
4.  {
5.  private:
6.      int m_number[3]; // здесь изменяем реализацию этого класса
7.
8.  public:
9.      // Нам нужно обновить переменные методов, чтобы заработала новая реализация
10.     void setNumber1(int number) { m_number[0] = number; }
11.     int getNumber1() { return m_number[0]; }
12. };
```

```
13.
14. int main()
15. {
16.     // Но наша программа продолжает работать как прежде
17.     Values value;
18.     value.setNumber1(7);
19.     std::cout << value.getNumber1() << '\n';
20. }
```

Обратите внимание, поскольку мы не изменяли прототипы каких-либо функций в открытом интерфейсе нашего класса, наша программа, использующая класс, продолжает работать без каких-либо изменений или проблем.

Аналогично, если бы ночью гномы пробрались в ваш дом и заменили внутреннюю часть вашего пульта от телевизора на другую (совместимую) технологию, вы, вероятно, даже не заметили бы ничего!

***Преимущество №4:* С инкапсулированными классами легче проводить отладку.**

И, наконец, инкапсуляция помогает проводить отладку программ, когда что-то идет не по плану. Часто причиной неправильной работы программы является некорректное значение одной из переменных. Если каждый объект имеет прямой доступ к переменной, то отследить часть кода, которая изменила переменную, может быть довольно-таки трудно. Однако, если для изменения значения нужно вызывать один и тот же метод, вы можете просто использовать точку останова для этого метода и посмотреть, как каждый вызывающий объект изменяет значение, пока не увидите что-то странное.

Функции доступа (геттеры и сеттеры)

В зависимости от класса, может быть уместным (в контексте того, что делает класс) иметь возможность получать/устанавливать значения закрытым переменным-членам класса.

Функция доступа - это короткая открытая функция, задачей которой является получение или изменение значения закрытой переменной-члена класса. Например:

```
1. class MyString
2. {
3. private:
4.     char *m_string; // динамически выделяем строку
5.     int m_length; // используем переменную для отслеживания длины строки
6.
7. public:
8.     int getLength() { return m_length; } // функция доступа для получения
   значения m_length
9. };
```

Здесь getLength() является функцией доступа, которая просто возвращает значение `m_length`.

Функции доступа обычно бывают двух типов:

- **геттеры** - это функции, которые возвращают значения закрытых переменных-членов класса;
- **сеттеры** - это функции, которые позволяют присваивать значения закрытым переменным-членам класса.

Вот пример класса, который использует геттеры и сеттеры для всех своих закрытых переменных-членов:

```
1. class Date
2. {
3. private:
4.     int m_day;
5.     int m_month;
6.     int m_year;
7.
8. public:
9.     int getDay() { return m_day; } // геттер для day
10.    void setDay(int day) { m_day = day; } // сеттер для day
11.
12.    int getMonth() { return m_month; } // геттер для month
13.    void setMonth(int month) { m_month = month; } // сеттер для month
14.
15.    int getYear() { return m_year; } // геттер для year
16.    void setYear(int year) { m_year = year; } // сеттер для year
17.};
```

В этом классе нет никаких проблем с тем, чтобы пользователь мог напрямую получать или присваивать значения закрытым переменным-членам этого класса, так как есть полный набор геттеров и сеттеров. В примере с классом MyString для переменной `m_length` не было предоставлено сеттера, так как не было необходимости в том, чтобы пользователь мог напрямую устанавливать длину.

Правило: Предоставляйте функции доступа только в том случае, когда нужно, чтобы пользователь имел возможность получать или присваивать значения членам класса.

Хотя иногда вы можете увидеть, что геттер возвращает неконстантную ссылку на переменную-член - этого следует избегать, так как в таком случае нарушается инкапсуляция, позволяя caller-у изменять внутреннее состояние класса вне этого же класса. Лучше, чтобы ваши геттеры использовали тип возврата по значению или по константной ссылке.

Правило: Геттеры должны использовать тип возврата по значению или по константной ссылке. Не используйте для геттеров тип возврата по неконстантной ссылке.

Заключение

Инкапсуляция имеет много преимуществ, основное из которых заключается в использовании класса без необходимости понимания его реализации. Инкапсулированные классы:

- проще в использовании;
- уменьшают сложность программы;
- защищают данные и предотвращают их неправильное использование;
- легче изменять;
- легче отлаживать.

Это значительно облегчает использование классов.

Урок №124. Конструкторы

Когда все члены класса (или структуры) являются открытыми, то мы можем инициализировать класс (или структуру) напрямую, используя список инициализаторов или uniform-инициализацию (в C++11):

```
1. class Boo
2. {
3. public:
4.     int m_a;
5.     int m_b;
6. };
7.
8. int main()
9. {
10.    Boo boo1 = { 7, 8 }; // список инициализаторов
11.    Boo boo2 { 9, 10 }; // uniform-инициализация (C++11)
12.
13.    return 0;
14.}
```

Однако, как только мы сделаем какие-либо переменные-члены класса закрытыми, то больше не сможем инициализировать их напрямую. Здесь есть смысл: если вы не можете напрямую обращаться к переменной (потому что она закрыта), то вы и не должны иметь возможность напрямую её инициализировать.

Как тогда инициализировать класс с закрытыми переменными-членами? Использовать конструкторы.

Конструктор - это особый тип метода класса, который автоматически вызывается при создании объекта этого же класса. Конструкторы обычно используются для инициализации переменных-членов класса значениями, которые предоставлены по умолчанию/пользователем, или для выполнения любых шагов настройки, необходимых для используемого класса (например, открыть определенный файл или базу данных).

В отличие от обычных методов, **конструкторы имеют определенные правила их именования**:

- конструкторы всегда должны иметь то же имя, что и класс (учитываются верхний и нижний регистры);
- конструкторы не имеют типа возврата (даже void).

Обратите внимание, конструкторы предназначены только для выполнения инициализации. Не следует пытаться вызывать конструктор для повторной инициализации существующего объекта. Хотя это может скомпилироваться без

ошибок, результаты могут получиться неожиданные (компилятор создаст временный объект, а затем удалит его).

Конструкторы по умолчанию

Конструктор, который не имеет параметров (или содержит параметры, которые все имеют значения по умолчанию), называется **конструктором по умолчанию**. Он вызывается, если пользователем не указаны значения для инициализации. Например:

```
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction() // конструктор по умолчанию
    {
        m_numerator = 0;
        m_denominator = 1;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};

int main()
{
    Fraction drob; // так как нет никаких аргументов, то вызывается конструктор по умолчанию Fraction()
    std::cout << drob.getNumerator() << "/" << drob.getDenominator() << '\n';

    return 0;
}
```

Этот класс содержит дробь в виде отдельных значений типа int. Конструктор по умолчанию называется Fraction (как и класс). Поскольку мы создали объект класса Fraction без аргументов, то конструктор по умолчанию сработал сразу же после выделения памяти для объекта, и инициализировал наш объект.

Результат выполнения программы:

```
0/1
```

Обратите внимание, наш числитель (`m_numerator`) и знаменатель (`m_denominator`) были инициализированы значениями, которые мы задали в конструкторе по умолчанию! Это настолько полезная особенность, что почти

каждый класс имеет свой конструктор по умолчанию. Без него значениями нашего числителя и знаменателя был бы мусор до тех пор, пока мы явно не присвоили бы им нормальные значения.

Конструкторы с параметрами

Хотя конструктор по умолчанию отлично подходит для обеспечения инициализации наших классов значениями по умолчанию, часто может быть нужно, чтобы экземпляры нашего класса имели определенные значения, которые мы предоставим позже. К счастью, конструкторы также могут быть объявлены с параметрами. Вот пример конструктора, который имеет два целочисленных параметра, которые используются для инициализации числителя и знаменателя:

```
#include <cassert>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction() // конструктор по умолчанию
    {
        m_numerator = 0;
        m_denominator = 1;
    }

    // Конструктор с двумя параметрами, один из которых имеет значение по умолчанию
    Fraction(int numerator, int denominator=1)
    {
        assert(denominator != 0);
        m_numerator = numerator;
        m_denominator = denominator;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};
```

Обратите внимание, теперь у нас есть два конструктора: конструктор по умолчанию, который будет вызываться, если мы не предоставим значения, и конструктор с параметрами, который будет вызываться, если мы предоставим значения. Эти два конструктора могут мирно сосуществовать в одном классе благодаря перегрузке функций. Фактически, вы можете определить любое количество конструкторов до тех пор, пока у них будут уникальные параметры (учитывается их количество и тип).

Как использовать конструктор с параметрами? Всё просто! Прямая инициализация:

```
1. int a(7); // прямая инициализация
2. Fraction drob(4, 5); // инициализируем напрямую, вызывается конструктор
   Fraction(int, int)
```

Здесь мы инициализировали нашу дробь числами `4` и `5`, результат - `4/5`!

В C++11 мы также можем использовать uniform-инициализацию:

```
1. int a { 7 }; // uniform-инициализация
2. Fraction drob {4, 5}; // uniform-инициализация, вызывается конструктор
   Fraction(int, int)
```

Мы также можем указать только один параметр для конструктора с параметрами, а второе значение будет значением по умолчанию:

```
1. Fraction seven(7); // вызывается конструктор Fraction(int, int), второй
   параметр использует значение по умолчанию
```

Значения по умолчанию для конструкторов работают точно так же, как и для любой другой функции, поэтому в вышеприведенном примере, когда мы вызываем `seven(7)`, вызывается `Fraction(int, int)`, второй параметр которого равен `1` (значение по умолчанию).

Правило: Используйте прямую инициализацию или uniform-инициализацию с объектами ваших классов.

Копирующая инициализация

Подобно обычным переменным, классы также можно инициализировать, используя копирующую инициализацию:

```
1. int a = 7; // копирующая инициализация
2. Fraction eight = Fraction(8); // копирующая инициализация, вызывается
   Fraction(8, 1)
3. Fraction nine = 9; // копирующая инициализация. Компилятор будет искать пути
   конвертации 9 в Fraction, что приведет к вызову конструктора Fraction(9, 1)
```

Однако рекомендуется избегать этой формы инициализации классов, так как она может быть менее эффективной. Хотя uniform-инициализация, прямая и копирующая инициализации работают одинаково с фундаментальными типами данных, с классами это не совсем так (хотя конечный результат часто совпадает). Мы рассмотрим этот момент более подробно на следующих уроках.

Правило: Не используйте копирующую инициализацию с объектами ваших классов.

Уменьшение количества конструкторов

В примере с классом Fraction и двумя конструкторами (по умолчанию и с параметрами), конструктор по умолчанию на самом деле лишний. Мы могли бы упростить этот класс следующим образом:

```
#include <cassert>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Конструктор по умолчанию
    Fraction(int numerator=0, int denominator=1)
    {
        assert(denominator != 0);
        m_numerator = numerator;
        m_denominator = denominator;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) /
m_denominator; }
};
```

Хотя этот конструктор по-прежнему является конструктором по умолчанию, он теперь определен таким образом, что может принимать одно или два значения, предоставленные пользователем:

```
Fraction drob; // вызов Fraction(0, 1)
Fraction seven(7); // вызов Fraction(7, 1)
Fraction sixTwo(6, 2); // вызов Fraction(6, 2)
```

На практике старайтесь сокращать количество конструкторов вашего класса.

Неявно генерируемый конструктор по умолчанию

Если ваш класс не имеет конструкторов, то язык C++ автоматически сгенерирует для вашего класса открытый конструктор по умолчанию. Его иногда называют **неявным конструктором** (или ***"неявно сгенерированным конструктором"***). Рассмотрим следующий класс:

```
class Date
{
private:
    int m_day = 12;
    int m_month = 1;
    int m_year = 2018;
};
```

У этого класса нет конструктора, поэтому компилятор сгенерирует следующий конструктор:

```
class Date
{
private:
    int m_day = 12;
    int m_month = 1;
    int m_year = 2018;

public:
    Date() // неявно генерируемый конструктор
    {
    }
};
```

Этот конструктор позволяет создавать объекты класса, но не выполняет их инициализацию и не присваивает значения членам класса.

Хотя вы не можете увидеть неявно сгенерированный конструктор, но его существование можно доказать:

```
class Date
{
private:
    int m_day = 12;
    int m_month = 1;
    int m_year = 2018;

    // Не было предоставлено конструктора, поэтому C++ автоматически создаст открытый конструктор по умолчанию
};

int main()
{
    Date date; // вызов неявного конструктора

    return 0;
}
```

Вышеприведенный код скомпилируется, поскольку в объекте `date` сработает неявный конструктор (который является открытым). Если ваш класс имеет другие конструкторы, то неявно генерируемый конструктор создаваться не будет. Например:

```
class Date
{
private:
    int m_day = 12;
    int m_month = 1;
    int m_year = 2018;

public:
    Date(int day, int month, int year) // обычный конструктор (не по умолчанию)
```

```
    {
        m_day = day;
        m_month = month;
        m_year = year;
    }

    // Неявный конструктор не создастся, так как мы уже определили свій конструктор
};

int main()
{
    Date date; // ошибка: Невозможно создать объект, так как конструктор по умолчанию не существует, и компилятор не сгенерировал неявный конструктор автоматически
    Date today(14, 10, 2020); // инициализируем объект today

    return 0;
}
```

Рекомендуется всегда создавать по крайней мере один конструктор в классе. Это позволит вам контролировать процесс создания объектов вашего класса, и предотвратит возникновение потенциальных проблем после добавления других конструкторов.

Правило: Создавайте хотя бы один конструктор в классе, даже если это пустой конструктор по умолчанию.

Классы, содержащие другие классы

Одни классы могут содержать другие классы в качестве переменных-членов. По умолчанию, при создании внешнего класса, для переменных-членов будут вызываться конструкторы по умолчанию. Это произойдет до того, как тело конструктора выполнится. Это можно продемонстрировать следующим образом:

```
#include <iostream>

class A
{
public:
    A() { std::cout << "A\n"; }
};

class B
{
private:
    A m_a; // B содержит A, как переменную-член

public:
    B() { std::cout << "B\n"; }
};

int main()
{
```

```
20.     B b;
21.     return 0;
22. }
```

Результат выполнения программы:

```
A
B
```

При создании переменной `b` вызывается конструктор B(). Прежде чем тело конструктора выполнится, `m_a` инициализируется, вызывая конструктор по умолчанию класса A. Таким образом выведется `A`. Затем управление возвратится обратно к конструктору B, и тело конструктора B начнет свое выполнение.

Здесь есть смысл, так как конструктор B() может захотеть использовать переменную `m_a`, поэтому сначала нужно инициализировать `m_a`!

Тест

Задание №1

а) Напишите класс Ball, который должен иметь следующие две закрытые переменные-члены со значениями по умолчанию:

- `m_color` (`Red`);
- `m_radius` (`20.0`).

В классе Ball должны быть следующие конструкторы:

- для установления значения только для `m_color`;
- для установления значения только для `m_radius`;
- для установления значений и для `m_radius`, и для `m_color`;
- для установления значений, когда значения не предоставлены вообще.

Не используйте параметры по умолчанию для конструкторов. Напишите еще одну функцию для вывода цвета (`m_color`) и радиуса (`m_radius`) шара (объекта класса Ball).

Следующий код функции main():

```
1. int main()
2. {
3.         Ball def;
4.         def.print();
5.
6.     Ball black("black");
```

```
7.      black.print();
8.
9.      Ball thirty(30.0);
10.     thirty.print();
11.
12.     Ball blackThirty("black", 30.0);
13.     blackThirty.print();
14.
15.     return 0;
16. }
```

Должен выдавать следующий результат:

```
color: red, radius: 20
color: black, radius: 20
color: red, radius: 30
color: black, radius: 30
```

b) Теперь обновите ваш код из предыдущего задания с использованием конструкторов с параметрами по умолчанию. Постарайтесь использовать как можно меньше конструкторов.

Задание №2

Что произойдет, если не объявить конструктор по умолчанию?

Урок №125. Список инициализации членов класса

На предыдущем уроке мы инициализировали члены нашего класса в конструкторе через оператор присваивания:

```
class Values
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Values()
    {
        // Это всё операции присваивания, а не инициализация
        m_value1 = 3;
        m_value2 = 4.5;
        m_value3 = 'd';
    }
};
```

Сначала создаются `m_value1`, `m_value2` и `m_value3`. Затем выполняется тело конструктора, где этим переменным присваиваются значения. Аналогичен код в не объектно-ориентированном C++:

```
int m_value1;
double m_value2;
char m_value3;

m_value1 = 3;
m_value2 = 4.5;
m_value3 = 'd';
```

Хотя в плане синтаксиса языка C++ вопросов никаких нет — всё корректно, но более эффективно — использовать инициализацию, а не присваивание после объявления.

Как мы уже знаем из предыдущих уроков, некоторые типы данных (например, константы и ссылки) должны быть инициализированы сразу. Рассмотрим следующий пример:

```
class Values
{
private:
    const int m_value;

public:
    Values()
    {
        m_value = 3; // ошибка: константам нельзя присваивать значения
    }
};
```

Аналогичен код в не объектно-ориентированном C++:

```
const int m_value; // ошибка: константы должны быть инициализированы значениями
m_value = 7; // ошибка: константам нельзя присваивать значения
```

Для решения этой проблемы в C++ добавили метод инициализации переменных-членов класса через **список инициализации членов**, вместо присваивания им значений после объявления. Не путайте этот список с аналогичным списком инициализаторов, который используется для инициализации массивов.

Из предыдущих уроков мы уже знаем, что инициализировать переменные можно тремя способами: через копирующую инициализацию, прямую инициализацию или uniform-инициализацию.

```
int value1 = 3; // копирующая инициализация
double value2(4.5); // прямая инициализация
char value3 {'d'} // uniform-инициализация
```

Использование списка инициализации почти идентично выполнению прямой инициализации (или uniform-инициализации в C++11).

Чтобы было понятнее, рассмотрим пример. Вот код с присваиванием значений переменным-членам класса в конструкторе:

```
class Values
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Values()
    {
        // Это всё операции присваивания, а не инициализация
        m_value1 = 3;
        m_value2 = 4.5;
        m_value3 = 'd';
    }
};
```

Теперь давайте перепишем этот код, но уже с использованием списка инициализации:

```
#include <iostream>

class Values
{
private:
    int m_value1;
    double m_value2;
    char m_value3;
```

```
9.
10. public:
11.     Values() : m_value1(3), m_value2(4.5), m_value3('d') // напрямую
    инициализируем переменные-члены класса
12.     {
13.     // Нет необходимости использовать присваивание
14.     }
15.
16.     void print()
17.     {
18.          std::cout << "Values(" << m_value1 << ", " << m_value2 << ", " <<
    m_value3 << ")\n";
19.     }
20. };
21.
22. int main()
23. {
24.     Values value;
25.     value.print();
26.     return 0;
27. }
```

Результат выполнения программы:

```
Values(3, 4.5, d)
```

Список инициализации членов находится сразу же после параметров конструктора. Он начинается с двоеточия (`:`), а затем значение для каждой переменной указывается в круглых скобках. Больше не нужно выполнять операции присваивания в теле конструктора. Также обратите внимание, что список инициализации членов не заканчивается точкой с запятой.

Можно также добавить возможность caller-у передавать значения для инициализации:

```
1.  #include <iostream>
2.
3.  class Values
4.  {
5.  private:
6.      int m_value1;
7.      double m_value2;
8.      char m_value3;
9.
10. public:
11.     Values(int value1, double value2, char value3='d')
12.         : m_value1(value1), m_value2(value2), m_value3(value3) // напрямую
    инициализируем переменные-члены класса
13.     {
14.     // Нет необходимости использовать присваивание
15.     }
16.
17.     void print()
18.     {
19.          std::cout << "Values(" << m_value1 << ", " << m_value2 << ", " <<
    m_value3 << ")\n";
```

```
    }

};

int main()
{
    Values value(3, 4.5); // value1 = 3, value2 = 4.5, value3 = 'd' (значение по умолчанию)
    value.print();
    return 0;
}
```

Результат выполнения программы:

```
Values(3, 4.5, d)
```

Мы можем использовать параметры по умолчанию для предоставления значений по умолчанию, если пользователь их не предоставил. Например, класс, который имеет константную переменную-член:

```
class Values
{
private:
    const int m_value;

public:
    Values(): m_value(7) // напрямую инициализируем константную переменную-член
    {
    }
};
```

Это работает, поскольку нам разрешено инициализировать константные переменные (но не присваивать им значения после объявления!).

Правило: Используйте списки инициализации членов, вместо операций присваивания, для инициализации переменных-членов вашего класса.

uniform-инициализация в C++11

В C++11 вместо прямой инициализации можно использовать uniform-инициализацию:

```
class Values
{
private:
    const int m_value;

public:
    Values(): m_value { 7 } // используем uniform-инициализацию
    {
    }
};
```

Настоятельно рекомендуется использовать этот синтаксис (даже если вы не используете константы или ссылки в качестве переменных-членов вашего класса), поскольку списки инициализации членов необходимы при композиции и наследовании (это рассмотрим несколько позже).

Правило: Используйте uniform-инициализацию вместо прямой инициализации в C++11.

Инициализация массивов в классе

Рассмотрим класс с массивом в качестве переменной-члена:

```
1. class Values
2. {
3. private:
4.     const int m_array[7];
5.
6. };
```

До C++11 мы могли только обнулить массив через список инициализации:

```
1. class Values
2. {
3. private:
4.     const int m_array[7];
5.
6. public:
7.     Values(): m_array {} // обнуляем массив
8.     {
9.         // Если мы хотим, чтобы массив имел значения, то мы должны здесь использовать присваивание
10.    }
11.
12.};
```

Однако в C++11 вы можете полностью инициализировать массив, используя uniform-инициализацию:

```
1. class Values
2. {
3. private:
4.     const int m_array[7];
5.
6. public:
7.     Values(): m_array { 3, 4, 5, 6, 7, 8, 9 } // используем uniform-инициализацию для инициализации массива
8.     {
9.     }
10.
11.};
```

Инициализация переменных-членов, которые являются классами

Список инициализации членов также может использоваться для инициализации членов, которые являются классами:

```
#include <iostream>

class A
{
public:
    A(int a) { std::cout << "A " << a << "\n"; }
};

class B
{
private:
    A m_a;
public:
    B(int b)
        : m_a(b - 1) // вызывается конструктор A(int) для инициализации члена m_a
    {
        std::cout << "B " << b << "\n";
    }
};

int main()
{
    B b(7);
    return 0;
}
```

Результат выполнения программы:

```
A 6
B 7
```

При создании переменной `b` вызывается конструктор `B(int)` со значением `7`. До того, как тело конструктора выполнится, инициализируется `m_a`, вызывая конструктор `A(int)` со значением `6`. Таким образом, выведется `A 6`. Затем управление возвратится обратно к конструктору B(), и тогда уже он выполнится и выведется `B 7`.

Использование списков инициализации

Если список инициализации помещается на той же строке, что и имя конструктора, то лучше всё разместить в одной строке:

```
class Values
{
private:
    int m_value1;
    double m_value2;
```

```
6.      char m_value3;
7.
8.  public:
9.      Values() : m_value1(3), m_value2(4.5), m_value3('d') // всё находится в
    одной строке
10.     {
11.     }
12. };
```

Если список инициализации членов не помещается в строке с именем конструктора, то на следующей строке (используя перенос) инициализаторы должны быть с отступом:

```
1.  class Values
2.  {
3.  private:
4.      int m_value1;
5.      double m_value2;
6.      char m_value3;
7.
8.  public:
9.      Values(int value1, double value2, char value3='d') // на этой строке уже
    и так много чего,
10.         : m_value1(value1), m_value2(value2), m_value3(value3) // поэтому
    переносим инициализаторы на новую строку (не забываем использовать отступ)
11.     {
12.     }
13.
14. };
```

Если все инициализаторы не помещаются на одной строке, то вы можете выделить для каждого инициализатора отдельную строку:

```
1.  class Values
2.  {
3.  private:
4.      int m_value1;
5.      double m_value2;
6.      char m_value3;
7.      float m_value4;
8.
9.  public:
10.     Values(int value1, double value2, char value3='d', float value4=17.5) // на
     этой строке уже и так много чего,
11.         : m_value1(value1), // поэтому выделяем каждому инициализатору
    отдельную строку, не забывая о запятой в конце
12.         m_value2(value2),
13.         m_value3(value3),
14.         m_value4(value4)
15.     {
16.     }
17.
18. };
```

Порядок выполнения в списке инициализации

Удивительно, но переменные в списке инициализации не инициализируются в том порядке, в котором они указаны. Вместо этого они инициализируются в том порядке, в котором объявлены в классе, поэтому следует соблюдать следующие рекомендации:

- Не инициализируйте переменные-члены таким образом, чтобы они зависели от других переменных-членов, которые инициализируются первыми (другими словами, убедитесь, что все ваши переменные-члены правильно инициализируются, даже если порядок в списке инициализации отличается).
- Инициализируйте переменные в списке инициализации в том порядке, в котором они объявлены в классе.

Заключение

Списки инициализации членов позволяют инициализировать члены, а не присваивать им значения. Это единственный способ инициализации констант и ссылок, которые являются переменными-членами вашего класса. Во многих случаях использование списка инициализации может быть более результативным, чем присваивание значений переменным-членам в теле конструктора. Списки инициализации работают как с переменными фундаментальных типов данных, так и с членами, которые сами являются классами.

Тест

Напишите класс с именем RGBA, который содержит 4 переменные-члены типа std::uint8_t (подключите заголовочный файл cstdint для доступа к типу std::uint8_t):

- `m_red`;
- `m_green`;
- `m_blue`;
- `m_alpha`.

Присвойте `0` в качестве значения по умолчанию для `m_red`, `m_green` и `m_blue`, и `255` для `m_alpha`. Создайте конструктор со списком инициализации членов, который позволит пользователю передавать значения для `m_red`, `m_green`, `m_blue` и `m_alpha`. Напишите функцию print(), которая будет выводить значения переменных-членов.

> ***Подсказка***: Если функция print() работает некорректно, то убедитесь, что вы конвертировали std::uint8_t в int.

Следующий код функции main():

```
1. int main()
2. {
3.     RGBA color(0, 135, 135);
4.     color.print();
5.
6.     return 0;
7. }
```

Должен выдавать следующий результат:

```
r=0 g=135 b=135 a=255
```

Урок №126. Инициализация нестатических членов класса

При написании класса с несколькими конструкторами, необходимость указывать значения по умолчанию всем членам в каждом конструкторе приведет к написанию лишнего кода. Если вы обновите значение по умолчанию какого-то одного члена, то вам придется лезть в каждый конструктор.

Начиная с C++11, обычным переменным-членам класса (те, которые не используют ключевое слово static) можно задать значение по умолчанию напрямую - использовать инициализацию:

```
#include <iostream>

class Something
{
private:
    double m_length = 3.5; // m_length имеет значение по умолчанию 3.5
    double m_width = 3.5; // m_width имеет значение по умолчанию 3.5

public:
    Something()
    {
    // Этот конструктор использует значения по умолчанию, приведенные выше, так как здесь эти значения не переопределяются
    }

    void print()
    {
        std::cout << "length: " << m_length << " and width: " << m_width << '\n';
    }
};

int main()
{
    Something a; // a.m_length = 3.5, a.m_width = 3.5
    a.print();

    return 0;
}
```

Результат выполнения программы:

```
length: 3.5 and width: 3.5
```

Таким образом, вы предоставляете значения по умолчанию вашим переменным-членам, которые будут использоваться вашими конструкторами, если сами конструкторы не предоставят эти значения (через списки инициализации членов).

Однако конструкторы все еще определяют тип объектов, которые могут быть созданы, например:

```
#include <iostream>

class Something
{
private:
    double m_length = 3.5;
    double m_width = 3.5;

public:

    // Обратите внимание, здесь нет конструктора по умолчанию

    Something(double length, double width)
        : m_length(length), m_width(width)
    {
        // m_length и m_width инициализируются этим конструктором (значения по умолчанию, приведенные выше, не используются)
    }

    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';

    }

};

int main()
{
    Something a; // не скомпилируется, так как требуется конструктор по умолчанию, даже если члены класса имеют значения по умолчанию

    return 0;
}
```

Несмотря на то, что мы предоставили значения по умолчанию всем переменным-членам класса, конструктор по умолчанию предоставлен не был, поэтому мы не можем создать объект класса Something без параметров.

Если предоставлено значение по умолчанию, и конструктор инициализирует член через список инициализации членов, то приоритет будет у списка инициализации членов:

```
#include <iostream>

class Something
{
private:
    double m_length = 3.5;
    double m_width = 3.5;

public:

```

```
    Something(double length, double width)
        : m_length(length), m_width(width)
    {
        // m_length и m_width инициализируются конструктором (значения по умолчанию, приведенные выше, не используются)
    }

    void print()
    {
        std::cout << "length: " << m_length << " and width: " << m_width << '\n';
    }

};

int main()
{
    Something a(4.5, 5.5);
    a.print();

    return 0;
}
```

Результат выполнения программы:

```
length: 4.5 and width: 5.5
```

Правило: Используйте инициализацию нестатических членов для указания значений по умолчанию переменным-членам.

Тест

Задание №1

Добавьте в следующую программу инициализацию нестатических членов и список инициализации членов:

```
#include <string>
#include <iostream>

class Thing
{
private:
    std::string m_color;
    double m_radius;

public:
        // Конструктор по умолчанию без параметров
    Thing()
    {
        m_color = "blue";
        m_radius = 20.0;
    }

        // Конструктор с параметром color (для radius предоставлено значение по умолчанию)
```

```
19.     Thing(const std::string &color)
20.     {
21.         m_color = color;
22.         m_radius = 20.0;
23.     }
24.
25.         // Конструктор с параметром radius (для color предоставлено значение
    по умолчанию)
26.     Thing(double radius)
27.     {
28.         m_color = "blue";
29.         m_radius = radius;
30.     }
31.
32.         // Конструктор с параметрами color и radius
33.     Thing(const std::string &color, double radius)
34.     {
35.         m_color = color;
36.         m_radius = radius;
37.     }
38.
39.     void print()
40.     {
41.         std::cout << "color: " << m_color << " and radius: " << m_radius <<
    '\n';
42.     }
43. };
44.
45. int main()
46. {
47.     Thing defl;
48.     defl.print();
49.
50.     Thing red("red");
51.     red.print();
52.
53.     Thing thirty(30.0);
54.     thirty.print();
55.
56.     Thing redThirty("red", 30.0);
57.     redThirty.print();
58.
59.     return 0;
60. }
```

Результат выполнения программы должен быть следующим:

```
color: blue and radius: 20
color: red and radius: 20
color: blue and radius: 30
color: red and radius: 30
```

Задание №2

Зачем мы объявили пустой конструктор по умолчанию в программе из задания №1? Все же переменные-члены и так имеют значения по умолчанию.

Урок №127. Делегирующие конструкторы

При создании нового объекта класса, компилятор C++ неявно вызывает конструктор этого объекта. Не редкость встретить класс с несколькими конструкторами, которые частично выполняют одно и то же, например:

```
1. class Boo
2. {
3. public:
4.     Boo()
5.     {
6.         // Часть кода X
7.     }
8.
9.     Boo(int value)
10.    {
11.        // Часть кода X
12.        // Часть кода Y
13.    }
14. };
```

Здесь есть 2 конструктора: конструктор по умолчанию и конструктор, который принимает целочисленное значение. Поскольку `Часть кода X` требуется обоим конструкторам, то она дублируется в каждом из них.

А как вы уже могли догадаться, дублирование кода - это то, чего следует избегать, поэтому давайте рассмотрим возможные решения этой проблемы.

Решение в C++11

Неплохо было бы, чтобы конструктор `Boo(int)` вызывал конструктор Boo() для выполнения `Часть кода X`:

```
1. class Boo
2. {
3. public:
4.     Boo()
5.     {
6.         // Часть кода X
7.     }
8.
9.     Boo(int value)
10.    {
11.        Boo(); // используем конструктор, указанный выше, для выполнения
   части кода X
12.        // Часть кода Y
13.    }
14. };
```

Или:

```
class Boo
{
public:
    Boo()
    {
        // часть кода X
    }

    Boo(int value): Boo() // используем конструктор, указанный выше, для выполнения части кода X
    {
        // часть кода Y
    }
};
```

Однако, если ваш компилятор не совместим с C++11, и вы попытаетесь вызвать один конструктор внутри другого конструктора, то это скомпилируется, но будет работать не так, как вы ожидаете.

До C++11 явный вызов одного конструктора из другого приводит к созданию временного объекта, который затем инициализируется с помощью конструктора этого объекта и отбрасывается, оставляя исходный объект неизменным.

Использование отдельного метода

Конструкторам разрешено вызывать другие методы класса, которые не являются конструкторами. Хотя у вас может возникнуть соблазн скопировать код из первого конструктора во второй конструктор, наличие дублированного кода сделает ваш класс более трудным для понимания и более обременительным для поддержки. Лучшим решением будет создание отдельного метода (не конструктора), который будет выполнять *общую* инициализацию, и оба конструктора будут вызывать этот метод. Например:

```
class Boo
{
private:
    void DoX()
    {
        // Часть кода X
    }

public:
    Boo()
    {
        DoX();
    }

    Boo(int nValue)
    {
        DoX();
```

```
        // Часть кода Y
    }

};
```

Здесь мы свели дублирование кода к минимуму.

Кроме того, вы можете оказаться в ситуации, когда вам нужно будет написать метод для повторной инициализации класса обратно до значений по умолчанию. Поскольку у вас, вероятно, уже есть конструктор, который это делает, то у вас может возникнуть соблазн попытаться вызвать этот конструктор из вашего метода. Однако это приведет к неожиданным результатам. Многие разработчики просто копируют код из конструктора в функцию инициализации — это сработает, но приведет также к дублированию кода. Лучшим решением будет переместить код из конструктора в вашу новую функцию и заставить конструктор вызывать вашу новую функцию для выполнения инициализации:

```
class Boo
{
public:
    Boo()
    {
        Init();
    }

    Boo(int value)
    {
        Init();
        // Делаем что-либо с value
    }

    void Init()
    {
        // Код инициализации Boo
    }
};
```

Здесь мы подключаем функцию Init() для инициализации переменных-членов обратно значениями по умолчанию, а затем каждый конструктор вызывает функцию Init() перед своим фактическим выполнением. Это сокращает дублирование кода до минимума и позволяет явно вызывать Init() из любого места в программе.

Делегирующие конструкторы в C++11

Начиная с C++11, конструкторам разрешено вызывать другие конструкторы. Этот процесс называется **делегированием конструкторов** (или ***"цепочкой конструкторов"***). Чтобы один конструктор вызывал другой, нужно просто сделать вызов этого конструктора в списке инициализации членов.

Например:

```
1. class Boo
2. {
3. private:
4.
5. public:
6.     Boo()
7.     {
8.         // Часть кода X
9.     }
10.
11.    Boo(int value): Boo() // используем конструктор по умолчанию Boo() для
   выполнения части кода X
12.    {
13.        // Часть кода Y
14.    }
15.
16.};
```

Всё работает как нужно. Убедитесь, что вы вызываете конструктор из списка инициализации членов, а не из тела конструктора.

Вот еще один пример использования делегирующих конструкторов для сокращения дублированного кода:

```
1. #include <iostream>
2. #include <string>
3.
4. class Employee
5. {
6. private:
7.     int m_id;
8.     std::string m_name;
9.
10.public:
11.    Employee(int id=0, const std::string &name=""):
12.        m_id(id), m_name(name)
13.    {
14.        std::cout << "Employee " << m_name << " created.\n";
15.    }
16.
17.    // Используем делегирующие конструкторы для сокращения дублированного кода
18.    Employee(const std::string &name) : Employee(0, name) { }
19.};
20.
21.int main()
22.{
23.    Employee a;
24.    Employee b("Ivan");
25.
26.    return 0;
27.}
```

Этот класс имеет 2 конструктора (один из которых вызывает другой). Таким образом, количество дублированного кода сокращено (нам нужно записать только одно определение конструктора вместо двух).

Несколько заметок о делегирующих конструкторах

Во-первых, конструктору, который вызывает другой конструктор, не разрешается выполнять какую-либо инициализацию членов класса. Поэтому конструкторы могут либо вызывать другие конструкторы, либо выполнять инициализацию, но не всё сразу.

Во-вторых, один конструктор может вызывать другой конструктор, в коде которого может находиться вызов первого конструктора. Это создаст бесконечный цикл и приведет к тому, что память стека закончится и произойдет сбой. Вы можете избежать этого, убедившись, что в конструкторе, который вызывается, нет вызова первого (и вообще любого другого) конструктора. Будьте аккуратны и не используйте вложенные вызовы конструкторов.

Урок №128. Деструкторы

Деструктор - это специальный тип метода класса, который выполняется при удалении объекта класса. В то время как конструкторы предназначены для инициализации класса, деструкторы предназначены для очистки памяти после него.

Когда объект автоматически выходит из области видимости или динамически выделенный объект явно удаляется с помощью ключевого слова delete, вызывается деструктор класса (если он существует) для выполнения необходимой очистки до того, как объект будет удален из памяти. Для простых классов (тех, которые только инициализируют значения обычных переменных-членов) деструктор не нужен, так как C++ автоматически выполнит очистку самостоятельно.

Однако, если объект класса содержит любые ресурсы (например, динамически выделенную память или файл/базу данных), или, если вам необходимо выполнить какие-либо действия до того, как объект будет уничтожен, деструктор является идеальным решением, поскольку он производит последние действия с объектом перед его окончательным уничтожением.

Имена деструкторов

Так же, как и конструкторы, деструкторы имеют свои правила, которые касаются их имен:

- деструктор должен иметь то же имя, что и класс, со знаком тильда (`~`) в самом начале;
- деструктор не может принимать аргументы;
- деструктор не имеет типа возврата.

Из второго правила вытекает еще одно правило: для каждого класса может существовать только один деструктор, так как нет возможности перегрузить деструкторы, как функции, и отличаться друг от друга аргументами они не могут.

Подобно конструкторам, деструкторы не вызываются явно. Однако их могут безопасно вызывать другие методы класса, так как объект не уничтожится до тех пор, пока не выполнится деструктор.

Пример использования деструктора на практике

Рассмотрим простой класс с деструктором:

```
1. #include <iostream>
2. #include <cassert>
3.
4. class Massiv
5. {
6. private:
7.     int *m_array;
8.     int m_length;
9.
10. public:
11.     Massiv(int length) // конструктор
12.     {
13.         assert(length > 0);
14.
15.         m_array = new int[length];
16.         m_length = length;
17.     }
18.
19.     ~Massiv() // деструктор
20.     {
21.         // Динамически удаляем массив, который выделили ранее
22.         delete[] m_array ;
23.     }
24.
25.     void setValue(int index, int value) { m_array[index] = value; }
26.     int getValue(int index) { return m_array[index]; }
27.
28.     int getLength() { return m_length; }
29. };
30.
31. int main()
32. {
33.     Massiv arr(15); // выделяем 15 целочисленных значений
34.     for (int count=0; count < 15; ++count)
35.         arr.setValue(count, count+1);
36.
37.     std::cout << "The value of element 7 is " << arr.getValue(7);
38.
39.     return 0;
40. } // объект arr удаляется здесь, поэтому деструктор ~Massiv() вызывается тоже
    здесь
```

Результат выполнения программы:

```
The value of element 7 is 8
```

В первой строке функции main() мы создаем новый объект класса Massiv с именем `arr` и передаем длину (`length`) 15. Это приводит к вызову конструктора, который динамически выделяет память для массива класса (`m_array`). Мы должны здесь использовать динамическое выделение, поскольку на момент компиляции мы не знаем длину массива (это значение нам передает caller).

В конце функции main() объект `arr` выходит из области видимости. Это приводит к вызову деструктора `~Massiv()` и к удалению массива, который мы выделили ранее в конструкторе!

Выполнение конструкторов и деструкторов

Как мы уже знаем, конструктор вызывается при создании объекта, а деструктор — при его уничтожении. В следующем примере мы будем использовать стейтменты с cout внутри конструктора и деструктора для отображения их времени выполнения:

```
#include <iostream>

class Another
{
private:
    int m_nID;

public:
    Another(int nID)
    {
        std::cout << "Constructing Another " << nID << '\n';
        m_nID = nID;
    }

    ~Another()
    {
        std::cout << "Destructing Another " << m_nID << '\n';
    }

    int getID() { return m_nID; }
};

int main()
{
    // Выделяем объект класса Another из стека
    Another object(1);
    std::cout << object.getID() << '\n';

    // Выделяем объект класса Another динамически из кучи
    Another *pObject = new Another(2);
    std::cout << pObject->getID() << '\n';
    delete pObject;

    return 0;
} // объект object выходит из области видимости здесь
```

Результат выполнения программы:

```
Constructing Another 1
1
Constructing Another 2
2
Destructing Another 2
Destructing Another 1
```

Обратите внимание, `Another 1` уничтожается после `Another 2`, так как мы удалили `pObject` до завершения выполнения функции main(), тогда как объект `object` не был удален до конца main().

Идиома программирования RAII

Идиома RAII (англ. *«**R**esource **A**cquisition **I**s **I**nitialization» = «Получение ресурсов есть инициализация»*) - это идиома объектно-ориентированного программирования, при которой использование ресурсов привязывается к времени жизни объектов с автоматической продолжительностью жизни. В языке C++ идиома RAII реализуется через классы с конструкторами и деструкторами. Ресурс (например, память, файл или база данных) обычно приобретается в конструкторе объекта (хотя этот ресурс может быть получен и после создания объекта, если в этом есть смысл). Затем этот ресурс можно использовать, пока объект жив. Ресурс освобождается в деструкторе при уничтожении объекта. Основным преимуществом RAII является то, что это помогает предотвратить утечку ресурсов (например, памяти, которая не была освобождена), так как все объекты, содержащие ресурсы, автоматически очищаются.

В рамках идиомы программирования RAII объекты, располагающие ресурсами, не должны быть динамически выделенными, так как деструкторы вызываются только при уничтожении объектов. Для объектов, выделенных из стека, это происходит автоматически, когда объект выходит из области видимости, поэтому нет необходимости беспокоиться о том, что ресурс в конечном итоге не будет очищен. Однако за очистку динамически выделенных объектов, которые выделяются из кучи, уже пользователь несет ответственность: если он забыл её выполнить, деструктор вызываться не будет, и память как для объекта класса, так и для управляемого ресурса будет потеряна — произойдет утечка памяти!

Класс Massiv из программы, приведенной в начале этого урока, является примером класса, который реализует принципы RAII: выделение в конструкторе, освобождение в деструкторе. std::string и std::vector — это примеры классов из Стандартной библиотеки C++, которые следуют принципам RAII: динамическая память выделяется при инициализации и автоматически освобождается при уничтожении.

Правило: Используйте идиому программирования RAII и не выделяйте объекты вашего класса динамически.

Предупреждение о функции exit()

Если вы используете функцию exit(), то ваша программа завершится, и никакие деструкторы не будут вызваны. Будьте осторожны, если в таком случае вы полагаетесь на свои деструкторы для выполнения необходимой работы по очистке (например, перед тем, как выйти, вы записываете что-нибудь в лог-файл или в базу данных).

Заключение

Используя конструкторы и деструкторы, ваши классы могут выполнять инициализацию и очистку после себя автоматически без вашего участия! Это уменьшает вероятность возникновения ошибок и упрощает процесс использования классов.

Урок №129. Скрытый указатель *this

Один из частых вопросов, которые новички задают по поводу классов: «При вызове метода класса, как C++ отслеживает то, какой объект его вызвал?». Ответ заключается в том, что C++ для этих целей использует скрытый указатель *this!

Скрытый указатель *this

Ниже приведен простой класс, который содержит целочисленное значение и имеет конструктор и функции доступа. Обратите внимание, деструктор здесь не нужен, так как язык C++ может очистить память после переменной-члена самостоятельно:

```
#include <iostream>

class Another
{
private:
    int m_number;

public:
    Another(int number)
    {
        setNumber(number);
    }

    void setNumber(int number) { m_number = number; }
    int getNumber() { return m_number; }
};

int main()
{
    Another another(3);
    another.setNumber(4);
    std::cout << another.getNumber() << '\n';

    return 0;
}
```

Результат выполнения программы:

```
4
```

При вызове `another.setNumber(4);` C++ понимает, что функция setNumber() работает с объектом `another`, а `m_number` — это фактически `another.m_number`. Рассмотрим детально, как это всё работает.

Возьмем, к примеру, следующую строку:

```
another.setNumber(4);
```

Хотя на первый взгляд кажется, что у нас здесь только один аргумент, но на самом деле у нас их два! Во время компиляции строка `another.setNumber(4);` конвертируется компилятором в следующее:

```
1. setNumber(&another, 4); // объект another конвертировался из объекта, который находился перед точкой, в аргумент функции!
```

Теперь это всего лишь стандартный вызов функции, а объект `another` (который ранее был отдельным объектом и находился перед точкой) теперь передается по адресу в качестве аргумента функции.

Но это только половина дела. Поскольку в вызове функции теперь есть два аргумента, то и метод нужно изменить соответствующим образом (чтобы он принимал два аргумента). Следовательно, следующий метод:

```
1. void setNumber(int number) { m_number = number; }
```

Конвертируется компилятором в:

```
1. void setNumber(Another* const this, int number) { this->m_number = number; }
```

При компиляции обычного метода, компилятор неявно добавляет к нему параметр *this. **Указатель *this** — это скрытый константный указатель, содержащий адрес объекта, который вызывает метод класса.

Есть еще одна деталь. Внутри метода также необходимо обновить все члены класса (функции и переменные), чтобы они ссылались на объект, который вызывает этот метод. Это легко сделать, добавив префикс `this->` к каждому из них. Таким образом, в теле функции setNumber(), `m_number` (переменная-член класса) будет конвертирована в `this->m_number`. И когда *this указывает на адрес `another`, то `this->m_number` будет указывать на `another.m_number`.

Соединяем всё вместе:

- При вызове `another.setNumber(4)`, компилятор фактически вызывает `setNumber(&another, 4)`.
- Внутри setNumber(), указатель *this содержит адрес объекта `another`.
- К любым переменным-членам внутри setNumber() добавляется префикс `this->`. Поэтому, когда мы говорим `m_number = number`, компилятор фактически выполняет `this -> m_number = number`, который, в этом случае, обновляет `another.m_number` на `number`.

Хорошей новостью является то, что это всё происходит скрыто от нас (программистов), и не имеет значения, помните ли вы, как это работает или нет. Всё, что вам нужно запомнить: все обычные методы класса имеют указатель *this, который указывает на объект, связанный с вызовом метода класса.

Указатель *this всегда указывает на текущий объект

Начинающие программисты иногда путают, сколько указателей *this существует. Каждый метод имеет в качестве параметра указатель *this, который указывает на адрес объекта, с которым в данный момент выполняется операция, например:

```
1. int main()
2. {
3.     Another X(3); // *this = &X внутри конструктора Another
4.     Another Y(4); // *this = &Y внутри конструктора Another
5.     X.setNumber(5); // *this = &X внутри метода setNumber
6.     Y.setNumber(6); // *this = &Y внутри метода setNumber
7.
8.     return 0;
9. }
```

Обратите внимание, указатель *this поочерёдно содержит адрес объектов `X` или `Y` в зависимости от того, какой метод вызван и сейчас выполняется.

Явное указание указателя *this

В большинстве случаев вам не нужно явно указывать указатель *this. Тем не менее, иногда это может быть полезным. Например, если у вас есть конструктор (или метод), который имеет параметр с тем же именем, что и переменная-член, то устранить неоднозначность можно с помощью указателя *this:

```
1. class Something
2. {
3. private:
4.     int data;
5.
6. public:
7.     Something(int data)
8.     {
9.         this->data = data;
10.    }
11. };
```

Здесь конструктор принимает параметр с тем же именем, что и переменная-член. В этом случае `data` относится к параметру, а `this->data` относится к переменной-члену. Хотя это приемлемая практика, но рекомендуется использовать префикс `m_` для всех имен переменных-членов вашего класса, так как это помогает предотвратить дублирование имен в целом!

Цепочки методов класса

Иногда бывает полезно, чтобы метод класса возвращал объект, с которым работает, в виде возвращаемого значения. Основной смысл здесь - это позволить нескольким методам объединиться в «цепочку», работая при этом с одним объектом! Мы на самом деле пользуемся этим уже давно. Например, когда мы выводим данные с помощью std::cout по частям:

```
1. std::cout << "Hello, " << userName;
```

В этом случае std::cout является объектом, а оператор `<<` - методом, который работает с этим объектом. Компилятор обрабатывает фрагмент, приведенный выше, следующим образом:

```
1. (std::cout << "Hello, ") << userName;
```

Сначала оператор `<<` использует std::cout и строковый литерал `Hello` для вывода `Hello` в консоль. Однако, поскольку это часть выражения, оператор `<<` также должен возвратить значение (или void). Если оператор `<<` возвращает void, то получается следующее:

```
1. (void) << userName;
```

Что явно не имеет никакого смысла (компилятор выдаст ошибку). Однако, вместо этого, оператор `<<` возвращает указатель *this, что в этом контексте является просто std::cout. Таким образом, после обработки первого оператора `<<`, мы получаем:

```
1. (std::cout) << userName;
```

Что приводит к выводу имени пользователя (`userName`).

Таким образом, нам нужно указать объект (в данном случае, std::cout) один раз, и каждый вызов функции будет передавать этот объект следующей функции, что позволит нам объединить несколько методов вместе.

Мы сами можем реализовать такое поведение. Рассмотрим следующий класс:

```
1.  class Mathem
2.  {
3.  private:
4.      int m_value;
5.
6.  public:
7.      Mathem() { m_value = 0; }
8.
9.      void add(int value) { m_value += value; }
10.     void sub(int value) { m_value -= value; }
```

```
11.     void multiply(int value) { m_value *= value; }
12.
13.     int getValue() { return m_value; }
14. };
```

Если вы хотите добавить 7, вычесть 5 и умножить всё на 3, то нужно сделать следующее:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      Mathem operation;
6.      operation.add(7); // возвращает void
7.      operation.sub(5); // возвращает void
8.      operation.multiply(3); // возвращает void
9.
10.     std::cout << operation.getValue() << '\n';
11.     return 0;
12. }
```

Результат:

```
6
```

Однако, если каждая функция будет возвращать указатель *this, то мы сможем связать эти вызовы методов в одну цепочку. Например:

```
1.  class Mathem
2.  {
3.  private:
4.      int m_value;
5.
6.  public:
7.      Mathem() { m_value = 0; }
8.
9.      Mathem& add(int value) { m_value += value; return *this; }
10.     Mathem& sub(int value) { m_value -= value; return *this; }
11.     Mathem& multiply(int value) { m_value *= value; return *this; }
12.
13.     int getValue() { return m_value; }
14. };
```

Обратите внимание, add(), sub() и multiply() теперь возвращают указатель *this, поэтому следующее будет корректным:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      Mathem operation;
6.      operation.add(7).sub(5).multiply(3);
7.
8.      std::cout << operation.getValue() << '\n';
9.      return 0;
10. }
```

Результат:

```
6
```

Мы фактически вместили три отдельные строки в одно выражение! Теперь рассмотрим это детально:

- Сначала вызывается `operation.add(7)`, который добавляет `7` к нашему `m_value`.
- Затем add() возвращает указатель *this, который является ссылкой на объект `operation`.
- Затем вызов `operation.sub(5)` вычитает `5` из `m_value` и возвращает `operation`.
- `multiply(3)` умножает `m_value` на `3` и возвращает `operation`, который уже игнорируется.
- Однако, поскольку каждая функция модифицировала `operation`, `m_value` объекта `operation` теперь содержит значение `((0 + 7) - 5) * 3)`, которое равно `6`.

Заключение

Указатель *this является скрытым параметром, который неявно добавляется к каждому методу класса. В большинстве случаев нам не нужно обращаться к нему напрямую, но при необходимости это можно сделать. Стоит отметить, что указатель *this является константным указателем - вы можете изменить значение исходного объекта, но вы не можете заставить указатель *this указывать на что-то другое!

Если у вас есть функции, которые возвращают void, то возвращайте *this вместо void. Таким образом, вы сможете соединить несколько методов в одну «цепочку». Это чаще всего используется при перегрузке операторов, но об этом несколько позже.

Урок №130. Классы и заголовочные файлы

Все классы, которые мы использовали до сих пор, были достаточно простыми, поэтому мы записывали методы непосредственно внутри тела классов, например:

```
1.  class Date
2.  {
3.  private:
4.      int m_day;
5.      int m_month;
6.      int m_year;
7.
8.  public:
9.      Date(int day, int month, int year)
10.     {
11.         setDate(day, month, year);
12.     }
13.
14.     void setDate(int day, int month, int year)
15.     {
16.         m_day = day;
17.         m_month = month;
18.         m_year = year;
19.     }
20.
21.     int getDay() { return m_day; }
22.     int getMonth() { return m_month; }
23.     int getYear()  { return m_year; }
24. };
```

Однако, как только классы становятся больше и сложнее, наличие всех методов внутри тела класса может затруднить его управление и работу с ним. Использование уже написанного класса требует понимания только его открытого интерфейса, а не того, как он реализован под капотом.

К счастью, язык C++ предоставляет способ отделить «объявление» от «реализации». Это делается путем определения методов вне тела самого класса. Для этого просто определите методы класса, как если бы они были обычными функциями, но в качестве префикса добавьте к имени функции имя класса с оператором разрешения области видимости (`::`).

Вот наш класс Date с конструктором Date() и методом setDate(), определенными вне тела класса. Обратите внимание, прототипы этих функций все еще находятся внутри тела класса, но их фактическая реализация находится за его пределами:

```
1.  class Date
2.  {
3.  private:
4.      int m_day;
5.      int m_month;
6.      int m_year;
```

```
7.
8.  public:
9.      Date(int day, int month, int year);
10.
11.     void SetDate(int day, int month, int year);
12.
13.     int getDay() { return m_day; }
14.     int getMonth() { return m_month; }
15.     int getYear()  { return m_year; }
16. };
17.
18. // Конструктор класса Date
19. Date::Date(int day, int month, int year)
20. {
21.     SetDate(day, month, year);
22. }
23.
24. // Метод класса Date
25. void Date::SetDate(int day, int month, int year)
26. {
27.     m_day = day;
28.     m_month = month;
29.     m_year = year;
30. }
```

Просто, не так ли? Поскольку во многих случаях функции доступа могут состоять всего из одной строки кода, то их обычно оставляют в теле класса, хотя переместить их за пределы класса можно всегда.

Вот еще один пример класса с конструктором, определенным извне, со списком инициализации членов:

```
1.  class Mathem
2.  {
3.  private:
4.      int m_value = 0;
5.
6.  public:
7.      Mathem(int value=0): m_value(value) {}
8.
9.      Mathem& add(int value) { m_value  += value;  return *this; }
10.     Mathem& sub(int value) { m_value -= value;  return *this; }
11.     Mathem& divide(int value) { m_value /= value;  return *this; }
12.
13.     int getValue() { return m_value ; }
14. };
```

Конвертируем в:

```
1.  class Mathem
2.  {
3.  private:
4.      int m_value = 0;
5.
6.  public:
7.      Mathem(int value=0);
8.
```

```
9.      Mathem& add(int value);
10.     Mathem& sub(int value);
11.     Mathem& divide(int value);
12.
13.     int getValue() { return m_value; }
14. };
15.
16. Mathem::Mathem(int value): m_value(value)
17. {
18. }
19.
20. Mathem& Mathem::add(int value)
21. {
22.     m_value += value;
23.     return *this;
24. }
25.
26. Mathem& Mathem::sub(int value)
27. {
28.     m_value -= value;
29.     return *this;
30. }
31.
32. Mathem& Mathem::divide(int value)
33. {
34.     m_value /= value;
35.     return *this;
36. }
```

Классы и заголовочные файлы

На уроке о заголовочных файлах мы узнали, что объявления функций можно поместить в заголовочные файлы, чтобы затем иметь возможность использовать эти функции в нескольких файлах или даже в нескольких проектах. Классы в этом плане ничем не отличаются от функций. Определения классов могут быть помещены в заголовочные файлы для облегчения их повторного использования в нескольких файлах или проектах. Обычно, определение класса помещается в заголовочный файл с тем же именем, что у класса, а методы, определенные вне тела класса, помещаются в файл .cpp с тем же именем, что у класса.

Вот наш класс Date, но уже разбитый на файлы .cpp и .h:

Date.h:

```
1.  #ifndef DATE_H
2.  #define DATE_H
3.
4.  class Date
5.  {
6.  private:
7.      int m_day;
8.      int m_month;
9.      int m_year;
10.
```

```
11. public:
12.     Date(int day, int month, int year);
13. 
14.     void SetDate(int day, int month, int year);
15. 
16.     int getDay() { return m_day; }
17.     int getMonth() { return m_month; }
18.     int getYear()  { return m_year; }
19. };
20. 
21. #endif
```

Date.cpp:

```
1.  #include "Date.h"
2.  
3.  // Конструктор класса Date
4.  Date::Date(int day, int month, int year)
5.  {
6.      SetDate(day, month, year);
7.  }
8.  
9.  // Метод класса Date
10. void Date::SetDate(int day, int month, int year)
11. {
12.     m_day = day;
13.     m_month = month;
14.     m_year = year;
15. }
```

Теперь любой другой файл .h или .cpp, который захочет использовать класс Date, сможет просто подключить заголовочный файл: `#include "Date.h"`. Обратите внимание, Date.cpp также необходимо добавить до компиляции в проект, который использует Date.h, чтобы линкер смог разобраться с реализацией класса Date.

Вопрос №1: **"Разве определение класса в заголовочном файле не нарушает правило одного определения?".**

Нет. Классы - это пользовательские типы данных, которые освобождаются от определения только в одном месте. Поэтому класс, определенный в заголовочном файле, можно свободно подключать в другие файлы.

Вопрос №2: **"Разве определения методов класса в заголовочном файле не нарушает правило одного определения?".**

Методы, определенные внутри тела класса, считаются неявно встроенными. Встроенные функции освобождаются от правила одного определения. А это означает, что проблем с определением простых методов (таких как функции доступа) внутри самого класса возникать не должно.

Методы, определенные вне тела класса, рассматриваются, как обычные функции, и подчиняются правилу одного определения, поэтому эти функции должны быть определены в файле .cpp, а не внутри .h. Единственным исключением являются шаблоны функций (но об этом чуть позже).

Параметры по умолчанию

Параметры по умолчанию для методов должны быть объявлены в теле класса (в заголовочном файле), где они будут видны всем, кто подключает этот заголовочный файл с классом.

Библиотеки

Разделение объявления класса и его реализации очень распространено в библиотеках, которые используются для расширения возможностей вашей программы. Вы также подключали такие заголовочные файлы из Стандартной библиотеки C++, как iostream, string, vector, array и другие. Обратите внимание, вы не добавляли iostream.cpp, string.cpp, vector.cpp или array.cpp в ваши проекты. Ваша программа нуждается только в объявлениях из заголовочных файлов, чтобы компилятор смог проверить корректность вашего кода в соответствии с правилами синтаксиса языка C++. Однако реализации классов, находящихся в Стандартной библиотеке C++, содержатся в предварительно скомпилированном файле, который добавляется на этапе линкинга. Вы нигде не встречаете этот код.

Вне программ с открытым исходным кодом (где предоставляются оба файла: .h и .cpp), большинство сторонних библиотек предоставляют только заголовочные файлы вместе с предварительно скомпилированным файлом библиотеки. На это есть несколько причин:

- На этапе линкинга быстрее будет подключить предварительно скомпилированную библиотеку, чем выполнять перекомпиляцию каждый раз, когда она нужна.
- Защита интеллектуальной собственности (создатели не хотят, чтобы другие просто "воровали" их код).

Наличие собственных файлов, разделенных на объявление (файлы .h) и реализацию (файлы .cpp), является не только хорошей формой содержания кода, но и упрощает создание собственных пользовательских библиотек.

Заключение

Возможно, у вас возникнет соблазн поместить все определения методов класса в заголовочный файл внутри тела класса. Хотя это скомпилируется, но здесь есть несколько нюансов:

- Во-первых, как упоминалось выше, это приведет к загромождению определения вашего класса.
- Во-вторых, функции, определенные внутри класса, являются неявно встроенными. Большие функции, которые вызываются из многих файлов, могут способствовать, таким образом, «раздуванию» вашего кода.
- В-третьих, если вы измените что-либо в заголовочном файле, то вам нужно будет перекомпилировать каждый файл, содержащий этот заголовок. Это может иметь "эффект бабочки", когда одно незначительное изменение заставит перекомпилировать всю программу (что может быть достаточно медленно и долго). Если же вы изменили код в файле .cpp, то вам необходимо перекомпилировать только этот файл .cpp!

Поэтому рекомендуется следующее:

- Классы, используемые только в одном файле, и которые повторно не используются, определяйте непосредственно в файле .cpp, где они используются.
- Классы, используемые в нескольких файлах или предназначенные для повторного использования, определяйте в заголовочном файле с тем же именем, что у класса.
- Тривиальные методы (обычные конструкторы или деструкторы, функции доступа и т.д.) определяйте внутри тела класса.
- Нетривиальные методы определяйте в файле .cpp с тем же именем, что у класса.

На следующих уроках большинство наших классов будут определены в файле .cpp со всеми методами, реализованными непосредственно в теле класса. Это делается для удобства и лаконичности примеров. В реальных проектах лучше, когда классы помещаются в отдельные файлы .cpp и .h.

Урок №131. Классы и const

Из предыдущих уроков мы уже знаем, что фундаментальные типы данных (int, double, char и т.д.) можно сделать константными, используя ключевое слово const, и что все константные переменные должны быть инициализированы во время объявления. В случае с константными фундаментальными типами данных инициализация может быть копирующей, прямой или uniform:

```
1. const int value1 = 6; // копирующая инициализация
2. const int value2(8); // прямая инициализация
3. const int value3 { 11 }; // uniform-инициализация (C++11)
```

Константные объекты классов

Объекты классов можно сделать константными (используя ключевое слово const). Инициализация выполняется через конструкторы классов:

```
1. const Date date1; // инициализация через конструктор по умолчанию
2. const Date date2(12, 11, 2018); // инициализация через конструктор с
   параметрами
3. const Date date3 { 12, 11, 2018 }; // инициализация через конструктор с
   параметрами в C++11
```

Как только константный объект класса инициализируется через конструктор, то любая попытка изменить переменные-члены объекта запрещена, так как это нарушает принципы константности объекта. Запрещается как изменение переменных-членов напрямую (если они являются public), так и вызов методов (сеттеров), с помощью которых можно установить значения переменным-членам. Рассмотрим следующий класс:

```
1. class Anything
2. {
3. public:
4.     int m_value;
5.
6.     Anything(): m_value(0) { }
7.
8.     void setValue(int value) { m_value = value; }
9.     int getValue() { return m_value ; }
10.};
11.
12.int main()
13.{
14.    const Anything anything; // вызываем конструктор по умолчанию
15.
16.    anything.m_value = 7; // ошибка компиляции: нарушение const
17.    anything.setValue(7); // ошибка компиляции: нарушение const
18.
19.    return 0;
20.}
```

Строки №16-17 вызовут ошибки компиляции, так как они нарушают принципы константности объекта, пытаясь напрямую изменить переменную-член и вызывая сеттер для изменения значения переменной-члена.

Константные методы классов

Теперь рассмотрим следующую строку кода:

```
1. std::cout << anything.getValue();
```

Удивительно, но это также вызовет ошибку компиляции, хотя метод getValue() не делает ничего для изменения переменной-члена! Оказывается, константные объекты класса могут явно вызывать только *константные* методы класса, а getValue() не указан, как константный метод. **Константный метод** — это метод, который гарантирует, что не будет изменять объект или вызывать неконстантные методы класса (поскольку они могут изменить объект).

Чтобы сделать getValue() константным, нужно просто добавить ключевое слово const к прототипу функции после списка параметров, но перед телом функции:

```
1. class Anything
2. {
3. public:
4.     int m_value;
5. 
6.     Anything() { m_value= 0; }
7. 
8.     void resetValue() { m_value = 0; }
9.     void setValue(int value) { m_value = value; }
10. 
11.     int getValue() const { return m_value; } // ключевое слово const
    находится после списка параметров, но перед телом функции
12. };
```

Теперь getValue() является константным методом. Это означает, что мы можем вызывать его через любой константный объект.

Для методов, определенных вне тела класса, ключевое слово const должно использоваться как в прототипе функции (в теле класса), так и в определении функции:

```
1. class Anything
2. {
3. public:
4.     int m_value;
5. 
6.     Anything() { m_value= 0; }
7. 
8.     void resetValue() { m_value = 0; }
9.     void setValue(int value) { m_value = value; }
```

```
10.
11.     int getValue() const; // обратите внимание на ключевое слово const здесь
12. };
13.
14. int Anything::getValue() const // и здесь
15. {
16.     return m_value;
17. }
```

Кроме того, любой константный метод, который пытается изменить переменную-член или вызвать неконстантный метод класса, также приведет к ошибке компиляции, например:

```
1. class Anything
2. {
3. public:
4.     int m_value ;
5.
6.     void resetValue() const { m_value = 0; } // ошибка компиляции, константные
   методы не могут изменять переменные-члены класса
7. };
```

В этом примере метод resetValue() был установлен константным, но он пытается изменить значение `m_value`. Это вызовет ошибку компиляции.

Обратите внимание, конструкторы не могут быть константными. Это связано с тем, что они должны иметь возможность инициализировать переменные-члены класса, а константный конструктор этого не может сделать. Следовательно, в языке C++ константные конструкторы запрещены.

Стоит отметить, что константный объект класса может вызывать конструктор, который будет инициализировать все или некоторые переменные-члены, или же не будет их инициализировать вообще!

Правило: Делайте все ваши методы, которые не изменяют данные объекта класса, константными.

Константные ссылки и классы

Еще одним способом создания константных объектов является передача объектов в функцию по константной ссылке.

Мы уже рассматривали преимущества передачи аргументов по константной ссылке, нежели по значению. Если вкратце, то передача аргументов по значению создает копию значения (что является медленным процессом). Большую часть времени нам не нужна копия, а ссылка уже указывает на исходный аргумент и является более эффективной, так как избегает создания и использования ненужной копии. Мы

обычно делаем ссылку константной для гарантии того, что функция не изменит значение аргумента и сможет работать с r-values (например, с литералами).

Можете ли вы определить, что не так со следующим кодом?

```
#include <iostream>

class Date
{
private:
    int m_day;
    int m_month;
    int m_year;

public:
    Date(int day, int month, int year)
    {
        setDate(day, month, year);
    }

    void setDate(int day, int month, int year)
    {
        m_day = day;
        m_month = month;
        m_year = year;
    }

    int getDay() { return m_day; }
    int getMonth() { return m_month; }
    int getYear() { return m_year; }
};

// Примечание: Мы передаем объект date по константной ссылке, дабы избежать создания копии объекта date
void printDate(const Date &date)
{
    std::cout << date.getDay() << "." << date.getMonth() << "." << date.getYear() << '\n';
}

int main()
{
    Date date(12, 11, 2018);
    printDate(date);

    return 0;
}
```

Ответ заключается в том, что внутри функции printDate(), объект `date` рассматривается как константный. И через этот константный `date` мы вызываем методы getDay(), getMonth() и getYear(), которые являются неконстантными. Поскольку мы не можем вызывать неконстантные методы через константные объекты, то здесь мы получим ошибку компиляции.

Решение простое — сделать getDay(), getMonth() и getYear() константными:

```
1. class Date
2. {
3. private:
4.     int m_day;
5.     int m_month;
6.     int m_year;
7.
8. public:
9.     Date(int day, int month, int year)
10.    {
11.        setDate(day, month, year);
12.    }
13.
14.    // Метод setDate() не может быть const, так как изменяет значения
   переменных-членов
15.    void setDate(int day, int month, int year)
16.    {
17.        m_day = day;
18.        m_month = month;
19.        m_year = year;
20.    }
21.
22.    // Все следующие геттеры могут быть const
23.    int getDay() const { return m_day; }
24.    int getMonth() const { return m_month; }
25.    int getYear() const { return m_year; }
26. };
```

Теперь в функции printDate() константный `date` сможет вызывать getDay(), getMonth() и getYear().

Перегрузка константных и неконстантных функций

Хотя это делается не очень часто, но функцию можно перегрузить таким образом, чтобы иметь константную и неконстантную версии одной и той же функции:

```
1. #include <string>
2.
3. class Anything
4. {
5. private:
6.     std::string m_value;
7.
8. public:
9.     Anything(const std::string &value="") { m_value= value; }
10.
11.    const std::string& getValue() const { return m_value; } // getValue() для
   константных объектов
12.    std::string& getValue() { return m_value; } // getValue() для неконстантных
    объектов
13. };
```

Константная версия функции будет вызываться для константных объектов, а неконстантная версия будет вызываться для неконстантных объектов:

```
1. int main()
2. {
3.     Anything anything;
4.     anything.getValue() = "Hello!"; // вызывается неконстантный getValue()
5.
6.     const Anything anything2;
7.     anything2.getValue(); // вызывается константный getValue()
8.
9.     return 0;
10. }
```

Перегрузка метода и разделение его на константную и неконстантную версии обычно выполняется, когда возвращаемое значение должно отличаться по константности (когда требуется константа, и когда она не требуется). В примере, приведенном выше, неконстантная версия getValue() будет работать только с неконстантными объектами, но эта версия более гибкая, так как мы можем использовать её как для чтения, так и для записи `m_value` (что мы, собственно, и делаем, присваивая строку `Hello!`).

Но, когда мы не изменяем данные объекта класса, тогда вызывается константная версия getValue().

Заключение

Любой метод, который не изменяет данные объекта класса, должен быть const!

Урок №132. Статические переменные-члены класса

Из предыдущих уроков мы уже знаем, что статические переменные сохраняют свои значения и не уничтожаются даже после выхода из блока, в котором они объявлены, например:

```
#include <iostream>

int generateID()
{
    static int s_id = 0;
    return ++s_id;
}

int main()
{
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';

    return 0;
}
```

Результат выполнения программы:

```
1
2
3
```

Обратите внимание, `s_id` сохраняет свое значение после каждого вызова функции generateID().

Ключевое слово static имеет другое значение, когда речь идет о глобальных переменных - оно предоставляет им внутреннюю связь (что ограничивает их видимость/использование за пределами файла, в котором они определены). Поскольку использование глобальных переменных – это зло, то ключевое слово static в этом контексте используется не очень часто.

В языке C++ ключевое слово static можно использовать в классах: статические переменные-члены и статические методы. Мы поговорим о статических переменных-членах на этом уроке, а о статических методах на следующем.

Прежде чем мы перейдем к ключевому слову static с переменными-членами класса, давайте сначала рассмотрим следующий класс:

```
#include <iostream>

class Anything
```

```
4. {
5. public:
6.     int m_value = 3;
7. };
8.
9. int main()
10.{
11.    Anything first;
12.    Anything second;
13.
14.    first.m_value = 4;
15.
16.    std::cout << first.m_value << '\n';
17.    std::cout << second.m_value << '\n';
18.
19.    return 0;
20.}
```

При создании объекта класса, каждый объект получает свою собственную копию всех переменных-членов класса. В этом случае, поскольку мы объявили два объекта класса Anything, у нас будет две копии `m_value`: `first.m_value` и `second.m_value`. Это разные значения, следовательно, результат выполнения программы:

```
4
3
```

Переменные-члены класса можно сделать статическими, используя ключевое слово static. В отличие от обычных переменных-членов, статические переменные-члены являются общими для всех объектов класса. Рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. class Anything
4. {
5. public:
6.     static int s_value;
7. };
8.
9. int Anything::s_value = 3;
10.
11.int main()
12.{
13.    Anything first;
14.    Anything second;
15.
16.    first.s_value = 4;
17.
18.    std::cout << first.s_value << '\n';
19.    std::cout << second.s_value << '\n';
20.    return 0;
21.}
```

Результат выполнения программы:

```
4
4
```

Поскольку `s_value` является статической переменной-членом, то она является общей для всех объектов класса Anything. Следовательно, `first.s_value` - это та же переменная, что и `second.s_value`. Вышеприведенная программа показывает, что к значению, которое мы установили через первый объект, можно получить доступ и через второй объект.

Статические члены не связаны с объектами класса

Хотя вы можете получить доступ к статическим членам через разные объекты класса (как в примере, приведенном выше), но, оказывается, статические члены существуют, даже если объекты класса не созданы! Подобно глобальным переменным, они создаются при запуске программы и уничтожаются, когда программа завершает свое выполнение.

Следовательно, статические члены принадлежат классу, а не объектам этого класса. Поскольку `s_value` существует независимо от любых объектов класса, то доступ к нему осуществляется напрямую через имя класса и оператор разрешения области видимости (в данном случае, через `Anything::s_value`):

```
#include <iostream>

class Anything
{
public:
    static int s_value; // объявляем статическую переменную-член
};

int Anything::s_value = 3; // определяем статическую переменную-член

int main()
{
    // Примечание: Мы не создаем здесь никаких объектов класса Anything

    Anything::s_value = 4;
    std::cout << Anything::s_value << '\n';
    return 0;
}
```

В вышеприведенном фрагменте, доступ к `s_value` осуществляется через имя класса, а не через объект этого класса. Обратите внимание, мы даже не создавали объект класса Anything, но мы все равно имеем доступ к `Anything::s_value` и можем использовать эту переменную-член.

Определение и инициализация статических переменных-членов класса

Когда мы объявляем статическую переменную-член внутри тела класса, то мы сообщаем компилятору о существовании статической переменной-члене, но не о её определении (аналогией является предварительное объявление). Поскольку статические переменные-члены не являются частью отдельных объектов класса (они обрабатываются аналогично глобальным переменным и инициализируются при запуске программы), то вы должны явно определить статический член вне тела класса — в глобальной области видимости.

В программе, приведенной выше, это делается следующей строчкой:

```
1. int Anything::s_value = 3; // определяем статическую переменную-член
```

Здесь мы определили статическую переменную-член класса и инициализировали её значением `3`. Если же инициализатор не предоставлен, то C++ инициализирует `s_value` значением `0`.

Обратите внимание, это определение статического члена не подпадает под действия спецификаторов доступа: вы можете определить и инициализировать `s_value`, даже если он будет private (или protected).

Если класс определен в заголовочном файле, то определение статического члена обычно помещается в файл с кодом класса (например, в Anything.cpp). Если класс определен в файле .cpp, то определение статического члена обычно пишется непосредственно под классом. Не пишите определение статического члена класса в заголовочном файле (подобно глобальным переменным). Если этот заголовочный файл подключают больше одного раза, то вы получите несколько определений одного члена, что приведет к ошибке компиляции.

Инициализация статических переменных-членов внутри тела класса

Есть несколько обходных путей определения статических членов внутри тела класса. Во-первых, если статический член является константным интегральным типом (к которому относятся и char, и bool) или константным перечислением, то статический член может быть инициализирован внутри тела класса:

```
1. class Anything
2. {
3. public:
4.     static const int s_value = 5; // статическую константную переменную типа int можно объявить и инициализировать напрямую
5. };
```

Поскольку здесь статическая переменная-член является целочисленной константой, то дополнительной строки явного определения вне тела класса уже не требуется.

Во-вторых, начиная с C++11 статические члены constexpr любого типа данных, поддерживающие инициализацию constexpr, могут быть инициализированы внутри тела класса:

```
#include <array>

class Anything
{
public:
    static constexpr double s_value = 3.4; // хорошо
    static constexpr std::array<int, 3> s_array = { 3, 4, 5 }; // это работает даже с классами, которые поддерживают инициализацию constexpr
};
```

Использование статических переменных-членов класса

Зачем использовать статические переменные-члены внутри классов? Для присваивания уникального идентификатора каждому объекту класса (как вариант):

```
#include <iostream>

class Anything
{
private:
    static int s_idGenerator;
    int m_id;

public:
    Anything() { m_id = s_idGenerator++; } // увеличиваем значение идентификатора для следующего объекта

    int getID() const { return m_id; }
};

// Мы определяем и инициализируем s_idGenerator несмотря на то, что он объявлен как private.
// Это нормально, поскольку определение не подпадает под действия спецификаторов доступа
int Anything::s_idGenerator = 1; // начинаем наш ID-генератор со значения 1

int main()
{
    Anything first;
    Anything second;
    Anything third;

    std::cout << first.getID() << '\n';
    std::cout << second.getID() << '\n';
    std::cout << third.getID() << '\n';
    return 0;
}
```

Результат выполнения программы:

```
1
2
3
```

Поскольку `s_idGenerator` является общим для всех объектов класса Anything, то при создании нового объекта класса Anything конструктор захватывает текущее значение `s_idGenerator`, а затем увеличивает его для следующего объекта. Это гарантирует уникальность идентификаторов для каждого созданного объекта класса Anything.

Статические переменные-члены также могут быть полезны, когда классу необходимо использовать внутреннюю таблицу поиска (например, массив, используемый для хранения набора предварительно вычисленных значений). Делая таблицу поиска статической, для всех объектов класса создастся только одна копия (нежели отдельная для каждого объекта класса). Это поможет сэкономить значительное количество памяти.

Урок №133. Статические методы класса

Если статические переменные-члены являются открытыми, то мы можем получить к ним доступ напрямую через имя класса и оператор разрешения области видимости. Но что, если статические переменные-члены являются закрытыми? Рассмотрим следующий код:

```
1. class Anything
2. {
3. private:
4.     static int s_value;
5.
6. };
7.
8. int Anything::s_value = 3; // определение статического члена, несмотря на то, что он является private
9.
10. int main()
11. {
12.     // Как получить доступ к Anything::s_value здесь, если s_value является private?
13. }
```

В этом случае мы не можем напрямую получить доступ к `Anything::s_value` из функции main(), так как этот член является private. Обычно, доступ к закрытым членам класса осуществляется через public-методы. Хотя мы могли бы создать обычный метод для получения доступа к `s_value`, но нам тогда пришлось бы создавать объект этого класса для использования метода! Есть вариант получше: мы можем сделать метод статическим.

Подобно статическим переменным-членам, **статические методы** не привязаны к какому-либо одному объекту класса. Вот вышеприведенный пример, но уже со статическим методом:

```
1. class Anything
2. {
3. private:
4.     static int s_value;
5. public:
6.     static int getValue() { return s_value; } // статический метод
7. };
8.
9. int Anything::s_value = 3; // определение статической переменной-члена класса
10.
11. int main()
12. {
13.     std::cout << Anything::getValue() << '\n';
14. }
```

Поскольку статические методы не привязаны к определенному объекту, то их можно вызывать напрямую через имя класса и оператор разрешения области видимости, а также через объекты класса (но это не рекомендуется).

Статические методы не имеют указателя *this

У статических методов есть две интересные особенности.

Во-первых, поскольку статические методы не привязаны к объекту, то они не имеют скрытого указателя *this! Здесь есть смысл, так как указатель *this всегда указывает на объект, с которым работает метод. Статические методы могут не работать через объект, поэтому и указатель *this не нужен.

Во-вторых, статические методы могут напрямую обращаться к другим статическим членам (переменным или функциям), но не могут напрямую обращаться к нестатическим членам. Это связано с тем, что нестатические члены принадлежат объекту класса, а статические методы - нет!

Еще один пример

Статические методы можно определять вне тела класса. Это работает так же, как и с обычными методами. Например:

```
#include <iostream>

class IDGenerator
{
private:
    static int s_nextID; // объявление статической переменной-члена

public:
     static int getNextID(); // объявление статического метода
};

// Определение статической переменной-члена находится вне тела класса. Обратите внимание, что мы не используем здесь ключевое слово static.
// Начинаем генерировать ID с 1
int IDGenerator::s_nextID = 1;

// Определение статического метода находится вне тела класса. Обратите внимание, что мы не используем здесь ключевое слово static
int IDGenerator::getNextID() { return s_nextID++; }

int main()
{
    for (int count=0; count < 4; ++count)
        std::cout << "The next ID is: " << IDGenerator::getNextID() << '\n';

    return 0;
}
```

Результат выполнения программы:

```
The next ID is: 1
The next ID is: 2
The next ID is: 3
The next ID is: 4
```

Обратите внимание, поскольку все переменные и функции этого класса являются статическими, то нам не нужно создавать объект этого класса для работы с ним! Статическая переменная-член используется для хранения значения следующего идентификатора, который должен быть ей присвоен, а статический метод - для возврата идентификатора и его увеличения.

Предупреждение о классах со всеми статическими членами

Будьте осторожны при написании классов со всеми статическими членами. Хотя такие «чисто статические классы» могут быть полезны, но они также имеют свои недостатки.

Во-первых, поскольку все статические члены создаются только один раз, то несколько копий «чисто статического класса» быть не может (без клонирования класса и его дальнейшего переименования). Например, если нам нужны два независимых объекта класса IDGenerator, то это будет невозможно через «чисто статический» класс.

Во-вторых, из урока о глобальных переменных мы знаем, что глобальные переменные опасны, поскольку любая часть кода может изменить их значения и, в конечном итоге, изменит другие фрагменты, казалось бы, не связанного с этими переменными кода. То же самое справедливо и для «чисто статических» классов. Поскольку все члены принадлежат классу (а не его объектам), а классы имеют глобальную область видимости, то в «чисто статическом классе» мы объявляем глобальные функции и переменные со всеми минусами, которые они имеют.

C++ не поддерживает статические конструкторы

Если вы можете инициализировать обычную переменную-член через конструктор, то по логике вещей вы должны иметь возможность инициализировать статические переменные-члены через статический конструктор. И, хотя некоторые современные языки действительно поддерживают статические конструкторы именно для этой цели, язык C++, к сожалению, не является одним из таковых.

Если ваша статическая переменная может быть инициализирована напрямую, то конструктор не нужен: вы можете определить статическую переменную-член, даже если она является private. Мы делали это в вышеприведенном примере с `s_nextID`. Вот еще один пример:

```
class Something
{
public:
    static std::vector<char> s_mychars;
};

std::vector<char> Something::s_mychars = { 'o', 'a', 'u', 'i', 'e' }; // определяем статическую переменную-член
```

Если для инициализации вашей статической переменной-члена требуется выполнить код (например, цикл), то есть несколько разных способов это сделать. Следующий способ является лучшим из них:

```
#include <iostream>
#include <vector>

class Something
{
private:
    static std::vector<char> s_mychars;

public:
    class _nested // определяем вложенный класс с именем _nested
    {
    public:
        _nested() // конструктор _nested() инициализирует нашу статическую переменную-член
        {
            s_mychars.push_back('o');
            s_mychars.push_back('a');
            s_mychars.push_back('u');
            s_mychars.push_back('i');
            s_mychars.push_back('e');
        }
    };

    // Статический метод для вывода s_mychars
    static void getSomething() {
        for (auto const &element : s_mychars)
            std::cout << element << ' ';
    }
private:
    static _nested s_initializer; // используем статический объект класса _nested для гарантии того, что конструктор _nested() выполнится
};

std::vector<char> Something::s_mychars; // определяем нашу статическую переменную-член
Something::_nested Something::s_initializer; // определяем наш статический s_initializer, который вызовет конструктор _nested() для инициализации s_mychars

```

```
35. int main() {
36.     Something::getSomething();
37.     return 0;
38. }
```

Результат выполнения программы:

```
o a u i e
```

При определении статического члена `s_initializer` вызовется конструктор по умолчанию _nested() (так как `s_initializer` является объектом класса _nested). Мы можем использовать этот конструктор для инициализации любых статических переменных-членов класса Something. Самое крутое здесь — это то, что весь код инициализации скрыт внутри исходного класса со статическим членом.

Заключение

Статические методы могут использоваться для работы со статическими переменными-членами класса. Для работы с ними не требуется создавать объекты класса.

Классы могут быть «чисто статические» (со всеми статическими переменными-членами и статическими методами). Однако, такие классы, по сути, эквивалентны объявлению функций и переменных в глобальной области видимости, и этого следует избегать, если у вас нет на это веских причин.

Урок №134. Дружественные функции и классы

На предыдущих уроках мы говорили о том, что данные вашего класса должны быть private. Однако может возникнуть ситуация, когда у вас есть класс и функция, которая работает с этим классом, но которая не находится в его теле. Например, есть класс, в котором хранятся данные, и функция (или другой класс), которая выводит эти данные на экран. Хотя код класса и код функции вывода разделены (для упрощения поддержки кода), код функции вывода тесно связан с данными класса. Следовательно, сделав члены класса private, мы желаемого эффекта не добьемся.

В таких ситуациях есть два варианта:

- Сделать открытыми методы класса и через них функция будет взаимодействовать с классом. Однако здесь есть несколько нюансов. Во-первых, эти открытые методы нужно будет определить, на что потребуется время, и они будут загромождать интерфейс класса. Во-вторых, в классе нужно будет открыть методы, которые не всегда должны быть открытыми и предоставляющими доступ извне.
- Использовать дружественные классы и дружественные функции, с помощью которых можно будет предоставить функции вывода доступ к закрытым данным класса. Это позволит функции вывода напрямую обращаться ко всем закрытым переменным-членам и методам класса, сохраняя при этом закрытый доступ к данным класса для всех остальных функций вне тела класса! На этом уроке мы рассмотрим, как это делается.

Дружественные функции

Дружественная функция - это функция, которая имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса. Во всех других отношениях дружественная функция является обычной функцией. Ею может быть, как обычная функция, так и метод другого класса. Для объявления дружественной функции используется **ключевое слово friend** перед прототипом функции, которую вы хотите сделать дружественной классу. Неважно, объявляете ли вы её в public- или в private-зоне класса. Например:

```
1. class Anything
2. {
3. private:
4.     int m_value;
5. public:
6.     Anything() { m_value = 0; }
```

```
7.      void add(int value) { m_value += value; }
8.
9.      // Делаем функцию reset() дружественной классу Anything
10.     friend void reset(Anything &anything);
11. };
12.
13. // Функция reset() теперь является другом класса Anything
14. void reset(Anything &anything)
15. {
16.     // И мы имеем доступ к закрытым членам объектов класса Anything
17.     anything.m_value = 0;
18. }
19.
20. int main()
21. {
22.     Anything one;
23.     one.add(4); // добавляем 4 к m_value
24.     reset(one); // сбрасываем m_value в 0
25.
26.     return 0;
27. }
```

Здесь мы объявили функцию reset(), которая принимает объект класса Anything и устанавливает `m_value` значение `0`. Поскольку reset() не является членом класса Anything, то в обычной ситуации reset() не имел бы доступа к закрытым членам Anything. Однако, поскольку эта функция является дружественной классу Anything, она имеет доступ к закрытым членам Anything.

Обратите внимание, мы должны передавать объект Anything в функцию reset() в качестве параметра. Это связано с тем, что функция reset() не является методом класса. Она не имеет указателя *this и, кроме как передачи объекта, она не сможет взаимодействовать с классом.

Еще один пример:

```
1.  class Something
2.  {
3.  private:
4.      int m_value;
5.  public:
6.      Something(int value) { m_value = value; }
7.      friend bool isEqual(const Something &value1, const Something &value2);
8.  };
9.
10. bool isEqual(const Something &value1, const Something &value2)
11. {
12.     return (value1.m_value == value2.m_value);
13. }
```

Здесь мы объявили функцию isEqual() дружественной классу Something. Функция isEqual() принимает в качестве параметров два объекта класса Something. Поскольку isEqual() является другом класса Something, то функция имеет доступ ко всем

закрытым членам объектов класса Something. Функция isEqual() сравнивает значения переменных-членов двух объектов и возвращает `true`, если они равны.

Дружественные функции и несколько классов

Функция может быть другом сразу для нескольких классов, например:

```
#include <iostream>

class Humidity;

class Temperature
{
private:
    int m_temp;
public:
    Temperature(int temp=0) { m_temp = temp; }

    friend void outWeather(const Temperature &temperature, const Humidity &humidity);
};

class Humidity
{
private:
    int m_humidity;
public:
    Humidity(int humidity=0) { m_humidity = humidity; }

    friend void outWeather(const Temperature &temperature, const Humidity &humidity);
};

void outWeather(const Temperature &temperature, const Humidity &humidity)
{
    std::cout << "The temperature is " << temperature.m_temp <<
       " and the humidity is " << humidity.m_humidity << '\n';
}

int main()
{
    Temperature temp(15);
    Humidity hum(11);

    outWeather(temp, hum);

    return 0;
}
```

Здесь есть две вещи, на которые следует обратить внимание. Во-первых, поскольку функция outWeather() является другом для обоих классов, то она имеет доступ к закрытым членам обоих классов. Во-вторых, обратите внимание на следующую строку в примере, приведенном выше:

```
class Humidity;
```

Это прототип класса, который сообщает компилятору, что мы определим класс Humidity чуть позже. Без этой строчки компилятор выдал бы ошибку, что не знает, что такое Humidity при анализе прототипа дружественной функции outWeather() внутри класса Temperature. Прототипы классов выполняют ту же роль, что и прототипы функций: они сообщают компилятору об объектах, которые позднее будут определены, но которые сейчас нужно использовать. Однако, в отличие от функций, классы не имеют типа возврата или параметров, поэтому их прототипы предельно лаконичны: `ключевое слово class + имя класса + ;` (например, `class Anything;`).

Дружественные классы

Один класс может быть дружественным другому классу. Это откроет всем членам первого класса доступ к закрытым членам второго класса, например:

```
#include <iostream>

class Values
{
private:
    int m_intValue;
    double m_dValue;
public:
    Values(int intValue, double dValue)
    {
        m_intValue = intValue;
        m_dValue = dValue;
    }

    // Делаем класс Display другом класса Values
    friend class Display;
};

class Display
{
private:
    bool m_displayIntFirst;

public:
    Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

    void displayItem(Values &value)
    {
        if (m_displayIntFirst)
            std::cout << value.m_intValue << " " << value.m_dValue << '\n';
        else // или сначала выводим double
            std::cout << value.m_dValue << " " << value.m_intValue << '\n';
    }
};

int main()
{
    Values value(7, 8.4);
    Display display(false);
```

```
40.
41.     display.displayItem(value);
42.
43.     return 0;
44. }
```

Поскольку класс Display является другом класса Values, то любой из членов Display имеет доступ к private-членам Values. Результат выполнения программы:

```
8.4 7
```

> ***Примечания о дружественных классах*:**
>
> - Во-первых, даже несмотря на то, что Display является другом Values, Display не имеет прямой доступ к указателю *this объектов Values.
> - Во-вторых, даже если Display является другом Values, это не означает, что Values также является другом Display. Если вы хотите сделать оба класса дружественными, то каждый из них должен указать в качестве друга противоположный класс. Наконец, если класс A является другом B, а B является другом C, то это не означает, что A является другом C.

Будьте внимательны при использовании дружественных функций и классов, поскольку это может нарушать принципы инкапсуляции. Если детали одного класса изменятся, то детали класса-друга также будут вынуждены измениться. Следовательно, ограничивайте количество и использование дружественных функций и классов.

Дружественные методы

Вместо того, чтобы делать дружественным целый класс, мы можем сделать дружественными только определенные методы класса. Их объявление аналогично объявлениям обычных дружественных функций, за исключением имени метода с префиксом `имяКласса::` в начале (например, `Display::displayItem()`).

Переделаем наш предыдущий пример, чтобы метод Display::displayItem() был дружественным классу Values. Мы могли бы сделать следующее:

```
1. class Display; // предварительное объявление класса Display
2.
3. class Values
4. {
5. private:
6.     int m_intValue;
7.     double m_dValue;
8. public:
9.     Values(int intValue, double dValue)
10.    {
```

```
11.         m_intValue = intValue;
12.         m_dValue = dValue;
13.     }
14.
15.     // Делаем метод Display::displayItem() другом класса Values
16.     friend void Display::displayItem(Values& value); // ошибка: Values не
    видит полного определения класса Display
17. };
18.
19. class Display
20. {
21. private:
22.     bool m_displayIntFirst;
23.
24. public:
25.     Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
26.
27.     void displayItem(Values &value)
28.     {
29.         if (m_displayIntFirst)
30.             std::cout << value.m_intValue << " " << value.m_dValue << '\n';
31.         else // или выводим сначала double
32.             std::cout << value.m_dValue << " " << value.m_intValue << '\n';
33.     }
34. };
```

Однако это не сработает. Чтобы сделать метод дружественным классу, компилятор должен увидеть полное определение класса, в котором дружественный метод определяется (а не только лишь его прототип). Поскольку компилятор, прочёсывая последовательно строчки кода не увидел полного определения класса Display, но успел увидеть прототип его метода, то он выдаст ошибку в строке определения этого метода дружественным классу Values (строка №16).

Можно попытаться переместить определение класса Display выше определения класса Values:

```
1.  class Display
2.  {
3.  private:
4.      bool m_displayIntFirst;
5.
6.  public:
7.      Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
8.
9.      void displayItem(Values &value) // ошибка: Компилятор не знает, что такое
    Values
10.     {
11.         if (m_displayIntFirst)
12.             std::cout << value.m_intValue << " " << value.m_dValue << '\n';
13.         else // или выводим сначала double
14.             std::cout << value.m_dValue << " " << value.m_intValue << '\n';
15.     }
16. };
17.
18. class Values
19. {
20. private:
```

```
    int m_intValue;
    double m_dValue;
public:
    Values(int intValue, double dValue)
    {
        m_intValue = intValue;
        m_dValue = dValue;
    }

    // Делаем метод Display::displayItem() другом класса Values
    friend void Display::displayItem(Values& value);
};
```

Однако теперь мы имеем другую проблему. Поскольку метод Display::displayItem() использует ссылку на объект класса Values в качестве параметра, а мы только что перенесли определение Display выше определения Values, то компилятор будет жаловаться, что он не знает, что такое Values. Получается замкнутый круг.

К счастью, это также можно очень легко решить:

- Во-первых, для класса Values используем предварительное объявление.
- Во-вторых, переносим определение метода Display::displayItem() за пределы класса Display и размещаем его после полного определения класса Values.

Вот как это будет выглядеть:

```
#include <iostream>

class Values; // предварительное объявление класса Values

class Display
{
private:
    bool m_displayIntFirst;

public:
    Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

    void displayItem(Values &value); // предварительное объявление,
приведенное выше, требуется для этой строки
};

class Values // полное определение класса Values
{
private:
    int m_intValue;
    double m_dValue;
public:
    Values(int intValue, double dValue)
    {
        m_intValue = intValue;
        m_dValue = dValue;
    }

    // Делаем метод Display::displayItem() другом класса Values
    friend void Display::displayItem(Values& value);
```

```
};

// Теперь мы можем определить метод Display::displayItem(), которому
   требуется увидеть полное определение класса Values
void Display::displayItem(Values &value)
{
    if (m_displayIntFirst)
        std::cout << value.m_intValue << " " << value.m_dValue << '\n';
    else // или выводим сначала double
        std::cout << value.m_dValue << " " << value.m_intValue << '\n';
}

int main()
{
    Values value(7, 8.4);
    Display display(false);

    display.displayItem(value);

    return 0;
}
```

Теперь всё будет работать правильно. Хотя это может показаться несколько сложным, но этот "танец" с перемещением классов и методов нужен только потому, что мы пытаемся сделать всё в одном файле. Лучшим решением было бы поместить каждое определение класса в отдельный заголовочный файл с определениями методов в соответствующих файлах .cpp. Таким образом, все определения классов стали бы видны сразу во всех файлах .cpp, и никакого "танца" с перемещениями не понадобилось бы!

Заключение

Дружественная функция/класс - это функция/класс, которая имеет доступ к закрытым членам другого класса, как если бы она сама была членом этого класса. Это позволяет функции/классу работать в тесном контакте с другим классом, не заставляя другой класс делать открытыми свои закрытые члены.

Тест

Точка в геометрии - это позиция в пространстве. Мы можем определить точку в 3D-пространстве как набор координат `x`, `y` и `z`. Например, `Point(0.0, 1.0, 2.0)` будет точкой в координатном пространстве `x = 0.0`, `y = 1.0` и `z = 2.0`.

Вектор в физике - это величина, которая имеет длину и направление (но не положение). Мы можем определить вектор в 3D-пространстве через значения `x`, `y` и `z`, представляющие направление вектора вдоль осей `x`, `y` и `z`. Например, `Vector(1.0, 0.0, 0.0)` будет вектором, представляющим направление только вдоль положительной оси `x` длиной `1.0`.

Вектор может применятся к точке для перемещения точки на новую позицию. Это делается путем добавления направления вектора к позиции точки. Например, `Point(0.0, 1.0, 2.0) + Vector(0.0, 2.0, 0.0)` даст точку `(0.0, 3.0, 2.0)`.

Точки и векторы часто используются в компьютерной графике (точка для представления вершин фигуры, а векторы - для перемещения фигуры).

Исходя из следующей программы:

```
#include <iostream>

class Vector3D
{
private:
    double m_x, m_y, m_z;

public:
    Vector3D(double x = 0.0, double y = 0.0, double z = 0.0)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    void print()
    {
        std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";
    }
};

class Point3D
{
private:
    double m_x, m_y, m_z;

public:
    Point3D(double x = 0.0, double y = 0.0, double z = 0.0)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    void print()
    {
        std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z << ")\n";
    }

    void moveByVector(const Vector3D &v)
    {
        // Реализуйте эту функцию как дружественную классу Vector3D
    }
};

int main()
{
    Point3D p(3.0, 4.0, 5.0);
    Vector3D v(3.0, 3.0, -2.0);
```

```
48.
49.     p.print();
50.     p.moveByVector(v);
51.     p.print();
52.
53.     return 0;
54. }
```

a) Сделайте класс Point3D дружественным классу Vector3D и реализуйте метод moveByVector() в классе Point3D.

b) Вместо того, чтобы класс Point3D был дружественным классу Vector3D, сделайте метод Point3D::moveByVector() дружественным классу Vector3D.

c) Переделайте свой ответ из задания b, используя 5 отдельных файлов: Point3D.h, Point3D.cpp, Vector3D.h, Vector3D.cpp и main.cpp.

Урок №135. Анонимные объекты

В некоторых случаях в языке C++ переменная может быть нам нужна только временно. Например:

```
#include <iostream>

int add(int a, int b)
{
    int result = a + b;
    return result;
}

int main()
{
    std::cout << add(4, 2);

    return 0;
}
```

В функции add() переменная `result` используется как временная переменная. Она не выполняет особой роли, функция использует её только для возврата значения.

Есть более простой способ написать функцию add() - через анонимный объект. **Анонимный объект** - это значение без имени. Поскольку имени нет, то и способа ссылаться на этот объект за пределами места, где он создан — тоже нет. Следовательно, анонимные объекты имеют область видимости выражения и они создаются, обрабатываются и уничтожаются в пределах одного выражения.

Вот функция add(), приведенная выше, но уже с использованием анонимного объекта:

```
#include <iostream>

int add(int a, int b)
{
    return a + b; // анонимный объект создается для хранения и возврата результата выражения a + b
}

int main()
{
    std::cout << add(4, 2);

    return 0;
}
```

При вычислении выражения `a + b`, результат помещается в анонимный объект. Затем копия анонимного объекта возвращается по значению обратно в caller, и анонимный объект уничтожается.

Это работает не только с возвращаемыми значениями, но и с параметрами функции. Например, вместо этого:

```
#include <iostream>

void printResult(int value)
{
    std::cout << value;
}

int main()
{
    int result = 4 + 2;
    printResult(result);
    return 0;
}
```

Мы можем написать это:

```
#include <iostream>

void printResult(int value)
{
    std::cout << value;
}

int main()
{
    printResult(4 + 2);
    return 0;
}
```

В этом случае выражение `4 + 2` генерирует результат `6`, который помещается в анонимный объект. Затем копия этого анонимного объекта передается в функцию printResult() (которая выводит значение `6`) и уничтожается.

Обратите внимание, насколько чище стал наш код - нам не нужно засорять его временными переменными, которые используются только один раз.

Анонимные объекты класса

Хотя в вышеприведенных примерах мы использовали только фундаментальные типы данных, анонимные объекты также могут использоваться и с классами. Достаточно просто не указывать имя объекта:

```
Dollars dollars(7); // обычный объект класса
Dollars(8); // анонимный объект класса
```

В примере, приведенном выше, строка `Dollars(8);` создаст анонимный объект класса Dollars, инициализирует его значением `8`, а затем уничтожит. В этом

контексте пользы мы много не получим. Рассмотрим пример, где это может принести пользу:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    int getDollars() const { return m_dollars; }
};

void print(const Dollars &dollars)
{
   std::cout << dollars.getDollars() << " dollars.";
}

int main()
{
    Dollars dollars(7);
    print(dollars);

    return 0;
}
```

Здесь функция main() передает объект `dollars` в функцию print(). Мы можем упростить эту программу, используя анонимные объекты:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    int getDollars() const { return m_dollars; }
};

void print(const Dollars &dollars)
{
   std::cout << dollars.getDollars() << " dollars.";
}

int main()
{
    print(Dollars(7)); // здесь мы передаем анонимный объект класса Dollars

    return 0;
}
```

Результат выполнения программы:

```
7 dollars.
```

Теперь рассмотрим пример сложнее:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    int getDollars() const { return m_dollars; }
};

Dollars add(const Dollars &d1, const Dollars &d2)
{
    Dollars sum = Dollars(d1.getDollars() + d2.getDollars());
    return sum;
}

int main()
{
    Dollars dollars1(7);
    Dollars dollars2(9);
    Dollars sum = add(dollars1, dollars2);
    std::cout << "I have " << sum.getDollars() << " dollars." << std::endl;

    return 0;
}
```

В функции add() у нас есть значение переменной `sum` класса Dollars, которое используется в качестве промежуточного значения для хранения результата и его возврата. И в функции main() у нас есть значение переменной `sum` класса Dollars, которое также используется, как промежуточное.

Можно сделать проще, используя анонимные объекты:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    int getDollars() const { return m_dollars; }
};

Dollars add(const Dollars &d1, const Dollars &d2)
```

```
{
    return Dollars(d1.getDollars() + d2.getDollars()); // возвращаем анонимный объект класса Dollars
}

int main()
{
    Dollars dollars1(7);
    Dollars dollars2(9);
    std::cout << "I have " << add(dollars1, dollars2).getDollars() << " dollars." << std::endl; // выводим анонимный объект класса Dollars

    return 0;
}
```

Эта версия функции add() идентична вышеприведенной функции add(), за исключением того, что вместо отдельного объекта используется анонимный объект класса Dollars. Также обратите внимание, в функции main() мы больше не используем переменную с именем `sum`. Вместо нее мы используем возвращаемое анонимное значение из функции add()!

В результате, наша программа стала короче, чище и проще. Фактически, поскольку `dollars1` и `dollars2` используются только в одном месте, мы можем еще упростить этот код с помощью анонимных объектов:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    int getDollars() const { return m_dollars; }
};

Dollars add(const Dollars &d1, const Dollars &d2)
{
    return Dollars(d1.getDollars() + d2.getDollars()); // возвращаем анонимный объект класса Dollars
}

int main()
{
    std::cout << "I have " << add(Dollars(7), Dollars(9)).getDollars() << " dollars." << std::endl; // выводим анонимный объект класса Dollars

    return 0;
}
```

Заключение

Анонимные объекты в языке C++ используются для передачи или возврата значений без необходимости создавать большое количество временных переменных. Динамическое выделение памяти также выполняется через анонимные объекты (поэтому адрес выделенной памяти должен быть присвоен указателю, иначе мы не имели бы способа ссылаться/использовать её).

Стоит еще отметить, что анонимные объекты рассматриваются как r-values (а не как l-values, у которых есть адрес). Это означает, что анонимные объекты могут передаваться или возвращаться только по значению или по константной ссылке. В противном случае, должна использоваться переменная.

Помните, что анонимные объекты можно использовать только один раз, так как они имеют область видимости выражения. Если вам нужно ссылаться на значение в нескольких выражениях, то для этого следует использовать отдельную переменную.

Урок №136. Вложенные типы данных в классах

Рассмотрим следующий код:

```
#include <iostream>

enum FruitList
{
    AVOCADO,
    BLACKBERRY,
    LEMON
};

class Fruit
{
private:
    FruitList m_type;

public:


    Fruit(FruitList type) :
        m_type(type)
    {
    }

    FruitList getType() { return m_type;  }
};

int main()
{
    Fruit avocado(AVOCADO);

    if (avocado.getType() == AVOCADO)
        std::cout << "I am an avocado!";
    else
        std::cout << "I am not an avocado!";

    return 0;
}
```

В этой программе всё работает. Но поскольку перечисление FruitList используется в связке с классом Fruit, то немного странно, что оно существует отдельно от самого класса.

Вложенные пользовательские типы данных в классах

В отличие от функций, которые не могут быть вложены (находится внутри друг друга), в языке C++ пользовательские типы данных могут быть определены (вложены) внутри класса. Для этого нужно просто определить пользовательский тип внутри класса под соответствующим спецификатором доступа.

Вот вышеприведенная программа, но уже с FruitList, определенным внутри класса:

```
#include <iostream>

class Fruit
{
public:
    // Мы переместили FruitList внутрь класса под спецификатор доступа public
    enum FruitList
    {
        AVOCADO,
        BLACKBERRY,
        LEMON
    };

private:
    FruitList m_type;

public:


    Fruit(FruitList type) :
        m_type(type)
    {
    }

    FruitList getType() { return m_type;  }
};

int main()
{
    // Доступ к FruitList осуществляется через Fruit
    Fruit avocado(Fruit::AVOCADO);

    if (avocado.getType() == Fruit::AVOCADO)
        std::cout << "I am an avocado!";
    else
        std::cout << "I am not an avocado!";

    return 0;
}
```

Обратите внимание:

- Во-первых, FruitList теперь определен внутри тела класса.
- Во-вторых, мы определили его под спецификатором доступа public, т.е. сделали доступ к FruitList открытым.

По сути, классы работают как пространства имен для любых вложенных типов. В первом примере мы имеем доступ к перечислителю `AVOCADO` напрямую, так как `AVOCADO` определен в глобальной области видимости (мы могли бы предотвратить это, используя класс enum вместо обычного enum, и тогда доступ к `AVOCADO` осуществлялся бы через `FruitList::AVOCADO`). Теперь, поскольку FruitList

считается частью класса, доступ к перечислителю `AVOCADO` осуществляется через имя класса, например: `Fruit::AVOCADO`.

Обратите внимание, поскольку классы enum также работают как пространства имен, и если бы мы поместили класс enum (вместо обычного enum) с именем FruitList внутрь класса Fruit, то доступ к перечислителю `AVOCADO` осуществлялся бы через `Fruit::FruitList::AVOCADO`.

Другие вложенные пользовательские типы данных в классах

Хотя перечисления являются наиболее распространенным вложенным пользовательским типом данных внутри классов, язык C++ также позволяет определять и другие пользовательские типы внутри классов, такие как псевдонимы типов (typedef и type alias) и даже другие классы!

Как и любой обычный член класса, вложенный класс будет иметь доступ ко всем членам класса-оболочки (в котором он размещен). Однако вложенные классы не имеют доступа к указателю *this класса-оболочки.

Определение вложенных типов не очень распространено, но Стандартная библиотека C++ все же использует это в некоторых случаях. Об этом детально мы поговорим на соответствующем уроке.

Урок №137. Измерение времени выполнения (тайминг) кода

Иногда, в процессе написания кода, вы можете столкнуться с ситуациями, когда не будете уверены, какая из двух функций окажется более эффективной (предполагается, что конечный результат у обеих функций одинаковый). Как это определить?

Один из самых простых способов — засечь время выполнения каждого из фрагментов кода. В C++11 это делается через **библиотеку chrono**. Мы можем легко инкапсулировать весь необходимый нам функционал в класс, который затем будем использовать в наших собственных программах.

Вот класс:

```
#include <chrono> // для функций из std::chrono

class Timer
{
private:
    // Псевдонимы типов используются для удобного доступа к вложенным типам
    using clock_t = std::chrono::high_resolution_clock;
    using second_t = std::chrono::duration<double, std::ratio<1> >;

    std::chrono::time_point<clock_t> m_beg;

public:
    Timer() : m_beg(clock_t::now())
    {
    }

    void reset()
    {
        m_beg = clock_t::now();
    }

    double elapsed() const
    {
        return std::chrono::duration_cast<second_t>(clock_t::now() - m_beg).count();
    }
};
```

Для его использования нужно определить объект класса Timer в верхней части функции main() (или откуда вы хотите начинать отсчет), а затем просто вызвать метод elapsed() после части кода, которую вы проверяете:

```
int main()
{
    Timer t;

```

```
    // Здесь находится код, к которому применяется тайминг

    std::cout << "Time elapsed: " << t.elapsed() << '\n';

    return 0;
}
```

Рассмотрим реальный пример, где нужно отсортировать массив из 10 000 элементов. Воспользуемся алгоритмом сортировки методом выбора:

```
#include <iostream>
#include <array>
#include <chrono> // для функций из std::chrono

const int g_arrayElements = 10000; // общее количество всех элементов массива

class Timer
{
private:
    // Псевдонимы типов используются для удобного доступа к вложенным типам
    using clock_t = std::chrono::high_resolution_clock;
    using second_t = std::chrono::duration<double, std::ratio<1> >;

    std::chrono::time_point<clock_t> m_beg;

public:
    Timer() : m_beg(clock_t::now())
    {
    }

    void reset()
    {
        m_beg = clock_t::now();
    }

    double elapsed() const
    {
        return std::chrono::duration_cast<second_t>(clock_t::now() - m_beg).count();
    }
};

void sortArray(std::array<int, g_arrayElements> &array)
{

    // Перебираем каждый элемент массива (кроме последнего, он уже будет отсортирован к тому времени, когда мы до него доберемся)
    for (int startIndex = 0; startIndex < g_arrayElements - 1; ++startIndex)
    {
        // В переменной smallestIndex хранится индекс наименьшего значения, которое мы нашли в этой итерации.
        // Начнем с того, что наименьший элемент в этой итерации - это первый элемент (индекс 0)
        int smallestIndex = startIndex;

        // Затем ищем элемент меньше нашего smallestIndex в оставшейся части массива
        for (int currentIndex = startIndex + 1; currentIndex < g_arrayElements; ++currentIndex)
        {
```

```
            // Если нашли элемент, который меньше нашего наименьшего элемента,
            if (array[currentIndex] < array[smallestIndex])
                // то записываем/запоминаем его
                smallestIndex = currentIndex;
        }

        // smallestIndex теперь наименьший элемент в оставшейся части массива.
        // Меняем местами наше стартовое наименьшее значение с тем, которое мы обнаружили
        std::swap(array[startIndex], array[smallestIndex]);
    }
}

int main()
{
    std::array<int, g_arrayElements> array;
    for (int i = 0; i < g_arrayElements; ++i)
        array[i] = g_arrayElements - i;

    Timer t;

    sortArray(array);

    std::cout << "Time taken: " << t.elapsed() << '\n';

    return 0;
}
```

На компьютере автора результат 3-х прогонов кода составляет `0.0508`, `0.0507` и `0.0499` секунды, т.е. около `0.05` секунды.

Теперь проделаем то же самое, но с std::sort из Стандартной библиотеки C++:

```
#include <iostream>
#include <array>
#include <chrono> // для функций из std::chrono
#include <algorithm> // для std::sort()

const int g_arrayElements = 10000; // общее количество всех элементов массива

class Timer
{
private:
    // Псевдонимы типов используются для удобного доступа к вложенным типам
    using clock_t = std::chrono::high_resolution_clock;
    using second_t = std::chrono::duration<double, std::ratio<1> >;

    std::chrono::time_point<clock_t> m_beg;

public:
    Timer() : m_beg(clock_t::now())
    {
    }

    void reset()
    {
        m_beg = clock_t::now();
    }

```

```
27.     double elapsed() const
28.     {
29.         return std::chrono::duration_cast<second_t>(clock_t::now() -
    m_beg).count();
30.     }
31. };
32.
33. void sortArray(std::array<int, g_arrayElements> &array)
34. {
35.
36.     // Перебираем каждый элемент массива (кроме последнего, он уже будет
    отсортирован к тому времени, когда мы до него доберемся)
37.     for (int startIndex = 0; startIndex < g_arrayElements - 1; ++startIndex)
38.     {
39.         // В переменной smallestIndex хранится индекс наименьшего значения,
    которое мы нашли в этой итерации.
40.         // Начнем с того, что наименьший элемент в этой итерации - это первый э
    лемент (индекс 0)
41.         int smallestIndex = startIndex;
42.
43.         // Затем ищем элемент меньше нашего smallestIndex в оставшейся части
    массива
44.         for (int currentIndex = startIndex + 1; currentIndex < g_arrayElements;
     ++currentIndex)
45.         {
46.             // Если нашли элемент, который меньше нашего наименьшего элемента
47.             if (array[currentIndex] < array[smallestIndex])
48.                 // то записываем/запоминаем его
49.                 smallestIndex = currentIndex;
50.         }
51.
52.         // smallestIndex теперь является наименьшим элементом в оставшейся
    части массива.
53.         // Меняем местами наше стартовое наименьшее значение с тем, которое
    мы обнаружили
54.         std::swap(array[startIndex], array[smallestIndex]);
55.     }
56. }
57.
58. int main()
59. {
60.     std::array<int, g_arrayElements> array;
61.     for (int i = 0; i < g_arrayElements; ++i)
62.         array[i] = g_arrayElements - i;
63.
64.     Timer t;
65.
66.     std::sort(array.begin(), array.end());
67.
68.     std::cout << "Time taken: " << t.elapsed() << '\n';
69.
70.     return 0;
71. }
```

Результаты 3-х прогонов на компьютере автора составляют `0.000694`, `0.000693` и `0.000697` секунды, т.е. около `0.0007` секунды.

Таким образом, алгоритм std::sort() в 75 раз быстрее, чем сортировка, которую написали мы сами!

Что влияет на тайминг кода?

Тайминг кода является достаточно простым и прозрачным, но **ваши результаты могут существенно отличаться из-за ряда вещей**:

Во-первых, **убедитесь, что вы используете режим конфигурации Release**, а не Debug. Во время режима Debug оптимизация обычно отключена, а она может оказывать значительное влияние на результаты. Например, в конфигурации Debug, выполнение сортировки элементов массива через std::sort() на компьютере автора заняло `0.0237` секунды, что в 34 раза больше, нежели в конфигурации Release!

Во-вторых, **на результаты тайминга влияют процессы, которые ваша система может выполнять в фоновом режиме**. Для достижения наилучших результатов убедитесь, что ваша ОС не делает ничего, что интенсивно нагружает процессор, жесткий диск (например, запущен поиск файла или сканирование антивирусом) или расходует много памяти (например, вы играете в игры или работаете в фото или видео редакторе).

Выполняйте тайминг как минимум 3 раза. Если результаты одинаковые — выбираем среднее. Если один или два результата значительно отличаются друг от друга, то запустите тайминг еще несколько раз, пока не получите лучшее представление о том, какие из результатов оказались «левыми». Обратите внимание, некоторые, казалось бы, невинные вещи, такие как веб-браузеры, могут временно увеличить нагрузку на ваш процессор до 100%, когда сайт, на котором вы находитесь в фоновом режиме, выполняет целую кучу скриптов JavaScript (рекламные баннеры, запуск видео, сложная анимация и т.д.). Запуск тайминга несколько раз позволит определить, повлияло ли подобное событие на ваши результаты.

В-третьих, **при сравнении двух фрагментов кода старайтесь не запускать ничего лишнего в фоновом режиме при прогонах кода**, так как это также может повлиять на результаты тайминга. Возможно, ваш антивирус начал сканирование в фоновом режиме, или, может быть, вы решили послушать музыку на стриминговом сервисе (и это всё в перерывах между прогонами).

Рандомизация также может повлиять на тайминг. Если бы мы отсортировали массив, заполненный случайными числами, то это бы повлияло на результаты тайминга (тот факт, что числа являются рандомными). Рандомизацию использовать можно, но убедитесь, что ваше стартовое значение является фиксированным (т.е. не используйте системные часы в качестве стартового значения) и результаты рандомизации идентичны при каждом запуске. Кроме того, убедитесь, что в

фрагментах кода не используется пользовательский ввод, так как время ожидания ввода от пользователя не должно учитываться при определении эффективности кода.

Наконец, **ваши результаты действительны только для архитектуры вашего компьютера, ОС, компилятора и системных/технических характеристик**. Вы можете получить совсем другие результаты на других системах, которые имеют другие сильные и слабые стороны.

Глава №8. Итоговый тест

Вот мы и рассмотрели сердце этого туториала — объектно-ориентированное программирование в языке C++. Теперь пора закрепить полученные знания.

Теория

Классы позволяют создавать собственные типы данных, которые объединяют данные и функции, работающие с этими данными. Данные и функции внутри класса называются членами. Доступ к членам класса осуществляется через **оператор выбора членов** `.` (или через оператор `->`, если вы получаете доступ к элементу через указатель).

Спецификаторы доступа позволяют указать, кто будет иметь доступ к членам класса. Доступ к открытым (**public**) членам класса имеют все. Доступ к закрытым (**private**) членам класса имеют только другие члены класса. О **protected** мы поговорим детально, когда будем рассматривать наследование в языке C++. По умолчанию все члены класса являются private, а все члены структуры - public.

Инкапсуляция - это когда все переменные-члены вашего класса являются закрытыми, и доступ к ним напрямую невозможен. Это защищает ваш класс от неправильного/некорректного использования.

Конструктор - это специальный тип метода класса, который позволяет инициализировать объекты класса. Конструктор, который не принимает никаких параметров (или имеет все параметры по умолчанию), называется **конструктором по умолчанию**. Конструктор по умолчанию выполняется, если пользователем не предоставлены значения для инициализации. Вы всегда должны иметь по крайней мере один конструктор для выполнения в каждом из своих классов.

Список инициализации членов класса позволяет инициализировать переменные-члены из конструктора (вместо присваивания значений переменным-членам).

В C++11 **инициализация нестатических членов класса** позволяет напрямую указывать значения по умолчанию для переменных-членов при их объявлении.

До C++11 конструкторы не должны вызывать другие конструкторы (хоть это и скомпилируется, но будет работать не так, как вы ожидаете). В C++11 конструкторам разрешено вызывать другие конструкторы. Этот процесс называется **делегированием конструкторов** (или ***"цепочкой конструкторов"***).

Деструктор - это специальный тип метода класса, с помощью которого выполняется очистка класса. Именно в деструкторах следует выполнять освобождение динамически выделенной памяти.

Все методы имеют **скрытый указатель *this**, который указывает на текущий объект класса (который используется в данный момент). В большинстве случаев вам не нужно напрямую обращаться к этому указателю.

Хорошей практикой в программировании является **использование заголовочных файлов при работе с классами**, когда определения классов находятся в заголовочном файле с тем же именем, что у класса, а определения методов класса - в файле .cpp с тем же именем, что у класса.

Методы класса могут (**и должны**) **быть const**, если они не изменяют данные класса. Константные объекты класса могут вызывать только константные методы класса.

Статические переменные-члены класса являются общими для всех объектов класса. Доступ к ним можно получить как из любого объекта класса, так и непосредственно через оператор разрешения области видимости `::`.

Аналогично, **статические методы класса** — это методы, которые не имеют указателя *this. Они имеют доступ только к статическим переменным-членам класса.

Дружественные функции - это внешние функции, которые имеют доступ к закрытым членам класса.

Дружественные классы - это классы, в которых все методы являются дружественными функциями.

Анонимные объекты создаются для обработки выражений или для передачи/возврата значений.

В качестве **вложенных типов в классах** обычно используются перечисления, но также можно использовать и другие пользовательские типы данных (включая другие классы), если это необходимо.

Тайминг кода осуществляется через библиотеку chrono и позволяет засечь время выполнения определенного фрагмента кода.

Тест

Задание №1

a) Напишите класс с именем Point. В классе Point должны быть две переменные-члены типа double: `m_a` и `m_b` со значениями по умолчанию `0.0`. Напишите конструктор для этого класса и функцию вывода print().

Следующая программа:

```
#include <iostream>

int main()
{
    Point first;
    Point second(2.0, 5.0);
    first.print();
    second.print();

    return 0;
}
```

Должна выдавать следующий результат:

```
Point(0, 0)
Point(2, 5)
```

b) Теперь добавим метод distanceTo(), который будет принимать второй объект класса Point в качестве параметра и будет вычислять расстояние между двумя объектами. Учитывая две точки (`a1`, `b1`) и (`a2`, `b2`), расстояние между ними можно вычислить следующим образом: `sqrt((a1 - a2) * (a1 - a2) + (b1 - b2) * (b1 - b2))`. Функция sqrt находится в заголовочном файле cmath.

Следующая программа:

```
int main()
{
    Point first;
    Point second(2.0, 5.0);
    first.print();
    second.print();
    std::cout << "Distance between two points: " << first.distanceTo(second)
<< '\n';

    return 0;
}
```

Должна выдавать следующий результат:

```
Point(0, 0)
Point(2, 5)
Distance between two points: 5.38516
```

c) Измените функцию distanceTo() из метода класса в дружественную функцию, которая будет принимать два объекта класса Point в качестве параметров. Переименуйте эту функцию на distanceFrom().

Следующая программа:

```
1. int main()
2. {
3.     Point first;
4.     Point second(2.0, 5.0);
5.     first.print();
6.     second.print();
7.     std::cout << "Distance between two points: " << distanceFrom(first, second)
    << '\n';
8.
9.     return 0;
10.}
```

Должна выдавать следующий результат:

```
Point(0, 0)
Point(2, 5)
Distance between two points: 5.38516
```

Задание №2

Напишите деструктор для следующего класса:

```
1. #include <iostream>
2.
3. class Welcome
4. {
5. private:
6.     char *m_data;
7.
8. public:
9.     Welcome()
10.    {
11.        m_data = new char[14];
12.        const char *init = "Hello, World!";
13.        for (int i = 0; i < 14; ++i)
14.            m_data[i] = init[i];
15.    }
16.
17.    ~Welcome()
18.    {
19.        // Реализация деструктора
```

```
    }

    void print() const
    {
        std::cout << m_data;
    }

};

int main()
{
    Welcome hello;
    hello.print();

    return 0;
}
```

Задание №3

Давайте создадим генератор случайных монстров.

a) Сначала создайте перечисление MonsterType со следующими типами монстров: `Dragon`, `Goblin`, `Ogre`, `Orc`, `Skeleton`, `Troll`, `Vampire` и `Zombie` + добавьте `MAX_MONSTER_TYPES`, чтобы иметь возможность подсчитать общее количество всех перечислителей.

b) Теперь создайте класс Monster со следующими тремя атрибутами (переменными-членами): тип (MonsterType), имя (std::string) и количество здоровья (int).

c) Перечисление MonsterType является специфичным для Monster, поэтому переместите его внутрь класса под спецификатор доступа public.

d) Создайте конструктор, который позволит инициализировать все переменные-члены класса.

Следующий фрагмент кода должен скомпилироваться без ошибок:

```
int main()
{
    Monster jack(Monster::Orc, "Jack", 90);

    return 0;
}
```

e) Теперь нам нужно вывести информацию про нашего монстра. Для этого нужно конвертировать MonsterType в std::string. Добавьте функцию getTypeString(), которая будет выполнять конвертацию, и функцию вывода print().

Следующая программа:

```
1. int main()
2. {
3.     Monster jack(Monster::Orc, "Jack", 90);
4.     jack.print();
5.
6.     return 0;
7. }
```

Должна выдавать следующий результат:

```
Jack is the orc that has 90 health points.
```

f) Теперь мы уже можем создать сам генератор монстров. Для этого создайте статический класс MonsterGenerator и статический метод с именем generateMonster(), который будет возвращать случайного монстра. Пока что возвратом метода пускай будет анонимный объект: `(Monster::Orc, "Jack", 90)`.

Следующая программа:

```
1. int main()
2. {
3.     Monster m = MonsterGenerator::generateMonster();
4.     m.print();
5.
6.     return 0;
7. }
```

Должна выдавать следующий результат:

```
Jack is the orc that has 90 health points.
```

g) Теперь MonsterGenerator должен генерировать некоторые случайные атрибуты. Для этого нам понадобится генератор случайного числа. Воспользуйтесь следующей функцией:

```
1. // Генерируем случайное число между min и max (включительно).
2. // Предполагается, что srand() уже был вызван
3. int getRandomNumber(int min, int max)
4. {
5.     static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);
    // используем static, так как это значение нужно вычислить единожды
6.     // Равномерно распределяем вычисление значения из нашего диапазона
7.     return static_cast<int>(rand() * fraction * (max - min + 1) + min);
8. }
```

Поскольку MonsterGenerator будет полагаться непосредственно на эту функцию, то поместите её внутрь класса в качестве статического метода.

h) Теперь измените функцию generateMonster() для генерации случайного MonsterType (между `0` и `Monster::MAX_MONSTER_TYPES-1`) и случайного количества здоровья (от 1 до 100). Это должно быть просто. После того, как вы это сделаете, определите один статический фиксированный массив `s_names` размером 6 элементов внутри функции generateMonster() и инициализируйте его 6-ю любыми именами на ваш выбор. Добавьте возможность выбора случайного имени из этого массива.

Следующий фрагмент должен скомпилироваться без ошибок:

```
#include <ctime> // для time()
#include <cstdlib> // для rand() и srand()

int main()
{
    srand(static_cast<unsigned int>(time(0))); // используем системные часы в качестве стартового значения
    rand(); // пользователям Visual Studio: делаем сброс первого случайного числа

    Monster m = MonsterGenerator::generateMonster();
    m.print();

    return 0;
}
```

i) Почему мы объявили массив `s_names` статическим?

Задание №4

Настало время для нашего и вашего любимого задания Blackjack. На этот раз мы перепишем игру Blackjack, которую написали ранее в итоговом тесте главы №6, но уже с использованием классов!

Вот полный код без классов:

```
#include <iostream>
#include <array>
#include <ctime> // для time()
#include <cstdlib> // для rand() и srand()

enum CardSuit
{
    SUIT_CLUB,
    SUIT_DIAMOND,
    SUIT_HEART,
    SUIT_SPADE,
    MAX_SUITS
};

enum CardRank
{
```

```
17.     RANK_2,
18.     RANK_3,
19.     RANK_4,
20.     RANK_5,
21.     RANK_6,
22.     RANK_7,
23.     RANK_8,
24.     RANK_9,
25.     RANK_10,
26.     RANK_JACK,
27.     RANK_QUEEN,
28.     RANK_KING,
29.     RANK_ACE,
30.     MAX_RANKS
31. };
32.
33. struct Card
34. {
35.     CardRank rank;
36.     CardSuit suit;
37. };
38.
39. void printCard(const Card &card)
40. {
41.     switch (card.rank)
42.     {
43.         case RANK_2:        std::cout << '2'; break;
44.         case RANK_3:        std::cout << '3'; break;
45.         case RANK_4:        std::cout << '4'; break;
46.         case RANK_5:        std::cout << '5'; break;
47.         case RANK_6:        std::cout << '6'; break;
48.         case RANK_7:        std::cout << '7'; break;
49.         case RANK_8:        std::cout << '8'; break;
50.         case RANK_9:        std::cout << '9'; break;
51.         case RANK_10:       std::cout << 'T'; break;
52.         case RANK_JACK:     std::cout << 'J'; break;
53.         case RANK_QUEEN:    std::cout << 'Q'; break;
54.         case RANK_KING:     std::cout << 'K'; break;
55.         case RANK_ACE:      std::cout << 'A'; break;
56.     }
57.
58.     switch (card.suit)
59.     {
60.         case SUIT_CLUB:     std::cout << 'C'; break;
61.         case SUIT_DIAMOND:  std::cout << 'D'; break;
62.         case SUIT_HEART:    std::cout << 'H'; break;
63.         case SUIT_SPADE:    std::cout << 'S'; break;
64.     }
65. }
66.
67. void printDeck(const std::array<Card, 52> deck)
68. {
69.     for (const auto &card : deck)
70.     {
71.         printCard(card);
72.         std::cout << ' ';
73.     }
74.
75.     std::cout << '\n';
76. }
77.
78. void swapCard(Card &a, Card &b)
```

```
{
    Card temp = a;
    a = b;
    b = temp;
}

// Генерируем случайное число между min и max (включительно).
// Предполагается, что srand() уже был вызван
int getRandomNumber(int min, int max)
{
    static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);
    // используем static, так как это значение нужно вычислить единожды
    // Равномерно распределяем вычисление значения из нашего диапазона
    return static_cast<int>(rand() * fraction * (max - min + 1) + min);
}

void shuffleDeck(std::array<Card, 52> &deck)
{
    // Перебираем каждую карту в колоде
    for (int index = 0; index < 52; ++index)
    {
        // Выбираем любую случайную карту
            int swapIndex = getRandomNumber(0, 51);
            // Меняем местами с нашей текущей картой
            swapCard(deck[index], deck[swapIndex]);
        }
    }

    int getCardValue(const Card &card)
    {
        switch (card.rank)
        {
        case RANK_2:        return 2;
        case RANK_3:        return 3;
        case RANK_4:        return 4;
        case RANK_5:        return 5;
        case RANK_6:        return 6;
        case RANK_7:        return 7;
        case RANK_8:        return 8;
        case RANK_9:        return 9;
        case RANK_10:       return 10;
        case RANK_JACK:     return 10;
        case RANK_QUEEN:    return 10;
        case RANK_KING:     return 10;
        case RANK_ACE:      return 11;
        }

        return 0;
    }

    char getPlayerChoice()
    {
        std::cout << "(h) to hit, or (s) to stand: ";
        char choice;
        do
        {
            std::cin >> choice;
        } while (choice != 'h' && choice != 's');

        return choice;
    }

```

```
140.      bool playBlackjack(const std::array<Card, 52> deck)
141.      {
142.          const Card *cardPtr = &deck[0];
143.
144.          int playerTotal = 0;
145.          int dealerTotal = 0;
146.
147.          // Дилер получает одну карту
148.          dealerTotal += getCardValue(*cardPtr++);
149.          std::cout << "The dealer is showing: " << dealerTotal << '\n';
150.
151.          // Игрок получает две карты
152.          playerTotal += getCardValue(*cardPtr++);
153.          playerTotal += getCardValue(*cardPtr++);
154.
155.          // Игрок начинает
156.          while (1)
157.          {
158.              std::cout << "You have: " << playerTotal << '\n';
159.              char choice = getPlayerChoice();
160.              if (choice == 's')
161.                  break;
162.
163.              playerTotal += getCardValue(*cardPtr++);
164.
165.              // Смотрим, не проиграл ли игрок
166.              if (playerTotal > 21)
167.                  return false;
168.          }
169.
170.          // Если игрок не проиграл (у него не больше 21 очка), тогда дилер
    получает карты до тех пор, пока у него в сумме будет не меньше 17 очков
171.          while (dealerTotal < 17)
172.          {
173.              dealerTotal += getCardValue(*cardPtr++);
174.              std::cout << "The dealer now has: " << dealerTotal << '\n';
175.          }
176.
177.          // Если у дилера больше 21, то он проиграл, а игрок выиграл
178.          if (dealerTotal > 21)
179.              return true;
180.
181.          return (playerTotal > dealerTotal);
182.      }
183.
184.      int main()
185.      {
186.          srand(static_cast<unsigned int>(time(0))); // используем системные
    часы в качестве стартового значения
187.          rand(); // пользователям Visual Studio: делаем сброс первого
    случайного числа
188.
189.          std::array<Card, 52> deck;
190.
191.          // Конечно, можно было бы инициализировать каждую карту отдельно,
    но зачем? Ведь есть циклы!
192.          int card = 0;
193.          for (int suit = 0; suit < MAX_SUITS; ++suit)
194.          for (int rank = 0; rank < MAX_RANKS; ++rank)
195.          {
196.              deck[card].suit = static_cast<CardSuit>(suit);
197.              deck[card].rank = static_cast<CardRank>(rank);
```

```
198.             ++card;
199.         }
200.
201.         shuffleDeck(deck);
202.
203.         if (playBlackjack(deck))
204.             std::cout << "You win!\n";
205.         else
206.             std::cout << "You lose!\n";
207.
208.         return 0;
209.     }
```

Нехило, правда? С чего же начинать? Для начала нам нужна стратегия. Программа Blackjack состоит из 4-х частей:

- Логика работы с картами.
- Логика работы с колодами карт.
- Логика раздачи карт из колоды.
- Логика игры.

Наша стратегия заключается в том, чтобы работать над каждой из этих частей индивидуально. Таким образом, вместо конвертации целой программы за один присест, мы сделаем это спокойно за 4 шага.

Скопируйте вышеприведенный код в вашу IDE, а затем закомментируйте всё, кроме строк, содержащих #include.

а) Начнем с того, что переделаем Card из структуры в класс. Хорошей новостью является то, что класс Card очень похож на класс Monster из предыдущего задания. Алгоритм действий следующий:

- Во-первых, переместите перечисления CardSuit и CardRank внутрь класса Card под спецификатор доступа public (они неотъемлемо связаны с Card, поэтому должны находиться внутри класса).
- Во-вторых, создайте закрытые переменные-члены `m_rank` и `m_suit` для хранения значений CardRank и CardSuit.
- В-третьих, создайте открытый конструктор класса Card с инициализацией карт (переменных-членов класса). Укажите параметры по умолчанию для конструктора (используйте `MAX_RANKS` и `MAX_SUITS`).
- Наконец, переместите функции printCard() и getCardValue() внутрь класса под спецификатор доступа public (не забудьте сделать их const!).

> ***Примечание***: При использовании std::array (или std::vector), где элементами являются объекты класса, класс должен иметь конструктор по умолчанию, чтобы

> элементы могли быть инициализированы разумными значениями по умолчанию. Если вы этого не сделаете, то получите ошибку попытки ссылаться на удаленную функцию.

Следующий фрагмент кода должен скомпилироваться без ошибок:

```
#include <iostream>

int main()
{
    const Card cardQueenHearts(Card::RANK_QUEEN, Card::SUIT_HEART);
    cardQueenHearts.printCard();
    std::cout << " has the value " << cardQueenHearts.getCardValue() << '\n';

    return 0;
}
```

b) Хорошо, теперь давайте работать над классом Deck:

- Во-первых, в Deck должно быть 52 карты, поэтому создайте private-член `m_deck`, который будет фиксированным массивом с 52-мя элементами (используйте std::array).
- Во-вторых, создайте конструктор, который не принимает никаких параметров и инициализирует каждый элемент массива `m_deck` случайной картой (используйте код из функции main() с циклами for из вышеприведенного примера, где присутствует полный код). Внутри циклов создайте анонимный объект Card и присваивайте его каждому элементу массива `m_deck`.
- В-третьих, переместите функцию printDeck() в класс Deck под спецификатор доступа public (не забудьте о const).
- В-четвертых, переместите функции getRandomNumber() и swapCard() в класс Deck в качестве закрытых статических членов.
- В-пятых, переместите функцию shuffleDeck() в класс в качестве открытого члена.

> ***Подсказка***: Самой сложной частью здесь является инициализация колоды карт с использованием модифицированного кода из исходной функции main(). В следующей строке показывается, как это сделать:
>
> `m_deck[card] = Card(static_cast<Card::CardRank>(rank), static_cast<Card::CardSuit>(suit));`

Следующий фрагмент должен скомпилироваться без ошибок:

```
#include <iostream>
#include <ctime> // для time()
#include <cstdlib> // для rand() и srand()

```

```
5. int main()
6. {
7.     srand(static_cast<unsigned int>(time(0))); // используем системные часы в качестве стартового значения
8.     rand(); // пользователям Visual Studio: делаем сброс первого случайного числа
9.
10.    Deck deck;
11.    deck.printDeck();
12.    deck.shuffleDeck();
13.    deck.printDeck();
14.
15.    return 0;
16. }
```

c) Теперь нам нужен способ отследить то, какая карта будет раздаваться следующей (в исходной программе для этого используется `cardptr`):

- Во-первых, добавьте в класс Deck целочисленный член `m_cardIndex` и инициализируйте его значение `0`.
- Во-вторых, создайте открытый метод dealCard(), который будет возвращать константную ссылку на текущую карту и увеличивать `m_cardIndex`.
- В-третьих, метод shuffleDeck() также должен быть обновлен для сброса `m_cardIndex` (так как после перетасовки колоды, раздается карта, которая является верхней).

Следующий фрагмент должен скомпилироваться без ошибок:

```
1. int main()
2. {
3.     srand(static_cast<unsigned int>(time(0))); // используем системные часы в качестве стартового значения
4.     rand(); // пользователям Visual Studio: делаем сброс первого случайного числа
5.
6.     Deck deck;
7.     deck.shuffleDeck();
8.     deck.printDeck();
9.     std::cout << "The first card has value: " << deck.dealCard().getCardValue() << '\n';
10.    std::cout << "The second card has value: " << deck.dealCard().getCardValue() << '\n';
11.
12.    return 0;
13. }
```

d) Почти готово! Теперь немного самостоятельности:

- Добавьте в программу функции getPlayerChoice() и playBlackjack().
- Измените функцию playBlackjack() в соответствие с уже имеющимся классом и методами.
- Удалите лишнее и добавьте нужное в функцию main() (см. полный код выше).

Урок №138. Введение в перегрузку операторов

Из предыдущих уроков мы уже знаем, что перегрузка функций обеспечивает механизм создания и выполнения вызовов функций с одним и тем же именем, но с разными параметрами. Это позволяет одной функции работать с несколькими разными типами данных (без необходимости придумывать уникальные имена для каждой из функций).

В языке C++ операторы реализованы в виде функций. Используя перегрузку функции оператора, вы можете определить свои собственные версии операторов, которые будут работать с разными типами данных (включая классы). Использование перегрузки функции для перегрузки оператора называется **перегрузкой оператора**.

Операторы, как функции

Рассмотрим следующий фрагмент:

```
1. int a = 5;
2. int b = 6;
3. std::cout << a + b << '\n';
```

Здесь компилятор использует встроенную версию оператора плюс (`+`) для целочисленных операндов - эта функция сложит два целочисленных значения (`a` и `b`), и возвратит целочисленный результат. Когда вы видите выражение `a + b`, то думайте о нем, как о вызове функции `operator+(a, b)` (где `operator+` является именем функции).

Теперь рассмотрим следующий фрагмент:

```
1. double m = 4.0;
2. double p = 5.0;
3. std::cout << m + p << '\n';
```

Компилятор также предоставит встроенную версию оператора плюс (`+`) для операндов типа double. Выражение `m + p` приведет к вызову функции `operator+(m, p)`, а, благодаря перегрузке оператора, вызовется версия double (вместо версии int).

Теперь рассмотрим, что произойдет, если мы попытаемся добавить два объекта класса:

```
1. Mystring hello = "Hello, ";
2. Mystring world = "World!";
3. std::cout << hello + world << '\n';
```

Как вы думаете, какой будет результат? Наверное, вывод строки `Hello, World!`? Нет, результатом будет ошибка, так как класс Mystring является пользовательским типом данных, а компилятор не имеет встроенной версии operator+() для использования с операндами Mystring. Для того, чтобы сделать то, что мы хотим, нам придется написать свою версию функции operator+() и указать в ней алгоритм работы с операндами типа Mystring. То, как это сделать в коде, мы рассмотрим на следующем уроке.

Вызов перегруженных операторов

При обработке выражения, содержащего оператор, компилятор использует следующие алгоритмы действий:

- Если *все операнды* являются фундаментальных типов данных, то вызывать следует встроенные соответствующие версии операторов (если таковые существуют). Если таковых не существует, то компилятор выдаст ошибку.
- Если *какой-либо из операндов* является пользовательского типа данных (например, объект класса или типа перечисления), то компилятор будет искать версию оператора, которая работает с таким типом данных. Если компилятор не найдет ничего подходящего, то попытается выполнить конвертацию одного или нескольких операндов пользовательского типа данных в фундаментальные типы данных, чтобы таким образом он мог использовать соответствующий встроенный оператор. Если это не сработает — компилятор выдаст ошибку.

Ограничения в перегрузке операторов

Во-первых, почти любой существующий оператор в языке C++ может быть перегружен. **Исключениями являются**:

- тернарный оператор (`?:`);
- оператор sizeof;
- оператор разрешения области видимости (`::`);
- оператор выбора члена (`.`);
- указатель, как оператор выбора члена (`.*`).

Во-вторых, **вы можете перегрузить только существующие операторы**. Вы не можете создавать новые или переименовывать существующие. Например, вы не можете создать оператор `**` для выполнения операции возведения в степень.

В-третьих, **по крайней мере один из операндов перегруженного оператора должен быть пользовательского типа данных**. Это означает, что вы не можете перегрузить operator+() для выполнения операции сложения значения типа int со значением типа double. Однако вы можете перегрузить operator+() для выполнения операции сложения значения типа int с объектом класса Mystring.

В-четвертых, **изначальное количество операндов, поддерживаемых оператором, изменить невозможно**. Т.е. с бинарным оператором используются только два операнда, с унарным — только один, с тернарным — только три.

Наконец, **все операторы сохраняют свой приоритет и ассоциативность по умолчанию** (независимо от того, для чего они используются), и это не может быть изменено.

Некоторые начинающие программисты пытаются перегрузить побитовый оператор XOR (`^`) для выполнения операции возведения в степень. Однако в языке C++ у оператора `^` приоритет ниже, чем у базовых арифметических операторов (`+`, `-`, `*`, `/`), и это приведет к некорректной обработке выражений.

В математике операция возведения в степень выполняется до выполнения базовых арифметических операций, поэтому `2 + 5 ^ 2` обрабатывается как `2 + (5 ^ 2) => 2 + 25 => 27`. Однако в языке C++ у базовых арифметических операторов приоритет выше, нежели у оператора `^`, поэтому `2 + 5 ^ 2` выполнится как `(2 + 5) ^ 2 => 7 ^ 2 => 49`.

Вам нужно будет явно заключать в скобки часть с возведением в степень (например, `2 + (5 ^ 2)`) каждый раз, когда вы хотите, чтобы она выполнялась первой, что очень легко забыть и, таким образом, наделать ошибок. Поэтому проводить подобные эксперименты не рекомендуется.

> ***Примечание*:** В языке C++ для возведения в степень используется функция pow() из заголовочного файла cmath. В примере, приведенном выше, с выполнением выражения `2 + 5 ^ 2` в языке C++, имеется в виду, что вы перегрузите побитовый оператор XOR (`^`) для выполнения операции возведения в степень.

Правило: При перегрузке операторов старайтесь максимально приближенно сохранять функционал операторов в соответствии с их первоначальными применениями.

Для чего использовать перегрузку операторов? Вы можете перегрузить оператор `+` для соединения объектов вашего класса String или для выполнения операции

сложения двух объектов вашего класса Fraction. Вы можете перегрузить оператор `<<` для вывода вашего класса на экран (или записи в файл). Вы можете перегрузить оператор равенства (`==`) для сравнения двух объектов класса и т.д. Подобные применения делают перегрузку операторов одной из самых полезных особенностей в языке C++, так как это упрощает процесс работы с классами и открывает новые возможности.

Урок №139. Перегрузка операторов через дружественные функции

Арифметические операторы плюс (`+`), минус (`-`), умножение (`*`) и деление (`/`) являются одними из наиболее используемых операторов в языке C++. Все они являются бинарными, то есть работают только с двумя операндами.

Есть три разных способа перегрузки операторов:

- через дружественные функции;
- через обычные функции;
- через методы класса.

Перегрузка операторов через дружественные функции

Используя следующий класс:

```
1. class Dollars
2. {
3. private:
4.     int m_dollars;
5. 
6. public:
7.     Dollars(int dollars) { m_dollars = dollars; }
8.     int getDollars() const { return m_dollars; }
9. };
```

Перегрузим оператор плюс (`+`) для выполнения операции сложения двух объектов класса Dollars:

```
1. #include <iostream>
2. 
3. class Dollars
4. {
5. private:
6.     int m_dollars;
7. 
8. public:
9.     Dollars(int dollars) { m_dollars = dollars; }
10. 
11.     // Выполняем Dollars + Dollars через дружественную функцию
12.     friend Dollars operator+(const Dollars &d1, const Dollars &d2);
13. 
14.     int getDollars() const { return m_dollars; }
15. };
16. 
17. // Примечание: Эта функция не является методом класса!
18. Dollars operator+(const Dollars &d1, const Dollars &d2)
19. {
20.     // Используем конструктор Dollars и operator+(int, int).
```

```
21.     // Мы имеем доступ к закрытому члену m_dollars, поскольку эта функция
    является дружественной классу Dollars
22.     return Dollars(d1.m_dollars + d2.m_dollars);
23. }
24. 
25. int main()
26. {
27.     Dollars dollars1(7);
28.     Dollars dollars2(9);
29.     Dollars dollarsSum = dollars1 + dollars2;
30.     std::cout << "I have " << dollarsSum.getDollars() << " dollars."
    << std::endl;
31. 
32.     return 0;
33. }
```

Результат выполнения программы:

```
I have 16 dollars.
```

Здесь мы:

- объявили дружественную функцию operator+();
- задали в качестве параметров два операнда, с которыми хотим работать — два объекта класса Dollars;
- указали соответствующий тип возврата — Dollars;
- записали реализацию операции сложения.

Для выполнения операции сложения двух объектов класса Dollars нам нужно добавить к переменной-члену `m_dollars` первого объекта `m_dollars` второго объекта. Поскольку наша перегруженная функция operator+() является дружественной классу Dollars, то мы можем напрямую обращаться к закрытому члену `m_dollars`. Кроме того, поскольку `m_dollars` является целочисленным значением, а C++ знает, как добавлять целочисленные значения, то компилятор будет использовать встроенную версию operator+() для работы с типом int, поэтому мы можем просто указать оператор `+` в нашей операции сложения двух объектов класса Dollars.

Перегрузка оператора минус (`-`) аналогична:

```
1. #include <iostream>
2. 
3. class Dollars
4. {
5. private:
6.     int m_dollars;
7. 
8. public:
9.     Dollars(int dollars) { m_dollars = dollars; }
10. 
```

```
    // Выполняем Dollars + Dollars через дружественную функцию
    friend Dollars operator+(const Dollars &d1, const Dollars &d2);

    // Выполняем Dollars - Dollars через дружественную функцию
    friend Dollars operator-(const Dollars &d1, const Dollars &d2);

    int getDollars() const { return m_dollars; }
};

// Примечание: Эта функция не является методом класса!
Dollars operator+(const Dollars &d1, const Dollars &d2)
{
    // Используем конструктор Dollars и operator+(int, int).
    // Мы имеем доступ к закрытому члену m_dollars, поскольку эта функция является дружественной классу Dollars
    return Dollars(d1.m_dollars + d2.m_dollars);
}

// Примечание: Эта функция не является методом класса!
Dollars operator-(const Dollars &d1, const Dollars &d2)
{
    // Используем конструктор Dollars и operator-(int, int).
    // Мы имеем доступ к закрытому члену m_dollars, поскольку эта функция является дружественной классу Dollars
    return Dollars(d1.m_dollars - d2.m_dollars);
}

int main()
{
    Dollars dollars1(5);
    Dollars dollars2(3);
    Dollars dollarsSum = dollars1 - dollars2;
    std::cout << "I have " << dollarsSum.getDollars() << " dollars." << std::endl;

    return 0;
}
```

Перегрузки оператора умножения (`*`) и оператора деления (`/`) аналогичны, только вместо знака минус указываете `*` или `/`.

Дружественные функции могут быть определены внутри класса

Несмотря на то, что дружественные функции не являются членами класса, они по-прежнему могут быть определены внутри класса, если это необходимо:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    // Выполняем Dollars + Dollars через дружественную функцию.
```

```
12.         // Эта функция не рассматривается как метод класса, хотя она и
    определена внутри класса
13.     friend Dollars operator+(const Dollars &d1, const Dollars &d2)
14.     {
15.         // Используем конструктор Dollars и operator+(int, int).
16.         // Мы имеем доступ к закрытому члену m_dollars, поскольку эта функция
    является дружественной классу Dollars
17.         return Dollars(d1.m_dollars + d2.m_dollars);
18.     }
19.
20.     int getDollars() const { return m_dollars; }
21. };
22.
23. int main()
24. {
25.     Dollars dollars1(7);
26.     Dollars dollars2(9);
27.     Dollars dollarsSum = dollars1 + dollars2;
28.     std::cout << "I have " << dollarsSum.getDollars() << " dollars." <<
    std::endl;
29.
30.     return 0;
31. }
```

Не рекомендуется так делать, поскольку нетривиальные определения функций лучше записывать в отдельном файле .cpp вне тела класса.

Перегрузка операторов с операндами разных типов

Один оператор может работать с операндами разных типов. Например, мы можем добавить `Dollars(5)` к числу `5` для получения результата `Dollars(10)`.

Когда C++ обрабатывает выражение `a + b`, то `a` становится первым параметром, а `b` — вторым параметром. Когда `a` и `b` одного и того же типа данных, то не имеет значения, пишете ли вы `a + b` или `b + a` - в любом случае вызывается одна и та же версия operator+(). Однако, если операнды разных типов, то `a + b` — это уже не то же самое, что `b + a`.

Например, `Dollars(5) + 5` приведет к вызову `operator+(Dollars, int)`, а `5 + Dollars(5)` приведет к вызову `operator+(int, Dollars)`. Следовательно, всякий раз, **при перегрузке бинарных операторов для работы с операндами разных типов, нужно писать две функции - по одной на каждый случай**. Например:

```
1. #include <iostream>
2.
3. class Dollars
4. {
5. private:
6.     int m_dollars;
7.
8. public:
```

```
    Dollars(int dollars) { m_dollars = dollars; }

    // Выполняем Dollars + int через дружественную функцию
    friend Dollars operator+(const Dollars &d1, int value);

    // Выполняем int + Dollars через дружественную функцию
    friend Dollars operator+(int value, const Dollars &d1);


    int getDollars() { return m_dollars; }
};

// Примечание: Эта функция не является методом класса!
Dollars operator+(const Dollars &d1, int value)
{
    // Используем конструктор Dollars и operator+(int, int).
    // Мы имеем доступ к закрытому члену m_dollars, поскольку эта функция является дружественной классу Dollars
    return Dollars(d1.m_dollars + value);
}

// Примечание: Эта функция не является методом класса!
Dollars operator+(int value, const Dollars &d1)
{
    // Используем конструктор Dollars и operator+(int, int).
    // Мы имеем доступ к закрытому члену m_dollars, поскольку эта функция является дружественной классу Dollars
    return Dollars(d1.m_dollars + value);
}

int main()
{
    Dollars d1 = Dollars(5) + 5;
    Dollars d2 = 5 + Dollars(5);

    std::cout << "I have " << d1.getDollars() << " dollars." << std::endl;
    std::cout << "I have " << d2.getDollars() << " dollars." << std::endl;

    return 0;
}
```

Обратите внимание, обе перегруженные функции имеют одну и ту же реализацию - это потому, что они выполняют одно и то же, но просто принимают параметры в разном порядке.

Еще один пример

Рассмотрим другой пример:

```
#include <iostream>

class Values
{
private:
    int m_min; // минимальное значение, которое мы обнаружили до сих пор
    int m_max; // максимальное значение, которое мы обнаружили до сих пор

public:
```

```
10.     Values(int min, int max)
11.     {
12.         m_min = min;
13.         m_max = max;
14.     }
15.
16.     int getMin() { return m_min; }
17.     int getMax() { return m_max; }
18.
19.     friend Values operator+(const Values &v1, const Values &v2);
20.     friend Values operator+(const Values &v, int value);
21.     friend Values operator+(int value, const Values &v);
22. };
23.
24. Values operator+(const Values &v1, const Values &v2)
25. {
26.     // Определяем минимальное значение между v1 и v2
27.     int min = v1.m_min < v2.m_min ? v1.m_min : v2.m_min;
28.
29.     // Определяем максимальное значение между v1 и v2
30.     int max = v1.m_max > v2.m_max ? v1.m_max : v2.m_max;
31.
32.     return Values(min, max);
33. }
34.
35. Values operator+(const Values &v, int value)
36. {
37.     // Определяем минимальное значение между v и value
38.     int min = v.m_min < value ? v.m_min : value;
39.
40.     // Определяем максимальное значение между v и value
41.     int max = v.m_max > value ? v.m_max : value;
42.
43.     return Values(min, max);
44. }
45.
46. Values operator+(int value, const Values &v)
47. {
48.     // Вызываем operator+(Values, int)
49.     return v + value;
50. }
51.
52. int main()
53. {
54.     Values v1(11, 14);
55.     Values v2(7, 10);
56.     Values v3(4, 13);
57.
58.     Values vFinal = v1 + v2 + 6 + 9 + v3 + 17;
59.
60.     std::cout << "Result: (" << vFinal.getMin() << ", " << vFinal.getMax() <<
    ")\n";
61.
62.     return 0;
63. }
```

Класс Values отслеживает минимальное и максимальное значения. Мы перегрузили оператор плюс (`+`) 3 раза для выполнения операции сравнения двух объектов класса Values и операции сложения целочисленного значения с объектом класса Values.

Результат выполнения программы:

```
Result: (4, 17)
```

Мы получили минимальное и максимальное значения из всех, которые указали в `vFinal`. Рассмотрим детально, как обрабатывается строка `Values vFinal = v1 + v2 + 6 + 9 + v3 + 17;`:

- Приоритет оператора `+` выше приоритета оператора `=`, а ассоциативность оператора `+` слева направо, поэтому сначала вычисляется `v1 + v2`. Это приводит к вызову `operator+(v1, v2)`, которое возвращает `Values(7, 14)`.
- Следующей выполняется операция `Values(7, 14) + 6`. Это приводит к вызову `operator+(Values(7, 14), 6)`, которое возвращает `Values(6, 14)`.
- Затем выполняется `Values(6, 14) + 9`, которое возвращает `Values(6, 14)`.
- Затем `Values(6, 14) + v3` возвращает `Values(4, 14)`.
- И, наконец, `Values(4, 14) + 17` возвращает `Values(4, 17)`. Это и является нашим конечным результатом, который и присваивается `vFinal`.

Другими словами, вышеприведенное выражение обрабатывается как `Values vFinal = (((((v1 + v2) + 6) + 9) + v3) + 17)`, причем каждая последующая операция сложения возвращает объект класса Values, который становится левым операндом для следующего оператора `+`.

> ***Примечание***: Мы определили `operator+(int, Values)` вызовом `operator+(Values, int)` (см. код выше). Это может быть менее эффективным, нежели отдельная полная реализация (за счет дополнительного вызова функции), но таким образом наш код стал короче и проще в поддержке + мы уменьшили дублирование кода. Когда это возможно, то определяйте перегруженный оператор вызовом другого перегруженного оператора (как в нашем примере, приведенном выше)!

Тест

а) Напишите класс Fraction, который имеет два целочисленных члена: числитель и знаменатель. Реализуйте функцию print(), которая будет выводить дробь.

Следующий фрагмент кода:

```
1. #include <iostream>
```

```
2.
3.  int main()
4.  {
5.      Fraction f1(1, 4);
6.      f1.print();
7.
8.      Fraction f2(1, 2);
9.      f2.print();
10. }
```

Должен выдавать следующий результат:

```
1/4
1/2
```

b) Добавьте перегрузку оператора умножения (`*`) для выполнения операции умножения объекта класса Fraction на целочисленное значение и для перемножения двух объектов класса Fraction. Используйте способ перегрузки оператора через дружественную функцию.

> ***Подсказка***: Умножение двух дробей осуществляется умножением двух числителей, а затем отдельно двух знаменателей. Для выполнения операции умножения объекта на целочисленное значение, умножьте только числитель на целочисленное значение (знаменатель не трогайте).

Следующий фрагмент кода:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      Fraction f1(3, 4);
6.      f1.print();
7.
8.      Fraction f2(2, 7);
9.      f2.print();
10.
11.     Fraction f3 = f1 * f2;
12.     f3.print();
13.
14.     Fraction f4 = f1 * 3;
15.     f4.print();
16.
17.     Fraction f5 = 3 * f2;
18.     f5.print();
19.
20.     Fraction f6 = Fraction(1, 2) * Fraction(2, 3) * Fraction(3, 4);
21.     f6.print();
22. }
```

Должен выдавать следующий результат:

```
3/4
2/7
6/28
9/4
6/7
6/24
```

Дополнительное задание

c) Дробь 2/4 — это та же дробь, что и 1/2, только 1/2 не делится до минимальных неделимых значений. Мы можем уменьшить любую заданную дробь до наименьших значений, найдя наибольший общий делитель (НОД) для числителя и знаменателя, а затем выполнить деление как числителя, так и знаменателя на НОД.

Ниже приведена функция поиска НОД:

```
1. int nod(int a, int b) {
2.     return (b == 0) ? (a > 0 ? a : -a) : nod(b, a % b);
3. }
```

Добавьте эту функцию в ваш класс и реализуйте метод reduce(), который будет уменьшать дробь. Убедитесь, что дробь будет максимально и корректно уменьшена.

Следующий фрагмент кода:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     Fraction f1(3, 4);
6.     f1.print();
7.
8.     Fraction f2(2, 7);
9.     f2.print();
10.
11.    Fraction f3 = f1 * f2;
12.    f3.print();
13.
14.    Fraction f4 = f1 * 3;
15.    f4.print();
16.
17.    Fraction f5 = 3 * f2;
18.    f5.print();
19.
20.    Fraction f6 = Fraction(1, 2) * Fraction(2, 3) * Fraction(3, 4);
21.    f6.print();
22.
23.    return 0;
24.}
```

Должен выдавать следующий результат:

```
3/4
2/7
3/14
9/4
6/7
1/4
```

Урок №140. Перегрузка операторов через обычные функции

На предыдущем уроке мы перегружали operator+() через дружественную функцию:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    // Выполняем Dollars + Dollars через дружественную функцию
    friend Dollars operator+(const Dollars &d1, const Dollars &d2);

    int getDollars() const { return m_dollars; }
};

// Примечание: Эта функция не является методом класса!
Dollars operator+(const Dollars &d1, const Dollars &d2)
{
    // Используем конструктор Dollars и operator+(int, int).
    // Мы имеем доступ к закрытому члену m_dollars, так как эта функция является дружественной классу Dollars
    return Dollars(d1.m_dollars + d2.m_dollars);
}

int main()
{
    Dollars dollars1(7);
    Dollars dollars2(9);
    Dollars dollarsSum = dollars1 + dollars2;
    std::cout << "I have " << dollarsSum.getDollars() << " dollars." << std::endl;

    return 0;
}
```

Использование дружественной функции для перегрузки оператора удобно тем, что мы имеем прямой доступ ко всем членам класса, с которым работаем. В примере, приведенном выше, наша дружественная функция перегрузки оператора `+` имеет прямой доступ к закрытому члену `m_dollars` класса Dollars.

Однако, если нам не нужен доступ к членам определенного класса, мы можем перегрузить оператор и через обычную функцию. Обратите внимание, в классе Dollars присутствует геттер getDollars(), с помощью которого мы можем получить доступ к `m_dollars` извне класса.

Перепишем перегрузку оператора + через обычную функцию:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    int getDollars() const { return m_dollars; }
};

// Примечание: Эта функция не является ни методом класса, ни дружественной классу Dollars!
Dollars operator+(const Dollars &d1, const Dollars &d2)
{
    // Используем конструктор Dollars и operator+(int, int).
        // Здесь нам не нужен прямой доступ к закрытым членам класса Dollars
    return Dollars(d1.getDollars() + d2.getDollars());
}

int main()
{
    Dollars dollars1(7);
    Dollars dollars2(9);
    Dollars dollarsSum = dollars1 + dollars2;
    std::cout << "I have " << dollarsSum.getDollars() << " dollars." << std::endl;

    return 0;
}
```

Поскольку принцип перегрузки операторов через обычные и дружественные функции почти идентичен (они просто имеют разные уровни/условия доступа к закрытым членам класса), то единственное отличие заключается в том, что в случае с дружественной функцией, её нужно обязательно объявить в классе + определить вне тела класса (или в классе), в то время как обычную функцию достаточно просто определить вне тела класса, без указания дополнительного прототипа функции.

Dollars.h:

```
class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    int getDollars() const { return m_dollars; }
};

```

```
12. // Указываем прототип operator+(), чтобы иметь возможность использовать
    перегруженный оператор + в других файлах
13. Dollars operator+(const Dollars &d1, const Dollars &d2);
```

Dollars.cpp:

```
1.  #include "Dollars.h"
2.
3.  // Примечание: Эта функция не является ни методом класса, ни дружественной
    классу Dollars!
4.  Dollars operator+(const Dollars &d1, const Dollars &d2)
5.  {
6.      // Используем конструктор Dollars и operator+(int, int).
7.          // Здесь нам не нужен прямой доступ к закрытым членам класса Dollars
8.      return Dollars(d1.getDollars() + d2.getDollars());
9.  }
```

main.cpp:

```
1.  #include <iostream>
2.  #include "Dollars.h"
3.
4.  int main()
5.  {
6.      Dollars dollars1(7);
7.      Dollars dollars2(9);
8.      Dollars dollarsSum = dollars1 + dollars2; // без явного указания
    прототипа operator+() в Dollars.h эта строка не скомпилировалась бы
9.      std::cout << "I have " << dollarsSum.getDollars() << " dollars." <<
    std::endl;
10.
11.     return 0;
12. }
```

Для перегрузки операторов рекомендуется использовать обычные функции, нежели дружественные, если в классе, конечно, присутствуют геттеры (чем меньше функций касается внутренних элементов вашего класса, тем лучше). Однако не добавляйте дополнительный геттер только для того, чтобы перегрузить оператор через обычную функцию вместо дружественной! Если геттера нет по умолчанию или он не используется вообще (в нем нет необходимости), то тогда используйте перегрузку через дружественные функции.

Правило: Используйте перегрузку операторов через обычные функции, вместо дружественных, если для этого не требуется добавление дополнительных функций в класс.

Урок №141. Перегрузка операторов ввода и вывода

Для классов с множеством переменных-членов, выводить в консоль каждую переменную по отдельности может быть несколько утомительно. Например, рассмотрим следующий класс:

```
class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    double getX() { return m_x; }
    double getY() { return m_y; }
    double getZ() { return m_z; }
};
```

Если вы захотите вывести объект этого класса на экран, то вам нужно будет сделать что-то вроде следующего:

```
Point point(3.0, 4.0, 5.0);
std::cout << "Point(" << point.getX() << ", " <<
    point.getY() << ", " <<
    point.getZ() << ")";
```

Конечно, было бы проще написать отдельную функцию для вывода, которую можно было бы повторно использовать. Например, функцию print():

```
class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    double getX() { return m_x; }
    double getY() { return m_y; }
    double getZ() { return m_z; }

    void print()
    {
        std::cout << "Point(" << m_x << ", " << m_y << ", " << m_z << ")";
    }
};
```

Теперь уже намного лучше, но здесь также есть свои нюансы. Поскольку метод print() имеет тип void, то его нельзя вызывать в середине стейтмента вывода. Вместо этого стейтмент вывода придется разбить на несколько частей (строк):

```
1. int main()
2. {
3.     Point point(3.0, 4.0, 5.0);
4. 
5.     std::cout << "My point is: ";
6.     point.print();
7.     std::cout << " in Cartesian space.\n";
8. }
```

А вот если бы мы могли просто написать:

```
1. Point point(3.0, 4.0, 5.0);
2. cout << "My point is: " << point << " in Cartesian space.\n";
```

И получить тот же результат, но без необходимости разбивать стейтмент вывода на несколько строк и помнить название функции вывода. К счастью, это можно сделать, перегрузив оператор вывода `<<`.

Перегрузка оператора вывода `<<` аналогична перегрузке оператора `+` (оба являются бинарными операторами), за исключением того, что их типы различны.

Рассмотрим выражение `std::cout << point`. Если оператором является `<<`, то чем тогда являются операнды? Левым операндом является объект std::cout, а правым - объект нашего класса Point. std::cout фактически является объектом типа std::ostream, поэтому перегрузка оператора `<<` выглядит следующим образом:

```
1. // std::cout - это объект std::ostream
2. friend std::ostream& operator<< (std::ostream &out, const Point &point);
```

Реализация перегрузки оператора `<<` для нашего класса Point довольно-таки проста, так как C++ уже знает, как выводить значения типа double, а все наши переменные-члены имеют тип double, поэтому мы можем просто использовать оператор `<<` для вывода переменных-членов нашего Point. Вот класс Point, приведенный выше, но уже с перегруженным оператором `<<`:

```
1. #include <iostream>
2. 
3. class Point
4. {
5. private:
6.     double m_x, m_y, m_z;
7. 
8. public:
9.     Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
10.    {
11.    }
```

```
12.
13.     friend std::ostream& operator<< (std::ostream &out, const Point &point);
14. };
15.
16. std::ostream& operator<< (std::ostream &out, const Point &point)
17. {
18.     // Поскольку operator<< является другом класса Point, то мы имеем прямой
    доступ к членам Point
19.     out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z
    << ")";
20.
21.     return out;
22. }
23.
24. int main()
25. {
26.     Point point1(5.0, 6.0, 7.0);
27.
28.     std::cout << point1;
29.
30.     return 0;
31. }
```

Всё довольно просто. Обратите внимание, насколько проще стал стейтмент вывода по сравнению с другими стейментами из вышеприведенных примеров. Наиболее заметным отличием является то, что std::cout стал параметром `out` в нашей функции перегрузки (который затем станет ссылкой на std::cout при вызове этого оператора).

Самое интересное здесь - тип возврата. С перегрузкой арифметических операторов мы вычисляли и возвращали результат по значению. Однако, если вы попытаетесь возвратить std::ostream по значению, то получите ошибку компилятора. Это случится из-за того, что std::ostream запрещает свое копирование.

В этом случае мы возвращаем левый параметр в качестве ссылки. Это не только предотвращает создание копии std::ostream, но также позволяет нам «связать» стейтменты вывода вместе, например, `std::cout << point << std::endl;`.

Вы могли бы подумать, что, поскольку оператор `<<` не возвращает значение обратно в caller, то мы должны были бы указать тип возврата void. Но подумайте, что произойдет, если наш оператор `<<` будет возвращать void. Когда компилятор обрабатывает `std::cout << point << std::endl;`, то, учитывая правила приоритета/ассоциативности, он будет обрабатывать это выражение как `(std::cout << point) << std::endl;`. Тогда `std::cout << point` приведет к вызову функции перегрузки оператора `<<`, которая возвратит void, и вторая часть выражения будет обрабатываться как `void << std::endl;` - в этом нет смысла!

Возвращая параметр `out` в качестве возвращаемого значения выражения `(std::cout << point)` мы возвращаем std::cout, и вторая часть нашего выражения обрабатывается как `std::cout << std::endl;` - вот где сила!

Каждый раз, когда мы хотим, чтобы наши перегруженные бинарные операторы были связаны таким образом, то левый операнд обязательно должен быть возвращен (по ссылке). Возврат левого параметра по ссылке в этом случае работает, так как он передается в функцию самим вызовом этой функции, и должен оставаться даже после выполнения и возврата этой функции. Таким образом, мы можем не беспокоиться о том, что ссылаемся на что-то, что выйдет из области видимости и уничтожится после выполнения функции. Например:

```
#include <iostream>

class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    friend std::ostream& operator<< (std::ostream &out, const Point &point);
};

std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Поскольку operator<< является другом класса Point, то мы имеем прямой доступ к членам Point
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";

    return out;
}

int main()
{
    Point point1(3.0, 4.7, 5.0);
    Point point2(9.0, 10.5, 11.0);

    std::cout << point1 << " " << point2 << '\n';

    return 0;
}
```

Результат выполнения программы:

```
Point(3, 4.7, 5) Point(9, 10.5, 11)
```

Перегрузка оператора ввода >>

Также можно перегрузить и оператор ввода. Всё почти так же, как и с оператором вывода, но главное, что нужно помнить - std::cin является объектом типа std::istream. Вот наш класс Point с перегруженным оператором ввода `>>`:

```
#include <iostream>

class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    friend std::ostream& operator<< (std::ostream &out, const Point &point);
    friend std::istream& operator>> (std::istream &in, Point &point);
};

std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Поскольку operator<< является другом класса Point, то мы имеем прямой доступ к членам Point
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";

    return out;
}

std::istream& operator>> (std::istream &in, Point &point)
{
    // Поскольку operator>> является другом класса Point, то мы имеем прямой доступ к членам Point.
    // Обратите внимание, параметр point (объект класса Point) должен быть неконстантным, чтобы мы имели возможность изменить члены класса
    in >> point.m_x;
    in >> point.m_y;
    in >> point.m_z;

    return in;
}
```

Вот пример программы с использованием как перегруженного оператора `<<`, так и оператора `>>`:

```
int main()
{
    std::cout << "Enter a point: \n";

    Point point;
    std::cin >> point;

    std::cout << "You entered: " << point << '\n';

```

```
    return 0;
}
```

Предположим, что пользователь введет `4.0`, `5.5` и `8.37`, тогда результат выполнения программы:

```
You entered: Point(4, 5.5, 8.37)
```

Заключение

Перегрузка операторов `<<` и `>>` намного упрощает процесс вывода класса на экран и получение пользовательского ввода с записью в класс.

Тест

Используя класс Fraction, представленный ниже, добавьте перегрузку операторов `<<` и `>>`.

Следующий фрагмент кода:

```
int main()
{

    Fraction f1;
    std::cout << "Enter fraction 1: ";
    std::cin >> f1;

    Fraction f2;
    std::cout << "Enter fraction 2: ";
    std::cin >> f2;

    std::cout << f1 << " * " << f2 << " is " << f1 * f2 << '\n';

    return 0;
}
```

Должен выдавать следующий результат:

```
Enter fraction 1: 3/4
Enter fraction 2: 4/9
3/4 * 4/9 is 1/3
```

Вот класс Fraction:

```
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

```

```
public:
    Fraction(int numerator=0, int denominator=1):
        m_numerator(numerator), m_denominator(denominator)
    {
        // Мы поместили метод reduce() в конструктор, чтобы убедиться, что все дроби, которые у нас есть, будут уменьшены!
        // Все дроби, которые перезаписаны, должны быть повторно уменьшены
        reduce();
    }

    // Делаем функцию nod статической, чтобы она могла быть частью класса Fraction и при этом, для её использования, нам не требовалось бы создавать объект класса Fraction
    static int nod(int a, int b)
    {
        return b == 0 ? a : nod(b, a % b);
    }

    void reduce()
    {
        int nod = Fraction::nod(m_numerator, m_denominator);
        m_numerator /= nod;
        m_denominator /= nod;
    }

    friend Fraction operator*(const Fraction &f1, const Fraction &f2);
    friend Fraction operator*(const Fraction &f1, int value);
    friend Fraction operator*(int value, const Fraction &f1);

    void print()
    {
        std::cout << m_numerator << "/" << m_denominator << "\n";
    }
};

Fraction operator*(const Fraction &f1, const Fraction &f2)
{
    return Fraction(f1.m_numerator * f2.m_numerator, f1.m_denominator * f2.m_denominator);
}

Fraction operator*(const Fraction &f1, int value)
{
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}

Fraction operator*(int value, const Fraction &f1)
{
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}
```

Урок №142. Перегрузка операторов через методы класса

Перегрузка операторов через методы класса очень похожа на перегрузку операторов через дружественные функции. Но при перегрузке оператора через метод класса левым операндом становится неявный объект, на который указывает скрытый указатель *this.

Перегрузка операторов через методы классов

Вспомним, как выглядит перегрузка оператора через дружественную функцию:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    // Выполняем Dollars + int
    friend Dollars operator+(const Dollars &dollars, int value);

    int getDollars() { return m_dollars; }
};

// Примечание: Эта функция не является методом класса!
Dollars operator+(const Dollars &dollars, int value)
{
    return Dollars(dollars.m_dollars + value);
}

int main()
{
    Dollars dollars1(7);
    Dollars dollars2 = dollars1 + 3;
    std::cout << "I have " << dollars2.getDollars() << " dollars.\n";

    return 0;
}
```

Конвертация перегрузки через дружественную функцию в перегрузку через метод класса довольно-таки проста:

- Перегружаемый оператор определяется как метод класса, вместо дружественной функции (`Dollars::operator+` вместо `friend operator+`).
- Левый параметр из функции перегрузки выбрасывается, вместо него - неявный объект, на который указывает указатель *this.

- Внутри тела функции перегрузки все ссылки на левый параметр могут быть удалены (например, `dollars.m_dollars` становится `m_dollars`, который неявно ссылается на текущий объект с помощью указателя *this).

Теперь та же перегрузка оператора `+`, но уже через метод класса:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    // Выполняем Dollars + int
    Dollars operator+(int value);

    int getDollars() { return m_dollars; }
};

// Примечание: Эта функция является методом класса!
// Вместо параметра dollars в перегрузке через дружественную функцию здесь неявный параметр, на который указывает указатель *this
Dollars Dollars::operator+(int value)
{
    return Dollars(m_dollars + value);
}

int main()
{
    Dollars dollars1(7);
    Dollars dollars2 = dollars1 + 3;
    std::cout << "I have " << dollars2.getDollars() << " dollars.\n";

    return 0;
}
```

Обратите внимание, использование оператора `+` не изменяется (в обоих случаях `dollars1 + 3`), но реализация отличается. Наша дружественная функция с двумя параметрами становится методом класса с одним параметром, причем левый параметр в перегрузке через дружественную функцию (`&dollars`), в перегрузке через метод класса становится неявным объектом, на который указывает указатель *this.

Рассмотрим детально, как обрабатывается выражение `dollars1 + 3`.

В перегрузке через дружественную функцию выражение `dollars1 + 3` приводит к вызову функции `operator+(dollars1, 3)`. Здесь есть два параметра.

В перегрузке через метод класса выражение `dollars1 + 3` приводит к вызову `dollars1.operator+(3)`. Обратите внимание, здесь уже один явный параметр, а `dollars1` используется как префикс к `operator+`. Этот префикс компилятор неявно конвертирует в скрытый левый параметр, на который указывает указатель *this. Таким образом, `dollars1.operator+(3)` становится вызовом `operator+(&dollars1, 3)`, что почти идентично перегрузке через дружественную функцию.

Итак, если мы можем перегрузить оператор через дружественную функцию или через метод класса, то что тогда выбрать? Прежде чем мы дадим ответ на этот вопрос, вам нужно узнать еще несколько деталей.

Не всё может быть перегружено через дружественные функции

Операторы присваивания (`=`), индекса (`[]`), вызова функции (`()`) и выбора члена (`->`) перегружаются через методы класса — это требование языка C++.

Не всё может быть перегружено через методы класса

На уроке о перегрузке операторов ввода и вывода мы перегружали оператор вывода `<<` для класса Point через дружественную функцию:

```
#include <iostream>

class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    friend std::ostream& operator<< (std::ostream &out, const Point &point);
};

std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Поскольку operator<<() является другом класса Point, то мы имеем доступ к закрытым членам Point
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";

    return out;
}

int main()
{
    Point point1(5.0, 6.0, 7.0);

```

```
28.     std::cout << point1;
29.
30.     return 0;
31. }
```

Однако через метод класса перегрузить оператор `<<` мы не сможем. Почему? Потому что при перегрузке через метод класса в качестве левого операнда используется текущий объект. В этом случае левым операндом является объект типа std::ostream. std::ostream является частью Стандартной библиотеки C++. Мы не можем использовать std::ostream в качестве левого неявного параметра, на который бы указывал скрытый указатель *this, так как указатель *this может указывать только на текущий объект текущего класса, члены которого мы можем изменить, поэтому перегрузка оператора `<<` должна осуществляться через дружественную функцию.

Аналогично, хотя мы можем перегрузить `operator+(Dollars, int)` через метод класса (как мы делали выше), мы не можем перегрузить `operator+(int, Dollars)` через метод класса, поскольку int теперь является левым операндом, на который указатель *this указывать не может.

Перегрузка операторов через методы класса не используется, если левый операнд не является классом (например, int), или это класс, который мы не можем изменить (например, std::ostream).

Какой способ перегрузки и когда следует использовать?

В большинстве случаев язык C++ позволяет выбирать самостоятельно способ перегрузки операторов.

Но при работе с бинарными операторами, которые не изменяют левый операнд (например, operator+()), обычно используется перегрузка через обычную или дружественную функции, поскольку такая перегрузка работает для всех типов данных параметров (даже если левый операнд не является объектом класса или является объектом класса, который изменить нельзя). Перегрузка через обычную/дружественную функцию имеет дополнительное преимущество «симметрии», так как все операнды становятся явными параметрами (а не как у перегрузки через метод класса, когда левый операнд становится неявным объектом, на который указывает указатель *this).

При работе с бинарными операторами, которые изменяют левый операнд (например, operator+=()), обычно используется перегрузка через методы класса. В этих случаях левым операндом всегда является объект класса, на который указывает скрытый указатель *this.

Унарные операторы обычно тоже перегружаются через методы класса, так как в таком случае параметры не используются вообще.

Поэтому:

- Для операторов присваивания (`=`), индекса (`[]`), вызова функции (`()`) или выбора члена (`->`) используйте перегрузку через методы класса.
- Для унарных операторов используйте перегрузку через методы класса.
- Для перегрузки бинарных операторов, которые изменяют левый операнд (например, operator+=()) используйте перегрузку через методы класса, если это возможно.
- Для перегрузки бинарных операторов, которые не изменяют левый операнд (например, operator+()) используйте перегрузку через обычные/дружественные функции.

Урок №143. Перегрузка унарных операторов +, - и логического НЕ

Рассмотрим унарные операторы плюс (`+`), минус (`-`) и логическое НЕ (`!`), которые работают с одним операндом. Так как они применяются только к одному объекту, то их перегрузку следует выполнять через методы класса.

Например, **перегрузим унарный оператор минус** (`-`) для класса Dollars:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    // Выполняем -Dollars через метод класса
    Dollars operator-() const;

    int getDollars() const { return m_dollars; }
};

// Эта функция является методом класса!
Dollars Dollars::operator-() const
{
    return Dollars(-m_dollars);
}

int main()
{
    const Dollars dollars1(7);
    std::cout << "My debt is " << (-dollars1).getDollars() << " dollars.\n";

    return 0;
}
```

> ***Примечание***: Определение метода можно записать и внутри класса. Здесь мы определили его вне тела класса для лучшей наглядности.

Всё довольно-таки просто. Перегрузка отрицательного унарного оператора минус (`-`) осуществляется через метод класса, так как явные параметры в функции перегрузки отсутствуют (только неявный объект, на который указывает скрытый указатель *this). Оператор `-` возвращает объект Dollars с отрицательным значением `m_dollars`. Поскольку этот оператор не изменяет объект класса Dollars, то мы можем (и должны) сделать функцию перегрузки константной (чтобы иметь возможность использовать этот оператор и с константными объектами класса Dollars).

Обратите внимание, путаницы между отрицательным унарным оператором минус (`-`) и бинарным оператором минус (`-`) нет, так как они имеют разное количество параметров.

Вот еще один пример: оператор `!` является логическим оператором НЕ, который возвращает `true`, если результатом выражения является `false` и возвращает `false`, если результатом выражения является `true`. Обычно это применяется к переменным типа bool, чтобы проверить, являются ли они `true` или нет:

```
if (!isHappy)
    std::cout << "I am not happy!\n";
else
    std::cout << "I am so happy!\n";
```

В языке C++ значение `0` обозначает `false`, а любое другое ненулевое значение обозначает `true`, поэтому, если логический оператор `!` применять к целочисленным значениям, то он будет возвращать `true`, если значением переменной является `0`, в противном случае — `false`.

Следовательно, при работе с классами, оператор `!` будет возвращать `true`, если значением объекта класса является `false`, `0` или любое другое значение, заданное как дефолтное (по умолчанию) при инициализации, в противном случае — оператор `!` будет возвращать `false`.

В следующем примере мы рассмотрим **перегрузку унарного оператора минус** (`-`) **и оператора логического НЕ** (`!`) для класса Something:

```
#include <iostream>

class Something
{
private:
    double m_a, m_b, m_c;

public:
    Something(double a = 0.0, double b = 0.0, double c = 0.0) :
        m_a(a), m_b(b), m_c(c)
    {
    }

    // Конвертируем объект класса Something в отрицательный
    Something operator- () const
    {
        return Something(-m_a, -m_b, -m_c);
    }

    // Возвращаем true, если используются значения по умолчанию, в противном случае - false
    bool operator! () const
    {
        return (m_a == 0.0 && m_b == 0.0 && m_c == 0.0);
```

```
    }

    double getA() { return m_a; }
    double getB() { return m_b; }
    double getC() { return m_c; }
};


int main()
{
    Something something; // используем конструктор по умолчанию со значениями 0.0, 0.0, 0.0

    if (!something)
        std::cout << "Something is null.\n";
    else
        std::cout << "Something is not null.\n";

    return 0;
}
```

Здесь перегруженный оператор НЕ (`!`) возвращает `true`, если в Something используются значения по умолчанию (`0.0`, `0.0`, `0.0`).

Результат выполнения программы:

```
Something is null.
```

Если же задать любые ненулевые значения для объекта класса Something:

```
Something something(23.11, 37.1, 20.12);
```

То результатом будет:

```
Something is not null.
```

Тест

Реализуйте **перегрузку унарного оператора плюс** (`+`) для класса Something.

Урок №144. Перегрузка операторов сравнения

Принципы перегрузки операторов сравнения те же, что и в перегрузке других операторов, которые мы рассматривали на предыдущих уроках. Поскольку все операторы сравнения являются бинарными и не изменяют свои левые операнды, то выполнять перегрузку следует через дружественные функции.

Например, **перегрузим оператор равенства** `==` **и оператор неравенства** `!=` для класса Car:

```
#include <iostream>
#include <string>

class Car
{
private:
    std::string m_company;
    std::string m_model;

public:
    Car(std::string company, std::string model)
        : m_company(company), m_model(model)
    {
    }

    friend bool operator== (const Car &c1, const Car &c2);
    friend bool operator!= (const Car &c1, const Car &c2);
};

bool operator== (const Car &c1, const Car &c2)
{
    return (c1.m_company == c2.m_company &&
            c1.m_model== c2.m_model);
}

bool operator!= (const Car &c1, const Car &c2)
{
    return !(c1== c2);
}

int main()
{
    Car mustang("Ford", "Mustang");
    Car logan("Renault", "Logan");

    if (mustang == logan)
        std::cout << "Mustang and Logan are the same.\n";

    if (mustang != logan)
        std::cout << "Mustang and Logan are not the same.\n";

    return 0;
}
```

Всё просто. Поскольку результат выполнения оператора `!=` является прямо противоположным результату выполнения оператора `==`, то мы определили оператор `!=`, используя уже перегруженный оператор `==` (уменьшив, таким образом, количество кода, сложность и возможность возникновения ошибок).

А как насчет операторов `<` и `>`? Здесь нужно определиться, чем один объект класса Car может быть лучше другого объекта класса Car, и как это всё выразить в коде. Неочевидно! Поэтому здесь мы и не перегружали операторы `<` и `>`.

> ***Совет:*** Не перегружайте операторы, которые являются бесполезными для вашего класса.

Однако, операторы `<` и `>` можно использовать для сортировки списка автомобилей (объектов класса Car) в алфавитном порядке, используя члены `m_company` и `m_model`, поэтому всегда рассматривайте разные варианты.

Некоторые классы-контейнеры Стандартной библиотеки C++ требуют перегрузки оператора `<`, чтобы они могли сохранять отсортированные элементы.

Перегрузим операторы сравнения `>`, `<`, `>=` и `<=`:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    friend bool operator> (const Dollars &d1, const Dollars &d2);
    friend bool operator<= (const Dollars &d1, const Dollars &d2);

    friend bool operator< (const Dollars &d1, const Dollars &d2);
    friend bool operator>= (const Dollars &d1, const Dollars &d2);
};

bool operator> (const Dollars &d1, const Dollars &d2)
{
    return d1.m_dollars > d2.m_dollars;
}

bool operator>= (const Dollars &d1, const Dollars &d2)
{
    return d1.m_dollars >= d2.m_dollars;
}

bool operator< (const Dollars &d1, const Dollars &d2)
{
    return d1.m_dollars < d2.m_dollars;
}
```

```
32.
33. bool operator<= (const Dollars &d1, const Dollars &d2)
34. {
35.     return d1.m_dollars <= d2.m_dollars;
36. }
37.
38. int main()
39. {
40.     Dollars ten(10);
41.     Dollars seven(7);
42.
43.     if (ten > seven)
44.         std::cout << "Ten dollars are greater than seven dollars.\n";
45.     if (ten >= seven)
46.         std::cout << "Ten dollars are greater than or equal to seven dollars.
    \n";
47.     if (ten < seven)
48.         std::cout << "Seven dollars are greater than ten dollars.\n";
49.     if (ten <= seven)
50.         std::cout << "Seven dollars are greater than or equal to ten dollars.
    \n";
51.
52.     return 0;
53. }
```

Всё просто.

Но, как вы уже могли бы заметить, операторы `>` и `<=` являются логическими противоположностями, поэтому один из них можно было бы определить через второй. Та же ситуация и с `<` и `>=`. Но, поскольку определения функций перегрузки столь просты, а операторы в строке объявления функции так хорошо сочетаются с операторами в строке возврата результата, мы решили этого не делать.

Тест

Задание №1

Используя класс Dollars, приведенный выше, перепишите операторы `<` и `<=`, используя их логические противоположности.

Задание №2

Добавьте перегрузку операторов `<<` и `<` в класс Car, представленный выше, чтобы следующий фрагмент кода:

```
1. #include <iostream>
2. #include <string>
3. #include <vector>
4. #include <algorithm>
5.
6. int main()
7. {
8.     std::vector<Car> v;
```

```
9.      v.push_back(Car("Ford", "Mustang"));
10.     v.push_back(Car("Renault", "Logan"));
11.     v.push_back(Car("Ford", "Ranger"));
12.     v.push_back(Car("Renault", "Duster"));
13.
14.     std::sort(v.begin(), v.end()); // требуется перегрузка оператора < для
    класса Car
15.
16.     for (auto &car : v)
17.         std::cout << car << '\n'; // требуется перегрузка оператора << для
    класса Car
18.
19.     return 0;
20. }
```

Выдавал следующий результат:

```
(Ford, Mustang)
(Ford, Ranger)
(Renault, Duster)
(Renault, Logan)
```

Урок №145. Перегрузка операторов инкремента и декремента

Перегрузка операторов инкремента (`++`) и декремента (`--`) довольно-таки проста, но с одним маленьким нюансом. Есть две версии операторов инкремента и декремента: версия префикс (например, `++x`, `--y`) и версия постфикс (например, `x++`, `y--`).

Поскольку операторы инкремента и декремента являются унарными и изменяют свои операнды, то перегрузку следует выполнять через методы класса.

Перегрузка операторов инкремента и декремента версии префикс

Перегрузка операторов инкремента и декремента версии префикс аналогична перегрузке любых других унарных операторов:

```
#include <iostream>

class Number
{
private:
    int m_number;
public:
    Number(int number=0)
        : m_number(number)
    {
    }

    Number& operator++();
    Number& operator--();

    friend std::ostream& operator<< (std::ostream &out, const Number &n);
};

Number& Number::operator++()
{
    // Если значением переменной m_number является 8, то выполняем сброс значения m_number на 0
    if (m_number == 8)
        m_number = 0;
    // В противном случае, просто увеличиваем m_number на единицу
    else
        ++m_number;

    return *this;
}

Number& Number::operator--()
{
    // Если значением переменной m_number является 0, то присваиваем m_number значение 8
    if (m_number == 0)
        m_number = 8;
```

```
36.     // В противном случае, просто уменьшаем m_number на единицу
37.     else
38.         --m_number;
39.
40.     return *this;
41. }
42.
43. std::ostream& operator<< (std::ostream &out, const Number &n)
44. {
45.     out << n.m_number;
46.     return out;
47. }
48.
49. int main()
50. {
51.     Number number(7);
52.
53.     std::cout << number;
54.     std::cout << ++number;
55.     std::cout << ++number;
56.     std::cout << --number;
57.     std::cout << --number;
58.
59.     return 0;
60. }
```

Здесь класс Number содержит число от `0` до `8`. Мы перегрузили операторы инкремента/декремента таким образом, чтобы они увеличивали/уменьшали `m_number` в соответствии с заданным диапазоном (если выполняется инкремент и `m_number` равно `8`, то сбрасываем значение `m_number` на `0`; если выполняется декремент и `m_number` равно `0`, то присваиваем значение `8` переменной `m_number`).

Результат выполнения программы:

```
78087
```

Обратите внимание, мы возвращаем скрытый указатель *this в функциях перегрузки операторов (т.е. текущий объект класса Number). Таким образом мы можем связать выполнение нескольких операторов в одну «цепочку».

Перегрузка операторов инкремента и декремента версии постфикс

Обычно перегрузка функций осуществляется, если они имеют одно и то же имя, но разное количество и типы параметров. Рассмотрим случай с операторами инкремента/декремента версий префикс/постфикс. Оба имеют одно и то же имя (например, `operator++`), унарные и принимают один параметр одного и того же типа данных. Как же тогда их различить при перегрузке?

Дело в том, что язык C++ использует **фиктивную переменную** (или ***«фиктивный параметр»***) для операторов версии постфикс. Этот фиктивный целочисленный параметр используется только с одной целью: отличить версию постфикс операторов инкремента/декремента от версии префикс. **Выполним перегрузку операторов инкремента/декремента версии префикс и постфикс в одном классе**:

```
#include <iostream>

class Number
{
private:
    int m_number;
public:
    Number(int number=0)
        : m_number(number)
    {
    }

    Number& operator++(); // версия префикс
    Number& operator--(); // версия префикс

    Number operator++(int); // версия постфикс
    Number operator--(int); // версия постфикс

    friend std::ostream& operator<< (std::ostream &out, const Number &n);
};

Number& Number::operator++()
{
    // Если значением переменной m_number является 8, то выполняем сброс значения m_number на 0
    if (m_number == 8)
        m_number = 0;
    // В противном случае, просто увеличиваем m_number на единицу
    else
        ++m_number;

    return *this;
}

Number& Number::operator--()
{
    // Если значением переменной m_number является 0, то присваиваем m_number значение 8
    if (m_number == 0)
        m_number = 8;
    // В противном случае, просто уменьшаем m_number на единицу
    else
        --m_number;

    return *this;
}

Number Number::operator++(int)
{
    // Создаем временный объект класса Number с текущим значением переменной m_number
    Number temp(m_number);

```

```
51.     // Используем оператор инкремента версии префикс для реализации перегрузки оператора инкремента версии постфикс
52.     ++(*this); // реализация перегрузки
53.
54.     // Возвращаем временный объект
55.     return temp;
56. }
57.
58. Number Number::operator--(int)
59. {
60.     // Создаем временный объект класса Number с текущим значением переменной m_number
61.     Number temp(m_number);
62.
63.     // Используем оператор декремента версии префикс для реализации перегрузки оператора декремента версии постфикс
64.     --(*this); // реализация перегрузки
65.
66.     // Возвращаем временный объект
67.     return temp;
68. }
69.
70. std::ostream& operator<< (std::ostream &out, const Number &n)
71. {
72.     out << n.m_number;
73.     return out;
74. }
75.
76. int main()
77. {
78.     Number number(6);
79.
80.     std::cout << number;
81.     std::cout << ++number; // вызывается Number::operator++();
82.     std::cout << number++; // вызывается Number::operator++(int);
83.     std::cout << number;
84.     std::cout << --number; // вызывается Number::operator--();
85.     std::cout << number--; // вызывается Number::operator--(int);
86.     std::cout << number;
87.
88.     return 0;
89. }
```

Результат выполнения программы:

```
6778776
```

Здесь есть несколько интересных моментов:

- Во-первых, мы отделили версию постфикс от версии префикс использованием целочисленного фиктивного параметра в версии постфикс.
- Во-вторых, поскольку фиктивный параметр не используется в реализации самой перегрузки, то мы даже не предоставляем ему имя. Таким образом, компилятор будет рассматривать эту переменную, как простую заглушку (заполнитель места), и даже не будет предупреждать нас о том, что мы объявили переменную, но никогда её не использовали.

- В-третьих, операторы версий префикс и постфикс выполняют одно и то же задание: оба увеличивают/уменьшают значение переменной объекта. Разница между ними только в значении, которое они возвращают.

Рассмотрим последний пункт детально. Операторы версии префикс возвращают объект после того, как он был увеличен или уменьшен. В версии постфикс нам нужно возвращать объект до того, как он будет увеличен или уменьшен. И тут конфуз! Если мы увеличиваем или уменьшаем объект, то мы не можем возвратить его до выполнения инкремента/декремента, так как операция увеличения/уменьшения уже произошла. С другой стороны, если мы возвращаем объект до выполнения инкремента/декремента, то сама операция увеличения/уменьшения объекта не выполнится.

Решением является использование временного объекта с текущим значением переменной-члена. Тогда можно будет увеличить/уменьшить исходный объект, а временный объект возвратить обратно в caller. Таким образом, caller получит копию объекта до того, как фактический объект будет увеличен или уменьшен, и сама операция инкремента/декремента выполнится успешно.

Обратите внимание, это означает, что возврат значения по ссылке невозможен, так как мы не можем возвратить ссылку на локальную переменную (объект), которая будет уничтожена после завершения выполнения тела функции. Также это означает, что операторы версии постфикс обычно менее эффективны, чем операторы версии префикс, из-за дополнительных расходов ресурсов на создание временного объекта и выполнения возврата по значению вместо возврата по ссылке.

Наконец, мы реализовали перегрузку операторов версии постфикс через уже перегруженные операторы версии префикс. Таким образом, мы сократили дублированный код и упростили внесение изменений в наш класс в будущем (т.е. упростили поддержку кода).

Урок №146. Перегрузка оператора индексации []

При работе с массивами **оператор индексации** (`[]`) используется для выбора определенных элементов:

```
myArray[0] = 8; // помещаем значение 8 в первый элемент массива
```

Рассмотрим следующий класс IntArray, в котором в качестве переменной-члена используется массив:

```
#include <iostream>

class IntArray
{
private:
    int m_array[10];
};

int main()
{
    IntArray array;
    // Как получить доступ к элементам m_array?
    return 0;
}
```

Поскольку переменная-член `m_array` является закрытой, то мы не имеем прямого доступа к `m_array` через объект `array`. Это означает, что мы не можем напрямую получить или установить значения элементов `m_array`. Что делать?

Можно использовать геттеры и сеттеры:

```
class IntArray
{
private:
    int m_array[10];

public:
    void setItem(int index, int value) { m_array[index] = value; }
    int getItem(int index) { return m_array[index]; }
};
```

Хотя это работает, но это не очень удобно. Рассмотрим следующий пример:

```
int main()
{
    IntArray array;
    array.setItem(4, 5);

    return 0;
}
```

Присваиваем ли мы элементу 4 значение `5` или элементу 5 значение `4`? Без просмотра определения метода setItem() этого не понять.

Можно также просто возвращать весь массив (`m_array`) и использовать оператор `[]` для доступа к его элементам:

```
#include <iostream>

class IntArray
{
private:
    int m_array[10];

public:
    int* getArray() { return m_array; }
};

int main()
{
    IntArray array;
    array.getArray()[4] = 5;

    return 0;
}
```

Но можно сделать еще проще, перегрузив оператор индексации.

Оператор индексации является одним из операторов, перегрузка которого должна выполняться через метод класса. Функция перегрузки оператора `[]` всегда будет принимать один параметр: значение индекса (элемент массива, к которому требуется доступ). В нашем случае с IntArray нам нужно, чтобы пользователь просто указал в квадратных скобках индекс для возврата значения элемента по этому индексу:

```
#include <iostream>

class IntArray
{
private:
    int m_array[10];

public:
    int& operator[] (const int index);
};

int& IntArray::operator[] (const int index)
{
    return m_array[index];
}
```

Теперь всякий раз, когда мы будем использовать оператор индексации (`[]`) с объектом класса IntArray, компилятор будет возвращать соответствующий элемент

массива `m_array`! Это позволит нам непосредственно как получать, так и присваивать значения элементам `m_array`:

```
1. int main()
2. {
3.     IntArray array;
4.     array[4] = 5; // присваиваем значение
5.     std::cout << array[4]; // выводим значение
6.
7.     return 0;
8. }
```

Всё просто. При обработке `array[4]` компилятор сначала проверяет, есть ли функция перегрузки оператора `[]`. Если есть, то он передает в функцию перегрузки значение внутри квадратных скобок (в данном случае `4`) в качестве аргумента.

Обратите внимание, хотя мы можем указать параметр по умолчанию для функции перегрузки оператора `[]`, но в данном случае, если мы, используя `[]`, не укажем внутри скобок значение индекса, то получим ошибку.

Почему оператор индексации [] использует возврат по ссылке?

Рассмотрим подробнее, как обрабатывается стейтмент `array[4] = 5`. Поскольку приоритет оператора индексации выше приоритета оператора присваивания, то сначала выполняется часть `array[4]`. `array[4]` приводит к вызову функции перегрузки оператора `[]`, которая возвратит `array.m_array[4]`. Поскольку оператор `[]` использует возврат по ссылке, то он возвращает фактический элемент `array.m_array[4]`. Наше частично обработанное выражение становится `array.m_array[4] = 5`, что является прямой операцией присваивания значения элементу массива.

Из предыдущих уроков мы уже знаем, что любое значение, которое находится слева от оператора присваивания, должно быть l-value (переменной с адресом в памяти). Поскольку результат выполнения оператора `[]` может использоваться в левой части операции присваивания (например, `array[4] = 5`), то возвращаемое значение оператора `[]` должно быть l-value. Ссылки же всегда являются l-values, так как их можно использовать только с переменными, которые имеют адреса памяти. Поэтому, используя возврат по ссылке, компилятор останется доволен, что возвращается l-value, и никаких проблем не будет.

Рассмотрим, что произойдет, если оператор `[]` будет использовать возврат по значению вместо возврата по ссылке. `array[4]` приведет к вызову функции перегрузки оператора `[]`, который будет возвращать значение элемента `array.m_array[4]` (не индекс, а значение по указанному индексу). Например,

если значением `m_array[4]` является `7`, то выполнение оператора `[]` приведет к возврату значения `7`. `array[4] = 5` будет обрабатываться как `7 = 5`, что является бессмысленным! Если вы попытаетесь это сделать, то компилятор выдаст следующую ошибку:

```
C:VCProjectsTest.cpp(386) : error C2106: '=' : left operand must be l-value
```

Использование оператора индексации с константными объектами класса

В вышеприведенном примере метод operator[]() не является константным, и мы можем использовать этот метод для изменения данных неконстантных объектов. Однако, что произойдет, если наш объект класса IntArray будет const? В этом случае мы не сможем вызывать неконстантный operator[](), так как он изменяет значения объекта.

Хорошей новостью является то, что мы можем определить отдельно неконстантную и константную версии operator[](). Неконстантная версия будет использоваться с неконстантными объектами, а версия const — с объектами const:

```
#include <iostream>

class IntArray
{
private:
    int m_array[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // указываем начальные значения

public:
    int& operator[] (const int index);
    const int& operator[] (const int index) const;
};

int& IntArray::operator[] (const int index) // для неконстантных объектов: может использоваться как для присваивания значений элементам, так и для их просмотра
{
    return m_array[index];
}

const int& IntArray::operator[] (const int index) const // для константных объектов: используется только для просмотра (вывода) элементов массива
{
    return m_array[index];
}

int main()
{
    IntArray array;
    array[4] = 5; // хорошо: вызывается неконстантная версия operator[]()
    std::cout << array[4];

    const IntArray carray;
```

```
30.     carray[4] = 5; // ошибка компиляции: вызывается константная версия
    operator[](), которая возвращает константную ссылку. Выполнять операцию
    присваивания нельзя
31.     std::cout << carray[4];
32.
33.     return 0;
34. }
```

Строку `carray[4] = 5;` нужно закомментировать и программа скомпилируется (это проверка на изменение данных константных объектов — изменять данные нельзя, можно только выводить).

Проверка ошибок

Еще одним преимуществом перегрузки оператора индексации является то, что мы можем выполнять проверку передаваемых значений индекса. При прямом доступе к элементам массива (через геттеры и сеттеры), оператор индекса не проверяет, является ли индекс корректным. Например, компилятор не будет жаловаться на следующий код:

```
1. int array[7];
2. array[9] = 4; // индекс 9 является некорректным (вне допустимого диапазона)!
```

Однако, если мы знаем длину нашего массива, мы можем выполнять проверку передаваемого индекса на корректность в функции перегрузки оператора `[]`:

```
1. #include <cassert> // для assert()
2.
3. class IntArray
4. {
5. private:
6.     int m_array[10];
7.
8. public:
9.     int& operator[] (const int index);
10. };
11.
12. int& IntArray::operator[] (const int index)
13. {
14.     assert(index >= 0 && index < 10);
15.
16.     return m_array[index];
17. }
```

В примере, приведенном выше, мы использовали стейтмент assert (который находится в заголовочном файле cassert) для проверки диапазона `index`. Если выражение внутри assert принимает значение `false` (т.е. пользователь ввел некорректный индекс), то программа немедленно завершится с выводом сообщения об ошибке, что лучше, нежели альтернативный вариант - повреждение

памяти. Это самый распространенный способ проверки ошибок с использованием функций перегрузки.

Указатели на объекты и перегруженный оператор []

Если вы попытаетесь вызвать operator[]() для указателя на объект, то C++ предположит, что вы пытаетесь индексировать массив. Рассмотрим следующий пример:

```
#include <cassert> // для assert()

class IntArray
{
private:
    int m_array[10];

public:
    int& operator[] (const int index);
};

int& IntArray::operator[] (const int index)
{
    assert(index >= 0 && index < 10);

    return m_array[index];
}

int main()
{
    IntArray *array = new IntArray;
    array[4] = 5; // ошибка
    delete array;

    return 0;
}
```

Дело в том, что указатель указывает на адрес памяти, а не на значение. Поэтому сначала указатель нужно разыменовать, а затем уже использовать оператор `[]`:

```
int main()
{
    IntArray *array = new IntArray;
    (*array)[4] = 5; // сначала разыменовываем указатель для получения объекта array, а затем вызываем operator[]
    delete array;

    return 0;
}
```

Это ужасно и здесь очень легко наделать ошибок. Не используйте указатели на объекты, если это не является обязательным.

Передаваемый аргумент не обязательно должен быть целым числом

Как упоминалось выше, C++ передает в функцию перегрузки то, что пользователь указал в квадратных скобках в качестве аргумента (в большинстве случаев, это целочисленное значение). Однако это не является обязательным требованием и, на самом деле, вы можете определить функцию перегрузки так, чтобы ваш перегруженный оператор `[]` принимал значения любого типа, которого вы только пожелаете (double, string и т.д.). Например:

```
#include <iostream>
#include <string>

class Something
{
private:

public:
    void operator[] (std::string index);
};

// Нет смысла перегружать оператор [] только для вывода чего-либо,
но это самый простой способ показать, что параметр функции может быть не
только целочисленным значением
void Something::operator[] (std::string index)
{
    std::cout << index;
}

int main()
{
    Something something;
    something["Hello, world!"];

    return 0;
}
```

Результат выполнения программы:

```
Hello, world!
```

Заключение

Перегрузка оператора индексации обычно используется для обеспечения прямого доступа к элементам массива, который находится внутри класса (в качестве переменной-члена). Поскольку строки часто используются в реализации массивов символов, то оператор `[]` часто перегружают в классах со строками, чтобы иметь доступ к каждому символу строки отдельно.

Тест

Задание №1

Контейнер map - это класс, в котором все элементы хранятся в виде пары ключ-значение. Ключ должен быть уникальным и использоваться для доступа к связанной паре. В этом задании вам нужно будет написать программу, которая позволит присваивать оценки ученикам, указывая только имя ученика. Для этого используйте контейнер map: имя ученика - ключ, оценка (тип char) - значение.

a) Сначала напишите структуру StudentGrade с двумя элементами: имя студента (std::string) и оценка (char).

b) Добавьте класс GradeMap, который содержит std::vector типа StudentGrade с именем `m_map`. Добавьте пустой конструктор по умолчанию.

c) Реализуйте перегрузку оператора `[]` для этого класса. Функция перегрузки должна принимать параметр std::string (имя ученика) и возвращать ссылку на его оценку. В функции перегрузки сначала выполните поиск указанного имени ученика в векторе (используйте цикл foreach). Если ученик нашелся, то возвращайте ссылку на его оценку, и всё - готово!

В противном случае, используйте функцию std::vector::push_back() для добавления StudentGrade нового ученика. Когда вы это сделаете, std::vector добавит себе копию нового StudentGrade (при необходимости изменив размер). Наконец, вам нужно будет возвратить ссылку на оценку студента, которого вы только что добавили в std::vector — для этого используйте std::vector::back().

Следующая программа должна скомпилироваться без ошибок:

```
#include <iostream>

int main()
{
    GradeMap grades;
    grades["John"] = 'A';
    grades["Martin"] = 'B';
    std::cout << "John has a grade of " << grades["John"] << '\n';
    std::cout << "Martin has a grade of " << grades["Martin"] << '\n';

    return 0;
}
```

Задание №2

Класс GradeMap и программа, которую мы написали, неэффективна по нескольким причинам. Опишите один способ улучшения класса GradeMap.

Задание №3

Почему следующая программа не работает должным образом?

```
1. #include <iostream>
2.
3. int main()
4. {
5.     GradeMap grades;
6.
7.     char& gradeJohn = grades["John"]; // выполняется push_back
8.     gradeJohn = 'A';
9.
10.    char& gradeMartin = grades["Martin"]; // выполняется push_back
11.    gradeMartin = 'B';
12.
13.    std::cout << "John has a grade of " << gradeJohn << '\n';
14.    std::cout << "Martin has a grade of " << gradeMartin << '\n';
15.
16.    return 0;
17. }
```

Урок №147. Перегрузка оператора ()

Все операторы, перегрузку которых мы рассматривали до сих пор, позволяли нам самостоятельно определять тип параметров в функции перегрузки оператора, но не их количество. Например, оператор `==` всегда принимает два параметра, тогда как оператор `!` всегда принимает один параметр. **Оператор** `()` является особенно интересным, поскольку позволяет изменять как тип параметров, так и их количество.

Но следует помнить о двух вещах:

- Во-первых, перегрузка круглых скобок должна осуществляться через метод класса.
- Во-вторых, в не объектно-ориентированном C++ оператор `()` является оператором вызова функции. В случае с классами перегрузка круглых скобок выполняется в методе `operator()(){}` (в объявлении функции перегрузки находятся две пары круглых скобок).

Рассмотрим следующий класс:

```
1.  class Matrix
2.  {
3.  private:
4.      double data[5][5];
5.  public:
6.      Matrix()
7.      {
8.          // Присваиваем всем элементам массива значение 0.0
9.          for (int row=0; row < 5; ++row)
10.             for (int col=0; col < 5; ++col)
11.                 data[row][col] = 0.0;
12.     }
13. };
```

Матрицы являются ключевой концепцией в линейной алгебре и часто используются в геометрическом моделировании и в 3D-графике. Всё, что вам нужно знать сейчас - это то, что класс Matrix является двумерным массивом (5×5 типа double).

На уроке о перегрузке оператора индексации мы использовали оператор `[]` для прямого доступа к элементам закрытого одномерного массива. Здесь же нам нужен доступ к элементам двумерного массива. Поскольку оператор `[]` ограничен лишь одним параметром, то его функциональности недостаточно для доступа к двумерному массиву.

Однако, поскольку оператор `()` может принимать разное количество параметров, мы можем объявить версию operator(), которая будет принимать два целочисленных параметра (два индекса), и использовать эти индексы для доступа к элементам нашего двумерного массива. Например:

```
#include <iostream>
#include <cassert> // для assert()

class Matrix
{
private:
    double data[5][5];
public:
    Matrix()
    {
        // Присваиваем всем элементам массива значение 0.0
        for (int row=0; row < 5; ++row)
            for (int col=0; col < 5; ++col)
                data[row][col] = 0.0;
    }

    double& operator()(int row, int col);
    const double& operator()(int row, int col) const; // для константных объектов
};

double& Matrix::operator()(int row, int col)
{
    assert(col >= 0 && col < 5);
    assert(row >= 0 && row < 5);

    return data[row][col];
}

const double& Matrix::operator()(int row, int col) const
{
    assert(col >= 0 && col < 5);
    assert(row >= 0 && row < 5);

    return data[row][col];
}

int main()
{
    Matrix matrix;
    matrix(2, 3) = 3.6;
    std::cout << matrix(2, 3);

    return 0;
}
```

Результат выполнения программы:

```
3.6
```

Выполним перегрузку оператора `()` еще раз, но уже без использования каких-либо параметров:

```
#include <iostream>
#include <cassert> // для assert()

class Matrix
{
private:
    double data[5][5];
public:
    Matrix()
    {
        // Присваиваем всем элементам массива значение 0.0
        for (int row=0; row < 5; ++row)
            for (int col=0; col < 5; ++col)
                data[row][col] = 0.0;
    }

    double& operator()(int row, int col);
    const double& operator()(int row, int col) const;
    void operator()();
};

double& Matrix::operator()(int row, int col)
{
    assert(col >= 0 && col < 5);
    assert(row >= 0 && row < 5);

    return data[row][col];
}

const double& Matrix::operator()(int row, int col) const
{
    assert(col >= 0 && col < 5);
    assert(row >= 0 && row < 5);

    return data[row][col];
}

void Matrix::operator()()
{
    // Сбрасываем значения всех элементов массива на 0.0
    for (int row=0; row < 5; ++row)
        for (int col=0; col < 5; ++col)
            data[row][col] = 0.0;
}

int main()
{
    Matrix matrix;
    matrix(2, 3) = 3.6;
    matrix(); // выполняем сброс
    std::cout << matrix(2, 3);

    return 0;
}
```

Результат выполнения программы:

```
0
```

Поскольку оператор `()` является очень гибким, то может возникнуть соблазн использовать его для самых разных целей. Однако это настоятельно не рекомендуется делать, поскольку оператор `()` не является очень информативным и зачастую может быть не понятно, что он делает. В примере, приведенном выше, функцию очистки массива лучше было бы записать в виде метода clear() или erase(), поскольку `matrix.erase()` выглядит намного информативнее, нежели `matrix()`.

Функторы в C++

Перегрузка оператора `()` используется в реализации **функторов** (или ***"функциональных объектов"***) - классы, которые работают как функции. Преимущество функтора над обычной функцией заключается в том, что функторы могут хранить данные в переменных-членах (поскольку они сами являются классами). Вот пример использования простого функтора:

```
#include <iostream>

class Accumulator
{
private:
    int m_counter = 0;

public:
    Accumulator()
    {
    }

    int operator() (int i) { return (m_counter += i); }
};

int main()
{
    Accumulator accum;
    std::cout << accum(30) << std::endl; // выведется 30
    std::cout << accum(40) << std::endl; // выведется 70

    return 0;
}
```

Обратите внимание, использование класса Accumulator выглядит так же, как и вызов обычной функции, но наш объект класса Accumulator может хранить значение, которое увеличивается.

Вы можете спросить: "Зачем использовать класс, если всё можно реализовать и через обычную функцию со статической локальной переменной?". Можно сделать и через static, но, поскольку функции представлены только одним глобальным экземпляром (т.е. нельзя создать несколько объектов функции), использовать эту функцию мы можем только для выполнения чего-то одного за раз. С помощью функторов мы можем создать любое количество отдельных функциональных объектов, которые нам нужны, и использовать их одновременно.

Заключение

Перегрузка оператора `()` с двумя параметрами используется для получения доступа к двумерным массивам или для возврата подмножеств одномерного массива (два параметра будут конкретизировать условия отбора элементов подмножества). Всё остальное лучше реализовать через отдельные методы с более информативными названиями, нежели через перегрузку оператора `()`.

Перегрузка оператора `()` также часто используется при создании функторов. Хотя функторы, которые мы использовали выше, являются довольно простыми и понятными, но обычно они используются в более продвинутых/сложных темах программирования и заслуживают отдельного урока.

Тест

Напишите класс, переменной-членом которого является строка. Перегрузите оператор `()` для возврата подстроки, которая начинается с индекса, указанного в значении первого параметра. Второй параметр должен указывать требуемую длину подстроки.

> ***Подсказки:***
>
> - Вы можете использовать индекс массива `[]` для доступа к отдельным символам строки.
> - Вы можете использовать оператор `+=` для добавления чего-либо к строке.

Следующий фрагмент кода:

```
1. int main()
2. {
3.     Mystring string("Hello, world!");
4.     std::cout << string(7, 6); // начинаем с 7 символа (индекса) и возвращаем следующие 6 символов
5.
6.     return 0;
7. }
```

Должен выдавать следующий результат:

```
world!
```

Урок №148. Перегрузка операций преобразования типов данных

Как мы уже знаем из урока о неявном преобразовании типов данных, C++ позволяет конвертировать значения из одного типа данных в другой. Например, преобразуем значение типа int в значение типа double:

```
1. int a = 7;
2. double b = a; // значение типа int неявно конвертируется в значение типа double
```

Язык C++ по умолчанию знает, как выполнять преобразования встроенных типов данных. Однако он не знает, как выполнять конвертацию с пользовательскими типами данных (например, с классами). Именно здесь вступает в игру перегрузка операций преобразования типов данных. Рассмотрим следующий класс:

```
1. class Dollars
2. {
3. private:
4.     int m_dollars;
5. public:
6.     Dollars(int dollars=0)
7.     {
8.         m_dollars = dollars;
9.     }
10.
11.    int getDollars() { return m_dollars; }
12.    void setDollars(int dollars) { m_dollars = dollars; }
13. };
```

Класс Dollars содержит некое количество долларов в виде целого числа (переменная-член `m_dollars`) и предоставляет функции доступа для получения и установления значения `m_dollars`. В нем также есть конструктор для конвертации значений типа int в тип Dollars (при создании объекта пользователь передает в качестве аргумента значение типа int, которое затем преобразуется в значение типа Dollars).

Если мы можем конвертировать int в Dollars, то логично было бы, если бы мы могли конвертировать и Dollars обратно в int, не так ли? Иногда это может быть полезным.

В следующем примере мы используем метод getDollars() для конвертации значения типа Dollars в тип int для его последующего вывода через функцию printInt():

```
1. void printInt(int value)
2. {
3.     std::cout << value;
4. }
5.
6. int main()
```

```
{
    Dollars dollars(9);
    printInt(dollars.getDollars()); // выведется 9

    return 0;
}
```

Согласитесь, вызывать каждый раз метод getDollars() не очень удобно. Было бы проще **перегрузить операцию преобразования значений типа Dollars в тип int. Делается это следующим образом**:

```
class Dollars
{
private:
    int m_dollars;
public:
    Dollars(int dollars=0)
    {
        m_dollars = dollars;
    }

    // Перегрузка операции преобразования значений типа Dollars в значения типа int
    operator int() { return m_dollars; }

    int getDollars() { return m_dollars; }
    void setDollars(int dollars) { m_dollars = dollars; }
};
```

Здесь есть две вещи, на которые следует обратить внимание:

- В качестве функции перегрузки используется метод `operator int()`. Обратите внимание, между словом operator и типом, в который мы хотим выполнить конвертацию (в данном случае, тип int), находится пробел.
- Функция перегрузки не имеет типа возврата. Язык C++ предполагает, что вы будете возвращать корректный тип.

Теперь функция printInt() вызывается проще:

```
int main()
{
    Dollars dollars(9);
    printInt(dollars); // выведется 9

    return 0;
}
```

Сначала компилятор видит, что функция printInt() должна принимать целочисленный параметр (из определения printInt()). Затем он видит, что переменная `dollars` не является типа int. Он смотрит, предоставили ли мы способ конвертации значения типа Dollars в тип int. Так как это у нас есть, то вызывается

operator int(), который возвращает значение типа int, и это значение передается в printInt().

Теперь мы можем явно конвертировать объект класса Dollars в тип int:

```
Dollars dollars(9);
int d = static_cast<int>(dollars);
```

Вы можете перегружать операции преобразования любых типов данных, включая ваши собственные (пользовательские) типы данных!

Например, вот класс Cents, в котором осуществлена перегрузка операции преобразования значения типа Cents в значение типа Dollars:

```
class Cents
{
private:
    int m_cents;
public:
    Cents(int cents=0)
    {
        m_cents = cents;
    }

     // Выполняем конвертацию Cents в Dollars
     operator Dollars() { return Dollars(m_cents / 100); }
};
```

Таким образом мы можем напрямую конвертировать центы в доллары:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;
public:
    Dollars(int dollars = 0)
    {
        m_dollars = dollars;
    }

    // Перегрузка операции преобразования значения типа Dollars в значение типа int
    operator int() { return m_dollars; }

    int getDollars() { return m_dollars; }
    void setDollars(int dollars) { m_dollars = dollars; }
};

class Cents
{
private:
    int m_cents;
public:
    Cents(int cents=0)
```

```
    {
        m_cents = cents;
    }

     // Выполняем конвертацию Cents в Dollars
     operator Dollars() { return Dollars(m_cents / 100); }
};

void printDollars(Dollars dollars)
{
    std::cout << dollars; // dollars неявно конвертируется в int здесь
}

int main()
{
    Cents cents(700);
    printDollars(cents); // cents неявно конвертируется в Dollars здесь

    return 0;
}
```

Результат выполнения программы:

```
7
```

Всё логично: 700 центов = 7 долларов!

Урок №149. Конструктор копирования

Рассмотрим примеры всех вышеприведенных инициализаций на практике, используя следующий класс Drob:

```
#include <cassert>
#include <iostream>

class Drob
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Конструктор по умолчанию
    Drob(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
};

std::ostream& operator<<(std::ostream& out, const Drob &d1)
{
    out << d1.m_numerator << "/" << d1.m_denominator;
    return out;
}
```

Мы можем выполнить прямую инициализацию:

```
int a(7); // прямая инициализация целочисленной переменной
Drob sixSeven(6, 7); // прямая инициализация объекта класса Drob, вызывается конструктор Drob(int, int)
```

В C++11 мы можем выполнить uniform-инициализацию:

```
int a { 7 }; // uniform-инициализация целочисленной переменной
Drob sixSeven {6, 7}; // uniform-инициализация объекта класса Drob, вызывается конструктор Drob(int, int)
```

И, наконец, мы можем выполнить копирующую инициализацию:

```
int a = 7; // копирующая инициализация целочисленной переменной
Drob eight = Drob(8); // копирующая инициализация объекта класса Drob, вызывается Drob(8, 1)
Drob nine = 9; // копирующая инициализация объекта класса Drob. Компилятор будет искать способ конвертации 9 в объект класса Drob, что приведет к вызову конструктора Drob(9, 1)
```

С прямой инициализацией и uniform-инициализацией создаваемый объект непосредственно инициализируется. Однако с копирующей инициализацией дела

обстоят несколько сложнее. Мы рассмотрим это детально на следующем уроке. Но перед этим нам еще нужно кое в чём разобраться.

Рассмотрим следующую программу:

```
#include <cassert>
#include <iostream>

class Drob
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Конструктор по умолчанию
    Drob(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
};

std::ostream& operator<<(std::ostream& out, const Drob &d1)
{
    out << d1.m_numerator << "/" << d1.m_denominator;
    return out;
}

int main()
{
    Drob sixSeven(6, 7); // прямая инициализация объекта класса Drob, вызывается конструктор Drob(int, int)
    Drob dCopy(sixSeven); // прямая инициализация - какой конструктор вызывается здесь?
    std::cout << dCopy << '\n';
}
```

Результат выполнения программы:

```
6/7
```

Рассмотрим подробнее, как работает эта программа.

С объектом `sixSeven` выполняется обычная прямая инициализация, которая приводит к вызову конструктора `Drob(int, int)`. Здесь нет никаких сюрпризов. А вот инициализация объекта `dCopy` также является прямой инициализацией, но какой конструктор вызывается здесь? Ответ: конструктор копирования.

Конструктор копирования - это особый тип конструктора, который используется для создания нового объекта через копирование существующего объекта. И, как в

случае с конструктором по умолчанию, если вы не предоставите конструктор копирования для своих классов самостоятельно, то язык C++ создаст public-конструктор копирования автоматически. Поскольку компилятор мало знает о вашем классе, то по умолчанию созданный конструктор копирования будет использовать почленную инициализацию.

Почленная инициализация означает, что каждый член объекта-копии инициализируется непосредственно из члена объекта-оригинала. Т.е. в примере, приведенном выше, `dCopy.m_numerator` будет иметь значение `sixSeven.m_numerator` (`6`), а `dCopy.m_denominator` будет равен `sixSeven.m_ denominator` (`7`).

Так же, как мы можем явно определить конструктор по умолчанию, так же мы можем явно определить и конструктор копирования. Конструктор копирования выглядит следующим образом:

```
1. #include <iostream>
2. #include <cassert>
3.
4. class Drob
5. {
6. private:
7.     int m_numerator;
8.     int m_denominator;
9.
10. public:
11.     // Конструктор по умолчанию
12.     Drob(int numerator=0, int denominator=1) :
13.         m_numerator(numerator), m_denominator(denominator)
14.     {
15.         assert(denominator != 0);
16.     }
17.
18.     // Конструктор копирования
19.     Drob(const Drob &drob) :
20.         m_numerator(drob.m_numerator), m_denominator(drob.m_denominator)
21.         // Примечание: Мы имеем прямой доступ к членам объекта drob,
    поскольку мы сейчас находимся внутри класса Drob
22.     {
23.         // Нет необходимости выполнять проверку denominator здесь, так как
    эта проверка уже осуществляется в конструкторе класса Drob
24.         std::cout << "Copy constructor worked here!\n"; // просто, чтобы
    показать, что это работает
25.     }
26.
27.     friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
28. };
29.
30. std::ostream& operator<<(std::ostream& out, const Drob &d1)
31. {
32.     out << d1.m_numerator << "/" << d1.m_denominator;
33.     return out;
34. }
35.
```

```
36. int main()
37. {
38.     Drob sixSeven(6, 7); // прямая инициализация объекта класса Drob, вызывается конструктор Drob(int, int)
39.     Drob dCopy(sixSeven); // прямая инициализация, вызывается конструктор копирования класса Drob
40.     std::cout << dCopy << '\n';
41. }
```

Результат выполнения программы:

```
Copy constructor worked here!
6/7
```

Конструктор копирования в вышеприведенном примере использует почленную инициализацию и функционально эквивалентен конструктору по умолчанию, за исключением того, что мы добавили стейтмент вывода, в котором указали текст (сработал конструктор копирования).

Предотвращение создания копий объектов

Мы можем предотвратить создание копий объектов наших классов, сделав конструктор копирования закрытым:

```
1. #include <iostream>
2. #include <cassert>
3.
4. class Drob
5. {
6. private:
7.     int m_numerator;
8.     int m_denominator;
9.
10.    // Конструктор копирования (закрытый)
11.    Drob(const Drob &drob) :
12.        m_numerator(drob.m_numerator), m_denominator(drob.m_denominator)
13.    {
14.        // Нет необходимости выполнять проверку denominator здесь, так как эта проверка уже осуществляется в конструкторе класса Drob
15.        std::cout << "Copy constructor worked here!\n"; // просто, чтобы показать, что это работает
16.    }
17.
18. public:
19.    // Конструктор по умолчанию
20.    Drob(int numerator=0, int denominator=1) :
21.        m_numerator(numerator), m_denominator(denominator)
22.    {
23.        assert(denominator != 0);
24.    }
25.
26.    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
27. };
28.
29. std::ostream& operator<<(std::ostream& out, const Drob &d1)
```

```
{
    out << d1.m_numerator << "/" << d1.m_denominator;
    return out;
}

int main()
{
    Drob sixSeven(6, 7); // прямая инициализация объекта класса Drob, вызывается конструктор Drob(int, int)
    Drob dCopy(sixSeven); // конструктор копирования является закрытым, поэтому эта строка вызовет ошибку компиляции
    std::cout << dCopy << '\n';
}
```

Здесь мы получим ошибку компиляции, так как `dCopy` должен использовать конструктор копирования, но он не видит его, поскольку конструктор копирования является закрытым.

Конструктор копирования может быть проигнорирован

Рассмотрим следующий код:

```
#include <cassert>
#include <iostream>

class Drob
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Конструктор по умолчанию
    Drob(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

        // Конструктор копирования
    Drob(const Drob &drob) :
        m_numerator(drob.m_numerator), m_denominator(drob.m_denominator)
    {
        // Нет необходимости выполнять проверку denominator здесь, так как эта проверка уже осуществляется в конструкторе класса Drob
        std::cout << "Copy constructor worked here!\n"; // просто, чтобы показать, что это работает
    }

    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
};

std::ostream& operator<<(std::ostream& out, const Drob &d1)
{
    out << d1.m_numerator << "/" << d1.m_denominator;
    return out;
}

```

```
35. int main()
36. {
37.     Drob sixSeven(Drob(6, 7));
38.     std::cout << sixSeven;
39.     return 0;
40. }
```

Сначала инициализируется анонимный объект Drob, который приводит к вызову конструктора `Drob(int, int)`. Затем этот анонимный объект используется для инициализации объекта `sixSeven` класса Drob. Поскольку анонимный объект является объектом класса Drob, как и `sixSeven`, то здесь должен вызываться конструктор копирования, верно?

Запустите эту программу самостоятельно. Ожидаемый результат:

```
Copy constructor worked here!
6/7
```

Реальный результат:

```
6/7
```

Почему наш конструктор копирования не сработал?

Дело в том, что инициализация анонимного объекта, а затем использование этого объекта для прямой инициализации уже не анонимного объекта выполняется в два этапа (первый этап - это создание анонимного объекта, второй этап - это вызов конструктора копирования). Однако, конечный результат по сути идентичен простому выполнению прямой инициализации, которая занимает всего лишь один шаг.

По этой причине в таких случаях компилятору разрешается отказаться от вызова конструктора копирования и просто выполнить прямую инициализацию. Этот процесс называется **элизией**.

Поэтому, даже если вы напишите:

```
1. Drob sixSeven(Drob(6, 7));
```

Компилятор может изменить это на:

```
1. Drob sixSeven(6, 7);
```

В последнем случае с прямой инициализацией потребуется вызов только одного конструктора (`Drob(int, int)`). Обратите внимание, в случаях, когда используется элизия, любые стейтменты в теле конструктора копирования не

выполняются, даже если они имеют побочные эффекты (например, выводят что-либо на экран)!

Наконец, если вы делаете свой конструктор копирования закрытым, то любая инициализация, использующая этот закрытый конструктор копирования, приведет к ошибкам компиляции, даже если конструктор копирования будет проигнорирован!

Урок №150. Копирующая инициализация

Рассмотрим следующую строку кода:

```
1. int a = 7;
```

Здесь используется **копирующая инициализация** для инициализации целочисленной переменной `a` значением `7`. С обычными переменными всё просто. Однако с классами дела обстоят несколько сложнее, поскольку в их инициализации используются конструкторы. На этом уроке мы рассмотрим использование копирующей инициализации с классами.

Использование копирующей инициализации с классами

Рассмотрим следующую программу:

```
1. #include <iostream>
2. #include <cassert>
3.
4. class Drob
5. {
6. private:
7.     int m_numerator;
8.     int m_denominator;
9.
10. public:
11.     // Конструктор по умолчанию
12.     Drob(int numerator=0, int denominator=1) :
13.         m_numerator(numerator), m_denominator(denominator)
14.     {
15.         assert(denominator != 0);
16.     }
17.
18.     friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
19. };
20.
21. std::ostream& operator<<(std::ostream& out, const Drob &d1)
22. {
23.     out << d1.m_numerator << "/" << d1.m_denominator;
24.     return out;
25. }
26.
27. int main()
28. {
29.     Drob seven = Drob(7);
30.     std::cout << seven;
31.     return 0;
32. }
```

Результат выполнения программы:

```
7/1
```

Форма копирующей инициализации в языке C++ в вышеприведенном примере обрабатывается точно так же, как и следующая:

```
1. Drob seven(Drob(7));
```

А, как мы уже знаем из предыдущего урока, это может привести к вызову как `Drob(int, int)`, так и конструктора копирования Drob (который может быть проигнорирован). Однако, поскольку гарантии на 100% игнорирования конструктора копирования не предоставляется, то лучше избегать использования копирующей инициализации при работе с классами и вместо нее использовать прямую инициализацию или uniform-инициализацию, так как в случае с использованием конструктора копирования у вас может получиться следующий результат:

```
7
```

Вместо необходимого:

```
7/1
```

Так как в конструкторе копирования (который язык C++ предоставит автоматически) значения по умолчанию для `m_denominator` не будет.

Правило: Избегайте использования копирующей инициализации при работе с классами, вместо нее используйте uniform-инициализацию.

Другие применения копирующей инициализации

Когда вы передаете или возвращаете объект класса по значению, то в этом процессе используется копирующая инициализация. Рассмотрим следующую программу:

```
1. #include <iostream>
2. #include <cassert>
3.
4. class Drob
5. {
6. private:
7.     int m_numerator;
8.     int m_denominator;
9.
10.public:
11.    // Конструктор по умолчанию
12.    Drob(int numerator=0, int denominator=1) :
13.        m_numerator(numerator), m_denominator(denominator)
14.    {
15.        assert(denominator != 0);
16.    }
```

```
17.
18.         // Конструктор копирования
19.     Drob(const Drob ©) :
20.         m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
21.     {
22.         // Нет необходимости выполнять проверку denominator здесь, так как
    она осуществляется в конструкторе по умолчанию
23.         std::cout << "Copy constructor worked here!\n"; // просто чтобы
    показать, что это работает
24.     }
25.
26.     friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
27.         int getNumerator() { return m_numerator; }
28.         void setNumerator(int numerator) { m_numerator = numerator; }
29. };
30.
31. std::ostream& operator<<(std::ostream& out, const Drob &d1)
32. {
33.     out << d1.m_numerator << "/" << d1.m_denominator;
34.     return out;
35. }
36.
37. Drob makeNegative(Drob d) // правильно было бы здесь использовать константную
    ссылку
38. {
39.     d.setNumerator(-d.getNumerator());
40.     return d;
41. }
42.
43. int main()
44. {
45.     Drob sixSeven(6, 7);
46.     std::cout << makeNegative(sixSeven);
47.
48.     return 0;
49. }
```

Здесь функция makeNegative() принимает объект класса Drob по значению и возвращает его так же по значению. Результат выполнения программы:

```
Copy constructor worked here!
Copy constructor worked here!
-6/7
```

Первый вызов конструктора копирования выполнится при передаче `sixSeven` в качестве аргумента в параметр `d` функции makeNegative(). Второй вызов выполнится при возврате объекта из функции makeNegative() обратно в функцию main(). Таким образом, объект `sixSeven` копируется дважды.

В примере, приведенном выше, компилятор не может проигнорировать использование конструктора копирования как в передаче аргумента по значению, так и в его возврате. Однако в некоторых случаях, если аргумент или возвращаемое значение соответствуют определенным критериям, компилятор может проигнорировать использование конструктора копирования.

Например:

```
#include <iostream>

class Something
{
};

Something boo()
{
  Something x;
  return x;
}

int main()
{
  Something x = boo();
  return 0;
}
```

В этом случае компилятор, скорее всего, проигнорирует использование конструктора копирования, хоть объект `x` и возвращается по значению.

Урок №151. Конструкторы преобразования, ключевые слова explicit и delete

По умолчанию язык C++ обрабатывает любой конструктор, как оператор неявного преобразования. Рассмотрим следующую программу:

```
#include <iostream>
#include <cassert>

class Drob
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Конструктор по умолчанию
    Drob(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    // Конструктор копирования
    Drob(const Drob &) :
        m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
    {
        // Нет необходимости выполнять проверку denominator здесь, так как эта проверка уже осуществлена в конструкторе по умолчанию
        std::cout << "Copy constructor worked here!\n"; // просто, чтобы показать, что это работает
    }

    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
    int getNumerator() { return m_numerator; }
    void setNumerator(int numerator) { m_numerator = numerator; }
};

std::ostream& operator<<(std::ostream& out, const Drob &d1)
{
    out << d1.m_numerator << "/" << d1.m_denominator;
    return out;
}

Drob makeNegative(Drob d)
{
    d.setNumerator(-d.getNumerator());
    return d;
}

int main()
{
    std::cout << makeNegative(7); // передаем целочисленное значение

    return 0;
}
```

Хотя функция makeNegative() ожидает объект класса Drob, мы передаем ей целочисленный литерал `7`. Поскольку у класса Drob есть конструктор, который может принимать одно целочисленное значение (конструктор по умолчанию), то компилятор выполнит неявную конвертацию литерала `7` в объект класса Drob. Это делается путем выполнения копирующей инициализации параметра `d` функции makeNegative() с помощью конструктора `Drob(int, int)`.

Результат выполнения программы:

```
Copy constructor worked here!
-7/1
```

Неявное преобразование работает для всех видов инициализации (прямой, uniform и копирующей).

Конструкторы, которые используются в неявных преобразованиях, называются **конструкторами преобразования** (или ***"конструкторами конвертации"***). До C++11 конструкторами преобразования могли быть конструкторы только с одним параметром. Однако в C++11 это ограничение было снято (наряду с добавлением uniform-инициализации), и конструкторы, имеющие несколько параметров, также уже могут быть конструкторами преобразования.

Ключевое слово explicit

Иногда выполнение неявных преобразований может иметь смысл, а иногда может быть крайне нежелательным и генерировать неожиданные результаты:

```
1. #include <iostream>
2. #include <string>
3.
4. class SomeString
5. {
6. private:
7.     std::string m_string;
8. public:
9.     SomeString(int a) // выделяем строку размером a
10.    {
11.        m_string.resize(a);
12.    }
13.
14.    SomeString(const char *string) // выделяем строку для хранения значения
   типа string
15.    {
16.        m_string = string;
17.    }
18.
19.    friend std::ostream& operator<<(std::ostream& out, const SomeString &s);
20.
21. };
```

```
22.
23. std::ostream& operator<<(std::ostream& out, const SomeString &s)
24. {
25.     out << s.m_string;
26.     return out;
27. }
28.
29. int main()
30. {
31.     SomeString mystring = 'a'; // выполняется копирующая инициализация
32.     std::cout << mystring;
33.     return 0;
34. }
```

В примере, приведенном выше, мы пытаемся инициализировать строку одним символом типа char. Поскольку переменные типа char являются частью семейства целочисленных типов, то компилятор будет использовать конструктор преобразования `SomeString(int)` для неявного преобразования символа типа char в тип SomeString. В результате переменная типа char будет конвертирована в тип int. А это не совсем то, что ожидается.

Один из способов решения этой проблемы — сделать конструктор явным, используя **ключевое слово explicit** (которое пишется перед именем конструктора). Явные конструкторы (с ключевым словом explicit) не используются для неявных конвертаций:

```
1.  #include <iostream>
2.  #include <string>
3.
4.  class SomeString
5.  {
6.  private:
7.      std::string m_string;
8.  public:
9.      // Ключевое слово explicit делает этот конструктор закрытым для
    выполнения любых неявных преобразований
10.     explicit SomeString(int a) // выделяем строку размером a
11.     {
12.         m_string.resize(a);
13.     }
14.
15.     SomeString(const char *string) // выделяем строку для хранения значения
    типа string
16.     {
17.         m_string = string;
18.     }
19.
20.     friend std::ostream& operator<<(std::ostream& out, const SomeString &s);
21.
22. };
23.
24. std::ostream& operator<<(std::ostream& out, const SomeString &s)
25. {
26.     out << s.m_string;
27.     return out;
28. }
```

```
29.
30. int main()
31. {
32.     SomeString mystring = 'a'; // ошибка компиляции, поскольку SomeString(int) теперь является explicit и, соответственно, недоступен, а другого подходящего конструктора для преобразования компилятор не видит
33.     std::cout << mystring;
34.     return 0;
35. }
```

Вышеприведенная программа не скомпилируется, так как `SomeString(int)` мы сделали явным, а другого конструктора преобразования, который выполнил бы неявную конвертацию `'a'` в SomeString, компилятор просто не нашел.

Однако использование явного конструктора только предотвращает выполнение неявных преобразований. Явные конвертации (через операторы явного преобразования) по-прежнему разрешены:

```
1. std::cout << static_cast<SomeString>(7); // разрешено: явное преобразование 7 в SomeString через оператор static_cast
```

При прямой- или uniform-инициализации неявная конвертация также будет выполняться:

```
1. SomeString str('a'); // разрешено
```

Правило: Для предотвращения возникновения ошибок с неявными конвертациями делайте ваши конструкторы явными, используя ключевое слово explicit.

Ключевое слово delete

Еще одним способом запретить конвертацию `'a'` в SomeString (неявным или явным способом) является добавление закрытого конструктора `SomeString(char)`:

```
1. #include <iostream>
2. #include <string>
3.
4. class SomeString
5. {
6. private:
7.     std::string m_string;
8.
9.     SomeString(char) // объекты типа SomeString(char) не могут быть созданы вне класса
10.    {
11.    }
12. public:
13.    // Ключевое слово explicit делает этот конструктор закрытым для выполнения любых неявных конвертаций
14.    explicit SomeString(int a) // выделяем строку размером a
15.    {
```

```
16.         m_string.resize(a);
17.     }
18.
19.     SomeString(const char *string) // выделяем строку для хранения значения
    типа string
20.     {
21.         m_string = string;
22.     }
23.
24.     friend std::ostream& operator<<(std::ostream& out, const SomeString &s);
25.
26. };
27.
28. std::ostream& operator<<(std::ostream& out, const SomeString &s)
29. {
30.     out << s.m_string;
31.     return out;
32. }
33.
34. int main()
35. {
36.     SomeString mystring('a'); // ошибка компиляции, поскольку SomeString(char)
    является private
37.     std::cout << mystring;
38.     return 0;
39. }
```

Тем не менее, этот конструктор все еще может использоваться внутри класса (private закрывает доступ к данным только для объектов вне тела класса).

Лучшее решение — **использовать ключевое слово delete** (добавленное в C++11) для удаления этого конструктора:

```
1.  #include <iostream>
2.  #include <string>
3.
4.  class SomeString
5.  {
6.  private:
7.      std::string m_string;
8.
9.  public:
10.     SomeString(char) = delete; // любое использование этого конструктора
    приведет к ошибке
11.
12.     // Ключевое слово explicit делает этот конструктор закрытым для выполнения
    любых неявных конвертаций
13.     explicit SomeString(int a) // выделяем строку размером a
14.     {
15.         m_string.resize(a);
16.     }
17.
18.     SomeString(const char *string) // выделяем строку для хранения значения
    типа string
19.     {
20.         m_string = string;
21.     }
22.
23.     friend std::ostream& operator<<(std::ostream& out, const SomeString &s);
```

```
24.
25. };
26.
27. std::ostream& operator<<(std::ostream& out, const SomeString &s)
28. {
29.     out << s.m_string;
30.     return out;
31. }
32.
33. int main()
34. {
35.     SomeString mystring('a'); // ошибка компиляции, поскольку SomeString(char)
    удален
36.     std::cout << mystring;
37.     return 0;
38. }
```

После удаления функции, любое её использование вызовет ошибку компиляции.

Обратите внимание, конструктор копирования и перегруженные операторы также могут быть удалены с помощью delete для предотвращения их использования.

Урок №152. Перегрузка оператора присваивания

Оператор присваивания (=) используется для копирования значений из одного объекта в другой (уже существующий) объект.

Присваивание vs. Конструктор копирования

Конструктор копирования и оператор присваивания выполняют почти идентичную работу: оба копируют значения из одного объекта в значения другого объекта. Однако конструктор копирования используется при инициализации новых объектов, тогда как оператор присваивания заменяет содержимое уже существующих объектов. Всё просто:

- Если новый объект создан перед выполнением операции копирования, то используется конструктор копирования (передача или возврат объектов выполняются по значению).
- Если создания нового объекта не было, а работа ведется с уже существующим объектом, то используется оператор присваивания.

Перегрузка оператора присваивания

Перегрузка оператора присваивания (=) довольно-таки проста и выполняется через метод класса, но есть один нюанс:

```
#include <iostream>
#include <cassert>

class Drob
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Конструктор по умолчанию
    Drob(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    // Конструктор копирования
    Drob(const Drob &) :
        m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
    {
        // Нет необходимости выполнять проверку denominator здесь, так как эта проверка уже осуществлена в конструкторе по умолчанию
        std::cout << "Copy constructor worked here!\n"; // просто, чтобы показать, что это работает
```

```
    }

        // Перегрузка оператора присваивания
        Drob& operator= (const Drob &drob)
        {
            // Выполняем копирование значений
            m_numerator = drob.m_numerator;
            m_denominator = drob.m_denominator;

            // Возвращаем текущий объект, чтобы иметь возможность связать в цепочку выполнение нескольких операций присваивания
            return *this;
        }

    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);

};

std::ostream& operator<<(std::ostream& out, const Drob &d1)
{
    out << d1.m_numerator << "/" << d1.m_denominator;
    return out;
}

int main()
{
    Drob sixSeven(6, 7);
    Drob d;
    d = sixSeven; // вызывается перегруженный оператор присваивания
    std::cout << d;

    return 0;
}
```

Результат выполнения программы:

```
6/7
```

До этого момента всё ок. Функция перегрузки operator=() возвращает скрытый указатель *this, и мы даже можем связать выполнение нескольких операций присваивания вместе:

```
int main()
{
    Drob d1(6,7);
    Drob d2(8,3);
    Drob d3(10,4);

    d1 = d2 = d3; // цепочка операций присваивания

    return 0;
}
```

Самоприсваивание

Здесь уже становится интереснее. Самоприсваивание — это тот нюанс, о котором упоминалось выше. Язык C++ позволяет выполнять самоприсваивание:

```
int main()
{
    Drob d1(6,7);
    d1 = d1; // самоприсваивание

    return 0;
}
```

В примере, приведенном выше, самоприсваивание не приведет к изменению состояния чего-либо и будет лишь пустой тратой времени и ресурсов. В большинстве случаев самоприсваивание не следует выполнять вообще.

Кроме того, в случаях, когда используется динамическое выделение памяти, самоприсваивание может быть даже опасным:

```
#include <iostream>

class SomeString
{
private:
    char *m_data;
    int m_length;

public:
    SomeString(const char *data="", int length=0) :
        m_length(length)
    {
        if (!length)
            m_data = nullptr;
        else
            m_data = new char[length];

        for (int i=0; i < length; ++i)
            m_data[i] = data[i];
    }


    SomeString& operator= (const SomeString &str);

    friend std::ostream& operator<<(std::ostream& out, const SomeString &s);
};

std::ostream& operator<<(std::ostream& out, const SomeString &s)
{
    out << s.m_data;
    return out;
}

// Перегрузка оператора присваивания (плохой вариант перегрузки)
SomeString& SomeString::operator= (const SomeString &str)
{
```

```
37.     // Если m_data уже имеет значение, то удаляем это значение
38.     if (m_data) delete[] m_data;
39.
40.     m_length = str.m_length;
41.
42.     // Копируем значение из str в m_data неявного объекта
43.     m_data = new char[str.m_length];
44.     for (int i=0; i < str.m_length; ++i)
45.         m_data[i] = str.m_data[i];
46.
47.     // Возвращаем текущий объект
48.     return *this;
49. }
50.
51. int main()
52. {
53.     SomeString anton("Anton", 7);
54.     SomeString employee;
55.     employee = anton;
56.     std::cout << employee;
57.
58.     return 0;
59. }
```

Запустите программу, и вы увидите, что выведется `Anton`, как и ожидалось.

Теперь замените функцию main() на следующую:

```
1.  int main()
2.  {
3.      SomeString anton("Anton", 7);
4.      anton = anton;
5.      std::cout << anton;
6.
7.      return 0;
8.  }
```

В результате вы получите либо значение-мусор, либо сбой.

Рассмотрим, что происходит при выполнении операции присваивания, когда неявный и переданный в качестве аргумента объекты являются объектом `anton`. В этом случае `m_data` равно `str.m_data` (т.е. `Anton`). Первое, что произойдет — функция перегрузки проверит, является ли уже значением неявного объекта строка `Anton`. Если является, то произойдет удаление этого значения, чтобы не случилась утечка памяти. Т.е. значение `m_data` неявного объекта удаляется, но дело в том, что `str.m_data` имеет тот же адрес памяти (значение которого удаляется)! Это означает, что `str.m_data` станет висячим указателем.

Позже, когда мы будем копировать данные из параметра `str` функции перегрузки в наш неявный объект, мы будем обращаться к висячему указателю `str.m_data`. Это приведет к тому, что мы либо скопируем данные-мусор, либо попытаемся получить

доступ к памяти, которую наше приложение больше не имеет в распоряжении (произойдет сбой).

Обнаружение и обработка самоприсваивания

К счастью, мы можем обнаружить выполнение самоприсваивания. Это делается с помощью достаточно простой проверки в функции перегрузки operator=():

```
1. // Перегрузка оператора присваивания (хороший вариант, его и используйте)
2. Drob& Drob::operator= (const Drob &drob)
3. {
4.     // Проверка на самоприсваивание
5.     if (this == &drob)
6.         return *this;
7. 
8.     // Выполняем копирование значений
9.     m_numerator = drob.m_numerator;
10.    m_denominator = drob.m_denominator;
11. 
12.    // Возвращаем текущий объект
13.    return *this;
14. }
```

Проверяя, является ли наш неявный объект тем же, что и передаваемый в качестве параметра, мы сможем сразу же возвратить его без выполнения какого-либо кода.

Обратите внимание, нет необходимости выполнять проверку на самоприсваивание в конструкторе копирования. Это связано с тем, что конструктор копирования вызывается только при создании новых объектов, а способа присвоить только что созданный объект самому себе, чтобы вызвать конструктор копирования — нет.

Оператор присваивания по умолчанию

В отличие от других операторов, компилятор автоматически предоставит открытый оператор присваивания по умолчанию для вашего класса при его использовании, если вы не предоставите его самостоятельно. В операторе присваивания по умолчанию выполняется почленное присваивание (которое является аналогичным почленной инициализации, используемой в конструкторах копирования, предоставляемых языком C++ по умолчанию).

Как и с другими конструкторами и операторами, вы можете запретить выполнение операции присваивания с объектами ваших классов, сделав оператор присваивания закрытым или используя ключевое слово delete:

```
1. #include <iostream>
2. #include <cassert>
3. 
4. class Drob
```

```
5. {
6. private:
7.     int m_numerator;
8.     int m_denominator;
9.
10.public:
11.    // Конструктор по умолчанию
12.    Drob(int numerator=0, int denominator=1) :
13.        m_numerator(numerator), m_denominator(denominator)
14.    {
15.        assert(denominator != 0);
16.    }
17.
18.    // Конструктор копирования
19.    Drob(const Drob ©) = delete;
20.
21.    // Перегрузка оператора присваивания
22.    Drob& operator= (const Drob &drob) = delete; // нет созданию копий объектов
    через операцию присваивания!
23.
24.    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
25.
26.};
27.
28.std::ostream& operator<<(std::ostream& out, const Drob &d1)
29.{
30.    out << d1.m_numerator << "/" << d1.m_denominator;
31.    return out;
32.}
33.
34.int main()
35.{
36.    Drob sixSeven(6, 7);
37.    Drob d;
38.    d = sixSeven; // ошибка компиляции, operator= был удален
39.    std::cout << d;
40.
41.    return 0;
42.}
```

Урок №153. Поверхностное и глубокое копирование

Поскольку язык C++ не может знать наперед всё о вашем классе, то конструктор копирования и оператор присваивания, которые C++ предоставляет по умолчанию, используют почленный метод копирования — **поверхностное копирование**. Это означает, что C++ выполняет копирование для каждого члена класса индивидуально (используя оператор присваивания по умолчанию вместо перегрузки оператора присваивания и прямую инициализацию вместо конструктора копирования). Когда классы простые (например, в них нет членов с динамически выделенной памятью), то никаких проблем с этим не должно возникать.

Рассмотрим следующий класс Drob:

```
#include <iostream>
#include <cassert>

class Drob
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Конструктор по умолчанию
    Drob(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
};

std::ostream& operator<<(std::ostream& out, const Drob &d1)
{
    out << d1.m_numerator << "/" << d1.m_denominator;
    return out;
}
```

Конструктор копирования и оператор присваивания по умолчанию, предоставляемые компилятором автоматически, выглядят примерно следующим образом:

```
#include <iostream>
#include <cassert>

class Drob
{
private:
    int m_numerator;
    int m_denominator;

```

```
public:
    // Конструктор по умолчанию
    Drob(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    // Конструктор копирования
    Drob(const Drob &d) :
        m_numerator(d.m_numerator), m_denominator(d.m_denominator)
    {
    }

    Drob& operator= (const Drob &drob);

    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
};

std::ostream& operator<<(std::ostream& out, const Drob &d1)
{
    out << d1.m_numerator << "/" << d1.m_denominator;
    return out;
}

// Перегрузка оператора присваивания
Drob& Drob::operator= (const Drob &drob)
{
    // Проверка на самоприсваивание
    if (this == &drob)
        return *this;

    // Выполняем копирование
    m_numerator = drob.m_numerator;
    m_denominator = drob.m_denominator;

    // Возвращаем текущий объект, чтобы иметь возможность выполнять цепочку операций присваивания
    return *this;
}
```

Поскольку эти конструктор копирования и оператор присваивания по умолчанию отлично подходят для выполнения копирования с объектами этого класса, то действительно нет никакого смысла писать здесь свои собственные версии конструктора копирования и перегрузки оператора.

Однако при работе с классами, в которых динамически выделяется память, почленное (поверхностное) копирование может вызывать проблемы! Это связано с тем, что **при поверхностном копировании указателя копируется только адрес указателя — никаких действий по содержимому адреса указателя не предпринимается**. Например:

```
#include <cstring> // для strlen()
#include <cassert> // для assert()

class SomeString
```

```
{
private:
    char *m_data;
    int m_length;

public:
    SomeString(const char *source="")
    {
        assert(source); // проверяем не является ли source нулевой строкой

        // Определяем длину source + плюс еще один символ для нуль-терминатора (символ завершения строки)
        m_length = strlen(source) + 1;

        // Выделяем достаточно памяти для хранения копируемого значения в соответствии с длиной этого значения
        m_data = new char[m_length];

        // Копируем значение по символам в нашу выделенную память
        for (int i=0; i < m_length; ++i)
            m_data[i] = source[i];

        // Убеждаемся, что строка завершена
        m_data[m_length-1] = '\0';
    }

    ~SomeString() // деструктор
    {
        // Освобождаем память, выделенную для нашей строки
        delete[] m_data;
    }

    char* getString() { return m_data; }
    int getLength() { return m_length; }
};
```

Вышеприведенный класс — это обычный строковый класс, в котором выделяется память для хранения передаваемой строки. Здесь мы не определяли конструктор копирования или перегрузку оператора присваивания. Следовательно, язык C++ предоставит конструктор копирования и оператор присваивания по умолчанию, которые будут выполнять поверхностное копирование. Конструктор копирования выглядит примерно следующим образом:

```
SomeString::SomeString(const SomeString &source) :
    m_length(source.m_length), m_data(source.m_data)
{
}
```

Здесь `m_data` - это всего лишь поверхностная копия указателя `source.m_data`, поэтому теперь они оба указывают на один и тот же адрес памяти. Теперь рассмотрим следующий фрагмент кода:

```
int main()
{
    SomeString hello("Hello, world!");
    {
```

```
5.          SomeString copy = hello; // используется конструктор копирования по
    умолчанию
6.      } // объект copy является локальной переменной, которая уничтожается
    здесь. Деструктор удаляет значение-строку объекта copy, оставляя,
    таким образом, hello с висячим указателем
7.
8.      std::cout << hello.getString() << '\n'; // здесь неопределенные результаты
9.
10.     return 0;
11. }
```

Хотя этот код выглядит достаточно безвредным, но он имеет в себе коварную проблему, которая приведет к сбою программы! Можете найти эту проблему? Если нет, то ничего страшного.

Разберем этот код по строкам:

```
1. SomeString hello("Hello, world!");
```

Строка кода, приведенная выше, безвредна. Здесь вызывается конструктор класса SomeString, который выделяет память, заставляет `hello.m_data` указывать на эту память, а затем копирует в выделенный адрес памяти значение – строку `Hello, world!`.

```
1. SomeString copy = hello; // используется конструктор копирования по умолчанию
```

Эта строка также кажется достаточно безвредной, но именно она и является источником нашей коварной проблемы! При обработке этой строки C++ будет использовать конструктор копирования по умолчанию (так как мы не предоставили своего). Выполнится поверхностное копирование, результатом чего будет инициализация `copy.m_data` адресом, на который указывает `hello.m_data`. И теперь `copy.m_data` и `hello.m_data` оба указывают на одну и ту же часть памяти!

```
1. } // объект copy уничтожается здесь
```

Когда объект-копия выходит из области видимости, то вызывается деструктор SomeString для этой копии. Деструктор удаляет динамически выделенную память, на которую указывают как `copy.m_data`, так и `hello.m_data`! Следовательно, удаляя копию, мы также (случайно) удаляем и данные `hello`. Объект `copy` затем уничтожается, но `hello.m_data` остается указывать на удаленную память!

```
1. std::cout << hello.getString() << '\n'; // здесь неопределенные результаты
```

Теперь вы поняли, почему эта программа работает не совсем так, как нужно. Мы удалили значение-строку, на которую указывал `hello`, а сейчас пытаемся вывести это значение.

Корнем этой проблемы является поверхностное копирование, выполняемое конструктором копирования по умолчанию. Такое копирование почти всегда приводит к проблемам.

Глубокое копирование

Одним из решений этой проблемы является выполнение глубокого копирования. **При глубоком копировании память сначала выделяется для копирования адреса, который содержит исходный указатель, а затем для копирования фактического значения**. Таким образом копия находится в отдельной, от исходного значения, памяти и они никак не влияют друг на друга. Для выполнения глубокого копирования нам необходимо написать свой собственный конструктор копирования и перегрузку оператора присваивания.

Рассмотрим это на примере с классом SomeString:

```
1.  // Конструктор копирования
2.  SomeString::SomeString(const SomeString& source)
3.  {
4.      // Поскольку m_length не является указателем, то мы можем выполнить
    поверхностное копирование
5.      m_length = source.m_length;
6.
7.      // m_data является указателем, поэтому нам нужно выполнить глубокое
    копирование, при условии, что этот указатель не является нулевым
8.      if (source.m_data)
9.      {
10.         // Выделяем память для нашей копии
11.         m_data = new char[m_length];
12.
13.         // Выполняем копирование
14.         for (int i=0; i < m_length; ++i)
15.             m_data[i] = source.m_data[i];
16.     }
17.     else
18.         m_data = 0;
19. }
```

Как вы видите, реализация здесь более углубленная, нежели при поверхностном копировании! Во-первых, мы должны проверить, имеет ли исходный объект ненулевое значение вообще (строка №8). Если имеет, то мы выделяем достаточно памяти для хранения копии этого значения (строка №11). Наконец, копируем значение-строку (строки №14-15).

Теперь рассмотрим перегрузку оператора присваивания:

```
1.  // Оператор присваивания
2.  SomeString& SomeString::operator=(const SomeString & source)
3.  {
4.      // Проверка на самоприсваивание
```

```
5.      if (this == &source)
6.          return *this;
7.
8.      // Сначала нам нужно очистить предыдущее значение m_data (члена неявного объекта)
9.      delete[] m_data;
10.
11.     // Поскольку m_length не является указателем, то мы можем выполнить поверхностное копирование
12.     m_length = source.m_length;
13.
14.     // m_data является указателем, поэтому нам нужно выполнить глубокое копирование, при условии, что этот указатель не является нулевым
15.     if (source.m_data)
16.     {
17.         // Выделяем память для нашей копии
18.         m_data = new char[m_length];
19.
20.         // Выполняем копирование
21.         for (int i=0; i < m_length; ++i)
22.             m_data[i] = source.m_data[i];
23.     }
24.     else
25.         m_data = 0;
26.
27.     return *this;
28. }
```

Заметили, что код перегрузки очень похож на код конструктора копирования? Но здесь есть 3 основных отличия:

- Мы добавили проверку на самоприсваивание.
- Мы возвращаем текущий объект (с помощью указателя *this), чтобы иметь возможность выполнить цепочку операций присваивания.
- Мы явно удаляем любое значение, которое объект уже хранит (чтобы не произошло утечки памяти).

При вызове перегруженного оператора присваивания, объект, которому присваивается другой объект, может содержать предыдущее значение, которое нам необходимо очистить/удалить, прежде чем мы выделим память для нового значения. С не динамически выделенными переменными (которые имеют фиксированный размер) нам не нужно беспокоиться, поскольку новое значение просто перезапишет старое. Однако с динамически выделенными переменными нам нужно явно освободить любую старую память до того, как мы выделим любую новую память. Если мы этого не сделаем, сбоя не будет, но произойдет утечка памяти, которая будет съедать нашу свободную память каждый раз, когда мы будем выполнять операцию присваивания!

Лучшее решение

В Стандартной библиотеке C++ классы, которые работают с динамически выделенной памятью, такие как std::string и std::vector, имеют свое собственное управление памятью и свои конструкторы копирования и перегрузку операторов присваивания, которые выполняют корректное глубокое копирование. Поэтому, вместо написания своих собственных конструкторов копирования и перегрузки оператора присваивания, вы можете выполнять инициализацию или присваивание строк, или векторов, как обычных переменных фундаментальных типов данных! Это гораздо проще, менее подвержено ошибкам, и вам не нужно тратить время на написание лишнего кода!

Заключение

- Конструктор копирования и оператор присваивания, предоставляемые по умолчанию языком C++, выполняют поверхностное копирование, что отлично подходит для классов без динамически выделенных членов.
- Классы с динамически выделенными членами должны иметь конструктор копирования и перегрузку оператора присваивания, которые выполняют глубокое копирование.
- Используйте функциональность классов из Стандартной библиотеки C++, нежели самостоятельно выполняйте/реализовывайте управление памятью.

Глава №9. Итоговый тест

В этой главе мы рассмотрели перегрузку операторов, перегрузку операций преобразования типов данных, а также несколько тем, связанных с конструктором копирования.

Теория

Перегрузка оператора - это специфическая перегрузка функции, которая позволяет использовать операторы с объектами пользовательских классов. При перегрузке операторов их функционал и назначение следует сохранять максимально приближенно к их первоначальному применению. Если суть применяемого оператора с объектами пользовательских классов интуитивно не понятна, то лучше использовать функцию с именем, вместо перегрузки оператора.

Операторы могут быть перегружены через обычные функции, через дружественные функции и через методы класса. Следующие правила помогут сориентироваться, какой способ перегрузки и когда следует использовать:

- Перегрузку операторов присваивания (`=`), индекса (`[]`), вызова функции (`()`) или выбора члена (`->`) выполняйте через методы класса.
- Перегрузку унарных операторов выполняйте через методы класса.
- Перегрузку бинарных операторов, которые изменяют свой левый операнд (например, оператор `+=`) выполняйте через методы класса.
- Перегрузку бинарных операторов, которые не изменяют свой левый операнд (например, оператор `+`) выполняйте через обычные или дружественные функции.

Перегрузка операций преобразования типов данных используется для явного или неявного преобразования объектов пользовательского класса в другой тип данных.

Конструктор копирования - это особый тип конструктора, используемый для инициализации объекта другим объектом того же класса. Конструкторы копирования используются в прямой/uniform-инициализации объектов объектами того же типа, копирующей инициализации (`Fraction f = Fraction(7,4)`) и при передаче или возврате параметров по значению.

Если вы не предоставите свой конструктор копирования, то компилятор автоматически его предоставит. Конструкторы копирования по умолчанию (предоставляемые компилятором) используют **почленную инициализацию**. Это

означает, что каждый член объекта копии инициализируется соответствующим членом исходного объекта. Конструктор копирования может быть проигнорирован компилятором в целях оптимизации, даже если он имеет побочные эффекты, поэтому сильно не полагайтесь на свой конструктор копирования.

Конструкторы считаются **конструкторами преобразования** по умолчанию. Это означает, что компилятор будет использовать их для неявной конвертации объектов других типов данных в объекты вашего класса. Вы можете избежать этого, используя **ключевое слово explicit**. Вы также можете удалить функции внутри своего класса, включая конструктор копирования и перегруженный оператор присваивания, если это необходимо. И если позже в программе будет вызываться удаленная функция, то компилятор выдаст ошибку.

Оператор присваивания может быть перегружен для выполнения операций присваивания с объектами вашего класса. Если вы не предоставите перегруженный оператор присваивания сами, то компилятор создаст его за вас. Перегруженные операторы присваивания всегда должны иметь проверку на самоприсваивание.

По умолчанию оператор присваивания и конструктор копирования, предоставляемые компилятором, выполняют почленную инициализацию/присваивание, что является **поверхностным копированием**. Если в вашем классе есть динамически выделенные члены, то это, скорее всего, приведет к проблемам (несколько объектов могут указывать на одну и ту же выделенную память). В таком случае вам нужно будет явно определить свой конструктор копирования и перегрузку оператора присваивания для выполнения **глубокого копирования**.

Тест

Задание №1

Предположим, что Square - это класс, а `square` - это объект этого класса. Какой способ перегрузки лучше использовать для следующих операторов?

- `square + square`
- `-square`
- `std::cout << square`
- `square = 7;`

Задание №2

Напишите класс Average, который будет вычислять среднее значение всех передаваемых ему целых чисел. Используйте два члена: первый должен быть типа int32_t и использоваться для вычисления суммы всех передаваемых чисел, второй должен быть типа int8_t и использоваться для вычисления количества передаваемых чисел. Чтобы найти среднее значение, нужно разделить сумму на количество.

a) Следующий код функции main():

```
int main()
{
    Average avg;

    avg += 5;
    std::cout << avg << '\n'; // 5 / 1 = 5

    avg += 9;
    std::cout << avg << '\n'; // (5 + 9) / 2 = 7

    avg += 19;
    std::cout << avg << '\n'; // (5 + 9 + 19) / 3 = 11

    avg += -9;
    std::cout << avg << '\n'; // (5 + 9 + 19 - 9) / 4 = 6

    (avg += 7) += 11; // выполнение цепочки операций
    std::cout << avg << '\n'; // (5 + 9 + 19 - 9 + 7 + 11) / 6 = 7

    Average copy = avg;
    std::cout << copy << '\n';

    return 0;
}
```

Должен выдавать следующий результат:

```
5
7
11
6
7
7
```

b) Требуется ли этому классу явный конструктор копирования или оператор присваивания?

Задание №3

Напишите свой собственный класс-массив целых чисел IntArray (не используйте std::array или std::vector). Пользователи должны передавать размер массива при создании объекта этого класса, а сам массив (переменная-член) должен выделяться динамически. Используйте стейтменты assert для проверки передаваемых значений, а также свой конструктор копирования и перегрузку оператора присваивания, если это необходимо, чтобы следующий код:

```
#include <iostream>

IntArray fillArray()
{
    IntArray a(6);
    a[0] = 6;
    a[1] = 7;
    a[2] = 3;
    a[3] = 4;
    a[4] = 5;
        a[5] = 8;

    return a;
}

int main()
{
    IntArray a = fillArray();
    std::cout << a << '\n';

    IntArray b(1);
    a = a;
    b = a;

    std::cout << b << '\n';

    return 0;
}
```

Выдавал следующий результат:

```
6 7 3 4 5 8
6 7 3 4 5 8
```

Задание №4

Значение типа с плавающей точкой - это число с десятичной дробью, где количество цифр после точки (дробная часть) может меняться. Значение типа с фиксированной точкой - это число с дробью, где дробь (после точки) фиксированная.

Вам нужно написать класс для реализации значений типа с фиксированной точкой с двумя цифрами после точки (например, `11.47`, `5.00` или `1465.78`). Диапазон

класса должен быть от `-32768.99` до `32767.99`, в дробной части могут быть любые две цифры, не допускайте проблем с точностью.

a) Какого типа данных переменную-член следует использовать для реализации значений типа с фиксированной точкой с 2-мя цифрами после точки? (Обязательно прочитайте ответ, прежде чем приступать к выполнению следующего задания)

b) Напишите класс FixedPoint, который реализует рекомендуемое решение из предыдущего задания. Если дробная или целая части значения являются отрицательными, то число должно рассматриваться, как отрицательное. Реализуйте перегрузку необходимых операторов и напишите необходимые конструкторы, чтобы следующий код функции main():

```
int main()
{
    FixedPoint a(37, 58);
    std::cout << a << '\n';

    FixedPoint b(-3, 9);
    std::cout << b << '\n';

    FixedPoint c(4, -7);
    std::cout << c << '\n';

    FixedPoint d(-5, -7);
    std::cout << d << '\n';

    FixedPoint e(0, -3);
    std::cout << e << '\n';

    std::cout << static_cast<double>(e) << '\n';

    return 0;
}
```

Выдавал следующий результат:

```
37.58
-3.09
-4.07
-5.07
-0.03
-0.03
```

> ***Подсказка***: Для вывода значения конвертируйте его в double, используя оператор static_cast.

c) Теперь добавьте конструктор, который будет принимать значение типа double. Вы можете округлить целую часть (слева от точки) с помощью функции round() (которая находится в заголовочном файле cmath).

> ***Подсказки:***
>
> - Вы можете получить целую часть от числа типа double путем конвертации числа типа double в число типа int.
> - Для перемещения одной цифры влево от точки используйте умножение на 10. Для перемещения двух цифр используйте умножение на 100.

Следующий код функции main():

```
1. int main()
2. {
3.     FixedPoint a(0.03);
4.     std::cout << a << '\n';
5.
6.     FixedPoint b(-0.03);
7.     std::cout << b << '\n';
8.
9.     FixedPoint c(4.01); // сохранится, как 4.0099999..., поэтому нам нужно
   это всё округлить
10.    std::cout << c << '\n';
11.
12.    FixedPoint d(-4.01); // сохранится, как -4.0099999…, поэтому нам нужно это
   всё округлить
13.    std::cout << d << '\n';
14.
15.    return 0;
16. }
```

Должен выдавать следующий результат:

```
0.03
-0.03
4.01
-4.01
```

d) Выполните перегрузку следующих операторов: `==`, `>>`, `-` (унарный) и `+` (бинарный).

Следующая программа:

```
1. void SomeTest()
2. {
3.     std::cout << std::boolalpha;
4.     std::cout << (FixedPoint(0.75) + FixedPoint(1.23) == FixedPoint(1.98)) <<
   '\n'; // оба значения положительные, никакого переполнения
5.     std::cout << (FixedPoint(0.75) + FixedPoint(1.50) == FixedPoint(2.25)) <<
   '\n'; // оба значения положительные, переполнение
6.     std::cout << (FixedPoint(-0.75) + FixedPoint(-1.23) == FixedPoint(-
   1.98)) << '\n'; // оба значения отрицательные, никакого переполнения
7.     std::cout << (FixedPoint(-0.75) + FixedPoint(-1.50) == FixedPoint(-
   2.25)) << '\n'; // оба значения отрицательные, переполнение
8.     std::cout << (FixedPoint(0.75) + FixedPoint(-1.23) == FixedPoint(-
   0.48)) << '\n'; // второе значение отрицательное, никакого переполнения
```

```
    std::cout << (FixedPoint(0.75) + FixedPoint(-1.50) == FixedPoint(-0.75)) << '\n'; // второе значение отрицательное, возможно переполнение
    std::cout << (FixedPoint(-0.75) + FixedPoint(1.23) == FixedPoint(0.48)) << '\n'; // первое значение отрицательное, никакого переполнения
    std::cout << (FixedPoint(-0.75) + FixedPoint(1.50) == FixedPoint(0.75)) << '\n'; // первое значение отрицательное, возможно переполнение
}

int main()
{
    SomeTest();

    FixedPoint a(-0.48);
    std::cout << a << '\n';

    std::cout << -a << '\n';

    std::cout << "Enter a number: "; // введите 5.678
    std::cin >> a;

    std::cout << "You entered: " << a << '\n';

    return 0;
}
```

Должна выдавать следующий результат:

```
true
true
true
true
true
true
true
true
-0.48
0.48
Enter a number: 5.678
You entered: 5.68
```

> ***Подсказка***: Для выполнения перегрузки оператора `>>` используйте конструктор с параметром типа double для создания анонимного объекта класса FixedPoint, а затем присвойте этот объект параметру функции перегрузки оператора `>>`.

Урок №154. Типы связей между объектами

Наша жизнь полна повторяющихся шаблонов, отношений и иерархий между объектами. Изучая их, мы получаем более глубокое представление о том, как они работают и взаимодействуют между собой в реальной жизни.

Например, мы идем по улице и видим ярко-желтый объект на зеленом продолговатом объекте. Мы понимаем, что ярко-желтый объект - это цветок, а зеленый - стебель. Даже если мы до этого никогда не видели такое растение, мы все равно понимаем, что объекты на стебле являются листьями, которые взаимодействуют с солнечным светом, а цветок помогает растению в размножении и выживании. Мы также знаем, что, если убить растение, цветок тоже погибнет.

Но как мы можем это знать и судить об этом объекте как о растении, не встречая его раньше никогда? Дело в том, что у нас есть знания о растениях и мы осознаем, что объект, который мы встретили на улице, относится именно к растениям. Мы знаем, что большинство растений имеют листья, а некоторые еще и цветки. Также мы знаем, что листья взаимодействуют с солнечным светом (даже если не понимаем полный процесс этого взаимодействия), и существование цветка напрямую зависит от растения. И поскольку мы всё это знаем, и это относится к растениям, то мы можем делать подобные выводы и об объекте на улице, который соответствует этим нашим знаниям о растениях.

Аналогично, в программировании также полно повторяющихся шаблонов, отношений и иерархий. В частности, когда речь заходит об объектах в программировании, то те же шаблоны, которыми мы руководствуемся по отношению к объектам в реальной жизни, применимы и к объектам в программировании, которые мы создаем сами. Изучая эти отношения, повторяющиеся шаблоны и иерархии подробнее, мы можем понять, как улучшить код для его повторного использования в других программах и как писать классы, функционал которых можно будет легко расширить.

На предыдущих уроках мы уже рассматривали принципы, относящиеся к повторяющимся шаблонам: мы писали циклы и функции, которые выполняли определенные действия много раз. Кроме того, мы создавали свои собственные перечисления, структуры, классы и объекты этих перечислений, структур и классов.

Мы также изучили некоторые примитивные формы иерархии, такие как массивы (которые позволяют группировать элементы в более крупную структуру) и рекурсию (когда функция вызывает сама себя).

Однако мы еще достаточно не фокусировались на взаимосвязях между объектами в программировании.

Связи между объектами в программировании

Существует много разных типов отношений, которые два объекта могут иметь в реальной жизни, и мы используем определенные слова (**типы отношений**) для их описания, например:

- Квадрат *«является»* геометрической фигурой.
- Автомобиль *«имеет»* руль.
- Программист *«использует»* клавиатуру.
- Цветок *«зависит от»* растения.
- Ученик является *«членом»* класса.
- Наш мозг существует как *«часть»* нас самих.

Все эти типы отношений имеют свои аналоги в языке C++.

На следующих уроках мы рассмотрим нюансы типов отношений «имеет», «использует», «зависит от», «член чего-то» и «часть чего-то», и покажем, как они могут быть полезными в контексте классов в C++.

Затем мы перейдем к изучению отношений типа «является», рассматривая наследование и виртуальные функции в C++.

Урок №155. Композиция объектов

В реальной жизни сложные объекты часто состоят из меньших, более простых объектов. Например, автомобиль состоит из металлической рамы, двигателя, 4-х колес, коробки передач, руля и большого количества других деталей. Персональный компьютер состоит из центрального процессора, материнской платы, памяти и т.д. Даже вы состоите из небольших частей: у вас есть голова, ноги, руки и т.д. Процесс построения сложных объектов из более простых называется **композицией объекта**.

Типы композиции объектов

В композиции между двумя объектами представлен тип отношения «имеет». Автомобиль «имеет» коробку передач. Ваш компьютер «имеет» центральный процессор. Вы «имеете» сердце. Сложный объект иногда называют **целым** (или ***"родителем"***). Более простой объект часто называют **частью** (или ***"дочерним элементом"***, ***"компонентом"***).

Ранее мы рассматривали, что структуры и классы могут иметь члены разных типов данных (например, фундаментальных или вообще других классов). Когда мы создаем классы с членами, то мы, по сути, создаем сложный объект из более простых частей, что и является композицией объекта. По этой причине структуры и классы еще называют **составными типами данных**.

Композиция объектов полезна в контексте языка C++, поскольку позволяет создавать сложные классы, объединяя более простые и легко управляемые части. Это уменьшает сложность и позволяет писать код быстрее и с меньшим количеством ошибок, так как мы можем повторно использовать код, который уже был написан, протестирован, и является рабочим.

Существует два основных подтипа композиции объекта: **композиция** и **агрегация**. На этом уроке мы рассмотрим композицию, а на следующем — агрегацию.

> ***Примечание по терминологии*:** Термин «композиция» часто используется для обозначения композиции и агрегации, как единого целого, а не только подтипа композиция. На этом уроке мы будем использовать термин «композиция объекта», когда будем иметь в виду целое (и композицию, и агрегацию), а термин «композиция», когда речь будет идти конкретно о подтипе композиция.

Композиция

Для реализации композиции объект и часть должны иметь следующие отношения:

- Часть (член) является частью объекта (класса).
- Часть (член) может принадлежать только одному объекту (классу) в моменте.
- Часть (член) существует, управляемая объектом (классом).
- Часть (член) не знает о существовании объекта (класса).

Хорошим примером композиции в жизни является взаимосвязь между телом человека и его сердцем. Рассмотрим это детально.

Отношения в композиции - это отношения части-целого. Например, сердце является частью тела человека. Часть в композиции может быть частью только одного объекта в моменте. Сердце, которое является частью тела одного человека, не может быть одновременно частью тела еще одного человека.

В отношениях внутри композиции объект несет ответственность за существование частей. Чаще всего это означает, что часть создается при создании объекта и уничтожается при его уничтожении. Но в более широком смысле это означает, что объект управляет временем жизни части таким образом, что пользователь, который использует объект, не должен участвовать в этом. Например, при создании тела создается и сердце. Когда тело человека уничтожается, то и его сердце уничтожается тоже.

И, наконец, часть не знает о существовании целого. Ваше сердце работает круглосуточно, не зная, что оно является частью более крупной организации. Это называется однонаправленным отношением, поскольку тело знает о сердце, а сердце о теле - нет.

Обратите внимание, в композиции ничего не говорится о переносимости частей. Сердце можно пересадить из тела одного человека в тело другому человеку. Однако даже после пересадки оно по-прежнему будет соответствовать требованиям композиции (сердце принадлежит другому человеку и может быть частью только этого другого человека и никого больше до тех пор, пока сердце не пересадят снова).

Наш уже любимый класс Drob является отличным примером композиции:

```
1. class Drob
2. {
```

```
3.  private:
4.      int m_numerator;
5.      int m_denominator;
6.
7.  public:
8.      Drob(int numerator=0, int denominator=1):
9.          m_numerator(numerator), m_denominator(denominator)
10.     {
11.         // Мы поместили метод reduce() в конструктор, чтобы убедиться, что
    все дроби будут уменьшены!
12.         reduce();
13.     }
14. };
```

Этот класс имеет два члена: `m_numerator` (числитель) и `m_denominator` (знаменатель). Числитель и знаменатель являются частью Drob, они находятся в этом классе. Они не могут принадлежать еще одному классу одновременно. `m_numerator` и `m_denominator` не знают, что они являются частью Drob, они просто хранят целые числа. При создании объекта класса Drob, создаются и `m_numerator`, и `m_denominator`. Когда объект класса Drob уничтожается, то и эти члены уничтожаются тоже.

Так как типом отношений в композиции объектов является «имеет» (тело «имеет» сердце, Drob «имеет» `m_denominator`), то мы можем сказать, что композиция имеет и тип отношения «часть чего-то» (сердце является «частью» тела, `m_numerator` является «частью» Drob). Композиция часто используется для моделирования физических отношений, где один объект физически находится внутри другого объекта.

Части в композиции могут быть как сингулярными (единственными в своем роде), так и мультипликативными (таких частей может быть несколько). Например, в теле человека есть только одно сердце (сердце является сингулярным), но также 20 пальцев (пальцы являются мультипликативными и могут быть реализованы в виде массива).

Реализация композиций

Композиции являются одними из самых простых типов отношений для реализации на языке C++. Это обычные структуры или классы с обычными членами. Поскольку члены существуют непосредственно как части структур/классов, то их продолжительность жизни напрямую зависит от продолжительности жизни объектов этих структур/классов.

Композиции, в которых выполняется динамическое выделение или освобождение памяти, могут быть реализованы с использованием указателей в виде членов этих

структур или классов. В этом случае управление памятью полностью накладывается на композицию.

В общем, если вы *можете* создать класс, используя композицию, то вы *должны* создать класс, используя композицию. Классы с реализованной композицией являются простыми, гибкими и надежными.

Еще один пример

Во многих играх есть существа или объекты, которые перемещаются по карте или вокруг каких-то объектов. Все эти существа/объекты имеют одну общую вещь - локацию. В следующем примере мы создадим класс Creature, который использует класс Point2D для хранения местоположения (локации) существа.

Сначала создадим класс Point2D. Наше существо будет находиться в 2D-измерении, поэтому в нашем классе будет 2 члена: `x` и `y`.

Point2D.h:

```
#ifndef POINT2D_H
#define POINT2D_H

#include <iostream>

class Point2D
{
private:
    int m_x;
    int m_y;

public:
    // Конструктор по умолчанию
    Point2D()
        : m_x(0), m_y(0)
    {
    }

    // Специфический конструктор
    Point2D(int x, int y)
        : m_x(x), m_y(y)
    {
    }

    // Перегрузка оператора вывода
    friend std::ostream& operator<<(std::ostream& out, const Point2D &point)
    {
        out << "(" << point.m_x << ", " << point.m_y << ")";
        return out;
    }

    // Функция доступа
    void setPoint(int x, int y)
    {
```

```
35.         m_x = x;
36.         m_y = y;
37.     }
38.
39. };
40.
41. #endif
```

Обратите внимание, поскольку мы реализовали все наши функции в заголовочном файле (ради сохранения краткости примера), у нас нет Point2D.cpp.

Класс Point2D является целым, которое состоит из частей: `x` и `y`, продолжительность жизни которых напрямую зависит от продолжительности жизни объектов класса Point2D.

Теперь создадим класс Creature. У нашего существа будет 2 свойства: имя (строка) и местоположение (объект класса Point2D).

Creature.h:

```
1.  #ifndef CREATURE_H
2.  #define CREATURE_H
3.
4.  #include <iostream>
5.  #include <string>
6.  #include "Point2D.h"
7.
8.  class Creature
9.  {
10. private:
11.     std::string m_name;
12.     Point2D m_location;
13.
14. public:
15.     Creature(const std::string &name, const Point2D &location)
16.         : m_name(name), m_location(location)
17.     {
18.     }
19.
20.     friend std::ostream& operator<<(std::ostream& out, const Creature
    &creature)
21.     {
22.         out << creature.m_name << " is at " << creature.m_location;
23.         return out;
24.     }
25.
26.     void moveTo(int x, int y)
27.     {
28.         m_location.setPoint(x, y);
29.     }
30. };
31. #endif
```

Класс Creature также является целым, которое состоит из частей: `m_name` и `m_location`, продолжительность жизни которых также зависит от продолжительности жизни объектов класса Creature.

И, наконец, main.cpp:

```
#include <iostream>
#include <string>
#include "Creature.h"
#include "Point2D.h"

int main()
{
    std::cout << "Enter a name for your creature: ";
    std::string name;
    std::cin >> name;
    Creature creature(name, Point2D(5, 6));

    while (1)
    {
        // Выводим имя существа и его местоположение
        std::cout << creature << '\n';

        std::cout << "Enter new X location for creature (-1 to quit): ";
        int x=0;
        std::cin >> x;
        if (x == -1)
            break;

        std::cout << "Enter new Y location for creature (-1 to quit): ";
        int y=0;
        std::cin >> y;
        if (y == -1)
            break;

        creature.moveTo(x, y);
        }

    return 0;
}
```

Результат выполнения программы:

```
Enter a name for your creature: Anton
Anton is at (5, 6)
Enter new X location for creature (-1 to quit): 7
Enter new Y location for creature (-1 to quit): 11
Anton is at (7, 11)
Enter new X location for creature (-1 to quit): 2
Enter new Y location for creature (-1 to quit): 4
Anton is at (2, 4)
Enter new X location for creature (-1 to quit): -1
```

Вариации композиции

Хотя в большинстве композиций создание/удаление частей происходит непосредственно при создании/удалении самой композиции, есть вариации композиции, где правила несколько видоизменены, например:

- Композиция может отложить создание некоторых из своих частей до тех пор, пока они не понадобятся. Например, строковый класс может не создавать динамический массив символов до тех пор, пока пользователь не предоставит данные, которые эта строка могла бы хранить.
- Композиция может предпочесть использовать часть, которая была предоставлена ей в качестве входных данных, а не создавать эту часть самостоятельно.
- Композиция может делегировать уничтожение своих частей другому объекту (например, процедуре сбора мусора).

Ключевым моментом является то, что композиция должна управлять своими частями самостоятельно, без вмешательства пользователя композиции.

Композиция и подклассы

Одним из самых частых вопросов, которые задают новички, когда дело доходит до композиции объекта, является: «Когда я должен использовать подкласс вместо непосредственной реализации?». Например, вместо использования класса Point2D для реализации местоположения Creature, мы могли бы просто добавить в класс Creature еще два члена (`m_x` и `m_y`) и записать весь код реализации местоположения в классе Creature. Тем не менее, в создании Point2D есть ряд преимуществ:

Преимущество №1: Каждый отдельный класс можно сохранить относительно простым/понятным и сфокусировать на выполнение одной конкретной задачи. Таким образом, писать классы легче и понимать их проще. Например, в Point2D всё вертится только вокруг местоположения и это позволяет сохранить общую картину программы более простой.

Преимущество №2: Каждый подкласс может быть автономным, что делает его многоразовым. Например, мы можем повторно использовать наш класс Point2D в совершенно другой программе. Или, если нашему Creature когда-либо понадобится еще один пункт в определении локации (например, место куда ему нужно будет добраться), мы можем просто добавить еще одну переменную-член в Point2D.

Преимущество №3: Родительский класс может оставить выполнение большей части сложной работы на подклассы, а сам сосредоточиться на координации потока данных между подклассами. Это поможет снизить общую сложность родительского объекта, поскольку родительский объект делегирует выполнение работы своим дочерним элементам, которые уже знают, как выполнять эти задания. Например, при перемещении нашего Creature, сам Creature делегирует выполнение перемещения классу Point2D, который уже понимает, как работать с местоположением. Таким образом, класс Creature не должен беспокоиться о том, как такие вещи реализовать.

Хорошим правилом является то, что **один класс должен выполнять одну конкретную задачу** (как в примере с функциями). Этой задачей может быть хранение, манипулирование данными или координация подклассов.

В нашем случае есть смысл в том, чтобы Creature не беспокоился о реализации местоположения. Задача Creature состоит не в том, чтобы знать все подробности, а в том, чтобы координировать поток данных и гарантировать, что каждый из подклассов знает, что он должен делать. То, как следует выполнять конкретные задания — зависит уже от каждого подкласса отдельно.

Урок №156. Агрегация

На уроке о композиции мы говорили, что композиция объекта - это процесс создания сложных объектов из более простых. Мы также говорили о подтипе композиции объектов - композиции. В отношениях внутри композиции целое (класс) несет ответственность за существование частей (членов).

На этом уроке мы рассмотрим второй подтип композиции объекта — агрегацию.

Агрегация

Для реализации агрегации целое и его части должны соответствовать следующим отношениям:

- Часть (член) является частью целого (класса).
- Часть (член) может принадлежать более чем одному целому (классу) в моменте.
- Часть (член) существует, *не* управляемая целым (классом).
- Часть (член) не знает о существовании целого (класса).

Как и в случае с подтипом композиция, отношения в агрегации также являются отношениями части-целого и однонаправленные. Однако, в отличие от композиции, части могут принадлежать более чем одному целому в моменте, и целое не управляет существованием и продолжительностью жизни частей. При создании/уничтожении агрегации, целое не несет ответственности за создание/уничтожение своих частей.

Например, рассмотрим отношения между человеком и его домашним адресом. У каждого человека есть свой адрес. Однако этот адрес может принадлежать более чем одному человеку в моменте, например, вам и вашему соседу по комнате или родственникам, которые живут вместе с вами. К тому же этот адрес не управляется человеком - адрес существовал до того, как человек заселился и будет существовать после того, как человек выселится. Кроме того, человек знает, по какому адресу он живет, но адрес, в свою очередь, не знает, что это за человек и вообще, сколько их там находится. Такие отношения и являются **агрегацией**.

В качестве альтернативы рассмотрим автомобиль и двигатель. Двигатель является частью автомобиля. И хотя двигатель принадлежит автомобилю, он может принадлежать и другим объектам, например, человеку, которому принадлежит автомобиль. Автомобиль не несет ответственности за создание или уничтожение

двигателя. И в то же время автомобиль знает, что у него есть двигатель (ведь благодаря ему он двигается), но сам двигатель не знает, что он является частью автомобиля.

Когда дело доходит до моделирования физических объектов, использование термина «уничтожение» может быть немного расплывчатым. Возникает вопрос: «Если бы метеорит упал с неба и раздавил машину, то можно ли считать, что и все части машины также были бы уничтожены?». Да, конечно. Но это вина метеорита, а не автомобиля. Важным моментом является то, что автомобиль не несет ответственности за уничтожение своих частей (но есть и внешняя сила, которая может этому поспособствовать).

Мы можем сказать, что типом отношений в агрегации является *«имеет»* (отдел «имеет» работников, автомобиль «имеет» двигатель).

Подобно композиции, части агрегации могут быть сингулярными или мультипликативными.

Реализация агрегации

Поскольку агрегация подобна композиции, так как обе состоят из отношений части-целого, то они реализуются почти одинаково, а разница между ними в основном семантическая. В композиции мы добавляем части к целому, используя обычные переменные-члены (или указатели, когда в классе происходит динамическое выделение/освобождение памяти).

В агрегации мы также добавляем части к целому, используя переменные-члены. Однако этими переменными-членами обычно являются либо ссылки, либо указатели, которые указывают на объекты, созданные за пределами класса. Следовательно, агрегация принимает части, на которые она будет указывать, в качестве параметров конструктора или, если параметров нет, части добавляются позже через функции доступа или через перегруженные операторы.

Поскольку эти части существуют вне области видимости класса, то при уничтожении класса, переменные-члены в виде ссылок или указателей также уничтожаются (но не удаляются значения, на которые они указывают). Следовательно, сами части продолжают существовать дальше.

Рассмотрим пример Работника и Отдела подробнее. Чтобы было проще, в Отделе работает только один Работник и он не знает, Работником какого именно Отдела он является:

```
#include <iostream>
#include <string>

class Worker
{
private:
    std::string m_name;

public:
    Worker(std::string name)
        : m_name(name)
    {
    }

    std::string getName() { return m_name; }
};

class Department
{
private:
    Worker *m_worker; // чтобы было проще, в этом Отделе работает только один Работник, но их может быть и несколько

public:
    Department(Worker *worker = nullptr)
        : m_worker(worker)
    {
    }
};

int main()
{
    // Создаем Работника вне области видимости класса Department
    Worker *worker = new Worker("Anton"); // создаем Работника
    {
        // Создаем Отдел и передаем Работника в Отдел через параметр конструктора
        Department department(worker);

    } // department выходит из области видимости и уничтожается здесь

    // worker продолжает существовать дальше

    std::cout << worker->getName() << " still exists!";

    delete worker;

    return 0;
}
```

Здесь Работник создается независимо от Отдела, а затем переходит в параметр конструктора класса Отдела. Когда `department` уничтожается, то указатель `m_worker` уничтожается также, но сам Работник то не удаляется, он продолжает существовать до тех пор, пока не будет уничтожен в функции main().

Выбирайте правильные отношения

Хотя в вышеприведенном примере может показаться немного глупым, что Работник не знает, в каком Отделе он работает, но это совершенно нормально в контексте данной программы. Когда вы определяете тип отношений для реализации, внедряйте самые простые и понятные отношения, которые соответствуют вашим потребностям, а не те, которые, как вам кажется, подойдут лучше всего, но они сложнее и навороченнее.

Например, если вы создаете симулятор ремонта автомобилей, то вы можете захотеть реализовать автомобиль и двигатель через агрегацию, чтобы двигатель можно было вынуть и отдельно в нем разобраться. Однако, если вы пишете гоночный симулятор, вы можете захотеть реализовать автомобиль и двигатель через композицию, поскольку в гонке двигателю существовать отдельно от автомобиля никак нельзя.

Правило: Реализовывайте самые простые отношения, которые соответствуют потребностям вашей программы, а не то, что, как вам кажется, будет лучше.

Композиция и агрегация

В композиции:

- Используются обычные переменные-члены.
- Используются указатели, если класс реализовывает собственное управление памятью (происходит динамическое выделение/освобождение памяти).
- Класс ответственный за создание/уничтожение своих частей.

В агрегации:

- Используются указатели/ссылки, которые указывают/ссылаются на части вне класса.
- Класс не несет ответственности за создание/уничтожение своих частей.

Стоит отметить, что идеи композиции и агрегации не являются взаимоисключающими и могут свободно использоваться в одном классе. Вполне возможно реализовать класс, который отвечает за создание/уничтожение только определенных частей. Например, наш класс Department мог бы иметь и Имя, и Работника. Имя было бы добавлено в класс через композицию и создавалось/уничтожалось бы вместе с объектами класса Department. А Работник

был бы добавлен в Department через агрегацию и создавался/уничтожался бы независимо/отдельно.

Хотя агрегации могут быть чрезвычайно полезными, они также потенциально опасны. Поскольку в агрегации автоматически не осуществляется освобождение памяти, которую могут занимать части, то это должны контролировать вы сами. Если вы забыли выполнить очистку (освободить память), думая, что это должен сделать класс, то тогда произойдет утечка памяти, поэтому следует быть осторожным.

> ***Примечание***: По разным историческим и другим причинам, в отличие от композиции, определение агрегации не является единственно правильным - вы можете увидеть, что другие ресурсы/сайты/учебники определяют агрегацию несколько иначе, нежели изложено здесь. Это нормально, просто знайте об этом.

Тест

Задание №1

Что бы вы использовали (агрегацию или композицию) для создания следующих объектов? Список создаваемых объектов:

- Красный шар.
- Работодатель, который нанимает людей.
- Факультет в университете.
- Ваш возраст.
- Мешок с шариками.

Задание №2

Обновите вышеприведенный пример с Работником/Отделом так, чтобы Отдел мог состоять из нескольких Работников. Следующий код:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     // Создаем Работников вне области видимости класса Department
6.     Worker *w1 = new Worker("Anton");
7.     Worker *w2 = new Worker("Ivan");
8.     Worker *w3 = new Worker("Max");
9.
10.    {
11.        // Создаем Отдел и передаем Работников в качестве параметров
   конструктора
12.        Department department; // содаем пустой Отдел
```

```
13.         department.add(w1);
14.         department.add(w2);
15.         department.add(w3);
16.
17.         std::cout << department;
18.
19.     } // department выходит из области видимости и уничтожается здесь
20.
21.     std::cout << w1->getName() << " still exists!\n";
22.     std::cout << w2->getName() << " still exists!\n";
23.     std::cout << w3->getName() << " still exists!\n";
24.
25.     delete w1;
26.     delete w2;
27.     delete w3;
28.
29.     return 0;
30. }
```

Должен выдавать следующий результат:

```
Department: Anton Ivan Max
Anton still exists!
Ivan still exists!
Max still exists!
```

Подсказки:

- Используйте std::vector для хранения Работников.
- Используйте std::vector::push_back() для добавления Работника.
- Используйте std::vector::size() для получения длины std::vector.

Урок №157. Ассоциация

На предыдущих уроках мы рассмотрели два подтипа композиции объектов: композицию и агрегацию. Композиция объектов используется для моделирования отношений, в которых сложный объект (целое) состоит из нескольких более простых объектов (частей).

На этом уроке мы рассмотрим следующий тип отношений между двумя несвязанными объектами - ассоциацию. В отличие от композиции объектов, в ассоциации нет отношений "частей-целого".

Ассоциация

В ассоциации два несвязанных объекта должны соответствовать следующим отношениям:

- Первый объект (член) не связан со вторым объектом (классом).
- Первый объект (член) может принадлежать одновременно сразу нескольким объектам (классам).
- Первый объект (член) существует, не управляемый вторым объектом (классом).
- Первый объект (член) может знать или не знать о существовании второго объекта (класса).

В отличие от композиции или агрегации, где часть является частью целого, в ассоциации объекты между собой никак не связаны. Подобно агрегации, первый объект может принадлежать сразу нескольким объектам одновременно и не управляется ими. Однако, в отличие от агрегации, где отношения однонаправленные, в ассоциации отношения могут быть как однонаправленными, так и двунаправленными (когда оба объекта знают о существовании друг друга).

Отношения между врачами и пациентами — это отличный пример **ассоциации**. Врач связан с пациентом, но эти отношения нельзя назвать отношениями "части-целого". Врач может принимать десятки пациентов в день, а пациент может обращаться к нескольким врачам.

Мы можем сказать, что типом отношений в ассоциации является ***«использует»***. Врач «использует» пациента для получения дохода. Пациент «использует» врача, чтобы вылечить болезнь или улучшить свое самочувствие.

Реализация ассоциаций

Ассоциации реализованы по-разному. Однако чаще всего они реализованы через указатели, где классы указывают на объекты друг друга.

В следующем примере мы реализуем двунаправленную связь между Врачом и Пациентом, так как Врач должен знать своих пациентов в лицо, а Пациенты могут обращаться к разным Врачам:

```
#include <iostream>
#include <string>
#include <vector>

// Поскольку отношения между этими классами двунаправленные, то для класса Doctor здесь нужно использовать предварительное объявление
class Doctor;

class Patient
{
private:
    std::string m_name;
    std::vector<Doctor *> m_doctor; // благодаря вышеприведенному предварительному объявлению Doctor, эта строка не вызовет ошибку компиляции

    // Мы объявляем addDoctor закрытым, так как не хотим его публичного использования.
    // Вместо этого доступ к нему будет осуществляться через Doctor::addPatient().
    // Мы определим этот метод после определения класса Doctor, так как нам сначала нужно определить Doctor, чтобы использовать что-либо, связанное с ним
    void addDoctor(Doctor *doc);

public:
    Patient(std::string name)
        : m_name(name)
    {
    }

    // Мы реализуем перегрузку оператора вывода ниже определения класса Doctor, так как он как раз и требуется для реализации перегрузки
    friend std::ostream& operator<<(std::ostream &out, const Patient &pat);

    std::string getName() const { return m_name; }

    // Мы делаем класс Doctor дружественным, чтобы иметь доступ к закрытому методу addDoctor().
    // Примечание: Мы бы хотели сделать дружественным только один метод addDoctor(), но мы не можем это сделать, так как Doctor предварительно объявлен
    friend class Doctor;
};

class Doctor
{
private:
    std::string m_name;
    std::vector<Patient *> m_patient;

public:
```

```
42.     Doctor(std::string name):
43.         m_name(name)
44.     {
45.     }
46.
47.     void addPatient(Patient *pat)
48.     {
49.         // Врач добавляет Пациента
50.         m_patient.push_back(pat);
51.
52.         // Пациент добавляет Врача
53.         pat->addDoctor(this);
54.     }
55.
56.
57.     friend std::ostream& operator<<(std::ostream &out, const Doctor &doc)
58.     {
59.         unsigned int length = doc.m_patient.size();
60.         if (length == 0)
61.         {
62.             out << doc.m_name << " has no patients right now";
63.             return out;
64.         }
65.
66.         out << doc.m_name << " is seeing patients: ";
67.         for (unsigned int count = 0; count < length; ++count)
68.              out << doc.m_patient[count]->getName() << ' ';
69.
70.         return out;
71.     }
72.
73.     std::string getName() const { return m_name; }
74. };
75.
76. void Patient::addDoctor(Doctor *doc)
77. {
78.     m_doctor.push_back(doc);
79. }
80.
81. std::ostream& operator<<(std::ostream &out, const Patient &pat)
82. {
83.     unsigned int length = pat.m_doctor.size();
84.     if (length == 0)
85.     {
86.         out << pat.getName() << " has no doctors right now";
87.         return out;
88.     }
89.
90.     out << pat.m_name << " is seeing doctors: ";
91.     for (unsigned int count = 0; count < length; ++count)
92.         out << pat.m_doctor[count]->getName() << ' ';
93.
94.     return out;
95. }
96.
97.
98. int main()
99. {
100.          // Создаем Пациентов вне области видимости класса Doctor
101.          Patient *p1 = new Patient("Anton");
102.          Patient *p2 = new Patient("Ivan");
103.          Patient *p3 = new Patient("Derek");
```

```
104.
105.          Doctor *d1 = new Doctor("John");
106.          Doctor *d2 = new Doctor("Tom");
107.
108.          d1->addPatient(p1);
109.
110.          d2->addPatient(p1);
111.          d2->addPatient(p3);
112.
113.          std::cout << *d1 << '\n';
114.          std::cout << *d2 << '\n';
115.          std::cout << *p1 << '\n';
116.          std::cout << *p2 << '\n';
117.          std::cout << *p3 << '\n';
118.
119.          delete p1;
120.          delete p2;
121.          delete p3;
122.
123.          delete d1;
124.          delete d2;
125.
126.          return 0;
127.      }
```

Результат выполнения программы:

```
John is seeing patients: Anton
Tom is seeing patients: Anton Derek
Anton is seeing doctors: John Tom
Ivan has no doctors right now
Derek is seeing doctors: Tom
```

Если говорить в общем, то лучше избегать двунаправленных ассоциаций, если для решения задания подходит и однонаправленная связь, так как двунаправленную связь написать сложнее (с учетом возникновения возможных ошибок) и она усложняет логику программы.

Рефлексивная ассоциация

Иногда объекты могут иметь отношения с другими объектами того же типа. Это называется **рефлексивной ассоциацией**. Хорошим примером рефлексивной ассоциации являются отношения между университетским курсом и его минимальными требованиями для студентов.

Рассмотрим упрощенный случай, когда Курс может иметь только одно Требование:

```
1. #include <string>
2.
3. class Course
4. {
5. private:
```

```
    std::string m_name;
    Course *m_condition;

public:
    Course(std::string &name, Course *condition=nullptr):
        m_name(name), m_condition(condition)
    {
    }

};
```

Это может привести к цепочке ассоциаций (курс имеет необходимое условие, выполнение которого включает еще одно условие и т.д.).

Ассоциации могут быть косвенными

В примерах, приведенных выше, мы использовали указатели для связывания объектов. Однако в ассоциации это не является обязательным условием. Можно использовать любые данные, которые позволяют связать два объекта. В следующем примере мы покажем, как класс Водитель может иметь однонаправленную связь с классом Автомобиль без переменной-члена в виде указателя на объект класса Автомобиль:

```
#include <iostream>
#include <string>

class Car
{
private:
    std::string m_name;
    int m_id;

public:
    Car(std::string name, int id)
        : m_name(name), m_id(id)
    {
    }

    std::string getName() { return m_name; }
    int getId() { return m_id;  }
};

// Наш CarLot, по сути, является статическим массивом, содержащим Автомобили, и имеет функцию для "выдачи" Автомобилей.
// Поскольку массив является статическим, то нам не нужно создавать объекты для использования класса CarLot
class CarLot
{
private:
    static Car s_carLot[4];

public:
    CarLot() = delete;

    static Car* getCar(int id)
    {
```

```
32.         for (int count = 0; count < 4; ++count)
33.             if (s_carLot[count].getId() == id)
34.                 return &(s_carLot[count]);
35.
36.         return nullptr;
37.     }
38. };
39.
40. Car CarLot::s_carLot[4] = { Car("Camry", 5), Car("Focus", 14), Car("Vito", 73),
     Car("Levante", 58) };
41.
42. class Driver
43. {
44. private:
45.     std::string m_name;
46.     int m_carId; // для связывания классов, вместо указателя, используется
    Идентификатор (целочисленное значение)
47.
48. public:
49.     Driver(std::string name, int carId)
50.         : m_name(name), m_carId(carId)
51.     {
52.     }
53.
54.     std::string getName() { return m_name; }
55.     int getCarId() { return m_carId; }
56.
57. };
58.
59. int main()
60. {
61.     Driver d("Ivan", 14); // Ivan использует машину с ID 14
62.
63.     Car *car = CarLot::getCar(d.getCarId()); // получаем этот Автомобиль из
    CarLot
64.
65.     if (car)
66.         std::cout << d.getName() << " is driving a " << car-
    >getName() << '\n';
67.     else
68.         std::cout << d.getName() << " couldn't find his car\n";
69.
70.     return 0;
71. }
```

Результат выполнения программы:

```
Ivan is driving a Focus
```

В примере, приведенном выше, у нас есть CarLot (Гараж) в котором находятся наши автомобили. Водитель, которому нужен Автомобиль, не имеет указателя на этот Автомобиль — вместо этого у него есть Идентификатор Автомобиля, который он может использовать для получения Автомобиля из Гаража, когда ему это нужно.

Конкретно в этом примере реализация выглядит несколько глупо, так как получение Автомобиля из Гаража требует дополнительного выполнения процессов (было бы

быстрее, если бы существовал указатель, соединяющий напрямую два класса). Тем не менее, есть и преимущества привязки объектов к Идентификаторам вместо использования указателя. Например, вы можете ссылаться на объекты, которые сейчас не находятся в памяти (возможно, они находятся в файле или в базе данных и могут быть загружены по запросу).

Композиция vs. Агрегация vs. Ассоциация

Вот таблица, которая поможет вам быстро разобраться/вспомнить различия между композицией, агрегацией и ассоциацией:

Свойства	Композиция	Агрегация	Ассоциация
Отношения	Части-целое	Части-целое	Объекты не связаны между собой
Члены могут принадлежать одновременно сразу нескольким классам	Нет	Да	Да
Существование членов управляется классами	Да	Нет	Нет
Вид отношений	Однонаправленные	Однонаправленные	Однонаправленные или Двунаправленные
Тип отношений	«Часть чего-то»	«Имеет»	«Использует»

Урок №158. Зависимость

Мы уже успели рассмотреть 3 типа отношений в языке C++: композицию, агрегацию и ассоциацию. Самый простой тип отношений — зависимость, мы приберегли напоследок.

В повседневной жизни мы используем термин «зависимость», чтобы указать, что один объект зависит от второго объекта для выполнения определенного задания. Например, если вы сломаете ногу, то будете зависеть от костылей, чтобы иметь возможность передвигаться (но не наоборот). Цветковые растения зависят от пчёл, которые опыляют их, чтобы те имели возможность размножаться (но не наоборот).

Зависимость возникает, когда один объект обращается к функционалу другого объекта для выполнения определенного задания. Эти отношения слабее отношений в ассоциации, но все же любое изменение объекта, который предоставляет свой функционал зависимому объекту, может стать причиной сбоя работы зависящего объекта. Зависимость всегда является однонаправленной.

Хорошим примером зависимости, которую вы уже видели много раз, являются отношения классов и std::cout (типа std::ostream). Классы используют std::cout для вывода чего-либо в консоль, но не наоборот. Например:

```
1. #include <iostream>
2.
3. class Point
4. {
5. private:
6.     double m_x, m_y, m_z;
7.
8. public:
9.     Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
10.    {
11.    }
12.
13.    friend std::ostream& operator<< (std::ostream &out, const Point &point);
14. };
15.
16. std::ostream& operator<< (std::ostream &out, const Point &point)
17. {
18.    // Поскольку функция перегрузки operator<< является дружественной классу
    Point, то мы имеем прямой доступ к закрытым членам класса Point
19.    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z <<
    ")";
20.
21.    return out;
22. }
23.
24. int main()
25. {
26.    Point point1(5.0, 6.0, 7.0);
```

```
27.
28.     std::cout << point1;
29.
30.     return 0;
31. }
```

Результат выполнения программы:

```
Point(5, 6, 7)
```

Здесь класс Point не имеет прямого отношения к std::cout, но имеет зависимость от std::cout, так как функция перегрузки оператора `<<` использует std::cout для вывода объектов класса Point в консоль.

Зависимость vs. Ассоциация

Очень часто возникает путаница: "А чем зависимость отличается от ассоциации?".

В языке C++ ассоциация — это отношения между двумя классами на уровне классов. То есть, первый класс сохраняет прямую или косвенную связь со вторым классом через переменную-член. Например, в классе Врач есть массив указателей на объекты класса Пациент в виде переменной-члена. Вы всегда можете спросить у Врача, кто его Пациенты. Класс Водитель содержит Идентификатор Автомобиля в виде целочисленной переменной-члена. Водитель всегда знает к чему привязан Автомобиль, и как получить к нему доступ.

Зависимости обычно не представлены на уровне классов, то есть зависимый объект не связан со вторым объектом через переменную-член. Зависимый объект обычно создается при необходимости (например, открытие файла для записи данных) или передается в функцию в качестве параметра (например, как std::ostream, участвующий в перегрузке оператора `<<`, в приведенной выше программе).

Урок №159. Контейнерные классы

В реальной жизни мы постоянно используем контейнеры. Гречка с куриной грудкой в контейнере для еды, страницы в книге с обложкой и переплетом, вещи в тумбочке/рюкзаке и т.д. Без этих контейнеров было бы крайне неудобно работать с объектами, находящимися внутри. Представьте, что вы пытаетесь читать книгу без переплета и обложки, или пытаетесь есть гречку с грудкой, не используя контейнер для еды/миску/тарелку и т.д. Непорядок! Ценность контейнеров заключается в том, что они помогают должным образом организовать и хранить объекты.

Контейнерные классы

Контейнерный класс (или ***"класс-контейнер"***) в языке C++ - это класс, предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных). Существует много разных контейнерных классов, каждый из которых имеет свои преимущества, недостатки или ограничения в использовании. Безусловно, наиболее часто используемым контейнером в программировании является массив, который мы уже использовали во многих примерах. Хотя в языке C++ есть стандартные обычные массивы, большинство программистов используют контейнерные классы-массивы: std::array или std::vector из-за преимуществ, которые они предоставляют. В отличие от стандартных массивов, контейнерные классы-массивы имеют возможность динамического изменения своего размера, когда элементы добавляются или удаляются. Это не только делает их более удобными, чем обычные массивы, но и безопаснее.

Обычно, **функционал классов-контейнеров** языка C++ следующий:

- Создание пустого контейнера (через конструктор).
- Добавление нового объекта в контейнер.
- Удаление объекта из контейнера.
- Просмотр количества объектов, находящихся на данный момент в контейнере.
- Очистка контейнера от всех объектов.
- Доступ к сохраненным объектам.
- Сортировка объектов/элементов (не всегда).

Иногда функционал контейнерных классов может быть не столь обширным, как это указано выше. Например, контейнерные классы-массивы часто не имеют

функционала добавления/удаления объектов, так как они и так медленные, и разработчик просто не хочет увеличивать нагрузку.

Типом отношений в классах-контейнерах является ***«член чего-то»***. Например, элементы массива «являются членами» массива (принадлежат ему). Обратите внимание, мы здесь используем термин «член чего-то» не в смысле члена класса C++.

Типы контейнерных классов

Контейнерные классы обычно бывают двух типов:

- **Контейнеры значения** – это композиции, которые хранят копии объектов (и, следовательно, ответственные за создание/уничтожение этих копий).
- **Контейнеры ссылки** — это агрегации, которые хранят указатели или ссылки на другие объекты (и, следовательно, не ответственные за создание/уничтожение этих объектов).

В отличие от реальной жизни, когда контейнеры могут хранить любые типы объектов, которые в них помещают, в языке C++ контейнеры обычно содержат только один тип данных. Например, если у вас целочисленный массив, то он может содержать только целочисленные значения. В отличие от некоторых других языков, C++ не позволяет смешивать разные типы данных внутри одного контейнера. Если вам нужны контейнеры для хранения значений типов int и double, то вам придется написать два отдельных контейнера (или использовать шаблоны, о которых мы поговорим на соответствующем уроке). Несмотря на ограничения их использования, контейнеры чрезвычайно полезны, так как делают программирование проще, безопаснее и быстрее.

Контейнерный класс-массив

Сейчас мы напишем целочисленный класс-массив с нуля, реализуя функционал контейнеров в языке C++. Этот класс-массив будет типа контейнера значения, в котором будут храниться копии элементов, а не сами элементы.

Сначала создадим файл ArrayInt.h:

```
1. #ifndef ARRAYINT_H
2. #define ARRAYINT_H
3.
4. class ArrayInt
5. {
6. };
7.
```

```
8. #endif
```

Наш ArrayInt должен отслеживать два значения: данные и свою длину. Поскольку мы хотим, чтобы наш массив мог изменять свою длину, то нам нужно использовать динамическое выделение памяти, что означает, что мы будем использовать указатель для хранения данных:

```
1. #ifndef ARRAYINT_H
2. #define ARRAYINT_H
3.
4. class ArrayInt
5. {
6. private:
7.     int m_length;
8.     int *m_data;
9. };
10.
11.#endif
```

Теперь нам нужно добавить конструкторы, чтобы иметь возможность создавать объекты класса ArrayInt. Мы добавим два конструктора: первый будет создавать пустой массив, второй - массив заданного размера:

```
1. #ifndef ARRAYINT_H
2. #define ARRAYINT_H
3.
4. #include <cassert> // для assert()
5.
6. class ArrayInt
7. {
8. private:
9.     int m_length;
10.    int *m_data;
11.
12.public:
13.    ArrayInt():
14.        m_length(0), m_data(nullptr)
15.    {
16.    }
17.
18.    ArrayInt(int length):
19.        m_length(length)
20.    {
21.        assert(length >= 0);
22.
23.        if (length > 0)
24.            m_data = new int[length];
25.        else
26.            m_data = nullptr;
27.    }
28.};
29.
30.#endif
```

Нам также потребуются функции, которые будут выполнять очистку ArrayInt. Во-первых, добавим деструктор, который будет просто освобождать любую

динамически выделенную память. Во-вторых, напишем функцию erase(), которая будет выполнять очистку массива и сбрасывать его длину на `0`:

```
~ArrayInt()
{
    delete[] m_data;
    // Здесь нам не нужно присваивать значение null для m_data или выполнять m_length = 0, так как объект и так будет уничтожен
}

void erase()
{
    delete[] m_data;

    // Здесь нам нужно указать m_data значение nullptr, чтобы на выходе не было висячего указателя
    m_data = nullptr;
    m_length = 0;
}
```

Теперь перегрузим оператор индексации [], чтобы иметь доступ к элементам массива. Мы также должны выполнить проверку корректности передаваемого индекса, что лучше всего сделать с помощью стейтмента assert. Также добавим функцию доступа для возврата длины массива:

```
#ifndef ARRAYINT_H
#define ARRAYINT_H

#include <cassert> // для assert()

class ArrayInt
{
private:
    int m_length;
    int *m_data;

public:
    ArrayInt():
        m_length(0), m_data(nullptr)
    {
    }

    ArrayInt(int length):
        m_length(length)
    {
        assert(length >= 0);

        if (length > 0)
            m_data = new int[length];
        else
            m_data = nullptr;
    }

    ~ArrayInt()
    {
        delete[] m_data;
    }

```

```
34.     void erase()
35.     {
36.         delete[] m_data;
37.         // Указываем m_data значение nullptr, чтобы на выходе не было
    висячего указателя
38.         m_data = nullptr;
39.         m_length = 0;
40.     }
41.
42.     int& operator[](int index)
43.     {
44.         assert(index >= 0 && index < m_length);
45.         return m_data[index];
46.     }
47.
48.     int getLength() { return m_length; }
49. };
50.
51. #endif
```

Теперь у нас уже есть класс ArrayInt, который мы можем использовать. Мы можем выделить массив определенного размера и использовать оператор `[]` для извлечения или изменения значений элементов.

Тем не менее, есть еще несколько вещей, которые мы не можем выполнить с нашим ArrayInt. Это автоматическое изменение размера массива, добавление/удаление элементов и сортировка элементов.

Во-первых, давайте реализуем возможность массива изменять свой размер. Мы напишем две разные функции для этого. Первая функция — reallocate(), при изменении размера массива будет уничтожать все существующие элементы (это быстро). Вторая функция — resize(), при изменении размера массива будет сохранять все существующие элементы (это медленно).

```
1.  // Функция reallocate() изменяет размер массива. Все существующие элементы
    внутри массива будут уничтожены. Процесс быстрый
2.  void reallocate(int newLength)
3.  {
4.      // Удаляем все существующие элементы внутри массива
5.      erase();
6.
7.      // Если наш массив должен быть пустым, то выполняем возврат здесь
8.      if (newLength <= 0)
9.          return;
10.
11.     // Дальше нам нужно выделить новые элементы
12.     m_data = new int[newLength];
13.     m_length = newLength;
14. }
15.
16. // Функция resize() изменяет размер массива. Все существующие элементы
    сохраняются. Процесс медленный
17. void resize(int newLength)
18. {
19.     // Если массив уже нужной длины, то выполняем return
```

```
20.     if (newLength == m_length)
21.         return;
22.
23.     // Если нужно сделать массив пустым, то делаем это и затем выполняем return
24.     if (newLength <= 0)
25.     {
26.         erase();
27.         return;
28.     }
29.
30.     // Теперь предположим, что newLength состоит, по крайней мере, из одного элемента. Выполняется следующий алгоритм действий:
31.     // 1. Выделяем новый массив.
32.     // 2. Копируем элементы из существующего массива в наш только что выделенный массив.
33.     // 3. Уничтожаем старый массив и даем команду m_data указывать на новый массив.
34.
35.     // Выделяем новый массив
36.     int *data = new int[newLength];
37.
38.     // Затем нам нужно разобраться с количеством копируемых элементов в новый массив.
39.     // Нам нужно скопировать столько элементов, сколько их есть в меньшем из массивов
40.     if (m_length > 0)
41.     {
42.         int elementsToCopy = (newLength > m_length) ? m_length : newLength;
43.
44.         // Поочередно копируем элементы
45.         for (int index=0; index < elementsToCopy ; ++index)
46.             data[index] = m_data[index];
47.     }
48.
49.     // Удаляем старый массив, так как он нам уже не нужен
50.     delete[] m_data;
51.
52.     // И используем вместо старого массива новый! Обратите внимание, m_data указывает на тот же адрес, на который указывает наш новый динамически выделенный массив.
53.     // Поскольку данные были динамически выделены, то они не будут уничтожены, когда выйдут из области видимости
54.     m_data = data;
55.     m_length = newLength;
56. }
```

Фух! Было непросто!

Функционал большинства контейнерных классов-массивов на этом заканчивается. Однако, если вы хотите увидеть, как реализовать возможность добавления/удаления элементов, то мы вам сейчас это покажем. Следующие два алгоритма очень похожи на функцию resize():

```
1. void insertBefore(int value, int index)
2. {
3.     // Проверка корректности передаваемого индекса
4.     assert(index >= 0 && index <= m_length);
5.
```

```
6.      // Создаем новый массив на один элемент больше старого массива
7.      int *data = new int[m_length+1];
8.
9.      // Копируем все элементы аж до index-а
10.     for (int before=0; before < index; ++before)
11.         data[before] = m_data[before];
12.
13.     // Вставляем наш новый элемент в наш новый массив
14.     data [index] = value;
15.
16.     // Копируем все значения после вставляемого элемента
17.     for (int after=index; after < m_length; ++after)
18.         data[after+1] = m_data[after];
19.
20.     // Удаляем старый массив и используем вместо него новый массив
21.     delete[] m_data;
22.     m_data = data;
23.     ++m_length;
24. }
25.
26. void remove(int index)
27. {
28.     // Проверка на корректность передаваемого индекса
29.     assert(index >= 0 && index < m_length);
30.
31.     // Если это последний элемент массива, то делаем массив пустым и выполняем
    return
32.     if (m_length == 1)
33.     {
34.         erase();
35.         return;
36.     }
37.
38.     // Создаем новый массив на один элемент меньше нашего старого массива
39.     int *data = new int[m_length-1];
40.
41.     // Копируем все элементы аж до index-а
42.     for (int before=0; before  < index; ++before)
43.         data[before] = m_data[before];
44.
45.     // Копируем все значения после удаляемого элемента
46.     for (int after=index+1; after < m_length; ++after )
47.         data[after-1] = m_data[after];
48.
49.     // Удаляем старый массив и используем вместо него новый массив
50.     delete[] m_data;
51.     m_data = data;
52.     --m_length;
53. }
54.
55. // Несколько дополнительных функций просто для удобства
56. void insertAtBeginning(int value) { insertBefore(value, 0); }
57. void insertAtEnd(int value) { insertBefore(value, m_length); }
```

Вот наш контейнерный класс-массив ArrayInt целиком.

ArrayInt.h:

```
1.  #ifndef ARRAYINT_H
2.  #define ARRAYINT_H
```

```
3.
4.  #include <cassert> // для assert()
5.
6.  class ArrayInt
7.  {
8.  private:
9.      int m_length;
10.     int *m_data;
11.
12. public:
13.     ArrayInt():
14.         m_length(0), m_data(nullptr)
15.     {
16.     }
17.
18.     ArrayInt(int length):
19.         m_length(length)
20.     {
21.         assert(length >= 0);
22.         if (length > 0)
23.             m_data = new int[length];
24.         else
25.             m_data = nullptr;
26.     }
27.
28.     ~ArrayInt()
29.     {
30.         delete[] m_data;
31.     }
32.
33.     void erase()
34.     {
35.         delete[] m_data;
36.         // Здесь нужно указать m_data значение nullptr, чтобы на выходе не было
    висячего указателя
37.         m_data = nullptr;
38.         m_length = 0;
39.     }
40.
41.     int& operator[](int index)
42.     {
43.         assert(index >= 0 && index < m_length);
44.         return m_data[index];
45.     }
46.
47.     // Функция reallocate() изменяет размер массива. Все существующие элементы
   внутри массива будут уничтожены. Процесс быстрый
48.     void reallocate(int newLength)
49.     {
50.         // Удаляем все существующие элементы внутри массива
51.         erase();
52.
53.         // Если наш массив должен быть пустым, то выполняем возврат здесь
54.         if (newLength <= 0)
55.             return;
56.
57.         // Затем выделяем новые элементы
58.         m_data = new int[newLength];
59.         m_length = newLength;
60.     }
61.
```

```
62.     // Функция resize() изменяет размер массива. Все существующие элементы сохраняются. Процесс медленный
63.     void resize(int newLength)
64.     {
65.         // Если массив нужной длины, то выполняем return
66.         if (newLength == m_length)
67.             return;
68.
69.         // Если нужно сделать массив пустым, то делаем это и затем выполняем return
70.         if (newLength <= 0)
71.         {
72.             erase();
73.             return;
74.         }
75.
76.         // Теперь предположим, что newLength состоит, по крайней мере, из одного элемента. Выполняется следующий алгоритм действий:
77.         // 1. Выделяем новый массив.
78.         // 2. Копируем элементы из существующего массива в наш только что выделенный массив.
79.         // 3. Уничтожаем старый массив и даем команду m_data указывать на новый массив.
80.
81.         // Выделяем новый массив
82.         int *data = new int[newLength];
83.
84.         // Затем нам нужно разобраться с количеством копируемых элементов в новый массив.
85.         // Нам нужно скопировать столько элементов, сколько их есть в меньшем из массивов
86.         if (m_length > 0)
87.         {
88.             int elementsToCopy = (newLength > m_length) ? m_length : newLength;
89.
90.             // Поочередно копируем элементы
91.             for (int index=0; index < elementsToCopy ; ++index)
92.                 data[index] = m_data[index];
93.         }
94.
95.         // Удаляем старый массив, так как он нам уже не нужен
96.         delete[] m_data;
97.
98.         // И используем вместо старого массива новый! Обратите внимание, m_data указывает на тот же адрес, на который указывает наш новый динамически выделенный массив.
99.         // Поскольку данные были динамически выделены, то они не будут уничтожены, когда выйдут из области видимости
100.            m_data = data;
101.            m_length = newLength;
102.        }
103.
104.        void insertBefore(int value, int index)
105.        {
106.            // Проверка корректности передаваемого индекса
107.            assert(index >= 0 && index <= m_length);
108.
109.            // Создаем новый массив на один элемент больше старого массива
110.            int *data = new int[m_length+1];
111.
112.            // Копируем все элементы аж до index-а
```

```
113.                 for (int before=0; before < index; ++before)
114.                     data [before] = m_data[before];
115.
116.                 // Вставляем наш новый элемент в наш новый массив
117.                 data [index] = value;
118.
119.                 // Копируем все значения после вставляемого элемента
120.                 for (int after=index; after < m_length; ++after)
121.                     data[after+1] = m_data[after];
122.
123.                 // Удаляем старый массив и используем вместо него новый массив
124.                 delete[] m_data;
125.                 m_data = data;
126.                 ++m_length;
127.             }
128.
129.             void remove(int index)
130.             {
131.                 // Проверка на корректность передаваемого индекса
132.                 assert(index >= 0 && index < m_length);
133.
134.                 // Если это последний элемент массива, то делаем массив пустым и
    выполняем return
135.                 if (m_length == 1)
136.                 {
137.                     erase();
138.                     return;
139.                 }
140.
141.                 // Создаем новый массив на один элемент меньше нашего старого
    массива
142.                 int *data = new int[m_length-1];
143.
144.                 // Копируем все элементы аж до index-а
145.                 for (int before=0; before  < index; ++before)
146.                     data[before] = m_data[before];
147.
148.                 // Копируем все значения после удаляемого элемента
149.                 for (int after=index+1; after < m_length; ++after )
150.                     data[after-1] = m_data[after];
151.
152.                 // Удаляем старый массив и используем вместо него новый массив
153.                 delete[] m_data;
154.                 m_data = data;
155.                 --m_length;
156.             }
157.
158.             // Несколько дополнительных функций просто для удобства
159.             void insertAtBeginning(int value) { insertBefore(value, 0); }
160.             void insertAtEnd(int value) { insertBefore(value, m_length); }
161.
162.             int getLength() { return m_length; }
163.         };
164.
165.         #endif
```

Теперь протестируем программу:

```
1.  #include <iostream>
2.  #include "ArrayInt.h"
3.
```

```
4.  int main()
5.  {
6.      // Объявляем массив с 10 элементами
7.      ArrayInt array(10);
8.
9.      // Заполняем массив числами от 1 до 10
10.     for (int i=0; i<10; i++)
11.         array[i] = i+1;
12.
13.     // Изменяем размер массива до 7 элементов
14.     array.resize(7);
15.
16.     // Вставляем число 15 перед элементом с индексом 4
17.     array.insertBefore(15, 4);
18.
19.     // Удаляем элемент с индексом 5
20.     array.remove(5);
21.
22.     // Добавляем числа 35 и 50 в конец и в начало
23.     array.insertAtEnd(35);
24.     array.insertAtBeginning(50);
25.
26.     // Выводим все элементы массива
27.     for (int j=0; j<array.getLength(); j++)
28.         std::cout << array[j] << " ";
29.
30.     return 0;
31. }
```

Результат:

```
50 1 2 3 4 15 6 7 35
```

Хотя написание контейнерных классов может быть несколько сложным, но хорошая новость заключается в том, что вам их нужно написать только один раз. Как только контейнерный класс работает, вы можете его повторно использовать где-угодно без каких-либо дополнительных действий/усилий по части программирования.

Также стоит отметить, что, хотя наш контейнерный класс ArrayInt содержит фундаментальный тип данных (int), мы также могли бы легко использовать и пользовательский тип данных.

> ***Примечание***: Если класс из Стандартной библиотеки C++ полностью соответствует вашим потребностям, то используйте его вместо написания своего контейнерного класса. Например, вместо ArrayInt лучше использовать std::vector<int>, так как реализация std::vector<int> протестирована/проверена уже многими годами, эффективна и отлично работает с другими классами из Стандартной библиотеки C++. Но так как не всегда может быть возможным использовать классы из Стандартной библиотеки C++, то вы уже знаете, как создавать свои собственные контейнерные классы.

Урок №160. Список инициализации std::initializer_list

Рассмотрим фиксированный массив целых чисел в языке C++:

```
int array[7];
```

Для инициализации этого массива мы можем использовать список инициализации:

```
#include <iostream>

int main()
{
    int array[7] { 7, 6, 5, 4, 3, 2, 1 }; // список инициализации
    for (int count=0; count < 7; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

Результат:

```
7 6 5 4 3 2 1
```

Это также работает и с динамически выделенными массивами:

```
#include <iostream>

int main()
{
    int *array = new int[7] { 7, 6, 5, 4, 3, 2, 1 }; // список инициализации
    for (int count = 0; count < 7; ++count)
        std::cout << array[count] << ' ';
    delete[] array;

    return 0;
}
```

На предыдущем уроке мы рассматривали контейнерные классы на примере класса-массива целых чисел ArrayInt:

```
#include <cassert> // для assert()

class ArrayInt
{
private:
    int m_length;
    int *m_data;

public:
    ArrayInt():
        m_length(0), m_data(nullptr)
    {
    }

```

```
    ArrayInt(int length):
        m_length(length)
    {
        m_data = new int[length];
    }

    ~ArrayInt()
    {
        delete[] m_data;
        // Нам не нужно здесь присваивать значение null для m_data или выполнять m_length = 0, так как объект будет уничтожен сразу же после выполнения этой функции
    }

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() { return m_length; }
};
```

Что произойдет, если мы попытаемся использовать список инициализации с этим контейнерным классом?

```
int main()
{
    ArrayInt array { 7, 6, 5, 4, 3, 2, 1 }; // эта строка вызовет ошибку компиляции
    for (int count=0; count < 7; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

Этот код не скомпилируется, так как класс ArrayInt не имеет конструктора, который бы знал, что делать со списком инициализации, поэтому каждый элемент нужно инициализировать в индивидуальном порядке:

```
int main()
{
    ArrayInt array(7);
    array[0] = 7;
    array[1] = 6;
    array[2] = 5;
    array[3] = 4;
    array[4] = 3;
    array[5] = 2;
    array[6] = 1;

    for (int count=0; count < 7; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

Как-то не очень, правда?

До C++11 списки инициализации могли использоваться только со статическими или динамически выделенными массивами. Однако в C++11 появилось решение этой проблемы.

Инициализация классов через std::initializer_list

Когда компилятор C++11 видит список инициализации, то он автоматически конвертирует его в объект типа std::initializer_list. Поэтому, если мы создадим конструктор, который принимает в качестве параметра std::initializer_list, мы сможем создавать объекты, используя список инициализации в качестве входных данных.

std::initializer_list находится в заголовочном файле initializer_list.

Есть несколько вещей, которые нужно знать о std::initializer_list. Так же, как и с std::array и std::vector, вы должны указать в угловых скобках std::initializer_list какой тип данных будет использоваться. По этой причине вы никогда не увидите пустой std::initializer_list. Вместо этого вы увидите что-то вроде `std::initializer_list<int>` или `std::initializer_list<std::string>`.

Во-вторых, std::initializer_list имеет **функцию size()**, которая возвращает количество элементов списка. Это полезно, когда нам нужно знать длину получаемого списка.

Обновим наш класс-массив ArrayInt, добавив конструктор, который принимает std::initializer_list:

```
1. #include <iostream>
2. #include <cassert> // для assert()
3. #include <initializer_list> // для std::initializer_list
4.
5. class ArrayInt
6. {
7. private:
8.     int m_length;
9.     int *m_data;
10.
11. public:
12.     ArrayInt() :
13.         m_length(0), m_data(nullptr)
14.     {
15.     }
16.
17.     ArrayInt(int length) :
18.         m_length(length)
19.     {
20.         m_data = new int[length];
21.     }
22.
```

```
23.     ArrayInt(const std::initializer_list<int> &list): // позволяем инициализацию ArrayInt через список инициализации
24.         ArrayInt(list.size()) // используем концепцию делегирования конструкторов для создания начального массива, в который будет выполняться копирование элементов
25.     {
26.         // Инициализация нашего начального массива значениями из списка инициализации
27.         int count = 0;
28.         for (auto &element : list)
29.         {
30.             m_data[count] = element;
31.             ++count;
32.         }
33.     }
34.
35.     ~ArrayInt()
36.     {
37.         delete[] m_data;
38.         // Нам не нужно здесь присваивать значение null для m_data или выполнять m_length = 0, так как объект будет уничтожен сразу же после выполнения этой функции
39.     }
40.
41.     int& operator[](int index)
42.     {
43.         assert(index >= 0 && index < m_length);
44.         return m_data[index];
45.     }
46.
47.     int getLength() { return m_length; }
48. };
49.
50. int main()
51. {
52.     ArrayInt array { 7, 6, 5, 4, 3, 2, 1 }; // список инициализации
53.     for (int count = 0; count < array.getLength(); ++count)
54.         std::cout << array[count] << ' ';
55.
56.     return 0;
57. }
```

Результат выполнения программы:

```
7 6 5 4 3 2 1
```

Работает! Теперь рассмотрим это всё подробнее.

Вот наш конструктор, который принимает std::initializer_list<int>:

```
1. ArrayInt(const std::initializer_list<int> &list): // позволяем инициализацию ArrayInt через список инициализации
2.     ArrayInt(list.size()) // используем концепцию делегирования конструкторов для создания начального массива, в который будет выполняться копирование элементов
3. {
4.     // Инициализируем наш начальный массив значениями из списка инициализации
5.     int count = 0;
6.     for (auto &element : list)
```

```
7.      {
8.          m_data[count] = element;
9.          ++count;
10.     }
11. }
```

Строка №1: Как мы уже говорили, обязательно нужно указывать используемый тип данных в угловых скобках std::initializer_list. В этом случае, поскольку это ArrayInt, то ожидается, что список будет заполнен значениями типа int. Обратите внимание, мы передаем список по константной ссылке, дабы избежать его копирования при передаче в конструктор.

Строка №2: Мы делегируем выделение памяти для начального объекта ArrayInt, в который будем выполнять копирование элементов, другому конструктору, используя концепцию делегирующих конструкторов, чтобы сократить лишний код. Этот другой конструктор должен знать длину выделяемого объекта, поэтому мы передаем ему `list.size()`, который указывает на количество элементов списка.

В теле нашего конструктора мы выполняем копирование элементов из списка инициализации в класс ArrayInt. По каким-то необъяснимым причинам std::initializer_list не предоставляет доступ к своим элементам через оператор индексации `[]`. Об этом много говорили, но официального решения так и не предоставили.

Тем не менее, есть способы, чтобы это обойти. Самый простой — использовать цикл foreach. Цикл foreach перебирает каждый элемент списка, и мы, таким образом, копируем каждый элемент в наш внутренний массив.

Присваивание значений и std::initializer_list

Вы также можете использовать std::initializer_list для присваивания новых значений классу, перегруззив оператор присваивания для получения std::initializer_list в качестве параметра. Пример того, как это делается, будет в тестовом задании ниже.

Обратите внимание, если вы создаете конструктор, который принимает std::initializer_list, то вы должны проследить, чтобы хоть одно из следующих действий было выполнено:

- Перегрузка оператора присваивания.
- Корректное глубокое копирование для оператора присваивания.
- Наличие явного конструктора (с ключевым словом explicit), чтобы он не мог использоваться для неявных преобразований.

Почему? Рассмотрим вышеприведенный класс (который не имеет перегрузки оператора присваивания или копирующего присваивания) со следующим стейтментом:

```
1. array = { 1, 3, 5, 7, 9, 11 }; // перезаписываем значения array значениями из списка инициализации
```

Во-первых, компилятор видит, что функции присваивания, которая принимает std::initializer_list в качестве параметра, не существует. Затем он ищет другие функции, которые он мог бы использовать, и находит неявно предоставленный копирующий оператор присваивания. Однако эта функция может использоваться, только если она сможет преобразовать список инициализации в ArrayInt, а поскольку у нас есть конструктор, который принимает std::initializer_list, и он не помечен как explicit, то компилятор будет использовать этот конструктор для преобразования списка инициализации во временный ArrayInt. Затем вызовется неявный оператор присваивания, который используется в конструкторе и который будет выполнять поверхностное копирование временного объекта ArrayInt в наш объект `array`.

И тогда `m_data` временного объекта ArrayInt, и `m_data` объекта `array` будут указывать на один и тот же адрес (из-за поверхностного копирования). Вы уже можете догадаться, к чему это приведет.

В конце стейтмента присваивания временный ArrayInt уничтожается. Вызывается деструктор, который удаляет временный `m_data` класса ArrayInt. Это оставляет наш объект `array` с висячим указателем `m_data`. Когда мы попытаемся использовать `m_data` объекта `array` для любых целей (в том числе, когда массив будет выходить из области видимости и деструктору нужно будет уничтожить `m_data`), то мы получим неопределенные результаты (или сбой).

Заключение

Реализация конструктора, который принимает std::initializer_list в качестве параметра (используется передача по ссылке для предотвращения копирования), позволяет нам использовать список инициализации с нашими пользовательскими классами. Мы также можем использовать std::initializer_list для реализации других функций, которым необходим список инициализации (например, для перегрузки оператора присваивания).

Тест

Используя вышеприведенный класс ArrayInt, реализуйте перегрузку оператора присваивания, который будет принимать список инициализации. Следующий код:

```
int main()
{
    ArrayInt array { 7, 6, 5, 4, 3, 2, 1 }; // список инициализации
    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    std::cout << '\n';

    array = { 1, 4, 9, 12, 15, 17, 19, 21 };

    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

Должен выдавать следующий результат:

```
7 6 5 4 3 2 1
1 4 9 12 15 17 19 21
```

Глава №10. Итоговый тест

Теория

Процесс построения сложных объектов из более простых называется **композицией объекта**. Существует два подтипа композиции объектов: **композиция** и **агрегация**.

В **композиции** типом отношений между членом и классом является «часть чего-то» (член является "частью" класса). В композиции класс управляет существованием членов. Для реализации композиции части и объект должны соответствовать следующим правилам:

- Часть (член) является частью объекта (класса).
- Часть (член) может принадлежать только одному объекту (классу) в моменте.
- Существование части (члена) управляется объектом (классом).
- Часть (член) не знает о существовании объекта (класса).

Реализация композиции обычно происходит через переменные-члены или через указатели, когда класс управляет памятью самостоятельно (выделение/освобождение). Если вы *можете* реализовать композицию в своем классе, то вы *должны* реализовать композицию в своем классе.

В **агрегации** типом отношений между членом и классом является «имеет» (класс "имеет" члены). В отношениях агрегации класс не управляет существованием членов. Для реализации агрегации части и объект должны соответствовать следующим правилам:

- Часть (член) является частью объекта (класса).
- Часть (член) может принадлежать более чем одному объекту (классу) в моменте.
- Существование части (члена) не управляется объектом (классом).
- Часть (член) не знает о существовании объекта (класса).

Агрегации обычно реализуются с помощью указателей или ссылок.

В **ассоциации** объекты не связаны между собой, нет отношений типа «части-целое». Для реализации ассоциации объекты должны соответствовать следующим правилам:

- Первый объект (член) не связан со вторым объектом (классом).

- Первый объект (член) может принадлежать сразу нескольким объектам (классам) одновременно.
- Существование первого объекта (члена) не управляется вторым объектом (классом).
- Первый объект (член) может знать или не знать о существовании второго объекта (класса).

Ассоциации могут быть реализованы как через указатели, так и через ссылки.

В **зависимости** один класс использует второй класс для выполнения определенного задания. Используемый класс обычно не является членом главного класса, а временно создается, используется, а затем уничтожается или передается в метод главного класса извне.

Контейнерные классы — это классы, предназначенные для хранения и организации нескольких объектов определенного типа данных. **Контейнер-значение** - это композиция, в которой хранятся копии объектов контейнера. **Контейнер-ссылка** - это агрегация, в которой хранятся указатели или ссылки на объекты, которые находятся вне тела контейнера.

Список инициализации std::initializer_list может использоваться для реализации конструкторов, перегрузки оператора присваивания и других функций, которые принимают список инициализации в качестве параметра. std::initailizer_list находится в заголовочном файле initializer_list.

Свойства	Композиция	Агрегация	Ассоциация	Зависимость
Отношения	«Части-Целое»	«Части-Целое»	Объекты не связаны между собой	Объекты не связаны между собой
Члены могут принадлежать одновременно сразу нескольким классам?	Нет	Да	Да	Да

Класс управляет существованием членов?	Да	Нет	Нет	Да
Вид отношений	Однонаправленные	Однонаправленные	Однонаправленные/Двунаправленные	Однонаправленные
Тип отношений	«Часть чего-то»	«Имеет»	«Использует»	«Зависит от»

Тест

Эта глава проще и более абстрактная, чем предыдущие главы, поэтому этот тест будет лаконичным.

Задание №1

Какие типы отношений (композиция, агрегация, ассоциация или зависимость) описываются ниже?

a) Класс Животное, который содержит Тип Животного и его Имя.

b) Класс TextEditor с методом Save(), который принимает объект `file`. Метод Save() записывает содержимое редактора на диск.

c) Класс Авантюрист, который может хранить разные Предметы: мечи, копья, зелья или книги заклинаний. Эти Предметы могут быть отброшены и подняты другими Авантюристами.

d) Программист использует Компьютер для просмотра видео с котами в Интернете.

e) Класс Компьютер, который содержит класс Процессор. Процессор можно вынуть из Компьютера и посмотреть.

Задание №2

Если вы можете создать класс, используя композицию, агрегацию, ассоциацию или зависимость, то что вы выберите?

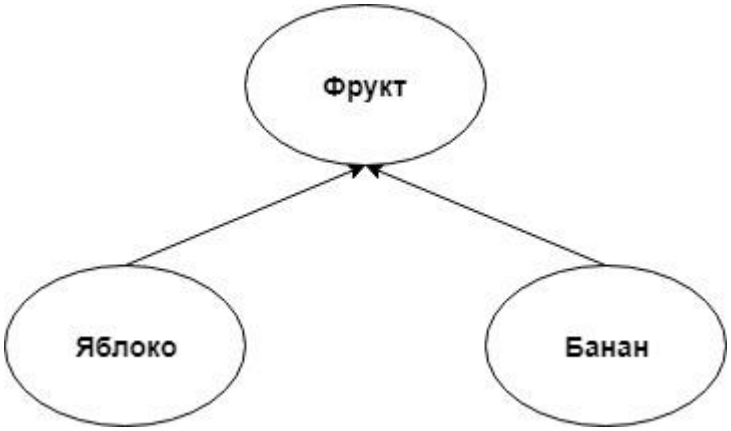
Урок №161. Введение в Наследование

На предыдущих уроках мы рассмотрели композицию объектов, когда сложные классы состоят из более простых классов и типов данных. Композиция объектов идеально подходит для создания новых объектов, типом отношений которых является «имеет». Однако композиция объектов является лишь одним из двух основных способов, с помощью которых вы можете создавать сложные классы в языке C++. Второй способ — это **наследование**, которое моделирует **тип отношения «является»** между двумя объектами.

В отличие от композиции объектов, которая включает в себя создание новых объектов путем объединения других объектов, наследование включает в себя создание новых объектов путем непосредственного сохранения свойств и поведения других объектов, а затем их расширения или наоборот — конкретизации. Подобно композиции объектов, наследование происходит повсюду в реальной жизни. Например, при рождении вы унаследовали гены от своих родителей, и вам передались определенные физические свойства от каждого из них (предрасположенность к болезням, видам деятельности и т.д.), но затем вы добавили свою личность ко всему приобретенному. Технологические продукты (компьютеры, смартфоны и т.д.) наследуют функционал от своих предшественников, при этом добавляя что-то свое (новое, уникальное) и сохраняя обратную совместимость. Например, процессор Intel Pentium унаследовал многие функциональные свойства от процессора Intel 486, который, в свою очередь, унаследовал свой функционал от более ранних процессоров. Язык C++ многое унаследовал от языка Си, на котором он основан, а язык Си унаследовал многие свойства от других языков программирования, которые были до него.

Рассмотрим пример с яблоками и бананами. Хотя яблоко и банан — это разные фрукты, но у них обоих есть одно общее свойство: они оба являются фруктами. И поскольку яблоки и бананы — это фрукты, то, следуя логике, всё, что верно для фруктов, верно и для яблок с бананами. Например, все фрукты имеют свое название, цвет и размер. Яблоки и бананы также имеют свои названия, цвет и размер. Мы можем сказать, что яблоки и бананы унаследовали (приобрели) все свойства фруктов, потому что они сами являются фруктами. Мы также знаем, что фрукты подвергаются процессу созревания, благодаря которому они становятся съедобными. Поскольку яблоки и бананы являются фруктами, то, соответственно, они также подвергаются процессу созревания, в результате чего становятся съедобными.

Если изобразить отношения между яблоками, бананами и фруктами на диаграмме, то это будет выглядеть примерно следующим образом:



Здесь мы видим иерархию.

Иерархии

Иерархия — это диаграмма со связями объектов. Большинство иерархий либо демонстрируют прогрессию с течением времени (`386 > 486 > Pentium`), либо классифицируют вещи таким образом, чтобы они переходили от общего к конкретному (`Фрукты > Яблоки > Макинтош`). Еще со школьной биологии царство, тип, класс, порядок, семейство, род и вид являются определением иерархии (движением от общего к конкретному или наоборот).

Вот еще один пример иерархии: квадрат является прямоугольником, который является четырехугольником, который является фигурой.

Правильный треугольник — это треугольник, который также является фигурой:

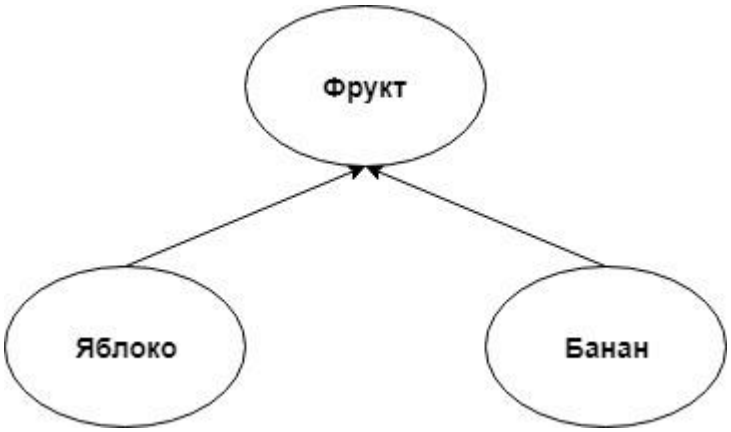
Здесь каждый элемент наследует свойства и поведение элемента над ним (движемся сверху вниз).

Что дальше?

На следующих уроках мы рассмотрим основы наследования и его реализацию в языке C++. Затем мы рассмотрим, как наследование позволяет использовать полиморфизм (одно из ключевых свойств объектно-ориентированного программирования) посредством виртуальных функций. Также поговорим о ключевых преимуществах наследования и о его недостатках.

Урок №162. Базовое наследование

Наследование в C++ происходит между классами и имеет тип отношений «является». Класс, от которого наследуют, называется **родительским** (или ***"базовым"***, ***"суперклассом"***), а класс, который наследует, называется **дочерним** (или ***"производным"***, ***"подклассом"***).



В диаграмме, представленной выше, `Фрукт` является родительским классом, а `Яблоко` и `Банан` — дочерними классами.

В этой диаграмме `Треугольник` является дочерним классом (родитель - `Фигура`) и родительским (для `Правильного треугольника`) одновременно.

Дочерний класс наследует как поведение (методы), так и свойства (переменные-члены) от родителя (с учетом некоторых ограничений доступа, которые мы рассмотрим чуть позже). Эти методы и переменные становятся членами дочернего класса.

Поскольку дочерние классы являются полноценными классами, то они могут (конечно) иметь и свои собственные члены. Сейчас мы это всё рассмотрим детально.

Класс Human

Вот простой класс Human для представления Человека:

```
#include <string>

class Human
{
public:
    std::string m_name;
    int m_age;

    Human(std::string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

};
```

В этом классе мы определили только те члены, которые являются общими для всех объектов этого класса. Каждый Человек (независимо от пола, профессии и т.д.) имеет Имя и Возраст.

Обратите внимание, в примере, приведенном выше, мы сделали все переменные-члены и методы класса открытыми. Это сделано ради простоты примера. Обычно переменные-члены нужно делать private. О средствах контроля доступа и о том, как это работает в наследовании, мы поговорим на соответствующих уроках.

Класс BasketballPlayer

Предположим, что нам нужно написать программу, которая будет отслеживать информацию о баскетболистах. Мы можем сохранять средний уровень игры баскетболиста и количество очков.

Вот наш незавершенный класс BasketballPlayer:

```
class BasketballPlayer
{
public:
    double m_gameAverage;
    int m_points;

    BasketballPlayer(double gameAverage = 0.0, int points = 0)
        : m_gameAverage(gameAverage), m_points(points)
    {
    }
};
```

Также нам нужно знать Имя и Возраст баскетболиста, а эта информация уже у нас есть: она хранится в классе Human.

У нас есть три варианта добавления Имени и Возраста в BasketballPlayer:

- Добавить Имя и Возраст в класс BasketballPlayer непосредственно в качестве членов. Это плохой вариант, так как произойдет дублирование кода, который уже существует в классе Human. Любые обновления в Human также должны быть продублированы и в BasketballPlayer.
- Добавить класс Human в качестве члена в класс BasketballPlayer, используя композицию. Но возникает вопрос: "Может ли BasketballPlayer иметь Human?". Нет, это некорректно.
- Сделать так, чтобы BasketballPlayer унаследовал необходимые атрибуты от Human. Помните, что тип отношений в наследовании — «является». Является ли BasketballPlayer Human-ом (т.е. Человеком)? Конечно! Поэтому наш выбор — наследование.

Делаем класс BasketballPlayer дочерним

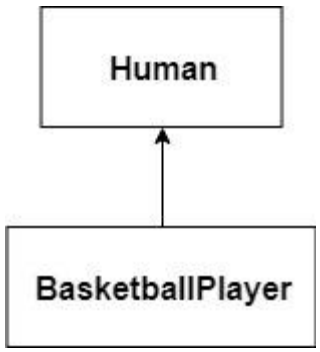
Чтобы класс BasketballPlayer унаследовал информацию от класса Human, нам нужно после объявления BasketballPlayer (`class BasketballPlayer`) использовать двоеточие, ключевое слово public и имя класса, от которого мы хотим унаследовать. Это называется **открытым наследованием**:

```
// BasketballPlayer открыто наследует Human
class BasketballPlayer : public Human
{
public:
    double m_gameAverage;
    int m_points;

    BasketballPlayer(double gameAverage = 0.0, int points = 0)
        : m_gameAverage(gameAverage), m_points(points)
    {
    }
```

```
};
```

Проиллюстрируем:



Когда BasketballPlayer наследует свойства класса Human, то BasketballPlayer приобретает методы и переменные-члены класса Human. Кроме того, BasketballPlayer имеет еще два своих собственных члена: `m_gameAverage` и `m_points`. Здесь есть смысл, так как эти свойства специфичны только для BasketballPlayer, а не для каждого Human-а.

Таким образом, объекты BasketballPlayer будут иметь 4 члена:

- `m_gameAverage` и `m_points` от BasketballPlayer;
- `m_name` и `m_age` от Human.

Полный код программы:

```
#include <iostream>
#include <string>

class Human
{
public:
    std::string m_name;
    int m_age;

    Human(std::string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

};

// BasketballPlayer открыто наследует Human
class BasketballPlayer : public Human
{
public:
    double m_gameAverage;
    int m_points;
```

```
26.
27.     BasketballPlayer(double gameAverage = 0.0, int points = 0)
28.         : m_gameAverage(gameAverage), m_points(points)
29.     {
30.     }
31. };
32.
33. int main()
34. {
35.     // Создаем нового Баскетболиста
36.     BasketballPlayer anton;
37.     // Присваиваем ему имя (мы можем делать это напрямую, так как m_name
    является public)
38.     anton.m_name = "Anton";
39.     // Выводим имя Баскетболиста
40.     std::cout << anton.getName() << '\n'; // используем метод getName(),
    который мы унаследовали от класса Human
41.
42.     return 0;
43. }
```

Результат выполнения программы:

```
Anton
```

Это работает, так как `anton` является объектом класса BasketballPlayer, а все объекты класса BasketballPlayer имеют переменную-член `m_name` и метод getName(), унаследованные от класса Human.

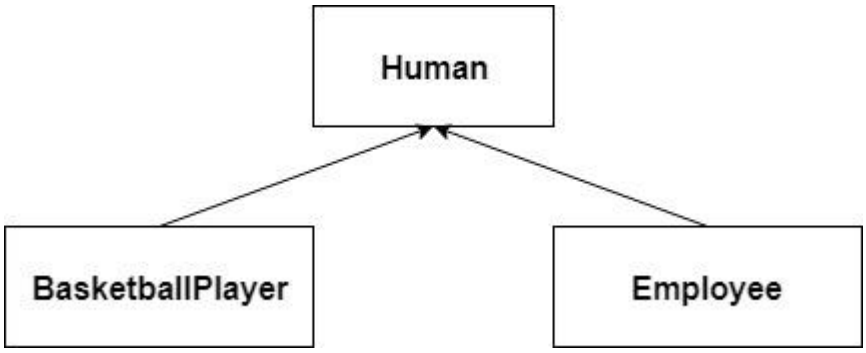
Дочерний класс Employee

Теперь напишем еще один класс, который также будет наследовать свойства Human. Например, класс Employee (Работник). Работник «является» Человеком, поэтому использовать наследование здесь уместно:

```
1. // Employee открыто наследует Human
2. class Employee: public Human
3. {
4. public:
5.     int m_wage;
6.     long m_employeeID;
7.
8.     Employee(int wage = 0, long employeeID = 0)
9.         : m_wage(wage), m_employeeID(employeeID)
10.    {
11.    }
12.
13.    void printNameAndWage() const
14.    {
15.        std::cout << m_name << ": " << m_wage << '\n';
16.    }
17. };
```

Работник наследует `m_name` и `m_age` от Human-а (а также два метода) и имеет еще две собственные переменные-члены и один метод. Обратите внимание, метод printNameAndWage() использует переменные как из класса, к которому принадлежит (`Employee::m_wage`), так и с родительского класса (`Human::m_name`).

Проиллюстрируем:



Обратите внимание, классы Employee и BasketballPlayer не имеют прямых отношений, хотя оба наследуют свойства класса Human.

Вот полный код:

```
#include <iostream>
#include <string>

class Human
{
public:
    std::string m_name;
    int m_age;

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

    Human(std::string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }
};

// Employee открыто наследует Human
class Employee: public Human
{
public:
    int m_wage;
    long m_employeeID;

    Employee(int wage = 0, long employeeID = 0)
        : m_wage(wage), m_employeeID(employeeID)
    {
    }

```

```
    void printNameAndWage() const
    {
        std::cout << m_name << ": " << m_wage << '\n';
    }
};

int main()
{
    Employee ivan(350, 787);
    ivan.m_name = "Ivan"; // мы можем это сделать, так как m_name является public

    ivan.printNameAndWage();

    return 0;
}
```

Результат выполнения программы:

```
Ivan: 350
```

Цепочка наследований

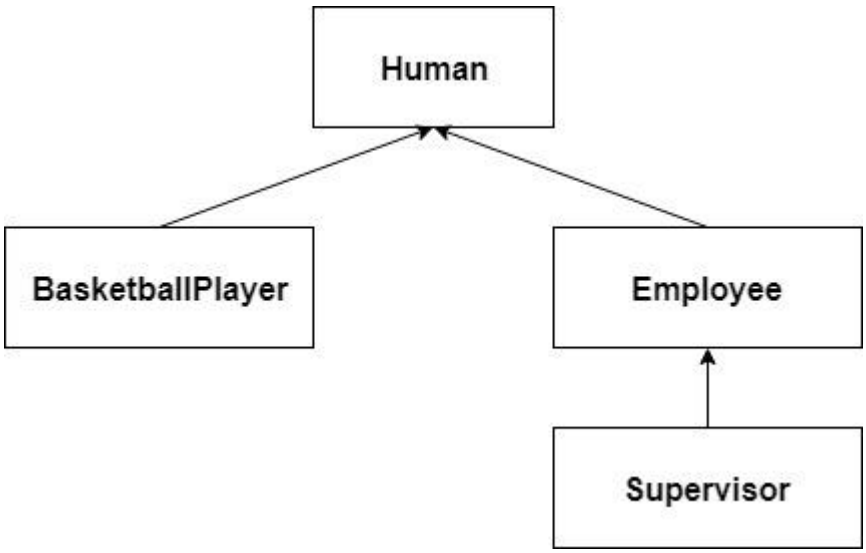
Можно наследовать от класса, который сам наследует от другого класса. При этом ничего примечательного или чего-нибудь особенного не происходит — всё аналогично тому, что мы рассмотрели выше. Например, напишем класс Supervisor (Супервайзер). Супервайзер — это Работник, который "является" Человеком. Мы уже написали класс Employee, поэтому будем его использовать в качестве родительского класса:

```
class Supervisor: public Employee
{
public:
    // Этот Супервайзер может наблюдать максимум за 5-тью Работниками
    long m_nOverseesIDs[5];

    Supervisor()
    {
    }

};
```

Смотрим:



Все объекты Supervisor наследуют методы и переменные от Employee и Human, а также имеют свою собственную переменную-член `m_nOverseesIDs`.

Построив такие цепочки наследований, мы можем создать набор повторно используемых классов, которые будут иметь общие свойства вверху и становиться всё более специфичными на каждом последующем уровне наследования.

Почему наследование является полезным?

Использование наследования означает, что нам не нужно переопределять информацию из родительских классов в дочерних. Мы автоматически получаем методы и переменные-члены суперкласса через наследование, а затем просто добавляем специфичные методы или переменные-члены, которые хотим. Это не только экономит время и усилия, но также является очень эффективным: если мы когда-либо обновим или изменим базовый класс (например, добавим новые функции или исправим ошибку), то все наши производные классы автоматически унаследуют эти изменения!

Например, если мы добавим новый метод в Human, то Employee и Supervisor автоматически получат доступ к нему. Если мы добавим новую переменную в Employee, Supervisor также получит доступ к ней. Это позволяет создавать новые классы более простым, интуитивно-понятным способом!

Заключение

Наследование позволяет повторно использовать классы путем наследования членов этих классов другими классами. На следующих уроках мы будем разбираться детально, как это всё работает.

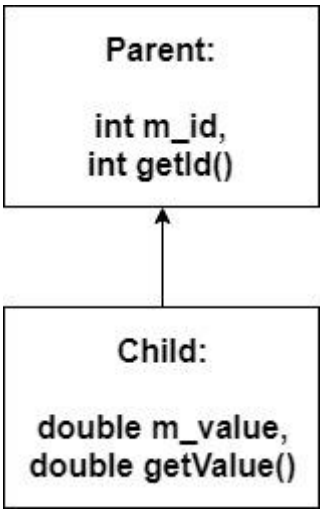
Урок №163. Порядок построения дочерних классов

На предыдущем уроке мы узнали, что классы могут наследовать переменные-члены и методы от других классов. На этом уроке мы рассмотрим порядок событий, которые выполняются при инициализации объектов дочернего класса.

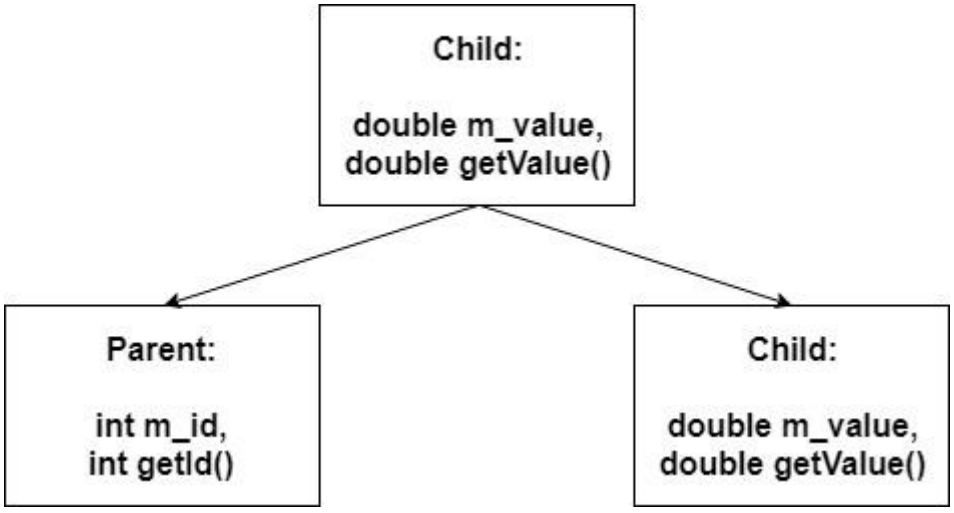
Во-первых, вот 2 класса, которые будут помогать нам иллюстрировать важные моменты:

```
1. class Parent
2. {
3. public:
4.     int m_id;
5. 
6.     Parent(int id=0)
7.         : m_id(id)
8.     {
9.     }
10. 
11.     int getId() const { return m_id; }
12. };
13. 
14. class Child: public Parent
15. {
16. public:
17.     double m_value;
18. 
19.     Child(double value=0.0)
20.         : m_value(value)
21.     {
22.     }
23. 
24.     double getValue() const { return m_value; }
25. };
```

В этом примере класс Child является дочерним, а класс Parent — родительским:



Поскольку Child наследует переменные-члены и методы класса Parent, то вы можете предположить, что члены класса Parent копируются в класс Child, но это не так. Вместо этого рассматривайте Child как класс, который состоит из двух частей: первая — Parent, вторая — Child:



Вы уже видели множество примеров того, что происходит при создании объектов обычного (не дочернего) класса:

```
1. int main()
2. {
3.     Parent parent;
4.
5.     return 0;
6. }
```

Parent — это не дочерний класс, так как он не наследует свойства каких-либо других классов. C++ выделяет память для Parent, затем вызывается конструктор по умолчанию класса Parent для выполнения инициализации.

Теперь рассмотрим, что происходит при создании объектов дочернего класса:

```
1. int main()
2. {
3.     Child child;
4.
5.     return 0;
6. }
```

На первый взгляд всё вроде бы так же, как и в предыдущем примере, но "под капотом" всё несколько иначе. Как мы уже говорили, класс Child состоит из двух частей: часть Parent и часть Child. Когда C++ создает объекты дочерних классов, то он делает это поэтапно. Сначала создается самый верхний класс иерархии (тот, который родитель). Затем создается дочерний класс, который идет следующим по

порядку, и так до тех пор, пока не будет создан последний класс (тот, который находится в самом низу иерархии).

Поэтому, при создании объекта класса Child, сначала создается часть Parent класса Child (с использованием конструктора по умолчанию класса Parent) и после того, как с частью Parent покончено, создается вторая часть Child (с использованием конструктора по умолчанию класса Child). В нашем примере больше нет дочерних классов, поэтому на этом всё и заканчивается.

Этот процесс на самом деле легко проиллюстрировать:

```
#include <iostream>

class Parent
{
public:
    int m_id;

    Parent(int id=0)
        : m_id(id)
    {
        std::cout << "Parent\n";
    }

    int getId() const { return m_id; }
};

class Child: public Parent
{
public:
    double m_value;

    Child(double value=0.0)
        : m_value(value)
    {
        std::cout << "Child\n";
    }

    double getValue() const { return m_value; }
};

int main()
{
    std::cout << "Instantiating Parent:\n";
    Parent dParent;

    std::cout << "Instantiating Child:\n";
    Child dChild;

    return 0;
}
```

Результат выполнения программы:

```
Instantiating Parent:
Parent
Instantiating Child:
Parent
Child
```

Как вы можете видеть, при создании Child сначала создается часть Parent класса Child. В этом есть смысл, так как по логике вещей ребенок не может существовать без родителей. Это также способствует безопасности и эффективности выполнения кода: дочерний класс часто использует переменные-члены и методы родителя, но родительский класс ничего не знает о своем дочернем классе. Первоначальная инициализация родительского класса гарантирует, что его переменные-члены и методы будут проинициализированы до момента использования их дочерним классом.

Порядок построения классов в цепочке наследования

Часто случаются ситуации, когда одни классы наследуют свойства других классов, которые, в свою очередь, наследуют свойства своих предыдущих (родительских) классов. Например:

```
1. class A
2. {
3. public:
4.     A()
5.     {
6.         std::cout << "A\n";
7.     }
8. };
9.
10.class B: public A
11.{
12.public:
13.    B()
14.    {
15.        std::cout << "B\n";
16.    }
17.};
18.
19.class C: public B
20.{
21.public:
22.    C()
23.    {
24.        std::cout << "C\n";
25.    }
26.};
27.
28.class D: public C
29.{
```

```
30. public:
31.     D()
32.     {
33.         std::cout << "D\n";
34.     }
35. };
```

Помните, что в C++ всегда идет построение с «первого» или «топового» класса иерархии. Затем C++ переходит к следующему классу в иерархии и выполняет его построение. Этот процесс последовательный.

Проиллюстрируем порядок построения классов в цепочке наследования, приведенной выше:

```
1.  int main()
2.  {
3.      std::cout << "Constructing A: \n";
4.      A cA;
5.
6.      std::cout << "Constructing B: \n";
7.      B cB;
8.
9.      std::cout << "Constructing C: \n";
10.     C cC;
11.
12.     std::cout << "Constructing D: \n";
13.     D cD;
14. }
```

Результат:

```
Constructing A:
A
Constructing B:
A
B
Constructing C:
A
B
C
Constructing D:
A
B
C
D
```

Заключение

Язык C++ выполняет построение дочерних классов поэтапно, начиная с верхнего класса иерархии и заканчивая нижним классом иерархии. По мере построения

каждого класса для выполнения инициализации вызывается соответствующий конструктор соответствующего класса.

На следующем уроке мы рассмотрим роль конструкторов в процессе построения дочерних классов.

Урок №164. Конструкторы и инициализация дочерних классов

На предыдущих уроках мы изучили основы наследования в языке C++ и порядок инициализации дочерних классов. На этом уроке мы подробнее рассмотрим роль конструкторов в инициализации дочерних классов.

Конструкторы и инициализация

А помогать нам в этом будут классы Parent и Child:

```
1. class Parent
2. {
3. public:
4.     int m_id;
5. 
6.     Parent(int id=0)
7.         : m_id(id)
8.     {
9.     }
10. 
11.     int getId() const { return m_id; }
12. };
13. 
14. class Child: public Parent
15. {
16. public:
17.     double m_value;
18. 
19.     Child(double value=0.0)
20.         : m_value(value)
21.     {
22.     }
23. 
24.     double getValue() const { return m_value; }
25. };
```

С обычными классами (не дочерними) конструктору нужно заморачиваться только с членами своего класса. Например, объект класса Parent создается следующим образом:

```
1. int main()
2. {
3.     Parent parent(7); // вызывается конструктор Parent(int)
4. 
5.     return 0;
6. }
```

Вот что на самом деле происходит при инициализации объекта `parent`:

- выделяется память для объекта `parent`;

- вызывается соответствующий конструктор класса Parent;
- список инициализации инициализирует переменные;
- выполняется тело конструктора;
- точка выполнения возвращается обратно в caller.

Всё довольно-таки просто. С дочерними классами дела обстоят несколько сложнее:

```
1. int main()
2. {
3.     Child child(1.5); // вызывается конструктор Child(double)
4. 
5.     return 0;
6. }
```

Вот что происходит при инициализации объекта `child`:

- выделяется память для объекта дочернего класса (достаточная порция памяти для части Parent и части Child объекта класса Child);
- вызывается соответствующий конструктор класса Child;
- создается объект класса Parent с использованием соответствующего конструктора класса Parent. Если такой конструктор программистом не предоставлен, то будет использоваться конструктор по умолчанию класса Parent;
- список инициализации инициализирует переменные;
- выполняется тело конструктора класса Child;
- точка выполнения возвращается обратно в caller.

Единственное различие между инициализацией объектов обычного и дочернего класса заключается в том, что при инициализации объекта дочернего класса, сначала выполняется конструктор родительского класса (для инициализации части родительского класса) и только потом уже выполняется конструктор дочернего класса.

Инициализация членов родительского класса

Одним из недостатков нашего дочернего класса Child является то, что мы не можем инициализировать `m_id` при создании объекта класса Child. Что, если мы хотим задать значение как для `m_value` (части Child), так и для `m_id` (части Parent)?

Новички часто пытаются решить эту проблему следующим образом:

```
1. class Child: public Parent
2. {
3. public:
4.     double m_value;
```

```
5.
6.      Child(double value=0.0, int id=0)
7.          // Не сработает
8.          : m_value(value), m_id(id)
9.      {
10.     }
11.
12.     double getValue() const { return m_value; }
13. };
```

Это хорошая попытка и почти правильная идея. Нам определенно нужно добавить еще один параметр в наш конструктор, иначе C++ не будет понимать, каким значением мы хотим инициализировать `m_id`.

Однако C++ запрещает дочерним классам инициализировать наследуемые переменные-члены родительского класса в списке инициализации своего конструктора. Другими словами, значение переменной может быть задано только в списке инициализации конструктора, принадлежащего тому же классу, что и переменная-член.

Почему C++ так делает? Ответ связан с константными переменными и ссылками. Подумайте, что произошло бы, если бы `m_id` был const. Поскольку константы должны быть инициализированы значениями при создании, то конструктор родительского класса должен установить это значение при создании переменной-члена. В то же время конструктор дочернего класса выполняется только после выполнения конструкторов родительского класса. Каждый дочерний класс имел бы тогда возможность инициализировать эту переменную, потенциально изменяя её значение! Ограничивая инициализацию переменных конструктором класса, к которому принадлежат эти переменные, язык C++ гарантирует, что все переменные будут инициализированы только один раз.

Конечным результатом выполнения кода, приведенного выше, является ошибка, так как `m_id` унаследован от класса Parent, а только ненаследуемые переменные-члены могут быть изменены в списке инициализации конструктора класса Child.

Однако наследуемые переменные могут по-прежнему изменять свои значения в теле конструктора через операцию присваивания. Следовательно, новички часто пытаются сделать следующее:

```
1. class Child: public Parent
2. {
3. public:
4.     double m_value;
5.
6.     Child(double value=0.0, int id=0)
7.         : m_value(value)
8.     {
```

```
9.          m_id = id;
10.     }
11.
12.     double getValue() const { return m_value; }
13. };
```

Хотя подобное действительно сработает в данном случае, но это не сработает, если `m_id` будет константой или ссылкой (поскольку константы и ссылки должны быть инициализированы в списке инициализации конструктора). Кроме того, это неэффективно, так как для `m_id` присваивают значение дважды: первый раз в списке инициализации конструктора класса Parent, а затем в теле конструктора класса Child. И, наконец, что, если классу Parent необходим доступ к этому значению во время инициализации?

Итак, как правильно инициализировать `m_id` при создании объекта класса Child?

Во всех наших примерах, при создании объекта класса Child, вызывался конструктор по умолчанию класса Parent. Почему так? Потому что мы не указывали иначе!

К счастью, язык C++ предоставляет нам возможность явно выбирать конструктор класса Parent для выполнения инициализации части Parent! Для этого нам необходимо просто добавить вызов нужного нам конструктора в списке инициализации конструктора дочернего класса:

```
1. class Child: public Parent
2. {
3. public:
4.     double m_value;
5.
6.     Child(double value=0.0, int id=0)
7.         : Parent(id), // вызывается конструктор Parent(int) со значением id!
8.             m_value(value)
9.     {
10.    }
11.
12.    double getValue() const { return m_value; }
13. };
```

Теперь при выполнении следующего кода:

```
1. int main()
2. {
3.     Child child(1.5, 7); // вызывается конструктор Child(double, int)
4.     std::cout << "ID: " << child.getId() << '\n';
5.     std::cout << "Value: " << child.getValue() << '\n';
6.
7.     return 0;
8. }
```

Конструктор `Parent(int)` будет использоваться для инициализации `m_id` значением `7`, а конструктор дочернего класса будет использоваться для инициализации `m_value` значением `1.5`!

Результат выполнения программы:

```
ID: 7
Value: 1.5
```

Рассмотрим детально, что происходит:

- Выделяется память для объекта `child`.
- Вызывается конструктор `Child(double, int)`, где `value = 1.5`, а `id = 7`.
- Компилятор смотрит, запрашиваем ли мы какой-нибудь конкретный конструктор класса Parent. И видит, что запрашиваем! Поэтому вызывается `Parent(int)` с параметром `id`, которому мы до этого присвоили значение `7`.
- Список инициализации конструктора класса Parent присваивает для `m_id` значение `7`.
- Выполняется тело конструктора класса Parent, которое ничего не делает.
- Завершается выполнения конструктора класса Parent.
- Список инициализации конструктора класса Child присваивает для `m_value` значение `1.5`.
- Выполняется тело конструктора класса Child, которое ничего не делает.
- Завершается выполнения конструктора класса Child.

Это может показаться несколько сложным, но на самом деле всё очень просто. Всё, что происходит — это вызов конструктором класса Child конкретного конструктора класса Parent для инициализации части Parent объекта класса Child. Поскольку `m_id` находится в части Parent, то только конструктор класса Parent может инициализировать это значение.

Обратите внимание, не имеет значения, где в списке инициализации конструктора класса Child вызывается конструктор класса Parent — он всегда будет выполняться первым.

Теперь мы можем сделать наши члены private

Теперь, когда мы знаем о инициализации членов родительского класса, нет никакой необходимости сохранять наши переменные-члены открытыми. Мы сделаем их private, как и должно быть.

> ***В качестве напоминания***: Доступ к членам public открыт для всех. Доступ к членам private открыт только для других членов этого же класса. Обратите внимание, это означает, что дочерние классы не могут напрямую обращаться к закрытым членам родительского класса! Дочерним классам нужно использовать геттеры и сеттеры для доступа к этим членам.

Например:

```
#include <iostream>

class Parent
{
private: // наш m_id теперь закрытый
    int m_id;

public:
    Parent(int id=0)
        : m_id(id)
    {
    }

    int getId() const { return m_id; }
};
class Child: public Parent
{
private: // наш m_value теперь закрытый
    double m_value;

public:
    Child(double value=0.0, int id=0)
        : Parent(id), // вызывается конструктор Parent(int) со значением id!
            m_value(value)
    {
    }

    double getValue() const { return m_value; }
};

int main()
{
    Child child(1.5, 7); // вызывается конструктор Child(double, int)
    std::cout << "ID: " << child.getId() << '\n';
    std::cout << "Value: " << child.getValue() << '\n';

    return 0;
}
```

В коде, приведенном выше, мы делаем `m_id` и `m_value` закрытыми. Для их инициализации используются соответствующие конструкторы, а для доступа — открытые функции доступа (геттеры).

Результат выполнения программы:

```
ID: 7
Value: 1.5
```

Еще один пример

Рассмотрим еще пару классов, с которыми мы работали ранее:

```
#include <iostream>
#include <string>

class Human
{
public:
    std::string m_name;
    int m_age;

    Human(std::string name = "", int age = 0)
        : m_name(name), m_age(age )
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }
};

// BasketballPlayer открыто наследует класс Human
class BasketballPlayer: public Human
{
public:
    double m_gameAverage;
    int m_points;

    BasketballPlayer(double gameAverage = 0.0, int points = 0)
       : m_gameAverage(gameAverage), m_points(points)
    {
    }
};
```

Как мы уже знаем, класс BasketballPlayer только инициализирует свои собственные члены и не указывает использование конкретного конструктора класса Human. Это означает, что каждый созданный объект класса BasketballPlayer будет использовать конструктор по умолчанию класса Human, который будет инициализировать переменную-член `name` пустым значением, а `age` — значением `0`. Поскольку мы хотим назвать нашего BasketballPlayer и указать его возраст при его создании, то мы должны изменить этот конструктор, добавив необходимые параметры.

Вот наши обновленные классы с членами private и с вызовом конкретного конструктора класса Human:

```
#include <string>

class Human
{
private:
    std::string m_name;
    int m_age;

public:
    Human(std::string name = "", int age = 0)
        : m_name(name), m_age(age )
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

};
// BasketballPlayer открыто наследует класс Human
class BasketballPlayer: public Human
{
private:
    double m_gameAverage;
    int m_points;

public:
    BasketballPlayer(std::string name = "", int age = 0,
        double gameAverage = 0.0, int points = 0)
        : Human(name, age), // вызывается Human(std::string, int) для инициализации членов name и age
            m_gameAverage(gameAverage), m_points(points)
    {
    }

    double getGameAverage() const { return m_gameAverage; }
    int getPoints() const { return m_points; }
};
```

Теперь мы можем создавать объекты класса BasketballPlayer следующим образом:

```
int main()
{
    BasketballPlayer anton("Anton Ivanovuch", 45, 300, 310);

    std::cout << anton.getName() << '\n';
    std::cout << anton.getAge() << '\n';
    std::cout << anton.getPoints() << '\n';

    return 0;
}
```

Результат выполнения программы:

```
Anton Ivanovuch
45
310
```

Как вы можете видеть, всё корректно инициализировано.

Цепочки наследований

Классы в цепочке наследований работают аналогичным образом:

```
#include <iostream>

class A
{
public:
    A(int a)
    {
        std::cout << "A: " << a << '\n';
    }
};

class B: public A
{
public:
    B(int a, double b)
    : A(a)
    {
        std::cout << "B: " << b << '\n';
    }
};

class C: public B
{
public:
    C(int a , double b , char c)
    : B(a, b)
    {
        std::cout << "C: " << c << '\n';
    }
};

int main()
{
    C c(7, 5.4, 'D');

    return 0;
}
```

В этом примере класс C наследует свойства класса B, который наследует свойства класса A. Что произойдет при создании объекта класса C? А вот что:

- функция main() вызовет `C(int, double, char);`
- конструктор класса C вызовет `B(int, double);`

- конструктор класса B вызовет `A(int)`;
- поскольку A не наследует никакой класс, то построение начнется именно с этого класса;
- построение A выполнено, выводится значение `7` и выполнение переходит в B;
- класс B построен, выводится значение `5.4` и выполнение переходит в C;
- класс C построен, выводится `D` и выполнение возвращается обратно в main();
- Финиш!

Таким образом, результат выполнения программы:

```
A: 7
B: 5.4
C: D
```

Стоит отметить, что конструкторы дочернего класса могут вызывать конструкторы только того родительского класса, от которого они напрямую наследуют. Следовательно, конструктор класса C не может напрямую вызывать или передавать параметры в конструктор класса A. Конструктор класса C может вызывать только конструктор класса B (который уже, в свою очередь, вызывает конструктор класса A).

Деструкторы

При уничтожении дочернего класса, каждый деструктор вызывается в порядке обратном построению классов. В примере, приведенном выше, при уничтожении объекта класса C, сначала вызывается деструктор класса C, затем деструктор класса B, а затем уже деструктор класса A.

Заключение

При инициализации объектов дочернего класса, конструктор дочернего класса отвечает за то, какой конструктор родительского класса вызывать. Если этот конструктор явно не указан, то вызывается конструктор по умолчанию родительского класса. Если же компилятор не может найти конструктор по умолчанию родительского класса (или этот конструктор не может быть создан автоматически), то компилятор выдаст ошибку.

Тест

Реализуем наш пример с Фруктом, о котором мы говорили на уроке №161. Создайте родительский класс Fruit, который имеет два закрытых члена: `name` (std::string) и `color` (std::string). Создайте класс Apple, который наследует

свойства Fruit. У Apple должен быть дополнительный закрытый член: `fiber` (тип double). Создайте класс Banana, который также наследует класс Fruit. Banana не имеет дополнительных членов.

Следующий код:

```
1. int main()
2. {
3.     const Apple a("Red delicious", "red", 7.3);
4.     std::cout << a;
5. 
6.     const Banana b("Cavendish", "yellow");
7.     std::cout << b;
8. 
9.     return 0;
10.}
```

Должен выдавать следующий результат:

```
Apple(Red delicious, red, 7.3)
Banana(Cavendish, yellow)
```

> ***Подсказка***: Поскольку `a` и `b` являются const, то убедитесь, что ваши параметры и функции соответствуют const.

Урок №165. Наследование и спецификатор доступа protected

На предыдущих уроках мы говорили о том, как работает наследование в языке C++. Во всех наших примерах мы использовали открытое наследование.

На этом уроке мы рассмотрим детально этот тип наследования, а также два других типа: private и protected. Также поговорим о том, как эти типы наследований взаимодействуют со спецификаторами доступа для разрешения или ограничения доступа к членам.

Спецификатор доступа protected

Мы уже рассматривали спецификаторы доступа private и public, которые определяют, кто может иметь доступ к членам класса. В качестве напоминания: доступ к членам public открыт для всех, к членам private доступ имеют только члены того же класса, в котором находится член private. Это означает, что дочерние классы не могут напрямую обращаться к членам private родительского класса!

```
1. class Parent
2. {
3. private:
4.     int m_private; // доступ к этому члену есть только у других членов класса Parent и у дружественных классов/функций (но не у дочерних классов)
5. public:
6.     int m_public; // доступ к этому члену открыт для всех объектов
7. };
```

Всё просто.

> ***Примечание*:** public = «открытый», private = «закрытый», protected = «защищенный».

В языке C++ есть третий спецификатор доступа, о котором мы еще не говорили, так как он полезен только в контексте наследования. **Спецификатор доступа protected открывает доступ к членам класса дружественным и дочерним классам**. Доступ к члену protected вне тела класса закрыт.

```
1. class Parent
2. {
3. public:
4.     int m_public; // доступ к этому члену открыт для всех объектов
5. private:
6.     int m_private; // доступ к этому члену открыт только для других членов класса Parent и для дружественных классов/функций (но не для дочерних классов)
7. protected:
```

```
8.      int m_protected; // доступ к этому члену открыт для других членов класса Parent, дружественных классов/функций, дочерних классов
9.  };
10.
11. class Child: public Parent
12. {
13. public:
14.     Child()
15.     {
16.         m_public = 1; // разрешено: доступ к открытым членам родительского класса из дочернего класса
17.         m_private = 2; // запрещено: доступ к закрытым членам родительского класса из дочернего класса
18.         m_protected = 3; // разрешено: доступ к защищенным членам родительского класса из дочернего класса
19.     }
20. };
21.
22. int main()
23. {
24.     Parent parent;
25.     parent.m_public = 1; // разрешено: доступ к открытым членам класса извне
26.     parent.m_private = 2; // запрещено: доступ к закрытым членам класса извне
27.     parent.m_protected = 3; // запрещено: доступ к защищенным членам класса извне
28. }
```

В примере, приведенном выше, вы можете видеть, что член `m_protected` класса Parent напрямую доступен дочернему классу Child, но доступ к нему для членов извне — закрыт.

Когда следует использовать спецификатор доступа protected?

К членам protected родительского класса доступ открыт для членов дочернего класса, а это означает, что если вы позже измените что-либо в члене protected (тип данных, значение и т.д.), то вам придется внести изменения как в родительский, так и во все дочерние классы. Поэтому использование спецификатора доступа protected наиболее полезно, когда вы будете наследовать только свои же классы и количество дочерних классов будет небольшое. Таким образом, если вы внесете изменения в реализацию родительского класса, и вам понадобится обновить все дочерние классы, то вы сможете сделать эти обновления сами и это не займет много времени (так как дочерних классов будет немного).

Создание членов private предоставляет лучшую инкапсуляцию и изолирует родительские классы от изменений, вызванных дочерними классами. Но цена этому — дополнительное создание открытого или защищенного интерфейса (способа взаимодействия других объектов с классами и их членами, т.е. геттеры и сеттеры). Это дополнительная работа, которая не стоит того, если вы сами работаете со своими же классами (чужие классы не обращаются к вашему классу) и количество дочерних классов небольшое.

Типы наследований. Доступ к членам

Существует три типа наследований классов:

- **public;**
- **private;**
- **protected**.

Для определения типа наследования нужно просто указать нужное ключевое слово возле наследуемого класса:

```
1. // Открытое наследование
2. class Pub: public Parent
3. {
4. };
5.
6. // Закрытое наследование
7. class Pri: private Parent
8. {
9. };
10.
11.// Защищенное наследование
12.class Pro: protected Parent
13.{
14.};
15.
16.class Def: Parent// по умолчанию язык C++ устанавливает закрытое наследование
17.{
18.};
```

Если вы сами не определили тип наследования, то в языке C++ по умолчанию будет выбран тип наследования private (аналогично и для членов класса, которые по умолчанию являются private, если не указано иначе).

Это дает нам 9 комбинаций: 3 спецификатора доступа (public, private и protected) и 3 типа наследования (public, private и protected).

Так в чем же разница между ними? Если вкратце, то при наследовании спецификатор доступа члена родительского класса может быть изменен в дочернем классе (в зависимости от типа наследования). Другими словами, члены, которые были public или protected в родительском классе, могут стать private в дочернем классе.

Это может показаться немного запутанным, но всё не так уж плохо. Мы сейчас со всем этим разберемся, но перед этим вспомним **следующие правила**:

- Класс всегда имеет доступ к своим (не наследуемым) членам.
- Доступ к члену класса основывается на его спецификаторе доступа.

- Дочерний класс имеет доступ к унаследованным членам родительского класса на основе спецификатора доступа этих членов в родительском классе.

Наследование типа public

Открытое наследование является одним из наиболее используемых типов наследования. Очень редко вы увидите или будете использовать другие типы, поэтому основной упор следует сделать на понимание именно этого типа наследования. К счастью, открытое наследование является самым легким и простым из всех типов. Когда вы открыто наследуете родительский класс, то унаследованные члены public остаются public, унаследованные члены protected остаются protected, а унаследованные члены private остаются недоступными для дочернего класса. Ничего не меняется.

Спецификатор доступа в родительском классе	Спецификатор доступа при наследовании типа public в дочернем классе
public	public
private	Недоступен
protected	protected

Например:

```
1. class Parent
2. {
3. public:
4.     int m_public;
5. private:
6.     int m_private;
7. protected:
8.     int m_protected;
9. };
10.
11.class Pub: public Parent // открытое наследование
12.{
13.    // Открытое наследование означает, что:
14.    // - члены public остаются public в дочернем классе;
15.    // - члены protected остаются protected в дочернем классе;
16.    // - члены private остаются недоступными в дочернем классе.
17.public:
18.    Pub()
19.    {
20.        m_public = 1; // разрешено: доступ к m_public открыт
```

```
        m_private = 2; // запрещено: доступ к m_private в дочернем классе из родительского класса закрыт
        m_protected = 3; // разрешено: доступ к m_protected в дочернем классе из родительского класса открыт
    }
};

int main()
{
    Parent parent;
    parent.m_public = 1; // разрешено: m_public доступен извне через родительский класс
    parent.m_private = 2; // запрещено: m_private недоступен извне через родительский класс
    parent.m_protected = 3; // запрещено: m_protected недоступен извне через родительский класс

    Pub pub;
    pub.m_public = 1; // разрешено: m_public доступен извне через дочерний класс
    pub.m_private = 2; // запрещено: m_private недоступен извне через дочерний класс
    pub.m_protected = 3; // запрещено: m_protected недоступен извне через дочерний класс
}
```

Правило: Используйте открытое наследование, если у вас нет веских причин делать иначе.

Наследование типа private

При закрытом наследовании все члены родительского класса наследуются как закрытые. Это означает, что члены private остаются недоступными, а члены protected и public становятся private в дочернем классе.

Обратите внимание, это не влияет на то, как дочерний класс получает доступ к членам родительского класса! Это влияет только на то, как другими объектами осуществляется доступ к этим членам через дочерний класс:

```
class Parent
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Priv: private Parent // закрытое наследование
{
    // Закрытое наследование означает, что:
    // - члены public становятся private (m_public теперь private) в дочернем классе;
    // - члены protected становятся private (m_protected теперь private) в дочернем классе;
```

```
    // - члены private остаются недоступными (m_private недоступен) в дочернем классе.
public:
    Priv()
    {
        m_public = 1; // разрешено: m_public теперь private в Priv
        m_private = 2; // запрещено: дочерние классы не имеют доступ к закрытым членам родительского класса
        m_protected = 3; // разрешено: m_protected теперь private в Priv
    }
};

int main()
{
    Parent parent;
    parent.m_public = 1; // разрешено: m_public доступен извне через родительский класс
    parent.m_private = 2; // запрещено: m_private недоступен извне через родительский класс
    parent.m_protected = 3; // запрещено: m_protected недоступен извне через родительский класс

    Priv priv;
    priv.m_public = 1; // запрещено: m_public недоступен извне через дочерний класс
    priv.m_private = 2; // запрещено: m_private недоступен извне через дочерний класс
    priv.m_protected = 3; // запрещено: m_protected недоступен извне через дочерний класс
}
```

Итого:

Спецификатор доступа в родительском классе	Спецификатор доступа при наследовании типа private в дочернем классе
public	private
private	Недоступен
protected	private

Закрытое наследование может быть полезно, когда дочерний класс не имеет очевидной связи с родительским классом, но использует его в своей реализации. В таком случае мы не хотим, чтобы открытый интерфейс родительского класса был доступен через объекты дочернего класса (как это было, когда мы использовали открытый тип наследования).

На практике наследование типа private используется редко.

Наследование типа protected

Этот тип наследования почти никогда не используется, за исключением особых случаев. С защищенным наследованием, члены public и protected становятся protected, а члены private остаются недоступными.

Поскольку этот тип наследования очень редко используется, то мы пропустим пример на практике и сразу перейдем к таблице:

Спецификатор доступа в родительском классе	Спецификатор доступа при наследовании типа protected в дочернем классе
public	protected
private	Недоступен
protected	protected

Финальный пример

```
1. class Parent
2. {
3. public:
4.     int m_public;
5. private:
6.     int m_private;
7. protected:
8.     int m_protected;
9. };
```

Класс Parent может обращаться к своим членам беспрепятственно. Доступ к `m_public` открыт для всех. Дочерние классы могут обращаться как к `m_public`, так и к `m_protected`:

```
1. class D2 : private Parent  // закрытое наследование
2. {
3.     // Закрытое наследование означает, что:
4.     // - члены public становятся private в дочернем классе;
5.     // - члены protected становятся private в дочернем классе;
6.     // - члены private недоступны для дочернего класса.
7. public:
8.     int m_public2;
9. private:
10.    int m_private2;
11. protected:
12.    int m_protected2;
```

```
13. };
```

Класс D2 может беспрепятственно обращаться к своим членам. D2 имеет доступ к членам `m_public` и `m_protected` класса Parent, но не к `m_private`. Поскольку D2 наследует класс Parent закрыто, то `m_public` и `m_protected` теперь становятся закрытыми при доступе через D2. Это означает, что другие объекты не смогут получить доступ к этим членам через использование объекта D2, а также любые другие классы, которые будут дочерними классу D2, не будут иметь доступ к этим членам:

```
1. class D3 : public D2
2. {
3.     // Открытое наследование означает, что:
4.     // - унаследованные члены public остаются public в дочернем классе;
5.     // - унаследованные члены protected остаются protected в дочернем классе;
6.     // - унаследованные члены private остаются недоступными в дочернем классе.

7. public:
8.     int m_public3;
9. private:
10.    int m_private3;
11. protected:
12.    int m_protected3;
13. };
```

Класс D3 может беспрепятственно обращаться к своим членам. D3 имеет доступ к членам `m_public2` и `m_protected2` класса D2, но не к `m_private2`. Поскольку D3 наследует D2 открыто, то `m_public2` и `m_protected2` сохраняют свои спецификаторы доступа и остаются public и protected при доступе через D3. D3 не имеет доступ к `m_private` класса Parent. Он также не имеет доступ к `m_protected` или `m_public` класса Parent, оба из которых стали закрытыми, когда D2 унаследовал их.

Заключение

Способ взаимодействия спецификаторов доступа, типов наследования и дочерних классов может вызывать путаницу. Чтобы это устранить, проясним все еще раз:

- Во-первых, класс всегда имеет доступ к своим собственным не унаследованным членам (и дружественные ему классы также имеют доступ). Спецификаторы доступа влияют только на то, могут ли объекты вне класса и дочерние классы обращаться к этим членам.
- Во-вторых, когда дочерние классы наследуют члены родительских классов, то члены родительского класса могут изменять свои спецификаторы доступа в дочернем классе. Это никак не влияет на собственные (не наследуемые) члены дочерних классов (которые определены в дочернем классе и имеют

свои собственные спецификаторы доступа). Это влияет только на то, могут ли объекты извне и классы, дочерние нашим дочерним классам, получить доступ к унаследованным членам родительского класса.

Общая таблица спецификаторов доступа и типов наследования:

Спецификатор доступа в родительском классе	Спецификатор доступа при наследовании типа public в дочернем классе	Спецификатор доступа при наследовании типа private в дочернем классе	Спецификатор доступа при наследовании типа protected в дочернем классе
public	public	private	protected
private	Недоступен	Недоступен	Недоступен
protected	protected	private	protected

Хотя в вышеприведенных примерах мы рассматривали использование переменных-членов, эти правила выполняются для всех членов классов (и для методов, и для типов, объявленных внутри класса).

Урок №166. Добавление нового функционала в дочерний класс

На уроке №161 мы говорили о том, что одним из самых больших преимуществ использования дочерних классов является возможность повторного использования уже написанного кода. Мы можем наследовать функционал родительского класса, а затем добавить свой функционал/изменить существующий функционал/скрыть ненужные части родительского функционала. На этом и следующих уроках мы подробно рассмотрим, как это всё сделать.

Во-первых, начнем с класса Parent:

```
#include <iostream>

class Parent
{
protected:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

    void identify() { std::cout << "I am a Parent\n"; }
};
```

Теперь создадим дочерний класс, который будет наследовать класс Parent. Поскольку мы хотим иметь возможность установить значение `m_value` при инициализации объектов дочернего класса, то сделаем так, чтобы конструктор класса Child вызывал конкретный конструктор класса Parent в списке инициализации:

```
class Child: public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }
};
```

Добавление нового функционала в дочерний класс

В примере, приведенном выше, поскольку мы имеем доступ к исходному коду класса Parent, то мы можем добавить новый функционал непосредственно в класс Parent, если захотим.

Есть случаи, когда мы имеем доступ к родительскому классу, но не хотим его изменять. Например, мы только что купили библиотеку кода у стороннего поставщика, но нам нужен дополнительный функционал. Мы можем добавить его в исходный код библиотеки, но это будет не лучшим решением. Что, если придет обновление? После обновления нам заново придется вручную перенести весь код, что займет время и, кроме того, после обновления остается риск, что наш старый код уже не будет работать так, как нужно с кодом библиотеки.

Также есть случаи, когда изменить родительский класс невозможно. Например, мы не можем изменить код, который является частью Стандартной библиотеки C++. Но мы можем унаследовать классы из этой библиотеки, а затем добавить нужный нам функционал в наши дочерние классы. То же самое касается сторонних библиотек, где нам предоставлены заголовочные файлы, но код поставляется предварительно скомпилированным.

В любом случае лучшим решением является написание собственного родительского класса и добавление нужного нам функционала в наши дочерние классы.

Один нюанс с классом Parent заключается в доступе других объектов к `m_value`. Мы можем это исправить, добавив функцию доступа в класс Parent, но, ради примера, добавим геттер в класс Child. Поскольку `m_value` объявлен как protected в классе Parent, то Child имеет прямой доступ к нему.

Добавление нового функционала в дочерний класс выполняется как обычно:

```
1. class Child: public Parent
2. {
3. public:
4.     Child(int value)
5.         : Parent(value)
6.     {
7.     }
8.
9.     int getValue() { return m_value; }
10. };
```

Теперь другие объекты извне смогут вызывать getValue() через объект класса Child для доступа к `m_value`:

```
1. int main()
2. {
3.     Child child(7);
4.     std::cout << "child has value " << child.getValue() << '\n';
5.
6.     return 0;
7. }
```

Результат:

```
child has value 7
```

Это очевидно, что объекты класса Parent не имеют доступа к методу getValue() в Child. Следующее не сработает:

```
1. int main()
2. {
3.     Parent parent(7);
4.     std::cout << "parent has value " << parent.getValue() << '\n';
5.
6.     return 0;
7. }
```

Это связано с тем, что в классе Parent нет метода getValue(). Метод getValue() принадлежит классу Child. А, поскольку Child является дочерним классу Parent, то Child имеет доступ к членам Parent, а Parent не имеет доступа ни к чему в классе Child.

Урок №167. Переопределение методов родительского класса

Дочерние классы по умолчанию наследуют все методы родительского класса. На этом уроке мы рассмотрим, как это происходит, а также то, как можно изменить методы родительских классов в дочерних классах.

Вызов методов родительского класса

При вызове метода через объект дочернего класса, компилятор сначала смотрит, существует ли этот метод в дочернем классе. Если нет, то он начинает продвигаться по цепочке наследования вверх и проверяет, был ли этот метод определен в любом из родительских классов. Компилятор будет использовать первое найденное определение. Например:

```
#include <iostream>

class Parent
{
protected:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

    void identify() { std::cout << "I am a Parent!\n"; }
};

class Child : public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }
};

int main()
{
    Parent parent(6);
    parent.identify();

    Child child(8);
    child.identify();

    return 0;
}
```

Результат выполнения программы:

```
I am a Parent!
I am a Parent!
```

При вызове `child.identify()`, компилятор смотрит, определен ли метод identify() в классе Child. Нет, поэтому компилятор переходит к классу Parent. В классе Parent есть определение метода identify(), поэтому компилятор использует именно это определение.

Переопределение методов родительского класса

Однако, если бы мы определили метод identify() в классе Child, то использовалось бы именно это определение. Это означает, что мы можем заставить родительские методы работать по-другому с нашими дочерними классами, просто переопределяя их в дочерних классах!

Вышеприведенный пример станет лучше, если `child.identify()` будет выводить `I am a Child!`. Давайте изменим метод identify() в классе Child так, чтобы он возвращал правильный ответ.

Переопределение родительского метода в дочернем классе происходит, как обычное определение метода:

```
1. class Child : public Parent
2. {
3. public:
4.     Child(int value)
5.         : Parent(value)
6.     {
7.     }
8.
9.     int getValue() { return m_value; }
10.
11.    // Вот наш изменяемый метод родительского класса
12.    void identify() { std::cout << "I am a Child!\n"; }
13.};
```

Вот тот же код main(), что и в примере, приведенном выше, но уже с внесенными изменениями в класс Child:

```
1. int main()
2. {
3.     Parent parent(6);
4.     parent.identify();
5.
6.     Child child(8);
7.     child.identify();
8.
9.     return 0;
```

```
10. }
```

Результат:

```
I am a Parent!
I am a Child!
```

Обратите внимание, когда мы переопределяем родительский метод в дочернем классе, то дочерний метод не наследует спецификатор доступа родительского метода с тем же именем. Используется тот спецификатор доступа, который указан в дочернем классе. Таким образом, метод, определенный как private в родительском классе, может быть переопределен как public в дочернем классе, или наоборот!

```
1. #include <iostream>
2.
3. class Parent
4. {
5. private:
6.     void print()
7.     {
8.         std::cout << "Parent!";
9.     }
10. };
11.
12. class Child : public Parent
13. {
14. public:
15.     void print()
16.     {
17.         std::cout << "Child!";
18.     }
19.
20. };
21.
22.
23. int main()
24. {
25.     Child child;
26.     child.print(); // вызов child::print(), который является public
27.     return 0;
28. }
```

Расширение функционала родительских методов

Могут быть случаи, когда нам не нужно полностью заменять метод родительского класса, но нужно просто расширить его функционал. Обратите внимание, в примере, приведенном выше, метод Child::identify() полностью перекрывает Parent::identify()! Возможно, это не то, что нам нужно. Мы можем вызвать метод родительского класса с тем же именем в методе дочернего класса (для повторного использования кода), а затем добавить дополнительно свой код.

Чтобы метод дочернего класса вызывал метод родительского класса с тем же именем, нужно просто выполнить обычный вызов функции, но с добавлением имени родительского класса и оператора разрешения области видимости. В следующем примере мы выполним переопределение identify() в классе Child, вызывая сначала Parent::identify(), а затем добавляя уже свой код:

```
1. class Child : public Parent
2. {
3. public:
4.     Child(int value)
5.         : Parent(value)
6.     {
7.     }
8. 
9.     int GetValue() { return m_value; }
10. 
11.     void identify()
12.     {
13.         Parent::identify(); // сначала выполняется вызов Parent::identify()
14.         std::cout << "I am a Child!\n"; // затем уже вывод этого текста
15.     }
16. };
```

Вместе с:

```
1. int main()
2. {
3.     Parent parent(6);
4.     parent.identify();
5. 
6.     Child child(8);
7.     child.identify();
8. 
9.     return 0;
10. }
```

Дает результат:

```
I am a Parent!
I am a Parent!
I am a Child!
```

При выполнении `child.identify()` выполняется вызов Child::identify(). В Child::identify() мы сначала вызываем Parent::identify(), который выводит `I am a Parent!`. Когда Parent::identify() завершает свое выполнение, Child::identify() продолжает свое выполнение и выводит `I am a Child!`.

Всё просто. Зачем тогда нужно использовать оператор разрешения области видимости (`::`)?

А затем, что, если бы мы определили Child::identify() следующим образом:

```
1.  class Child : public Parent
2.  {
3.  public:
4.      Child(int value)
5.          : Parent(value)
6.      {
7.      }
8.
9.      int GetValue() { return m_value; }
10.
11.     void identify()
12.     {
13.         identify(); // нет оператора разрешения области видимости!
14.         std::cout << "I am a Child!";
15.     }
16. };
```

То вызов метода identify() без указания оператора разрешения области видимости привел бы к вызову identify() в текущем классе, т.е. Child::identify(). Затем снова вызов Child::identify(), и ура — у нас получился бесконечный цикл. Поэтому использование оператора разрешения области видимости является обязательным условием при изменении методов родительского класса.

Урок №168. Сокрытие методов родительского класса

Язык C++ предоставляет возможность изменить спецификатор доступа родительского члена в дочернем классе. Это делается с помощью "using-объявления". Например, рассмотрим следующий класс Parent:

```
#include <iostream>

class Parent
{
private:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

protected:
    void printValue() { std::cout << m_value; }
};
```

Поскольку Parent::printValue() объявлен как protected, то он доступен только другим членам Parent и своим дочерним классам. Для других объектов доступ к нему закрыт.

Определим класс Child, который изменяет спецификатор доступа printValue() с protected на public:

```
class Child: public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }

    // Parent::printValue является protected, поэтому доступ к нему не является открытым для всех объектов.
    // Но мы можем это исправить с помощью "using-объявления"
    using Parent::printValue; // обратите внимание, нет никаких скобок
};
```

Это означает, что следующий код выполнится без ошибок:

```
int main()
{
    Child child(9);

    // Метод printValue() является public в классе Child, поэтому всё хорошо
    child.printValue(); // выведется 9
    return 0;
}
```

Здесь есть два примечания:

- Во-первых, вы можете изменить спецификаторы доступа только для тех членов родительского класса, к которым есть доступ у дочернего класса. Вы не сможете изменить спецификатор доступа члена родительского класса с private на protected или public, поскольку дочерний класс не имеет доступа к членам private родительского класса.
- Во-вторых, начиная с C++11 использование "using-объявления" является предпочтительным способом изменения спецификаторов доступа. Однако вы также можете использовать "access-объявление". Это работает идентично "using-объявлению", только без ключевого слова using. Сейчас этот способ считается устаревшим, но, проглядывая более старый код, вы можете увидеть «access-объявление», поэтому об этом стоит знать.

Сокрытие родительских методов в дочернем классе

В языке C++ невозможно удалить или ограничить функционал родительского класса, кроме как с помощью непосредственного изменения исходного кода. Однако в дочернем классе вы можете скрыть функционал, существующий в родительском классе, так чтобы к нему нельзя было получить доступ через объекты дочернего класса. Это делается путем изменения соответствующих спецификаторов доступа.

Например, вы можете сделать открытый член родительского класса закрытым:

```
1. #include <iostream>
2.
3. class Parent
4. {
5. public:
6.     int m_value;
7. };
8.
9. class Child : public Parent
10.{
11.private:
12.    using Parent::m_value;
13.
14.public:
15.    Child(int value)
16.    // Мы не можем инициализировать m_value, поскольку это член класса Parent
   (Parent должен инициализировать m_value)
17.    {
18.        // Но мы можем присвоить значение
19.        m_value = value;
20.    }
21.};
22.
23.int main()
24.{
25.    Child child(9);
```

```
26.
27.     // Следующее не сработает, поскольку m_value был переопределен как private
28.     std::cout << child.m_value;
29.
30.     return 0;
31. }
```

Это позволяет инкапсулировать данные родительского класса в дочернем классе. В качестве альтернативы можно использовать наследование типа private, что приведет к тому, что все наследуемые члены public и protected класса Parent станут private в классе Child.

Вы также можете закрыть родительские методы в дочернем классе, используя ключевое слово delete:

```
1.  #include <iostream>
2.
3.  class Parent
4.  {
5.  private:
6.      int m_value;
7.
8.  public:
9.      Parent(int value)
10.         : m_value(value)
11.     {
12.     }
13.
14.     int getValue() { return m_value; }
15. };
16.
17. class Child : public Parent
18. {
19. public:
20.     Child(int value)
21.         : Parent(value)
22.     {
23.     }
24.
25.
26.     int getValue() = delete; // делаем этот метод недоступным
27. };
28.
29. int main()
30. {
31.     Child child(9);
32.
33.     // Следующее не сработает, поскольку getValue() удален
34.     std::cout << child.getValue();
35.
36.     return 0;
37. }
```

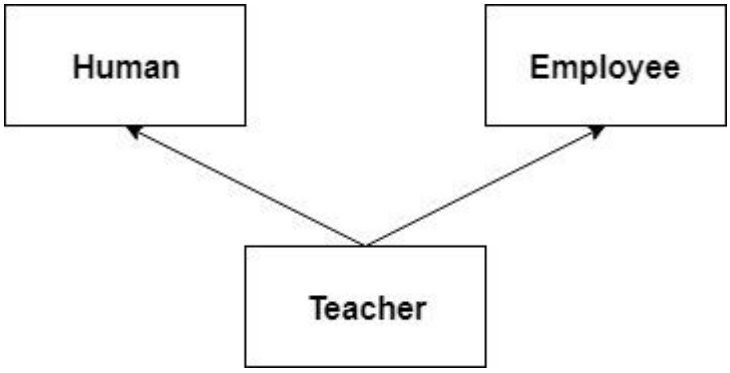
Таким образом, компилятор будет жаловаться, если мы попытаемся вызвать метод getValue() через объект класса Child. Однако через объект родительского класса всё будет работать, так как мы «удалили» getValue() только в дочернем классе.

Урок №169. Множественное наследование

До сих пор мы рассматривали только одиночные наследования, когда дочерний класс имеет только одного родителя. Однако C++ предоставляет возможность множественного наследования.

Множественное наследование

Множественное наследование позволяет одному дочернему классу иметь несколько родителей. Предположим, что мы хотим написать программу для отслеживания работы учителей. Учитель - это Human. Тем не менее, он также является Сотрудником (Employee).



Множественное наследование может быть использовано для создания класса Teacher, который будет наследовать свойства как Human, так и Employee. Для использования множественного наследования нужно просто указать через запятую тип наследования и второй родительский класс:

```
#include <string>

class Human
{
private:
    std::string m_name;
    int m_age;

public:
    Human(std::string name, int age)
        : m_name(name), m_age(age)
    {
    }

    std::string getName() { return m_name; }
    int getAge() { return m_age; }
};

class Employee
{
private:
```

```
22.     std::string m_employer;
23.     double m_wage;
24.
25. public:
26.     Employee(std::string employer, double wage)
27.         : m_employer(employer), m_wage(wage)
28.     {
29.     }
30.
31.     std::string getEmployer() { return m_employer; }
32.     double getWage() { return m_wage; }
33. };
34.
35. // Класс Teacher открыто наследует свойства классов Human и Employee
36. class Teacher: public Human, public Employee
37. {
38. private:
39.      int m_teachesGrade;
40.
41. public:
42.     Teacher(std::string name, int age, std::string employer, double wage, int
    teachesGrade)
43.         : Human(name, age), Employee(employer, wage), m_teachesGrade(teachesGra
    de)
44.     {
45.     }
46. };
```

Здесь мы используем наследование типа public.

Проблемы с множественным наследованием

Хотя множественное наследование кажется простым расширением одиночного наследования, оно может привести к множеству проблем, которые могут заметно увеличить сложность программ и сделать кошмаром дальнейшую поддержку кода. Рассмотрим некоторые из подобных ситуаций.

Во-первых, может возникнуть неоднозначность, когда несколько родительских классов имеют метод с одним и тем же именем, например:

```
1. #include <iostream>
2.
3. class USBDevice
4. {
5. private:
6.     long m_id;
7.
8. public:
9.     USBDevice(long id)
10.         : m_id(id)
11.     {
12.     }
13.
14.     long getID() { return m_id; }
15. };
16.
```

```
17. class NetworkDevice
18. {
19. private:
20.     long m_id;
21. 
22. public:
23.     NetworkDevice(long id)
24.         : m_id(id)
25.     {
26.     }
27. 
28.     long getID() { return m_id; }
29. };
30. 
31. class WirelessAdapter: public USBDevice, public NetworkDevice
32. {
33. public:
34.     WirelessAdapter(long usbId, long networkId)
35.         : USBDevice(usbId), NetworkDevice(networkId)
36.     {
37.     }
38. };
39. 
40. int main()
41. {
42.     WirelessAdapter c54G(6334, 292651);
43.     std::cout << c54G.getID(); // какую версию getID() здесь следует вызывать?
44. 
45.     return 0;
46. }
```

При компиляции `c54G.getID()` компилятор смотрит, есть ли у WirelessAdapter метод getID(). Этого метода у него нет, поэтому компилятор двигается по цепочке наследования вверх и смотрит, есть ли этот метод в каком-либо из родительских классов. И здесь возникает проблема — getID() есть как у USBDevice, так и у NetworkDevice. Следовательно, вызов этого метода приведет к неоднозначности и мы получим ошибку, так как компилятор не будет знать какую версию getID() ему вызывать.

Тем не менее, есть способ обойти эту проблему. Мы можем явно указать, какую версию getID() следует вызывать:

```
1. int main()
2. {
3.     WirelessAdapter c54G(6334, 292651);
4.     std::cout << c54G.USBDevice::getID();
5. 
6.     return 0;
7. }
```

Хотя это решение довольно простое, но всё может стать намного сложнее, если наш класс будет иметь от 4 родительских классов, которые, в свою очередь, будут иметь свои родительские классы. Возможность возникновения конфликтов имен увеличивается экспоненциально с каждым добавленным родительским классом, и

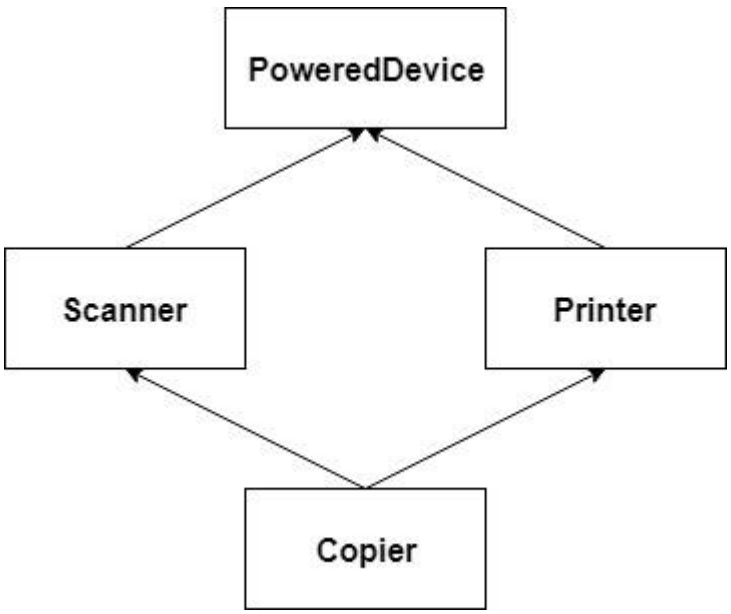
в каждом из таких случаев нужно будет явно указывать версии методов, которые следует вызывать, дабы избежать возможности возникновения конфликтов имен.

Во-вторых, более серьезной проблемой является **«алмаз смерти»** (или *«алмаз обреченности»*). Это ситуация, когда один класс имеет 2 родительских класса, каждый из которых, в свою очередь, наследует свойства одного и того же родительского класса. Иллюстративно мы получаем форму алмаза.

Например, рассмотрим следующие классы:

```
1. class PoweredDevice
2. {
3. };
4.
5. class Scanner: public PoweredDevice
6. {
7. };
8.
9. class Printer: public PoweredDevice
10.{
11.};
12.
13.class Copier: public Scanner, public Printer
14.{
15.};
```

Сканеры и принтеры - это устройства, которые получают питание от розетки, поэтому они наследуют свойства PoweredDevice. Однако ксерокс (Copier) включает в себя функции как сканеров, так и принтеров.



В этом контексте возникает много проблем, включая неоднозначность при вызове методов и копирование данных PoweredDevice в класс Copier дважды. Хотя большинство из этих проблем можно решить с помощью явного указания,

поддержка и обслуживание такого кода может привести к непредсказуемым временным затратам. Мы поговорим детально о способах решения проблемы "алмаза смерти" на соответствующем уроке.

Стоит ли использовать множественное наследование?

Большинство задач, решаемых с помощью множественного наследования, можно решить и с использованием одиночного наследования. Многие объектно-ориентированные языки программирования (например, Smalltalk, PHP) даже не поддерживают множественное наследование. Многие, относительно современные языки, такие как Java и C#, ограничивают классы одиночным наследованием обычных классов, но допускают множественное наследование интерфейсных классов. Суть идеи, запрещающей множественное наследование в этих языках, заключается в том, что это излишняя сложность, которая порождает больше проблем, чем удобств.

Многие опытные программисты считают, что множественное наследование в языке C++ следует избегать любой ценой из-за потенциальных проблем, которые могут возникнуть. Однако, все же остается вероятность, когда множественное наследование будет лучшим решением, нежели придумывание двухуровневых «костылей».

Стоит отметить, что вы сами уже использовали классы, написанные с использованием множественного наследования, даже не подозревая об этом: такие объекты, как std::cin и std::cout библиотеки iostream, реализованы с использованием множественного наследования!

Правило: Используйте множественное наследование только в крайних случаях, когда задачу нельзя решить одиночным наследованием, либо другим альтернативным способом (без изобретения "велосипеда").

Глава №11. Итоговый тест

В этой главе мы рассмотрели наследование в языке C++. Пора закрепить пройденный материал.

Теория

Наследование позволяет моделировать отношения типа «является» между двумя объектами. Объект, который наследует, называется дочерним классом. Объект, которого наследуют, называется родительским классом.

При наследовании **дочерний класс наследует все члены родительского класса**.

При инициализации объектов дочернего класса, **сначала выполняется построение родительской части объекта, а затем уже дочерней части объекта**. Рассмотрим детально:

- Сначала выделяется память для объекта дочернего класса (достаточная порция для 2-х частей, из которых состоит объект: родительская и дочерняя).
- Вызывается соответствующий конструктор дочернего класса.
- Выполняется построение родительской части с использованием соответствующего конструктора родительского класса. Если конструктор не указан, то используется конструктор по умолчанию родительского класса.
- Список инициализации дочернего класса инициализирует члены дочернего класса.
- Выполняется тело конструктора дочернего класса.
- Управление возвращается обратно в caller.

Освобождение памяти (уничтожение) происходит в порядке, противоположном построению: от дочерних до родительских классов.

Язык C++ имеет 3 спецификатора доступа: public, private и protected.
Спецификатор protected используется для разрешения доступа к членам дружественным классам/функциям и дочерним классам, всем остальным объектам — доступ закрыт.

Есть 3 типа наследования: public, private и protected. Наиболее распространенный тип наследования - public.

Таблица спецификаторов доступа и типов наследования:

Спецификатор доступа в родительском классе	Спецификатор доступа при наследовании типа public в дочернем классе	Спецификатор доступа при наследовании типа private в дочернем классе	Спецификатор доступа при наследовании типа protected в дочернем классе
public	public	private	protected
private	Недоступен	Недоступен	Недоступен
protected	protected	private	protected

Дочерние классы могут изменять методы родительского класса, добавлять свой функционал, изменять спецификатор доступа наследуемых членов или даже скрывать методы родительского класса. Всё это выполняется в теле дочернего класса.

Множественное наследование позволяет дочернему классу иметь сразу несколько родительских классов. Не рекомендуется использовать множественное наследование, если есть альтернативные решения.

Тест

Задание №1

Для каждой из следующих программ определите результат выполнения. Если программа не скомпилируется, то объясните почему. Запускать код не нужно, вы должны определить результат/ошибки программ без помощи компилятора.

а)

```
#include <iostream>

class Parent
{
public:
    Parent()
    {
        std::cout << "Parent()\n";
    }
    ~Parent()
```

```
11.     {
12.         std::cout << "~Parent()\n";
13.     }
14. };
15. 
16. class Child: public Parent
17. {
18. public:
19.     Child()
20.     {
21.         std::cout << "Child()\n";
22.     }
23.     ~Child()
24.     {
25.         std::cout << "~Child()\n";
26.     }
27. };
28. 
29. int main()
30. {
31.     Child ch;
32. }
```

b)

```
1.  #include <iostream>
2.  
3.  class Parent
4.  {
5.  public:
6.      Parent()
7.      {
8.          std::cout << "Parent()\n";
9.      }
10.     ~Parent()
11.     {
12.         std::cout << "~Parent()\n";
13.     }
14. };
15. 
16. class Child: public Parent
17. {
18. public:
19.     Child()
20.     {
21.         std::cout << "Child()\n";
22.     }
23.     ~Child()
24.     {
25.         std::cout << "~Child()\n";
26.     }
27. };
28. 
29. int main()
30. {
31.     Child ch;
32.     Parent p;
33. }
```

> ***Подсказка***: Локальные переменные уничтожаются в порядке противоположном определению.

c)

```
#include <iostream>

class Parent
{
private:
    int m_x;
public:
    Parent(int x): m_x(x)
    {
        std::cout << "Parent()\n";
    }
    ~Parent()
    {
        std::cout << "~Parent()\n";
    }

    void print() { std::cout << "Parent: " << m_x << '\n';  }
};

class Child: public Parent
{
public:
    Child(int y):  Parent(y)
    {
        std::cout << "Child()\n";
    }
    ~Child()
    {
        std::cout << "~Child()\n";
    }

    void print() { std::cout << "Child: " << m_x << '\n'; }
};

int main()
{
    Child ch(7);
    ch.print();
}
```

d)

```
#include <iostream>

class Parent
{
protected:
    int m_x;
public:
    Parent(int x): m_x(x)
    {
        std::cout << "Parent()\n";
    }
    ~Parent()
```

```
13.     {
14.         std::cout << "~Parent()\n";
15.     }
16.
17.     void print() { std::cout << "Parent: " << m_x << '\n';  }
18. };
19.
20. class Child: public Parent
21. {
22. public:
23.     Child(int y):  Parent(y)
24.     {
25.         std::cout << "Child()\n";
26.     }
27.     ~Child()
28.     {
29.         std::cout << "~Child()\n";
30.     }
31.
32.     void print() { std::cout << "Child: " << m_x << '\n'; }
33. };
34.
35. int main()
36. {
37.     Child ch(7);
38.     ch.print();
39. }
```

e)

```
1.  #include <iostream>
2.
3.  class Parent
4.  {
5.  protected:
6.      int m_x;
7.  public:
8.      Parent(int x): m_x(x)
9.      {
10.         std::cout << "Parent()\n";
11.     }
12.     ~Parent()
13.     {
14.         std::cout << "~Parent()\n";
15.     }
16.
17.     void print() { std::cout << "Parent: " << m_x << '\n';  }
18. };
19.
20. class Child: public Parent
21. {
22. public:
23.     Child(int y):  Parent(y)
24.     {
25.         std::cout << "Child()\n";
26.     }
27.     ~Child()
28.     {
29.         std::cout << "~Child()\n";
30.     }
31.
```

```
32.     void print() { std::cout << "Child: " << m_x << '\n'; }
33. };
34. 
35. class D2 : public Child
36. {
37. public:
38.     D2(int z): Child(z)
39.     {
40.         std::cout << "D2()\n";
41.     }
42.     ~D2()
43.     {
44.         std::cout << "~D2()\n";
45.     }
46. 
47.         // Обратите внимание, здесь нет метода print()
48. };
49. 
50. int main()
51. {
52.     D2 d(7);
53.     d.print();
54. }
```

Задание №2

a) Создайте классы Apple и Banana, которые наследуют класс Fruit. У класса Fruit есть две переменные-члены: `name` и `color`.

Следующий код:

```
1. int main()
2. {
3.     Apple a("red");
4.     Banana b;
5. 
6.     std::cout << "My " << a.getName() << " is " << a.getColor() << ".\n";
7.     std::cout << "My " << b.getName() << " is " << b.getColor() << ".\n";
8. 
9.     return 0;
10. }
```

Должен выдавать следующий результат:

```
My apple is red.
My banana is yellow.
```

b) Добавьте новый класс GrannySmith, который наследует класс Apple.

Следующий код:

```
1. int main()
2. {
3.     Apple a("red");
4.     Banana b;
```

```
5.      GrannySmith c;
6.
7.      std::cout << "My " << a.getName() << " is " << a.getColor() << ".\n";
8.      std::cout << "My " << b.getName() << " is " << b.getColor() << ".\n";
9.      std::cout << "My " << c.getName() << " is " << c.getColor() << ".\n";
10.
11.     return 0;
12. }
```

Должен выдавать следующий результат:

```
My apple is red.
My banana is yellow.
My Granny Smith apple is green.
```

Задание №3

Самое любимое! Будем создавать простую игру, в которой вы будете сражаться с монстрами. Цель игры — собрать максимум золота, прежде чем вы умрете или достигнете 20 уровня.

Игра состоит из 3-х классов: Creature, Player и Monster. Player и Monster наследуют класс Creature.

а) Сначала создайте класс Creature со следующими членами:

- имя (std::string);
- символ (char);
- количество здоровья (int);
- количество урона, которое он наносит врагу во время атаки (int);
- количество золота, которое он имеет (int).

Создайте полный набор геттеров (по одному на каждую переменную-член класса). Добавьте еще три метода:

- void reduceHealth(int), который уменьшает здоровье Creature на указанное целочисленное значение;
- bool isDead(), который возвращает true, если здоровье Creature равно `0` или меньше;
- void addGold(int), который добавляет золото Creature-у.

Следующий код:

```
1. #include <iostream>
2. #include <string>
3.
```

```
4. int main()
5. {
6.     Creature o("orc", 'o', 4, 2, 10);
7.     o.addGold(5);
8.     o.reduceHealth(1);
9.     std::cout << "The " << o.getName() << " has " << o.getHealth() << " health
   and is carrying " << o.getGold() << " gold.";
10.
11.    return 0;
12. }
```

Должен выдавать следующий результат:

```
The orc has 3 health and is carrying 15 gold.
```

b) Теперь нам нужно создать класс Player, который наследует Creature. Player имеет:

- переменную-член `level`, которая начинается с `1`;
- имя (пользователь вводит с клавиатуры);
- символ `@`;
- 10 очков здоровья;
- 1 очко урона (для начала);
- и 0 золота.

Напишите метод levelUp(), который увеличивает уровень Player-а и его урон на 1. Также напишите геттер для члена `level` и метод hasWon(), который возвращает true, если Player достиг 20 уровня.

Допишите в функцию main() код, который спрашивает у пользователя его имя и выводит количество его здоровья и золота:

```
Enter your name: Anton
Welcome, Anton.
You have 10 health and are carrying 0 gold.
```

c) Следующий класс Monster также наследует Creature и у него нет собственных переменных-членов. Но есть перечисление Type, которое содержит 3 перечислителя, они обозначают типы монстров: `DRAGON`, `ORC` и `SLIME` (вам также нужен дополнительный перечислитель `MAX_TYPES`).

d) Каждый тип Монстра имеет свое имя, символ, определенное количество здоровья, урона и золота:

Type	Name	Symbol	Health	Damage	Gold
DRAGON	dragon	D	20	4	100
ORC	orc	o	4	2	25
SLIME	slime	s	1	1	10

Следующий шаг — реализация конструктора класса Monster, с помощью которого можно создавать монстров. Этот конструктор должен принимать перечисление Type в качестве параметра, а затем создавать монстра с соответствующими таблице характеристиками.

Это можно реализовать по-разному. Однако, поскольку все наши свойства типов монстров предопределены (не случайны), то мы будем использовать таблицу поиска. **Таблица поиска** — это массив, который содержит все предопределенные атрибуты (свойства) чего-либо. Мы можем использовать таблицу поиска для просмотра характеристики определенного типа монстра по мере необходимости.

Как это сделать? Нам нужны всего лишь две вещи. Во-первых, массив с отдельным элементом для каждого типа монстра. Во-вторых, этот элемент будет содержать структуру, в которой будут находиться все предопределенные значения атрибутов для конкретного типа монстра.

- *Шаг №1:* Создайте структуру MonsterData внутри класса Monster. Эта структура должна иметь следующие перечислители: `name`, `symbol`, `health`, `damage` и `gold`.
- *Шаг №2:* Объявите статический массив этой структуры с именем `monsterData`.
- *Шаг №3:* Добавьте код определения нашей таблицы поиска вне тела класса:

```
1. Monster::MonsterData Monster::monsterData[Monster::MAX_TYPES]
2. {
3.     { "dragon", 'D', 20, 4, 100 },
4.     { "orc", 'o', 4, 2, 25 },
5.     { "slime", 's', 1, 1, 10 }
6. };
```

Теперь мы можем искать любые значения, которые нам нужны! Например, чтобы узнать количество золота Dragon, мы можем использовать `monsterData[DRAGON].gold`.

Используйте эту таблицу поиска для реализации вашего конструктора:

```
1. Monster(Type type): Creature(monsterData[type].name, ...)
```

Следующий код:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     Monster m(Monster::ORC);
7.     std::cout << "A " << m.getName() << " (" << m.getSymbol() << ") was created
   .\n";
8. }
```

Должен выдавать следующий результат:

```
A orc (o) was created.
```

е) Наконец, добавьте статический метод getRandomMonster() в класс Monster. Этот метод должен генерировать случайное число от `0` до `MAX_TYPES-1` и возвращать (возврат по значению) определенный тип монстра (вам нужно использовать оператор static_cast для конвертации int в Type, чтобы передать его конструктору класса Monster).

Вы можете использовать следующий код для генерации случайного числа:

```
1. #include <cstdlib> // для rand() и srand()
2. #include <ctime> // для time()
3.
4. // Генерируем рандомное число между min и max
5. int getRandomNumber(int min, int max)
6. {
7.     static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);

8.     // Равномерно распределяем генерацию значения из диапазона
9.     return static_cast<int>(rand() * fraction * (max - min + 1) + min);
10.}
```

Следующий код:

```
1. #include <iostream>
2. #include <string>
3. #include <cstdlib> // для rand() и srand()
4. #include <ctime> // для time()
5.
6. int main()
```

```
{
    srand(static_cast<unsigned int>(time(0))); // устанавливаем значение системных часов в качестве стартового числа
    rand(); // сбрасываем первый результат

    for (int i = 0; i < 10; ++i)
    {
        Monster m = Monster::getRandomMonster();
        std::cout << "A " << m.getName() << " (" << m.getSymbol() << ") was created.\n";
    }

    return 0;
}
```

Должен сгенерировать 10 рандомных монстров:

```
A slime (s) was created.
A orc (o) was created.
A slime (s) was created.
A slime (s) was created.
A orc (o) was created.
A orc (o) was created.
A dragon (D) was created.
A slime (s) was created.
A orc (o) was created.
A orc (o) was created.
```

f) Готово, теперь нам нужно разобраться с логикой выполнения нашей игры!

Суть:

- Игрок сталкивается с одним случайно выбранным монстром.
- С каждым монстром игрок может либо (`R`)un, либо (`F`)ight.
- Если игрок решает Run, то шансы на удачный побег составляют 50%.
- Если игроку удается сбежать, то он благополучно переходит к следующему монстру (его здоровье/урон/золото при этом не уменьшается).
- Если игроку не удается сбежать, то монстр его атакует. Здоровье игрока уменьшается от урона монстра. Затем игрок выбирает свое следующее действие.
- Если игрок выбирает Fight, то он атакует монстра. Здоровье монстра уменьшается от урона игрока.
- Если монстр умирает, то игрок забирает всё золото монстра + увеличивает свой `level` и урон на 1.
- Если монстр не умирает, то он атакует игрока. Здоровье игрока уменьшается от урона монстра.

- Игра заканчивается, если игрок умер (проигрыш) или достиг 20 уровня (выигрыш).
- Если игрок умирает, то программа должна сообщить игроку, какой уровень у него был и сколько золота он имел.
- Если игрок побеждает, то игра должна сообщить игроку, что он выиграл и сколько у него есть золота.

Пример игры:

```
Enter your name: Anton
Welcome, Anton
You have encountered a orc (o).
(R)un or (F)ight: r
You successfully fled.
You have encountered a slime (s).
(R)un or (F)ight: f
You hit the slime for 1 damage.
You killed the slime.
You are now level 2.
You found 10 gold.
You have encountered a dragon (D).
(R)un or (F)ight: f
You hit the dragon for 2 damage.
The dragon hit you for 4 damage.
(R)un or (F)ight: f
You hit the dragon for 2 damage.
The dragon hit you for 4 damage.
(R)un or (F)ight: f
You hit the dragon for 2 damage.
The dragon hit you for 4 damage.
You died at level 2 and with 10 gold.
Too bad you can't take it with you!
```

> ***Подсказка***: У вас должны быть следующие 4 функции:
>
> - Функция создания Игрока и основной игровой цикл (в функции main()).
> - Функция fightMonster(), которая обрабатывает бой между Игроком и Монстром, и спрашивает у игрока, что он хочет сделать: Run или Fight.
> - Функция attackMonster(), которая обрабатывает атаку монстра игроком, включая увеличение уровня игрока.
> - Функция attackPlayer(), которая обрабатывает атаку игрока монстром.

Урок №170. Указатели, Ссылки и Наследование

Из предыдущих уроков мы уже знаем, что при создании объекта дочернего класса выполняется построение 2-х частей, из которых этот объект и состоит: родительская и дочерняя. Например:

```
1. class Parent
2. {
3. protected:
4.     int m_value;
5. 
6. public:
7.     Parent(int value)
8.         : m_value(value)
9.     {
10.    }
11. 
12.    const char* getName() { return "Parent"; }
13.    int getValue() { return m_value; }
14.};
15. 
16.class Child: public Parent
17.{
18.public:
19.    Child(int value)
20.        : Parent(value)
21.    {
22.    }
23. 
24.    const char* getName() { return "Child"; }
25.    int getValueDoubled() { return m_value * 2; }
26.};
```

При создании объекта класса Child сначала выполняется построение части Parent, а затем уже части Child. Помните, что тип отношений в наследовании - «является». Поскольку Child «является» Parent, то логично, что Child содержит часть Parent.

Мы можем дать команду указателям и ссылкам класса Child указывать на другие объекты класса Child:

```
1. #include <iostream>
2. 
3. int main()
4. {
5.     Child child(7);
6.     std::cout << "child is a " << child.getName() << " and has value " << child.getValue() << '\n';
7. 
8.     Child &rChild = child;
9.     std::cout << "rChild is a " << rChild.getName() << " and has value " << rChild.getValue() << '\n';
10. 
11.    Child *pChild = &child;
```

```
12.     std::cout << "pChild is a " << pChild->getName() <<
    " and has value " << pChild->getValue() << '\n';
13.
14.     return 0;
15. }
```

Результат:

```
child is a Child and has value 7
rChild is a Child and has value 7
pChild is a Child and has value 7
```

Интересно, поскольку Child имеет часть Parent, то можем ли мы дать команду указателю или ссылке класса Parent указывать на объект класса Child? Оказывается, можем!

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      Child child(7);
6.
7.      // Всё корректно!
8.      Parent &rParent = child;
9.      Parent *pParent = &child;
10.
11.     std::cout << "child is a " << child.getName() << " and has value " <<
    child.getValue() << '\n';
12.     std::cout << "rParent is a " << rParent.getName() << " and has value " <<
    rParent.getValue() << '\n';
13.     std::cout << "pParent is a " << pParent->getName() <<
14. " and has value " << pParent->getValue() << '\n';
15.
16.     return 0;
17. }
```

Результат:

```
child is a Child and has value 7
rParent is a Parent and has value 7
pParent is a Parent and has value 7
```

Но это может быть не совсем то, что вы ожидали увидеть!

Поскольку `rParent` и `pParent` являются ссылкой и указателем класса Parent, то они могут видеть только члены класса Parent (и члены любых других классов, которые наследует Parent). Таким образом, указатель/ссылка класса Parent не может увидеть Child::getName(). Следовательно, вызывается Parent::getName(), а `rParent` и `pParent` сообщают, что они относятся к классу Parent, а не к классу Child.

Обратите внимание, это также означает, что невозможно вызвать Child::getValueDoubled() через `rParent` или `pParent`. Они не могут видеть что-либо в классе Child.

Вот еще один более сложный пример:

```
#include <iostream>
#include <string>

class Animal
{
protected:
    std::string m_name;

    // Мы делаем этот конструктор protected так как не хотим, чтобы пользователи создавали объекты класса Animal напрямую,
    // но хотим, чтобы у дочерних классов доступ был открыт
    Animal(std::string name)
        : m_name(name)
    {
    }

public:
    std::string getName() { return m_name; }
    const char* speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string name)
        : Animal(name)
    {
    }

    const char* speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string name)
        : Animal(name)
    {
    }

    const char* speak() { return "Woof"; }
};

int main()
{
    Cat cat("Matros");
    std::cout << "cat is named " << cat.getName() << ", and it says " << cat.speak() << '\n';

    Dog dog("Barsik");
    std::cout << "dog is named " << dog.getName() << ", and it says " << dog.speak() << '\n';

```

```
51.     Animal *pAnimal = &cat;
52.     std::cout << "pAnimal is named " << pAnimal-
    >getName() << ", and it says " << pAnimal->speak() << '\n';
53.
54.     pAnimal = &dog;
55.     std::cout << "pAnimal is named " << pAnimal-
    >getName() << ", and it says " << pAnimal->speak() << '\n';
56.
57.     return 0;
58. }
```

Результат выполнения программы:

```
cat is named Matros, and it says Meow
dog is named Barsik, and it says Woof
pAnimal is named Matros, and it says ???
pAnimal is named Barsik, and it says ???
```

Мы видим здесь ту же проблему. Поскольку `pAnimal` является указателем типа Animal, то он может видеть только часть Animal. Следовательно, `pAnimal->speak()` вызывает Animal::speak(), а не Dog::Speak() или Cat::speak().

Указатели, ссылки и родительские классы

Теперь вы можете сказать: «Примеры, приведенные выше, кажутся глупыми. Почему я должен использовать указатель или ссылку родительского класса на объект дочернего класса, если я могу просто использовать дочерний объект?». Оказывается, на это есть несколько веских причин.

Во-первых, предположим, что вы хотите написать функцию, которая выводит имя и звук животного. Без использования указателя на родительский класс, вам придется реализовать это через перегрузку функций. Например:

```
1. void report(Cat &cat)
2. {
3.     std::cout << cat.getName() << " says " << cat.speak() << '\n';
4. }
5.
6. void report(Dog &dog)
7. {
8.     std::cout << dog.getName() << " says " << dog.speak() << '\n';
9. }
```

Не слишком сложно, но представьте, что у нас 30 разных типов животных. Нам пришлось бы написать 30 перегрузок! Кроме того, если вы когда-либо добавите новый тип животных, то вам также придется написать новую функцию для этого типа животных. Это огромная трата времени.

Однако, поскольку Cat и Dog наследуют Animal, Cat и Dog имеют часть Animal, поэтому мы можем сделать следующее:

```
1. void report(Animal &rAnimal)
2. {
3.     std::cout << rAnimal.getName() << " says " << rAnimal.speak() << '\n';
4. }
```

Это позволит нам передавать любой класс, который является дочерним классу Animal! Вместо отдельного метода на каждый дочерний класс мы записали один метод, который работает сразу со всеми дочерними классами!

Проблема, конечно, в том, что, поскольку `rAnimal` является ссылкой класса Animal, то `rAnimal.speak()` вызовет Animal::speak() вместо метода speak() дочернего класса.

Во-вторых, допустим, у нас есть 3 кошки и 3 собаки, которых мы бы хотели сохранить в массиве для легкого доступа к ним. Поскольку массивы могут содержать объекты только одного типа, то без указателей/ссылок на родительский класс, нам бы пришлось создавать отдельный массив для каждого дочернего класса. Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     Cat cats[] = { Cat("Matros"), Cat("Ivan"), Cat("Martun") };
6.     Dog dogs[] = { Dog("Barsik"), Dog("Tolik"), Dog("Tyzik") };
7.
8.     for (int iii=0; iii < 3; ++iii)
9.         std::cout << cats[iii].getName() << " says " << cats[iii].speak() <<
   '\n';
10.
11.    for (int iii=0; iii < 3; ++iii)
12.        std::cout << dogs[iii].getName() << " says " << dogs[iii].speak() <<
   '\n';
13.
14.    return 0;
15. }
```

Теперь представьте, что у нас 30 разных типов животных. Нам пришлось бы создать 30 массивов — по одному на каждый тип животного!

Однако, поскольку Cat и Dog наследуют Animal, то можно сделать следующее:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     Cat matros("Matros"), ivan("Ivan"), martun("Martun");
6.     Dog barsik("Barsik"), tolik("Tolik"), tyzik("Tyzik");
```

```
7.
8.      // Создаем массив указателей на наши объекты Cat и Dog
9.      Animal *animals[] = { &matros, &ivan, &martun, &barsik, &tolik, &tyzik};
10.     for (int iii=0; iii < 6; ++iii)
11.         std::cout << animals[iii]->getName() << " says " << animals[iii]-
    >speak() << '\n';
12.
13.     return 0;
14. }
```

Хотя это скомпилируется и выполнится, но, к сожалению, тот факт, что каждый элемент массива `animals` является указателем на Animal, означает, что `animals[iii]->speak()` будет вызывать Animal::speak(), вместо методов speak() дочерних классов.

Хотя оба этих способа могут сэкономить нам много времени и энергии, они имеют одну и ту же проблему: указатель или ссылка родительского класса вызывает родительскую версию функции, а не дочернюю. Если бы был какой-то способ заставить родительские указатели вызывать методы дочерних классов.

Угадайте теперь, зачем нужны виртуальные функции? ☺

Тест

Наш пример с Animal/Cat/Dog не работает так, как мы хотим, потому что ссылка/указатель класса Animal не может получить доступ к методам speak() дочерних классов. Один из способов обойти эту проблему — сделать так, чтобы данные, возвращаемые методом speak(), стали доступными в виде родительской части класса Animal (так же, как `name` класса Animal доступен через член `m_name`).

Обновите классы Animal, Cat и Dog в коде, приведенном выше, добавив новый член `m_speak` в класс Animal. Инициализируйте его соответствующим образом. Следующая программа должна работать корректно:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     Cat matros("Matros"), ivan("Ivan"), martun("Martun");
6.     Dog barsik("Barsik"), tolik("Tolik"), tyzik("Tyzik");
7.
8.     // Создаем массив указателей на наши объекты Cat и Dog
9.     Animal *animals[] = { &matros, &ivan, &martun, &barsik, &tolik, &tyzik};
10.    for (int iii=0; iii < 6; iii++)
11.        std::cout << animals[iii]->getName() << " says " << animals[iii]-
    >speak() << '\n';
12.
13.    return 0;
14. }
```

Урок №171. Виртуальные функции и Полиморфизм

На предыдущем уроке мы рассматривали ряд примеров, в которых использование указателей или ссылок родительского класса упрощало логику и уменьшало количество кода. Тем не менее, мы сталкивались с проблемой, когда родительский указатель или ссылка вызывали только родительские методы, а не дочерние. Например:

```
#include <iostream>

class Parent
{
public:
    const char* getName() { return "Parent"; }
};

class Child: public Parent
{
public:
    const char* getName() { return "Child"; }
};

int main()
{
    Child child;
    Parent &rParent = child;
    std::cout << "rParent is a " << rParent.getName() << '\n';
}
```

Результат:

```
rParent is a Parent
```

Поскольку `rParent` является ссылкой класса Parent, то вызывается Parent::getName(), хотя фактически мы ссылаемся на часть Parent объекта `child`.

На этом уроке мы рассмотрим, как можно решить эту проблему с помощью виртуальных функций.

Виртуальная функция в языке C++ - это особый тип функции, которая, при её вызове, выполняет «наиболее» дочерний метод, который существует между родительским и дочерними классами. Это свойство еще известно, как **полиморфизм**. Дочерний метод вызывается тогда, когда совпадает сигнатура (имя, типы параметров и является ли метод константным) и тип возврата дочернего метода с сигнатурой и типом возврата метода родительского класса. Такие методы называются **переопределениями** (или ***"переопределенными методами"***).

Чтобы сделать функцию виртуальной, нужно просто указать **ключевое слово virtual** перед объявлением функции. Например:

```
#include <iostream>

class Parent
{
public:
    virtual const char* getName() { return "Parent"; } // добавили ключевое слово virtual
};

class Child: public Parent
{
public:
    virtual const char* getName() { return "Child"; }
};

int main()
{
    Child child;
    Parent &rParent = child;
    std::cout << "rParent is a " << rParent.getName() << '\n';

    return 0;
}
```

Результат:

```
rParent is a Child
```

Поскольку `rParent` является ссылкой на родительскую часть объекта `child`, то, обычно, при обработке `rParent.getName()` вызывался бы Parent::getName(). Тем не менее, поскольку Parent::getName() является виртуальной функцией, то компилятор понимает, что нужно посмотреть, есть ли переопределения этого метода в дочерних классах. И компилятор находит Child::getName()!

Рассмотрим пример посложнее:

```
#include <iostream>

class A
{
public:
    virtual const char* getName() { return "A"; }
};

class B: public A
{
public:
    virtual const char* getName() { return "B"; }
};

class C: public B
{
```

```
public:
    virtual const char* getName() { return "C"; }
};

class D: public C
{
public:
    virtual const char* getName() { return "D"; }
};

int main()
{
    C c;
    A &rParent = c;
    std::cout << "rParent is a " << rParent.getName() << '\n';

    return 0;
}
```

Как вы думаете, какой результат выполнения этой программы?

Рассмотрим всё по порядку:

- Сначала создается объект `c` класса C.
- `rParent` - это ссылка класса A, которой мы указываем ссылаться на часть A объекта `c`.
- Затем вызывается метод `rParent.getName()`.
- Вызов `rParent.GetName()` приводит к вызову A::getName(). Однако, поскольку A::getName() является виртуальной функцией, то компилятор ищет «наиболее» дочерний метод между A и C. В этом случае — это C::getName().

Обратите внимание, компилятор не будет вызывать D::getName(), поскольку наш исходный объект был класса C, а не класса D, поэтому рассматриваются методы только между классами A и C.

Результат выполнения программы:

```
rParent is a C
```

Более сложный пример

Рассмотрим класс Animal из предыдущего урока, добавив тестовый код:

```
#include <iostream>
#include <string>

class Animal
{
protected:
    std::string m_name;

```

```
9.      // Мы делаем этот конструктор protected так как не хотим, чтобы
   пользователи имели возможность создавать объекты класса Animal напрямую,
10.     // но хотим, чтобы в дочерних классах доступ был открыт
11.     Animal(std::string name)
12.         : m_name(name)
13.     {
14.     }
15.
16. public:
17.     std::string getName() { return m_name; }
18.     const char* speak() { return "???"; }
19. };
20.
21. class Cat: public Animal
22. {
23. public:
24.     Cat(std::string name)
25.         : Animal(name)
26.     {
27.     }
28.
29.     const char* speak() { return "Meow"; }
30. };
31.
32. class Dog: public Animal
33. {
34. public:
35.     Dog(std::string name)
36.         : Animal(name)
37.     {
38.     }
39.
40.     const char* speak() { return "Woof"; }
41. };
42.
43. void report(Animal &animal)
44. {
45.     std::cout << animal.getName() << " says " << animal.speak() << '\n';
46. }
47.
48. int main()
49. {
50.     Cat cat("Matros");
51.     Dog dog("Barsik");
52.
53.     report(cat);
54.     report(dog);
55. }
```

Результат выполнения программы:

```
Matros says ???
Barsik says ???
```

А теперь рассмотрим тот же класс, но сделав метод speak() виртуальным:

```
1. #include <iostream>
2. #include <string>
3.
```

```
4.  class Animal
5.  {
6.  protected:
7.      std::string m_name;
8.
9.      // Мы делаем этот конструктор protected так как не хотим, чтобы
    пользователи имели возможность создавать объекты класса Animal напрямую,
10.     // но хотим, чтобы в дочерних классах доступ был открыт
11.     Animal(std::string name)
12.         : m_name(name)
13.     {
14.     }
15.
16. public:
17.     std::string getName() { return m_name; }
18.     virtual const char* speak() { return "???"; }
19. };
20.
21. class Cat: public Animal
22. {
23. public:
24.     Cat(std::string name)
25.         : Animal(name)
26.     {
27.     }
28.
29.     virtual const char* speak() { return "Meow"; }
30. };
31.
32. class Dog: public Animal
33. {
34. public:
35.     Dog(std::string name)
36.         : Animal(name)
37.     {
38.     }
39.
40.     virtual const char* speak() { return "Woof"; }
41. };
42.
43. void report(Animal &animal)
44. {
45.     std::cout << animal.getName() << " says " << animal.speak() << '\n';
46. }
47.
48. int main()
49. {
50.     Cat cat("Matros");
51.     Dog dog("Barsik");
52.
53.     report(cat);
54.     report(dog);
55. }
```

Результат выполнения программы:

```
Matros says Meow
Barsik says Woof
```

Сработало!

При обработке `animal.speak()`, компилятор видит, что метод Animal::speak() является виртуальной функцией. Когда `animal` ссылается на часть Animal объекта `cat`, то компилятор просматривает все классы между Animal и Cat, чтобы найти наиболее дочерний метод speak(). И находит Cat::speak(). В случае, когда `animal` ссылается на часть Animal объекта `dog`, компилятор находит Dog::speak().

Обратите внимание, мы не сделали Animal::GetName() виртуальной функцией. Это из-за того, что GetName() никогда не переопределяется ни в одном из дочерних классов, поэтому в этом нет необходимости.

Аналогично со следующим примером с массивом животных:

```
Cat matros("Matros"), ivan("Ivan"), martun("Martun");
Dog barsik("Barsik"), tolik("Tolik"), tyzik("Tyzik");

// Создаем массив указателей на наши объекты Cat и Dog
Animal *animals[] = { &matros, &barsik, &ivan, &tolik, &martun, &tyzik};
for (int iii=0; iii < 6; ++iii)
    std::cout << animals[iii]->getName() << " says " << animals[iii]->speak() << '\n';
```

Результат:

```
Matros says Meow
Barsik says Woof
Ivan says Meow
Tolik says Woof
Martun says Meow
Tyzik says Woof
```

Несмотря на то, что эти два примера используют только классы Cat и Dog, любые другие дочерние классы также будут работать с нашей функцией report() и с массивом животных, без внесения дополнительных модификаций! Это, пожалуй, самое большое преимущество виртуальных функций - возможность структурировать код таким образом, чтобы новые дочерние классы автоматически работали со старым кодом, без необходимости внесения изменений со стороны программиста!

> ***Предупреждение***: Сигнатура виртуального метода дочернего класса должна *полностью* соответствовать сигнатуре виртуального метода родительского класса. Если у дочернего метода будет другой тип параметров, нежели у родительского, то вызываться этот метод не будет.

Использование ключевого слова virtual

Если функция отмечена как виртуальная, то все соответствующие переопределения тоже считаются виртуальными, даже если возле них явно не указано ключевое слова virtual. Однако, наличие ключевого слова virtual возле методов дочерних классов послужит полезным напоминанием о том, что эти методы являются виртуальными, а не обычными. Следовательно, полезно указывать ключевое слово virtual возле переопределений в дочерних классах, даже если это не является строго необходимым.

Типы возврата виртуальных функций

Типы возврата виртуальной функции и её переопределений должны совпадать. Рассмотрим следующий пример:

```
1. class Parent
2. {
3. public:
4.     virtual int getValue() { return 7; }
5. };
6.
7. class Child: public Parent
8. {
9. public:
10.     virtual double getValue() { return 9.68; }
11. };
```

В этом случае Child::getValue() не считается подходящим переопределением для Parent::getValue(), так как типы возвратов разные (метод Child::getValue() считается полностью отдельной функцией).

Не вызывайте виртуальные функции в теле конструкторов или деструкторов

Вот еще одна ловушка для новичков. Вы не должны вызывать виртуальные функции в теле конструкторов или деструкторов. Почему?

Помните, что при создании объекта класса Child сначала создается родительская часть этого объекта, а затем уже дочерняя? Если вы будете вызывать виртуальную функцию из конструктора класса Parent при том, что дочерняя часть создаваемого объекта еще не была создана, то вызвать дочерний метод вместо родительского будет невозможно, так как объект `child` для работы с методом класса Child еще не будет создан. В таких случаях, в языке C++ будет вызываться родительская версия метода.

Аналогичная проблема существует и с деструкторами. Если вы вызываете виртуальную функцию в теле деструктора класса Parent, то всегда будет вызываться метод класса Parent, так как дочерняя часть объекта уже будет уничтожена.

Правило: Никогда не вызывайте виртуальные функции в теле конструкторов или деструкторов.

Недостаток виртуальных функций

«Если всё так хорошо с виртуальными функциями, то почему бы не сделать все методы виртуальными?» - спросите Вы. Ответ: "Это неэффективно!". Обработка и выполнение вызова виртуального метода занимает больше времени, чем обработка и выполнение вызова обычного метода. Кроме того, компилятор также должен выделять один дополнительный указатель для каждого объекта класса, который имеет одну или несколько виртуальных функций.

Тест

Какой результат выполнения следующих программ? Не нужно запускать/выполнять следующий код, вы должны определить результат без помощи своих IDE.

а)

```
#include <iostream>

class A
{
public:
    virtual const char* getName() { return "A"; }
};

class B: public A
{
public:
    virtual const char* getName() { return "B"; }
};

class C: public B
{
public:
// Примечание: Здесь нет метода getName()
};

class D: public C
{
public:
    virtual const char* getName() { return "D"; }
};

int main()
{
```

```
    C c;
    A &rParent = c;
    std::cout << rParent.getName() << '\n';

    return 0;
}
```

b)

```
#include <iostream>

class A
{
public:
    virtual const char* getName() { return "A"; }
};

class B: public A
{
public:
    virtual const char* getName() { return "B"; }
};

class C: public B
{
public:
    virtual const char* getName() { return "C"; }
};

class D: public C
{
public:
    virtual const char* getName() { return "D"; }
};

int main()
{
    C c;
    B &rParent = c; // примечание: rParent на этот раз класса B
    std::cout << rParent.getName() << '\n';

    return 0;
}
```

c)

```
#include <iostream>

class A
{
public:
    const char* getName() { return "A"; } // примечание: Нет ключевого слова virtual
};

class B: public A
{
public:
    virtual const char* getName() { return "B"; }
};
```

```
14.
15. class C: public B
16. {
17. public:
18.     virtual const char* getName() { return "C"; }
19. };
20.
21. class D: public C
22. {
23. public:
24.     virtual const char* getName() { return "D"; }
25. };
26.
27. int main()
28. {
29.     C c;
30.     A &rParent = c;
31.     std::cout << rParent.getName() << '\n';
32.
33.     return 0;
34. }
```

d)

```
1.  #include <iostream>
2.
3.  class A
4.  {
5.  public:
6.      virtual const char* getName() { return "A"; }
7.  };
8.
9.  class B: public A
10. {
11. public:
12.     const char* getName() { return "B"; }  // примечание: Нет ключевого слова
    virtual
13. };
14.
15. class C: public B
16. {
17. public:
18.     const char* getName() { return "C"; } // примечание: Нет ключевого слова
    virtual
19. };
20.
21. class D: public C
22. {
23. public:
24.     const char* getName() { return "D"; } // примечание: Нет ключевого слова
    virtual
25. };
26.
27. int main()
28. {
29.     C c;
30.     B &rParent = c; // примечание: rParent на этот раз класса B
31.     std::cout << rParent.getName() << '\n';
32.
33.     return 0;
34. }
```

e)

```
#include <iostream>

class A
{
public:
    virtual const char* getName() const { return "A"; } // примечание: Метод является const
};

class B: public A
{
public:
    virtual const char* getName() { return "B"; }
};

class C: public B
{
public:
    virtual const char* getName() { return "C"; }
};

class D: public C
{
public:
    virtual const char* getName() { return "D"; }
};

int main()
{
    C c;
    A &rParent = c;
    std::cout << rParent.getName() << '\n';

    return 0;
}
```

f)

```
#include <iostream>

class A
{
public:
    A() { std::cout << getName(); } // обратите внимание на наличие конструктора

    virtual const char* getName() { return "A"; }
};

class B : public A
{
public:
    virtual const char* getName() { return "B"; }
};

class C : public B
{
public:
```

```
20.     virtual const char* getName() { return "C"; }
21. };
22. 
23. class D : public C
24. {
25. public:
26.     virtual const char* getName() { return "D"; }
27. };
28. 
29. int main()
30. {
31.     C c;
32. 
33.     return 0;
34. }
```

Урок №172. Модификаторы override и final

Для решения определенных проблем в наследовании в C++11 добавили два специальных модификатора: override и final. Обратите внимание, эти модификаторы не являются ключевыми словами — это обычные модификаторы, которые имеют особое значение в определенных местах использования.

Хотя final используется не часто, override же является фантастическим дополнением, которое вы должны использовать регулярно. На этом уроке мы рассмотрим оба этих модификатора, а также одно исключение из правил, когда тип возврата переопределения может не совпадать с типом возврата виртуальной функции родительского класса.

Модификатор override

Как мы уже знаем из предыдущего урока, виртуальная функция дочернего класса является переопределением, только если совпадают её сигнатура и тип возврата с сигнатурой и типом возврата виртуальной функции родительского класса. А это, в свою очередь, может привести к проблемам, когда функция, которая должна быть переопределением, на самом деле, им не является.

Рассмотрим следующий пример:

```
1. #include <iostream>
2.
3. class A
4. {
5. public:
6.     virtual const char* getName1(int x) { return "A"; }
7.     virtual const char* getName2(int x) { return "A"; }
8. };
9.
10. class B : public A
11. {
12. public:
13.     virtual const char* getName1(short int x) { return "B"; } // тип параметра short int
14.     virtual const char* getName2(int x) const { return "B"; } // метод является const
15. };
16.
17. int main()
18. {
19.     B b;
20.     A &rParent = b;
21.     std::cout << rParent.getName1(1) << '\n';
22.     std::cout << rParent.getName2(2) << '\n';
23.
24.     return 0;
25. }
```

Поскольку `rParent` — это ссылка класса A на объект `b`, то с помощью виртуальных функций мы намереваемся получить доступ к B::getName1() и к B::getName2(). Однако, поскольку в B::getName1() другой тип параметра (short int вместо int), то он не является переопределением метода A::getName1(). Более того, поскольку B::getName2() является const, а A::getName2() — нет, то B::getName2() также не считается переопределением A::getName2().

Следовательно, результат выполнения программы:

```
A
A
```

Конкретно в этом случае, поскольку A и B просто выводят свои имена, довольно легко увидеть, что что-то пошло не так, и переопределения не вызываются. Однако в более сложной программе, когда методы могут и не возвращать значения, которые выводятся на экран, найти ошибку уже будет довольно проблематично.

Для решения такого типа проблем и добавили **модификатор override** в C++11. Модификатор override может использоваться с любым методом, который должен быть переопределением. Достаточно просто указать override в том месте, где обычно указывается const (после скобок с параметрами). Если метод не переопределяет виртуальную функцию родительского класса, то компилятор выдаст ошибку:

```
1. #include <iostream>
2.
3. class A
4. {
5. public:
6.     virtual const char* getName1(int x) { return "A"; }
7.     virtual const char* getName2(int x) { return "A"; }
8.     virtual const char* getName3(int x) { return "A"; }
9. };
10.
11. class B : public A
12. {
13. public:
14.     virtual const char* getName1(short int x) override { return "B"; } // ошибка компиляции, метод не является переопределением
15.     virtual const char* getName2(int x) const override { return "B"; } // ошибка компиляции, метод не является переопределением
16.     virtual const char* getName3(int x) override { return "B"; } // всё хорошо, метод является переопределением A::getName3(int)
17.
18. };
19.
20. int main()
21. {
22.     return 0;
23. }
```

Здесь мы получим две ошибки: первая для B::getName1() и вторая для B::getName2(), так как ни один из этих методов не является переопределением виртуальных функций класса A. Метод B::getName3() является переопределением, поэтому с ним никаких проблем нет.

Использование модификатора override никак не влияет на эффективность или производительность программы, но помогает избежать непреднамеренных ошибок. Следовательно, настоятельно рекомендуется использовать модификатор override для каждого из своих переопределений.

Правило: Используйте модификатор override для каждого из своих переопределений.

Модификатор final

Могут быть случаи, когда вы не хотите, чтобы кто-то мог переопределить виртуальную функцию или наследовать определенный класс. **Модификатор final** используется именно для этого. Если пользователь пытается переопределить метод или наследовать класс с модификатором final, то компилятор выдаст ошибку.

Указывается final в том же месте, в котором и модификатор override, например:

```
1. class A
2. {
3. public:
4.     virtual const char* getName() { return "A"; }
5. };
6.
7. class B : public A
8. {
9. public:
10.     // Заметили final в конце? Это означает, что метод переопределить уже
   нельзя
11.     virtual const char* getName() override final { return "B"; } // всё хорошо,
     переопределение A::getName()
12. };
13.
14. class C : public B
15. {
16. public:
17.     virtual const char* getName() override { return "C"; } // ошибка
   компиляции: переопределение метода B::getName(), который является final
18. };
```

В этом коде метод B::getName() переопределяет метод A::getName(). Но B::getName() имеет модификатор final, это означает, что любые дальнейшие переопределения этого метода будут вызывать ошибку компиляции. И действительно, C::getName() уже не может переопределить B::getName() - компилятор выдаст ошибку.

В случае, если мы хотим запретить наследование определенного класса, то модификатор final указывается после имени класса:

```
class A
{
public:
    virtual const char* getName() { return "A"; }
};

class B final : public A // обратите внимание на модификатор final здесь
{
public:
    virtual const char* getName() override { return "B"; }
};

class C : public B // ошибка компиляции: нельзя наследовать класс final
{
public:
    virtual const char* getName() override { return "C"; }
};
```

В этом примере класс B объявлен как final. Таким образом, класс C не может наследовать класс B - компилятор выдаст ошибку.

Ковариантный тип возврата

Есть один случай, когда тип возврата переопределения может не совпадать с типом возврата виртуальной функции родительского класса, но при этом оставаться переопределением. Если типом возврата виртуальной функции является указатель или ссылка на класс, то переопределения могут возвращать указатель или ссылку на свой собственный класс (т.е. вместо родительского класса указывать на дочерний класс). Это называется **ковариантным типом возврата**. Например:

```
#include <iostream>

class Parent
{
public:
    // Этот метод getThis() возвращает указатель на класс Parent
    virtual Parent* getThis() { std::cout << "called Parent::getThis()\n"; return this; }
    void printType() { std::cout << "returned a Parent\n"; }
};

class Child : public Parent
{
public:
    // Обычно, типы возврата переопределений и виртуальных функций родительского класса должны совпадать.
    // Однако, поскольку Child наследует класс Parent, следующий метод может возвращать Child* вместо Parent*
    virtual Child* getThis() { std::cout << "called Child::getThis()\n"; return this; }
    void printType() { std::cout << "returned a Child\n"; }
```

```
};

int main()
{
    Child ch;
    Parent *p = &ch;
    ch.getThis()->printType(); // вызывается Child::getThis(), возвращается Child*, вызывается Child::printType
    p->getThis()->printType(); // вызывается Child::getThis(), возвращается Parent*, вызывается Parent::printType
}
```

Результат выполнения программы:

```
called Child::getThis()
returned a Child
called Child::getThis()
returned a Parent
```

Некоторые старые компиляторы могут не поддерживать ковариантные типы возврата.

В примере, приведенном выше, мы сначала вызываем `ch.getThis()`. Поскольку `ch` является объектом класса Child, то вызывается Child::getThis(), который возвращает Child*. Этот Child* затем используется для вызова не виртуальной функции Child::printType().

Затем выполняется `p->getThis()`. Переменная `p` является указателем класса Parent на объект `ch` класса Child. Parent::getThis() - это виртуальная функция, поэтому вызывается переопределение Child::getThis(). Хотя Child::getThis() и возвращает Child*, но, поскольку родительская часть объекта возвращает Parent*, возвращаемый Child* преобразовывается в Parent*. И, таким образом, вызывается Parent::printType().

Другими словами, в вышеприведенном примере мы получим Child* только в том случае, если будем вызывать getThis() с объектом класса Child.

Урок №173. Виртуальные деструкторы и Виртуальное присваивание

Хотя язык C++ автоматически предоставляет деструкторы для ваших классов, если вы не предоставляете их самостоятельно, все же иногда вы можете сделать это сами. **При работе с наследованием ваши деструкторы всегда должны быть виртуальными**. Рассмотрим следующий пример:

```
1. #include <iostream>
2.
3. class Parent
4. {
5. public:
6.     ~Parent() // примечание: Деструктор не виртуальный
7.     {
8.         std::cout << "Calling ~Parent()" << std::endl;
9.     }
10. };
11.
12. class Child: public Parent
13. {
14. private:
15.     int* m_array;
16.
17. public:
18.     Child(int length)
19.     {
20.         m_array = new int[length];
21.     }
22.
23.     ~Child() // примечание: Деструктор не виртуальный
24.     {
25.         std::cout << "Calling ~Child()" << std::endl;
26.         delete[] m_array;
27.     }
28. };
29.
30. int main()
31. {
32.     Child *child = new Child(7);
33.     Parent *parent = child;
34.     delete parent;
35.
36.     return 0;
37. }
```

Поскольку `parent` является указателем класса Parent, то при его уничтожении компилятор будет смотреть, является ли деструктор класса Parent виртуальным. Поскольку это не так, то компилятор вызовет только деструктор класса Parent.

Результат выполнения программы:

```
Calling ~Parent()
```

Тем не менее, нам нужно, чтобы delete вызывал деструктор класса Child (который, в свою очередь, будет вызывать деструктор класса Parent), иначе `m_array` не будет удален. Это можно выполнить, сделав деструктор класса Parent виртуальным:

```
#include <iostream>

class Parent
{
public:
    virtual ~Parent() // примечание: Деструктор виртуальный
    {
        std::cout << "Calling ~Parent()" << std::endl;
    }
};

class Child: public Parent
{
private:
    int* m_array;

public:
    Child(int length)
    {
        m_array = new int[length];
    }

    virtual ~Child() // примечание: Деструктор виртуальный
    {
        std::cout << "Calling ~Child()" << std::endl;
        delete[] m_array;
    }
};

int main()
{
    Child *child = new Child(7);
    Parent *parent = child;
    delete parent;

    return 0;
}
```

Результат выполнения программы:

```
Calling ~Child()
Calling ~Parent()
```

Правило: При работе с наследованием ваши деструкторы должны быть виртуальными.

Виртуальное присваивание

Оператор присваивания можно сделать виртуальным. Однако, в отличие от деструктора, виртуальное присваивание не всегда является хорошей идеей.

Почему? Это уже выходит за рамки этого урока. Следовательно, для сохранения простоты в вашем коде, не рекомендуется использовать виртуальное присваивание.

Игнорирование виртуальных функций

В языке C++ мы можем игнорировать вызов переопределений. Например:

```
1. class Parent
2. {
3. public:
4.     virtual const char* getName() { return "Parent"; }
5. };
6.
7. class Child: public Parent
8. {
9. public:
10.     virtual const char* getName() { return "Child"; }
11. };
```

Здесь мы хотим, чтобы ссылка класса Parent на объект класса Child вызывала Parent::getName() вместо Child::getName(). Чтобы это сделать, нужно просто использовать оператор разрешения области видимости:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     Child child;
6.     Parent &parent = child;
7.     // Вызов Parent::GetName() вместо переопределения Child::GetName()
8.     std::cout << parent.Parent::getName() << std::endl;
9. }
```

Вы, скорее всего, не будете использовать это очень часто, но знать об этом стоит.

Урок №174. Раннее и Позднее Связывания

Как мы уже знаем из предыдущих уроков, выполнение программы в языке C++ происходит последовательно, строка за строкой, начиная с функции main(). Когда компилятор встречает вызов функции, то точка выполнения переходит к началу кода вызываемой функции. Откуда компилятор знает, что это нужно сделать?

При компиляции программы компилятор конвертирует каждый стейтмент программы в одну или несколько строк машинного кода. Каждой строке машинного кода присваивается собственный уникальный адрес. Так же и с функциями: когда компилятор встречает функцию, она конвертируется в машинный код и получает свой адрес.

Связывание — это процесс, который используется для конвертации идентификаторов (таких как имена переменных или функций) в адреса. Хотя связывание используется как для переменных, так и для функций, на этом уроке мы сосредоточимся только на функциях.

Раннее связывание

Большинство вызовов функций, которые встречает компилятор, являются прямыми вызовами функций. **Прямой вызов функции** - это стейтмент, который напрямую вызывает функцию. Например:

```
1. #include <iostream>
2.
3. void printValue(int value)
4. {
5.     std::cout << value;
6. }
7.
8. int main()
9. {
10.    printValue(7); // это прямой вызов функции
11.    return 0;
12.}
```

Прямые вызовы функций выполняются с помощью раннего связывания. **Раннее связывание** (или ***«статическая привязка»***) означает, что компилятор (или линкер) может напрямую связать имя идентификатора (например, имя функции или переменной) с машинным адресом. Помните, что все функции имеют свой уникальный адрес. Поэтому, когда компилятор (или линкер) встречает вызов функции, он заменяет его инструкцией машинного кода, которая сообщает процессору перейти к адресу функции.

Рассмотрим простую программу-калькулятор, в которой используется раннее связывание:

```
#include <iostream>

int add(int a, int b)
{
    return a + b;
}

int subtract(int a, int b)
{
    return a - b;
}

int multiply(int a, int b)
{
    return a * b;
}

int main()
{
    int a;
    std::cout << "Enter a number: ";
    std::cin >> a;

    int b;
    std::cout << "Enter another number: ";
    std::cin >> b;

    int op;
    do
    {
        std::cout << "Enter an operation (0 = add, 1 = subtract, 2 = multiply): ";
        std::cin >> op;
    } while (op < 0 || op > 2);

    int result = 0;
    switch (op)
    {
        // Вызываем конкретную функцию напрямую. Используется раннее связывание
        case 0: result = add(a, b); break;
        case 1: result = subtract(a, b); break;
        case 2: result = multiply(a, b); break;
    }

    std::cout << "The answer is: " << result << std::endl;

    return 0;
}
```

Поскольку `add(a, b)`, `subtract(a, b)` и `multiply(a, b)` являются прямыми вызовами функций, то компилятор будет использовать раннее связывание. Он заменит вызов `add(a, b)` инструкцией, которая сообщит процессору перейти к адресу add(). То же самое выполнится и для `subtract(a, b)`, и для `multiply(a, b)`.

Позднее связывание

В некоторых программах невозможно знать наперёд, какая функция будет вызываться первой. В таком случае используется **позднее связывание** (или ***«динамическая привязка»***). В языке C++ для выполнения позднего связывания используются указатели на функции. Вкратце, указатель на функцию - это тип указателя, который указывает на функцию вместо переменной. Функция, на которую указывает указатель, может быть вызвана через указатель и оператор вызова функции. Например, вызовем функцию add():

```
#include <iostream>

int add(int a, int b)
{
    return a + b;
}

int main()
{
    // Создаем указатель на функцию add
    int (*pFcn)(int, int) = add;
    std::cout << pFcn(4, 5) << std::endl; // вызов add(4 + 5)

    return 0;
}
```

Вызов функции через указатель на функцию также известен как **непрямой** (или ***"косвенный"***) **вызов функции**. Следующая программа-калькулятор идентична вышеприведенной программе, за исключением того, что вместо прямых вызовов функций используется указатель на функцию:

```
#include <iostream>

int add(int a, int b)
{
    return a + b;
}

int subtract(int a, int b)
{
    return a - b;
}

int multiply(int a, int b)
{
    return a * b;
}

int main()
{
    int a;
    std::cout << "Enter a number: ";
    std::cin >> a;

```

```
    int b;
    std::cout << "Enter another number: ";
    std::cin >> b;

    int op;
    do
    {
        std::cout << "Enter an operation (0 = add, 1 = subtract, 2 = multiply):
";
        std::cin >> op;
    } while (op < 0 || op > 2);

    // Создаем указатель на функцию с именем pFcn (согласен, синтаксис ужасен)
    int (*pFcn)(int, int) = nullptr;

    // Указываем pFcn указывать на функцию, которую выберет пользователь
    switch (op)
    {
        case 0: pFcn = add; break;
        case 1: pFcn = subtract; break;
        case 2: pFcn = multiply; break;
    }

    // Вызываем функцию, на которую указывает pFcn с параметрами a и b.
    // Используется позднее связывание
    std::cout << "The answer is: " << pFcn(a, b) << std::endl;

    return 0;
}
```

Здесь мы указываем `pFcn` указывать на функцию, которую выберет пользователь. Затем мы вызываем через указатель функцию, которую выбрал пользователь. Компилятор не может использовать раннее связывание для выполнения вызова функции `pFcn(a, b)`, так как он не может наперёд определить, на какую функцию `pFcn` будет указывать!

Позднее связывание менее эффективное, так как присутствует "посредник" между процессором и функцией. С ранним связыванием процессор может перейти непосредственно к адресу функции. С поздним связыванием процессор должен прочитать адрес, хранящийся в указателе, а затем только перейти к этому адресу. Этот дополнительный шаг и замедляет весь процесс. Однако преимущество позднего связывания заключается в том, что оно более гибкое, нежели раннее связывание, так как не нужно решать, какую функцию следует вызывать до, собственно, запуска самой программы.

На следующем уроке мы рассмотрим, как позднее связывание используется для реализации виртуальных функций.

Урок №175. Виртуальные таблицы

Для реализации виртуальных функций язык C++ использует специальную форму позднего связывания - виртуальные таблицы.

Виртуальные таблицы

Виртуальная таблица в языке C++ — это таблица поиска функций для выполнения вызовов функций в режиме позднего (динамического) связывания. Виртуальную таблицу еще называют ***«vtable»***, ***«таблицей виртуальных функций»*** или ***«таблицей виртуальных методов»***.

Виртуальная таблица на самом деле довольно-таки проста, хотя её сложно описать словами.

Во-первых, любой класс, который использует виртуальные функции (или дочерний класс, родительский класс которого использует виртуальные функции), имеет свою собственную виртуальную таблицу. Это обычный статический массив, который создается компилятором во время компиляции. Виртуальная таблица содержит по одной записи на каждую виртуальную функцию, которая может быть вызвана объектами класса. Каждая запись в этой таблице - это указатель на функцию, указывающий на наиболее дочерний метод, доступный объекту этого класса.

Во-вторых, компилятор также **добавляет скрытый указатель на родительский класс, который мы будем называть** `*__vptr`. Этот указатель автоматически создается при создании объекта класса и указывает на виртуальную таблицу этого класса. В отличие от скрытого указателя *this, который фактически является параметром функции, используемым компилятором для "указания на самого себя", `*__vptr` является реальным указателем. Следовательно, размер каждого объекта увеличивается на размер этого указателя. `*__vptr` также наследуется дочерними классами.

Сейчас вы, скорее всего, немного удивлены и задаетесь вопросом: «Как это всё вместе работает?». Поэтому давайте рассмотрим следующий простой пример:

```
1. class Parent
2. {
3. public:
4.     virtual void function1() {};
5.     virtual void function2() {};
6. };
7.
8. class C1: public Parent
9. {
```

```
public:
    virtual void function1() {};
};

class C2: public Parent
{
public:
    virtual void function2() {};
};
```

Здесь у нас есть 3 класса, соответственно, компилятор создаст 3 виртуальные таблицы: одна для Parent, одна для C1 и одна для C2.

Компилятор также добавит скрытый указатель на главный родительский класс с виртуальными функциями. Хотя компилятор делает это автоматически, мы покажем, где этот указатель добавляется:

```
class Parent
{
public:
    FunctionPointer *__vptr; // здесь
    virtual void function1() {};
    virtual void function2() {};
};

class C1: public Parent
{
public:
    virtual void function1() {};
};

class C2: public Parent
{
public:
    virtual void function2() {};
};
```

При создании объектов классов Parent, C1 или C2, *__vptr будет указывать на виртуальную таблицу класса Parent, C1 или C2 (соответственно).

Как заполняются виртуальные таблицы?

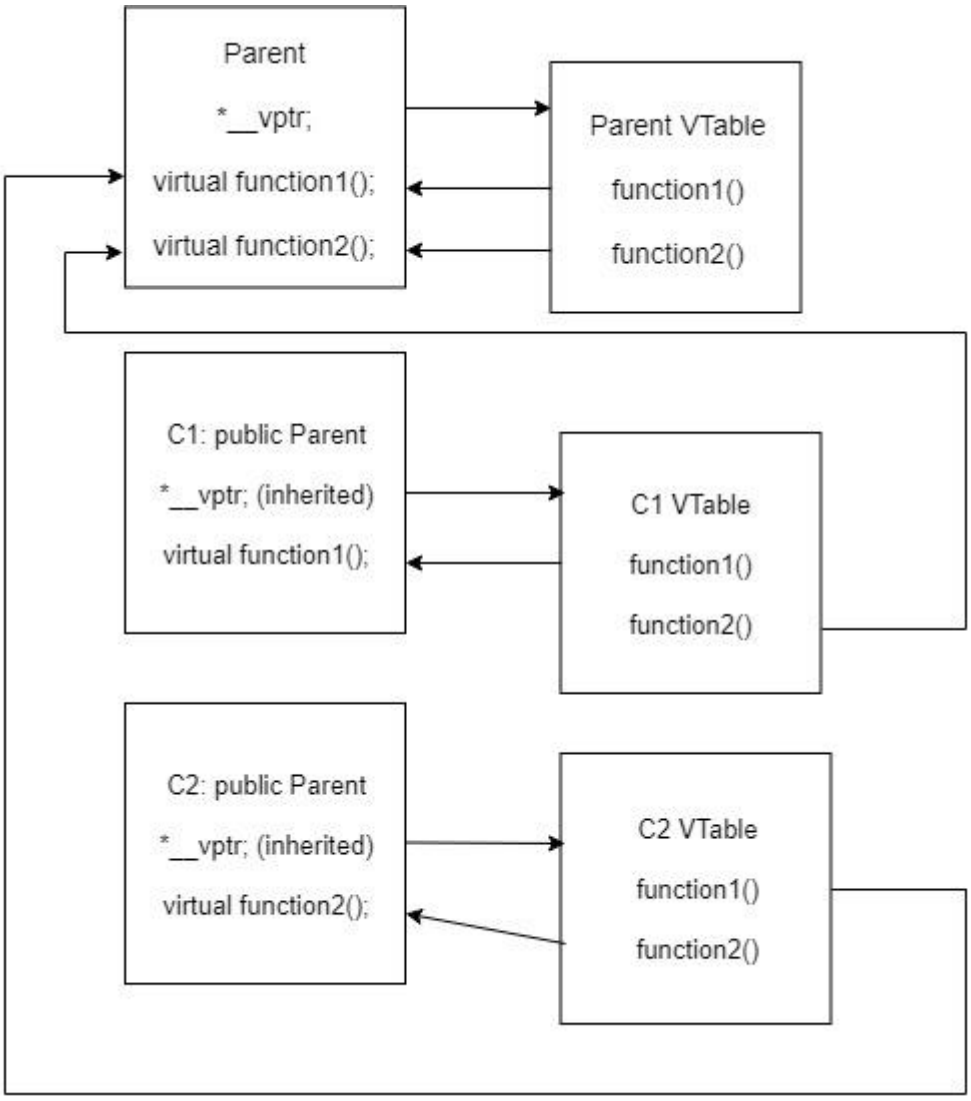
В примере, приведенном выше, у нас есть только две виртуальные функции, поэтому каждая виртуальная таблица будет иметь две записи (одна для function1() и одна для function2()). Помните, что при заполнении виртуальных таблиц выбираются наиболее дочерние методы, доступ к которым имеют объекты.

Виртуальная таблица для объектов класса Parent проста. Объект класса Parent имеет доступ только к членам класса Parent, он не имеет доступ к членам классов C1 и C2. Следовательно, запись function1 будет указывать на Parent::function1(), а запись function2 будет указывать на Parent::function2().

Виртуальная таблица для C1 уже немного сложнее. Объект класса C1 имеет доступ как к членам C1, так и к членам Parent. Однако C1 имеет переопределение function1(), что делает C1::function1() более дочерним методом, нежели Parent::function1(). Следовательно, запись function1 будет указывать на C1::function1(). C1 не переопределяет function2(), поэтому запись function2 остается указывать на Parent::function2().

В виртуальной таблице для C2 запись function1 будет указывать на Parent::function1(), а запись function2 будет указывать на C2::function2().

Смотрим:



Хотя здесь уже можно удивиться во второй раз, всё, на самом деле, очень просто: `*__vptr` каждого класса указывает на виртуальную таблицу этого же класса. Записи в виртуальной таблице указывают на наиболее дочерние методы (переопределения), доступ к которым имеют объекты.

Рассмотрим, что произойдет при создании объекта класса C1:

```
1. int main()
2. {
3.     C1 c1;
4. }
```

Поскольку `c1` является объектом класса C1, то он имеет свой `*__vptr`, который указывает на виртуальную таблицу класса C1.

Теперь создадим указатель класса Parent на объект `c1`:

```
1. int main()
2. {
3.     C1 c1;
4.     Parent *cPtr = &c1;
5. }
```

Поскольку `cPtr` является указателем класса Parent, то он указывает только на часть Parent объекта `c1`. Однако `*__vptr` тоже находится в части Parent, поэтому `cPtr` имеет доступ к этому указателю. Наконец, `cPtr->__vptr` будет указывать на виртуальную таблицу C1, поскольку `cPtr` указывает на объект класса C1! Даже если `cPtr` является указателем класса Parent, он все равно имеет доступ к виртуальной таблице C1.

Поэтому, что произойдет, если мы попытаемся вызвать `cPtr->function1()`?

```
1. int main()
2. {
3.     C1 c1;
4.     Parent *cPtr = &c1;
5.     cPtr->function1();
6. }
```

Во-первых, компилятор распознает, что function1() является виртуальной функцией. Во-вторых, он будет использовать `cPtr->__vptr` для перехода к виртуальной таблице C1. В-третьих, он будет искать, какую версию function1() вызывать в виртуальной таблице C1. Он найдет C1::function1(). Следовательно, `cPtr->function1()` будет вызывать C1::function1()!

Теперь вы можете спросить: «А если бы `cPtr` указывал на объект класса Parent вместо объекта класса C1? Вызывал бы он по-прежнему C1::function1()?». Ответ: "Нет, не вызывал бы!".

```
1. int main()
2. {
3.     Parent p;
4.     Parent *pPtr = &p;
5.     pPtr->function1();
```

```
6. }
```

В этом случае, при создании объекта `p`, `*__vptr` указывает на виртуальную таблицу класса Parent вместо C1. Следовательно, `pPtr->__vptr` также будет указывать на виртуальную таблицу класса Parent. Запись function1() в виртуальной таблице класса Parent будет указывать на Parent::function1(). Таким образом, `pPtr->function1()` будет вызывать Parent::function1(), который является наиболее дочерним методом, доступ к которому имеет объект `p`.

С помощью виртуальных таблиц компилятор и программа могут гарантировать, что вызовы функций будут вызывать соответствующие виртуальные функции/переопределения, даже если вы будете использовать только указатель или ссылку на родительский класс!

Вызов виртуальной функции происходит медленнее, чем вызов не виртуальной функции из-за следующего:

- Во-первых, мы должны использовать `*__vptr` для перехода к соответствующей виртуальной таблице.
- Во-вторых, мы должны индексировать виртуальную таблицу, чтобы найти правильную функцию для вызова.
- И только теперь мы сможем выполнить вызов функции.

В результате мы делаем 3 операции, чтобы вызвать функцию, в отличие от двух операций для обычного непрямого вызова функции или одной операции для прямого вызова функции. Однако для современных компьютеров затраченное дополнительное время не является значительным.

Заключение

Любой класс, который использует виртуальные функции, имеет свой `*__vptr`, и размер каждого объекта этого класса увеличивается на размер этого указателя. Виртуальные функции мощные, но цена этому - производительность.

Урок №176. Чистые виртуальные функции, Интерфейсы и Абстрактные классы

До этого момента мы записывали определения всех наших виртуальных функций. Однако C++ позволяет создавать особый вид виртуальных функций, так называемых **чистых виртуальных функций** (или ***«абстрактных функций»***), которые вообще не имеют определения! Переопределяют их дочерние классы.

При создании чистой виртуальной функции, вместо определения (написания тела) виртуальной функции, мы просто присваиваем ей значение `0`.

```
1. class Parent
2. {
3. public:
4.     const char* sayHi() { return "Hi"; } // обычная не виртуальная функция
5.
6.     virtual const char* getName() { return "Parent"; } // обычная виртуальная
   функция
7.
8.     virtual int getValue() = 0; // чистая виртуальная функция
9.
10.    int doSomething() = 0; // ошибка компиляции: нельзя присвоить не
   виртуальным функциям значение 0
11. };
```

Таким образом, мы сообщаем компилятору: «Реализацией этой функции займутся дочерние классы».

Использование чистой виртуальной функции имеет два основных последствия. Во-первых, любой класс с одной и более чистыми виртуальными функциями становится **абстрактным классом**, объекты которого создавать нельзя! Подумайте, что произойдет, если мы создадим объект класса Parent:

```
1. int main()
2. {
3.     Parent parent; // мы не можем создавать объекты абстрактного класса, но,
   ради эксперимента, представьте, что это возможно
4.     parent.getValue(); // какой результат выполнения этой строки кода?
5. }
```

Поскольку мы не определяли метод getValue(), то какой результат выполнения `parent.getValue()`?

Во-вторых, все дочерние классы абстрактного родительского класса должны переопределять все чистые виртуальные функции, в противном случае — они также будут считаться абстрактными классами.

Пример чистой виртуальной функции

Рассмотрим пример чистой виртуальной функции на практике. На одном из предыдущих уроков мы создавали родительский класс Animal и дочерние классы Cat и Dog:

```
1. #include <iostream>
2. #include <string>
3.
4. class Animal
5. {
6. protected:
7.     std::string m_name;
8.
9.     // Мы сделали этот конструктор protected так как не хотим, чтобы
   пользователи могли создавать объекты класса Animal напрямую,
10.    // но хотим, чтобы эта возможность оставалась в дочерних классах
11.    Animal(std::string name)
12.        : m_name(name)
13.    {
14.    }
15.
16.public:
17.    std::string getName() { return m_name; }
18.    virtual const char* speak() { return "???"; }
19.};
20.
21.class Cat: public Animal
22.{
23.public:
24.    Cat(std::string name)
25.        : Animal(name)
26.    {
27.    }
28.
29.    virtual const char* speak() { return "Meow"; }
30.};
31.
32.class Dog: public Animal
33.{
34.public:
35.    Dog(std::string name)
36.        : Animal(name)
37.    {
38.    }
39.
40.    virtual const char* speak() { return "Woof"; }
41.};
```

Мы запретили создавать объекты класса Animal, сделав конструктор protected. Однако, остаются две проблемы:

- Конструктор по-прежнему доступен дочерним классам, что позволяет создавать объекты класса Animal.

- По-прежнему могут быть дочерние классы, которые не переопределяют метод speak().

Например:

```
1. class Lion: public Animal
2. {
3. public:
4.     Lion(std::string name)
5.         : Animal(name)
6.     {
7.     }
8.
9.     // Мы забыли переопределить метод speak()
10.};
11.
12.int main()
13.{
14.    Lion lion("John");
15.    std::cout << lion.getName() << " says " << lion.speak() << '\n';
16.}
```

Результат выполнения программы:

```
John says ???
```

Что случилось? Мы забыли переопределить метод speak(), поэтому `lion.Speak()` вызвал `Animal.speak()` и получили то, что получили.

Решение - использовать чистую виртуальную функцию:

```
1. #include <iostream>
2. #include <string>
3.
4. class Animal // этот Animal является абстрактным родительским классом
5. {
6. protected:
7.     std::string m_name;
8.
9. public:
10.    Animal(std::string name)
11.        : m_name(name)
12.    {
13.    }
14.
15.    std::string getName() { return m_name; }
16.    virtual const char* speak() = 0; // обратите внимание, speak() является
   чистой виртуальной функцией
17.};
```

Здесь есть несколько вещей, на которые следует обратить внимание. Во-первых, speak() теперь является чистой виртуальной функцией. Это означает, что Animal теперь абстрактный родительский класс, и нам уже не нужен спецификатор protected (хотя он и не будет лишним). Во-вторых, поскольку наш класс Lion

является дочерним классу Animal, но мы не определили Lion::speak(), то Lion считается также абстрактным классом. Поэтому, если мы попытаемся скомпилировать следующий код:

```
1. class Lion: public Animal
2. {
3. public:
4.     Lion(std::string name)
5.         : Animal(name)
6.     {
7.     }
8.
9.     // Мы забыли переопределить метод speak()
10.};
11.
12.int main()
13.{
14.    Lion lion("John");
15.    std::cout << lion.getName() << " says " << lion.speak() << '\n';
16.}
```

То получим ошибку, сообщающую о том, что Lion является абстрактным классом, а создавать объекты абстрактного класса нельзя. Из этого можно сделать вывод, что для того, чтобы создать объект класса Lion, нам нужно переопределить метод speak():

```
1. class Lion: public Animal
2. {
3. public:
4.     Lion(std::string name)
5.         : Animal(name)
6.     {
7.     }
8.
9.     virtual const char* speak() { return "RAWRR!"; }
10.};
11.
12.int main()
13.{
14.    Lion lion("John");
15.    std::cout << lion.getName() << " says " << lion.speak() << '\n';
16.}
```

Теперь уже другое дело:

```
John says RAWRR!
```

Чистая виртуальная функция полезна, когда у нас есть функция, которую мы хотим поместить в родительский класс, но реализацию оставить дочерним классам. Чистая виртуальная функция абстрактного родительского класса вынуждает дочерние классы переопределить эту функцию, иначе объекты этих классов создавать будет невозможно.

Чистые виртуальные функции с определениями

Оказывается, мы можем определить чистые виртуальные функции:

```
#include <iostream>
#include <string>

class Animal // это абстрактный родительский класс
{
protected:
    std::string m_name;

public:
    Animal(std::string name)
        : m_name(name)
    {
    }

    std::string getName() { return m_name; }
    virtual const char* speak() = 0; // присваивание значения "= 0" говорит о том, что эта функция является чистой виртуальной функцией
};

const char* Animal::speak()  // несмотря на то, что вот здесь находится её определение
{
    return "buzz";
}
```

В этом случае speak() по-прежнему считается чистой виртуальной функцией (хотя позже мы её определили), а Animal по-прежнему считается абстрактным родительским классом (и, следовательно, объекты этого класса не могут быть созданы). Любой класс, который наследует класс Animal, должен переопределить метод speak() или он также будет считаться абстрактным классом.

При определении чистой виртуальной функции, её тело (определение) должно быть записано отдельно (не встроено).

Это полезно, когда вы хотите, чтобы дочерние классы имели возможность переопределять виртуальную функцию или оставить её реализацию по умолчанию (которую предоставляет родительский класс). В случае, если дочерний класс доволен реализацией по умолчанию, он может просто вызвать её напрямую. Например:

```
#include <iostream>
#include <string>

class Animal // это абстрактный родительский класс
{
protected:
    std::string m_name;

public:
```

```
    Animal(std::string name)
        : m_name(name)
    {
    }

    std::string getName() { return m_name; }
    virtual const char* speak() = 0; // обратите внимание, speak() является чистой виртуальной функцией
};

const char* Animal::speak()
{
    return "buzz"; // реализация по умолчанию
}

class Dragonfly: public Animal
{

public:
    Dragonfly(std::string name)
        : Animal(name)
    {
    }

    virtual const char* speak() // этот класс уже не является абстрактным, так как мы переопределили функцию speak()
    {
        return Animal::speak(); // используется реализация по умолчанию класса Animal
    }
};

int main()
{
    Dragonfly dfly("Barbara");
    std::cout << dfly.getName() << " says " << dfly.speak() << '\n';
}
```

Результат выполнения программы:

```
Barbara says buzz
```

Интерфейсы

Интерфейс - это класс, который не имеет переменных-членов и все методы которого являются чистыми виртуальными функциями! Интерфейсы еще называют ***«классами-интерфейсами»*** или ***«интерфейсными классами»***.

Интерфейсные классы принято называть с `I` в начале, например:

```
class IErrorLog
{
public:
    virtual bool openLog(const char *filename) = 0;
    virtual bool closeLog() = 0;

    virtual bool writeError(const char *errorMessage) = 0;
```

```
8.
9.      virtual ~IErrorLog() {}; // создаем виртуальный деструктор, чтобы вызывался
    соответствующий деструктор дочернего класса в случае, если удалим указатель
    на IErrorLog
10. };
```

Любой класс, который наследует IErrorLog, должен предоставить свою реализацию всех 3-х методов класса IErrorLog. Вы можете создать дочерний класс с именем FileErrorLog, где openLog() открывает файл на диске, closeLog() — закрывает файл, а writeError() — записывает сообщение в файл. Вы можете создать еще один дочерний класс с именем ScreenErrorLog, где openLog() и closeLog() ничего не делают, а writeError() выводит сообщение во всплывающем окне.

Теперь предположим, что вам нужно написать программу, которая использует журнал ошибок. Если вы будете писать классы FileErrorLog или ScreenErrorLog напрямую, то это не эффективно. Например, следующая функция заставляет все объекты, вызывающие mySqrt(), использовать FileErrorLog, что может быть не всегда уместно:

```
1. #include <cmath> // для sqrt()
2.
3. double mySqrt(double value, FileErrorLog &log)
4. {
5.     if (value < 0.0)
6.     {
7.         log.writeError("Tried to take square root of value less than 0");
8.         return 0.0;
9.     }
10.    else
11.        return sqrt(value);
12. }
```

Намного лучшим вариантом будет реализация через IErrorLog:

```
1. #include <cmath> // для sqrt()
2.
3. double mySqrt(double value, IErrorLog &log)
4. {
5.     if (value < 0.0)
6.     {
7.         log.writeError("Tried to take square root of value less than 0");
8.         return 0.0;
9.     }
10.    else
11.        return sqrt(value);
12. }
```

Теперь пользователь через передачу объектов может определить самостоятельно, какой класс следует вызывать. Если он хочет, чтобы ошибка была записана в файле, то он передаст в функцию mySqrt() объект класса FileErrorLog. Если он хочет, чтобы ошибка выводилась на экран, то он передаст объект класса ScreenErrorLog. Или,

если он хочет сделать то, что вы не предусмотрели, например, отправить кому-то Email-ом сообщение ошибки, то он может создать новый дочерний класс EmailErrorLog, который будет наследовать IErrorLog, и передавать объект этого класса! Таким образом, реализация через IErrorLog делает нашу функцию более гибкой и независимой.

Не забудьте о подключении виртуальных деструкторов в ваши интерфейсные классы, чтобы при удалении указателя на интерфейс вызывался деструктор соответствующего (дочернего) класса.

Интерфейсы чрезвычайно популярны, так как они просты в использовании, удобны в поддержке, и их функционал легко расширять. Некоторые языки, такие как Java и C#, даже добавили в свой синтаксис **ключевое слово interface**, которое позволяет программистам напрямую определять интерфейсный класс, не указывая явно, что все методы являются абстрактными.

Чистые виртуальные функции и виртуальная таблица

Абстрактные классы имеют виртуальные таблицы, которые могут использоваться, если у вас есть указатель или ссылка на абстрактный класс. Запись чистой виртуальной функции в виртуальной таблице обычно содержит либо нулевой указатель, либо указывает на общую функцию, которая выводит ошибку (иногда эта функция называется `__purecall`), если не было обнаружено переопределения.

Тест

Чем отличается абстрактный класс от интерфейса в языке C++?

Урок №177. Виртуальный базовый класс

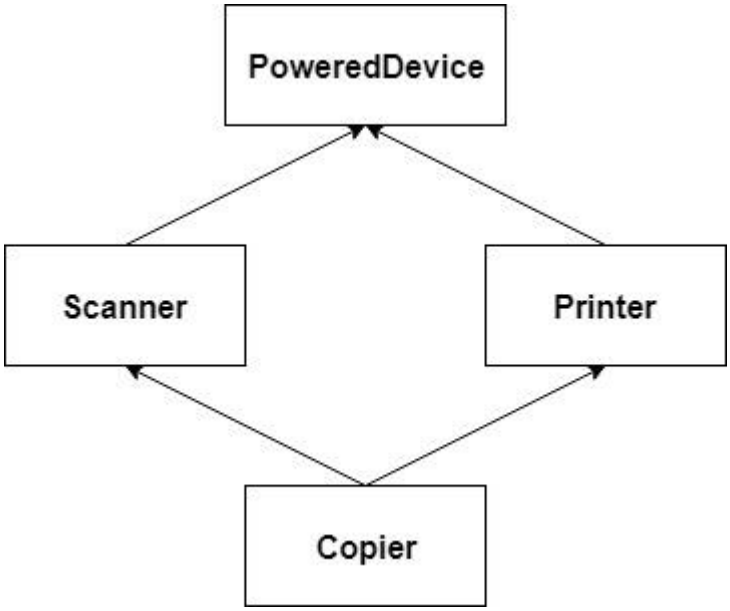
На уроке о множественном наследовании мы говорили о проблеме «алмаза смерти». На этом уроке мы продолжим эту тему.

Алмаз смерти

Код из того же урока, иллюстрирующий "алмаз смерти" (мы добавили еще конструкторы):

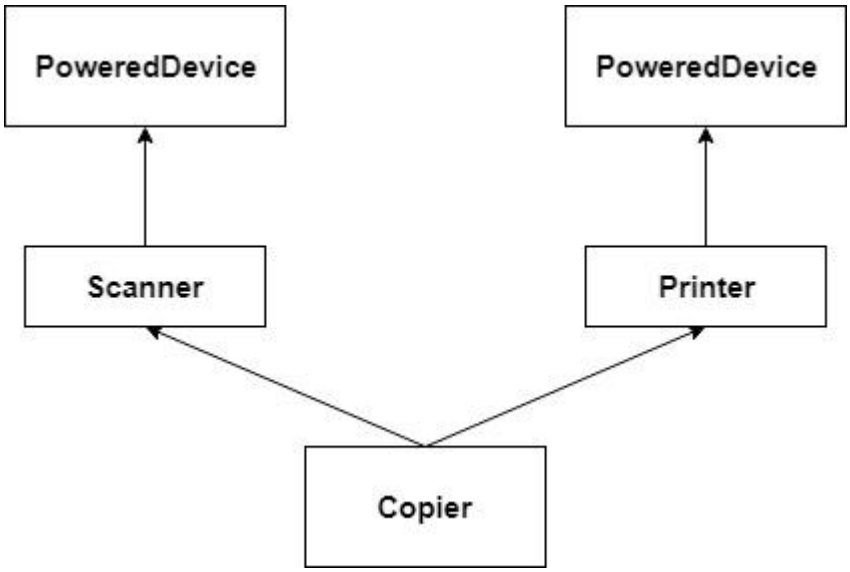
```
1. class PoweredDevice
2. {
3. public:
4.     PoweredDevice(int power)
5.     {
6.         std::cout << "PoweredDevice: " << power << '\n';
7.     }
8. };
9.
10.class Scanner: public PoweredDevice
11.{
12.public:
13.    Scanner(int scanner, int power)
14.        : PoweredDevice(power)
15.    {
16.        std::cout << "Scanner: " << scanner << '\n';
17.    }
18.};
19.
20.class Printer: public PoweredDevice
21.{
22.public:
23.    Printer(int printer, int power)
24.        : PoweredDevice(power)
25.    {
26.        std::cout << "Printer: " << printer << '\n';
27.    }
28.};
29.
30.class Copier: public Scanner, public Printer
31.{
32.public:
33.    Copier(int scanner, int printer, int power)
34.        : Scanner(scanner, power), Printer(printer, power)
35.    {
36.    }
37.};
```

Хотя вы можете ожидать, что диаграмма наследования будет следующая:



На самом деле, это не так. Если вы создадите объект класса Copier, то получите две копии класса PoweredDevice: одну от Printer и одну от Scanner.

Диаграмму получим следующую:



Рассмотрим пример в коде:

```
1. int main()
2. {
3.     Copier copier(1, 2, 3);
4. }
```

Результат:

```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
```

Как вы видите, PoweredDevice создается дважды. Иногда так и нужно, а иногда нужно, чтобы была одна копия PoweredDevice: общая как для Scanner, так и для Printer.

Виртуальные базовые классы

Чтобы сделать родительский (базовый) класс общим, используется **ключевое слово virtual** в строке объявления дочернего класса. **Виртуальный базовый класс** — это класс, объект которого является общим для использования всеми дочерними классами. Вот пример (без конструкторов для простоты) создания общего родительского класса:

```
1. class PoweredDevice
2. {
3. };
4.
5. class Scanner: virtual public PoweredDevice
6. {
7. };
8.
9. class Printer: virtual public PoweredDevice
10.{
11.};
12.
13.class Copier: public Scanner, public Printer
14.{
15.};
```

Теперь, при создании класса Copier, мы получим только одну копию PoweredDevice, которая будет общей как для Scanner, так и для Printer.

Следует вопрос: «Если Scanner и Printer совместно используют родительский класс PoweredDevice, то кто ответственный за его создание?». Оказывается, Copier. Конструктор Copier отвечает за создание объекта PoweredDevice. Это один из тех случаев, когда дочернему классу разрешено вызывать конструктор родительского класса, который не является его непосредственным родителем:

```
1. #include <iostream>
2.
3. class PoweredDevice
4. {
5. public:
```

```
6.      PoweredDevice(int power)
7.      {
8.          std::cout << "PoweredDevice: " << power << '\n';
9.      }
10. };
11.
12. class Scanner: virtual public PoweredDevice // примечание: PoweredDevice
    теперь виртуальный базовый класс
13. {
14. public:
15.     Scanner(int scanner, int power)
16.         : PoweredDevice(power) // эта строка необходима для создания объектов
    класса Scanner, но в этой программе она игнорируется
17.     {
18.         std::cout << "Scanner: " << scanner << '\n';
19.     }
20. };
21.
22. class Printer: virtual public PoweredDevice // примечание: PoweredDevice
    теперь виртуальный базовый класс
23. {
24. public:
25.     Printer(int printer, int power)
26.         : PoweredDevice(power) // эта строка необходима для создания объектов
    класса Printer, но в этой программе она игнорируется
27.     {
28.         std::cout << "Printer: " << printer << '\n';
29.     }
30. };
31.
32. class Copier: public Scanner, public Printer
33. {
34. public:
35.     Copier(int scanner, int printer, int power)
36.         : Scanner(scanner, power), Printer(printer, power),
37.         PoweredDevice(power) // построение PoweredDevice выполняется здесь
38.     {
39.     }
40. };
41.
42. int main()
43. {
44.     Copier copier(1, 2, 3);
45. }
```

Результат выполнения программы:

```
PoweredDevice: 3
Scanner: 1
Printer: 2
```

Здесь уже PoweredDevice создается только один раз.

Обсудим несколько деталей.

Во-первых, виртуальные базовые классы всегда создаются перед не виртуальными базовыми классами, что обеспечивает построение всех базовых классов до построения их производных классов.

Во-вторых, конструкторы Scanner и Printer по-прежнему вызывают конструктор PoweredDevice. При создании объекта Copier эти вызовы конструктора просто игнорируются, так как именно Copier отвечает за создание PoweredDevice, а не Scanner или Printer. Однако, если бы мы создавали объекты Scanner или Printer, то эти конструкторы вызывались бы и применялись обычные правила наследования.

В-третьих, если класс, становясь дочерним, наследует один или несколько классов, которые, в свою очередь, имеют виртуальные родительские классы, то наиболее дочерний класс отвечает за создание виртуального родительского класса. В программе, приведенной выше, Copier наследует Printer и Scanner, которые оба имеют общий виртуальный родительский класс PoweredDevice. Copier, наиболее дочерний класс, отвечает за создание PoweredDevice. Это работает даже в случае одиночного наследования: когда Copier наследует только Printer, а Printer виртуально наследует PoweredDevice, то Copier по-прежнему ответственный за создание PoweredDevice.

Урок №178. Обрезка объектов

Вернемся к примеру с классами Parent и Child:

```
#include <iostream>

class Parent
{
protected:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

    virtual const char* getName() const { return "Parent"; }
    int getValue() const { return m_value; }
};

class Child: public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }

    virtual const char* getName() const { return "Child"; }
};

int main()
{
    Child child(7);
    std::cout << "child is a " << child.getName() << " and has value " << child.getValue() << '\n';

    Parent &ref = child;
    std::cout << "ref is a " << ref.getName() << " and has value " << ref.getValue() << '\n';

    Parent *ptr = &child;
    std::cout << "ptr is a " << ptr->getName() << " and has value " << ptr->getValue() << '\n';

    return 0;
}
```

Здесь ссылка `ref` и указатель `ptr` ссылаются/указывают на объект `child`, который имеет как часть Parent, так и часть Child. Поскольку `ref` и `ptr` являются типа Parent, то они могут видеть часть Parent объекта `child`. Часть Child объекта `child` существует на протяжении всего времени жизни объекта, но доступ к ней для `ref` или `ptr` — закрыт. Однако, используя виртуальные функции, мы получаем доступ к наиболее дочернему методу.

Следовательно, результат выполнения программы:

```
child is a Child and has value 7
ref is a Child and has value 7
ptr is a Child and has value 7
```

Но что бы произошло, если бы мы, вместо создания ссылки или указателя класса Parent на объект класса Child, просто присвоили объект класса Child объекту класса Parent?

```
1. int main()
2. {
3.     Child child(7);
4.     Parent parent = child; // что произойдет здесь?
5.     std::cout << "parent is a " << parent.getName() << " and has value " <<
   parent.getValue() << '\n';
6. 
7.     return 0;
8. }
```

Помните, что `child` имеет как часть Parent, так и часть Child. Когда мы присваиваем объект класса Child объекту класса Parent, то копируется только часть Parent, часть Child не копируется. В примере, приведенном выше, `parent` получает копию части Parent объекта `child`, а часть Child объекта `child` «обрезается». Это называется **обрезкой объектов** (или просто ***«обрезкой»***).

Поскольку переменная `parent` не имеет части Child, то `parent.getName()` вызывает Parent::getName().

Результат:

```
parent is a Parent and has value 7
```

Обрезка объектов и функции

Сейчас вы можете подумать, что вышеприведенный пример нелепый. В конце концов, зачем нам присваивать объект `child` объекту `parent` таким образом? Повторять это, скорее всего, вы не будете. Однако обрезка объектов довольно-таки нередко случается с функциями. Например:

```
1. void printName(const Parent parent) // примечание: Передача по значению
2. {
3.     std::cout << "I am a " << parent.getName() << '\n';
4. }
```

Это простая функция с константным объектом `parent` в качестве параметра, который передается по значению. Если мы будем вызывать эту функцию следующим образом:

```
1. int main()
2. {
3.     Child ch(7);
4.     printName(ch); // упс, передача по значению
5.
6.     return 0;
7. }
```

То получим:

```
I am a Parent
```

Вы, наверное, не заметили, что `parent` является параметром-значением, а не параметром-ссылкой. При выполнении `printName(ch)`, вы, наверное, ожидали, что `parent.getName()` вызовет переопределение getName(), которое выведет `I am a Child`, но это не так. Вместо этого объект `ch` класса Child обрезается, и только часть Parent копируется в передаваемый параметр `parent`. При выполнении `parent.getName()`, несмотря на то, что функция getName() является виртуальной, для нее не существует части Child. Следовательно, получили то, что получили.

В этом случае всё довольно очевидно по тому, что выводится на экран. Но если у вас есть функции, которые ничего не выводят на экран, то отследить такую ошибку будет уже проблематично.

Конечно, обрезки здесь можно было бы легко избежать, используя передачу по ссылке вместо передачи по значению (вот еще одна причина, по которой передача классов по ссылке вместо передачи по значению является хорошей идеей):

```
1. void printName(const Parent &parent) // примечание: Передача по ссылке
2. {
3.     std::cout << "I am a " << parent.getName() << '\n';
4. }
5.
6. int main()
7. {
8.     Child ch(7);
9.     printName(ch);
10.
11.    return 0;
12.}
```

Результат:

```
I am a Child
```

Обрезка векторов

Еще одна ошибка, с которой сталкиваются новички при работе с обрезкой, заключается в попытке реализовать полиморфизм, используя std::vector. Добавим к нашей программе следующий код:

```
#include <vector>

int main()
{
    std::vector<Parent> v;
    v.push_back(Parent(7)); // добавляем объект класса Parent в наш вектор
    v.push_back(Child(8)); // добавляем объект класса Child в наш вектор

        // Выводим все элементы нашего вектора
    for (int count = 0; count < v.size(); ++count)
        std::cout << "I am a " << v[count].getName() << " with value " << v[count].getValue() << "\n";

    return 0;
}
```

Результат выполнения программы:

```
I am a Parent with value 7
I am a Parent with value 8
```

Поскольку std::vector был объявлен как вектор класса Parent, то при добавлении к нему `Child(8)` выполнилась обрезка объекта.

Исправить это немного сложнее. Новички пытаются сделать вектор с ссылками на объекты, например:

```
std::vector<Parent&> v;
```

К сожалению, это не сработает. Элементы std::vector должны быть объектами, которым можно переприсваивать значения, тогда как ссылки могут быть инициализированы только раз и переприсваивать им значения нельзя.

Одним из способов решения этой проблемы является создание вектора с указателями на объекты:

```
#include <vector>

int main()
{
    std::vector<Parent*> v;
    v.push_back(new Parent(7)); // добавляем объект класса Parent в наш вектор
    v.push_back(new Child(8)); // добавляем объект класса Child в наш вектор

        // Выводим все элементы нашего вектора
```

```
    for (int count = 0; count < v.size(); ++count)
        std::cout << "I am a " << v[count]->getName() <<
   " with value " << v[count]->getValue() << "\n";

    for (int count = 0; count < v.size(); ++count)
        delete v[count];

    return 0;
}
```

Результат выполнения программы:

```
I am a Parent with value 7
I am a Child with value 8
```

Работает! Но это боль, так как теперь нам придется иметь дело с динамическим выделением памяти.

К счастью, есть еще один способ решения этой проблемы. Стандартная библиотека C++ предоставляет класс std::reference_wrapper. По сути, **std::reference_wrapper** — это класс, который работает как ссылка, но позволяет выполнять операции присваивания и копирования и совместим с std::vector.

Хорошая новость заключается в том, что вам не нужно знать, как он реализован для того, чтобы его использовать. Всё, что вам нужно знать:

- std::reference_wrapper находится в заголовочном файле functional.
- При создании объекта класса std::reference_wrapper, этот объект не может быть анонимным (поскольку анонимные объекты имеют область видимости выражения, что может привести к висячей ссылке).
- Для получения объекта из std::reference_wrapper используется метод get().

Перепишем наш код, добавив std::reference_wrapper:

```
#include <vector>
#include <functional> // для std::reference_wrapper

int main()
{
    std::vector<std::reference_wrapper<Parent> > v;
    Parent p(7); // p и ch не могут быть анонимными объектами
    Child ch(8);
    v.push_back(p); // добавляем объект класса Parent в наш вектор
    v.push_back(ch); // добавляем объект класса Child в наш вектор

    // Выводим все элементы нашего вектора
    for (int count = 0; count < v.size(); ++count)
        std::cout << "I am a " << v[count].get().getName() << " with value " <<
     v[count].get().getValue() << "\n"; // используем .get() для получения
   элементов из std::reference_wrapper

```

```
16.
17.     return 0;
18. }
```

Результат выполнения программы:

```
I am a Parent with value 7
I am a Child with value 8
```

Всё отлично, и нам не нужно заморачиваться с динамическим выделением памяти.

Заключение

Хотя язык C++ поддерживает присваивание объектов дочерних классов объектам родительского класса посредством обрезки объектов, это приносит больше боли, нежели пользы, поэтому рекомендуется избегать случаев с выполнением обрезки объектов. Когда дело доходит до работы с дочерними классами, всегда перепроверяйте параметры своих функций, чтобы ни в коем случае не выполнялась передача по значению.

Урок №179. Динамическое приведение типов. Оператор dynamic_cast

На уроке о явном преобразовании типов данных мы рассматривали использование оператора static_cast для конвертации переменных из одного типа данных в другой. На этом уроке мы рассмотрим еще один оператор явного преобразования — dynamic_cast.

Зачем нужен dynamic_cast?

Применяя полиморфизм на практике вы часто будете сталкиваться с ситуациями, когда у вас есть указатель на родительский класс, но вам нужно получить доступ к данным, которые есть только в дочернем классе. Например:

```
#include <iostream>
#include <string>

class Parent
{
protected:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

    virtual ~Parent() {}
};

class Child: public Parent
{
protected:
    std::string m_name;

public:
    Child(int value, std::string name)
        : Parent(value), m_name(name)
    {
    }

    const std::string& getName() { return m_name; }
};

Parent* getObject(bool bReturnChild)
{
    if (bReturnChild)
        return new Child(1, "Banana");
    else
        return new Parent(2);
}

int main()
```

```
{
    Parent *p = getObject(true);

    // Как мы выведем имя объекта класса Child здесь, имея лишь один указатель класса Parent?

    delete p;

    return 0;
}
```

В этой программе метод getObject() всегда возвращает указатель класса Parent, но этот указатель может указывать либо на объект класса Parent, либо на объект класса Child. В случае, когда указатель указывает на объект класса Child, как мы будем вызывать Child::getName()?

Один из способов - добавить виртуальную функцию getName() в класс Parent (чтобы иметь возможность вызывать переопределение через объект класса Parent). Но, используя этот вариант, мы будем загромождать класс Parent тем, что должно быть заботой только класса Child.

Язык C++ позволяет нам неявно конвертировать указатель класса Child в указатель класса Parent (фактически, это и делает getObject()). Эта конвертация называется **приведением к базовому типу** (или ***«повышающим приведением типа»***). Однако, что, если бы мы могли конвертировать указатель класса Parent обратно в указатель класса Child? Таким образом, мы могли бы напрямую вызывать Child::getName(), используя тот же указатель, и вообще не заморачиваться с виртуальными функциями.

Оператор dynamic_cast

В языке C++ **оператор dynamic_cast** используется именно для этого. Хотя динамическое приведение позволяет выполнять не только конвертацию указателей родительского класса в указатели дочернего класса, это является наиболее распространенным применением оператора dynamic_cast. Этот процесс называется **приведением к дочернему типу** (или ***«понижающим приведением типа»***).

Использование dynamic_cast почти идентично использованию static_cast. Вот функция main() из вышеприведенного примера, где мы используем dynamic_cast для конвертации указателя класса Parent обратно в указатель класса Child:

```
int main()
{
    Parent *p = getObject(true);

        Child *ch = dynamic_cast<Child*>(p); // используем dynamic_cast для конвертации указателя класса Parent в указатель класса Child
```

```
6.
7.          std::cout << "The name of the Child is: " << ch->getName() << '\n';
8.
9.      delete p;
10.
11.     return 0;
12. }
```

Результат:

```
The name of the Child is: Banana
```

Невозможность конвертации через dynamic_cast

Вышеприведенный пример работает только из-за того, что указатель `p` на самом деле указывает на объект класса Child, поэтому конвертация успешна.

«А что произошло бы, если бы `p` не указывал на объект класса Child?» - спросите Вы. Это легко проверить, изменив аргумент метода getObject() из true на false. В таком случае getObject() будет возвращать указатель класса Parent на объект класса Parent. Если затем мы попытаемся использовать dynamic_cast для конвертации в Child, то потерпим неудачу, так как подобное преобразование невозможно.

Если dynamic_cast не может выполнить конвертацию, то он возвращает нулевой указатель.

Поскольку в коде, приведенном выше, мы не добавили проверку на нулевой указатель, то при выполнении `ch->getName()` мы попытаемся разыменовать нулевой указатель, что, в свою очередь, приведет к неопределенным результатам (или к сбою).

Чтобы сделать программу безопасной, необходимо добавить проверку результата выполнения dynamic_cast:

```
1. int main()
2. {
3.     Parent *p = getObject(true);
4.
5.         Child *ch = dynamic_cast<Child*>(p); // используем dynamic_cast для
   конвертации указателя класса Parent в указатель класса Child
6.
7.         if (ch) // выполняем проверку ch на нулевой указатель
8.             std::cout << "The name of the Child is: " << ch-
   >getName() << '\n';
9.
10.     delete p;
11.
12.     return 0;
13. }
```

Правило: Всегда делайте проверку результата динамического приведения на нулевой указатель.

Обратите внимание, поскольку динамическое приведение выполняет проверку во время запуска программы (чтобы гарантировать возможность выполнения конвертации), использование оператора dynamic_cast чуть снижает производительность программы.

Также обратите внимание на **случаи, в которых понижающее приведение с использованием оператора dynamic_cast не работает**:

- Наследование типа private или protected.
- Классы, которые не объявляют или не наследуют классы с какими-либо виртуальными функциями (и, следовательно, не имеют виртуальных таблиц). В примере, приведенном выше, если бы мы удалили виртуальный деструктор класса Parent, то преобразование через dynamic_cast не выполнилось бы.
- Случаи, связанные с виртуальными базовыми классами (на **сайте Microsoft** вы можете посмотреть примеры таких случаев и их решения).

Понижающее приведение и оператор static_cast

Оказывается, понижающее приведение также может быть выполнено и через оператор static_cast. Основное отличие заключается в том, что static_cast не выполняет проверку во время запуска программы, чтобы убедиться в том, что вы делаете то, что имеет смысл. Это позволяет оператору static_cast быть быстрее, но опаснее оператора dynamic_cast. Если вы будете конвертировать Parent* в Child*, то операция будет «успешной», даже если указатель класса Parent не будет указывать на объект класса Child. А сюрприз вы получите тогда, когда попытаетесь получить доступ к этому указателю (который после конвертации должен быть класса Child, но, фактически, указывает на объект класса Parent).

Если вы абсолютно уверены, что операция с понижающим приведением указателя будет успешна, то использование static_cast является приемлемым. Один из способов убедиться в этом - использовать виртуальную функцию:

```
1. #include <iostream>
2. #include <string>
3. 
4. // Идентификаторы классов
5. enum ClassID
6. {
7.     PARENT,
8.     CHILD
9.     // Здесь можно добавить еще несколько классов
```

```
10. };
11.
12. class Parent
13. {
14. protected:
15.     int m_value;
16.
17. public:
18.     Parent(int value)
19.         : m_value(value)
20.     {
21.     }
22.
23.     virtual ~Parent() {}
24.     virtual ClassID getClassID() { return PARENT; }
25. };
26.
27. class Child: public Parent
28. {
29. protected:
30.     std::string m_name;
31.
32. public:
33.     Child(int value, std::string name)
34.         : Parent(value), m_name(name)
35.     {
36.     }
37.
38.     std::string& getName() { return m_name; }
39.     virtual ClassID getClassID() { return CHILD; }
40.
41. };
42.
43. Parent* getObject(bool bReturnChild)
44. {
45.     if (bReturnChild)
46.         return new Child(1, "Banana");
47.     else
48.         return new Parent(2);
49. }
50.
51. int main()
52. {
53.     Parent *p = getObject(true);
54.
55.     if (p->getClassID() == CHILD)
56.     {
57.         // Мы уже доказали, что p указывает на объект класса Child, поэтому
    никаких проблем здесь не должно быть
58.         Child *ch = static_cast<Child*>(p);
59.         std::cout << "The name of the Child is: " << ch->getName() << '\n';
60.     }
61.
62.     delete p;
63.
64.     return 0;
65. }
```

Но, если вы не уверены в успешности конвертации и не хотите заморачиваться с проверкой через виртуальные функции, вы можете просто использовать оператор dynamic_cast.

Оператор dynamic_cast и Ссылки

Хотя во всех примерах, приведенных выше, мы использовали динамическое приведение с указателями (что является наиболее распространенным), оператор dynamic_cast также может использоваться и со ссылками. Работа dynamic_cast со ссылками аналогична работе с указателями:

```
#include <iostream>
#include <string>

class Parent
{
protected:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

    virtual ~Parent() {}
};

class Child: public Parent
{
protected:
    std::string m_name;

public:
    Child(int value, std::string name)
        : Parent(value), m_name(name)
    {
    }

    const std::string& getName() { return m_name; }
};

int main()
{
    Child banana(1, "Banana");
    Parent &p = banana;
    Child &ch = dynamic_cast<Child&>(p); // используем оператор dynamic_cast для конвертации ссылки класса Parent в ссылку класса Child

    std::cout << "The name of the Child is: " << ch.getName() << '\n';

    return 0;
}
```

Поскольку в языке C++ не существует «нулевой ссылки», то dynamic_cast не может возвратить «нулевую ссылку» при сбое. Вместо этого, dynamic_cast генерирует исключение типа std::bad_cast (мы поговорим об исключениях чуть позже).

Оператор dynamic_cast vs. Оператор static_cast

Начинающие программисты путают, в каких случаях следует использовать static_cast, а в каких — dynamic_cast. Ответ довольно прост: **используйте оператор dynamic_cast при понижающем приведении, а во всех остальных случаях используйте оператор static_cast**. Однако, вам также следует рассматривать возможность использования виртуальных функций вместо операторов преобразования типов данных.

Понижающее приведение vs. Виртуальные функции

Есть программисты, которые считают, что dynamic_cast — это зло и моветон. Они же советуют использовать виртуальные функции вместо оператора dynamic_cast.

В общем, использование виртуальных функций ***должно*** быть предпочтительнее использования понижающего приведения. Однако **в следующих случаях понижающее приведение является лучшим вариантом**:

- Если вы не можете изменить родительский класс, чтобы добавить в него свою виртуальную функцию (например, если родительский класс является частью Стандартной библиотеки C++). При этом, чтобы использовать понижающее приведение, в родительском классе должны уже присутствовать виртуальные функции.
- Если вам нужен доступ к чему-либо, что есть только в дочернем классе (например, к функции доступа, которая существует только в дочернем классе).
- Если добавление виртуальной функции в родительский класс не имеет смысла. В таком случае, в качестве альтернативы, если вам не нужно создавать объект родительского класса, вы можете использовать чистую виртуальную функцию.

Урок №180. Вывод объектов классов через оператор вывода

Рассмотрим следующую программу:

```
#include <iostream>

class Parent
{
public:
    Parent() {}

    virtual void print() const { std::cout << "Parent";  }
};

class Child: public Parent
{
public:
    Child() {}

    virtual void print() const override { std::cout << "Child"; }
};

int main()
{
    Child ch;
    Parent &p = ch;
    p.print(); // вызывается Child::print()

    return 0;
}
```

Здесь понятно, что `p.print()` вызывает Child::print() (поскольку `p` ссылается на объект класса Child, то Parent::print() является виртуальной функцией, а Child::print() является переопределением).

Такой способ вывода неплохой, но с std::cout не очень хорошо сочетается:

```
int main()
{
    Child ch;
    Parent &p = ch;

        std::cout << "p is a ";
    p.print(); // разрываем стейтмент cout ради функции print(). Не дело!
        std::cout << '\n';

    return 0;
}
```

На этом уроке мы рассмотрим, как переопределить оператор вывода `<<` для классов с наследованием, чтобы иметь возможность использовать оператор `<<` следующим образом:

```
1. std::cout << "p is a " << p << '\n'; // гораздо лучше
```

Начнем с обычной перегрузки оператора вывода `<<`:

```
1. #include <iostream>
2.
3. class Parent
4. {
5. public:
6.     Parent() {}
7.
8.     virtual void print() const { std::cout << "Parent";  }
9.
10.     friend std::ostream& operator<<(std::ostream &out, const Parent &p)
11.         {
12.             out << "Parent";
13.             return out;
14.         }
15. };
16.
17. class Child: public Parent
18. {
19. public:
20.     Child() {}
21.
22.     virtual void print() const override { std::cout << "Child"; }
23.
24.     friend std::ostream& operator<<(std::ostream &out, const Child &ch)
25.         {
26.             out << "Child";
27.             return out;
28.         }
29.
30. };
31.
32. int main()
33. {
34.     Parent p;
35.     std::cout << p << '\n';
36.
37.     Child ch;
38.     std::cout << ch << '\n';
39.
40.     return 0;
41. }
```

Поскольку здесь нет виртуальных функций, то всё довольно-таки просто и ясно:

```
Parent
Child
```

Теперь заменим функцию main() на следующую:

```
1. int main()
2. {
3.     Child ch;
4.     Parent &pref = ch;
5.     std::cout << pref << '\n';
6. 
7.     return 0;
8. }
```

Результат:

```
Parent
```

А это уже не то, что нам нужно. Поскольку перегрузка оператора `<<` для объектов класса Parent не является виртуальной, то `std::cout << pref` вызывает версию оператора `<<`, которая работает только с объектами класса Parent. В этом и суть проблемы.

Можем ли мы сделать operator<< виртуальным?

Нет, и на это есть ряд причин.

Во-первых, только методы могут быть виртуальными — это логично, так как только классы могут наследовать другие классы, и переопределить функцию, которая находится вне тела класса — невозможно (мы можем **перегрузить** функции, которые не являются методами, но **не можем переопределить** их). Поскольку оператор `<<` обычно перегружается через дружественную функцию, а дружественные функции не являются методами, то дружественная функция operator<< не может быть переопределена.

Во-вторых, даже если бы мы могли сделать operator<< виртуальной функцией, то проблема заключается в том, что параметры Parent::operator<< и Child::operator<< отличаются (версия Parent принимает в качестве параметра объект класса Parent, а версия Child - объект класса Child). Следовательно, версия Child не может считаться переопределением версии Parent и вызываться в качестве переопределения тоже не может.

Что остается делать программисту?

Решение

Ответ на удивление прост.

Сначала мы делаем operator<< дружественной функцией классу Parent. Но вместо того, чтобы operator<< производил вывод самостоятельно, мы делегируем эту задачу обычному методу, который является виртуальной функцией!

Рассмотрим это на практике:

```
#include <iostream>

class Parent
{
public:
    Parent() {}

    // Перегрузка оператора вывода <<
    friend std::ostream& operator<<(std::ostream &out, const Parent &p)
    {
        // Делегируем выполнение операции вывода методу print()
        return p.print(out);
    }

    // Делаем метод print() виртуальным
    virtual std::ostream& print(std::ostream& out) const
    {
        out << "Parent";
        return out;
    }
};

class Child: public Parent
{
public:
    Child() {}

    // Переопределение метода print() для работы с объектами класса Child
    virtual std::ostream& print(std::ostream& out) const override
    {
        out << "Child";
        return out;
    }
};

int main()
{
    Parent p;
    std::cout << p << '\n';

    Child ch;
    std::cout << ch << '\n'; // обратите внимание, всё работает даже без наличия перегрузки оператора вывода в классе Child

    Parent &pref = ch;
    std::cout << pref << '\n';

    return 0;
}
```

Вышеприведенная программа работает во всех 3-х случаях:

```
Parent
Child
Child
```

Рассмотрим детально.

В случае с объектом класса Parent, мы вызываем operator<<, который вызывает виртуальную функцию print(). Поскольку мы ссылаемся на объект класса Parent, то `p.print()` вызывает Parent::print(), который и выполняет вывод на экран. Здесь всё просто.

В случае с объектом класса Child, компилятор сначала смотрит, есть ли operator<<, который принимает объект класса Child. Он ничего не находит (так как мы это не определили), затем смотрит, есть ли operator<<, который принимает объект класса Parent. Есть, компилятор находит и выполняет неявное преобразование (повышающее приведение) объекта класса Child (ссылки на объект класса Child) в ссылку класса Parent и вызывает виртуальную функцию print(), которая, в свою очередь, вызывает переопределение Child::print().

Обратите внимание, нам не нужно записывать перегрузку operator<< в каждом дочернем классе! Перегрузка, которая находится в классе Parent, отлично работает как с объектами класса Parent, так и с объектами любого дочернего класса (который наследует класс Parent)!

В последнем случае компилятор сопоставляет ссылку `pref` с operator<< класса Parent. Вызывается виртуальная функция print(). Поскольку ссылка `pref` фактически указывает на объект класса Child, то вызывается переопределение Child::print(), как мы и предполагали.

Проблема решена.

Глава №12. Итоговый тест

Итак, наше путешествие в наследование и виртуальные функции в языке C++ подошло к концу. Пора закрепить пройденный материал.

Теория

Язык C++ позволяет создавать указатели/ссылки родительского класса на объекты дочерних классов. Это полезно при использовании функций или массивов, которые должны работать с объектами дочерних классов.

Без виртуальных функций указатели/ссылки родительского класса на объект дочернего класса будут иметь доступ только к членам родительского класса.

Виртуальная функция - это особый тип функции, которая, при обращении к ней, вызывает наиболее дочерний метод (переопределение), существующий между родительским и дочерними классами. Чтобы считаться переопределением, метод дочернего класса должен иметь ту же сигнатуру и тип возврата, что и виртуальная функция родительского класса. Единственное исключение - **ковариантный тип возврата**, который позволяет переопределению возвращать указатель или ссылку на дочерний класс, если метод родительского класса возвращает указатель или ссылку на себя.

Модификатор override используется для обозначения метода переопределением.

Модификатор final запрещает переопределять виртуальную функцию или наследовать определенный класс.

Используя виртуальные функции, не забывайте добавлять в родительский класс **виртуальный деструктор**, чтобы, в случае удаления указателя на родительский класс, вызывался соответствующий деструктор.

Вы можете игнорировать вызов переопределений виртуальной функции, используя оператор разрешения области видимости, чтобы напрямую указать, какую функцию вы хотите вызвать. Например, `parent.Parent::GetName()`.

Раннее связывание происходит, когда компилятор встречает прямой вызов функции. Компилятор или линкер могут напрямую обрабатывать прямые вызовы функций. **Позднее связывание** происходит при вызове указателя на функцию. В таких случаях невозможно знать наперёд, какая функция будет вызываться первой.

Виртуальные функции используют позднее связывание и **виртуальные таблицы** для определения того, какую версию функции следует вызывать.

Относительные недостатки виртуальных функций:

- Вызов виртуальных функций занимает больше времени.
- Необходимость наличия виртуальной таблицы увеличивает размер каждого объекта класса, содержащего виртуальную функцию, на размер одного указателя.

Виртуальную функцию можно сделать **чистой виртуальной/абстрактной функцией**, добавив `= 0` в конец её прототипа. Класс, содержащий чистую виртуальную функцию, называется **абстрактным классом**. Объекты абстрактного класса не могут быть созданы. Класс, который наследует чистые виртуальные функции, должен предоставить свои переопределения этих функций, или он также будет считаться абстрактным. Чистые виртуальные функции могут иметь тело (определение, записанное отдельно), но они по-прежнему считаются абстрактными функциями.

Интерфейс (или ***«интерфейсный класс»***) - это класс без переменных-членов, все методы которого являются чистыми виртуальными функциями. Имена интерфейсов часто начинаются с `I`.

Виртуальный базовый класс - это родительский класс, объект которого является общим для использования всеми дочерними классами.

При присваивании объекта дочернего класса объекту родительского класса, в объект родительского класса копируется лишь родительская часть копируемого объекта, дочерняя часть копируемого объекта обрезается. Этот процесс называется **обрезкой объектов**.

Динамическое приведение используется для конвертации указателя родительского класса в указатель дочернего класса. Это называется **понижающим приведением типа**. Если конвертация прошла неудачно, то возвращается нулевой указатель.

Самый простой способ **перегрузить оператор вывода** `<<` **для классов с наследованием** - записать перегрузку оператора `<<` в родительском классе, а выполнение операции вывода делегировать виртуальному методу.

Тест

Задание №1

Каждая из следующих программ имеет какую-то ошибку. Ваша задача состоит в том, чтобы найти эту ошибку визуально (не запуская код). Предполагаемый вывод каждой программы:

```
Child
```

a)

```
#include <iostream>

class Parent
{
protected:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

    const char* getName() const { return "Parent"; }
};

class Child: public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }

    const char* getName() const { return "Child"; }
};

int main()
{
    Child ch(7);
    Parent &p = ch;
    std::cout << p.getName();

    return 0;
}
```

b)

```
#include <iostream>

class Parent
{
protected:
    int m_value;
```

```
7.
8.  public:
9.      Parent(int value)
10.         : m_value(value)
11.     {
12.     }
13.
14.     virtual const char* getName() { return "Parent"; }
15. };
16.
17. class Child: public Parent
18. {
19. public:
20.     Child(int value)
21.         : Parent(value)
22.     {
23.     }
24.
25.     virtual const char* getName() const { return "Child"; }
26. };
27.
28. int main()
29. {
30.     Child ch(7);
31.     Parent &p = ch;
32.     std::cout << p.getName();
33.
34.     return 0;
35. }
```

c)

```
1.  #include <iostream>
2.
3.  class Parent
4.  {
5.  protected:
6.      int m_value;
7.
8.  public:
9.      Parent(int value)
10.         : m_value(value)
11.     {
12.     }
13.
14.     virtual const char* getName() { return "Parent"; }
15. };
16.
17. class Child: public Parent
18. {
19. public:
20.     Child(int value)
21.         : Parent(value)
22.     {
23.     }
24.
25.     virtual const char* getName() override { return "Child"; }
26. };
27.
28. int main()
29. {
```

```
    Child ch(7);
    Parent p = ch;
    std::cout << p.getName();

    return 0;
}
```

d)

```
#include <iostream>

class Parent final
{
protected:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

    virtual const char* getName() { return "Parent"; }
};

class Child: public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }

    virtual const char* getName() override { return "Child"; }
};

int main()
{
    Child ch(7);
    Parent &p = ch;
    std::cout << p.getName();

    return 0;
}
```

e)

```
#include <iostream>

class Parent
{
protected:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

```

```
14.     virtual const char* getName() { return "Parent"; }
15. };
16.
17. class Child: public Parent
18. {
19. public:
20.     Child(int value)
21.         : Parent(value)
22.     {
23.     }
24.
25.     virtual const char* getName() = 0;
26. };
27.
28. const char* Child::getName()
29. {
30.     return "Child";
31. }
32.
33. int main()
34. {
35.     Child ch(7);
36.     Parent &p = ch;
37.     std::cout << p.getName();
38.
39.     return 0;
40. }
```

f)

```
1.  #include <iostream>
2.
3.  class Parent
4.  {
5.  protected:
6.      int m_value;
7.
8.  public:
9.      Parent(int value)
10.         : m_value(value)
11.     {
12.     }
13.
14.     virtual const char* getName() { return "Parent"; }
15. };
16.
17. class Child: public Parent
18. {
19. public:
20.     Child(int value)
21.         : Parent(value)
22.     {
23.     }
24.
25.     virtual const char* getName() { return "Child"; }
26. };
27.
28. int main()
29. {
30.     Child *ch = new Child(7);
31.     Parent *p = ch;
```

```
32.     std::cout << p->getName();
33.     delete p;
34.
35.     return 0;
36. }
```

Задание №2

a) Создайте абстрактный класс Shape. Этот класс должен иметь три метода:

- чистую виртуальную функцию print() с параметром типа std::ostream;
- перегрузку operator<<;
- пустой виртуальный деструктор.

b) Создайте два класса: Triangle и Circle, которые наследуют класс Shape.

- Triangle должен иметь 3 точки в качестве переменных-членов.
- Circle должен иметь одну центральную точку и целочисленный радиус в качестве переменных-членов.

Перегрузите функцию print(), чтобы следующий код:

```
1. int main()
2. {
3.     Circle c(Point(1, 2, 3), 7);
4.     std::cout << c << '\n';
5.
6.     Triangle t(Point(1, 2, 3), Point(4, 5, 6), Point(7, 8, 9));
7.     std::cout << t << '\n';
8.
9.     return 0;
10. }
```

Выдавал следующий результат:

```
Circle(Point(1, 2, 3), radius 7)
Triangle(Point(1, 2, 3), Point(4, 5, 6), Point(7, 8, 9))
```

Вот класс Point, который вы можете использовать:

```
1. class Point
2. {
3. private:
4.     int m_x = 0;
5.     int m_y = 0;
6.     int m_z = 0;
7.
8. public:
9.     Point(int x, int y, int z)
10.         : m_x(x), m_y(y), m_z(z)
11.     {
12.
```

```
    }

    friend std::ostream& operator<<(std::ostream &out, const Point &p)
    {
        out << "Point(" << p.m_x << ", " << p.m_y << ", " << p.m_z << ")";
        return out;
    }
};
```

c) Используя код из предыдущих заданий (классы Point, Shape, Circle и Triangle) завершите следующую программу:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<Shape*> v;
    v.push_back(new Circle(Point(1, 2, 3), 7));
    v.push_back(new Triangle(Point(1, 2, 3), Point(4, 5, 6), Point(7, 8, 9)));
    v.push_back(new Circle(Point(4, 5, 6), 3));

    // Вывод элементов вектора v здесь

        std::cout << "The largest radius is: " << getLargestRadius(v) << '\n'; // реализуйте эту функцию

    // Удаление элементов вектора v здесь
}
```

> ***Подсказка***: Вам нужно добавить метод getRadius() в Circle и выполнить понижающее приведение Shape* в Circle*, чтобы получить доступ к этому методу.

Урок №181. Шаблоны функций

Хотя функции и классы являются мощными и гибкими инструментами для эффективного программирования, в некоторых случаях они ограничены из-за требования C++ указывать типы всех используемых параметров. Например, предположим, что нам нужно написать функцию для вычисления наибольшего среди двух чисел:

```
1. int max(int a, int b)
2. {
3.     return (a > b) ? a : b;
4. }
```

Всё отлично до тех пор, пока мы работаем с целочисленными значениями. А что если нам придется работать и со значениями типа double? Вы, вероятно, решите перегрузить функцию max() для работы с типом double:

```
1. double max(double a, double b)
2. {
3.     return (a > b) ? a : b;
4. }
```

Теперь у нас есть две версии одной функции, которые работают с типами char, int, double и, если мы перегрузим оператор `>`, даже с классами! Однако, поскольку C++ требует, чтобы мы указывали типы наших переменных, нам приходится записывать несколько версий одной и той же функции, где единственное, что меняется - это тип параметров.

А это, в свою очередь, головная боль для программистов, так как поддерживать такой код непросто как по затраченным усилиям, так и по времени. И, что самое важное, это нарушает одну из концепций эффективного программирования — сократить до минимума дублирование кода. Правда, было бы неплохо написать одну версию функции max(), которая работала бы с параметрами ЛЮБОГО типа?

Это возможно. Добро пожаловать в мир шаблонов!

Если посмотреть определение слова «шаблон» в словаре, то увидим следующее: «Шаблон - это образец, по которому изготавливаются похожие изделия». Например, шаблоном является трафарет — объект (например, пластинка), в котором прорезан рисунок/узор/символ. Если приложить трафарет к другому объекту и распылить краску, то получим этот же рисунок, прилагая минимум усилий, быстро и, что не менее важно, мы сможем сделать десятки этих рисунков

разных цветов! При этом нам нужен лишь один трафарет и нам не нужно определять цвет рисунка заранее (до использования трафарета).

В языке C++ **шаблоны функций** - **это функции, которые служат образцом для создания других подобных функций**. Главная идея — создание функций без указания точного типа(ов) некоторых или всех переменных. Для этого мы определяем функцию, указывая **тип параметра шаблона**, который используется вместо любого типа данных. После того, как мы создали функцию с типом параметра шаблона, мы фактически создали «трафарет функции».

При вызове шаблона функции, компилятор использует «трафарет» в качестве образца функции, заменяя тип параметра шаблона на фактический тип переменных, передаваемых в функцию! Таким образом, мы можем создать 50 «оттенков» функции, используя всего лишь один шаблон!

Создание шаблонов функций

Сейчас вам, вероятно, интересно, как создаются шаблоны функций в языке C++. Оказывается, это не так уж и сложно. Рассмотрим еще раз целочисленную версию функции max():

```
1. int max(int a, int b)
2. {
3.     return (a > b) ? a : b;
4. }
```

Здесь мы трижды указываем тип данных: в параметрах `a`, `b` и в типе возврата функции. Для создания шаблона этой функции нам нужно заменить тип int на тип параметра шаблона функции. Поскольку в этом случае используется только один тип данных (int), то нам нужно указать только один тип параметра шаблона.

Мы можем назвать этот тип как угодно, главное, чтобы это не было зарезервированным/ключевым словом. В языке C++ принято называть типы параметров шаблонов большой буквой `T` (сокр. от `Type`).

Вот наша переделанная функция max():

```
1. T max(T a, T b)
2. {
3.     return (a > b) ? a : b;
4. }
```

Но это еще не всё. Программа работать не будет, так как компилятор не знает, что такое `T`!

Чтобы всё заработало, нам нужно сообщить компилятору две вещи:

- Определение шаблона функции.
- Указание того, что `T` является типом параметра шаблона функции.

Мы можем сделать это в одной строке кода, выполнив **объявление шаблона** (а точнее — **объявление параметров шаблона**):

```
1. template <typename T> // объявление параметра шаблона функции
2. T max(T a, T b)
3. {
4.     return (a > b) ? a : b;
5. }
```

Эврика! Работает!

Рассмотрим детально объявление параметров шаблона:

- Сначала пишем **ключевое слово template**, которое сообщает компилятору, что дальше мы будем объявлять параметры шаблона.
- Параметры шаблона функции указываются в угловых скобках (`<>`).
- Для создания типов параметров шаблона используются **ключевые слова typename и class**. В базовых случаях использования шаблонов функций разницы между typename и class нет, поэтому вы можете выбрать любое из двух. Если вы используете ключевое слово class, то фактический тип параметров не обязательно должен быть классом (это может быть переменная фундаментального типа данных, указатель или что-то другое).
- Затем называем тип параметра шаблона (обычно `T`).

Если требуется несколько типов параметров шаблона, то они разделяются запятыми:

```
1. template <typename T1, typename T2>
2. // Шаблон функции здесь
```

Если параметров несколько, то их обычно называют `T1`, `T2` или другими буквами: `T`, `S`.

> ***Примечание***: Поскольку тип аргумента функции, передаваемый в тип T, может быть классом, а классы, как правило, не рекомендуется передавать по значению, то лучше сделать параметры и возвращаемое значение нашего шаблона функции константными ссылками, например:
>
> ```
> template <typename T>
> const T& max(const T& a, const T& b)
> ```

```
{
    return (a > b) ? a : b;
}
```

Использование шаблонов функций

Использование шаблонов функций аналогично использованию обычных функций:

```
1. #include <iostream>
2.
3. template <typename T>
4. const T& max(const T& a, const T& b)
5. {
6.     return (a > b) ? a : b;
7. }
8.
9. int main()
10.{
11.    int i = max(4, 8);
12.    std::cout << i << '\n';
13.
14.    double d = max(7.56, 21.434);
15.    std::cout << d << '\n';
16.
17.    char ch = max('b', '9');
18.    std::cout << ch << '\n';
19.
20.    return 0;
21.}
```

Результат:

```
8
21.434
b
```

Обратите внимание, все три вызова функции max() имеют параметры разных типов! Поскольку мы вызываем функцию max() с тремя разными типами параметров, то компилятор использует шаблон функции для создания трех разных версий функции max():

- Версия с параметрами типа int (`max<int>`).
- Версия с параметрами типа double (`max<double>`).
- Версия с параметрами типа char (`max<char>`).

Нам не нужно явно указывать тип передаваемых значений (часть `<int>` в `max<int>`), компилятор вычислит это самостоятельно.

Заключение

Шаблоны функций экономят много времени, так как шаблон мы пишем только один раз, а использовать можем с разными типами данных. Как только вы привыкнете к написанию шаблонов функций, вы обнаружите, что это по времени занимает не больше написания обычной функции (одной версии обычной функции). Шаблоны функций намного упрощают дальнейшую поддержку кода, и они более безопасные, так как нет необходимости выполнять вручную перегрузку функции, копируя код и изменяя лишь типы данных, когда нужна поддержка нового типа данных.

У шаблонов функций **есть несколько недостатков**, и было бы непростительно, если бы мы о них не поговорили:

- Во-первых, некоторые старые компиляторы могут не поддерживать шаблоны функций или поддерживать, но с ограничениями. Однако сейчас это уже не такая проблема, как раньше.
- Во-вторых, шаблоны функций часто выдают сумасшедшие сообщения об ошибках, которые намного сложнее расшифровать, чем ошибки обычных функций.
- В-третьих, шаблоны функций могут увеличить время компиляции и размер кода, так как один шаблон может быть «реализован» и перекомпилирован в нескольких файлах.

Данные недостатки довольно незначительны по сравнению с мощью и гибкостью шаблонов функций!

> ***Примечание*:** Стандартная библиотека C++ имеет в своем арсенале шаблон функции max() (который находится в заголовочном файле algorithm), поэтому вы можете не реализовывать эту функцию вручную в будущем. Кроме этого, если вы пишете свои собственные шаблоны функций и используете стейтмент `using namespace std;`, то не забывайте о возможности возникновения конфликтов имен, так как компилятор не сможет определить, хотите ли вы использовать свою версию функции max() или версию std::max().

Урок №182. Экземпляры шаблонов функций

Язык C++ не компилирует шаблоны функций напрямую. Вместо этого, когда компилятор встречает вызов шаблона функции, он копирует шаблон функции и заменяет типы параметров шаблона функции фактическими (передаваемыми) типами данных. Функция с фактическими типами данных называется **экземпляром шаблона функции** (или ***«объектом шаблона функции»***).

Рассмотрим это на практике. Во-первых, создадим шаблон функции:

```
1. template <typename T> // объявление параметра шаблона функции
2. const T& max(const T& a, const T& b)
3. {
4.     return (a > b) ? a : b;
5. }
```

Затем сделаем вызов шаблона функции:

```
1. int i = max(4, 8); // вызывается max(int, int)
```

Компилятор видит, что оба числа являются целочисленными, поэтому он копирует шаблон функции и создает экземпляр шаблона `max(int, int)`:

```
1. const int& max(const int &a, const int &b)
2. {
3.     return (a > b) ? a : b;
4. }
```

Это теперь уже «обычная функция». Допустим, что нам нужно снова вызвать функцию max(), но уже с другим типом данных:

```
1. double d = max(7.58, 19.378); // вызывается max(double, double)
```

Язык C++ автоматически создает экземпляр шаблона `max(double, double)`:

```
1. const double& max(const double &a, const double &b)
2. {
3.     return (a > b) ? a : b;
4. }
```

И затем компилирует его. Также стоит отметить, что, если вы создадите шаблон функции, но не вызовете его, экземпляры этого шаблона созданы не будут.

Операторы, вызовы функций и шаблоны функций

Шаблоны функций работают как с фундаментальными типами данных (char, int, double и т.д.), так и с классами (но есть нюанс). Экземпляр шаблона компилируется

как обычная функция. В обычной функции любые операторы или вызовы других функций, которые используются в этой функции, должны быть определены/перегружены, или вы получите ошибку компиляции. Аналогично, любые операторы или вызовы других функций, которые присутствуют в шаблоне функции, должны быть определены/перегружены для работы с фактическими (передаваемыми) типами данных. Рассмотрим это на практике.

Во-первых, создадим простой класс:

```
1. class Dollars
2. {
3. private:
4.     int m_dollars;
5. public:
6.     Dollars(int dollars)
7.         : m_dollars(dollars)
8.     {
9.     }
10. };
```

Теперь посмотрим, что произойдет при попытке вызова функции max() с объектами класса Dollars:

```
1. template <typename T> // объявление параметра шаблона функции
2. const T& max(const T& a, const T& b)
3. {
4.     return (a > b) ? a : b;
5. }
6.
7. class Dollars
8. {
9. private:
10.     int m_dollars;
11. public:
12.     Dollars(int dollars)
13.         : m_dollars(dollars)
14.     {
15.     }
16. };
17.
18. int main()
19. {
20.     Dollars seven(7);
21.     Dollars twelve(12);
22.
23.     Dollars bigger = max(seven, twelve);
24.
25.     return 0;
26. }
```

Язык C++ создаст следующий экземпляр шаблона функции max():

```
1. const Dollars& max(const Dollars &a, const Dollars &b)
2. {
3.     return (a > b) ? a : b;
```

```
4. }
```

А затем компилятор попытается скомпилировать эту функцию, но ничего не получится, так как C++ не имеет понятия, как обрабатывать выражение `a > b`! Следовательно, это приведет к ошибке:

```
Ошибка C2676 бинарный ">": "const T" не определяет этот оператор или преобразование к типу приемлемо к встроенному оператору
```

Сообщение об ошибке указывает на тот факт, что мы не перегрузили оператор `>` для класса Dollars. Давайте перегрузим:

```
1. class Dollars
2. {
3. private:
4.     int m_dollars;
5. public:
6.     Dollars(int dollars)
7.         : m_dollars(dollars)
8.     {
9.     }
10.
11.    friend bool operator>(const Dollars &d1, const Dollars &d2)
12.    {
13.        return (d1.m_dollars > d2.m_dollars);
14.    }
15.};
```

Теперь C++ знает, как обрабатывать выражение `a > b`, когда в качестве переменных используются объекты класса Dollars!

Еще один пример

Создадим шаблон функции, которая вычисляет среднее арифметическое элементов массива:

```
1. template <class T>
2. T average(T *array, int length)
3. {
4.     T sum = 0;
5.     for (int count=0; count < length; ++count)
6.         sum += array[count];
7.
8.     sum /= length;
9.     return sum;
10.}
```

Протестируем:

```
1. #include <iostream>
2.
```

```
3.  template <class T>
4.  T average(T *array, int length)
5.  {
6.      T sum = 0;
7.      for (int count=0; count < length; ++count)
8.          sum += array[count];
9.
10.     sum /= length;
11.     return sum;
12. }
13.
14. int main()
15. {
16.     int array1[] = { 6, 4, 1, 3, 7 };
17.     std::cout << average(array1, 5) << '\n';
18.
19.     double array2[] = { 4.25, 5.37, 8.44, 9.25 };
20.     std::cout << average(array2, 4) << '\n';
21.
22.     return 0;
23. }
```

Результат:

```
4
6.8275
```

Как вы видите, всё отлично работает с фундаментальными типами данных!

Поскольку тип возврата шаблона функции тот же, что и тип передаваемых элементов массива в функцию, то вычисление среднего арифметического целочисленных значений приведет к целочисленному результату (с отбрасыванием любой дробной части), как и вычисление значений типа double приведет к результату типа double. Это может быть не очевидным, поэтому хорошим тоном будет указать на это в комментариях.

Теперь посмотрим, что произойдет при вызове функции average() с объектами класса Dollars:

```
1.  #include <iostream>
2.
3.  class Dollars
4.  {
5.  private:
6.      int m_dollars;
7.  public:
8.      Dollars(int dollars)
9.          : m_dollars(dollars)
10.     {
11.     }
12.
13.     friend bool operator>(const Dollars &d1, const Dollars &d2)
14.     {
15.         return (d1.m_dollars > d2.m_dollars);
```

```
    }
};

template <class T>
T average(T *array, int length)
{
    T sum = 0;
    for (int count=0; count < length; ++count)
        sum += array[count];

    sum /= length;
    return sum;
}

int main()
{
    Dollars array3[] = { Dollars(7), Dollars(12), Dollars(18), Dollars(15) };
    std::cout << average(array3, 4) << '\n';

    return 0;
}
```

Результат:

```
1>c:\users\kicli\source\repos\consoleapplication10\consoleappli
cation10\consoleapplication10.cpp(37): error C2679: бинарный
"<<": не найден оператор, принимающий правый операнд типа "T"
(или приемлемое преобразование отсутствует) 1> with
1> [
1> T=Dollars
1> ]
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(508): note: может быть
"std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
<<(std::basic_streambuf<char,std::char_traits> *)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(480): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator <<(const
void *)" 1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(460): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator <<(long
double)" 1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(440): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
<<(double)" 1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(420): note: или "std::basic_ostream<char,std::char_traits>
```

```
&std::basic_ostream<char,std::char_traits>::operator <<(float)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(400): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
<<(unsigned __int64)" 1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(380): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
<<(__int64)" 1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(360): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
<<(unsigned long)" 1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(340): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator <<(long)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(320): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
<<(unsigned int)" 1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(295): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator <<(int)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(275): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
<<(unsigned short)" 1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(241): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator <<(short)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(221): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator <<(bool)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(215): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
<<(std::ios_base &(__cdecl *)(std::ios_base &))" 1>c:\program
files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(209): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
```

```
<<(std::basic_ios<char,std::char_traits> &(__cdecl
*)(std::basic_ios<char,std::char_traits> &))"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(204): note: или "std::basic_ostream<char,std::char_traits>
&std::basic_ostream<char,std::char_traits>::operator
<<(std::basic_ostream<char,std::char_traits> &(__cdecl
*)(std::basic_ostream<char,std::char_traits> &))"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(702): note: или "std::basic_ostream<char,std::char_traits>
&std::operator
<<<char,std::char_traits>(std::basic_ostream<char,std::char_tra
its> &,const char *)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(749): note: или "std::basic_ostream<char,std::char_traits>
&std::operator
<<<char,std::char_traits>(std::basic_ostream<char,std::char_tra
its> &,char)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(787): note: или "std::basic_ostream<char,std::char_traits>
&std::operator
<<<std::char_traits>(std::basic_ostream<char,std::char_traits>
&,const char *)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(834): note: или "std::basic_ostream<char,std::char_traits>
&std::operator
<<<std::char_traits>(std::basic_ostream<char,std::char_traits>
&,char)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(960): note: или "std::basic_ostream<char,std::char_traits>
&std::operator
<<<std::char_traits>(std::basic_ostream<char,std::char_traits>
&,const signed char *)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(967): note: или "std::basic_ostream<char,std::char_traits>
&std::operator
<<<std::char_traits>(std::basic_ostream<char,std::char_traits>
&,signed char)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
```

```
(974): note: или "std::basic_ostream<char,std::char_traits>
&std::operator
<<<std::char_traits>(std::basic_ostream<char,std::char_traits>
&,const unsigned char *)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(981): note: или "std::basic_ostream<char,std::char_traits>
&std::operator
<<<std::char_traits>(std::basic_ostream<char,std::char_traits>
&,unsigned char)"
1>c:\program files (x86)\microsoft visual
studio\2017\community\vc\tools\msvc\14.15.26726\include\ostream
(1047): note: или "std::basic_ostream<char,std::char_traits>
&std::operator
<<<char,std::char_traits>(std::basic_ostream<char,std::char_tra
its> &,const std::error_code &)"
1>c:\users\kicli\source\repos\consoleapplication10\consoleappli
cation10\consoleapplication10.cpp(37): note: при попытке
сопоставить список аргументов "(std::ostream, T)"
1> with
1> [
1> T=Dollars
1> ]
```

Компилятор сошел с ума. Мы говорили о таких ошибках на предыдущем уроке. Несмотря на столь объемный «результат», здесь всё довольно просто. В первых строках сообщается, что компилятор не смог найти перегрузку оператора `<<` для класса Dollars. Далее указываются функции с типами данных, которые вызывались для сравнения, но так и не подошли. И в конце указываются параметр шаблона и заменяемый (фактический) тип параметра.

Visual Studio бережет наши нервы и предоставляет нам альтернативный вывод ошибок:

```
Ошибка E0349 отсутствует оператор "<<", соответствующий этим
операндам
Ошибка C2679 бинарный "<<": не найден оператор, принимающий
правый операнд типа "T" (или приемлемое преобразование
отсутствует)
```

Помните, что average() возвращает объект класса Dollars, а мы пытаемся этот объект вывести с помощью оператора вывода `<<` и std::cout. Однако, мы не перегрузили оператор `<<` для класса Dollars.

Давайте исправим это:

```
class Dollars
{
private:
    int m_dollars;
public:
    Dollars(int dollars)
        : m_dollars(dollars)
    {
    }

    friend bool operator>(const Dollars &d1, const Dollars &d2)
    {
        return (d1.m_dollars > d2.m_dollars);
    }

    friend std::ostream& operator<< (std::ostream &out, const Dollars &dollars)
    {
        out << dollars.m_dollars << " dollars ";
        return out;
    }
};
```

Если же теперь запустить программу, то получим следующее:

```
Ошибка C2676 бинарный "+=": "T" не определяет этот оператор или преобразование к типу приемлемо к встроенному оператору
Ошибка C2676 бинарный "/=": "T" не определяет этот оператор или преобразование к типу приемлемо к встроенному оператору
```

Эти ошибки были вызваны экземпляром шаблона функции, созданным при вызове `average(Dollars*, int)`. Помните, что при вызове шаблона функции, компилятор копирует шаблон функции с типами параметров, а затем заменяет типы параметров шаблона на фактические (передаваемые) типы данных. Вот экземпляр шаблона функции average(), где `T` является классом Dollars:

```
template <class T>
Dollars average(Dollars *array, int length)
{
    Dollars sum = 0;
    for (int count=0; count < length; ++count)
        sum += array[count];

    sum /= length;
    return sum;
}
```

Причина, по которой мы получили сообщение об ошибке, кроется в следующей строке:

```
sum += array[count];
```

В этом случае `sum` является объектом класса Dollars. А чтобы всё заработало, нам нужно перегрузить операторы `+=` и `/=` для класса Dollars:

```
1. class Dollars
2. {
3. private:
4.     int m_dollars;
5. public:
6.     Dollars(int dollars)
7.         : m_dollars(dollars)
8.     {
9.     }
10.
11.    friend bool operator>(const Dollars &d1, const Dollars &d2)
12.    {
13.        return (d1.m_dollars > d2.m_dollars);
14.    }
15.
16.    friend std::ostream& operator<< (std::ostream &out, const Dollars &dollars)
17.    {
18.        out << dollars.m_dollars<< " dollars ";
19.        return out;
20.    }
21.
22.    Dollars& operator+=(Dollars dollars)
23.    {
24.        m_dollars += dollars.m_dollars;
25.        return *this;
26.    }
27.
28.    Dollars& operator/=(int value)
29.    {
30.        m_dollars /= value;
31.        return *this;
32.    }
33. };
```

Наконец, наш код скомпилируется, и результат:

```
13 dollars
```

Хотя проделанная работа может показаться очень большой, но это только из-за того, что наш класс Dollars с самого начала был "кожа да кости". Ключевой момент здесь в том, что нам не нужно модифицировать average(), чтобы он работал с объектами класса Dollars (или с любым другим типом данных). Вся работа была проделана только с классом Dollars, а обо всем остальном компилятор позаботился самостоятельно!

Урок №183. Шаблоны классов

На предыдущих уроках мы узнали, как с помощью шаблонов функций сделать одну версию функции, которая будет работать с разными типами данных. Хотя это значительный шаг на пути к обобщенному программированию, это не решает всех наших проблем. Рассмотрим пример такой проблемы и то, как шаблоны могут нам помочь в её решении.

Шаблоны и контейнерные классы

На уроке о контейнерных классах мы узнали то, как, используя композицию, реализовать классы, содержащие несколько объектов определенного типа данных. В качестве примера мы использовали класс ArrayInt:

```
#ifndef ARRAYINT_H
#define ARRAYINT_H

#include <assert.h> // для assert()

class ArrayInt
{
private:
    int m_length;
    int *m_data;

public:
    ArrayInt()
    {
        m_length = 0;
        m_data = nullptr;
    }

    ArrayInt(int length)
    {
        assert(length > 0);
        m_data = new int[length];
        m_length = length;
    }

    ~ArrayInt()
    {
        delete[] m_data;
    }

    void Erase()
    {
        delete[] m_data;
        // Присваиваем значение nullptr для m_data, чтобы на выходе не получить висячий указатель!
        m_data = nullptr;
        m_length = 0;
    }

    int& operator[](int index)
```

```
40.     {
41.         assert(index >= 0 && index < m_length);
42.         return m_data[index];
43.     }
44.
45.     int getLength() { return m_length; }
46. };
47.
48. #endif
```

Хотя этот класс обеспечивает простой способ создания массива целочисленных значений, но что, если нам нужно будет работать со значениями типа double? Используя традиционные методы программирования мы создали бы новый класс ArrayDouble для работы со значениями типа double:

```
1.  #ifndef ARRAYDOUBLE_H
2.  #define ARRAYDOUBLE_H
3.
4.  #include <assert.h> // для assert()
5.
6.  class ArrayDouble
7.  {
8.  private:
9.      int m_length;
10.     double *m_data;
11.
12. public:
13.     ArrayDouble()
14.     {
15.         m_length = 0;
16.         m_data = nullptr;
17.     }
18.
19.     ArrayDouble(int length)
20.     {
21.         assert(length > 0);
22.         m_data = new double[length];
23.         m_length = length;
24.     }
25.
26.     ~ArrayDouble()
27.     {
28.         delete[] m_data;
29.     }
30.
31.     void Erase()
32.     {
33.         delete[] m_data;
34.         // Присваиваем значение nullptr для m_data, чтобы на выходе не получить
    висячий указатель!
35.         m_data = nullptr;
36.         m_length = 0;
37.     }
38.
39.     double & operator[](int index)
40.     {
41.         assert(index >= 0 && index < m_length);
42.         return m_data[index];
43.     }
```

```
44.
45.     int getLength() { return m_length; }
46. };
47.
48. #endif
```

Хотя кода много, но классы почти идентичны, меняется только тип данных! Как вы уже могли бы догадаться, это идеальный случай для использования шаблонов. **Создание шаблона класса аналогично созданию шаблона функции**. Например, создадим шаблон класса Array:

Array.h:

```
1.  #ifndef ARRAY_H
2.  #define ARRAY_H
3.
4.  #include <assert.h> // для assert()
5.
6.  template <class T> // это шаблон класса с T вместо фактического (передаваемого)
     типа данных
7.  class Array
8.  {
9.  private:
10.     int m_length;
11.     T *m_data;
12.
13. public:
14.     Array()
15.     {
16.         m_length = 0;
17.         m_data = nullptr;
18.     }
19.
20.     Array(int length)
21.     {
22.         m_data = new T[length];
23.         m_length = length;
24.     }
25.
26.     ~Array()
27.     {
28.         delete[] m_data;
29.     }
30.
31.     void Erase()
32.     {
33.         delete[] m_data;
34.         // Присваиваем значение nullptr для m_data, чтобы на выходе не получить
    висячий указатель!
35.         m_data = nullptr;
36.         m_length = 0;
37.     }
38.
39.
40.     T& operator[](int index)
41.     {
42.         assert(index >= 0 && index < m_length);
43.         return m_data[index];
```

```
    }

    // Длина массива всегда является целочисленным значением, она не зависит от типа элементов массива
    int getLength(); // определяем метод и шаблон метода getLength() ниже
};

template <typename T> // метод, определенный вне тела класса, нуждается в собственном определении шаблона метода
int Array<T>::getLength() { return m_length; } // обратите внимание, имя класса - Array<T>, а не просто Array

#endif
```

Как вы можете видеть, эта версия почти идентична версии ArrayInt, за исключением того, что мы добавили объявление параметра шаблона класса и изменили тип данных с int на `T`.

Обратите внимание, мы определили функцию getLength() вне тела класса. Это необязательно, но новички обычно спотыкаются на этом из-за синтаксиса. Каждый метод шаблона класса, объявленный вне тела класса, нуждается в собственном объявлении шаблона. Также обратите внимание, что имя шаблона класса — `Array<T>`, а не Array (Array будет указывать на не шаблонную версию класса Array).

Вот пример использования шаблона класса Array:

```
#include <iostream>
#include "Array.h"

int main()
{
    Array<int> intArray(10);
    Array<double> doubleArray(10);

    for (int count = 0; count < intArray.getLength(); ++count)
    {
        intArray[count] = count;
        doubleArray[count] = count + 0.5;
    }

    for (int count = intArray.getLength()-1; count >= 0; --count)
        std::cout << intArray[count] << "\t" << doubleArray[count] << '\n';

    return 0;
}
```

Результат:

```
9       9.5
8       8.5
7       7.5
6       6.5
5       5.5
```

```
4       4.5
3       3.5
2       2.5
1       1.5
0       0.5
```

Шаблоны классов работают точно так же, как и шаблоны функций: компилятор копирует шаблон класса, заменяя типы параметров шаблона класса на фактические (передаваемые) типы данных, а затем компилирует эту копию. Если у вас есть шаблон класса, но вы его не используете, то компилятор не будет его даже компилировать.

Шаблоны классов идеально подходят для реализации контейнерных классов, так как очень часто таким классам приходится работать с разными типами данных, а шаблоны позволяют это организовать в минимальном количестве кода. Хотя синтаксис несколько уродлив, и сообщения об ошибках иногда могут быть «объемными», шаблоны классов действительно являются одной из лучших и наиболее полезных конструкций языка C++.

Шаблоны классов в Стандартной библиотеке C++

Теперь вы уже поняли, чем на самом деле является std::vector<int>? Правильно, std::vector — это шаблон класса, а int — это всего лишь передаваемый тип данных! Стандартная библиотека C++ полна предопределенных шаблонов классов, доступных для вашего использования.

Шаблоны классов и Заголовочные файлы

Шаблон не является ни классом, ни функцией - это трафарет, используемый для создания классов или функций. Таким образом, шаблоны работают не так, как обычные функции или классы. В большинстве случаев это не является проблемой, но на практике случаются разные ситуации.

Работая с обычными классами мы помещаем определение класса в заголовочный файл, а определения методов этого класса в отдельный файл .cpp с аналогичным именем. Таким образом, фактическое определение класса компилируется как отдельный файл внутри проекта. Однако с шаблонами всё происходит несколько иначе. Рассмотрим следующее:

Array.h:

```
1. #ifndef ARRAY_H
2. #define ARRAY_H
```

```
3.
4.  #include <assert.h> // для assert()
5.
6.  template <class T>
7.  class Array
8.  {
9.  private:
10.     int m_length;
11.     T *m_data;
12.
13. public:
14.     Array()
15.     {
16.         m_length = 0;
17.         m_data = nullptr;
18.     }
19.
20.     Array(int length)
21.     {
22.         m_data = new T[length];
23.         m_length = length;
24.     }
25.
26.     ~Array()
27.     {
28.         delete[] m_data;
29.     }
30.
31.     void Erase()
32.     {
33.         delete[] m_data;
34.         // Присваиваем значение nullptr для m_data, чтобы на выходе не получить
    висячий указатель!
35.         m_data = nullptr;
36.         m_length = 0;
37.     }
38.
39.
40.     T& operator[](int index)
41.     {
42.         assert(index >= 0 && index < m_length);
43.         return m_data[index];
44.     }
45.
46.     // Длина массива всегда является целочисленным значением, она не зависит от
    типа элементов массива
47.     int getLength();
48. };
49.
50. #endif
```

Array.cpp:

```
1.  #include "Array.h"
2.
3.  template <typename T>
4.  int Array<T>::getLength() { return m_length; }
```

main.cpp:

```
#include "Array.h"

int main()
{
    Array<int> intArray(10);
    Array<double> doubleArray(10);

    for (int count = 0; count < intArray.getLength(); ++count)
    {
        intArray[count] = count;
        doubleArray[count] = count + 0.5;
    }

    for (int count = intArray.getLength()-1; count >= 0; --count)
        std::cout << intArray[count] << "\t" << doubleArray[count] << '\n';

    return 0;
}
```

Вышеприведенная программа скомпилируется, но вызовет следующую ошибку линкера:

```
unresolved external symbol "public: int __thiscall
Array::getLength(void)" (?GetLength@?$Array@H@@QAEHXZ)
```

Почему так? Сейчас разберемся.

Для использования шаблона компилятор должен видеть как определение шаблона (а не только объявление), так и тип шаблона, применяемый для создания экземпляра шаблона. Помним, что язык C++ компилирует файлы по отдельности. Когда заголовочный файл Array.h подключается в main.cpp, то определение шаблона класса копируется в этот файл. В main.cpp компилятор видит, что нам нужны два экземпляра шаблона класса: `Array<int>` и `Array<double>`, он создаст их, а затем скомпилирует весь этот код как часть файла main.cpp. Однако, когда дело дойдет до компиляции Array.cpp (отдельным файлом), компилятор забудет, что мы использовали `Array<int>` и `Array<double>` в main.cpp и не создаст экземпляр шаблона функции getLength(), который нам нужен для выполнения программы. Мы получим ошибку линкера, так как компилятор не сможет найти определение `Array<int>::getLength()` или `Array<double>::getLength()`.

Эту проблему можно решить несколькими способами.

Самый простой вариант - поместить код из Array.cpp в Array.h ниже класса. Таким образом, когда мы будем подключать Array.h, весь код шаблона класса (полное объявление и определение как класса, так и его методов) будет находиться в одном

месте. Плюс этого способа — простота. Минус — если шаблон класса используется во многих местах, то мы получим много локальных копий шаблона класса, что увеличит время компиляции и линкинга файлов (линкер должен будет удалить дублирование определений класса и методов, дабы исполняемый файл не был «слишком раздутым»). Рекомендуется использовать это решение до тех пор, пока время компиляции или линкинга не является проблемой.

Если вы считаете, что размещение кода из Array.cpp в Array.h сделает Array.h слишком большим/беспорядочным, то альтернативой будет переименование Array.cpp в Array.inl (***.inl*** от англ. *"**inl**ine" = "встроенный"*), а затем подключение Array.inl из нижней части файла Array.h. Это даст тот же результат, что и размещение всего кода в заголовочном файле, но таким образом код получится немного чище.

Есть еще решение - подключение файлов .cpp, но этот вариант не рекомендуется использовать из-за нестандартного применения директивы #include.

Еще один альтернативный вариант - **использовать подход 3-х файлов**:

- Определение шаблона класса хранится в заголовочном файле.
- Определения методов шаблона класса хранятся в отдельном файле .cpp.
- Затем добавляем третий файл, который содержит все необходимые нам экземпляры шаблона класса.

Например, templates.cpp:

```
1. // Таким образом, мы гарантируем, что компилятор увидит полное определение шаблона класса Array
2. #include "Array.h"
3. #include "Array.cpp" // мы нарушаем правила хорошего тона в программировании, но только в этом месте
4.
5. // Здесь вы #include другие файлы .h и .cpp с определениями шаблонов, которые вам нужны
6.
7. template class Array<int>; // явно создаем экземпляр шаблона класса Array<int>
8. template class Array<double>; // явно создаем экземпляр шаблона класса Array<double>
9.
10. // Здесь вы явно создаете другие экземпляры шаблонов, которые вам нужны
```

Часть `template class` заставит компилятор явно создать указанные экземпляры шаблона класса. В примере, приведенном выше, компилятор создаст `Array<int>` и `Array<double>` внутри templates.cpp. Поскольку templates.cpp находится внутри нашего проекта, то он скомпилируется и удачно свяжется с другими файлами (пройдет линкинг).

Этот метод более эффективен, но требует создания/поддержки третьего файла (templates.cpp) для каждой из ваших программ (проектов) отдельно.

Урок №184. Параметр non-type в шаблоне

На предыдущих уроках мы узнали, как использовать параметр типа шаблона для создания функций и классов, которые не зависят от определенного типа данных. Однако параметр типа не является единственным параметром, который может иметь шаблон. Шаблоны классов и функций могут иметь еще один параметр, известный как параметр non-type.

Параметр non-type

Параметр non-type в шаблоне — это специальный параметр шаблона, который заменяется не типом данных, а конкретным значением. Этим значением может быть:

- целочисленное значение или перечисление;
- указатель или ссылка на объект класса;
- указатель или ссылка на функцию;
- указатель или ссылка на метод класса;
- std::nullptr_t.

В следующем примере мы создадим шаблон класса StaticArray, который использует как параметр типа, так и параметр non-type. Параметр типа отвечает за тип данных элементов статического массива, а параметр non-type отвечает за размер выделяемого массива:

```
1. #include <iostream>
2.
3. template <class T, int size> // size является параметром non-type в шаблоне класса
4. class StaticArray
5. {
6. private:
7.     // Параметр non-type в шаблоне класса отвечает за размер выделяемого массива
8.     T m_array[size];
9.
10. public:
11.     T* getArray();
12.
13.     T& operator[](int index)
14.     {
15.         return m_array[index];
16.     }
17. };
18.
19. // Синтаксис определения шаблона метода и самого метода вне тела класса с параметром non-type
20. template <class T, int size>
```

```
21. T* StaticArray<T, size>::getArray()
22. {
23.     return m_array;
24. }
25. 
26. int main()
27. {
28.     // Объявляем целочисленный массив из 10 элементов
29.     StaticArray<int, 10> intArray;
30. 
31.     // Заполняем массив значениями
32.     for (int count=0; count < 10; ++count)
33.         intArray[count] = count;
34. 
35.     // Выводим элементы массива в обратном порядке
36.     for (int count=9; count >= 0; --count)
37.         std::cout << intArray[count] << " ";
38.     std::cout << '\n';
39. 
40.     // Объявляем массив типа double из 5 элементов
41.     StaticArray<double, 5> doubleArray;
42. 
43.     // Заполняем массив значениями
44.     for (int count=0; count < 5; ++count)
45.         doubleArray[count] = 5.5 + 0.1*count;
46. 
47.     // Выводим элементы массива
48.     for (int count=0; count < 5; ++count)
49.         std::cout << doubleArray[count] << ' ';
50. 
51.     return 0;
52. }
```

Результат выполнения программы:

```
9 8 7 6 5 4 3 2 1 0
5.5 5.6 5.7 5.8 5.9
```

Примечательно то, что нам не пришлось динамически выделять переменную-член `m_array`! Это связано с тем, что для любого созданного объекта класса StaticArray его размер является конкретно заданным значением (можно сказать константой), которое передает пользователь. Например, если мы создадим экземпляр `StaticArray<int, 10>`, то компилятор заменит переменную размера массива (`size`) на 10. Таким образом, мы получим `m_array` типа `int[10]`, который можно выделить статическим образом.

Эту особенность использует уже известный нам класс из Стандартной библиотеки C++ - std::array. Когда мы выделяем `std::array<int, 5>`, то `int` является параметром типа, а `5` — параметром non-type в шаблоне класса!

Урок №185. Явная специализация шаблона функции

При создании экземпляра шаблона функции для определенного типа данных компилятор копирует шаблон функции и заменяет параметр типа шаблона функции на фактический (передаваемый) тип данных. Это означает, что все экземпляры функции имеют одну реализацию, но разные типы данных. Хотя в большинстве случаев это именно то, что требуется, иногда может понадобиться, чтобы реализация шаблона функции для одного типа данных отличалась от реализации шаблона функции для другого типа данных.

Специализация шаблонов именно для этого и предназначена.

Рассмотрим очень простой шаблон класса:

```
template <class T>
class Repository
{
private:
    T m_value;
public:
    Repository(T value)
    {
        m_value = value;
    }

    ~Repository()
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};
```

Вышеприведенный код работает со многими типами данных:

```
int main()
{
    // Инициализируем объекты класса
    Repository<int> nValue(7);
    Repository<double> dValue(8.4);

    // Выводим значения объектов класса
    nValue.print();
    dValue.print();
}
```

Результат:

```
7
8.4
```

Теперь, предположим, что нам нужно, чтобы значения типа double (только типа double) выводились в экспоненциальной записи. Для этого мы можем использовать **специализацию шаблона функции** (или ***«полную/явную специализацию шаблона функции»***) для создания отдельной версии функции print() для вывода значений типа double.

Всё просто: записываем экземпляр шаблона функции (если функция является методом класса, то делаем это за пределами класса), указывая нужный нам тип данных. Например, вот специальный шаблон функции print() для значений типа double:

```
template <>
void Repository<double>::print()
{
    std::cout << std::scientific << m_value << '\n';
}
```

Когда компилятору нужно будет создать экземпляр `Repository<double>::print()`, он увидит, что мы уже явно определили эту функцию, и поэтому он будет использовать именно этот экземпляр, а не копировать общую для всех типов данных версию шаблона функции print().

Часть `template <>` сообщает компилятору, что это шаблон функции, но без параметров (так как в этом случае мы явно указываем нужный нам тип данных).

Результат выполнения программы:

```
7
8.400000e+00
```

Еще один пример

Рассмотрим еще один случай, где специализация шаблонов функций может быть полезна. Например, что произойдет, если мы попытаемся использовать наш шаблон класса Repository с типом данных char*?

```
int main()
{
    // Динамически выделяем временную строку
    char *string = new char[40];

    // Просим пользователя ввести свое имя
    std::cout << "Enter your name: ";
    std::cin >> string;

    // Сохраняем то, что ввел пользователь
    Repository<char*> repository(string);

```

```
    // Удаляем временную строку
    delete[] string;

    // Пытаемся вывести то, что ввел пользователь
    repository.print(); // получаем мусор
}
```

Оказывается, вместо вывода имени пользователя, `repository.print()` выведет мусор! Почему?

При создании экземпляра шаблона для типа char*, конструктор `Repository<char*>` выглядит следующим образом:

```
template <>
Repository<char*>::Repository(char* value)
{
     m_value = value;
}
```

Другими словами, это просто присваивание указателя (поверхностное копирование)! В результате `m_value` указывает на ту же область памяти, что и переменная `string`. А когда мы удаляем `string` в main(), то мы удаляем значение, на которое указывает и `m_value`! Таким образом, происходит утечка памяти, и мы получаем мусор при попытке вывода `m_value`.

К счастью, мы можем это исправить, используя явную специализацию шаблона функции. Вместо копирования указателя на `string`, нам нужно, чтобы конструктор копировал само значение `string`. Напишем отдельный конструктор для типа данных char*, который будет именно это и делать:

```
template <>
Repository<char*>::Repository(char* value)
{
    // Определяем длину value
    int length=0;
    while (value[length] != '\0')
        ++length;
    ++length; // +1, учитывая нуль-терминатор

    // Выделяем память для хранения значения value
    m_value = new char[length];

    // Копируем фактическое значение value в m_value
    for (int count=0; count < length; ++count)
        m_value[count] = value[count];
}
```

Теперь при выделении переменной типа `Repository<char*>` именно этот конструктор будет использоваться вместо стандартного. В результате `m_value` получит свою собственную копию `string`. Следовательно, когда мы удалим `string` в main(), `m_value` это никак не заденет.

Однако, теперь класс имеет утечку памяти для типа char*, поскольку `m_value` не будет удален, когда переменная `repository` выйдет из области видимости. Как вы уже могли догадаться, это также можно решить, сделав отдельный деструктор для типа char*:

```
template <>
Repository<char*>::~Repository()
{
    delete[] m_value;
}
```

Теперь, когда переменные типа `Repository<char*>` выйдут из области видимости, память, выделенная в специальном конструкторе, будет удалена в специальном деструкторе.

Хотя во всех примерах, приведенных выше, мы работаем с методами класса, вы также можете аналогично выполнять явную специализацию шаблонов обычных функций (которые не являются методами классов).

Урок №186. Явная специализация шаблона класса

На предыдущем уроке мы говорили о том, как специализировать шаблон функции, чтобы при работе с одним типом данных была одна реализация функции, а при работе с другим типом данных — другая реализация функции. Оказывается, мы можем специализировать не только шаблоны функций, но и шаблоны классов.

Рассмотрим класс-массив, который может хранить 8 объектов:

```
template <class T>
class Repository8
{
private:
    T m_array[8];

public:
    void set(int index, const T &value)
    {
        m_array[index] = value;
    }

    const T& get(int index)
    {
        return m_array[index];
    }
};
```

Поскольку это шаблон класса, то он будет работать с любым типом данных:

```
#include <iostream>

int main()
{
    // Объявляем целочисленный объект-массив
    Repository8<int> intRepository;

    for (int count=0; count<8; ++count)
        intRepository.set(count, count);

    for (int count=0; count<8; ++count)
        std::cout << intRepository.get(count) << '\n';

    // Объявляем объект-массив типа bool
    Repository8<bool> boolRepository;

    for (int count=0; count<8; ++count)
        boolRepository.set(count, count % 5);

    for (int count=0; count<8; ++count)
        std::cout << (boolRepository.get(count) ? "true" : "false") << '\n';

    return 0;
}
```

Результат:

```
0
1
2
3
4
5
6
7
false
true
true
true
true
false
true
true
```

Хотя всё работает правильно, реализация `Repository8<bool>`, на самом деле, не столь эффективна, какой она могла бы быть. Поскольку все переменные должны иметь адрес, а ЦП не может дать адрес чему-либо меньшему, чем 1 байт, то размер всех переменных должен быть не менее 1 байта. Следовательно, каждая переменная типа bool занимает целый байт, хотя технически ей нужен только 1 бит для хранения значения true или false! Таким образом, переменная типа bool - это 1 бит полезной информации и 7 бит потраченного впустую места. Получается, что наш класс `Repository8<bool>`, который имеет 8 переменных типа bool, фактически работает только с 1 байтом данных, а остальные 7 байтов тратятся зря.

Выход есть: мы можем сжать 8 переменных типа bool в 1 байт, сэкономив при этом остальные 7 байтов. Однако для этого нам нужно будет изменить реализацию класса, заменив массив из 8 переменных типа bool (8 байтов) на 1 переменную типа unsigned char (1 байт). Мы могли бы, конечно, использовать для этого новый отдельный класс, но это неэффективно + программисту придется помнить, что `Repository8<T>` работает со всеми типами данных, кроме bool, а при работе с bool следует вызывать `Repository8Bool` (неважно, какое имя будет у этого класса). Это лишняя работа, которую можно избежать, используя явную специализацию шаблона класса.

Специализация шаблона класса

Специализация шаблона класса (или ***«явная специализация шаблона класса»***) позволяет специализировать шаблон класса для работы с определенным типом

данных (или сразу с несколькими типами данных, если есть несколько параметров шаблона).

Специализация шаблона класса рассматривается компилятором как полностью отдельный и независимый класс, хоть и выделяется как обычный шаблон класса. Это означает, что мы можем изменить в классе всё что угодно, включая его реализацию/методы/спецификаторы доступа и т.д.

Рассмотрим специализацию шаблона класса Repository8 для работы с типом bool:

```
1. template <>
2. class Repository8<bool> // специализируем шаблон класса Repository8 для работы с типом bool
3. {
4. // Реализация класса
5. private:
6.     unsigned char m_data;
7.
8. public:
9.     Repository8() : m_data(0)
10.    {
11.    }
12.
13.    void set(int index, bool value)
14.    {
15.        // Выбираем оперируемый бит
16.        unsigned char mask = 1 << index;
17.
18.        if (value) // если на входе у нас true, то бит нужно "включить"
19.            m_data |= mask; // используем побитовое ИЛИ, чтобы "включить" бит
20.        else // если на входе у нас false, то бит нужно "выключить"
21.            m_data &= ~mask; // используем побитовое И, чтобы "выключить" бит
22.    }
23.
24.    bool get(int index)
25.    {
26.        // Выбираем бит
27.        unsigned char mask = 1 << index;
28.        // Используем побитовое И для получения значения бита, а затем выполняется его неявное преобразование в тип bool
29.        return (m_data & mask) != 0;
30.    }
31. };
```

Во-первых, начинаем с `template<>`. **Ключевое слово template** сообщает компилятору, что это шаблон, а пустые угловые скобки означают, что нет никаких параметров. А параметров нет из-за того, что мы заменяем единственный параметр шаблона (`T`, который отвечает за тип данных) конкретным типом данных (bool). Затем мы пишем имя класса и добавляем к нему `<bool>`, сообщая компилятору, что будем работать с типом bool.

Все остальные изменения - это просто детали реализации класса. Обратите внимание, вместо массива из 8 переменных типа bool (8 байтов) мы используем 1 переменную типа unsigned char (1 байт).

Теперь при объявлении объекта класса `Repository8<T>`, где `T` не является bool, мы получим экземпляр общего шаблона класса, тогда как при объявлении объекта `Repository8<bool>`, мы получим экземпляр шаблона `Repository8<bool>`. Обратите внимание, мы не изменяли интерфейс класса, а оставили его открытым (каким он и был), в то время как язык C++ предоставляет нам возможность добавлять/изменять/удалять методы класса. Дело в том, что изменение интерфейса класса в специализациях шаблона не всегда приветствуется, так как программист может это дело забыть, а оно, в свою очередь, приведет к ошибкам.

Создадим объекты `Repository8<T>` и `Repository8<bool>`:

```
1. int main()
2. {
3.     // Объявляем целочисленный объект-массив (создается экземпляр
   Repository8<T>, где T = int)
4.     Repository8<int> intRepository;
5.
6.     for (int count=0; count<8; ++count)
7.         intRepository.set(count, count);
8.
9.     for (int count=0; count<8; ++count)
10.        std::cout << intRepository.get(count) << '\n';
11.
12.    // Объявляем объект-массив типа bool (создается экземпляр
   специализации Repository8<bool>)
13.    Repository8<bool> boolRepository;
14.
15.    for (int count=0; count<8; ++count)
16.        boolRepository.set(count, count % 5);
17.
18.    for (int count=0; count<8; ++count)
19.        std::cout << (boolRepository.get(count) ? "true" : "false") << '\n';
20.
21.    return 0;
22.}
```

Как вы можете видеть, результат тот же, что и выше, где использовался общий шаблон класса Repository8:

```
0
1
2
3
4
5
6
```

```
7
false
true
true
true
true
false
true
true
```

Следует еще раз отметить, что сохранение открытого интерфейса в шаблонах ваших классов вместе со специализациями, как правило, является хорошей идеей, поскольку это упрощает использование классов и их логику, хоть и не считается строго необходимым.

Урок №187. Частичная специализация шаблона

На уроке №184 мы узнали, каким образом можно использовать дополнительный параметр шаблона. Рассмотрим еще раз класс StaticArray из материалов того же урока:

```
1. template <class T, int size> // size является non-type параметром шаблона
2. class StaticArray
3. {
4. private:
5.     // Параметр size отвечает за длину массива
6.     T m_array[size];
7. 
8. public:
9.     T* getArray() { return m_array; }
10. 
11.     T& operator[](int index)
12.     {
13.         return m_array[index];
14.     }
15. };
```

Здесь у нас есть 2 параметра шаблона класса: параметр типа и параметр non-type.

Теперь предположим, что нам нужно написать функцию для вывода всех элементов массива. Хотя мы можем сделать это через метод класса, мы реализуем это через отдельную функцию (ради лучшего погружения в тему).

Используя шаблон функции, мы можем написать следующее:

```
1. template <typename T, int size>
2. void print(StaticArray<T, size> &array)
3. {
4.     for (int count=0; count < size; ++count)
5.         std::cout << array[count] << ' ';
6. }
```

Это позволит нам сделать:

```
1. #include <iostream>
2. #include <cstring>
3. 
4. template <class T, int size> // size является non-type параметром шаблона
5. class StaticArray
6. {
7. private:
8.     // Параметр size отвечает за длину массива
9.     T m_array[size];
10. 
11. public:
12.     T* getArray() { return m_array; }
13. 
14.     T& operator[](int index)
```

```
    {
        return m_array[index];
    }
};

template <typename T, int size>
void print(StaticArray<T, size> &array)
{
    for (int count = 0; count < size; ++count)
        std::cout << array[count] << ' ';
}

int main()
{
    // Объявляем целочисленный массив
    StaticArray<int, 5> int5;
    int5[0] = 0;
    int5[1] = 1;
    int5[2] = 2;
    int5[3] = 3;
    int5[4] = 4;

    // Выводим элементы массива
    print(int5);

    return 0;
}
```

И получить:

```
0 1 2 3 4
```

Хотя всё работает правильно, но есть один нюанс. Рассмотрим следующий код функции main():

```
int main()
{
    // Объявляем массив типа char
    StaticArray<char, 14> char14;

    strcpy_s(char14.getArray(), 14, "Hello, world!");

    // Выводим элементы массива
    print(char14);

    return 0;
}
```

(мы рассматривали strcpy_s на уроке о строках C-style)

Программа скомпилируется со следующим результатом:

```
H e l l o ,   w o r l d !
```

Для всех типов, кроме char, имеет смысл помещать пробел между каждым элементом массива, чтобы элементы не «слипались». Однако с типом char есть

смысл вывести всё вместе, как строку C-style, чтобы не было лишних пробелов. Как мы можем это исправить?

Полная специализация шаблона - решение?

Сначала мы могли бы подумать об использовании специализации шаблона функции. Однако **проблема с полной специализацией шаблона заключается в том, что все параметры шаблона должны быть явно определены**. Например:

```
#include <iostream>
#include <cstring>

template <class T, int size> // size является non-type параметром шаблона
class StaticArray
{
private:
    // Параметр size отвечает за длину массива
    T m_array[size];

public:
    T* getArray() { return m_array; }

    T& operator[](int index)
    {
        return m_array[index];
    }
};

template <typename T, int size>
void print(StaticArray<T, size> &array)
{
    for (int count = 0; count < size; ++count)
        std::cout << array[count] << ' ';
}

// Шаблон функции print() с полной специализацией шаблона класса StaticArray для работы с типом char и длиной массива 14
template <>
void print(StaticArray<char, 14> &array)
{
    for (int count = 0; count < 14; ++count)
        std::cout << array[count];
}

int main()
{
    // Объявляем массив типа char
    StaticArray<char, 14> char14;

    strcpy_s(char14.getArray(), 14, "Hello, world!");

    // Выводим элементы массива
    print(char14);

    return 0;
}
```

Как вы можете видеть, мы добавили шаблон функции print() для работы с типом char. Результат:

```
Hello, world!
```

Хотя одна проблема решена, возникает другая проблема: использование полной специализации шаблона класса означает, что мы должны явно указывать длину передаваемого массива! Рассмотрим следующий пример:

```
1. int main()
2. {
3.     // Объявляем массив типа char
4.     StaticArray<char, 12> char12;
5.
6.     strcpy_s(char12.getArray(), 12, "Hello, dad!");
7.
8.     // Выводим элементы массива
9.     print(char12);
10.
11.    return 0;
12.}
```

Вызов `print(char12)` вызовет шаблон функции print() с общим шаблоном `StaticArray<T, size>`, так как char12 является типа `StaticArray<char, 12>`, а шаблон функции print() принимает только `StaticArray<char, 14>` (длина массива отличается).

Хотя мы могли бы скопировать еще раз шаблон функции print() для работы со `StaticArray<char, 12>`, но это неэффективно. А что, если нам нужно будет позднее использовать массив с 5 или 20 элементами? Опять копировать шаблон? Это лишняя работа.

Очевидно, что полная специализация шаблона класса здесь является решением-костылем. Частичная специализация шаблона — вот, что нам нужно.

Частичная специализация шаблона

Частичная специализация шаблона позволяет выполнить специализацию шаблона класса (но не функции!), где некоторые (но не все) параметры шаблона явно определены. Для нашей вышеприведенной задачи идеальное решение заключается в том, чтобы шаблон функции print() работал со StaticArray типа char, но при этом размер массива не являлся фиксированным значением, а мог варьироваться.

Вот наш шаблон функции print(), который принимает частично специализированный шаблон класса StaticArray:

```
1. // Шаблон функции print() с частично специализированным шаблоном класса
   StaticArray<char, size> в качестве параметра
2. template <int size> // size по-прежнему является non-type параметром
3. void print(StaticArray<char, size> &array) // мы здесь явно указываем тип char
4. {
5.     for (int count = 0; count < size; ++count)
6.         std::cout << array[count];
7. }
```

Как вы можете видеть, мы здесь явно указали тип char, но `size` оставили не фиксированным, поэтому функция print() будет работать с массивами типа char любого размера. Вот и всё!

Полный код программы:

```
1. #include <iostream>
2. #include <cstring>
3.
4. template <class T, int size> // size является non-type параметром шаблона
5. class StaticArray
6. {
7. private:
8.     // Параметр size отвечает за длину массива
9.     T m_array[size];
10.
11. public:
12.     T* getArray() { return m_array; }
13.
14.     T& operator[](int index)
15.     {
16.         return m_array[index];
17.     }
18. };
19.
20. template <typename T, int size>
21. void print(StaticArray<T, size> &array)
22. {
23.     for (int count = 0; count < size; ++count)
24.         std::cout << array[count] << ' ';
25. }
26.
27. // Шаблон функции print() с частично специализированным шаблоном класса
   StaticArray<char, size> в качестве параметра
28. template <int size>
29. void print(StaticArray<char, size> &array)
30. {
31.     for (int count = 0; count < size; ++count)
32.         std::cout << array[count];
33. }
34.
35. int main()
36. {
37.     // Объявляем массив типа char длиной 14
38.     StaticArray<char, 14> char14;
39.
```

```
40.     strcpy_s(char14.getArray(), 14, "Hello, world!");
41.
42.     // Выводим элементы массива
43.     print(char14);
44.
45.     // Теперь объявляем массив типа char длиной 12
46.     StaticArray<char, 12> char12;
47.
48.     strcpy_s(char12.getArray(), 12, "Hello, dad!");
49.
50.     // Выводим элементы массива
51.     print(char12);
52.
53.     return 0;
54. }
```

Результат:

```
Hello, world! Hello, dad!
```

Как и ожидалось.

Обратите внимание, начиная с C++14 частичная специализация шаблона может использоваться только с классами, но не с отдельными функциями (для функций используется только полная специализация шаблона). Наш пример `void print(StaticArray<char, size> & array)` работает только потому, что шаблон функции print() принимает в качестве параметра шаблон класса, который, в свою очередь, частично специализирован.

Частичная специализация шаблонов методов

Ограничение частичной специализации для функций может привести к некоторым проблемам при работе с методами класса. Например, что, если бы мы определили StaticArray следующим образом:

```
1.  template <class T, int size> // size является non-type параметром шаблона
2.  class StaticArray
3.  {
4.  private:
5.      // Параметр size отвечает за длину массива
6.      T m_array[size];
7.
8.  public:
9.      T* getArray() { return m_array; }
10.
11.     T& operator[](int index)
12.     {
13.         return m_array[index];
14.     }
15.
16.     void print()
17.     {
18.         for (int i = 0; i < size; i++)
```

```
            std::cout << m_array[i] << ' ';
        std::cout << "\n";
    }
};
```

Функция print() является методом класса `StaticArray<T, int>`. Что произойдет, если мы захотим частично специализировать шаблон функции print(), чтобы метод работал по-другому? Мы можем попробовать сделать следующее:

```
// Не сработает
template <int size>
void StaticArray<double, size>::print()
{
    for (int i = 0; i < size; i++)
        std::cout << std::scientific << m_array[i] << " ";
    std::cout << "\n";
}
```

К сожалению, это не сработает, так как мы пытаемся частично специализировать шаблон функции, что делать запрещено.

Как же это можно обойти? Одним из очевидных решений является частичная специализация шаблона всего класса:

```
#include <iostream>

template <class T, int size> // size является non-type параметром шаблона
class StaticArray
{
private:
    // Параметр size отвечает за длину массива
    T m_array[size];

public:
    T* getArray() { return m_array; }

    T& operator[](int index)
    {
        return m_array[index];
    }
    void print()
    {
        for (int i = 0; i < size; i++)
            std::cout << m_array[i] << ' ';
        std::cout << "\n";
    }
};

template <int size> // size является non-type параметром шаблона
class StaticArray<double, size>
{
private:
    // Параметр size отвечает за длину массива
    double m_array[size];

public:
    double* getArray() { return m_array; }
```

```
34.
35.     double& operator[](int index)
36.     {
37.         return m_array[index];
38.     }
39.     void print()
40.     {
41.         for (int i = 0; i < size; i++)
42.             std::cout << std::scientific << m_array[i] << ' ';
43.         std::cout << "\n";
44.     }
45. };
46.
47. int main()
48. {
49.     // Объявляем целочисленный массив длиной 5
50.     StaticArray<int, 5> intArray;
51.
52.     // Заполняем массив, а затем выводим его
53.     for (int count = 0; count < 5; ++count)
54.         intArray[count] = count;
55.     intArray.print();
56.
57.     // Объявляем массив типа double длиной 4
58.     StaticArray<double, 4> doubleArray;
59.
60.     for (int count = 0; count < 4; ++count)
61.         doubleArray[count] = (4.0 + 0.1 * count);
62.     doubleArray.print();
63.
64.     return 0;
65. }
```

Результат:

```
0 1 2 3 4
4.000000e+00 4.100000e+00 4.200000e+00 4.300000e+00
```

Хотя это работает, но это не самый лучший вариант, так как у нас теперь куча дублированного кода из `StaticArray<T, size>` в `StaticArray<double, size>`.

Если бы можно было использовать код из `StaticArray<T, size>` в `StaticArray<double, size>` без дублирования. Ничего вам это не напоминает? Как по мне, то это звучит, как отличный вариант для применения наследования!

Вы можете начать с:

```
1. template <int size> // size является non-type параметром шаблона
2. class StaticArray<double, size>: public StaticArray< // а затем что?
```

Но как мы можем ссылаться на StaticArray? Никак, но, к счастью, есть обходной путь с использованием общего родительского класса:

```
#include <iostream>

template <class T, int size> // size является non-type параметром шаблона
class StaticArray_Base
{
protected:
    // Параметр size отвечает за длину массива
    T m_array[size];

public:
    T* getArray() { return m_array; }

    T& operator[](int index)
    {
        return m_array[index];
    }
    virtual void print()
    {
        for (int i = 0; i < size; i++)
            std::cout << m_array[i] << ' ';
        std::cout << "\n";
    }
};

template <class T, int size> // size является non-type параметром шаблона
class StaticArray: public StaticArray_Base<T, size>
{
public:
    StaticArray()
    {

    }
};

template <int size> // size является non-type параметром шаблона
class StaticArray<double, size>: public StaticArray_Base<double, size>
{
public:

    virtual void print() override
    {
        for (int i = 0; i < size; i++)
            std::cout << std::scientific << this->m_array[i] << " ";
// Примечание: Префикс this-> на вышеприведенной строке необходим.
   Почему? Читайте здесь - https://stackoverflow.com/a/6592617
        std::cout << "\n";
    }
};

int main()
{
    // Объявляем целочисленный массив длиной 5
    StaticArray<int, 5> intArray;

    // Заполняем его, а затем выводим
    for (int count = 0; count < 5; ++count)
        intArray[count] = count;
```

```
57.     intArray.print();
58.
59.     // Объявляем массив типа double длиной 4
60.     StaticArray<double, 4> doubleArray;
61.
62.     // Заполняем его, а затем выводим
63.     for (int count = 0; count < 4; ++count)
64.         doubleArray[count] = (4. + 0.1*count);
65.     doubleArray.print();
66.
67.     return 0;
68. }
```

Результат тот же, что и в примере, приведенном выше, но дублированного кода меньше.

Урок №188. Частичная специализация шаблонов и Указатели

На уроке о специализации шаблона функции мы рассматривали шаблон класса Repository:

```
#include <iostream>

template <class T>
class Repository
{
private:
    T m_value;
public:
    Repository(T value)
    {
        m_value = value;
    }

    ~Repository()
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};
```

Мы говорили о проблеме этого шаблона при работе с типом char*, когда выполнялось поверхностное копирование (присваивание указателя) в конструкторе класса Repository. В качестве решения мы использовали полную специализацию шаблона для создания специализированной версии конструктора класса Repository для работы с типом char*, в котором выделялась память и выполнялось глубокое копирование `m_value`. Вот специализация конструктора и деструктора класса Repository для работы с типом char* (из материалов того же урока):

```
template <>
Repository<char*>::Repository(char* value)
{
    // Определяем длину value
    int length=0;
    while (value[length] != '\0')
        ++length;
    ++length; // +1, учитывая нуль-терминатор

    // Выделяем память для хранения значения value
    m_value = new char[length];

    // Копируем фактическое значение из value в m_value
    for (int count=0; count < length; ++count)
        m_value[count] = value[count];
}
```

```
17.
18. template<>
19. Repository<char*>::~Repository()
20. {
21.     delete[] m_value;
22. }
```

Хотя всё отлично работает с типом char*, но как насчет других типов указателей (например, int*)? Поскольку `T` — это любой тип указателя, то при работе с тем же int* выполнится поверхностное копирование (что нам не нужно), либо нам придется дублировать вышеприведенный код (специализация конструктора и деструктора), но уже вместо char* использовать int*. А дублирование кода, как мы уже знаем, не самый лучший вариант!

К счастью, используя частичную специализацию шаблона, мы можем определить специальную версию класса Repository, которая работала бы со всеми типами указателей (при этом не нужно указывать конкретные типы указателей):

```
1.  #include <iostream>
2.
3.  // Общий шаблон класса Repository
4.  template <class T>
5.  class Repository
6.  {
7.  private:
8.      T m_value;
9.  public:
10.     Repository(T value)
11.     {
12.         m_value = value;
13.     }
14.
15.     ~Repository()
16.     {
17.     }
18.
19.     void print()
20.     {
21.         std::cout << m_value << '\n';
22.     }
23. };
24.
25. template <typename T>
26. class Repository<T*> // частичная специализация шаблона класса Repository для
    работы с типами указателей
27. {
28. private:
29.     T* m_value;
30. public:
31.     Repository(T* value) // T - тип указателя
32.     {
33.         // Выполняем глубокое копирование
34.         m_value = new T(*value); // здесь копируется только одно отдельное
    значение (не массив значений)
35.     }
36.
```

```
    ~Repository()
    {
        delete m_value; // а здесь выполняется удаление этого значения
    }

    void print()
    {
        std::cout << *m_value << '\n';
    }
};
```

И пример из практики:

```
int main()
{
    // Объявляем целочисленный объект для проверки работы общего шаблона класса
    Repository<int> myint(6);
    myint.print();

    // Объявляем объект с типом указатель для проверки работы частичной специализации шаблона класса
    int x = 8;
    Repository<int*> myintptr(&x);

    // Если бы в myintptr выполнилось поверхностное копирование (присваивание указателя), то изменение значения x изменило бы и значение myintptr
    x = 10;
    myintptr.print();

    return 0;
}
```

Результат:

```
6
8
```

При объявлении объекта `myintptr` с типом int*, компилятор видит, что мы ранее определили частичную специализацию шаблона класса для работы с типами указателей, и, учитывая, что мы использовали тип int*, компилятор создаст экземпляр частичной специализации шаблона для работы с типом указателя. Конструктор этой специализации выполняет глубокое копирование параметра `x`. Позже, когда мы изменяем значение `x` на `10`, `myintptr.m_value` никак не задевается, так как выполнилось глубокое копирование, при котором `m_value` получил свою собственную копию `x`.

Если бы этой частичной специализации не существовало, то создался бы экземпляр общего шаблона класса, в котором выполнилось бы поверхностное копирование, а `myintptr.m_value`, и `x` указывали бы на один и тот же адрес памяти. В таком случае, при изменении значения переменной `x` на `10`, мы также затронули бы и значение `myintptr` (оно также стало бы равно `10`).

Стоит отметить, что, поскольку в нашей частичной специализации копируется только одно значение, при работе со строками C-style копироваться будет только первый символ (так как строка — это массив, а указатель на массив указывает только на первый элемент массива). Если же нужно полностью скопировать строку, то специализация конструктора (и деструктора) для типа char* должна быть полной. В таком случае, полная специализация будет иметь приоритет выше, чем частичная специализация. Например, вот программа, в которой используется как частичная специализация для работы с типами указателей, так и полная специализация для работы с типом char*:

```
#include <iostream>
#include <cstring>

// Общий шаблон класса Repository для работы с не указателями
template <class T>
class Repository
{
private:
    T m_value;
public:
    Repository(T value)
    {
        m_value = value;
    }

    ~Repository()
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};

// Частичная специализация шаблона класса Repository для работы с указателями
template <class T>
class Repository<T*>
{
private:
    T* m_value;
public:
    Repository(T* value)
    {
        m_value = new T(*value);
    }

    ~Repository()
    {
        delete m_value;
    }

    void print()
    {
        std::cout << *m_value << '\n';
    }
```

```
};

// Полная специализация шаблона конструктора класса Repository для работы с типом char*
template <>
Repository<char*>::Repository(char* value)
{
    // Определяем длину value
    int length = 0;
    while (value[length] != '\0')
        ++length;
    ++length; // +1, учитывая нуль-терминатор

    // Выделяем память для хранения значения value
    m_value = new char[length];

    // Копируем фактическое значение value в m_value
    for (int count = 0; count < length; ++count)
        m_value[count] = value[count];
}

// Полная специализация шаблона деструктора класса Repository для работы с типом char*
template<>
Repository<char*>::~Repository()
{
    delete[] m_value;
}

// Полная специализация шаблона метода print() для работы с типом char*.
// Без этого вывод Repository<char*> привел бы к вызову Repository<T*>::print(), который выводит только одно значение (в случае со строкой C-style - только первый символ)
template<>
void Repository<char*>::print()
{
    std::cout << m_value;
}

int main()
{
    // Объявляем целочисленный объект для проверки работы общего шаблона класса
    Repository<int> myint(6);
    myint.print();

    // Объявляем объект с типом указатель для проверки работы частичной специализации шаблона
    int x = 8;
    Repository<int*> myintptr(&x);

    // Если бы в myintptr выполнилось поверхностное копирование (присваивание указателя), то изменение значения x изменило бы и значение myintptr
    x = 10;
    myintptr.print();

    // Динамически выделяем временную строку
    char *name = new char[40]{ "Anton" }; // необходим C++14

    // Если ваш компилятор не поддерживает C++14, то закомментируйте строку выше и раскомментируйте строки, приведенные ниже
    //  char *name = new char[40];
```

```
101.      //  strcpy(name, "Anton");
102.
103.          // Сохраняем имя
104.          Repository<char*> myname(name);
105.
106.          // Удаляем временную строку
107.          delete[] name;
108.
109.          // Выводим имя
110.          myname.print();
111.      }
```

Всё работает как нужно:

```
6
8
Anton
```

В итоге, использование частичной специализации шаблона класса для работы с типами указателей особенно полезно, так как позволяет предусмотреть все возможные варианты использования кода на практике.

Глава №13. Итоговый тест

Еще одна глава позади. Пора закрепить пройденный материал.

Теория

Шаблоны позволяют написать одну версию функции или класса, которая будет работать с разными типами данных. Функция или класс, реализованная через шаблон, с фактическим (одним) типом данных называется **экземпляром**.

Все шаблоны функций или классов должны начинаться с **ключевого слова template и объявления параметров шаблона**. В объявлении параметров шаблона указываются параметры типа и параметры non-type.

Параметр типа шаблона - это параметр, который отвечает за типы данных, с которыми будет работать шаблон, обычно его называют `T`, `T1`, `T2` или другими (одиночными) буквами (например, `S`).

Параметром non–type может быть переменная интегрального типа данных (например, char, bool, int, long, short), указатель/ссылка на функцию или на метод/объект класса, std::nullptr_t.

Разделение определения шаблонов класса и его методов по разным файлам не работает как с обычными классами - вы не можете поместить определение шаблона класса в заголовочный файл, а определение шаблонов методов этого класса в отдельный файл .cpp. Как правило, лучше всё хранить в заголовочном файле с определениями шаблонов методов под определением шаблона класса.

Явная специализация шаблона используется для определения реализации, отличающейся от общей, функции или класса при работе с определенным типом данных. Если все параметры специализации шаблона явно определены, то это **полная специализация**. Классы также поддерживают **частичную специализацию**, при которой не все параметры шаблона должны быть явно определены. В C++14 частичная специализация шаблонов функций запрещена.

Многие классы в Стандартной библиотеке C++ используют шаблоны, такие как std::array и std::vector. Шаблоны часто применяются для реализации контейнерных классов, которые можно один раз написать и использовать с любыми типами данных.

Тест

Задание №1

Предположим, что нам нужно передавать данные парами. Реализуйте шаблон класса Pair1, который позволяет пользователю передавать данные одного типа парами. Следующий код:

```
1. int main()
2. {
3.     Pair1<int> p1(6, 9);
4.     std::cout << "Pair: " << p1.first() << ' ' << p1.second() << '\n';
5.
6.     const Pair1<double> p2(3.4, 7.8);
7.     std::cout << "Pair: " << p2.first() << ' ' << p2.second() << '\n';
8.
9.     return 0;
10. }
```

Должен выдавать следующий результат:

```
Pair: 6 9
Pair: 3.4 7.8
```

Задание №2

Реализуйте класс Pair, который позволяет пользователю использовать разные типы данных в передаваемых парах. Следующий код:

```
1. int main()
2. {
3.     Pair<int, double> p1(6, 7.8);
4.     std::cout << "Pair: " << p1.first() << ' ' << p1.second() << '\n';
5.
6.     const Pair<double, int> p2(3.4, 5);
7.     std::cout << "Pair: " << p2.first() << ' ' << p2.second() << '\n';
8.
9.     return 0;
10. }
```

Должен выдавать следующий результат:

```
Pair: 6 7.8
Pair: 3.4 5
```

> ***Подсказка***: Для определения шаблона с использованием двух разных типов, просто разделите параметры типа шаблона запятой.

Задание №3

Напишите шаблон класса StringValuePair, в котором первое значение всегда является типа string, а второе может быть любого типа. Этот шаблон класса должен наследовать частично специализированный класс Pair (в котором первый параметр типа std::string, а второй — «любой тип данных»). Следующий код:

```
1. int main()
2. {
3.     StringValuePair<int> svp("Amazing", 7);
4.     std::cout << "Pair: " << svp.first() << ' ' << svp.second() << '\n';
5. 
6.     return 0;
7. }
```

Должен выдавать следующий результат:

```
Pair: Amazing 7
```

> ***Подсказка***: При вызове конструктора класса Pair из конструктора класса StringValuePair, не забудьте указать, что параметры относятся к классу Pair.

Урок №189. Исключения. Зачем они нужны?

Мы уже ранее говорили о механизмах обработки ошибок в языке C++, таких как cerr(), exit() и assert(). Однако мы не успели поговорить о еще одной очень важной теме — "Исключения в языке C++". Сейчас мы это исправим.

Когда коды возврата не работают

При написании повторно используемого кода возникает необходимость в обработке ошибок. Одним из наиболее распространенных способов обработки потенциальных ошибок является использование **кодов возврата** (или ***«кодов завершения»***), которые возвращает оператор return. Например:

```
1. int findFirstChar(const char* string, char ch)
2. {
3.     // Перебираем каждый символ строки
4.     for (int index=0; index < strlen(string); ++index)
5.         // Если текущий символ совпадает со значением переменной ch, то
   возвращаем индекс этого символа
6.         if (string[index] == ch)
7.             return index;
8.
9.     // Если совпадение не найдено, то возвращаем -1
10.    return -1;
11.}
```

Эта функция возвращает индекс первого символа передаваемой строки, совпадающего со значением переменной `ch`. Если символ не найден, то функция возвращает `-1` в качестве индикатора ошибки.

Главным преимуществом этого подхода является его простота. Однако есть ряд недостатков, которые могут быстро проявиться в нетривиальных случаях.

Во-первых, возвращаемые значения не всегда понятны. Если функция возвращает `-1`, обозначает ли это какую-то специфическую ошибку или это корректное возвращаемое значение? Часто бывает трудно это понять, не видя перед глазами код самой функции.

Во-вторых, функции могут возвращать только одно значение. А что, если нам нужно будет возвратить как результат выполнения функции, так и код завершения? Например:

```
1. double divide(int a, int b)
2. {
3.     return static_cast<double>(a)/b;
4. }
```

Здесь нужен механизм обработки ошибок, потому что, если пользователь передаст `0` в качестве параметра `b`, произойдет сбой. Кроме того, функция также должна возвратить и результат выполнения операции `static_cast<double>(a)/b`. Как же это сделать? Один из вариантов — возврат результата операции или кода завершения по ссылке, например:

```
#include <iostream>

double divide(int a, int b, bool &success)
{
    if (b == 0)
    {
        success = false;
        return 0.0;
    }

    success = true;
    return static_cast<double>(a)/b;
}

int main()
{
    bool success;
    double result = divide(7, 4, success); // мы сейчас передаем значение типа bool, чтобы знать заранее, будет ли операция успешной

    if (!success) // проверяем результат выполнения операции перед фактическим использованием result
        std::cerr << "An error occurred" << std::endl;
    else
        std::cout << "The answer is " << result << '\n';
}
```

В-третьих, когда кода много, то многие вещи могут пойти не так, как нужно, поэтому коды возврата нужно постоянно проверять. Рассмотрим следующий фрагмент программы, в котором проводится анализ текстового файла на наличие определенных значений:

```
std::ifstream setupIni("setup.ini"); // открываем setup.ini для чтения
// Если файл нельзя открыть (например, потому что он отсутствует), то возвращаем ошибку
if (!setupIni)
    return ERROR_OPENING_FILE;

// Если же файл можно открыть, то считываем значения из этого файла
if (!readIntegerFromFile(setupIni, m_firstParameter)) // пытаемся найти значение типа int в файле
    return ERROR_READING_VALUE; // возвращаем ошибку, если значение не найдено

if (!readDoubleFromFile(setupIni, m_secondParameter)) // пытаемся найти значение типа double в файле
    return ERROR_READING_VALUE;

if (!readFloatFromFile(setupIni, m_thirdParameter)) // пытаемся найти значение типа float в файле
    return ERROR_READING_VALUE;
```

Мы еще не рассматривали работу с файлами, поэтому не волнуйтесь, если вы не понимаете, как и что здесь работает — просто обратите внимание на то, что для каждого вызова функции требуется проверка и возврат состояния обратно в caller. Теперь представьте, если бы у нас было двадцать параметров разных типов — нам бы пришлось выполнять проверку и возврат `ERROR_READING_VALUE` двадцать раз! Весь этот механизм обработки ошибок только затрудняет понимание (чтение) того, что же на самом деле должна делать эта функция.

В-четвертых, коды возврата не очень хорошо сочетаются с конструкторами. Что произойдет, если мы создадим объект, а внутри конструктора случится что-то катастрофическое? Конструкторы не могут использовать оператор return для возврата индикатора состояния, а передача по ссылке может причинить массу неудобств, и её нужно явно проверять. Кроме того, даже если мы это сделаем, объект все равно создастся, и лечить мы уже будем последствия (либо обрабатывать, либо удалять).

Наконец, при возврате ошибки обратно в caller, сам caller может не всегда быть готовым обработать эту ошибку. Если caller не хочет обрабатывать ошибку, он либо игнорирует её (что уже плохо), либо возвращает ошибку обратно в функцию, от которой он её и получил. Это не то что неудобно, это может привести к сбою программы или к неопределенным результатам.

Основная проблема с кодами возврата заключается в том, что они плотно связаны с общим потоком выполнения кода, а это, в свою очередь, ограничивает наши возможности.

Исключения

Обработка исключений как раз и обеспечивает механизм, позволяющий отделить обработку ошибок или других исключительных обстоятельств от общего потока выполнения кода. Это предоставляет больше свободы в конкретных ситуациях, уменьшая при этом беспорядок, который вызывают коды возврата.

Урок №190. Обработка исключений. Операторы throw, try и catch

На предыдущем уроке мы говорили о необходимости и пользе исключений. Исключения в языке C++ реализованы с помощью 3-х ключевых слов, которые работают в связке друг с другом: **throw**, **try** и **catch**.

Генерация исключений

Мы постоянно используем сигналы в реальной жизни для обозначения того, что произошли определенные события. Например, во время игры в баскетбол, если игрок совершил серьезный фол, то арбитр свистит, и игра останавливается. Затем идет штрафной бросок. Как только штрафной бросок выполнен, игра возобновляется.

В языке C++ **оператор throw** используется для сигнализирования о возникновении исключения или ошибки (аналогия тому, когда свистит арбитр). Сигнализирование о том, что произошло исключение, называется **генерацией исключения** (или ***"выбрасыванием исключения"***).

Для использования оператора throw применяется ключевое слово throw, а за ним указывается значение любого типа данных, которое вы хотите использовать, чтобы сигнализировать об ошибке. Как правило, этим значением является код ошибки, описание проблемы или настраиваемый класс-исключение. Например:

```
1. throw -1; // генерация исключения типа int
2. throw ENUM_INVALID_INDEX; // генерация исключения типа enum
3. throw "Can not take square root of negative number"; // генерация исключения
   типа const char* (строка C-style)
4. throw dX; // генерация исключения типа double (переменная типа double, которая
   была определена ранее)
5. throw MyException("Fatal Error"); // генерация исключения с использованием
   объекта класса MyException
```

Каждая из этих строк сигнализирует о том, что возникла какая-то ошибка, которую нужно обработать.

Поиск исключения

Выбрасывание исключений - это лишь одна часть процесса обработки исключений. Вернемся к нашей аналогии с баскетболом: как только просвистел арбитр, что происходит дальше? Игроки останавливаются, и игра временно прекращается. Обычный ход игры нарушен.

В языке C++ мы используем **ключевое слово try** для определения блока стейтментов (так называемого ***«блока try»***). Блок try действует как наблюдатель в поисках исключений, которые были выброшены каким-либо из операторов в этом же блоке try, например:

```
1. try
2. {
3.     // Здесь мы пишем стейтменты, которые будут генерировать следующее
   исключение
4.     throw -1; // типичный стейтмент throw
5. }
```

Обратите внимание, блок try не определяет, КАК мы будем обрабатывать исключение. Он просто сообщает компилятору: «Эй, если какой-либо из стейтментов внутри этого блока try выбросит исключение — поймай его!».

Обработка исключений

Пока арбитр не объявит о штрафном броске, и пока этот штрафной бросок не будет выполнен, игра не возобновится. Другими словами, штрафной бросок должен быть обработан до возобновления игры.

Фактически, обработка исключений - это работа блока(ов) catch. **Ключевое слово catch** используется для определения блока кода (так называемого ***«блока catch»***), который обрабатывает исключения определенного типа данных.

Вот пример блока catch, который обрабатывает (ловит) исключения типа int:

```
1. catch (int a)
2. {
3.     // Обрабатываем исключение типа int
4.     std::cerr << "We caught an int exception with value" << a << '\n';
5. }
```

Блоки try и catch работают вместе. Блок try обнаруживает любые исключения, которые были выброшены в нем, и направляет их в соответствующий блок catch для обработки. Блок try должен иметь, по крайней мере, один блок catch, который находится сразу же за ним, но также может иметь и несколько блоков catch, размещенных последовательно (друг за другом).

Как только исключение было поймано блоком try и направлено в блок catch для обработки, оно считается обработанным (после выполнения кода блока catch), и выполнение программы возобновляется.

Параметры catch работают так же, как и параметры функции, причем параметры одного блока catch могут быть доступны и в другом блоке catch (который находится

за ним). Исключения фундаментальных типов данных могут быть пойманы по значению (параметром блока catch является значение), но исключения не фундаментальных типов данных должны быть пойманы по константной ссылке (параметром блока catch является константная ссылка), дабы избежать ненужного копирования.

Как и в случае с функциями, если параметр не используется в блоке catch, то имя переменной можно не указывать:

```
1. catch (double) // примечание: Мы не указываем имя переменной, так как в этом
   нет надобности (мы её нигде в блоке не используем)
2. {
3.     // Обрабатываем исключение типа double здесь
4.     std::cerr << "We caught an exception of type double" << '\n';
5. }
```

Это предотвратит вывод предупреждений компилятора о неиспользуемых переменных.

Использование throw, try и catch вместе

Вот полная программа, которая использует throw, try и несколько блоков catch:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     try
7.     {
8.         // Здесь мы пишем стейтменты, которые будут генерировать следующее
   исключение
9.         throw -1; // типичный стейтмент throw
10.    }
11.    catch (int a)
12.    {
13.        // Любые исключения типа int, сгенерированные в блоке try, приведенном
   выше, обрабатываются здесь
14.        std::cerr << "We caught an int exception with value: " << a << '\n';
15.    }
16.    catch (double) // мы не указываем имя переменной, так как в этом нет
   надобности (мы её нигде в блоке не используем)
17.    {
18.        // Любые исключения типа double, сгенерированные в блоке try,
   приведенном выше, обрабатываются здесь
19.        std::cerr << "We caught an exception of type double" << '\n';
20.    }
21.    catch (const std::string &str) // ловим исключения по константной ссылке
22.    {
23.        // Любые исключения типа std::string, сгенерированные внутри блока try,
    приведенном выше, обрабатываются здесь
24.        std::cerr << "We caught an exception of type std::string" << '\n';
25.    }
26.
```

```
27.     std::cout << "Continuing our way!\n";
28.
29.     return 0;
30. }
```

Результат выполнения программы:

```
We caught an int exception with value -1
Continuing our way!
```

Оператор throw используется для генерации исключения `-1` типа int. Затем блок try обнаруживает оператор throw и перемещает его в соответствующий блок catch, который обрабатывает исключения типа int. Блок catch типа int и выводит соответствующее сообщение об ошибке.

После обработки исключения, программа продолжает свое выполнение и выводит на экран `Continuing our way!`.

Резюмируем

Обработка исключений, на самом деле, довольно-таки проста, и всё, что вам нужно запомнить, размещено в следующих двух абзацах:

- При выбрасывании исключения (оператор throw), точка выполнения программы немедленно переходит к ближайшему блоку try. Если какой-либо из обработчиков catch, прикрепленных к блоку try, обрабатывает этот тип исключения, то точка выполнения переходит в этот обработчик и, после выполнения кода блока catch, исключение считается обработанным.
- Если подходящих обработчиков catch не существует, то выполнение программы переходит в следующий блок try. Если до конца программы не найдены соответствующие обработчики catch, то программа завершает свое выполнение с ошибкой исключения.

Обратите внимание, компилятор не выполняет неявные преобразования при сопоставлении исключений с блоками catch! Например, исключение типа char не будет обрабатываться блоком catch типа int, исключение типа int, в свою очередь, не будет обрабатываться блоком catch типа float.

Это действительно всё, что вам нужно запомнить.

Исключения обрабатываются немедленно

Вот маленькая программа, которая демонстрирует, что исключения обрабатываются немедленно:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     try
6.     {
7.         throw 7.4; // выбрасывается исключение типа double
8.         std::cout << "This never prints\n";
9.     }
10.    catch(double a) // обрабатывается исключение типа double
11.    {
12.        std::cerr << "We caught a double of value: " << a << '\n';
13.    }
14.
15.    return 0;
16.}
```

Рассмотрим выполнение этой программы пошагово:

- Оператор throw - это первый оператор, который выполняется. Это приводит к генерации исключения типа double.
- Точка выполнения *немедленно* переходит к ближайшему блоку try, который является единственным блоком try в этой программе (внутри которого и размещено это исключение).
- Затем проверяются обработчики catch на соответствие типу данных. Поскольку у нас исключение типа double, то компилятор ищет обработчик catch типа double. У нас такой есть, поэтому он и выполняется.

Следовательно, результат выполнения программы:

```
We caught a double of value: 7.4
```

Обратите внимание, строка `This never prints` никогда не выводится, так как генерация исключения заставило точку выполнения программы немедленно перейти к обработчику исключений типа double.

Еще один пример

Рассмотрим более популярный пример:

```
1. #include <iostream>
2. #include "math.h" // для функции sqrt()
3.
4. int main()
```

```
{
    std::cout << "Enter a number: ";
    double a;
    std::cin >> a;

    try // ищем исключения внутри этого блока и отправляем их в соответствующий обработчик catch
    {
        // Если пользователь ввел отрицательное число, то выбрасывается исключение
        if (a < 0.0)
            throw "Can not take sqrt of negative number"; // выбрасывается исключение типа const char*

        // Если пользователь ввел положительное число, то выполняется операция и выводится результат
        std::cout << "The sqrt of " << a << " is " << sqrt(a) << '\n';
    }
    catch (const char* exception) // обработчик исключений типа const char*
    {
        std::cerr << "Error: " << exception << '\n';
    }
}
```

Здесь мы просим пользователя ввести число. Если пользователь ввел положительное число, то стейтмент if не выполняется, исключение не генерируется, и пользователь получает квадратный корень из числа. Поскольку исключение не генерируется, то код внутри блока catch никогда не выполняется.

Результат:

```
Enter a number: 16
The sqrt of 16 is 4
```

Если же пользователь ввел отрицательное число, то генерируется исключение типа const char*. Поскольку мы уже находимся в блоке try, то компилятор ищет соответствующий обработчик catch типа const char*, и точка выполнения немедленно перемещается в этот блок.

Результат:

```
Enter a number: -3
Error: Can not take sqrt of negative number
```

Что обычно делают блоки catch?

Если исключение направлено в блок catch, то оно считается «обработанным», даже если блок catch пуст. Однако, как правило, вы захотите, чтобы ваши блоки catch делали что-то полезное.

Есть три распространенные вещи, которые выполняют блоки catch, когда они поймали исключение:

- Во-первых, блок catch может вывести сообщение об ошибке (либо в консоль, либо в лог-файл).
- Во-вторых, блок catch может возвратить значение или код ошибки обратно в caller.
- В-третьих, блок catch может сгенерировать другое исключения. Поскольку блок catch не находится внутри блока try, то новое сгенерированное исключение будет обрабатываться следующим блоком try.

Сейчас вы уже должны были получить основное представление об исключениях. На следующем уроке мы рассмотрим еще несколько примеров для проверки того, насколько гибкими исключения могут быть.

Урок №191. Исключения, Функции и Раскручивание стека

На предыдущем уроке мы говорили о том, как, используя ключевые слова throw, try и catch, обрабатывать исключения. На этом уроке мы рассмотрим, как взаимодействуют функции во время обработки исключений в языке C++.

Генерация исключений за пределами блока try

На предыдущем уроке операторы throw помещались непосредственно в блок try. Если бы это было обязательным условием, то согласитесь, что обработка исключений не была бы гибкой вообще.

На самом деле стейтменты throw вовсе не обязаны находиться непосредственно в блоке try, благодаря выполнению такой операции, как "раскручивание стека". Это предоставляет нам необходимую гибкость в разделении общего потока выполнения кода программы и обработки исключений. Продемонстрируем это, переписав программу из предыдущего урока, вынеся генерацию исключения и вычисление квадратного корня в отдельную функцию:

```
1. #include <cmath> // для sqrt()
2. #include <iostream>
3.
4. // Отдельная функция вычисления квадратного корня
5. double mySqrt(double a)
6. {
7.     // Если пользователь ввел отрицательное число, то выбрасываем исключение
8.     if (a < 0.0)
9.         throw "Can not take sqrt of negative number"; // выбрасывается
   исключение типа const char*
10.
11.    return sqrt(a);
12.}
13.
14.int main()
15.{
16.    std::cout << "Enter a number: ";
17.    double a;
18.    std::cin >> a;
19.
20.    try // ищем исключения, которые выбрасываются в блоке try, и отправляем их
   для обработки в блок(и) catch
21.    {
22.        double d = mySqrt(a);
23.        std::cout << "The sqrt of " << a << " is " << d << '\n';
24.    }
25.    catch (const char* exception) // обработка исключений типа const char*
26.    {
27.        std::cerr << "Error: " << exception << std::endl;
28.    }
29.
30.    return 0;
31.}
```

Здесь мы переместили генерацию исключения и операцию вычисления квадратного корня в отдельную функцию mySqrt(). Затем мы вызываем эту функцию в блоке try. Убедимся, что всё работает, как нужно:

```
Enter a number: -3
Error: Can not take sqrt of negative number
```

Ура! Однако, давайте вернемся к моменту генерации исключения и рассмотрим ход выполнения программы. Во-первых, при генерации исключения компилятор смотрит, можно ли сразу же обработать это исключение (для этого нужно, чтобы исключение выбрасывалось внутри блока try). Поскольку точка выполнения не находится внутри блока try, то и обработать исключение немедленно не получится. Таким образом, выполнение функции mySqrt() приостанавливается, и программа смотрит, может ли caller (который и вызывает mySqrt()) обработать это исключение.

Если нет, то компилятор завершает выполнение caller-а и переходит на уровень выше - к caller-у, который вызывает текущего caller-а, чтобы проверить, сможет ли тот обработать исключение. И так последовательно до тех пор, пока не будет найден соответствующий обработчик исключения, или пока функция main() не завершит свое выполнение без обработки исключения. Этот процесс называется **раскручиванием стека**.

Теперь рассмотрим детально, как это относится к нашей программе. Сначала компилятор проверяет, генерируется ли исключение внутри блока try. В нашем случае — нет, поэтому стек начинает раскручиваться. При этом функция mySqrt() завершает свою работу, и точка выполнения перемещается обратно в функцию main(). Теперь компилятор проверяет снова, находимся ли мы внутри блока try. Поскольку вызов функции mySqrt() был выполнен из блока try, то компилятор начинает искать соответствующий обработчик catch. Он находит обработчик типа const char*, и исключение обрабатывается блоком catch внутри main().

Подводя итог, функция mySqrt() генерирует исключение, но блоки try/catch, которые находятся в main(), ловят и обрабатывают это исключение. Другими словами, **блок try ловит исключения не только внутри себя, но и внутри функций, которые вызываются в этом блоке try**.

Самое интересное здесь в том, что mySqrt() как бы говорит: «Эй, компилятор, здесь проблема!». Но обрабатывать эту проблему mySqrt() отказывается. Это, по сути, делегирование ответственности за обработку исключения на caller (аналогично тому, как при использовании кодов завершения ответственность за обработку ошибок перекладывается обратно на caller).

Сейчас некоторые из вас, вероятно, спросят: «Зачем передавать ошибки обратно в caller? Почему бы просто не заставить функцию mySqrt() обрабатывать собственные исключения?». Проблема в том, что разные программы обрабатывают ошибки/исключения по-разному. Консольная программа выводит сообщение об ошибке. Приложение Windows выводит диалоговое окно с ошибкой. В одной программе это может быть фатальной ошибкой, а в другой — нет. Передавая ошибку обратно в стек, каждое приложение может обрабатывать исключение mySqrt() таким образом, который является наиболее подходящим по контексту! В конечном счете, это позволяет отделить функционал mySqrt() от кода обработки исключений, который можно разместить в других (менее важных) частях кода.

Еще один пример раскручивания стека

Здесь у нас стек уже побольше. Хотя всё кажется слишком сложным, но на самом деле это не так:

- main() вызывает one();
- one() вызывает two();
- two() вызывает three();
- three() вызывает last();
- last() выбрасывает исключение.

Смотрим:

```
#include <iostream>

void last() // вызывается функцией three()
{
    std::cout << "Start last\n";
    std::cout << "last throwing int exception\n";
    throw -1;
    std::cout << "End last\n";
}

void three() // вызывается функцией two()
{
    std::cout << "Start three\n";
    last();
    std::cout << "End three\n";
}

void two() // вызывается функцией one()
{
    std::cout << "Start two\n";
    try
    {
        three();
    }
    catch(double)
    {
```

```
27.         std::cerr << "two caught double exception\n";
28.     }
29.     std::cout << "End two\n";
30. }
31. 
32. void one() // вызывается функцией main()
33. {
34.     std::cout << "Start one\n";
35.     try
36.     {
37.         two();
38.     }
39.     catch (int)
40.     {
41.         std::cerr << "one caught int exception\n";
42.     }
43.     catch (double)
44.     {
45.         std::cerr << "one caught double exception\n";
46.     }
47.     std::cout << "End one\n";
48. }
49. 
50. int main()
51. {
52.     std::cout << "Start main\n";
53.     try
54.     {
55.         one();
56.     }
57.     catch (int)
58.     {
59.         std::cerr << "main caught int exception\n";
60.     }
61.     std::cout << "End main\n";
62. 
63.     return 0;
64. }
```

Взгляните на эту программу еще раз. Можете ли вы понять, что выведется на экран? Результат:

```
Start main
Start one
Start two
Start three
Start last
last throwing int exception
one caught int exception
End one
End main
```

Рассмотрим ход выполнения программы детально. Думаю не нужно объяснять вывод строчек `Start`. Функция last() выводит `last throwing int exception`, а затем выбрасывает исключение типа int. Вот где начинается самое интересное.

Поскольку функция last() не обрабатывает исключения самостоятельно, то стек начинает раскручиваться. Функция last() немедленно завершает свое выполнение, и точка выполнения возвращается обратно в caller (в функцию three()).

Функция three() не обрабатывает какие-либо исключения, поэтому стек раскручивается дальше, выполнение функции three() прекращается, и точка выполнения возвращается в two().

Функция two() имеет блок try, в котором находится вызов three(), поэтому компилятор пытается найти обработчик исключений типа int, но, так как его не находит, точка выполнения возвращается обратно в one(). Обратите внимание, компилятор не выполняет неявное преобразование, чтобы сопоставить исключение типа int с обработчиком типа double.

Функция one() также имеет блок try с вызовом two() внутри, поэтому компилятор смотрит, есть ли подходящий обработчик catch. Есть — функция one() обрабатывает исключение и выводит `one caught int exception`.

Поскольку исключение было обработано, то точка выполнения перемещается в конец блока catch внутри one(). Это означает, что one() выводит `End one`, а затем завершает свое выполнение, как обычно.

Точка выполнения возвращается обратно в main(). Хотя main() имеет обработчик исключений типа int, но наше исключение уже было обработано функцией one(), поэтому блок catch внутри main() не выполняется. Функция main() выводит `End main`, а затем завершает свое выполнение.

Из этой программы можно сделать несколько интересных выводов:

- Во-первых, непосредственный caller, вызывающий функцию, в которой выбрасывается исключение, не обязан обрабатывать это исключение, если он этого не хочет. В примере, приведенном выше, функция three() не обрабатывает исключение, генерируемое функцией last(). Она делегирует эту ответственность на другой caller из стека.
- Во-вторых, если блок try не имеет обработчика catch соответствующего типа, то раскручивание стека происходит так же, как если бы этого блока try не было вообще. В примере, приведенном выше, функция two() не обрабатывает исключение, потому что у нее нет соответствующего обработчика catch.
- В-третьих, когда исключение обработано, выполнение кода продолжается как обычно, начиная с конца блока catch (в котором это исключение было

обработано). В примере, приведенном выше, функция one() обработала исключение, а затем продолжила свое выполнение выводом строки `End one`. К тому времени, когда точка выполнения возвращается обратно в функцию main(), исключение уже было сгенерировано и обработано. Функция main() выполняется так, как если бы этого исключения не было вообще!

Раскручивание стека является очень полезным механизмом, так как позволяет функциям не обрабатывать исключения, если они этого не хотят. Операция раскручивания стека выполняется до тех пор, пока не будет обнаружен соответствующий блок catch! Таким образом, мы можем сами решать, где следует обрабатывать исключения.

Урок №192. Непойманные исключения и обработчики catch-all

На предыдущем уроке мы научились делегировать обработку исключений caller-у (или другой функции, которая "находится выше" в стеке вызовов). В следующем примере функция mySqrt() выбрасывает исключение и предполагает, что его кто-то обработает. Но что произойдет, если этого никто не сделает?

Вот наша программа вычисления квадратного корня числа без блока try в функции main():

```
1. #include <iostream>
2. #include <cmath> // для sqrt()
3.
4. // Отдельная функция вычисления квадратного корня числа
5. double mySqrt(double a)
6. {
7.     // Если пользователь ввел отрицательное число,
8.     if (a < 0.0)
9.         throw "Can not take sqrt of negative number"; // то выбрасывается
   исключение типа const char*
10.
11.    return sqrt(a);
12.}
13.
14.int main()
15.{
16.    std::cout << "Enter a number: ";
17.    double a;
18.    std::cin >> a;
19.
20.    // Здесь нет никакого обработчика исключений!
21.    std::cout << "The sqrt of " << a << " is " << mySqrt(a) << '\n';
22.
23.    return 0;
24.}
```

Теперь предположим, что пользователь ввел `-5`, и `mySqrt(-5)` сгенерировало исключение. Функция mySqrt() не обрабатывает свои исключения самостоятельно, поэтому стек начинает раскручиваться, и точка выполнения возвращается обратно в функцию main(). Но, поскольку в main() также нет обработчика исключений, выполнение main() и всей программы прекращается.

Когда main() завершает свое выполнение с необработанным исключением, то операционная система обычно уведомляет нас о том, что произошла ошибка необработанного исключения. Как она это сделает — зависит от каждой операционной системы отдельно:

- либо выведет сообщение об ошибке;

- либо откроет диалоговое окно с ошибкой;
- либо просто сбой.

Это то, что мы не должны допускать, как программисты!

Обработчики всех типов исключений

А теперь загадка: "Функции могут генерировать исключения любого типа данных, и, если исключение не поймано, это приведет к раскручиванию стека и потенциальному завершению выполнения всей программы. Поскольку мы можем вызывать функции, не зная их реализации (и, следовательно, какие исключения они могут генерировать), то как мы можем это предотвратить?".

К счастью, язык C++ предоставляет нам механизм обнаружения/обработки всех типов исключений - обработчик catch-all. **Обработчик catch-all** работает так же, как и обычный блок catch, за исключением того, что вместо обработки исключений определенного типа данных, он использует эллипсис (`...`) в качестве типа данных.

А как мы уже знаем, эллипсисы могут использоваться для передачи аргументов любого типа данных в функцию. В этом контексте они представляют собой исключения любого типа данных. Вот простой пример:

```
#include <iostream>

int main()
{
    try
    {
        throw 7; // выбрасывается исключение типа int
    }
    catch (double a)
    {
        std::cout << "We caught an exception of type double: " << a << '\n';
    }
    catch (...) // обработчик catch-all
    {
        std::cout << "We caught an exception of an undetermined type!\n";
    }
}
```

Поскольку для типа int не существует специального обработчика catch, то обработчик catch-all ловит это исключение. Следовательно, результат:

```
We caught an exception of an undetermined type!
```

Обработчик catch-all должен находиться последним в цепочке блоков catch. Это делается для того, чтобы исключения сначала могли быть пойманы обработчиками catch, адаптированными к конкретным типам данных (если они вообще

существуют). В Visual Studio это контролируется, насчет других компиляторов - не уверен, есть ли такое ограничение.

Часто блок обработчика catch-all оставляют пустым:

```
1. catch(...) {} // игнорируются любые непредвиденные исключения
```

Этот обработчик ловит любые непредвиденные исключения и предотвращает раскручивание стека (и, следовательно, потенциальное завершение выполнения всей программы), но здесь он не выполняет никакой обработки исключений.

Использование обработчика catch-all в функции main()

Рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int main()
4. {
5.
6.     try
7.     {
8.         runGame();
9.     }
10.    catch(...)
11.    {
12.        std::cerr << "Abnormal termination\n";
13.    }
14.
15.    saveState(); // сохраняем текущее состояние игрока
16.    return 1;
17.}
```

В этом случае, если функция runGame() или любая другая из функций, которые вызываются в runGame(), выбросит исключение, которое не будет поймано функциями в стеке выше, то, в конечном итоге, оно попадет в обработчик catch-all. Это предотвратит завершение выполнения функции main() и даст нам возможность вывести сообщение с указанием ошибки на наше усмотрение, а затем сохранить состояние пользователя до выхода из программы. Это может быть полезно для обнаружения и устранения непредвиденных проблем.

Спецификации исключений

Эту тему можете рассматривать как дополнительное чтиво, так как спецификации исключений редко используются на практике, плохо поддерживаются компиляторами, а Бьёрн Страуструп (создатель языка C++) считает их неудачным экспериментом.

Спецификации исключений - это механизм объявления функций с указанием того, будет ли функция генерировать исключения (и какие именно) или нет. Это может быть полезно при определении необходимости помещения вызова функции в блок try.

Существуют три типа спецификации исключений, каждый из которых использует так называемый синтаксис **throw (...)**.

Во-первых, мы можем использовать пустой оператор throw для обозначения того, что функция не генерирует никакие исключения, которые выходят за её пределы:

```
1. int doSomething() throw(); // не выбрасываются исключения
```

Обратите внимание, функция doSomething() все еще может генерировать исключения, только обрабатывать она должна их самостоятельно. Любая функция, объявленная с использованием throw() (как в вышеприведенном примере), должна немедленно прекратить выполнение программы, если она попытается сгенерировать исключение, которое приведет к раскручиванию стека. Другими словами, мы сообщаем, что все исключения функции doSomething(), функция doSomething() будет обрабатывать самостоятельно.

Во-вторых, мы можем использовать оператор throw с указанием типа исключения, которое может генерировать эта функция:

```
1. int doSomething() throw(double); // могут генерироваться исключения типа double
```

Наконец, мы можем использовать эллипсис с оператором throw для обозначения того, что функция может генерировать разные типы исключений:

```
1. int doSomething() throw(...); // могут генерироваться любые исключения
```

Из-за плохой (неполной) реализации и совместимости с компиляторами, и учитывая тот факт, что спецификации исключений больше напоминают заявления о намерениях, чем гарантии чего-либо, и то, что они плохо совместимы с шаблонами функций, и то, что большинство программистов C++ не знают о их существовании, приводит к тому, что использовать спецификации исключений не рекомендуется.

Урок №193. Классы-Исключения и Наследование

К этому моменту мы рассматривали использование исключений только в обычных функциях, которые не являются методами класса. Тем не менее, исключения одинаково полезны и в методах, и даже в перегрузке операторов.

Исключения в перегрузке операторов

Рассмотрим следующую перегрузку оператора индексации [] для простого целочисленного класса-массива:

```
1. int& ArrayInt::operator[](const int index)
2. {
3.     return m_data[index];
4. }
```

Хотя эта функция отлично работает, но это только до тех пор, пока значением переменной `index` является корректный индекс массива. Здесь явно не хватает механизма обработки ошибок. Добавляем стейтмент assert для проверки `index`:

```
1. int& ArrayInt::operator[](const int index)
2. {
3.     assert (index >= 0 && index < getLength());
4.     return m_data[index];
5. }
```

Теперь, если пользователь передаст недопустимый `index`, то программа выдаст ошибку. Хотя это сообщит пользователю, что что-то пошло не так, лучшим вариантом было бы "по-тихому" сообщить caller-у, что что-то пошло не так и пусть он с этим разберется соответствующим образом (как именно — мы пропишем позднее).

К сожалению, поскольку перегрузка операторов имеет особые требования к количеству и типу параметров, которые они могут принимать и возвращать, нет никакой гибкости для передачи кодов ошибок или логических значений обратно в caller. Однако, мы можем использовать исключения, которые не изменяют сигнатуру функции, например:

```
1. int& ArrayInt::operator[](const int index)
2. {
3.     if (index < 0 || index >= getLength())
4.         throw index;
5. 
6.     return m_data[index];
7. }
```

Теперь, если пользователь передаст недопустимый `index`, operator[] сгенерирует исключение типа int.

Когда конструкторы терпят неудачу

Конструкторы - это еще одна часть классов, в которой исключения могут быть очень полезными. Если конструктор не сработал, то сгенерируйте исключение, которое сообщит, что объект не удалось создать. Создание объекта прерывается, а деструктор никогда не выполняется (обратите внимание, это означает, что ваш конструктор должен самостоятельно выполнять очистку памяти перед генерацией исключения).

Классы-Исключения

Одной из основных проблем использования фундаментальных типов данных (например, типа int) в качестве типов исключений является то, что они, по своей сути, являются неопределенными. Еще более серьезной проблемой является неоднозначность того, что означает исключение, когда в блоке try имеется несколько стейтментов или вызовов функций:

```
1. // Используем перегрузку operator[] для ArrayInt
2. 
3. try
4. {
5.     int *value = new int(array[index1] + array[index2]);
6. }
7. catch (int value)
8. {
9.     // Какие исключения мы здесь ловим?
10. }
```

В этом примере, если мы поймаем исключение типа int, что оно нам сообщит? Был ли передаваемый `index` недопустим? Может оператор `+` вызвал целочисленное переполнение или может оператор new не сработал из-за нехватки памяти? Хотя мы можем генерировать исключения типа const char*, которые будут указывать ПРИЧИНУ сбоя, это все еще не даст нам возможности обрабатывать исключения из разных источников по-разному.

Одним из способов решения этой проблемы является использование классов-исключений. **Класс-Исключение** - это обычный класс, который выбрасывается в качестве исключения. Создадим простой класс-исключение, который будет использоваться с нашим ArrayInt:

```
1. #include <string>
2. 
3. class ArrayException
```

```
{
private:
    std::string m_error;

public:
    ArrayException(std::string error)
        : m_error(error)
    {
    }

    const char* getError() { return m_error.c_str(); }
};
```

Вот полная программа:

```
#include <iostream>
#include <string>

class ArrayException
{
private:
    std::string m_error;

public:
    ArrayException(std::string error)
        : m_error(error)
    {
    }

     const char* getError() { return m_error.c_str(); }
};

class ArrayInt
{
private:

    int m_data[4]; // ради сохранения простоты примера укажем значение 4 в качестве длины массива
public:
    ArrayInt() {}

    int getLength() { return 4; }

    int& operator[](const int index)
    {
        if (index < 0 || index >= getLength())
            throw ArrayException("Invalid index");

        return m_data[index];
    }

};

int main()
{
    ArrayInt array;

    try
    {
        int value = array[7];
    }
```

```
46.     catch (ArrayException &exception)
47.     {
48.         std::cerr << "An array exception occurred (" << exception.getError()
    << ")\n";
49.     }
50. }
```

Используя такой класс, мы можем генерировать исключение, возвращающее описание возникшей проблемы, это даст нам точно понять, что именно пошло не так. И, поскольку исключение ArrayException имеет уникальный тип, мы можем обрабатывать его соответствующим образом (не так как другие исключения).

Обратите внимание, в обработчиках исключений объекты класса-исключения принимать нужно по ссылке, а не по значению. Это предотвратит создание копии исключения компилятором, что является затратной операцией (особенно в случае, когда исключение является объектом класса), и предотвратит обрезку объектов при работе с дочерними классами-исключениями. Передачу по адресу лучше не использовать, если у вас нет на это веских причин.

Исключения и Наследование

Так как мы можем выбрасывать объекты классов в качестве исключений, а классы могут быть получены из других классов, то нам нужно учитывать, что произойдет, если мы будем использовать унаследованные классы в качестве исключений. Оказывается, обработчики могут обрабатывать исключения не только одного определенного класса, но и исключения дочерних ему классов!

Рассмотрим следующий пример:

```
1. #include <iostream>
2. #include <cassert>
3.
4. class Parent
5. {
6. public:
7.     Parent() {}
8. };
9.
10. class Child: public Parent
11. {
12. public:
13.     Child() {}
14. };
15.
16. int main()
17. {
18.     try
19.     {
20.         throw Child();
21.     }
22.     catch (Parent &parent)
```

```
    {
        std::cerr << "caught Parent";
    }
    catch (Child &child)
    {
        std::cerr << "caught Child";
    }

    return 0;
}
```

Здесь выбрасывается исключение типа Child. Однако, результат выполнения данной программы:

```
caught Parent
```

Что случилось?

Во-первых, как мы уже говорили, дочерние классы могут быть пойманы обработчиком родительского класса. Поскольку Child является дочерним классу Parent, то из этого следует, что Child «является» Parent («является» - тип отношения). Во-вторых, когда C++ пытается найти обработчик для выброшенного исключения, он делает это последовательно. Первое, что он проверяет - подходит ли обработчик исключений класса Parent для исключений класса Child. Поскольку Child «является» Parent, то блок catch для объектов класса Parent подходит и, соответственно, выполняется! В этом случае блок catch для объектов класса Child никогда не выполнится.

Чтобы этот пример работал по-другому, нам нужно изменить порядок последовательности блоков catch:

```
#include <iostream>
#include <cassert>

class Parent
{
public:
    Parent() {}
};

class Child: public Parent
{
public:
    Child() {}
};

int main()
{
    try
    {
        throw Child();
    }
```

```
22.     catch (Child &child)
23.     {
24.         std::cerr << "caught Child";
25.     }
26.     catch (Parent &parent)
27.     {
28.         std::cerr << "caught Parent";
29.     }
30.
31.     return 0;
32. }
```

Результат:

```
caught Child
```

Таким образом, обработчик Child будет ловить и обрабатывать исключения класса Child. Исключения класса Parent не соответствуют обработчику Child (Child «является» Parent, но Parent «не является» Child) и, соответственно, будут обрабатываться только обработчиком Parent.

Правило: Обработчики исключений дочерних классов должны находиться перед обработчиками исключений родительского класса.

Интерфейсный класс std::exception

Многие классы и операторы из Стандартной библиотеки C++ выбрасывают классы-исключения при сбое. Например, оператор new и std::string могут выбрасывать std::bad_alloc при нехватке памяти. Неудачное динамическое приведение типов с помощью оператора dynamic_cast выбрасывает исключение std::bad_cast и т.д. Начиная с C++14, существует больше 20 классов-исключений, которые могут быть выброшены, а в C++17 их еще больше.

Хорошей новостью является то, что все эти классы-исключения являются дочерними классу std::exception. **std::exception** — это небольшой интерфейсный класс, который используется в качестве родительского класса для любого исключения, которое выбрасывается в Стандартной библиотеке C++.

В большинстве случаев, если исключение выбрасывается Стандартной библиотекой C++, нам все равно, было ли это неудачное выделение, конвертирование или что-либо другое. Нам достаточно знать, что произошло что-то катастрофическое, из-за чего в нашей программе произошел сбой.

Благодаря std::exception мы можем настроить обработчик исключений типа std::exception, который будет ловить и обрабатывать как std::exception, так и все (20+) дочерние ему классы-исключения!

```
#include <iostream>
#include <exception> // для std::exception
#include <string> // для этого примера

int main()
{
    try
    {
        // Здесь должен находиться код, использующий Стандартную библиотеку C++.
        // Сейчас мы намеренно спровоцируем генерацию одного из исключений
        std::string s;
        s.resize(-1); // генерируется исключение std::bad_alloc
    }
    // Этот обработчик ловит std::exception и все дочерние ему классы-исключения
    catch (std::exception &exception)
    {
        std::cerr << "Standard exception: " << exception.what() << '\n';
    }

    return 0;
}
```

Результат выполнения программы:

```
Standard exception: string too long
```

В этом примере всё довольно просто. В std::exception есть **виртуальный метод what()**, который возвращает строку C-style с описанием исключения. Большинство дочерних классов переопределяют функцию what(), изменяя это сообщение. Обратите внимание, эта строка C-style предназначена для использования только в качестве описания.

Иногда нам нужно будет обрабатывать определенный тип исключений несколько иначе, нежели остальные типы исключений. В таком случае мы можем добавить обработчик исключений для этого конкретного типа, а все остальные исключения «перенаправить» в родительский обработчик. Например:

```
try
{
     // Здесь должен находиться код, использующий Стандартную библиотеку C++
}
// Этот обработчик ловит std::bad_alloc и все дочерние ему классы-исключения
catch (std::bad_alloc &exception)
{
    std::cerr << "You ran out of memory!" << '\n';
}
// Этот обработчик ловит std::exception и все дочерние ему классы-исключения
```

```
11. catch (std::exception &exception)
12. {
13.     std::cerr << "Standard exception: " << exception.what() << '\n';
14. }
```

В этом примере исключения типа std::bad_alloc перехватываются и обрабатываются первым обработчиком. Исключения типа std::exception и всех других дочерних ему классов-исключений обрабатываются вторым обработчиком.

Такие иерархии наследования позволяют использовать определенные обработчики для перехвата определенного типа исключений или для перехвата одним (родительским) обработчиком всей иерархии исключений.

Использование стандартных исключений напрямую

Ничто не генерирует std::exception напрямую, и вы также должны придерживаться этого правила. Однако, вы можете генерировать исключения других классов из Стандартной библиотеки C++, если они адекватно отражают ваши потребности. Найти список всех стандартных классов-исключений из Стандартной библиотеки C++ вы можете **здесь**.

std::runtime_error (находится в заголовочном файле stdexcept) является популярным выбором, так как имеет общее имя, а конструктор принимает настраиваемое сообщение:

```
1. #include <iostream>
2. #include <stdexcept>
3.
4. int main()
5. {
6.     try
7.     {
8.         throw std::runtime_error("Bad things happened");
9.     }
10.    // Этот обработчик ловит std::exception и все дочерние ему классы-
   исключения
11.    catch (std::exception &exception)
12.    {
13.        std::cerr << "Standard exception: " << exception.what() << '\n';
14.    }
15.
16.    return 0;
17. }
```

Результат:

```
Standard exception: Bad things happened
```

Создание собственных классов-исключений, дочерних классу std::exception

Конечно, вы можете создать свои собственные классы-исключения, дочерние классу std::exception, и переопределить виртуальный константный метод what(). Вот программа, приведенная выше, но уже с классом-исключением ArrayException, дочерним std::exception:

```
#include <iostream>
#include <string>
#include <exception> // для std::exception

class ArrayException: public std::exception
{
private:
    std::string m_error;

public:
    ArrayException(std::string error)
        : m_error(error)
    {
    }

    // Возвращаем std::string в качестве константной строки C-style
//  const char* what() const { return m_error.c_str(); } // до C++11
    const char* what() const noexcept { return m_error.c_str(); } // C++11 и выше
};

class ArrayInt
{
private:

    int m_data[4]; // чтобы не усложнять, укажем значение 4 в качестве длины массива
public:
    ArrayInt() {}

    int getLength() { return 4; }

    int& operator[](const int index)
    {
        if (index < 0 || index >= getLength())
            throw ArrayException("Invalid index");

        return m_data[index];
    }

};

int main()
{
    ArrayInt array;

    try
    {
        int value = array[7];
    }
```

```
49.     catch (ArrayException &exception) // сначала ловим исключения дочернего класса-исключения
50.     {
51.         std::cerr << "An array exception occurred (" << exception.what() << ")\n";
52.     }
53.     catch (std::exception &exception)
54.     {
55.         std::cerr << "Some other std::exception occurred (" << exception.what() << ")\n";
56.     }
57. }
```

В C++11 к виртуальной функции what() добавили **спецификатор noexcept** (который означает, что функция обещает не выбрасывать исключения самостоятельно). Следовательно, в C++11 и в более новых версиях наше переопределение метода what() также должно иметь спецификатор noexcept.

Вам решать, хотите ли вы создавать свои собственные классы-исключения, использовать классы-исключения из Стандартной библиотеки C++ или писать классы-исключения, дочерние std::exception. Всё зависит от ваших целей.

Урок №194. Повторная генерация исключений

Иногда вы можете столкнуться с ситуацией, когда нужно поймать исключение, но обрабатывать его в данный момент времени не нужно (или нет возможности). Например, вы можете записать ошибку в лог-файл, а затем передать её обратно в caller для выполнения фактической обработки. Это легко выполнимая задача при использовании кодов возврата, например:

```
Database* createDatabase(std::string filename)
{
    try
    {
        Database *d = new Database(filename);
        d->open(); // предположим, что выполнение этой строки приведет к генерации исключения типа int
        return d;
    }
    catch (int exception)
    {
        // Неудачное создание объекта Database.
        // Записываем ошибку в лог-файл
        g_log.logError("Creation of Database failed");
    }

    return nullptr;
}
```

В примере, приведенном выше, функции поручено создать объект класса Database, открыть базу данных, а затем возвратить объект класса Database обратно. В таком случае, если что-то пойдет не так (например, передастся неправильный `filename`), то обработчик исключений запишет ошибку в лог-файл, а затем возвратит нулевой указатель.

Теперь рассмотрим следующую функцию:

```
int getIntValueFromDatabase(Database *d, std::string table, std::string key)
{
    assert(d);

    try
    {
        return d->getIntValue(table, key); // генерируется исключение типа int
    }
    catch (int exception)
    {
        // Записываем ошибку в лог-файл
        g_log.logError("doSomethingImportant failed");

        // Однако мы не обработали эту ошибку.
        // Что нам здесь делать?
    }
```

```
17. }
```

В случае успеха выполнения этой функции возвращается целочисленное значение.

Но, что если что-то пойдет не так с getIntValue()? В таком случае, функция getIntValue() сгенерирует целочисленное исключение, которое будет перехвачено блоком catch в getIntValueFromDatabase(), который запишет ошибку в лог-файл. Но как мы затем сообщим caller-у функции getIntValueFromDatabase(), что что-то пошло не так? В отличие от функции из первого примера, здесь нет хорошего кода возврата, который мы могли бы использовать (поскольку функция getIntValueFromDatabase() возвращает целочисленное значение, то любое целочисленное значение в качестве кода возврата является допустимым).

Генерация нового исключения

Одним из очевидных решений является генерация нового исключения:

```
1. int getIntValueFromDatabase(Database *d, std::string table, std::string key)
2. {
3.     assert(d);
4.
5.     try
6.     {
7.         return d-
   >getIntValue(table, key); // генерируется исключение типа int
8.     }
9.     catch (int exception)
10.    {
11.        // Записываем ошибку в лог-файл
12.        g_log.logError("doSomethingImportant failed");
13.
14.        throw 'q'; // генерируется исключение 'q' типа char, которое будет
   обрабатывать caller функции getIntValueFromDatabase()
15.    }
16. }
```

В примере, приведенном выше, программа ловит исключение типа int из getIntValue(), записывает ошибку в лог-файл, а затем выбрасывает новое исключение со значением `q` типа char. Хотя генерация исключения в блоке catch может показаться странной затеей, это не запрещено. Помните, что только исключения, сгенерированные в блоке try, могут быть перехвачены блоком catch. Это означает, что исключение, сгенерированное в блоке catch, не будет перехвачено блоком catch, в котором оно находится. Вместо этого стек начнет раскручиваться, и исключение будет передано caller-у, который находится на уровне выше в стеке вызовов.

Исключение, сгенерированное в блоке catch, может быть исключением любого типа - оно не обязательно должно быть того же типа, что и исключение, которое обрабатывает блок catch.

Неправильная повторная генерация исключений

Альтернативным решением является повторная генерация одного и того же исключения:

```
1. int getIntValueFromDatabase(Database *d, std::string table, std::string key)
2. {
3.     assert(d);
4.
5.     try
6.     {
7.         return d-
   >getIntValue(table, key); // генерируется исключение типа int
8.     }
9.     catch (int exception)
10.    {
11.        // Записываем ошибку в лог-файл
12.        g_log.logError("doSomethingImportant failed");
13.
14.        throw exception;
15.    }
16. }
```

Хотя это работает, но здесь есть пара нюансов. Во-первых, в блоке catch не генерируется точно такое же исключение, которое обрабатывает блок catch, а генерируется копия переменной `exception`. Этот вариант плох тем, что снижает производительность (незначительно, но можно было бы и без этого обойтись).

Рассмотрим, что произойдет в следующем случае:

```
1. int getIntValueFromDatabase(Database *d, std::string table, std::string key)
2. {
3.     assert(d);
4.
5.     try
6.     {
7.         return d->getIntValue(table, key); // генерируется класс-
   исключение Child
8.     }
9.     catch (Parent &exception)
10.    {
11.        // Записываем ошибку в лог-файл
12.        g_log.logError("doSomethingImportant failed");
13.
14.        throw exception; // опасно: Эта строка выбрасывает в качестве
   исключения объект класса Parent, а не объект класса Child
15.    }
16. }
```

Здесь функция getIntValue() выбрасывает объект класса Child в качестве исключения, но блок catch принимает по ссылке объект класса Parent. Это нормально, так как мы уже знаем из предыдущих уроков, что ссылка родительского класса может использоваться для указания на дочерний объект. Однако в таком случае копия `exception` является класса Parent, а не класса Child! Другими словами, произойдет обрезка объекта класса Child()!

Мы можем это увидеть в следующей программе:

```
#include <iostream>

class Parent
{
public:
    Parent() {}
    virtual void print() { std::cout << "Parent"; }
};

class Child: public Parent
{
public:
    Child() {}
    virtual void print() { std::cout << "Child"; }
};

int main()
{
    try
    {
        try
        {
            throw Child();
        }
        catch (Parent& p)
        {
            std::cout << "Caught Parent p, which is actually a ";
            p.print();
            std::cout << "\n";
            throw p; // обрезка объекта класса Child происходит здесь
        }
    }
    catch (Parent& p)
    {
        std::cout << "Caught Parent p, which is actually a ";
        p.print();
        std::cout << "\n";
    }

    return 0;
}
```

Результат выполнения программы:

```
Caught Parent p, which is actually a Child
Caught Parent p, which is actually a Parent
```

Тот факт, что вторая строка указывает на то, что Parent на самом деле является Parent, а не Child, доказывает, что произошла обрезка объекта Child.

Правильная повторная генерация исключений

К счастью, язык C++ предоставляет способ повторной генерации одного и того же исключения. Для этого нужно просто использовать ключевое слово throw внутри блока catch без указания какого-либо идентификатора. Например:

```
#include <iostream>

class Parent
{
public:
    Parent() {}
    virtual void print() { std::cout << "Parent"; }
};

class Child: public Parent
{
public:
    Child() {}
    virtual void print() { std::cout << "Child"; }
};

int main()
{
    try
    {
        try
        {
            throw Child();
        }
        catch (Parent& p)
        {
            std::cout << "Caught Parent p, which is actually a ";
            p.print();
            std::cout << "\n";
            throw; // примечание: Мы здесь повторно выбрасываем исключение
        }
    }
    catch (Parent& p)
    {
        std::cout << "Caught Parent p, which is actually a ";
        p.print();
        std::cout << "\n";
    }

    return 0;
}
```

Результат выполнения программы:

```
Caught Parent p, which is actually a Child
Caught Parent p, which is actually a Child
```

Ключевое слово throw в блоке catch, которое, как кажется на первый взгляд, не генерирует что-либо конкретное, на самом деле генерирует точно такое же исключение, которое было только что обработано блоком catch. Никакого копирования исключения и, следовательно, обрезки объекта не выполняется.

Таким образом, этот метод повторной генерации исключения является более предпочтительным для использования.

Правило: При повторной генерации исключения используйте ключевое слово throw без указания какого-либо идентификатора.

Урок №195. Функциональный try-блок

Функционала блоков try и catch вполне достаточно в большинстве случаев, но есть одна конкретная ситуация, в которой их стандартного функционала не хватает. Рассмотрим следующий код:

```
#include <iostream>

class A
{
private:
    int m_x;
public:
    A(int x) : m_x(x)
    {
        if (x <= 0)
            throw 1;
    }
};

class B : public A
{
public:
    B(int x) : A(x)
    {
        // Что произойдет, если создать A не удастся, а исключение нужно обрабатывать здесь?
    }
};

int main()
{
    try
    {
        B b(0);
    }
    catch (int)
    {
        std::cout << "Oops!\n";
    }
}
```

В программе, приведенной выше, дочерний класс B вызывает конструктор родительского класса A, который генерирует исключение (при успешном выполнении условия). Поскольку объект `b` создается в блоке try функции main(), то если A выбросит исключение, блок try функции main() поймает его и передаст обработчику `catch (int)`.

Следовательно, результат выполнения программы:

```
Oops!
```

Но что, если нам нужно обрабатывать исключения внутри класса B? Вызов конструктора родительского класса A происходит через список инициализации членов, перед выполнением тела конструктора класса B. Поэтому использовать стандартный блок try здесь не получится.

В этой ситуации мы должны использовать слегка модифицированный блок try - **функциональный try-блок**. Функциональный try-блок используется для расположения обработчика исключений вокруг всего тела функции, а не только вокруг её определенной части (блока кода).

Синтаксис немного сложен для описания, поэтому рассмотрим всё на примере:

```
#include <iostream>

class A
{
private:
    int m_x;
public:
    A(int x) : m_x(x)
    {
        if (x <= 0)
            throw 1;
    }
};

class B : public A
{
public:
    B(int x) try : A(x) // обратите внимание на ключевое слово try здесь
    {
    }
    catch (...) // обратите внимание, этот блок находится на том же уровне отступа, что и конструктор
    {
                // Исключения из списка инициализации членов класса B или тела конструктора обрабатываются здесь

                std::cerr << "Construction of A failed\n";

                // Если мы здесь не будем явно выбрасывать исключение, то текущее (пойманное) исключение будет повторно сгенерировано и отправлено в стек вызовов
    }
};

int main()
{
    try
    {
        B b(0);
    }
    catch (int)
    {
        std::cout << "Oops!\n";
    }
```

```
41.}
```

Результат выполнения программы:

```
Construction of A failed
Oops!
```

Рассмотрим эту программу более детально.

Во-первых, обратите внимание на добавление ключевого слова try перед списком инициализации членов класса B. Это означает, что всё, что находится после этого ключевого слова (вплоть до конца функции), рассматривается как часть блока try.

Во-вторых, блок catch находится на том же уровне отступа, что и вся функция. Любое исключение, выброшенное между ключевым словом try и концом тела конструктора, будет обработано этим же блоком catch.

Наконец, в отличие от обычных блоков catch, которые либо обрабатывают исключения, либо выбрасывают новое исключение, либо повторно генерируют пойманное исключение, при использовании функциональных try-блоков вы должны либо выбросить новое исключение, либо повторно сгенерировать пойманное исключение. Если вы это не сделаете, то пойманное исключение будет повторно сгенерировано и стек начнет раскручиваться.

В программе, приведенной выше, поскольку мы явно не генерируем исключение внутри блока catch, исключение повторно генерируется и передается caller-у на уровень выше, т.е. в функцию main(). Блок catch функции main() ловит и обрабатывает исключение и мы получаем `Oops!`.

Хотя функциональные try-блоки также могут использоваться и с обычными функциями, которые не являются методами класса, это не распространенная практика, так как случаев, где они могут быть полезны, очень и очень мало. Они почти всегда используются только с конструкторами!

Не используйте функциональный try-блок для очистки памяти

Если операция создания объекта неуспешна, то деструктор класса не вызывается. Следовательно, у вас может возникнуть соблазн использовать функциональный try-блок для очистки классом частично выделенной ему памяти. Однако, обращение к членам объекта, который не создался, считается неопределенным поведением, так как объект «мёртв» еще до начала выполнения блока catch. Это означает, что вы не

можете использовать функциональный try-блок для выполнения очистки памяти класса.

Заключение

Функциональные try-блоки полезны в основном для записывания ошибок в лог-файл перед их передачей на уровень выше в стеке вызовов или для изменения типа выбрасываемого исключения.

Урок №196. Недостатки и опасности использования исключений

Кроме преимуществ, исключения имеют и недостатки. Этот урок не предназначен быть исчерпывающим в плане разбора недостатков исключений, он предназначен лишь указать на некоторые основные проблемы, которые могут возникнуть при использовании исключений (или при принятии решения об их использовании).

Очистка памяти

Одной из самых больших проблем, с которой сталкиваются начинающие программисты, используя исключения, является проблема очистки выделенных ресурсов после генерации исключения. Рассмотрим следующий пример:

```
try
{
    openFile(filename);
    writeFile(filename, data);
    closeFile(filename);
}
catch (FileException &exception)
{
    cerr << "Failed to write to file: " << exception.what() << std::endl;
}
```

Что произойдет, если функция writeFile() не сработает и выбросит объект класса-исключения FileException? К этому моменту мы уже открыли файл, и точка выполнения перейдет к обработчику catch, который выведет ошибку и завершит свое выполнение. Обратите внимание, операция закрытия файла `closeFile(filename)` никогда не выполнится! Этот код должен быть переписан следующим образом:

```
try
{
    openFile(filename);
    writeFile(filename, data);
    closeFile(filename);
}
catch (FileException &exception)
{
    // Убеждаемся, что файл закрыт
    closeFile(filename);
    // Выводим ошибку
    cerr << "Failed to write to file: " << exception.what() << std::endl;
}
```

Такой тип ошибок часто возникает и в другой форме при работе с динамическим выделением памяти:

```
try
{
    Person *alex = new Person("Alex", 20, PERSON_MALE);
    processPerson(alex);
    delete alex;
}
catch (PersonException &exception)
{
    cerr << "Failed to process person: " << exception.what() << '\n';
}
```

Если функция processPerson() выбросит исключение, то точка выполнения перейдет к обработчику catch. В результате, память, выделенная `alex`, никогда не освободится! Этот пример немного сложнее, чем предыдущий, поскольку `alex` находится внутри блока try, и выходит из области видимости при завершении выполнения блока try. Это означает, что обработчик исключений вообще не может получить доступ к `alex` (этот объект к моменту выполнения блока catch уже будет уничтожен), поэтому возможности освободить память у нас нет.

Однако есть два относительно простых способа это исправить. Во-первых, нужно объявить `alex` вне блока try, чтобы он не уничтожался при завершении выполнения блока try:

```
Person *alex = NULL;
try
{
    alex = new Person("Alex", 20, PERSON_MALE);
    processPerson(alex);
    delete alex;
}
catch (PersonException &exception)
{
    delete alex;
    cerr << "Failed to process person: " << exception.what() << '\n';
}
```

Поскольку `alex` объявлен вне блока try, то он доступен как в блоке try, так и в обработчике catch. Это означает, что обработчик catch сможет выполнить очистку памяти должным образом.

Второй способ - локально использовать объект класса, который умеет самостоятельно выполнять после себя очистку памяти при выходе из области видимости (его еще называют ***«умным указателем»***). Стандартная библиотека C++ предоставляет класс std::unique_ptr, который может использоваться в этих целях.

std::unique_ptr – это шаблон класса, который содержит указатель и освобождает выделенную ему память при выходе из области видимости.

```
#include <memory> // для std::unique_ptr

try
{
    Person *alex = new Person("Alex", 20, PERSON_MALE);
    unique_ptr<Person> upAlex(alex); // upAlex теперь владеет alex-ом

    ProcessPerson(alex);

    // Когда upAlex выйдет из области видимости, он удалит alex и выполнит соответствующую очистку выделенных ресурсов
}
catch (PersonException &exception)
{
    cerr << "Failed to process person: " << exception.what() << '\n';
}
```

Мы поговорим больше об умных указателях на соответствующих уроках.

Исключения и деструкторы

В отличие от конструкторов, где генерация исключений может быть полезным способом указать, что создать объект не удалось, исключения *никогда* не должны генерироваться в деструкторах.

Проблема возникает, когда исключение генерируется в деструкторе во время раскручивания стека. Если это происходит, то компилятор оказывается в ситуации, когда он не знает, продолжать ли процесс раскручивания стека или обработать новое исключение. Конечным результатом будет немедленное прекращение выполнения вашей программы.

Следовательно, лучшее, что вы можете сделать — это не использовать исключения в деструкторах. Лучше вместо этого записать ошибку в лог-файл.

Проблемы с производительностью

Исключения имеют свою небольшую цену производительности. Они увеличивают размер вашего исполняемого файла и также могут заставить его выполняться медленнее из-за дополнительной проверки, которая должна быть выполнена. Тем не менее, основное снижение производительности происходит при выбрасывании исключения. В этот момент стек начинает раскручиваться, и выполняется поиск соответствующего обработчика исключений, что само по себе является относительно затратной операцией.

> ***Примечание*:** Некоторые современные компьютерные архитектуры поддерживают модель исключений **«Исключения с нулевой стоимостью»** (или ***"исключения zero-cost"***). Исключения с нулевой стоимостью, если они поддерживаются, не требуют дополнительных затрат при выполнении программы в случае отсутствия ошибок (именно в этом случае мы больше всего заботимся о производительности). Тем не менее, исключения с нулевой стоимостью потребляют еще больше ресурсов в случае обнаружения исключения.

В каких случаях стоит использовать исключения?

Исключения и их обработку лучше всего использовать, **если выполняются все следующие условия**:

- Обрабатываемая ошибка возникает редко.
- Ошибка является серьезной, и выполнение программы не может продолжаться без её обработки.
- Ошибка не может быть обработана в том месте, где она возникает.
- Нет хорошего альтернативного способа вернуть код ошибки обратно в caller.

В качестве примера давайте рассмотрим функцию, которая ожидает, что пользователь передаст имя файла на диске. Ваша функция откроет этот файл, прочитает определенные данные, закроет файл и передаст определенный результат обратно в caller. Теперь допустим, что пользователь передал имя несуществующего файла или пустую строку. Отличный ли это сценарий для использования исключений?

Давайте рассмотрим:

- Первые два условия выполняются: это не то, что происходит часто, и ваша функция не может возвратить результат (т.е. продолжать свое выполнение), если у нее не будет данных для работы.
- Функция также не может обработать эту ошибку: повторно запрашивать пользователя ввести (новое) имя файла — это не то, что должна выполнять данная функция, и это может быть даже неуместным (в зависимости от логики реализации вашей программы).
- И четвертое условие: есть ли хороший альтернативный способ возвратить код ошибки обратно в caller? Это зависит от деталей реализации вашей программы. Если такой способ есть (например, вы можете вернуть нулевой указатель или код возврата, указывающий на ошибку), это, вероятно, лучшее решение. Если нет, то лучшим решением будет использовать исключение и его дальнейшую обработку.

Учитывая всё вышесказанное вы должны научиться понимать и отличать риски и пользу от использования (и обработки) исключений.

Глава №14. Итоговый тест

Вот мы и прошли тему «Исключения в языке C++», пора закрепить полученные знания.

Теория

Обработка исключений обеспечивает механизм отделения обработки ошибок от общего потока выполнения кода. Это предоставляет больше свободы для обработки ошибок в каждой конкретной ситуации, уменьшая многие (если не все) проблемы, порожденные использованием кодов возврата.

Оператор throw используется для выбрасывания (генерации) исключения. **Блоки try** ловят исключения, которые выбрасываются в их пределах, и перенаправляют пойманные исключения **блокам catch**. Если тип исключения совпадает с типом обработчика catch, обработчик catch обрабатывает пойманное исключение.

Исключения обрабатываются немедленно. Если исключение было выброшено, то точка выполнения переходит к ближайшему блоку try в поисках обработчика catch, который сможет обработать сгенерированное исключение. Если блок try не был найден или тип блока catch не совпал, то **стек начнет раскручиваться** до тех пор, пока не будет найден соответствующий обработчик catch. Если стек полностью раскручен, а обработчик catch так и не найден, то программа завершит свое выполнение с ошибкой необработанного исключения.

Исключения могут быть любого типа данных, включая **классы**.

Блоки catch могут обрабатывать исключения определенного типа данных или сразу всех типов данных — так называемые **«обработчики catch–all»**, в которых эллипсис (…) указывается в качестве типа исключения. Блок catch, принимающий объект родительского класса по ссылке, также будет перехватывать и объекты дочерних классов. Все исключения, выбрасываемые Стандартной библиотекой C++, являются дочерними классу-исключению **std::exception** (который находится в заголовочном файле exception), поэтому блок catch, принимая std::exception по ссылке, также будет обрабатывать все остальные исключения, генерируемые Стандартной библиотекой C++. **Метод what()** используется для конкретизации того, какой тип std::exception был выброшен.

Внутри блока catch может генерироваться новое исключение. Поскольку это новое исключение выбрасывается за пределами блока try, связанного с этим блоком catch, то оно не будет перехвачено текущим блоком catch (в котором выброшено).

Исключения можно **повторно выбрасывать** в блоке catch с помощью ключевого слова throw без указания какого-либо идентификатора. Не выбрасывайте повторно пойманное блоком catch исключение (объект класса-исключения), дабы избежать обрезки объекта.

Функциональные try-блоки используются для расположения обработчика исключений вокруг всего тела функции, а не только вокруг её определенной части (блока кода). Обычно они используются только с конструкторами дочерних классов.

Никогда не выбрасывайте исключения внутри деструкторов.

Наконец, обработка исключений имеет свою стоимость. В большинстве случаев код, использующий исключения, будет работать медленнее, а стоимость обработки исключения относительно высока. Вы должны использовать исключения только для обработки исключительных ситуаций, а не для тривиальных случаев, в которых можно обойтись альтернативными методами (например, стейтментами assert).

Тест

Напишите класс Fraction, конструктор которого принимает числитель и знаменатель. Если пользователь передал в качестве знаменателя `0`, то выбрасывайте исключение типа std::runtime_error (которое находится в заголовочном файле stdexcept). В функции main() попросите пользователя ввести два целых числа. Если числа, которые ввел пользователь, корректные, то выводите создаваемый объект класса Fraction. Если же числа недопустимые, то вы должны обрабатывать исключение типа std::exception и сообщить пользователю, что он ввел некорректные данные.

> ***Подсказка*:** std::runtime_error является дочерним классу-исключению std::exception, поэтому у вас должен быть только один блок catch.

Пример выполнения программы:

```
Enter the numerator: 7
Enter the denominator: 0
Your fraction has an invalid denominator.
```

Урок №197. Умные указатели и Семантика перемещения

Рассмотрим функцию, в которой динамически выделяется переменная:

```
1. void myFunction()
2. {
3.     Item *ptr = new Item; // Item-ом может быть структура или класс
4. 
5.     // Делаем что-либо с ptr здесь
6. 
7.     delete ptr;
8. }
```

Хотя код, приведенный выше, кажется довольно простым, можно очень легко забыть в конце освободить память, выделенную `ptr`. Даже если вы не забудете это сделать, существует множество причин, по которым `ptr` не будет удален. Это может произойти из-за досрочного возврата return:

```
1. #include <iostream>
2. 
3. void myFunction()
4. {
5.     Item *ptr = new Item;
6. 
7.     int a;
8.     std::cout << "Enter an integer: ";
9.     std::cin >> a;
10. 
11.     if (a == 0)
12.         return; // функция выполняет досрочный возврат, вследствие чего ptr
    не будет удален!
13. 
14.     // Делаем что-либо с ptr здесь
15. 
16.     delete ptr;
17. }
```

Или из-за генерации исключения:

```
1. #include <iostream>
2. 
3. void myFunction()
4. {
5.     Item *ptr = new Item;
6. 
7.     int a;
8.     std::cout << "Enter an integer: ";
9.     std::cin >> a;
10. 
11.     if (a == 0)
12.         throw 0; // генерируется исключение > функция преждевременно завершает
    свое выполнение > ptr не удаляется!
13. 
14.     // Делаем что-либо с ptr здесь
15. 
```

```
16.     delete ptr;
17. }
```

В обоих случая функция завершает свое выполнения до того, как произойдет удаление `ptr`. Следовательно, мы получим утечку памяти, и так будет повторяться до тех пор, пока будет вызываться эта функция и пока будет срабатывать её досрочное завершение (из-за генерации исключения, досрочного return-а или чего-либо другого). По сути, такие проблемы возникают из-за того, что указатели не имеют встроенного механизма самостоятельной очистки памяти после себя.

Умные указатели

Одна из лучших особенностей классов – это деструкторы, которые автоматически выполняются при выходе объекта класса из области видимости. При выделении памяти в конструкторе класса, вы можете быть уверены, что эта память будет освобождена в деструкторе при уничтожении объекта класса (независимо от того, выйдет ли он из области видимости, будет ли явно удален и т.д.). Это лежит в основе парадигмы программирования RAII.

Так что же, выходом является использование класса для управления указателями и выполнения соответствующей очистки памяти? Да, именно так!

Например, рассмотрим класс, единственными задачами которого является хранение и «управление» переданным ему указателем, а затем корректное освобождение памяти при выходе объекта класса из области видимости. До того момента, пока объекты этого класса создаются как локальные переменные, мы можем гарантировать, что, как только они выйдут из области видимости (независимо от того, когда или как), переданный указатель будет уничтожен.

Вот первый набросок:

```
1. #include <iostream>
2.
3. template<class T>
4. class Auto_ptr1
5. {
6.     T* m_ptr;
7. public:
8.     // Получаем указатель для "владения" через конструктор
9.     Auto_ptr1(T* ptr=nullptr)
10.         :m_ptr(ptr)
11.     {
12.     }
13.
14.     // Деструктор позаботится об удалении указателя
15.     ~Auto_ptr1()
16.     {
17.         delete m_ptr;
```

```
    }

    // Выполняем перегрузку оператора разыменования и оператора ->, чтобы иметь возможность использовать Auto_ptr1 как m_ptr
    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
};

// Класс для проверки работоспособности вышеприведенного кода
class Item
{
public:
    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
};

int main()
{
    Auto_ptr1<Item> item(new Item); // динамическое выделение памяти

        // ... но никакого явного delete здесь не нужно

    // Также обратите внимание на то, что Item-у в угловых скобках не требуется символ *, поскольку это предоставляется шаблоном класса

    return 0;
} // item выходит из области видимости здесь и уничтожает выделенный Item вместо нас
```

Результат выполнения программы:

```
Item acquired
Item destroyed
```

Рассмотрим детально, как работают эти программа и класс. Сначала мы динамически выделяем объект класса Item и передаем его в качестве параметра нашему шаблону класса Auto_ptr1. С этого момента объект `item` класса Auto_ptr1 владеет выделенным объектом класса Item (Auto_ptr1 имеет композиционную связь с `m_ptr`). Поскольку `item` объявлен в качестве локальной переменной и имеет область видимости блока, он выйдет из области видимости после завершения выполнения блока, в котором находится, и будет уничтожен. А поскольку это объект класса, то при его уничтожении будет вызван деструктор Auto_ptr1. Этот деструктор и обеспечит удаление указателя Item, который он хранит!

До тех пор, пока объект класса Auto_ptr1 определен как локальная переменная (с автоматической продолжительностью жизни, отсюда и часть *«Auto»* в имени класса), Item гарантированно будет уничтожен в конце блока, в котором он объявлен, независимо от того, как этот блок (функция main()) завершит свое выполнение (досрочно или нет).

Такой класс называется умным указателем. **Умный указатель** - это класс, предназначенный для управления динамически выделенной памятью и обеспечения освобождения (удаления) выделенной памяти при выходе объекта этого класса из области видимости. Соответственно, встроенные (обычные) указатели иногда еще называют *«глупыми указателями»*, так как они не могут выполнять после себя очистку памяти.

Теперь вернемся к нашему примеру с myFunction() и покажем, как использование класса умного указателя сможет решить нашу проблему:

```
#include <iostream>

template<class T>
class Auto_ptr1
{
    T* m_ptr;
public:
    // Получаем указатель для "владения" через конструктор
    Auto_ptr1(T* ptr=nullptr)
        :m_ptr(ptr)
    {
    }

    // Деструктор позаботится об удалении указателя
    ~Auto_ptr1()
    {
        delete m_ptr;
    }

    // Выполняем перегрузку оператора разыменования и оператора ->, чтобы иметь возможность использовать Auto_ptr1 как m_ptr
    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
};

// Класс для проверки работоспособности вышеприведенного кода
class Item
{
public:
    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
    void sayHi() { std::cout << "Hi!\n"; }
};

void myFunction()
{
    Auto_ptr1<Item> ptr(new Item); // ptr теперь "владеет" Item-ом

    int a;
    std::cout << "Enter an integer: ";
    std::cin >> a;

    if (a == 0)
        return; // досрочный возврат функции

    // Использование ptr
    ptr->sayHi();
```

```
47. }
48. 
49. int main()
50. {
51.     myFunction();
52. 
53.     return 0;
54. }
```

Если пользователь введет ненулевое целое число, то результат выполнения программы:

```
Item acquired
Enter an integer: 7
Hi!
Item destroyed
```

Если же пользователь введет ноль, то функция myFunction() завершит свое выполнение досрочно, и мы увидим:

```
Item acquired
Enter an integer: 0
Item destroyed
```

Обратите внимание, даже в случае, когда пользователь введет ноль, и функция завершит свое выполнение досрочно, Item по-прежнему будет корректно удален.

Поскольку переменная `ptr` является локальной переменной, то она уничтожается при завершении выполнения функции (независимо от того, как это будет сделано: досрочно или нет). И поскольку деструктор Auto_ptr1 выполняет очистку Item, то мы можем быть уверены, что Item будет корректно удален.

Критический недостаток

Класс Auto_ptr1, приведенный выше, имеет критическую ошибку, которая скрывается за некоторым автоматически генерируемым кодом. Прежде чем продолжить, посмотрите, сможете ли вы определить, что это за ошибка.

> ***Подсказка***: Подумайте, какие части класса генерируются автоматически, если вы их не предоставляете самостоятельно.

(Напряженная музыка)

Хорошо, время истекло.

Мы не будем сейчас вам это говорить, мы сейчас вам это покажем:

```
#include <iostream>

// Шаблон класса тот же, что и в примере, приведенном выше
template<class T>
class Auto_ptr1
{
    T* m_ptr;
public:
    Auto_ptr1(T* ptr=nullptr)
        :m_ptr(ptr)
    {
    }

    ~Auto_ptr1()
    {
        delete m_ptr;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
};

class Item
{
public:
    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
};

int main()
{
    Auto_ptr1<Item> item1(new Item);
    Auto_ptr1<Item> item2(item1); // в качестве альтернативы вы можете не инициализировать item2 значением item1, а просто выполнить присваивание item2 = item1

    return 0;
}
```

Результат выполнения программы:

```
Item acquired
Item destroyed
Item destroyed
```

Очень вероятно (но не обязательно), что в нашей программе произойдет сбой именно в этот момент. Нашли проблему? Поскольку мы не предоставили конструктор копирования или свой оператор присваивания (перегрузку оператора присваивания), то язык C++ предоставил их самостоятельно. И то, что он предоставил, выполняет поверхностное копирование. Поэтому, когда мы инициализируем `item2` значением `item1`, оба объекта класса Auto_ptr1 указывают на один и тот же Item. Когда `item2` выходит из области видимости, он удаляет Item,

оставляя `item1` с висячим указателем. Когда же `item1` отправляется на удаление своего (уже удаленного) Item, происходит «Бум!».

Вы получите ту же проблему, используя следующую функцию:

```
1. void passByValue(Auto_ptr1<Item> item)
2. {
3. }
4.
5. int main()
6. {
7.     Auto_ptr1<Item> item1(new Item);
8.     passByValue(item1)
9.
10.    return 0;
11. }
```

В этой программе `item1` передается по значению в параметр `item` функции passByValue(), что приведет к дублированию указателя Item. Мы вновь получим «Бум!».

Так быть не должно. Что мы можем сделать?

Мы можем явно определить и удалить конструктор копирования с оператором присваивания, тем самым предотвращая выполнение любого копирования. Это также предотвратит передачу по значению.

Но как нам тогда вернуть Auto_ptr1 из функции обратно в caller?

```
1. ??? generateItem()
2. {
3.      Item *item = new Item;
4.      return Auto_ptr1(item);
5. }
```

Мы не можем вернуть Auto_ptr1 по ссылке, так как локальный Auto_ptr1 будет уничтожен в конце функции, и в caller передастся ссылка, которая будет указывать на удаленную память. Передача по адресу имеет ту же проблему. Мы могли бы вернуть указатель `item` по адресу, но потом мы можем забыть удалить `item`, что является основным смыслом использования умных указателей. Так что возврат Auto_ptr1 по значению — это единственная опция, которая имеет смысл, но тогда мы получим поверхностное копирование, дублирование указателей и «Бум!».

Другой вариант — переопределить конструктор копирования и оператор присваивания для выполнения глубокого копирования. Таким образом, мы, по крайней мере, гарантированно избежим дублирования указателей (которые будут указывать на один и тот же объект). Но глубокое копирование может быть

затратной операцией (а также нежелательной или даже невозможной), и мы не хотим делать ненужные копии объектов просто для того, чтобы вернуть Auto_ptr1 из функции. Кроме того, присваивание или инициализация глупого указателя не копирует объект, на который указывает, так почему же мы ожидаем, что умные указатели будут вести себя по-другому?

Что же делать?

Семантика перемещения

А что, если бы наш конструктор копирования и оператор присваивания не копировали указатель (семантика копирования), а передавали владение указателем из источника в объект назначения? Это основная идея семантики перемещения. **Семантика перемещения** означает, что класс, вместо копирования, передает право собственности на объект.

Давайте обновим наш класс Auto_ptr1 с использованием семантики перемещения:

```
#include <iostream>

template<class T>
class Auto_ptr2
{
    T* m_ptr;
public:
    Auto_ptr2(T* ptr=nullptr)
        :m_ptr(ptr)
    {
    }

    ~Auto_ptr2()
    {
        delete m_ptr;
    }

    // Конструктор копирования, который реализовывает семантику перемещения
    Auto_ptr2(Auto_ptr2& a) // примечание: Ссылка не является константной
    {
        m_ptr = a.m_ptr; // перемещаем наш глупый указатель от источника к нашему локальному объекту
        a.m_ptr = nullptr; // подтверждаем, что источник больше не владеет указателем
    }

    // Оператор присваивания, который реализовывает семантику перемещения
    Auto_ptr2& operator=(Auto_ptr2& a) // примечание: Ссылка не является константной
    {
        if (&a == this)
            return *this;

        delete m_ptr; // подтверждаем, что удалили любой указатель, который наш локальный объект имел до этого
```

```
32.         m_ptr = a.m_ptr; // затем перемещаем наш глупый указатель из источника к нашему локальному объекту
33.         a.m_ptr = nullptr; // подтверждаем, что источник больше не владеет указателем
34.         return *this;
35.     }
36.
37.     T& operator*() const { return *m_ptr; }
38.     T* operator->() const { return m_ptr; }
39.     bool isNull() const { return m_ptr == nullptr;  }
40. };
41.
42. class Item
43. {
44. public:
45.     Item() { std::cout << "Item acquired\n"; }
46.     ~Item() { std::cout << "Item destroyed\n"; }
47. };
48.
49. int main()
50. {
51.     Auto_ptr2<Item> item1(new Item);
52.     Auto_ptr2<Item> item2; // начнем с nullptr
53.
54.     std::cout << "item1 is " << (item1.isNull() ? "null\n" : "not null\n");
55.     std::cout << "item2 is " << (item2.isNull() ? "null\n" : "not null\n");
56.
57.     item2 = item1; // item2 теперь является "владельцем" значения item1, объекту item1 присваивается null
58.
59.     std::cout << "Ownership transferred\n";
60.
61.     std::cout << "item1 is " << (item1.isNull() ? "null\n" : "not null\n");
62.     std::cout << "item2 is " << (item2.isNull() ? "null\n" : "not null\n");
63.
64.     return 0;
65. }
```

Результат выполнения программы:

```
Item acquired
item1 is not null
item2 is null
Ownership transferred
item1 is null
item2 is not null
Item destroyed
```

Обратите внимание, перегруженный operator= передает право собственности на `m_ptr` от `item1` к `item2`! Следовательно, у нас не выполняется дублирования указателей, и всё аккуратно очищается (удаляется).

std::auto_ptr и почему его лучше не использовать

Теперь самое время поговорить о std::auto_ptr. std::auto_ptr, представленный в C++98, был первой попыткой в языке C++ сделать стандартизированный умный указатель. В std::auto_ptr решили реализовать семантику перемещения точно так же, как это сделано в классе Auto_ptr2.

Однако, std::auto_ptr (как и наш класс Auto_ptr2) имеет ряд проблем, которые делают его использование опасным.

Во-первых, поскольку std::auto_ptr реализовывает семантику перемещения через конструктор копирования и оператор присваивания, то передача std::auto_ptr в функцию по значению приведет к тому, что ваш Item будет перемещен в параметр функции и, следовательно, будет уничтожен в конце функции, когда параметры этой функции выйдут из области видимости (в нашем классе Auto_ptr2 передача выполняется по ссылке). Затем, когда вы попытаетесь получить доступ к аргументу std::auto_ptr из caller-а (не осознавая, что он был передан и удален), вы внезапно выполните разыменование нулевого указателя. Бум!

Во-вторых, std::auto_ptr всегда удаляет свое содержимое, используя оператор delete, который не работает с массивами. Это означает, что std::auto_ptr не будет правильно работать с динамическими массивами, поскольку использует неправильный тип удаления. Хуже того, std::auto_ptr не помешает вам передать ему динамический массив, который затем будет неправильно обработан, что приведет к утечке памяти.

Наконец, std::auto_ptr не очень хорошо работает со многими другими классами из Стандартной библиотеки C++ (особенно с контейнерными классами и классами алгоритмов). Это происходит из-за того, что классы Стандартной библиотеки C++ предполагают, что, когда они копируют элемент, они фактически выполняют копирование, а не перемещение.

Из-за вышеупомянутых недостатков в C++11 перестали использовать std::auto_ptr, а в C++17 планировали удалить его из Стандартной библиотеки C++.

Правило: std::auto_ptr устарел и не должен использоваться. Используйте вместо него std::unique_ptr или std::shared_ptr.

Что дальше?

Основная проблема с std::auto_ptr заключается в том, что до C++11 в языке C++ просто не было механизма, позволяющего отличить «семантику копирования» от «семантики перемещения». Переопределение семантики копирования для реализации семантики перемещения привело к неопределенным результатам и непреднамеренным ошибкам. Например, вы можете написать `item1 = item2` и вообще не знать, изменится ли `item2` или нет!

По этой причине в C++11 понятие «перемещение» было определено формально, в следствии чего в язык C++ было добавлено «семантику перемещения», чтобы должным образом отличать копирование от перемещения. Теперь, когда вы понимаете, чем семантика перемещения может быть полезной, мы рассмотрим её детально на следующих уроках.

В C++11 std::auto_ptr был заменен кучей **других типов умных указателей**:

- std::scoped_ptr;
- std::unique_ptr;
- std::weak_ptr;
- std::shared_ptr.

Мы также рассмотрим два самых популярных из них: std::unique_ptr (который является прямой заменой std::auto_ptr) и std::shared_ptr.

Урок №198. Ссылки r-value

Мы уже ранее рассматривали l-values и r-values. Тогда мы говорили, что вам не нужно слишком беспокоиться о них. И это было правдой до версии C++11. Сейчас же, для понимания семантики перемещения, нам нужно пересмотреть эту тему.

l-values и r-values

Несмотря на то, что в обоих терминах есть слово «value» (значение), l-values и r-values на самом деле являются не свойствами значений, а скорее свойствами выражений.

Каждое выражение в языке C++ имеет два свойства: **тип** и **категорию значения** (определяет, можно ли результат выражения присвоить другому объекту). В C++03 и в более ранних версиях C++ l-values и r-values были единственными категориями значений.

О **l-value** проще всего думать, как о функции, объекте или переменной (или выражении, результатом которого является функция, объект или переменная), которая имеет свой адрес памяти. Изначально l-values были определены как «значения, которые должны находиться в левой части операции присваивания». Однако позже в язык C++ было добавлено ключевое слово const, и l-values были разделены на **две подкатегории**:

- Модифицируемые l-values, которые можно изменить (например, переменной `x` можно присвоить другое значение).
- Немодифицируемые l-values, которые являются const (например, константа `PI`).

О **r-value** проще всего думать, как «обо всем остальном, что не является l-value». Это литералы (например, `5`), временные значения (например, `x + 1`) и анонимные объекты (например, `Fraction(7, 3)`). r-values имеют область видимости выражения (уничтожаются в конце выражения, в котором находятся) и им нельзя что-либо присвоить. Этот запрет на присваивание имеет смысл, так как присваивая значение мы вызываем в объекта побочные эффекты.

А поскольку r-values имеют область видимости выражения, то, если бы мы присваивали какое-либо значение для r-value, r-value либо выходило бы из области видимости, прежде чем у нас была бы возможность использовать присвоенное значение в следующем выражении (что делает операцию присваивания

бесполезной), либо нам пришлось бы использовать переменную с побочным эффектом, который возникал бы больше одного раза в выражении (что, как вы уже должны знать, привело бы к неопределенным результатам!).

Для поддержки семантики перемещения в C++11 ввели **3 новые категории значений**:

- pr-values;
- x-values;
- gl-values.

Их понимание не столь важно в изучении или эффективном использовании семантики перемещения, поэтому в значительной степени мы будем их игнорировать. Однако, если вам интересно, то вы можете изучить их более детально **здесь**.

Ссылки l-value

До версии C++11 существовал только один тип ссылок, его называли просто - *"ссылка"*. В C++11 этот тип ссылки еще называют "ссылкой l-value". **Ссылки l-value** могут быть инициализированы только изменяемыми l-values.

Ссылки l-value	Могут ли быть инициализированы	Могут ли значения изменяться
Изменяемые l-values	Да	Да
Неизменяемые l-values	Нет	Нет
r-values	Нет	Нет

Ссылки l-value на константные объекты могут быть инициализированы с помощью как l-values, так и r-values.

Однако эти значения не могут быть изменены (константы не изменяют свои значения).

Ссылки l-value на const	Могут ли быть инициализированы	Могут ли значения изменяться
Изменяемые l-values	Да	Нет
Неизменяемые l-values	Да	Нет
r-values	Да	Нет

Ссылки l-value на константные объекты особенно полезны, так как позволяют передавать аргументы любого типа (l-value или r-value) в функцию без выполнения копирования аргумента.

Ссылки r-value

В C++11 добавили новый тип ссылок - ссылки r-value. **Ссылки r-value** - это ссылки, которые инициализируются только значениями r-values. Хотя ссылка l-value создается с использованием одного амперсанда, ссылка r-value создается с использованием двойного амперсанда:

```
1. int x = 7;
2. int &lref = x; // инициализация ссылки l-value переменной x (значение l-value)
3. int &&rref = 7; // инициализация ссылки r-value литералом 7 (значение r-value)
```

Ссылки r-value не могут быть инициализированы значениями l-values.

Ссылки r-value	Могут ли быть инициализированы	Могут ли значения изменяться
Изменяемые l-values	Нет	Нет
Неизменяемые l-values	Нет	Нет

r-values	Да	Да

И ссылки r-value на константные объекты:

Ссылки r-value на const	Могут ли быть инициализированы	Могут ли значения изменяться
Изменяемые l-values	Нет	Нет
Неизменяемые l-values	Нет	Нет
r-values	Да	Нет

Ссылки r-value имеют **два полезных свойства**:

- Они увеличивают продолжительность жизни объекта, которым инициализируются, до продолжительности жизни ссылки r-value (ссылки l-value на константные объекты также могут это делать).
- Неконстантные ссылки r-value позволяют нам изменять значения r-values, на которые указывают ссылки r-value!

Рассмотрим следующую программу:

```
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
    {
        out << f1.m_numerator << "/" << f1.m_denominator;
        return out;
    }
};

```

```
int main()
{
    Fraction &&rref = Fraction(4, 7); // ссылка r-value на анонимный объект класса Fraction
    std::cout << rref << '\n';

    return 0;
} // rref (и анонимный объект класса Fraction) выходит из области видимости здесь
```

Результат:

```
4/7
```

Создаваемый анонимный объект `Fraction(4, 7)` обычно вышел бы из области видимости в конце выражения, в котором он определен. Однако, так как мы инициализируем ссылку r-value этим анонимным объектом, то его продолжительность жизни увеличивается до продолжительности жизни самой ссылки r-value, т.е. до конца функции main(). Затем мы используем ссылку r-value для вывода значения анонимного объекта класса Fraction.

Теперь рассмотрим менее наглядный пример:

```
#include <iostream>

int main()
{
    int &&rref = 7; // поскольку мы инициализируем ссылку r-value литералом 7, то создается временный объект со значением 7, на который указывает ссылка r-value
    rref = 12;
    std::cout << rref;

    return 0;
}
```

Результат выполнения программы:

```
12
```

Хотя это может показаться странным, но при инициализации ссылки r-value литералом, создается временный объект, на который ссылается ссылка r-value (она не ссылается на сам литерал).

Ссылки r-value не очень часто используются так, как это представлено в вышеприведенных примерах.

Ссылки r-value в качестве параметров функции

Ссылки r-value чаще всего используются в качестве параметров функции. Это наиболее полезно при перегрузке функций, когда вы хотите, чтобы выполнение функции отличалось в зависимости от аргументов (l-values или r-values). Например:

```
#include <iostream>

void fun(const int &lref) // перегрузка функции для работы с аргументами l-values
{
    std::cout << "l-value reference to const\n";
}

void fun(int &&rref) // перегрузка функции для работы с аргументами r-values
{
    std::cout << "r-value reference\n";
}

int main()
{
    int x = 7;
    fun(x); // аргумент l-value вызывает функцию с ссылкой l-value
    fun(7); // аргумент r-value вызывает функцию с ссылкой r-value

    return 0;
}
```

Результат выполнения программы:

```
l-value reference to const
r-value reference
```

Как вы можете видеть, при передаче l-value, выполняется перегрузка функции с ссылкой l-value в качестве параметра, а при передаче r-value — выполняется перегрузка функции с ссылкой r-value в качестве параметра.

Зачем это может нам понадобиться? Более детально об этом мы поговорим на следующем уроке. Излишне говорить, что это важная часть семантики перемещения.

Возврат ссылки r-value

Вы почти никогда не должны возвращать ссылку r-value из функции по той же причине, по которой вы почти никогда не должны возвращать ссылку l-value из функции. В большинстве случаев вы возвратите висячую ссылку (указывающую на удаленную память), а объект, на который будет ссылаться ссылка, выйдет из области видимости в конце функции.

Тест

Какие из следующих стейтментов, обозначенные буквами, не скомпилируются:

```
1. int main()
2. {
3.     int x;
4.
5.     // Ссылки l-value
6.     int &ref1 = x; // A
7.     int &ref2 = 7; // B
8.
9.     const int &ref3 = x; // C
10.    const int &ref4 = 7; // D
11.
12.    // Ссылки r-value
13.    int &&ref5 = x; // E
14.    int &&ref6 = 7; // F
15.
16.    const int &&ref7 = x; // G
17.    const int &&ref8 = 7; // H
18.
19.    return 0;
20. }
```

Урок №199. Конструктор перемещения и Оператор присваивания перемещением

На предыдущих уроках мы говорили об std::auto_ptr, обсуждали необходимость использования семантики перемещения и рассмотрели некоторые недостатки, которые возникают при переопределении функций, использующих семантику копирования (конструктор копирования и оператор присваивания копированием), для работы с семантикой перемещения.

На этом уроке мы рассмотрим более детально, как C++11 решил эти проблемы с помощью конструктора перемещения и оператора присваивания перемещением.

Конструктор копирования и оператор присваивания копированием

Давайте немного поговорим о семантике копирования.

Конструктор копирования используется для инициализации класса путем создания копии необходимого объекта. **Оператор присваивания копированием** (или ***«копирующее присваивание»***) используется для копирования одного класса в другой (существующий) класс. По умолчанию язык C++ автоматически предоставляет конструктор копирования и оператор присваивания копированием, если вы не предоставили их сами. Предоставляемые компилятором функции выполняют поверхностное копирование, что может вызывать проблемы у классов, которые работают с динамически выделенной памятью. Одним из вариантов решения таких проблем является переопределение конструктора копирования и оператора присваивания копированием для выполнения глубокого копирования.

Возвращаясь к нашему примеру с классом Auto_ptr (который является умным указателем) из предыдущих уроков, давайте рассмотрим версию этого класса, в которой конструктор копирования и оператор присваивания копированием переопределены для выполнения глубокого копирования:

```
1.  #include <iostream>
2.
3.  template<class T>
4.  class Auto_ptr3
5.  {
6.      T* m_ptr;
7.  public:
8.      Auto_ptr3(T* ptr = nullptr)
9.          :m_ptr(ptr)
10.     {
11.     }
12.
13.     ~Auto_ptr3()
```

```
14.     {
15.         delete m_ptr;
16.     }
17.
18.     // Конструктор копирования, который выполняет глубокое копирование x.m_ptr
    в m_ptr
19.     Auto_ptr3(const Auto_ptr3& x)
20.     {
21.         m_ptr = new T;
22.         *m_ptr = *x.m_ptr;
23.     }
24.
25.     // Оператор присваивания копированием, который выполняет глубокое
    копирование x.m_ptr в m_ptr
26.     Auto_ptr3& operator=(const Auto_ptr3& x)
27.     {
28.         // Проверка на самоприсваивание
29.         if (&x == this)
30.             return *this;
31.
32.         // Удаляем всё, что к этому моменту может хранить указатель
33.         delete m_ptr;
34.
35.         // Копируем передаваемый объект
36.         m_ptr = new T;
37.         *m_ptr = *x.m_ptr;
38.
39.         return *this;
40.     }
41.
42.     T& operator*() const { return *m_ptr; }
43.     T* operator->() const { return m_ptr; }
44.     bool isNull() const { return m_ptr == nullptr; }
45. };
46.
47. class Item
48. {
49. public:
50.     Item() { std::cout << "Item acquired\n"; }
51.     ~Item() { std::cout << "Item destroyed\n"; }
52. };
53.
54. Auto_ptr3<Item> generateItem()
55. {
56.     Auto_ptr3<Item> item(new Item);
57.     return item; // это возвращаемое значение приведет к вызову конструктора
    копирования
58. }
59.
60. int main()
61. {
62.     Auto_ptr3<Item> mainItem;
63.     mainItem = generateItem(); // эта операция присваивания приведет к вызову
    оператора присваивания копированием
64.
65.     return 0;
66. }
```

В программе, приведенной выше, мы используем функцию generateItem() для создания инкапсулированного умного указателя Item, который затем передается

обратно в функцию main(). В функции main() мы присваиваем его объекту `mainItem`.

Результат выполнения программы:

```
Item acquired
Item acquired
Item destroyed
Item acquired
Item destroyed
Item destroyed
```

Рассмотрим выполнение этой программы детально. Здесь происходит 6 ключевых действий (по одному на каждую строку вывода):

- При создании объекта `mainItem` внутри generateItem() создается локальная переменная (объект) `item` и инициализируется динамически выделенным Item, вследствие чего мы получаем первую строку вывода — `Item acquired`.
- Затем `item` возвращается обратно в функцию main() по значению. Почему по значению? Потому что `item` является локальной переменной и её нельзя вернуть по адресу или по ссылке, так как `item` будет уничтожена при завершении выполнения функции generateItem(). Таким образом, `item` — это копия (временный объект). Поскольку наш конструктор копирования выполняет глубокое копирование, то выделяется новый Item, результатом чего является вторая строка вывода — `Item acquired`.
- При завершении выполнения функции generateItem() переменная `item` выходит из области видимости, уничтожая первоначально созданный Item, в результате чего мы получаем третью строку вывода — `Item destroyed`.
- Временный объект, созданный вследствие глубокого копирования, присваивается `mainItem` в функции main() путем использования оператора присваивания копированием. Поскольку мы перегрузили оператор присваивания копированием для выполнения глубокого копирования (вместо поверхностного), то выделяется новый Item, вследствие чего мы получаем четвертую строку вывода — `Item acquired`.
- Операция присваивания временного объекта объекту `mainItem` заканчивается, и временный объект выходит из области видимости выражения и уничтожается, в результате чего мы получаем пятую строку вывода — `Item destroyed`.

- В конце функции main() объект `mainItem` выходит из области видимости, и мы получаем шестую (и последнюю) строку вывода — `Item destroyed`.

> ***Примечание***: Ваш результат может состоять из 4-х строк вывода, если ваш компилятор игнорирует возвращаемое значение из функции generateItem():
>
> ```
> Item acquired
> Item acquired
> Item destroyed
> Item destroyed
> ```

Короче говоря, используя конструктор копирования и оператор присваивания копированием с выполнением глубокого копирования мы, в итоге, выделяем и уничтожаем 3 отдельных объекта.

Неэффективно, скажете вы, но, по крайней мере, всё работает без сбоев! Однако с семантикой перемещения мы можем добиться большего.

Конструктор перемещения и оператор присваивания перемещением

В C++11 добавили две новые функции для работы с семантикой перемещения: конструктор перемещения и оператор присваивания перемещением. В то время как цель семантики копирования состоит в том, чтобы выполнять копирование одного объекта в другой, цель семантики перемещения состоит в том, чтобы переместить владение ресурсами из одного объекта в другой (что менее затратно, чем выполнение операции копирования).

Определение **конструктора перемещения и оператора присваивания перемещением** выполняется аналогично определению конструктора копирования и оператора присваивания копированием. Однако, в то время как функции с копированием принимают в качестве параметра константную ссылку l-value, функции с перемещением принимают в качестве параметра неконстантную ссылку r-value.

Вот вышеприведенный класс Auto_ptr3, но уже с добавленными конструктором перемещения и оператором присваивания перемещением. Мы не стали удалять конструктор копирования и оператор присваивания копированием:

```
1. #include <iostream>
2.
3. template<class T>
4. class Auto_ptr4
5. {
6.     T* m_ptr;
7. public:
```

```
8.      Auto_ptr4(T* ptr = nullptr)
9.          :m_ptr(ptr)
10.     {
11.     }
12.
13.     ~Auto_ptr4()
14.     {
15.         delete m_ptr;
16.     }
17.
18.     // Конструктор копирования, который выполняет глубокое копирование x.m_ptr
    в m_ptr
19.     Auto_ptr4(const Auto_ptr4& x)
20.     {
21.         m_ptr = new T;
22.         *m_ptr = *x.m_ptr;
23.     }
24.
25.     // Конструктор перемещения, который передает право собственности на x.m_ptr
     в m_ptr
26.     Auto_ptr4(Auto_ptr4&& x)
27.         : m_ptr(x.m_ptr)
28.     {
29.         x.m_ptr = nullptr; // мы поговорим об этом чуть позже
30.     }
31.
32.     // Оператор присваивания копированием, который выполняет глубокое
    копирование x.m_ptr в m_ptr
33.     Auto_ptr4& operator=(const Auto_ptr4& x)
34.     {
35.         // Проверка на самоприсваивание
36.         if (&x == this)
37.             return *this;
38.
39.         // Удаляем всё, что к этому моменту может хранить указатель
40.         delete m_ptr;
41.
42.         // Копируем передаваемый объект
43.         m_ptr = new T;
44.         *m_ptr = *x.m_ptr;
45.
46.         return *this;
47.     }
48.
49.     // Оператор присваивания перемещением, который передает право собственности
     на x.m_ptr в m_ptr
50.     Auto_ptr4& operator=(Auto_ptr4&& x)
51.     {
52.         // Проверка на самоприсваивание
53.         if (&x == this)
54.             return *this;
55.
56.         // Удаляем всё, что к этому моменту может хранить указатель
57.         delete m_ptr;
58.
59.         // Передаем право собственности на x.m_ptr в m_ptr
60.         m_ptr = x.m_ptr;
61.         x.m_ptr = nullptr; // мы поговорим об этом чуть позже
62.
63.         return *this;
64.     }
65.
```

```
66.     T& operator*() const { return *m_ptr; }
67.     T* operator->() const { return m_ptr; }
68.     bool isNull() const { return m_ptr == nullptr; }
69. };
70. 
71. class Item
72. {
73. public:
74.     Item() { std::cout << "Item acquired\n"; }
75.     ~Item() { std::cout << "Item destroyed\n"; }
76. };
77. 
78. Auto_ptr4<Item> generateItem()
79. {
80.     Auto_ptr4<Item> item(new Item);
81.     return item; // это возвращаемое значение приведет к вызову конструктора перемещения
82. }
83. 
84. int main()
85. {
86.     Auto_ptr4<Item> mainItem;
87.     mainItem = generateItem(); // эта операция присваивания приведет к вызову оператора присваивания перемещением
88. 
89.     return 0;
90. }
```

Всё просто! Вместо выполнения глубокого копирования исходного объекта в неявный объект, мы просто перемещаем (воруем) ресурсы исходного объекта. Под этим подразумевается поверхностное копирование указателя на исходный объект в неявный (временный) объект, а затем присваивание исходному указателю значения null (точнее nullptr) и в конце удаление неявного объекта.

Результат выполнения программы:

```
Item acquired
Item destroyed
```

Уже гораздо лучше!

Ход выполнения этой программы точно такой же, как и предыдущей программы. Однако вместо вызова конструктора копирования и оператора присваивания копированием в этой программе вызывается конструктор перемещения и оператор присваивания перемещением. Рассмотрим детально:

- При создании объекта `mainItem` внутри generateItem() создается локальная переменная (объект) `item` и инициализируется динамически выделенным Item, вследствие чего мы получаем первую строку вывода — `Item acquired`.

- Затем `item` возвращается обратно в функцию main() по значению. Используя семантику перемещения, программа создает временный объект, в который перемещается `item`.
- При завершении выполнения функции generateItem() переменная `item` выходит из области видимости. Поскольку локальный `item` (который находится в функции generateItem()) больше не управляет указателем на себя (этот указатель был перемещен во временный объект), т.е. не владеет выделенными ресурсами, то ничего интересного здесь не происходит.
- В функции main() временный объект с помощью оператора присваивания перемещением перемещается в `mainItem`.
- Операция присваивания временного объект в `mainItem` завершается, и временный объект выходит из области видимости выражения и уничтожается. Однако, поскольку временный объект больше не управляет указателем (этот указатель был перемещен в `mainItem`), здесь тоже ничего интересного не происходит.
- В конце функции main() объект `mainItem` выходит из области видимости, и мы получаем вторую (и последнюю) строку вывода — `Item destroyed`.

В этой программе, вместо выполнения копирования нашего Item-а дважды (один раз для конструктора копирования и один раз для оператора присваивания копированием), мы передаем этот Item дважды. Такой расклад более эффективный, так как Item создается и уничтожается только один раз (вместо трех).

Когда вызываются конструктор перемещения и оператор присваивания перемещением?

Конструктор перемещения и оператор присваивания перемещением вызываются, когда аргументом для создания или присваивания является r-value. Чаще всего этим r-value будет литерал или временное значение (временный объект).

В большинстве случаев конструктор перемещения и оператор присваивания перемещением не предоставляются по умолчанию. Однако в тех редких случаях, когда они могут быть предоставлены по умолчанию, эти функции будут выполнять то же самое, что и конструктор копирования вместе с оператором присваивания копированием - копирование, а не перемещение.

Правило: Если вам нужен конструктор перемещения и оператор присваивания перемещением, которые выполняют перемещение (а не копирование), то вам их нужно предоставить (написать) самостоятельно.

Ключевое понимание семантики перемещения

Если мы создаем объект или выполняем присваивание, где аргументом является l-value, то единственное разумное, что мы можем сделать - это скопировать l-value. Мы не можем сказать, что изменять l-value безопасно, так как он может использоваться в программе позже. Если у нас есть выражение `a = b`, то нам бы очень не хотелось, чтобы `b` каким-либо образом был изменен.

Однако, если мы создаем объект или выполняем присваивание, где аргументом является r-value, то мы знаем, что r-value - это просто некоторый временный объект. Вместо того, чтобы копировать его (что может быть затратно), мы можем просто переместить его ресурсы (что не так затратно) в другой объект, который мы создаем или которому присваиваем текущий. Это безопасно, поскольку временный объект будет уничтожен в конце выражения в любом случае, поэтому мы можем быть уверены, что он никогда не будет повторно использован!

В C++11 через ссылки r-value мы можем изменять поведение функций в зависимости от того, чем является аргумент: r-value или l-value. А это, в свою очередь, позволяет нам принимать более разумные и эффективные решения о том, как должен работать наш код.

В примерах, приведенных выше, в конструкторе перемещения и перегрузке оператора присваивания мы присваивали для `x.m_ptr` значение nullptr. Это может показаться лишним - в конце концов, если x является временным r-value, то зачем нам беспокоиться о выполнении какой-либо «очистки», если параметр `x` все равно будет уничтожен?

Дело в том, что, когда `x` выходит из области видимости, вызывается деструктор для уничтожения `x.m_ptr`. Если в этот момент `x.m_ptr` все еще указывает на тот же объект, что и `m_ptr`, то `m_ptr` превратится в висячий указатель после уничтожения `x.m_ptr`. Когда объект, содержащий `m_ptr`, в конечном итоге будет использован (или уничтожен), то мы получим неопределенное поведение/результаты.

На следующем уроке мы рассмотрим ситуации, в которых параметром `x` является l-value. В таких случаях `x` не будет немедленно уничтожен, и его еще можно будет использовать некоторое время.

Использование семантики перемещения с l-values

В функции generateItem() класса Auto_ptr4 вышеприведенного примера, когда переменная `item` возвращается по значению, её ресурсы перемещаются, а не

копируются, даже если `item` - это l-value. В языке C++ есть правило, согласно которому автоматические объекты, возвращаемые функцией по значению, можно перемещать (а не копировать), даже если они являются l-values. Это имеет смысл, так как `item` все равно будет уничтожен в конце функции в любом случае!

Отключение копирования

В классе Auto_ptr4 из примера, приведенного выше, мы оставили конструктор копирования и оператор присваивания копированием в целях сравнения. Но в классах с поддержкой перемещения иногда желательно удалить конструктор копирования и оператор присваивания копированием, чтобы убедиться, что копирование объектов никогда не будет выполнено. В нашем случае с Auto_ptr мы не хотим копировать объекты, поэтому удалим всё лишнее:

```
1.  template<class T>
2.  class Auto_ptr5
3.  {
4.      T* m_ptr;
5.  public:
6.      Auto_ptr5(T* ptr = nullptr)
7.          :m_ptr(ptr)
8.      {
9.      }
10.
11.     ~Auto_ptr5()
12.     {
13.         delete m_ptr;
14.     }
15.
16.     // Конструктор копирования - запрещаем любое копирование!
17.     Auto_ptr5(const Auto_ptr5& x) = delete;
18.
19.     // Конструктор перемещения, который передает право собственности на x.m_ptr
    в m_ptr
20.     Auto_ptr5(Auto_ptr5&& x)
21.         : m_ptr(x.m_ptr)
22.     {
23.         x.m_ptr = nullptr;
24.     }
25.
26.     // Оператор присваивания копированием - запрещаем любое копирование!
27.     Auto_ptr5& operator=(const Auto_ptr5& x) = delete;
28.
29.     // Оператор присваивания перемещением, который передает право собственности
    на x.m_ptr в m_ptr
30.     Auto_ptr5& operator=(Auto_ptr5&& x)
31.     {
32.         // Проверка на самоприсваивание
33.         if (&x == this)
34.             return *this;
35.
36.         // Удаляем всё, что может хранить указатель до этого момента
37.         delete m_ptr;
38.
39.         // Передаем право собственности на x.m_ptr в m_ptr
```

```
        m_ptr = x.m_ptr;
        x.m_ptr = nullptr;

        return *this;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
    bool isNull() const { return m_ptr == nullptr; }
};
```

Если вы попытаетесь передать l-value в Auto_ptr5 по значению, то компилятор будет жаловаться, что конструктор копирования, необходимый для инициализации аргумента конструктора копирования, был удален. Это хорошо, поскольку мы должны передавать l-value в Auto_ptr5 по константной ссылке в любом случае!

Наконец, Auto_ptr5 — это отличный пример класса умного указателя. Больше того, Стандартная библиотека C++ имеет класс, очень похожий на этот (который вы должны использовать) — std::unique_ptr. Подробнее о std::unique_ptr мы поговорим на соответствующих уроках.

Семантика копирования vs. Семантика перемещения

Рассмотрим другой класс, который использует динамически выделенную память - простой шаблон класса DynamicArray. Этот класс имеет конструктор копирования и оператор присваивания копированием, которые выполняют глубокое копирование:

```
template <class T>
class DynamicArray
{
private:
    T* m_array;
    int m_length;

public:
    DynamicArray(int length)
        : m_array(new T[length]), m_length(length)
    {
    }

    ~DynamicArray()
    {
        delete[] m_array;
    }

    // Конструктор копирования
    DynamicArray(const DynamicArray &arr)
        : m_length(arr.m_length)
    {
        m_array = new T[m_length];
        for (int i = 0; i < m_length; ++i)
            m_array[i] = arr.m_array[i];
    }

```

```
    // Оператор присваивания копированием
    DynamicArray& operator=(const DynamicArray &arr)
    {
        if (&arr == this)
            return *this;

        delete[] m_array;

        m_length = arr.m_length;
        m_array = new T[m_length];

        for (int i = 0; i < m_length; ++i)
            m_array[i] = arr.m_array[i];

        return *this;
    }

    int getLength() const { return m_length; }
    T& operator[](int index) { return m_array[index]; }
    const T& operator[](int index) const { return m_array[index]; }

};
```

Теперь давайте попробуем использовать этот класс на практике. Например, давайте выделим массив (объект класса DynamicArray), который будет хранить миллион целочисленных значений. Чтобы проверить эффективность этого кода, мы будем использовать класс Timer, который разрабатывали на уроке №137. Классом Timer мы определим скорость выполнения нашего кода и покажем разницу в производительности между семантикой копирования и семантикой перемещения.

```
#include <iostream>
#include <chrono> // для функций из std::chrono

template <class T>
class DynamicArray
{
private:
    T* m_array;
    int m_length;

public:
    DynamicArray(int length)
        : m_array(new T[length]), m_length(length)
    {
    }

    ~DynamicArray()
    {
        delete[] m_array;
    }

    // Конструктор копирования
    DynamicArray(const DynamicArray &arr)
        : m_length(arr.m_length)
    {
        m_array = new T[m_length];
        for (int i = 0; i < m_length; ++i)
            m_array[i] = arr.m_array[i];
```

```
29.     }
30.
31.     // Оператор присваивания копированием
32.     DynamicArray& operator=(const DynamicArray &arr)
33.     {
34.         if (&arr == this)
35.             return *this;
36.
37.         delete[] m_array;
38.
39.         m_length = arr.m_length;
40.         m_array = new T[m_length];
41.
42.         for (int i = 0; i < m_length; ++i)
43.             m_array[i] = arr.m_array[i];
44.
45.         return *this;
46.     }
47.
48.     int getLength() const { return m_length; }
49.     T& operator[](int index) { return m_array[index]; }
50.     const T& operator[](int index) const { return m_array[index]; }
51.
52. };
53.
54. class Timer
55. {
56. private:
57.     // Используем псевдонимы типов для удобного доступа к вложенным типам
58.     using clock_t = std::chrono::high_resolution_clock;
59.     using second_t = std::chrono::duration<double, std::ratio<1> >;
60.
61.     std::chrono::time_point<clock_t> m_beg;
62.
63. public:
64.     Timer() : m_beg(clock_t::now())
65.     {
66.     }
67.
68.     void reset()
69.     {
70.         m_beg = clock_t::now();
71.     }
72.
73.     double elapsed() const
74.     {
75.         return std::chrono::duration_cast<second_t>(clock_t::now() -
    m_beg).count();
76.     }
77. };
78.
79. // Возвращаем копию arr со значениями, умноженными на 2
80. DynamicArray<int> cloneArrayAndDouble(const DynamicArray<int> &arr)
81. {
82.     DynamicArray<int> dbl(arr.getLength());
83.     for (int i = 0; i < arr.getLength(); ++i)
84.         dbl[i] = arr[i] * 2;
85.
86.     return dbl;
87. }
88.
89. int main()
```

```
{
    Timer t;

    DynamicArray<int> arr(1000000);

    for (int i = 0; i < arr.getLength(); i++)
        arr[i] = i;

    arr = cloneArrayAndDouble(arr);

        std::cout << t.elapsed();
    }
```

Результат выполнения программы на компьютере автора в режиме Release:

```
0.0225438
```

> ***Примечание***: Результат измеряется в секундах.

Теперь давайте запустим эту же программу, заменив конструктор копирования и оператор присваивания копированием на конструктор перемещения и оператор присваивания перемещением:

```
#include <iostream>
#include <chrono> // для функций из std::chrono

template <class T>
class DynamicArray
{
private:
    T* m_array;
    int m_length;

public:
    DynamicArray(int length)
        : m_array(new T[length]), m_length(length)
    {
    }

    ~DynamicArray()
    {
        delete[] m_array;
    }

    // Конструктор копирования
    DynamicArray(const DynamicArray &arr) = delete;

    // Оператор присваивания копированием
    DynamicArray& operator=(const DynamicArray &arr) = delete;

    // Конструктор перемещения
    DynamicArray(DynamicArray &&arr)
        : m_length(arr.m_length), m_array(arr.m_array)
    {
        arr.m_length = 0;
        arr.m_array = nullptr;
    }

```

```
36.     // Оператор присваивания перемещением
37.     DynamicArray& operator=(DynamicArray &&arr)
38.     {
39.         if (&arr == this)
40.             return *this;
41.
42.         delete[] m_array;
43.
44.         m_length = arr.m_length;
45.         m_array = arr.m_array;
46.         arr.m_length = 0;
47.         arr.m_array = nullptr;
48.
49.         return *this;
50.     }
51.
52.     int getLength() const { return m_length; }
53.     T& operator[](int index) { return m_array[index]; }
54.     const T& operator[](int index) const { return m_array[index]; }
55.
56. };
57.
58. class Timer
59. {
60. private:
61.     // Используем псевдонимы типов для удобного доступа к вложенным типам
62.     using clock_t = std::chrono::high_resolution_clock;
63.     using second_t = std::chrono::duration<double, std::ratio<1> >;
64.
65.     std::chrono::time_point<clock_t> m_beg;
66.
67. public:
68.     Timer() : m_beg(clock_t::now())
69.     {
70.     }
71.
72.     void reset()
73.     {
74.         m_beg = clock_t::now();
75.     }
76.
77.     double elapsed() const
78.     {
79.         return std::chrono::duration_cast<second_t>(clock_t::now() -
    m_beg).count();
80.     }
81. };
82.
83. // Возвращаем копию arr со значениями, умноженными на 2
84. DynamicArray<int> cloneArrayAndDouble(const DynamicArray<int> &arr)
85. {
86.     DynamicArray<int> dbl(arr.getLength());
87.     for (int i = 0; i < arr.getLength(); ++i)
88.         dbl[i] = arr[i] * 2;
89.
90.     return dbl;
91. }
92.
93. int main()
94. {
95.     Timer t;
96.
```

```
97.     DynamicArray<int> arr(1000000);
98.
99.     for (int i = 0; i < arr.getLength(); i++)
100.            arr[i] = i;
101.
102.        arr = cloneArrayAndDouble(arr);
103.
104.        std::cout << t.elapsed();
105.    }
```

Результат выполнения программы на компьютере автора в режиме Release:

```
0.0131518
```

Сравниваем время выполнения двух программ: `0.0131518 / 0.0225438 = 58.3%`. Версия с использованием семантики перемещения была почти на 42% быстрее версии с использованием семантики копирования!

Урок №200. Функция std::move()

Как только вы начнете регулярно использовать семантику перемещения, вы поймете насколько полезной она может быть. Очень часто объекты, с которыми вам придется работать, будут не r-values, а l-values. Например, рассмотрим следующий шаблон функции swap():

```
#include <iostream>
#include <string>

template<class T>
void swap(T& x, T& y)
{
  T tmp { x }; // вызывает конструктор копирования
  x = y; // вызывает оператор присваивания копированием
  y = tmp; // вызывает оператор присваивания копированием
}

int main()
{
    std::string x{ "Anton" };
    std::string y{ "Max" };

    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';

    swap(x, y);

    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';

    return 0;
}
```

Принимая два объекта типа `T` (в данном случае std::string) функция swap() меняет местами их значения, делая при этом три копии.

Следовательно, результат выполнения программы:

```
x: Anton
y: Max
x: Max
y: Anton
```

А как мы уже узнали на предыдущем уроке, копирование — это не очень эффективно. А поскольку мы выполняем копирование трижды, то это еще и сравнительно медленно. Мы можем этого избежать. Наша цель — поменять местами значения `x` и `y`. А для этого мы можем использовать перемещение вместо копирования, сделав наш код более производительным!

Но как? Проблема состоит в том, что параметры `x` и `y` являются ссылками l-value, а не ссылками r-value, поэтому у нас нет способа вызвать конструктор перемещения или оператор присваивания перемещением вместо конструктора копирования и оператора присваивания копированием. По умолчанию у нас используется семантика копирования. Что делать?

Функция std::move()

Функция std::move() - это стандартная библиотечная функция, которая конвертирует передаваемый аргумент в r-value. Мы можем передать l-value в функцию std::move(), и std::move() вернет нам ссылку r-value. Для работы с std::move() нужно подключить заголовочный файл utility.

Вот вышеприведенная программа, но уже с функцией swap(), которая использует std::move() для преобразования l-values в r-values, чтобы мы имели возможность использовать семантику перемещения вместо семантики копирования:

```
1. #include <iostream>
2. #include <string>
3. #include <utility>
4.
5. template<class T>
6. void swap(T& x, T& y)
7. {
8.   T tmp { std::move(x) }; // вызывает конструктор перемещения
9.   x = std::move(y); // вызывает оператор присваивания перемещением
10.  y = std::move(tmp); // вызывает оператор присваивания перемещением
11.}
12.
13.int main()
14.{
15.    std::string x{ "Anton" };
16.    std::string y{ "Max" };
17.
18.    std::cout << "x: " << x << '\n';
19.    std::cout << "y: " << y << '\n';
20.
21.    swap(x, y);
22.
23.    std::cout << "x: " << x << '\n';
24.    std::cout << "y: " << y << '\n';
25.
26.    return 0;
27.}
```

Результат выполнения один и тот же:

```
x: Anton
y: Max
x: Max
y: Anton
```

Но эта версия программы гораздо эффективнее. При инициализации `tmp`, вместо создания копии `x`, мы используем std::move() для конвертации переменной `x`, которая является l-value, в r-value. А поскольку параметром становится r-value, то с помощью семантики перемещения `x` перемещается в `tmp`.

Затем, спустя несколько дополнительных перестановок, значение переменной `x` перемещается в переменную `y`, а значение `y` перемещается в переменную `x`.

Еще один пример

Мы также можем использовать std::move() для заполнения контейнерных классов (таких как std::vector) значениями l-values.

В следующей программе мы сначала добавляем элемент в вектор, используя семантику копирования, а затем добавляем элемент в вектор, используя семантику перемещения:

```
#include <iostream>
#include <string>
#include <utility>
#include <vector>

int main()
{
    std::vector<std::string> v;
    std::string str = "Bye";

    std::cout << "Copying str\n";
    v.push_back(str); // вызывает версию l-value метода push_back(), которая копирует str в элемент массива

    std::cout << "str: " << str << '\n';
    std::cout << "vector: " << v[0] << '\n';

    std::cout << "\nMoving str\n";

    v.push_back(std::move(str)); // вызывает версию r-value метода push_back(), которая перемещает str в элемент массива

    std::cout << "str: " << str << '\n';
    std::cout << "vector: " << v[0] << ' ' << v[1] << '\n';

    return 0;
}
```

Результат выполнения программы:

```
Copying str
str: Bye
vector: Bye
```

```
Moving str
str:
vector: Bye Bye
```

В первом случае мы передаем l-value в push_back(), поэтому используется семантика копирования для добавления элемента в вектор. По этой причине переменная `str` остается с прежним значением.

Во втором случае мы передаем r-value (фактически l-value, которое конвертируется в r-value через std::move()) в push_back(), поэтому используется семантика перемещения для добавления элемента в вектор. Это более эффективно, так как элемент вектора может украсть значение переменной std::string, вместо его копирования. По этой же причине `str` лишается своего значения.

На этом этапе стоит сообщить, что std::move() как бы подсказывает компилятору, что нам больше не нужен этот объект (по крайней мере, в его текущем состоянии). Следовательно, вы не должны использовать std::move() с любым «постоянным» объектом, который вы не хотите изменять, и вам не следует ожидать, что объекты, которые используются с std::move(), останутся прежними.

Функции перемещения

Как мы уже говорили на предыдущем уроке, вы должны оставлять объекты, ресурсы которых вы перемещаете, в четко определенном состоянии. В идеале это должно быть *«нулевое состояние»* (*null/nullptr*).

Теперь мы можем поговорить о том, почему при использовании функции std::move(), объект, ресурсы которого перемещаются, может не быть временным. Дело в том, что пользователь может захотеть повторно использовать этот же (теперь пустой) объект или протестировать его каким-либо образом.

В примере, приведенном выше, строка `str` остается пустой после выполнения перемещения (что всегда делает std::string после успешного перемещения). Таким образом, мы можем повторно её использовать, если, конечно, захотим, или проигнорировать, если она нам больше не нужна.

Чем еще полезна функция std::move()?

Функция std::move() также может быть полезна при сортировке элементов массива. Многие алгоритмы сортировки (такие как «метод выбора» и «сортировка пузырьком») работают путем замены целых пар элементов. На предыдущих уроках нам приходилось использовать семантику копирования для выполнения таких

замен. Теперь же мы можем использовать семантику перемещения, которая эффективнее.

Функция std::move() также может быть полезна при перемещении содержимого из одного умного указателя в другой.

Заключение

Функция std::move() может использоваться всякий раз, когда нужно обрабатывать l-value как r-value с целью использования семантики перемещения вместо семантики копирования.

Урок №201. Умный указатель std::unique_ptr

На предыдущих уроках мы говорили о том, как использование указателей может привести к ошибкам и утечкам памяти в некоторых ситуациях. Например, при досрочном завершении функции или при генерации исключения, когда указатель может не быть удален должным образом:

```
1. #include <iostream>
2.
3. void someFunction()
4. {
5.     Item *ptr = new Item;
6.
7.     int a;
8.     std::cout << "Enter an integer: ";
9.     std::cin >> a;
10.
11.    if (a == 0)
12.        throw 0; // в случае генерации исключения функция завершит свое выполнение досрочно, и ptr не будет удален!
13.
14.    // Делаем что-либо с ptr здесь
15.
16.    delete ptr;
17. }
```

Умный указатель - это класс, который управляет динамически выделенной памятью/ресурсом/объектом. Главная фишка умных указателей заключается в управлении и обеспечении корректного удаления динамически выделенного ресурса в соответствующее время (обычно, когда умный указатель выходит из области видимости).

Из-за этого умные указатели никогда нельзя выделять динамически (в противном случае, существует риск того, что умный указатель будет неправильно удален, это означает, что принадлежащий ему ресурс также не будет удален, и произойдет утечка памяти). Всегда выделяя умные указатели статическим образом (как локальные переменные), вы получаете гарантию, что умный указатель корректно выйдет из области видимости и удалит хранимый объект.

Стандартная библиотека в C++11 имеет **4 класса умных указателей**:

- std::auto_ptr (который не следует использовать - он удален в C++17);
- std::unique_ptr;
- std::shared_ptr;
- std::weak_ptr.

Умный указатель std::unique_ptr, по сути, является наиболее часто используемым классом умного указателя, поэтому сначала рассмотрим именно его. На следующих уроках рассмотрим умные указатели std::shared_ptr и std::weak_ptr.

Умный указатель std::unique_ptr

Умный указатель std::unique_ptr является заменой std::auto_ptr в C++11. Вы должны использовать именно его для управления любым динамически выделенным объектом/ресурсом, но с условием, что std::unique_ptr полностью владеет переданным ему объектом, а не делится «владением» еще с другими классами. Умный указатель std::unique_ptr находится в заголовочном файле memory.

Рассмотрим простой пример использования std::unique_ptr:

```
1. #include <iostream>
2. #include <memory> // для std::unique_ptr
3.
4. class Item
5. {
6. public:
7.     Item() { std::cout << "Item acquired\n"; }
8.     ~Item() { std::cout << "Item destroyed\n"; }
9. };
10.
11.int main()
12.{
13.    // Выделяем объект класса Item и передаем право собственности на него
   std::unique_ptr
14.    std::unique_ptr<Item> item(new Item);
15.
16.    return 0;
17.} // item выходит из области видимости здесь, соответственно, Item уничтожается
    также здесь
```

Когда std::unique_ptr выходит из области видимости, он удаляет Item, которым владеет.

В отличие от std::auto_ptr, std::unique_ptr корректно реализовывает семантику перемещения:

```
1. #include <iostream>
2. #include <memory> // для std::unique_ptr
3.
4. class Item
5. {
6. public:
7.     Item() { std::cout << "Item acquired\n"; }
8.     ~Item() { std::cout << "Item destroyed\n"; }
9. };
10.
11.int main()
```

```
{
    std::unique_ptr<Item> item1(new Item); // выделение Item
    std::unique_ptr<Item> item2; // присваивается значение nullptr

    std::cout << "item1 is " << (static_cast<bool>(item1) ? "not null\n" : "null\n");
    std::cout << "item2 is " << (static_cast<bool>(item2) ? "not null\n" : "null\n");

    // item2 = item1; // не скомпилируется: семантика копирования отключена
    item2 = std::move(item1); // item2 теперь владеет item1, а для item1 присваивается значение null

    std::cout << "Ownership transferred\n";

    std::cout << "item1 is " << (static_cast<bool>(item1) ? "not null\n" : "null\n");
    std::cout << "item2 is " << (static_cast<bool>(item2) ? "not null\n" : "null\n");

    return 0;
} // Item уничтожается здесь, когда item2 выходит из области видимости
```

Результат выполнения программы:

```
Item acquired
item1 is not null
item2 is null
Ownership transferred
item1 is null
item2 is not null
Item destroyed
```

Поскольку std::unique_ptr разработан с учетом семантики перемещения, то семантика копирования по умолчанию отключена. Если вы хотите передать содержимое, управляемое std::unique_ptr, то вы должны использовать семантику перемещения. В программе, приведенной выше, мы передаем содержимое std::unique_ptr с помощью функции std::move() (которая конвертирует `item1` в r-value, являющееся триггером для выполнения семантики перемещения вместо семантики копирования).

Доступ к объекту, который хранит умный указатель

Умный указатель std::unique_ptr имеет **перегруженные операторы** `*` и `->`, которые используются для доступа к хранимым объектам. Оператор `*` возвращает ссылку на управляемый ресурс, а оператор `->` возвращает указатель.

Умный указатель std::unique_ptr не всегда может управлять объектом: либо потому, что объект был создан пустым (с использованием конструктора по умолчанию, или в объект передан в качестве параметра nullptr), либо потому, что ресурс, которым

он управлял, был перемещен в другой std::unique_ptr. Поэтому, прежде чем использовать какой-либо из этих операторов, вы должны проверить, действительно ли std::unique_ptr управляет ресурсом. К счастью, это легко сделать: std::unique_ptr имеет неявное преобразование в тип bool, возвращая `true`, если std::unique_ptr владеет ресурсом. Например:

```
#include <iostream>
#include <memory> // для std::unique_ptr

class Item
{
public:
    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
    friend std::ostream& operator<<(std::ostream& out, const Item &item)
    {
        out << "I am an Item!\n";
        return out;
    }
};

int main()
{
    std::unique_ptr<Item> item(new Item);

    if (item) // используем неявное преобразование item в тип bool, чтобы убедиться, что item владеет Item-ом
        std::cout << *item; // выводим Item, которым владеет item

    return 0;
}
```

Результат:

```
Item acquired
I am an Item!
Item destroyed
```

В программе, приведенной выше, мы используем оператор `*` для доступа к Item, которым владеет объект `item` класса std::unique_ptr, который затем мы выводим с помощью std::cout.

Умный указатель std::unique_ptr и динамические массивы

В отличие от std::auto_ptr, std::unique_ptr достаточно умен, чтобы знать, когда использовать единичный оператор delete, а когда форму оператора delete для массива, поэтому std::unique_ptr можно использовать как с единичными объектами, так и с динамическими массивами.

Однако использование std::vector почти всегда является лучшим выбором, чем использование std::unique_ptr с динамическим массивом.

Правило: Используйте std::vector вместо использования умного указателя, который владеет динамическим массивом.

Функция std::make_unique()

В C++14 добавили новую функцию - **std::make_unique()**. Это шаблон функции, который создает объект типа шаблона и инициализирует его аргументами, переданными в функцию. Например:

```
#include <iostream>
#include <memory> // для std::unique_ptr и std::make_unique

class Fraction
{
private:
    int m_numerator = 0;
    int m_denominator = 1;

public:
    Fraction(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
    {
        out << f1.m_numerator << "/" << f1.m_denominator;
        return out;
    }
};


int main()
{
    // Создаем объект с динамически выделенным Fraction с numerator = 7 и denominator = 9
    std::unique_ptr<Fraction> f1 = std::make_unique<Fraction>(7, 9);
    std::cout << *f1 << '\n';

    // Создаем объект с динамически выделенным массивом Fraction длиной 5.
    // Используем автоматическое определение типа данных с помощью ключевого слова auto
    auto f2 = std::make_unique<Fraction[]>(5);
    std::cout << f2[0] << '\n';

    return 0;
}
```

Результат выполнения программы:

```
7/9
0/1
```

Использование функции std::make_unique() является необязательным, но рекомендуется вместо использования умного указателя std::unique_ptr. Дело в простоте. Кроме того, std::make_unique() решает проблему безопасности использования исключений, которая может возникнуть в результате неопределенного порядка обработки аргументов функции (так как язык C++ явно не указывает этот порядок).

Правило: Используйте функцию std::make_unique() вместо создания умного указателя std::unique_ptr и использования оператора new.

Безопасность использования исключений

Вот пример для тех, кто не понял, что это за проблема с безопасностью использования исключений, которая прозвучала выше:

```
1. some_function(std::unique_ptr<T>(new T), function_that_can_throw_exception());
```

Здесь компилятору предоставляется большая гибкость при обработке вызова функции. Он может сначала выделить новый `T`, затем вызвать function_that_can_throw_exception(), а затем уже создать std::unique_ptr, который управляет динамически выделенным `T`. Если функция function_that_can_throw_exception() выбросит исключение, то выделенный `T` не будет корректно удален, поскольку умный указатель, который должен выполнить его удаление, не успеет создаться. Это приведет к утечке памяти.

Функция std::make_unique() лишена этой проблемы, поскольку выделение объекта `T` и создание std::unique_ptr происходят внутри функции std::make_unique(), где порядок обработки аргументов четко определен.

Возврат умного указателя std::unique_ptr из функции

Умный указатель std::unique_ptr можно возвращать из функции по значению:

```
1. std::unique_ptr<Item> createItem()
2. {
3.     return std::make_unique<Item>();
4. }
5.
6. int main()
7. {
```

```
    std::unique_ptr<Item> ptr = createItem();

    // Делаем что-либо

    return 0;
}
```

Здесь createItem() возвращает std::unique_ptr по значению. Если возвращаемое значение не присваивается какому-либо объекту, то оно выходит из области видимости, и Item удаляется. Если значение присваивается объекту (как показано в функции main()), то с помощью семантики перемещения Item перемещается из возвращаемого значения в нужный объект (в данном случае в `ptr`). Это делает возврат ресурсов с помощью std::unique_ptr намного безопаснее, чем возврат «необработанных» указателей!

В общем, вы не должны возвращать std::unique_ptr по адресу (вообще) или по ссылке (если у вас нет на это веских причин).

Передача умного указателя std::unique_ptr в функцию

Если вы хотите, чтобы функция стала владельцем содержимого умного указателя, то передавать std::unique_ptr в функцию нужно по значению. Обратите внимание, поскольку семантика копирования отключена, то вам придется использовать std::move() для фактической передачи ресурса в функцию:

```
#include <iostream>
#include <memory> // для std::unique_ptr

class Item
{
public:
    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
    friend std::ostream& operator<<(std::ostream& out, const Item &item)
    {
        out << "I am an Item!\n";
        return out;
    }
};

void takeOwnership(std::unique_ptr<Item> item)
{
     if (item)
          std::cout << *item;
} // Item уничтожается здесь

int main()
{
    auto ptr = std::make_unique<Item>();

//    takeOwnership(ptr); // это не скомпилируется. Мы должны использовать семантику перемещения
    takeOwnership(std::move(ptr)); // используем семантику перемещения
```

```
28.
29.     std::cout << "Ending program\n";
30.
31.     return 0;
32. }
```

Результат выполнения программы:

```
Item acquired
I am an Item!
Item destroyed
Ending program
```

Обратите внимание, в данном случае право собственности на Item было передано в takeOwnership(), поэтому Item уничтожается в конце takeOwnership(), а не в конце main().

Однако в большинстве случаев вам не нужно будет, чтобы функция владела ресурсом. Хотя вы можете передать std::unique_ptr по ссылке (что позволит функции использовать объект без передачи ей права собственности на этот объект), вы должны делать это только тогда, когда caller может изменить передаваемый объект.

Вместо этого лучше передавать сам объект по указателю или по ссылке (в зависимости от того, является ли null допустимым аргументом). Это позволит функции оставаться в стороне от управления объектом. Чтобы получить необработанный указатель на объект из std::unique_ptr, вы можете использовать **метод get()**:

```
1. #include <iostream>
2. #include <memory> // для std::unique_ptr
3.
4. class Item
5. {
6. public:
7.     Item() { std::cout << "Item acquired\n"; }
8.     ~Item() { std::cout << "Item destroyed\n"; }
9.
10.    friend std::ostream& operator<<(std::ostream& out, const Item &item)
11.    {
12.        out << "I am an Item!\n";
13.        return out;
14.    }
15. };
16.
17. // Эта функция использует только Item, поэтому мы передаем указатель на Item, а
    не ссылку на весь std::unique_ptr<Item>
18. void useItem(Item *item)
19. {
20.     if (item)
21.         std::cout << *item;
```

```
}

int main()
{
    auto ptr = std::make_unique<Item>();

    useItem(ptr.get()); // примечание: Метод get() используется для получения указателя на Item

    std::cout << "Ending program\n";

    return 0;
} // Item уничтожается здесь
```

Результат выполнения программы:

```
Item acquired
I am an Item!
Ending program
Item destroyed
```

Умный указатель std::unique_ptr и классы

Конечно, вы можете использовать std::unique_ptr в качестве члена композиции вашего класса. Таким образом, вам не нужно будет беспокоиться о том, удалит ли деструктор вашего класса ресурс std::unique_ptr, так как std::unique_ptr будет автоматически уничтожен при уничтожении объекта класса. Тем не менее, если объект вашего класса выделяется динамически, то сам ресурс std::unique_ptr подвергается риску неправильного удаления, и в таком случае даже умный указатель не поможет.

Неправильное использование умного указателя std::unique_ptr

Существует два способа неправильного использования std::unique_ptr, оба из которых легко избежать. Во-первых, не позволяйте нескольким классам «владеть» одним и тем же ресурсом. Например:

```
Item *item = new Item;
std::unique_ptr<Item> item1(item);
std::unique_ptr<Item> item2(item);
```

Хотя это синтаксически допустимо, конечным результатом будет то, что и `item1`, и `item2` попытаются удалить Item, что приведет к неопределенному поведению/результатам.

Во-вторых, не удаляйте выделенный ресурс вручную из-под std::unique_ptr:

```
Item *item = new Item;
std::unique_ptr<Item> item1(item);
```

```
3. delete item;
```

Если вы это сделаете, std::unique_ptr попытается удалить уже удаленный ресурс, что опять приведет к неопределенному поведению/результатам.

Обратите внимание, функция std::make_unique() предотвращает непреднамеренное возникновение обеих ситуаций, приведенных выше.

Тест

Задание №1

Если в вашем классе есть умный указатель в качестве члена вашего класса, то почему вы должны стараться избегать динамического выделения объектов этого класса?

Задание №2

Измените следующую программу, заменив обычный указатель на умный указатель std::unique_ptr, где это необходимо:

```
#include <iostream>

class Fraction
{
private:
    int m_numerator = 0;
    int m_denominator = 1;

public:
    Fraction(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
    {
        out << f1.m_numerator << "/" << f1.m_denominator;
        return out;
    }
};

void printFraction(const Fraction* const ptr)
{
    if (ptr)
        std::cout << *ptr;
}

int main()
{
    Fraction *ptr = new Fraction(7, 9);

    printFraction(ptr);
```

```
33.
34.     delete ptr;
35.
36.     return 0;
37. }
```

Урок №202. Умный указатель std::shared_ptr

В отличие от std::unique_ptr, который предназначен для единоличного владения и управления переданным ему ресурсом/объектом, **std::shared_ptr** предназначен для случаев, когда несколько умных указателей совместно владеют одним динамически выделенным ресурсом.

Умный указатель std::shared_ptr

Вы можете иметь несколько умных указателей std::shared_ptr, указывающих на один и тот же ресурс. Умный указатель std::shared_ptr отслеживает количество владельцев у каждого полученного ресурса. До тех пор, пока хотя бы один std::shared_ptr владеет ресурсом, этот ресурс не будет уничтожен, даже если удалить все остальные std::shared_ptr (которые также владеют этим ресурсом). Как только последний std::shared_ptr, владеющий ресурсом, выйдет из области видимости (или ему передадут другой ресурс для управления), тогда ресурс будет уничтожен.

Как и std::unique_ptr, std::shared_ptr находится в заголовочном файле memory.

```
#include <iostream>
#include <memory> // для std::shared_ptr

class Item
{
public:
    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
};

int main()
{
    // Выделяем Item и передаем его в std::shared_ptr
    Item *item = new Item;
    std::shared_ptr<Item> ptr1(item);
    {
        std::shared_ptr<Item> ptr2(ptr1); // используем оператор присваивания копированием для создания второго std::shared_ptr из ptr1, указывающего на тот же Item

        std::cout << "Killing one shared pointer\n";
    } // ptr2 выходит из области видимости здесь, но больше ничего не происходит

    std::cout << "Killing another shared pointer\n";

    return 0;
} // ptr1 выходит из области видимости здесь, и выделенный Item уничтожается также здесь
```

Результат выполнения программы:

```
Item acquired
Killing one shared pointer
Killing another shared pointer
Item destroyed
```

Здесь мы динамически выделяем объект класса Item и передаем его указателю std::shared_ptr с именем `ptr1`. Внутри вложенного блока функции main() мы используем семантику копирования (которая разрешена в std::shared_ptr, так как одним ресурсом могут владеть сразу несколько умных указателей) для создания второго std::shared_ptr (`ptr2`), который указывает на тот же выделенный Item. Когда `ptr2` выходит из области видимости, Item не уничтожается, так как `ptr1` по-прежнему указывает на него. Когда `ptr1` выходит из области видимости, то он замечает, что больше нет std::shared_ptr, которые бы указывали на Item, и удаляет Item.

Обратите внимание, мы создали второй умный указатель из первого умного указателя (используя оператор присваивания копированием). Это важно.

Рассмотрим следующую программу:

```
#include <iostream>
#include <memory> // для std::shared_ptr

class Item
{
public:
    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
};

int main()
{
    Item *item = new Item;
    std::shared_ptr<Item> ptr1(item);
    {
        std::shared_ptr<Item> ptr2(item); // создаем ptr2 напрямую из item (вместо ptr1)

        std::cout << "Killing one shared pointer\n";
    } // ptr2 выходит из области видимости здесь и выделенный Item уничтожается также здесь

    std::cout << "Killing another shared pointer\n";

    return 0;
} // ptr1 выходит из области видимости здесь, и уже удаленный Item опять уничтожается здесь
```

Результат выполнения программы:

```
Item acquired
Killing one shared pointer
Item destroyed
Killing another shared pointer
Item destroyed
```

И затем «Бум!» - генерация исключения (по крайней мере, на компьютере автора).

Разница здесь в том, что мы создали два отдельных, независимых друг от друга умных указателя std::shared_ptr. Хотя они оба указывают на один и тот же Item, они не знают о существовании друг друга. Когда `ptr2` выходит из области видимости, он думает, что является единственным владельцем Item-а, поэтому уничтожает его. Когда позже `ptr1` выходит из области видимости, он думает так же и пытается снова удалить (уже удаленный) Item. Бум!

К счастью, этого легко избежать, используя семантику копирования для создания нескольких умных указателей, указывающих на один динамически выделенный ресурс.

Правило: Всегда выполняйте копирование существующего std::shared_ptr, если вам нужно более одного std::shared_ptr, указывающего на один и тот же динамически выделенный ресурс.

Функция std::make_shared()

Как функцию std::make_unique() можно использовать для создания std::unique_ptr в C++14, так и **функцию std::make_shared()** можно (и нужно) использовать для создания std::shared_ptr. Функцию std::make_shared() добавили в C++11.

Давайте перепишем первую программу из данного урока, добавив функцию std::make_shared():

```
1. #include <iostream>
2. #include <memory> // для std::shared_ptr
3.
4. class Item
5. {
6. public:
7.     Item() { std::cout << "Item acquired\n"; }
8.     ~Item() { std::cout << "Item destroyed\n"; }
9. };
10.
11. int main()
12. {
13.     // Выделяем Item и передаем его в std::shared_ptr
```

```
14.     auto ptr1 = std::make_shared<Item>();
15.     {
16.         auto ptr2 = ptr1; // создаем ptr2 из ptr1, используя семантику
    копирования
17. 
18.         std::cout << "Killing one shared pointer\n";
19.     } // ptr2 выходит из области видимости здесь, но ничего больше не
    происходит
20. 
21.     std::cout << "Killing another shared pointer\n";
22. 
23.     return 0;
24. } // ptr1 выходит из области видимости здесь, и выделенный Item также
    уничтожается здесь
```

Причины использования функции std::make_shared() такие же, как и причины использования функции std::make_unique(): проще, безопаснее и производительнее за счет того, что std::shared_ptr отслеживает, сколько умных указателей владеют ресурсом.

Детали реализации умного указателя std::shared_ptr

В отличие от std::unique_ptr, который использует внутри ("под капотом") один указатель, std::shared_ptr использует внутри два указателя. Один указывает на передаваемый ресурс, а второй — на **«блок управления»** - динамически выделенный объект, который отслеживает кучу разных вещей, включая и то, сколько умных указателей std::shared_ptr одновременно указывают на каждый полученный ресурс. При создании std::shared_ptr с помощью конструктора std::shared_ptr, память для полученного ресурса и блока управления (который также создает конструктор) выделяется отдельно. Однако в std::make_shared() это оптимизировано в единое выделение памяти, что, соответственно, повышает производительность.

Это также объясняет то, почему независимое создание двух std::shared_ptr, указывающих на один и тот же ресурс, приводит к проблемам. Каждый std::shared_ptr имеет один указатель, указывающий на полученный ресурс. Однако каждый std::shared_ptr еще и независимо выделяет свой собственный блок управления, который сообщает указателю, что он является единственным «владельцем» полученного ресурса (даже если это не так). Таким образом, когда данный std::shared_ptr выходит из области видимости, он уничтожает ресурс, которым владеет, не осознавая того, что могут быть еще другие умные указатели std::shared_ptr, которые владеют этим же ресурсом.

Однако, когда std::shared_ptr клонируется с использованием семантики копирования, данные в блоке управления соответствующим образом обновляются

и говорят о том, что появился еще один std::shared_ptr, указывающий на полученный ресурс.

Создание std::shared_ptr из std::unique_ptr

Умный указатель std::unique_ptr может быть конвертирован в умный указатель std::shared_ptr через специальный конструктор std::shared_ptr, который принимает std::unique_ptr в качестве r-value. Таким образом, содержимое std::unique_ptr перемещается в std::shared_ptr.

Однако std::shared_ptr нельзя безопасно конвертировать в std::unique_ptr. Поэтому, если вы хотите создать функцию, которая будет возвращать умный указатель, вам лучше возвращать std::unique_ptr и затем присваивать его std::shared_ptr, когда это будет уместно.

Опасности использования умного указателя std::shared_ptr

У умного указателя std::shared_ptr есть некоторые из проблем, которые имеет std::unique_ptr. Если std::shared_ptr не уничтожается должным образом (либо потому, что он был динамически выделен и не удален должным образом, либо потому, что он был частью объекта, который был динамически выделен и не удален), тогда ресурс, которым управляет std::shared_ptr, тоже не будет удален. С std::unique_ptr вам нужно беспокоиться об удалении лишь одного указателя. А с std::shared_ptr вам придется беспокоиться об удалении всех указателей, владеющих ресурсом. Если какой-либо из std::shared_ptr, владеющий ресурсом, не будет должным образом уничтожен, то и сам ресурс также не будет уничтожен.

Умный указатель std::shared_ptr и массивы

В C++14 и в более ранних версиях C++ std::shared_ptr не имеет поддержки управления динамическими массивами и, соответственно, не должен использоваться с ними. Начиная с C++17, в std::shared_ptr добавили поддержку динамических массивов. Однако в C++17 «забыли» добавить эту поддержку в std::make_shared(), поэтому данную функцию не следует использовать для создания std::shared_ptr, указывающего на динамические массивы. Возможно, эта проблема будет решена в C++20.

Заключение

Умный указатель std::shared_ptr дает возможность сразу нескольким умным указателям управлять одним динамически выделенным ресурсом. Ресурс

уничтожается лишь в том случае, когда уничтожены все std::shared_ptr, указывающие на него.

Урок №203. Умный указатель std::weak_ptr

На предыдущем уроке мы рассматривали умный указатель std::shared_ptr и то, как с его помощью сразу несколько умных указателей могут владеть одним динамически выделенным ресурсом. Однако, иногда это может быть проблематично. Например, рассмотрим случай, когда два умных указателя типа std::shared_ptr владеют двумя разными объектами и «пересекаются» между собой:

```
#include <iostream>
#include <memory> // для std::shared_ptr
#include <string>

class Human
{
    std::string m_name;
    std::shared_ptr<Human> m_partner; // изначально пустой

public:

    Human(const std::string &name): m_name(name)
    {
        std::cout << m_name << " created\n";
    }
    ~Human()
    {
        std::cout << m_name << " destroyed\n";
    }

    friend bool partnerUp(std::shared_ptr<Human> &h1, std::shared_ptr<Human> &h2)
    {
        if (!h1 || !h2)
            return false;

        h1->m_partner = h2;
        h2->m_partner = h1;

        std::cout << h1->m_name << " is now partnered with " << h2->m_name << "\n";

        return true;
    }
};

int main()
{
    auto anton = std::make_shared<Human>("Anton"); // создание умного указателя с объектом Anton класса Human
    auto ivan = std::make_shared<Human>("Ivan"); // создание умного указателя с объектом Ivan класса Human

    partnerUp(anton, ivan); // Anton указывает на Ivan-a, а Ivan указывает на Anton-a

    return 0;
}
```

Здесь мы динамически выделяем два объекта (`Anton` и `Ivan`) класса Human и, используя std::make_shared, передаем их в два отдельно созданных умных указателя типа std::shared_ptr. Затем «связываем» их с помощью дружественной функции partnerUp().

Результат выполнения программы:

```
Anton created
Ivan created
Anton is now partnered with Ivan
```

И это всё? Никаких уничтожений? Почему? Сейчас разберемся.

После вызова функции partnerUp() у нас образуется 4 умных указателя типа std::shared_ptr:

- Два умных указателя указывают на объект `Ivan`: `ivan` (из функции main()) и `m_partner` (из класса Human) объекта `Anton`.
- Два умных указателя указывают на объект `Anton`: `anton` и `m_partner` объекта `Ivan`.

В конце функции partnerUp() умный указатель `ivan` выходит из области видимости первым. Когда это происходит, он проверяет, есть ли другие умные указатели, которые владеют объектом `Ivan` класса Human. Есть — `m_partner` объекта `Anton`. Из-за этого умный указатель не уничтожает `Ivan`-а (если он это сделает, то `m_partner` объекта `Anton` останется висячим указателем). Таким образом у нас остается один умный указатель, владеющий `Ivan`-ом (`m_partner` объекта `Anton`) и два умных указателя, владеющие `Anton`-ом (`anton` и `m_partner` объекта `Ivan`).

Затем умный указатель `anton` выходит из области видимости, и происходит то же самое. `anton` проверяет, есть ли другие умные указатели, которые также владеют объектом `Anton` класса Human. Есть — `m_partner` объекта `Ivan`, поэтому объект `Anton` не уничтожается. Таким образом, остаются два умных указателя:

- `m_partner` объекта `Ivan`, который указывает на `Anton`-а;
- `m_partner` объекта `Anton`, который указывает на `Ivan`-а.

Затем программа завершает свое выполнение, и ни объект `Anton`, ни объект `Ivan` не уничтожаются! По сути, `Anton` не дает уничтожить `Ivan`-а, а `Ivan` не дает уничтожить `Anton`-а.

Это тот случай, когда умные указатели типа std::shared_ptr формируют циклическую зависимость.

Циклическая зависимость

Циклическая зависимость (или ***«циклические ссылки»***) - это серия «ссылок», где текущий объект ссылается на следующий, а последний объект ссылается на первый. Эти «ссылки» не обязательно должны быть обычными ссылками в языке C++, они могут быть указателями, уникальными ID или любыми другими средствами идентификации конкретных объектов.

В контексте std::shared_ptr этими «ссылками» являются указатели.

Это именно то, что мы видим выше: `Anton` указывает на `Ivan`-а, а `Ivan` указывает на `Anton`-а. Аналогично, `A` указывает на `B`, `B` указывает на `C`, а `C` указывает на `A`. Практическая ценность такой циклической зависимости в том, что текущий объект «оставляет в живых» (не дает уничтожить) следующий объект.

Упрощенная циклическая зависимость

Оказывается, проблема циклической ссылки может возникнуть даже с одним умным указателем типа std::shared_ptr. Такая **циклическая зависимость называется упрощенной**. Хотя это редко случается на практике, но все же рассмотрим и этот случай:

```
1. #include <iostream>
2. #include <memory> // для std::shared_ptr
3.
4. class Item
5. {
6. public:
7.     std::shared_ptr<Item> m_ptr; // изначально пустой
8.
9.     Item() { std::cout << "Item acquired\n"; }
10.    ~Item() { std::cout << "Item destroyed\n"; }
11.};
12.
13.int main()
14.{
15.    auto ptr1 = std::make_shared<Item>();
16.
17.    ptr1->m_ptr = ptr1; // m_ptr теперь является владельцем Item-а, членом
   которого он является сам
18.
19.    return 0;
20.}
```

В примере, приведенном выше, когда `ptr1` выходит из области видимости, он не уничтожает Item, поскольку член `m_ptr` класса Item также владеет Item-ом. Таким

образом, не остается никого, кто мог бы удалить Item (`m_ptr` никогда не выходит из области видимости, поэтому он этого не сделает).

Результат выполнения программы:

```
Item acquired
```

Всё!

Так что же такое умный указатель std::weak_ptr?

Умный указатель std::weak_ptr был разработан для решения проблемы «циклической зависимости», описанной выше. std::weak_ptr является наблюдателем - он может наблюдать и получать доступ к тому же объекту, на который указывает std::shared_ptr (или другой std::weak_ptr), но не считаться владельцем этого объекта. Помните, когда std::shared_ptr выходит из области видимости, он проверяет, есть ли другие *владельцы* std::shared_ptr. **std::weak_ptr владельцем не считается**!

Давайте перепишем первую программу этого урока, но уже с использованием std::weak_ptr:

```
#include <iostream>
#include <memory> // для std::shared_ptr и std::weak_ptr
#include <string>

class Human
{
    std::string m_name;
    std::weak_ptr<Human> m_partner; // обратите внимание, здесь std::weak_ptr

public:

    Human(const std::string &name): m_name(name)
    {
        std::cout << m_name << " created\n";
    }
    ~Human()
    {
        std::cout << m_name << " destroyed\n";
    }

    friend bool partnerUp(std::shared_ptr<Human> &h1, std::shared_ptr<Human>
&h2)
    {
        if (!h1 || !h2)
            return false;

        h1->m_partner = h2;
        h2->m_partner = h1;

```

```
29.         std::cout << h1->m_name << " is now partnered with " << h2-
    >m_name << "\n";
30.
31.         return true;
32.     }
33. };
34.
35. int main()
36. {
37.     auto anton = std::make_shared<Human>("Anton");
38.     auto ivan = std::make_shared<Human>("Ivan");
39.
40.     partnerUp(anton, ivan);
41.
42.     return 0;
43. }
```

Результат выполнения программы:

```
Anton created
Ivan created
Anton is now partnered with Ivan
Ivan destroyed
Anton destroyed
```

Функционально всё работает почти идентично программе, приведенной в начале этого урока. Однако теперь, когда `ivan` выходит из области видимости, он видит, что нет другого std::shared_ptr, указывающего на `Ivan`-а (std::weak_ptr из `Anton`-а не считается). Поэтому он уничтожает `Ivan`-а. То же самое происходит и с `Anton`-ом.

Использование умного указателя std::weak_ptr

Недостатком умного указателя std::weak_ptr является то, что его нельзя использовать напрямую (нет оператора `->`). Чтобы использовать std::weak_ptr, вы сначала должны конвертировать его в std::shared_ptr (с помощью **метода lock()**), а затем уже использовать std::shared_ptr. Например, перепишем вышеприведенную программу:

```
1. #include <iostream>
2. #include <memory> // для std::shared_ptr и std::weak_ptr
3. #include <string>
4.
5. class Human
6. {
7.     std::string m_name;
8.     std::weak_ptr<Human> m_partner; // обратите внимание, здесь std::weak_ptr
9.
10. public:
11.
12.     Human(const std::string &name) : m_name(name)
13.     {
```

```
14.         std::cout << m_name << " created\n";
15.     }
16.     ~Human()
17.     {
18.         std::cout << m_name << " destroyed\n";
19.     }
20.
21.     friend bool partnerUp(std::shared_ptr<Human> &h1, std::shared_ptr<Human>
    &h2)
22.     {
23.         if (!h1 || !h2)
24.             return false;
25.
26.         h1->m_partner = h2;
27.         h2->m_partner = h1;
28.
29.         std::cout << h1->m_name << " is now partnered with " << h2-
    >m_name << "\n";
30.
31.         return true;
32.     }
33.
34.     const std::shared_ptr<Human> getPartner() const { return m_partner.lock();
    } // используем метод lock() для конвертации std::weak_ptr в std::shared_ptr
35.     const std::string& getName() const { return m_name; }
36. };
37.
38. int main()
39. {
40.     auto anton = std::make_shared<Human>("Anton");
41.     auto ivan = std::make_shared<Human>("Ivan");
42.
43.     partnerUp(anton, ivan);
44.
45.     auto partner = ivan->getPartner(); // передаем partner-у содержимое
    умного указателя, которым владеет ivan
46.     std::cout << ivan->getName() << "'s partner is: " << partner-
    >getName() << '\n';
47.
48.     return 0;
49. }
```

Результат выполнения программы:

```
Anton created
Ivan created
Anton is now partnered with Ivan
Ivan's partner is: Anton
Ivan destroyed
Anton destroyed
```

Нам не нужно беспокоиться о циклической зависимости с переменной `partner` (типа std::shared_ptr), так как она является простой локальной переменной внутри функции main() и уничтожается при завершении выполнения функции main().

Заключение

Умный указатель std::shared_ptr используется для владения одним динамически выделенным ресурсом сразу несколькими умными указателями. Ресурс будет уничтожен, когда последний std::shared_ptr выйдет из области видимости. std::weak_ptr используется, когда нужен умный указатель, который имеет доступ к ресурсу, но не считается его владельцем.

Тест

Исправьте вышеприведенную программу с упрощенной циклической зависимостью, чтобы Item был корректно освобожден.

Глава №15. Итоговый тест

Еще одна глава позади. Последний итоговый тест. Пора закрепить пройденный материал.

Теория

Класс умного указателя - это класс, предназначенный для управления динамически выделенной памятью и для её удаления, когда объект умного указателя выходит из области видимости.

Семантика копирования позволяет копировать классы с помощью конструктора копирования и оператора присваивания копированием.

Семантика перемещения позволяет классу передать владение ресурсами/объектом с помощью конструктора перемещения и оператора присваивания перемещением другому объекту (без выполнения копирования).

Умный указатель std::auto_ptr устарел и его следует избегать.

Ссылка r-value - это ссылка, которая инициализируется значениями r-values. Ссылка r-value создается с использованием двойного амперсанда. Помните, что писать функции, которые принимают в качестве параметров ссылки r-value — хорошо, но возвращать ссылки r-value — плохо.

Если мы создаем объект или выполняем операцию присваивания, где **аргументом является l-value**, то единственное разумное, что мы можем сделать - это скопировать l-value. Дело в том, что не всегда безопасно изменять l-value, так как он еще может быть использован позже в программе. Если у нас есть выражение `a = b`, то мы предполагаем, что `b` не будет каким-либо образом изменен позже.

Однако, если мы создаем объект или выполняем операцию присваивания, где **аргументом является r-value**, то мы знаем, что r-value - это всего лишь временный объект. Вместо того, чтобы копировать его (что может быть затратно), мы можем просто переместить его ресурсы (что не очень затратно) в объект, который создаем или которому присваиваем. Это безопасно, поскольку временный объект в любом случае будет уничтожен в конце выражения, и мы можем быть уверены, что он больше никогда не будет использован снова!

Вы можете использовать ключевое слово delete для **отключения семантики копирования** в создаваемых классах, удалив конструктор копирования и оператор присваивания копированием.

Функция std::move() позволяет конвертировать l-value в r-value. Это полезно, когда вы хотите использовать семантику перемещения вместо семантики копирования с аргументом, который изначально является l-value.

Умный указатель std::unique_ptr - это класс умного указателя, который единолично владеет переданным ему динамически выделенным ресурсом. **Функция std::make_unique()** — это функция, добавленная в C++14, которую вы должны использовать для создания нового std::unique_ptr. В std::unique_ptr семантика копирования по умолчанию отключена.

Умный указатель std::shared_ptr - это класс умного указателя, который следует использовать, когда нужно, чтобы одним динамически выделенным ресурсом владели сразу несколько умных указателей. Ресурс не будет уничтожен до тех пор, пока не будет уничтожен последний владеющий им std::shared_ptr. Для создания нового std::shared_ptr предпочтительнее использовать **функцию std::make_shared()**. Для создания дополнительного std::shared_ptr, который бы указывал на текущий ресурс, вы должны использовать семантику копирования.

Умный указатель std::weak_ptr - это класс умного указателя, который вы должны использовать, когда вам нужен один или несколько указателей с возможностью просмотра и доступа к динамически выделенному ресурсу, которым управляет другой std::shared_ptr, но, чтобы этот указатель не участвовал в уничтожении ресурса. std::weak_ptr владельцем ресурса не считается.

Тест

Задание №1

Объясните, когда следует использовать следующие типы умных указателей:

a) Умный указатель std::unique_ptr

b) Умный указатль std::shared_ptr

c) Умный указатель std::weak_ptr

d) Умный указатель std::auto_ptr

Задание №2

Объясните, как ссылки r-value участвуют в реализации семантики перемещения.

Задание №3

Что не так со следующими программами? Обновите их в соответствии с рекомендациями, полученными в этой главе.

a)

```
#include <iostream>
#include <memory> // для std::shared_ptr

class Item
{
public:
    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
};

int main()
{
    Item *item = new Item;
    std::shared_ptr<Item> ptr1(item);
    std::shared_ptr<Item> ptr2(item);

    return 0;
}
```

b)

```
#include <iostream>
#include <memory> // для std::shared_ptr

class Something; // предположим, что Something - это класс, который может выбросить исключение

int main()
{
    doSomething(std::shared_ptr<Something>(new Something), std::shared_ptr<Something>(new Something));

    return 0;
}
```

Урок №204. Стандартная библиотека шаблонов (STL)

Поздравляем! Вы прошли основную часть туториала по языку C++! На предыдущих уроках мы рассматривали много особенностей языка C++ (включая и некоторые фишки, которые были добавлены в версиях C++11/14/17).

Закрадывается очевидный вопрос: «Что дальше?». Вы, возможно, уже замечали, что большинство программ используют одни и те же концепции снова и снова: циклы, строки, массивы, способы сортировок и т.д. Вы, вероятно, еще заметили, что написание программ без использования контейнерных классов и общих алгоритмов — дело, которое подвержено многим ошибкам. Хорошей новостью является то, что язык C++ поставляется в комплекте с библиотекой классов (и не только), которую вы можете использовать при создании своих программ. Эта библиотека называется Стандартной библиотекой шаблонов.

Стандартная библиотека шаблонов

Стандартная библиотека шаблонов (сокр. **"*STL*"** от *"**S**tandard **T**emplate **L**ibrary"*) - это часть Стандартной библиотеки C++, которая содержит набор шаблонов контейнерных классов (например, std::vector и std::array), алгоритмов и итераторов. Изначально она была сторонней разработкой, но позже была включена в Стандартную библиотеку C++. Если вам нужен какой-нибудь общий класс или алгоритм, то, скорее всего, в Стандартной библиотеке шаблонов он уже есть. Положительным моментом является то, что вы можете использовать эти классы без необходимости писать и отлаживать их самостоятельно (и разбираться в том, как они реализованы). Кроме того, вы получаете достаточно эффективные (и уже много раз протестированные) версии этих классов. Недостатком является то, что не всё так просто/очевидно с функционалом Стандартной библиотеки шаблонов и это может быть несколько непонятно новичку, так как большинство классов на самом деле являются шаблонами классов.

К счастью, вы можете отделить себе кусочек Стандартной библиотеки шаблонов, чтобы его "распробовать" и при этом игнорировать всё остальное до тех пор, пока вы в нем не разберетесь.

На следующих нескольких уроках мы рассмотрим типы контейнерных классов, алгоритмов и итераторов, которые предоставляет Стандартная библиотека шаблонов. Затем мы еще углубимся в изучение некоторых конкретных классов.

Урок №205. Контейнеры STL

Безусловно, наиболее часто используемым функционалом библиотеки STL являются контейнерные классы (или как их еще называют — ***«контейнеры»***). Библиотека STL содержит много разных контейнерных классов, которые можно использовать в разных ситуациях. Если говорить в общем, то **контейнеры STL делятся на три основные категории**:

- последовательные;
- ассоциативные;
- адаптеры.

Сейчас сделаем их краткий обзор.

Последовательные контейнеры

Последовательные контейнеры (или ***«контейнеры последовательности»***) - это контейнерные классы, элементы которых находятся в последовательности. Их определяющей характеристикой является то, что вы можете добавить свой элемент в любое место контейнера. Наиболее распространенным примером последовательного контейнера является массив: при добавлении 4-х элементов в массив, эти элементы будут находиться (в массиве) в точно таком же порядке, в котором вы их добавили.

Начиная с C++11, **STL содержит 6 контейнеров последовательности**:

- std::vector;
- std::deque;
- std::array;
- std::list;
- std::forward_list;
- std::basic_string.

Класс vector (или просто ***«вектор»***) – это динамический массив, способный увеличиваться по мере необходимости для содержания всех своих элементов. Класс vector обеспечивает произвольный доступ к своим элементам через **оператор индексации** `[]`, а также поддерживает добавление и удаление элементов.

В следующей программе мы добавим 5 целых чисел в вектор и с помощью перегруженного оператора индексации `[]` получим к ним доступ для их последующего вывода:

```
#include <iostream>
#include <vector>

int main()
{

    std::vector<int> vect;
    for (int count=0; count < 5; ++count)
        vect.push_back(10 - count); // добавляем числа в конец массива

    for (int index=0; index < vect.size(); ++index)
        std::cout << vect[index] << ' ';

    std::cout << '\n';
}
```

Результат выполнения программы:

```
10 9 8 7 6
```

Класс deque (или просто ***«дек»***) - это двусторонняя очередь, реализованная в виде динамического массива, который может расти с обоих концов. Например:

```
#include <iostream>
#include <deque>

int main()
{
    std::deque<int> deq;
    for (int count=0; count < 4; ++count)
    {
        deq.push_back(count); // вставляем числа в конец массива
        deq.push_front(10 - count); // вставляем числа в начало массива
    }

    for (int index=0; index < deq.size(); ++index)
        std::cout << deq[index] << ' ';

    std::cout << '\n';
}
```

Результат выполнения программы:

```
7 8 9 10 0 1 2 3
```

List (или просто ***«список»***) - это двусвязный список, каждый элемент которого содержит 2 указателя: один указывает на следующий элемент списка, а другой — на предыдущий элемент списка. list предоставляет доступ только к началу и к концу списка - произвольный доступ запрещен. Если вы хотите найти значение где-то в

середине, то вы должны начать с одного конца и перебирать каждый элемент списка до тех пор, пока не найдете то, что ищете. Преимуществом двусвязного списка является то, что добавление элементов происходит очень быстро, если вы, конечно, знаете, куда хотите добавлять. Обычно для перебора элементов двусвязного списка используются итераторы.

Хотя о классе string (и wstring) обычно не говорят, как о последовательном контейнере, но он, по сути, таковым является, поскольку его можно рассматривать как вектор с элементами типа char (или wchar).

Ассоциативные контейнеры

Ассоциативные контейнеры - это контейнерные классы, которые автоматически сортируют все свои элементы (в том числе и те, которые добавляете вы). По умолчанию ассоциативные контейнеры выполняют сортировку элементов, используя оператор сравнения `<`.

- **set** - это контейнер, в котором хранятся только уникальные элементы, и повторения запрещены. Элементы сортируются в соответствии с их значениями.
- **multiset** - это set, но в котором допускаются повторяющиеся элементы.
- **map** (или ***«ассоциативный массив»***) - это set, в котором каждый элемент является парой "ключ-значение". "Ключ" используется для сортировки и индексации данных и должен быть уникальным. А "значение" - это фактические данные.
- **multimap** (или ***«словарь»***) - это map, который допускает дублирование ключей. Все ключи отсортированы в порядке возрастания, и вы можете посмотреть значение по ключу.

Адаптеры

Адаптеры - это специальные предопределенные контейнерные классы, которые адаптированы для выполнения конкретных заданий. Самое интересное заключается в том, что вы сами можете выбрать, какой последовательный контейнер должен использовать адаптер.

- **stack** (**стек**) - это контейнерный класс, элементы которого работают по принципу **LIFO** (англ. *«**L**ast **I**n, **F**irst **O**ut» = «Последним Пришёл, Первым Ушёл»*), т.е. элементы добавляются (вносятся) в конец контейнера и удаляются (выталкиваются) оттуда же (из конца контейнера). Обычно в стеках используется deque в качестве последовательного контейнера по умолчанию

(что немного странно, поскольку vector был бы более подходящим вариантом), но вы также можете использовать vector или list.

- **queue** (**очередь**) - это контейнерный класс, элементы которого работают по принципу **FIFO** (англ. *«**F**irst **I**n, **F**irst **O**ut» = «Первым Пришёл, Первым Ушёл»*), т.е. элементы добавляются (вносятся) в конец контейнера, но удаляются (выталкиваются) из начала контейнера. По умолчанию в очереди используется deque в качестве последовательного контейнера, но также может использоваться и list.
- **priority_queue** (**очередь с приоритетом**) - это тип очереди, в которой все элементы отсортированы (с помощью оператора сравнения $<$). При добавлении элемента, он автоматически сортируется. Элемент с наивысшим приоритетом (самый большой элемент) находится в самом начале очереди с приоритетом, также, как и удаление элементов выполняется с самого начала очереди с приоритетом.

На следующем уроке мы рассмотрим итераторы STL.

Урок №206. Итераторы STL

Итератор - это объект, который способен перебирать элементы контейнерного класса без необходимости пользователю знать реализацию определенного контейнерного класса. Во многих контейнерах (особенно в списке и в ассоциативных контейнерах) итераторы являются основным способом доступа к элементам этих контейнеров.

Функционал итераторов

Об итераторе можно думать, как об указателе на определенный элемент контейнерного класса с дополнительным набором перегруженных операторов для выполнения четко определенных функций:

- **Оператор** `*` возвращает элемент, на который в данный момент указывает итератор.
- **Оператор** `++` перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют **оператор** `--` для перехода к предыдущему элементу.
- **Операторы** `==` и `!=` используются для определения того, указывают ли два итератора на один и тот же элемент или нет. Для сравнения значений, на которые указывают два итератора, нужно сначала разыменовать эти итераторы, а затем использовать оператор `==` или оператор `!=`.
- **Оператор** `=` присваивает итератору новую позицию (обычно начальный или конечный элемент контейнера). Чтобы присвоить значение элемента, на который указывает итератор, другому объекту, нужно сначала разыменовать итератор, а затем использовать оператор `=`.

Каждый контейнерный класс имеет **4 основных метода для работы с оператором** `=`:

- **метод begin()** возвращает итератор, представляющий начальный элемент контейнера.
- **метод end()** возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере.
- **метод cbegin()** возвращает константный (только для чтения) итератор, представляющий начальный элемент контейнера.
- **метод cend()** возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.

Может показаться странным, что метод end() не указывает на последний элемент контейнера, но это сделано в целях упрощения использования циклов: цикл перебирает элементы до тех пор, пока итератор не достигнет метода end(), и тогда уже всё — «Баста!».

Наконец, все контейнеры предоставляют (как минимум) **два типа итераторов**:

- **container::iterator** - итератор для чтения/записи;
- **container::const_iterator** - итератор только для чтения.

Рассмотрим несколько примеров использования итераторов.

Итерация по вектору

Заполним вектор 5 числами и с помощью итераторов выведем значения вектора:

```
1. #include <iostream>
2. #include <vector>
3.
4. int main()
5. {
6.     std::vector<int> myVector;
7.     for (int count=0; count < 5; ++count)
8.         myVector.push_back(count);
9.
10.    std::vector<int>::const_iterator it; // объявляем итератор только для чтения
11.    it = myVector.begin(); // присваиваем ему начальный элемент вектора
12.    while (it != myVector.end()) // пока итератор не достигнет последнего элемента
13.        {
14.        std::cout << *it << " "; // выводим значение элемента, на который указывает итератор
15.        ++it; // и переходим к следующему элементу
16.        }
17.
18.    std::cout << '\n';
19.}
```

Результат выполнения программы:

```
0 1 2 3 4
```

Итерация по списку

Выполним все то же самое, что выше, но уже со списком:

```
1. #include <iostream>
2. #include <list>
3.
4. int main()
5. {
```

```
    std::list<int> myList;
    for (int count=0; count < 5; ++count)
        myList.push_back(count);

    std::list<int>::const_iterator it; // объявляем итератор
    it = myList.begin(); // присваиваем ему начальный элемент списка
    while (it != myList.end()) // пока итератор не достигнет последнего элемента
    {
        std::cout << *it << " "; // выводим значение элемента, на который указывает итератор
        ++it; // и переходим к следующему элементу
    }

    std::cout << '\n';
}
```

Результат выполнения программы:

```
0 1 2 3 4
```

Обратите внимание, код этой программы почти идентичен предыдущему примеру, хотя реализация векторов и списков значительно отличается друг от друга!

Итерация по set-у

В следующей программе мы создадим set из 5 чисел и, используя итератор, выведем эти значения:

```
#include <iostream>
#include <set>

int main()
{
    std::set<int> mySet;
    mySet.insert(8);
    mySet.insert(3);
    mySet.insert(-4);
    mySet.insert(9);
    mySet.insert(2);

    std::set<int>::const_iterator it; // объявляем итератор
    it = mySet.begin(); // присваиваем ему начальный элемент set-а
    while (it != mySet.end()) // пока итератор не достигнет последнего элемента
    {
        std::cout << *it << " "; // выводим значение элемента, на который указывает итератор
        ++it; // и переходим к следующему элементу
    }

    std::cout << '\n';
}
```

Результат выполнения программы:

```
-4 2 3 8 9
```

Обратите внимание, хотя заполнение set-а элементами отличается от способа заполнения вектора и списка выше, но код, используемый для перебора элементов set-а - идентичен.

Итерация по ассоциативному массиву

Этот пример немного сложнее. Контейнеры map и multimap принимают пары элементов (определенные как std::pair). Мы используем вспомогательную **функцию make_pair()** для создания пар. std::pair позволяет получить доступ к элементу (паре "ключ-значение") через первый и второй члены. В нашем ассоциативном массиве мы используем первый член в качестве "ключа", а второй в качестве "значения":

```
1. #include <iostream>
2. #include <map>
3. #include <string>
4.
5. int main()
6. {
7.     std::map<int, std::string> myMap;
8.     myMap.insert(std::make_pair(3, "cat"));
9.     myMap.insert(std::make_pair(2, "dog"));
10.    myMap.insert(std::make_pair(5, "chicken"));
11.    myMap.insert(std::make_pair(4, "lion"));
12.    myMap.insert(std::make_pair(1, "spider"));
13.
14.    std::map<int, std::string>::const_iterator it; // объявляем итератор
15.    it = myMap.begin(); // присваиваем ему начальный элемент вектора
16.    while (it != myMap.end()) // пока итератор не достигнет последнего
   элемента
17.    {
18.        std::cout << it->first << "=" << it-
   >second << " "; // выводим значение элемента, на который указывает итератор
19.        ++it; // и переходим к следующему элементу
20.    }
21.
22.    std::cout << '\n';
23. }
```

Результат выполнения программы:

```
1=spider 2=dog 3=cat 4=lion 5=chicken
```

Обратите внимание, насколько легко с помощью итераторов перебирать элементы контейнеров. Вам не нужно заботиться о том, как ассоциативный массив хранит свои данные!

Заключение

Итераторы предоставляют простой способ перебора элементов контейнерного класса без необходимости знать реализацию определенного контейнерного класса. В сочетании с алгоритмами STL и методами контейнерных классов итераторы становятся еще более мощными.

Стоит отметить еще один момент: итераторы должны быть реализованы для каждого контейнера отдельно, поскольку итератор должен знать реализацию контейнерного класса. Таким образом, итераторы всегда привязаны к конкретным контейнерным классам.

Урок №207. Алгоритмы STL

Помимо контейнеров и итераторов, библиотека STL также предоставляет ряд универсальных алгоритмов для работы с элементами контейнеров. Они позволяют выполнять такие операции, как поиск, сортировка, вставка, изменение позиции, удаление и копирование элементов контейнера.

Алгоритмы STL

Алгоритмы STL реализованы в виде глобальных функций, которые работают с использованием итераторов. Это означает, что каждый алгоритм нужно реализовать всего лишь один раз, и он будет работать со всеми контейнерами, которые предоставляют набор итераторов (включая и ваши собственные (пользовательские) контейнерные классы). Хотя это имеет огромный потенциал и предоставляет возможность быстро писать сложный код, у алгоритмов также есть и "тёмная сторона" — некоторая комбинация алгоритмов и типов контейнеров может не работать/работать с плохой производительностью/вызывать бесконечные циклы, поэтому следует быть осторожным.

Библиотека STL предоставляет довольно много алгоритмов. На этом уроке мы затронем лишь некоторые из наиболее распространенных и простых в использовании алгоритмов. Для их работы нужно подключить заголовочный файл algorithm.

Алгоритмы min_element() и max_element()

Алгоритмы min_element() и max_element() находят минимальный и максимальный элементы в контейнере:

```
#include <iostream>
#include <list>
#include <algorithm>

int main()
{
    std::list<int> li;
    for (int nCount=0; nCount < 5; ++nCount)
        li.push_back(nCount);

    std::list<int>::const_iterator it; // объявляем итератор
    it = min_element(li.begin(), li.end());
        std::cout << *it << ' ';
    it = max_element(li.begin(), li.end());
        std::cout << *it << ' ';

    std::cout << '\n';
}
```

Результат выполнения программы:

```
0 4
```

Алгоритмы find() и list::insert()

В следующем примере мы используем алгоритм find(), чтобы найти определенное значение в списке, а затем используем функцию list::insert() для добавления нового значения в список:

```
1. #include <iostream>
2. #include <list>
3. #include <algorithm>
4.
5. int main()
6. {
7.     std::list<int> li;
8.     for (int nCount=0; nCount < 5; ++nCount)
9.         li.push_back(nCount);
10.
11.    std::list<int>::iterator it; // объявляем итератор
12.    it = find(li.begin(), li.end(), 2); // ищем в списке число 2
13.    li.insert(it, 7); // используем алгоритм list::insert() для добавления
   числа 7 перед числом 2
14.
15.    for (it = li.begin(); it != li.end(); ++it) // выводим с помощью цикла и
   итератора элементы списка
16.        std::cout << *it << ' ';
17.
18.    std::cout << '\n';
19. }
```

Результат выполнения программы:

```
0 1 7 2 3 4
```

Алгоритмы sort() и reverse()

В следующем примере мы отсортируем весь вектор, выведем отсортированные элементы, а затем выведем их в обратном порядке:

```
1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
4.
5. int main()
6. {
7.
8.     std::vector<int> vect;
9.     vect.push_back(4);
10.    vect.push_back(8);
11.    vect.push_back(-3);
12.    vect.push_back(3);
13.    vect.push_back(-8);
```

```
14.     vect.push_back(12);
15.     vect.push_back(5);
16.
17.     std::sort(vect.begin(), vect.end()); // выполняем сортировку элементов
    вектора
18.
19.     std::vector<int>::const_iterator it; // объявляем итератор
20.     for (it = vect.begin(); it != vect.end(); ++it) // выводим с помощью цикла
    и итератора элементы вектора
21.         std::cout << *it << ' ';
22.
23.     std::cout << '\n';
24.
25.     std::reverse(vect.begin(), vect.end()); // сортируем элементы вектора в
    обратную сторону
26.
27.     for (it = vect.begin(); it != vect.end(); ++it) // выводим с помощью цикла
    и итератора элементы вектора
28.         std::cout << *it << ' ';
29.
30.     std::cout << '\n';
31. }
```

Результат выполнения программы:

```
-8 -3 3 4 5 8 12
12 8 5 4 3 -3 -8
```

Обратите внимание, общий алгоритм sort() не работает с вектором, у вектора есть свой собственный метод sort(), который, в данном случае, является более эффективным.

Заключение

Хотя это всего лишь небольшой пример алгоритмов, которые предоставляет STL, но этого уже должно быть достаточно, чтобы вы увидели пользу и то, насколько легко использовать алгоритмы STL в сочетании с итераторами и контейнерами.

Урок №208. Строковые классы std::string и std::wstring

Стандартная библиотека C++ содержит много полезных классов, но одним из наиболее полезных является std::string. **std::string** (и **std::wstring**) - это строковый класс, который позволяет выполнять операции присваивания, сравнения и изменения строк. На следующих нескольких уроках мы подробно рассмотрим строковые классы Стандартной библиотеки C++.

> ***Примечание*:** Строки C-style обычно называют *«строками C-style»*, тогда как std::string (и std::wstring) обычно называют просто *«строками»*.

Зачем нужен std::string?

Мы уже знаем, что строки C-style используют массивы типа char для хранения целой строки. Если вы попытаетесь что-либо сделать со строками C-style, то вы очень быстро обнаружите, что работать с ними трудно, запутаться легко, а проводить отладку сложно.

Строки C-style имеют много недостатков, в первую очередь связанных с тем, что вы должны самостоятельно управлять памятью. Например, если вы захотите поместить строку `Hello!` в буфер, то вам сначала нужно будет динамически выделить буфер правильной длины:

```
1. char *strHello = new char[7];
```

Не забудьте учесть дополнительный символ для нуль-терминатора! Затем вам нужно будет скопировать значение:

```
1. strcpy(strHello, "Hello!");
```

И здесь вам нельзя прогадать с длиной буфера, иначе произойдет переполнение! И, конечно, поскольку строка выделяется динамически, то вы должны её еще и правильно удалить:

```
1. delete[] strHello;
```

Не забудьте использовать форму оператора delete, которая работает с массивами, а не обычную форму оператора delete.

Кроме того, многие из интуитивно понятных операторов, которые предоставляет язык C++ для работы с числами, такие как `=`, `==`, `!=`, `<`, `>`, `>=` и `<=` попросту не работают со строками C-style. Иногда они могут работать без ошибок со стороны компилятора, но результат будет неверным. Например, сравнение двух строк C-style

с использованием оператора == на самом деле выполнит сравнение указателей, а не строк. Присваивание одной строки C-style другой строке C-style с использованием оператора = будет работать, но выполняться будет копирование указателя (поверхностное копирование), что не всегда то, что нам нужно. Такие вещи могут легко привести к ошибкам и сбоям в программе, а разбираться с ними не так уж и легко (относительно)!

Суть в том, что работая со строками C-style, вам нужно помнить множество придирчивых правил о том, что делать безопасно, а что — нет; запоминать много функций, таких как strcat() и strcmp(), чтобы использовать их вместо интуитивных операторов; а также самостоятельно выполнять управление памятью.

К счастью, язык C++ предоставляет гораздо лучший способ для работы со строками: классы std::string и std::wstring. Используя такие концепции C++, как конструкторы, деструкторы и перегрузку операторов, std::string позволяет создавать и манипулировать строками в интуитивно понятной форме и, что не менее важно, выполнять это безопасно! Никакого управления памятью, запоминания странных названий функций и значительно меньшая вероятность возникновения ошибок/сбоев.

Класс std::string

Весь функционал класса std::string находится в заголовочном файле string:

```
1. #include <string>
```

На самом деле в заголовочном файле есть 3 разных строковых класса. Первый - это шаблон класса с именем `basic_string<>`, который является родительским классом:

```
1. namespace std
2. {
3.     template<class charT, class traits = char_traits<charT>, class Allocator =
   allocator<charT> >
4.         class basic_string;
5. }
```

Вы не будете работать с этим классом напрямую, так что не беспокойтесь о том, что такое `traits` или `Allocator`. Значений по умолчанию, которые присваиваются этим объектам, будет достаточно почти во всех мыслимых и немыслимых случаях.

Дальше идут две разновидности `basic_string<>`:

```
1. namespace std
2. {
```

```
3.     typedef basic_string<char> string;
4.     typedef basic_string<wchar_t> wstring;
5. }
```

Это те два класса, которые вы будете использовать. **std::string используется для стандартных ASCII-строк (кодировка UTF-8), а std::wstring используется для Unicode-строк (кодировка UTF-16)**. Встроенного класса для строк UTF-32 нет.

Хотя вы будете напрямую использовать std::string и std::wstring, весь функционал реализован в классе `basic_string<>`. string и wstring имеют доступ к этому функционалу напрямую. Следовательно, все функции, приведенные ниже, работают как со string, так и с wstring.

Функционал std::string

Создание и удаление:

- **конструктор** - создает или копирует строку;
- **деструктор** - уничтожает строку.

Размер и ёмкость:

- **capacity()** - возвращает количество символов, которое строка может хранить без дополнительного перевыделения памяти;
- **empty()** - возвращает логическое значение, указывающее, является ли строка пустой;
- **length(), size()** - возвращают количество символов в строке;
- **max_size()** - возвращает максимальный размер строки, который может быть выделен;
- **reserve()** - расширяет или уменьшает ёмкость строки.

Доступ к элементам:

- `[]`, **at()** - доступ к элементу по заданному индексу.

Изменение:

- `=`, **assign()** - присваивают новое значение строке;
- `+=`, **append()**, **push_back()** - добавляют символы к концу строки;
- **insert()** - вставляет символы в произвольный индекс строки;
- **clear()** - удаляет все символы строки;
- **erase()** - удаляет символы по произвольному индексу строки;

- **replace()** - заменяет символы произвольных индексов строки другими символами;
- **resize()** - расширяет или уменьшает строку (удаляет или добавляет символы в конце строки);
- **swap()** - меняет местами значения двух строк.

Ввод/вывод:

- `>>`, **getline()** - считывают значения из входного потока в строку;
- `<<` - записывает значение строки в выходной поток;
- **c_str()** - конвертирует строку в строку C-style с нуль-терминатором в конце;
- **copy()** - копирует содержимое строки (которое без нуль-терминатора) в массив типа char;
- **data()** - возвращает содержимое строки в виде массива типа char, который не заканчивается нуль-терминатором.

Сравнение строк:

- `==`, `!=` - сравнивают, являются ли две строки равными/неравными (возвращают значение типа bool);
- `<`, `<=`, `>`, `>=` - сравнивают, являются ли две строки меньше или больше друг друга (возвращают значение типа bool);
- **compare()** - сравнивает, являются ли две строки равными/неравными (возвращает `-1`, `0` или `1`).

Подстроки и конкатенация:

- `+` - соединяет две строки;
- **substr()** - возвращает подстроку.

Поиск:

- **find** - ищет индекс первого символа/подстроки;
- **find_first_of** - ищет индекс первого символа из набора символов;
- **find_first_not_of** - ищет индекс первого символа НЕ из набора символов;
- **find_last_of** - ищет индекс последнего символа из набора символов;
- **find_last_not_of** - ищет индекс последнего символа НЕ из набора символов;
- **rfind** - ищет индекс последнего символа/подстроки.

Поддержка итераторов и распределителей (allocators):

- **begin()**, **end()** - возвращают "прямой" итератор, указывающий на первый и последний (элемент, который идет за последним) элементы строки;
- **get_allocator()** - возвращает распределитель;
- **rbegin()**, **rend()** - возвращают "обратный" итератор, указывающий на последний (т.е. "обратное" начало) и первый (элемент, который предшествует первому элементу строки — "обратный" конец) элементы строки. Отличие от begin() и end() в том, что движение итераторов происходит в обратную сторону.

Заключение

Вот и весь функционал std::string. Большинство из этих функций имеют несколько разновидностей для обработки разных типов входных данных, об этом мы еще поговорим.

Хотя функционал достаточно широк, все же есть несколько заметных упущений:

- поддержка регулярных выражений;
- конструкторы для создания строк из чисел;
- прописные/строчные функции;
- токенизация/разбиение строк на массивы;
- простые функции для получения левой или правой части строки;
- обрезка пробелов;
- конвертация из UTF-8 в UTF-16 и наоборот.

Всё вышеперечисленное вам придется реализовать самостоятельно, либо конвертировать вашу строку в строку C-style (используя функцию c_str()) и тогда уже использовать функционал, который предлагает язык C++.

> ***Примечание*:** Хотя мы используем string в наших примерах, всё это также применимо и к wstring.

Урок №209. Создание, уничтожение и конвертация std::string

Строковые классы имеют ряд конструкторов и деструктор, которые можно использовать для создания строк. Мы рассмотрим каждый из них.

string::string()

- Конструктор по умолчанию, который создает пустую строку.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sSomething;
7.     std::cout << sSomething;
8.
9.     return 0;
10. }
```

Результат:

string::string(const string& strString)

- Конструктор копирования, который создает новую строку путем копирования `strString`.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sSomething("What a string!");
7.     std::string sOutput(sSomething);
8.     std::cout << sOutput;
9.
10.    return 0;
11. }
```

Результат:

```
What a string!
```

string::string(const string& strString, size_type unIndex)
string::string(const string& strString, size_type unIndex, size_type unLength)

- Конструкторы, которые создают новые строки, которые состоят из строки `strString` (начиная с индекса `unIndex`) и количества символов, указанных в `unLength`.
- Если компилятор встречает `NULL`, то копирование строки завершается, даже если `unLength` не был достигнут.
- Если `unLength` не был указан, то все символы, начиная с `unIndex`, будут использованы.
- Если `unIndex` больше, чем размер строки, то выбрасывается исключение out_of_range.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sSomething("What a string!");
7.     std::string sOutput(sSomething, 3);
8.     std::cout << sOutput<< std::endl;
9.     std::string sOutput2(sSomething, 5, 6);
10.    std::cout << sOutput2 << std::endl;
11.
12.    return 0;
13.}
```

Результат:

```
t a string
a stri
```

string::string(const char *szCString)

- Конструктор, который создает новую строку из передаваемой строки C-style `szCString` вплоть до нуль-терминатора (но его не включает). Если размер результата превышает максимальную длину строки, то генерируется исключение length_error.

> ***Предупреждение:*** `szCString` не должен быть `NULL`.

Например:

```
1. #include <iostream>
2. #include <string>
3.
```

```
4.  int main()
5.  {
6.      const char *szSomething("What a string!");
7.      std::string sOutput(szSomething);
8.      std::cout << sOutput << std::endl;
9.
10.     return 0;
11. }
```

Результат:

```
What a string!
```

string::string(const char *szCString, size_type unLength)

- Конструктор, который создает новую строку из строки C-style `szCString` с количеством символов, указанных в `unLength`.
- Если размер результата превышает максимальную длину строки, то генерируется исключение length_error.

> ***Предупреждение:*** Только для этого конструктора значение `NULL` не обрабатывается как объект, указывающий на завершение строки `szCString`! Это означает, что компилятор дойдет до конца строки (если это позволяет `unLength`), даже если встретит `NULL`.

Например:

```
1.  #include <iostream>
2.  #include <string>
3.
4.  int main()
5.  {
6.      const char *szSomething("What a string!");
7.      std::string sOutput(szSomething, 7);
8.      std::cout << sOutput << std::endl;
9.
10.     return 0;
11. }
```

Результат:

```
What a
```

string::string(size_type nNum, char chChar)

- Конструктор, который создает новую строку, инициализированную символом `chChar` и требуемым количеством вхождений этого символа (указывается в `nNum`).

- Если размер результата превышает максимальную длину строки, то генерируется исключение length_error.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sOutput(5, 'G');
7.     std::cout << sOutput << std::endl;
8.
9.     return 0;
10. }
```

Результат:

```
GGGGG
```

template string::string(InputIterator itBeg, InputIterator itEnd)

- Конструктор, который создает новую строку, инициализированную символами диапазона `[itBeg, itEnd)`.
- Если размер результата превышает максимальную длину строки, то генерируется исключение length_error.

Здесь нет примера, так как вероятность того, что вы будете использовать этот конструктор, ничтожно мала.

string::~string()

- Деструктор, который уничтожает строку и освобождает память.

Примера нет, так как деструктор вызывается неявно.

Создание std::string из чисел

Одно заметное упущение в классе std::string — это отсутствие возможности создавать строки из чисел. Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sFive(5);
7.
8.     return 0;
```

```
}
```

Здесь мы получим ошибку неудачной конвертации значения типа int в std::basic_string. Самый простой способ конвертировать числа в строки - это **задействовать класс std::ostringstream**, который находится в заголовочном файле sstream. std::ostringstream уже настроен для приёма разных входных данных: символов, чисел, строк и т.д. А с помощью **std::istringstream** можно выполнять обратную конвертацию - выводить строки (либо через оператор вывода `>>`, либо через функцию str()).

Например, создадим std::string из разных входных данных:

```
#include <iostream>
#include <sstream>
#include <string>

template <typename T>
inline std::string ToString(T tX)
{
    std::ostringstream oStream;
    oStream << tX;
    return oStream.str();
}

int main()
{
    std::string sFive(ToString(5));
    std::string sSevenPointEight(ToString(7.8));
    std::string sB(ToString('B'));
    std::cout << sFive << std::endl;
    std::cout << sSevenPointEight << std::endl;
    std::cout << sB << std::endl;
}
```

Результат:

```
5
7.8
B
```

Обратите внимание, здесь отсутствует проверка на ошибки. Может случиться так, что конвертация `tX` в std::string будет неудачной. В таком случае, хорошим вариантом было бы подключить генерацию исключения.

Конвертация std::string в числа

Аналогично вышеприведенному решению:

```
#include <iostream>
#include <sstream>
#include <string>
```

```
4.
5.  template <typename T>
6.  inline bool FromString(const std::string& sString, T &tX)
7.  {
8.      std::istringstream iStream(sString);
9.      return (iStream >> tX) ? true : false; // извлекаем значение в tX,
    возвращаем true (если удачно) или false (если неудачно)
10. }
11.
12. int main()
13. {
14.     double dX;
15.     if (FromString("4.5", dX))
16.         std::cout << dX << std::endl;
17.     if (FromString("TOM", dX))
18.         std::cout << dX << std::endl;
19. }
```

Результат:

```
4.5
```

Обратите внимание, наша вторая конвертация потерпела неудачу, и мы получили `false`.

Урок №210. Длина и ёмкость std::string

При создании строки не помешало бы указать её длину и ёмкость (или хотя бы знать эти параметры).

Длина std::string

Длина строки - это количество символов, которые она содержит. Есть две идентичные функции для определения длины строки:

- **size_type string::length() const**
- **size_type string::size() const**

Обе эти функции возвращают текущее количество символов, которые содержит строка, исключая нуль-терминатор. Например:

```
#include <iostream>

int main()
{
    std::string sSomething("012345");
    std::cout << sSomething.length() << std::endl;

    return 0;
}
```

Результат:

```
6
```

Хотя также можно использовать функцию length() для определения того, содержит ли строка какие-либо символы или нет, эффективнее использовать функцию empty():

- **bool string::empty() const** — эта функция возвращает `true`, если в строке нет символов, и `false` — в противном случае.

Например:

```
#include <iostream>

int main()
{
    std::string sString1("Not Empty");
    std::cout << (sString1.empty() ? "true" : "false") << std::endl;
    std::string sString2; // пустая строка
    std::cout << (sString2.empty() ? "true" : "false")  << std::endl;

```

```
10.     return 0;
11. }
```

Результат:

```
false
true
```

Есть еще одна функция, связанная с длиной строки, которую вы, вероятно, никогда не будете использовать, но мы все равно её рассмотрим:

- **size_type string::max_size() const** — эта функция возвращает максимальное количество символов, которое может хранить строка. Это значение может варьироваться в зависимости от операционной системы и архитектуры операционной системы.

Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::string sString("MyString");
6.     std::cout << sString.max_size() << std::endl;
7. }
```

Результат:

```
2147483647
```

Ёмкость std::string

Ёмкость строки - это максимальный объем памяти, выделенный строке для хранения содержимого. Это значение измеряется в символах строки, исключая нуль-терминатор. Например, строка с ёмкостью 8 может содержать 8 символов.

- **size_type string::capacity() const** — эта функция возвращает количество символов, которое может хранить строка без дополнительного перераспределения/перевыделения памяти.

Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::string sString("0123456789");
6.     std::cout << "Length: " << sString.length() << std::endl;
```

```
7.      std::cout << "Capacity: " << sString.capacity() << std::endl;
8.
9.      return 0;
10. }
```

Результат:

```
Length: 10
Capacity: 15
```

> ***Примечание***: Запускать эту и следующие программы следует в полноценных IDE, а не в веб-компиляторах.

Обратите внимание, ёмкость строки больше её длины! Хотя длина нашей строки равна 10, памяти для нее выделено аж на 15 символов! Почему так?

Здесь важно понимать, что, если пользователь захочет поместить в строку больше символов, чем она может вместить, строка будет перераспределена и, соответственно, ёмкость будет больше. Например, если строка имеет длину и ёмкость равную 10, то добавление новых символов в строку приведет к её перераспределению. Делая ёмкость строки больше её длины, мы предоставляем пользователю некоторое буферное пространство для расширения строки (добавление новых символов).

Но в перераспределении есть также несколько нюансов:

- Во-первых, это сравнительно ресурсозатратно. Сначала должна быть выделена новая память. Затем каждый символ строки копируется в новую память. Если строка большая, то тратится много времени. Наконец, старая память должна быть удалена/освобождена. Если вы делаете много перераспределений, то этот процесс может значительно снизить производительность вашей программы.
- Во-вторых, всякий раз, когда строка перераспределяется, её содержимое получает новый адрес памяти. Это означает, что все текущие ссылки, указатели и итераторы строки становятся недействительными!

Обратите внимание, не всегда строки создаются с ёмкостью, превышающей её длину. Рассмотрим следующую программу:

```
1. #include <iostream>
2.
3. int main()
4. {
5.      std::string sString("0123456789abcde");
6.      std::cout << "Length: " << sString.length() << std::endl;
7.      std::cout << "Capacity: " << sString.capacity() << std::endl;
```

```
8.
9.      return 0;
10. }
```

Результат:

```
Length: 15
Capacity: 15
```

> ***Примечание***: Результаты могут отличаться в зависимости от компилятора.

Теперь давайте добавим еще один символ в конец строки и посмотрим на изменение её ёмкости:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      std::string sString("0123456789abcde");
6.      std::cout << "Length: " << sString.length() << std::endl;
7.      std::cout << "Capacity: " << sString.capacity() << std::endl;
8.
9.      // Добавляем новый символ
10.     sString += "f";
11.     std::cout << "Length: " << sString.length() << std::endl;
12.     std::cout << "Capacity: " << sString.capacity() << std::endl;
13.
14.     return 0;
15. }
```

Результат:

```
Length: 15
Capacity: 15
Length: 16
Capacity: 31
```

Есть еще одна функция (а точнее 2 варианта этой функции) для работы с ёмкостью строки:

- **void string::reserve(size_type unSize)** - при вызове этой функции мы устанавливаем ёмкость строки, равную, как минимум, `unSize` (она может быть и больше). Обратите внимание, для выполнения этой функции может потребоваться перераспределение.
- **void string::reserve()** - если вызывается эта функция или вышеприведенная функция с `unSize` меньше текущей ёмкости, то компилятор попытается срезать (уменьшить) ёмкость строки до размера её длины. Это необязательный запрос.

Например:

```
#include <iostream>

int main()
{
    std::string sString("0123456789");
    std::cout << "Length: " << sString.length() << std::endl;
    std::cout << "Capacity: " << sString.capacity() << std::endl;

    sString.reserve(300);
    std::cout << "Length: " << sString.length() << std::endl;
    std::cout << "Capacity: " << sString.capacity() << std::endl;

    sString.reserve();
    std::cout << "Length: " << sString.length() << std::endl;
    std::cout << "Capacity: " << sString.capacity() << std::endl;

    return 0;
}
```

Результат:

```
Length: 10
Capacity: 15
Length: 10
Capacity: 303
Length: 10
Capacity: 303
```

Здесь мы можем наблюдать две интересные вещи. Во-первых, хотя мы запросили ёмкость равную 300, мы фактически получили 303. Ёмкость всегда будет не меньше, чем мы запрашиваем (но может быть и больше). Затем мы запрашиваем изменение ёмкости в соответствии со строкой. Этот запрос был проигнорирован, так как очевидно, что ёмкость не изменилась.

Если вы заранее знаете, что вам нужна строка побольше, так как вы будете выполнять с ней множество операций, которые потенциально могут увеличить её длину или ёмкость, то вы можете избежать перераспределений, сразу установив строке её окончательную емкость:

```
#include <iostream>
#include <string>
#include <cstdlib> // для rand() и srand()
#include <ctime> // для time()

int main()
{
    srand(time(0)); // генерация случайного числа

    std::string sString; // длина 0
    sString.reserve(80); // резервируем 80 символов
```

```
12.
13.     // Заполняем строку случайными строчными символами
14.     for (int nCount = 0; nCount < 80; ++nCount)
15.         sString += 'a' + rand() % 26;
16.
17.     std::cout << sString;
18. }
```

Результат этой программы будет меняться при каждом её новом запуске:

```
tregsxxmselsqlfoahsvsxfmfwurcmmjclfcqqgzkzohztirriibtoibucswaud
yirkvjbwxfysoqzcc
```

Вместо того, чтобы перераспределять `sString` несколько раз, мы устанавливаем её ёмкость один раз, а затем просто заполняем данными.

Урок №211. Доступ к символам std::string. Конвертация std::string в строки C-style

Есть два практически идентичных способа доступа к символам std::string. Наиболее простой и быстрый — использовать перегруженный оператор индексации [].

char& string::operator[](size_type nIndex)
const char& string::operator[](size_type nIndex) const

- Обе эти функции возвращают символ под индексом `nIndex`.
- Передача неверного индекса приведет к неопределенным результатам.
- Использование функции length() в качестве индекса допустимо только для константных строк и возвращает значение, сгенерированное конструктором по умолчанию std::string. Это не рекомендуется делать.
- Поскольку `char&` — это тип возврата, то вы можете использовать его для изменения символов строки.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sSomething("abcdefg");
7.     std::cout << sSomething[4] << std::endl;
8.     sSomething[4] = 'A';
9.     std::cout << sSomething << std::endl;
10. }
```

Результат:

```
e
abcdAfg
```

Другой способ доступа к символам std::string медленнее, чем вышеприведенный вариант, так как использует исключения для проверки корректности `nIndex`. Если вы не уверены в корректности передаваемого `nIndex`, то вы должны использовать именно этот способ (тот, что описан ниже) для доступа к символам строки.

char& string::at(size_type nIndex)
const char& string::at(size_type nIndex) const

- Обе эти функции возвращают символ под индексом `nIndex`.

- Передача неверного индекса приведет к генерации исключения out_of_range.
- Поскольку `char&` — это тип возврата, то вы можете использовать его для изменения символов строки.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sSomething("abcdefg");
7.     std::cout << sSomething.at(4) << std::endl;
8.     sSomething.at(4) = 'A';
9.     std::cout << sSomething << std::endl;
10. }
```

Результат:

```
e
abcdAfg
```

Конвертация std::string в строки C-style

Многие функции (включая все функции языка C++) ожидают форматирования строк как строк C-style, а не как std::string. По этой причине std::string предоставляет 3 разных способа конвертации std::string в строки C-style.

const char* string::c_str() const

- Возвращает содержимое std::string в виде константной строки C-style.
- Добавляется нуль-терминатор.
- Строка C-style принадлежит std::string и не должна быть удалена.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sSomething("abcdefg");
7.     std::cout << strlen(sSomething.c_str());
8. }
```

Результат:

```
7
```

const char* string::data() const

- Возвращает содержимое std::string в виде константной строки C-style.
- Не добавляется нуль-терминатор.
- Строка C-style принадлежит std::string и не должна быть удалена.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sSomething("abcdefg");
7.     const char *szString = "abcdefg";
8.     // Функция memcmp() сравнивает две вышеприведенные строки C-style и
   возвращает 0, если они равны
9.     if (memcmp(sSomething.data(), szString, sSomething.length()) == 0)
10.        std::cout << "The strings are equal";
11.    else
12.        std::cout << "The strings are not equal";
13. }
```

Результат:

```
The strings are equal
```

size_type string::copy(char *szBuf, size_type nLength) const
size_type string::copy(char *szBuf, size_type nLength, size_type nIndex) const

- Отличие второго варианта этой функции от первого состоит в том, что копирование не более `nLength` символов передаваемой строки в `szBuf` начинается с символа под индексом `nIndex`. В первой же функции копирование всегда начинается с символа под индексом `[0]`.
- Количество скопированных символов возвращается.
- Caller отвечает за то, чтобы не произошло переполнения строки `szBuf`.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sSomething("lorem ipsum dolor sit amet");
7.
8.     char szBuf[20];
9.     int nLength = sSomething.copy(szBuf, 5, 6);
10.    szBuf[nLength] = '\0'; // завершаем строку в буфере
11.
12.    std::cout << szBuf << std::endl;
```

```
13.}
```

Результат:

```
ipsum
```

Если вы не гонитесь за максимальной эффективностью, то c_str() - это самый простой и безопасный способ конвертации std::string в строки C-style.

Урок №212. Присваивание и перестановка значений с std::string

Самый простой способ присвоить std::string другое значение - использовать перегруженный оператор присваивания =. Или, в качестве альтернативы, **метод assign()**.

string& string::operator=(const string& str)
string& string::assign(const string& str)
string& string::operator=(const char* str)
string& string::assign(const char* str)
string& string::operator=(char c)

- Эти функции позволяют присваивать std::string значения разных типов.
- Они возвращают скрытый указатель *this, что позволяет «связывать» объекты.
- Обратите внимание, функции assign(), которая бы принимала один символ, нет.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     using namespace std;
7.
8.     string sString;
9.
10.    // Присваиваем std::string другую строку
11.    sString = string("One");
12.    cout << sString << endl;
13.
14.    const string sTwo("Two");
15.    sString.assign(sTwo);
16.    cout << sString << endl;
17.
18.    // Присваиваем std::string строку C-style
19.    sString = "Three";
20.    cout << sString << endl;
21.
22.    sString.assign("Four");
23.    cout << sString << endl;
24.
25.    // Присваиваем std::string значение типа char
26.    sString = '5';
27.    cout << sString << endl;
28.
29.    // Связываем объекты
```

```
30.     string sOther;
31.     sString = sOther = "Six";
32.     cout << sString << " " << sOther << endl;
33.
34.     return 0;
35. }
```

Результат:

```
One
Two
Three
Four
5
Six Six
```

Метод assign() также имеет несколько других разновидностей.

string& string::assign(const string& str, size_type index, size_type len)

- Эта функция присваивает std::string подстроку `str` длиной `len`, начиная с `index`-а.
- Генерирует исключение out_of_range, если `index` недопустим.
- Возвращает скрытый указатель *this, что позволяет «связывать» объекты.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     const std::string sSomething("abcdefgh");
7.     std::string sDest;
8.
9.     sDest.assign(sSomething, 3, 5); // присваиваем sDest подстроку sSomething
   длиной 5, начиная с индекса 3
10.    std::cout << sDest << std::endl;
11.
12.    return 0;
13. }
```

Результат:

```
defgh
```

string& string::assign(const char* chars, size_type len)

- Эта функция присваивает std::string строку C-style длиной `len`.

- Генерирует исключение length_error, если результат превышает максимальное количество символов.
- Возвращает скрытый указатель *this, что позволяет «связывать» объекты.

Например:

```
#include <iostream>
#include <string>

int main()
{
    std::string sDest;

    sDest.assign("abcdefgh", 5);
    std::cout << sDest << std::endl;

    return 0;
}
```

Результат:

```
abcde
```

Эта функция потенциально опасна, поэтому использовать её не рекомендуется.

string& string::assign(size_type len, char c)

- Эта функция присваивает std::string определенное количество вхождений символа `c`. Количество вхождений указывается в `len`.
- Генерирует исключение length_error, если результат превышает максимальное количество символов.
- Возвращает скрытый указатель *this, что позволяет «связывать» объекты.

Например:

```
#include <iostream>
#include <string>

int main()
{
    std::string sDest;

    sDest.assign(5, 'h');
    std::cout << sDest << std::endl;

    return 0;
}
```

Результат:

```
hhhhh
```

Перестановка значений двух строк

Если у вас есть две строки, значения которых вы хотите поменять местами, используйте **функцию swap()**.

void string::swap(string &str)
void swap(string &str1, string &str2)

- Обе функции меняют местами значения двух строк. Первый вариант функции swap() меняет местами значения *this и `str`, а второй - `str1` и `str2`.
- Используйте данные функции вместо операции присваивания, если нужно поменять местами значения двух строк.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. using namespace std;
5.
6. int main()
7. {
8.     string sStr1("green");
9.     string sStr2("white");
10.
11.    cout << sStr1 << " " << sStr2 << endl;
12.    swap(sStr1, sStr2);
13.    cout << sStr1 << " " << sStr2 << endl;
14.    sStr1.swap(sStr2);
15.    cout << sStr1 << " " << sStr2 << endl;
16.
17.    return 0;
18. }
```

Результат:

```
green white
white green
green white
```

На следующем уроке мы рассмотрим добавление значений к std::string.

Урок №213. Добавление к std::string

Чтобы добавить одну строку к другой строке, можно использовать перегруженный оператор `+=`, функцию append() или функцию push_back().

string& string::operator+=(const string& str)
string& string::append(const string& str)

- Обе функции добавляют к std::string строку `str`.
- Возвращают скрытый указатель *this, что позволяет «связывать» объекты.
- Генерируют исключение length_error, если результат превышает максимально допустимое количество символов.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sString("one");
7.
8.     sString += std::string(" two");
9.
10.    std::string sThree(" three");
11.    sString.append(sThree);
12.
13.    std::cout << sString << std::endl;
14. }
```

Результат:

```
one two three
```

Существует также разновидность функции append(), которая может добавлять подстроку.

string& string::append(const string& str, size_type index, size_type num)

- Эта функция добавляет к std::string строку `str` с количеством символов, которые указываются в `num`, начиная с `index`-а.
- Возвращает скрытый указать *this, что позволяет «связывать» объекты.
- Генерирует исключение out_of_range, если `index` некорректен.
- Генерирует исключение length_error, если результат превышает максимально допустимое количество символов.

Например:

```
#include <iostream>
#include <string>

int main()
{
    std::string sString("one ");

    const std::string sTemp("twothreefour");
    sString.append(sTemp, 8, 4); // добавляем к std::string подстроку sTemp длиной 4, начиная с символа под индексом 8
    std::cout << sString << std::endl;
}
```

Результат:

```
one four
```

Оператор `+=` и функция append() также имеют версии, которые работают со строками C-style.

string& string::operator+=(const char* str)
string& string::append(const char* str)

- Обе функции добавляют к std::string строку C-style `str`.
- Возвращают скрытый указать *this, что позволяет «связывать» объекты.
- Генерируют исключение length_error, если результат превышает максимально допустимое количество символов.
- `str` не должен быть `NULL`.

Например:

```
#include <iostream>
#include <string>

int main()
{
    std::string sString("one");

    sString += " two";
    sString.append(" three");
    std::cout << sString << std::endl;
}
```

Результат:

```
one two three
```

И есть еще одна разновидность функции append(), которая работает со строками C-style.

string& string::append(const char* str, size_type len)

- Добавляет к std::string количество символов (которые указаны в `len`) строки C-style `str`.
- Возвращает скрытый указать *this, что позволяет «связывать» объекты.
- Генерирует исключение length_error, если результат превышает максимально допустимое количество символов.
- Игнорирует специальные символы (включая `”`).

Например:

```
#include <iostream>
#include <string>

int main()
{
    std::string sString("two ");

    sString.append("fivesix", 4);
    std::cout << sString << std::endl;
}
```

Результат:

```
two five
```

Эта функция опасна, поэтому использовать её не рекомендуется. Существуют также функции, которые добавляют отдельные (единичные) символы.

string& string::operator+=(char c)
void string::push_back(char c)

- Обе функции добавляют к std::string символ `c`.
- Оператор `+=` возвращает скрытый указатель *this, что позволяет «связывать» объекты.
- Обе функции генерируют исключение length_error, если результат превышает максимально допустимое количество символов.

Например:

```
#include <iostream>
#include <string>

int main()
```

```
{
    std::string sString("two");

    sString += ' ';
    sString.push_back('3');
    std::cout << sString << std::endl;
}
```

Результат:

```
two 3
```

string& string::append(size_type num, char c)

- Эта функция добавляет к std::string количество вхождений (которые указываются в `num`) символа `c`.
- Возвращает скрытый указатель *this, что позволяет «связывать» объекты.
- Генерирует исключение length_error, если результат превышает максимально допустимое количество символов.

Например:

```
#include <iostream>
#include <string>

int main()
{
    std::string sString("eee");

    sString.append(5, 'f');
    std::cout << sString << std::endl;
}
```

Результат:

```
eeefffff
```

Есть еще одна (последняя) вариация функции append(), использование которой вы не поймете, если не знакомы с итераторами.

string& string::append(InputIterator start, InputIterator end)

- Эта функция добавляет к std::string все символы из диапазона `(start, end)`.
- Возвращает скрытый указатель *this, что позволяет «связывать» объекты.
- Генерирует исключение length_error, если результат превышает максимально допустимое количество символов.

Урок №214. Вставка символов и строк в std::string

Вставлять символы/строки в std::string можно с помощью **функции insert()**.

string& string::insert(size_type index, const string& str)
string& string::insert(size_type index, const char* str)

- Обе функции вставляют символы/строки, начиная с определенного `index` std::string.
- Возвращают скрытый указатель *this, что позволяет «связывать» объекты.
- Генерируют исключение out_of_range, если `index` некорректен.
- Генерируют исключение length_error, если результат превышает максимально допустимое количество символов.
- Во второй версии функции insert() `str` не должен быть `NULL`.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sString("bbb");
7.     std::cout << sString << std::endl;
8.
9.     sString.insert(2, std::string("mmm"));
10.    std::cout << sString << std::endl;
11.
12.    sString.insert(5, "aaa");
13.    std::cout << sString << std::endl;
14. }
```

Результат:

```
bbb
bbmmmb
bbmmmaaab
```

А вот версия функции insert(), которая позволяет вставить с определенного `index` std::string подстроку.

string& string::insert(size_type index, const string& str, size_type startindex, size_type num)

- Эта функция вставляет с определенного `index` std::string указанное количество символов (`num`) строки `str`, начиная со `startindex`-а.

- Возвращает скрытый указатель *this, что позволяет «связывать» объекты.
- Генерирует исключение out_of_range, если `index` или `startindex` некорректны.
- Генерирует исключение length_error, если результат превышает максимально допустимое количество символов.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sString("bbb");
7.
8.     const std::string sInsert("012345");
9.     sString.insert(1, sInsert, 2, 4); // вставляем подстроку sInsert длиной 4, начиная с символа под индексом 2, в строку sString, начиная с индекса 1
10.    std::cout << sString << std::endl;
11. }
```

Результат:

```
b2345bb
```

А вот версия функции insert(), с помощью которой в std::string можно вставить часть строки C-style.

string& string::insert(size_type index, const char* str, size_type len)

- Эта функция вставляет с определенного `index` std::string указанное количество символов (`len`) строки C-style `str`.
- Возвращает скрытый указатель *this, что позволяет «связывать» объекты.
- Генерирует исключение out_of_range, если `index` некорректен.
- Генерирует исключение length_error, если результат превышает максимально допустимое количество символов.
- Игнорирует специальные символы (такие как `”`).

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sString("bbb");
7.
8.     sString.insert(2, "acdef", 4);
9.     std::cout << sString << std::endl;
```

```
10. }
```

Результат:

```
bbacdeb
```

А вот версия функции insert(), которая вставляет в std::string один и тот же символ несколько раз.

string& string::insert(size_type index, size_type num, char c)

- Эта функция вставляет с определенного `index` std::string указанное количество вхождений (`num`) символа `c`.
- Возвращает скрытый указатель *this, что позволяет «связывать» объекты.
- Генерирует исключение out_of_range, если `index` некорректен.
- Генерирует исключение length_error, если результат превышает максимально допустимое количество символов.

Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     std::string sString("bbb");
7.
8.     sString.insert(2, 3, 'a');
9.     std::cout << sString << std::endl;
10. }
```

Результат:

```
bbaaab
```

И, наконец, функция insert() имеет три разные версии, которые работают с итераторами.

void insert(iterator it, size_type num, char c)
iterator string::insert(iterator it, char c)
void string::insert(iterator it, InputIterator begin, InputIterator end)

- Первая версия функции вставляет в std::string указанное количество вхождений (`num`) символа `c` перед итератором `it`.
- Вторая версия функции вставляет в std::string одиночный символ `c` перед итератором `it` и возвращает итератор в позицию вставленного символа.

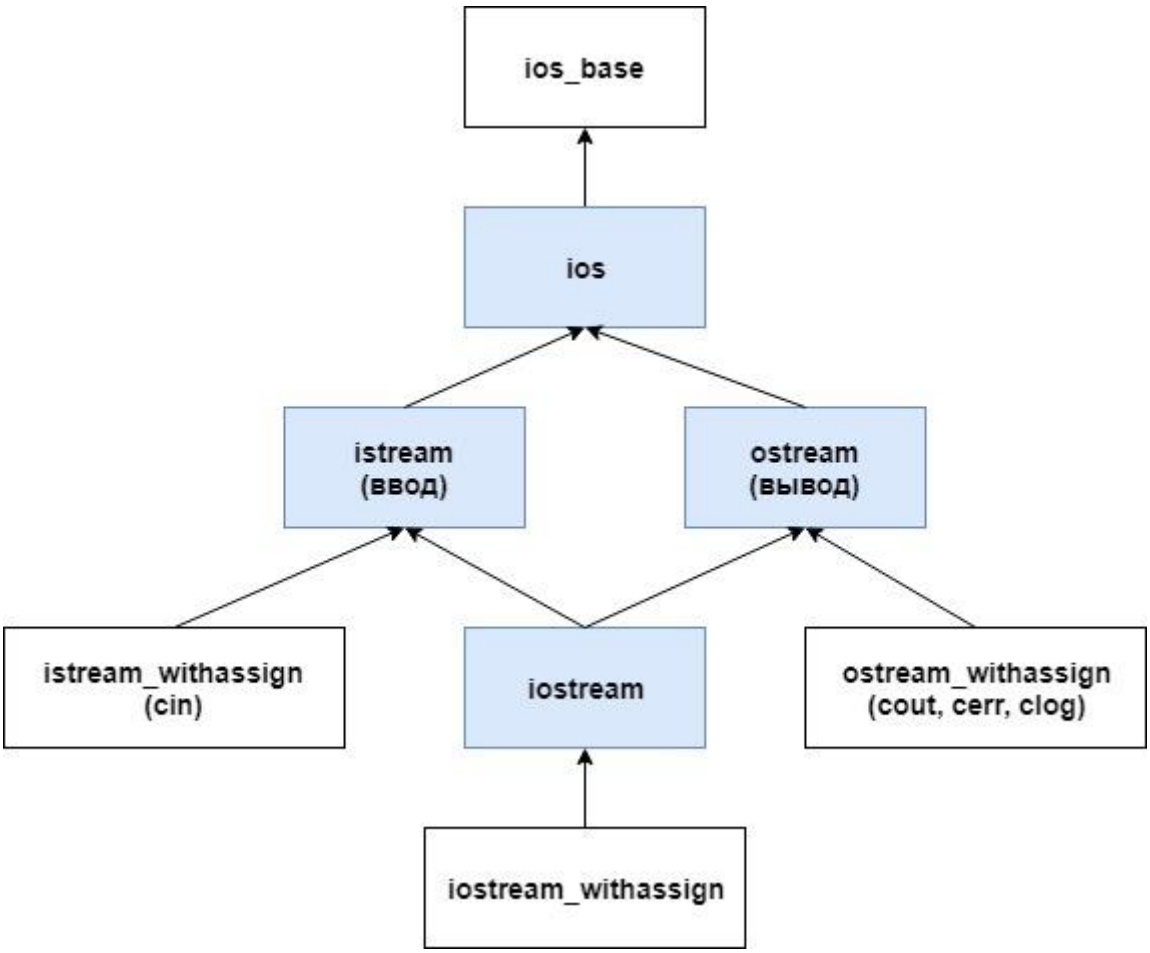
- Третья версия функции вставляет в std::string все символы диапазона `(begin, end)` перед итератором `it`.
- Все функции генерируют исключение length_error, если результат превышает максимально допустимое количество символов.

Урок №215. Потоки ввода и вывода

Функционал потоков ввода/вывода не определен как часть языка C++, а предоставляется Стандартной библиотекой C++ (и, следовательно, находится в пространстве имен std). На предыдущих уроках мы подключали заголовочный файл библиотеки iostream и использовали объекты cin и cout для простого ввода/вывода данных. На этом уроке мы детально рассмотрим библиотеку iostream.

Библиотека iostream

При подключении заголовочного файла iostream, мы получаем доступ ко всей иерархии классов библиотеки iostream, отвечающих за функционал ввода/вывода данных (включая класс, который называется iostream). Иерархия этих классов выглядит примерно следующим образом:



Первое, что вы можете заметить в этой иерархии – множественное наследование (то, что на самом деле не рекомендуется использовать). Тем не менее, библиотека iostream была разработана и тщательно протестирована соответствующим образом,

дабы избежать типичных ошибок, которые возникают при работе с множественным наследованием, поэтому вы можете спокойно использовать эту библиотеку.

Потоки в C++

Второе, что вы могли бы заметить - это частое использование слова *«stream»* (т.е. *«поток»*). По сути, ввод/вывод в языке C++ реализован с помощью потоков. Абстрактно, **поток** - это последовательность символов, к которым можно получить доступ. Со временем поток может производить или потреблять потенциально неограниченные объемы данных.

Мы будем иметь дело с двумя типами потоков. **Поток ввода** (или ***«входной поток»***) используется для хранения данных, полученных от источника данных: клавиатуры, файла, сети и т.д. Например, пользователь может нажать клавишу на клавиатуре в то время, когда программа не ожидает ввода. Вместо игнорирования нажатия клавиши, данные помещаются во входной поток, где затем ожидают ответа от программы.

И наоборот, **поток вывода** (или ***«выходной поток»***) используется для хранения данных, предоставляемых конкретному потребителю данных: монитору, файлу, принтеру и т.д. При записи данных на устройство вывода, это устройство может быть не готовым принять данные немедленно. Например, принтер все еще может прогреваться, когда программа уже записывает данные в выходной поток. Таким образом, данные будут находиться в потоке вывода до тех пор, пока принтер не начнет их использовать.

Некоторые устройства, такие как файлы и сети, могут быть источниками как ввода, так и вывода данных.

Хорошая новость заключается в том, что программисту не нужно знать детали взаимодействия потоков с разными устройствами и источниками данных, ему нужно только научиться взаимодействовать с этими потоками для чтения и записи данных.

Ввод/вывод в C++

Хотя класс `ios` является дочерним классу `ios_base`, очень часто именно этот класс будет наиболее родительским классом, с которым вы будете работать/взаимодействовать напрямую. Класс `ios` определяет кучу разных вещей, которые являются общими для потоков ввода/вывода.

Класс istream используется для работы с входными потоками. **Оператор извлечения** `>>` используется для извлечения значений из потока. Это имеет смысл: когда пользователь нажимает на клавишу клавиатуры, код этой клавиши помещается во входной поток. Затем программа извлекает это значение из потока и использует его.

Класс ostream используется для работы с выходными потоками. **Оператор вставки** `<<` используется для помещения значений в поток. Это также имеет смысл: вы вставляете свои значения в поток, а затем потребитель данных (например, монитор) использует их.

Класс iostream может обрабатывать как ввод, так и вывод данных, что позволяет ему осуществлять двунаправленный ввод/вывод.

Наконец, остались 3 класса, оканчивающиеся на `_withassign`. Эти потоковые классы являются дочерними классам `istream`, `ostream` и `iostream` (соответственно). В большинстве случаев вы не будете работать с ними напрямую.

Стандартные потоки в C++

Стандартный поток — это предварительно подключенный поток, который предоставляется программе её окружением. Язык C++ поставляется с 4-мя предварительно определенными стандартными объектами потоков, которые вы можете использовать (первые три вы уже встречали):

- **cin** — класс `istream_withassign`, связанный со стандартным вводом (обычно это клавиатура);
- **cout** — класс `ostream_withassign`, связанный со стандартным выводом (обычно это монитор);
- **cerr** — класс `ostream_withassign`, связанный со стандартной ошибкой (обычно это монитор), обеспечивающий небуферизованный вывод;
- **clog** — класс `ostream_withassign`, связанный со стандартной ошибкой (обычно это монитор), обеспечивающий буферизованный вывод.

Небуферизованный вывод обычно обрабатывается сразу же, тогда как буферизованный вывод обычно сохраняется и выводится как блок. Поскольку `clog` используется редко, то его обычно игнорируют.

Пример на практике

Вот пример использования ввода/вывода данных со стандартными потоками:

```
#include <iostream>
#include <cstdlib> // для exit()

int main()
{
    // Сначала мы используем оператор вставки с объектом cout для вывода текста на монитор
    std::cout << "Enter your age: " << std::endl;

    // Затем мы используем оператор извлечения с объектом cin для получения пользовательского ввода
    int nAge;
    std::cin >> nAge;

    if (nAge <= 0)
    {
        // В этом случае мы используем оператор вставки с объектом cerr для вывода сообщения об ошибке
        std::cerr << "Oops, you entered an invalid age!" << std::endl;
        exit(1);
    }

    // А здесь мы используем оператор вставки с объектом cout для вывода результата
    std::cout << "You entered " << nAge << " years old" << std::endl;

    return 0;
}
```

На следующих уроках мы детально рассмотрим функционал потоков ввода/вывода.

Урок №216. Функционал класса istream

Библиотека iostream по своей сути довольно сложная, поэтому мы не сможем охватить её полностью в рамках данных уроков. Тем не менее, мы можем рассмотреть её основной функционал. На этом уроке мы разберемся с классом `istream`.

> ***Примечание***: Весь функционал объектов, которые работают с потоками ввода/вывода, находится в пространстве имен std. Это означает, что вам нужно либо добавлять префикс `std::` ко всем объектам и функциям ввода/вывода, либо использовать строку `using namespace std;`.

Оператор извлечения

Как вы уже узнали на предыдущем уроке, мы можем использовать оператор извлечения `>>` для считывания информации из входного потока. Одной из наиболее распространенных проблем при считывании строк из входного потока является предотвращение переполнения. Например:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     char buf[12];
6.     std::cin >> buf;
7. }
```

Что произойдет, если пользователь введет 20 символов? Правильно, переполнение. Одним из способов решения этой проблемы является использование манипуляторов. **Манипулятор** — это объект, который применяется для изменения потока данных с использованием операторов извлечения (`>>`) или вставки (`<<`).

Мы уже работали с одним из манипуляторов — endl, который одновременно выводит символ новой строки и удаляет текущие данные из буфера. Язык C++ предоставляет еще один манипулятор - **setw()** (из заголовочного файла iomanip), который используется для ограничения количества символов, считываемых из потока. Для использования setw() вам нужно просто передать в качестве параметра максимальное количество символов для извлечения и вставить вызов этого манипулятора следующим образом:

```
1. #include <iostream>
2. #include <iomanip>
3.
4. int main()
5. {
```

```
6.      char buf[12];
7.      std::cin >> std::setw(12) >> buf;
8.  }
```

Эта программа теперь прочитает только первые 11 символов из входного потока (+ один символ для нуль-терминатора). Все остальные символы останутся в потоке до следующего извлечения.

Извлечение и пробелы

Важный момент: оператор извлечения работает с «отформатированными» данными, т.е. он игнорирует все пробелы, символы табуляции и символ новой строки. Например:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      char ch;
6.      while (std::cin >> ch)
7.          std::cout << ch;
8.
9.      return 0;
10. }
```

Если пользователь введет следующее:

```
Hello! My name is Anton
```

То оператор извлечения пропустит все пробелы и символы новой строки. Следовательно, результат выполнения программы:

```
Hello!MynameisAnton
```

Часто пользовательский ввод все же нужен со всеми его пробелами. Для этого класс `istream` предоставляет множество функций. Одной из наиболее полезных является **фунция get()**, которая извлекает символ из входного потока. Вот вышеприведенная программа, но уже с использованием функции get():

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      char ch;
6.      while (std::cin.get(ch))
7.          std::cout << ch;
8.
9.      return 0;
10. }
```

Теперь, если мы введем следующее:

```
Hello! My name is Anton
```

То получим:

```
Hello! My name is Anton
```

Функция get() также имеет строковую версию, в которой можно указать максимальное количество символов для извлечения. Например:

```
#include <iostream>

int main()
{
    char strBuf[12];
    std::cin.get(strBuf, 12);
    std::cout << strBuf << std::endl;

    return 0;
}
```

Если мы введем следующее:

```
Hello! My name is Anton
```

То получим:

```
Hello! My n
```

Обратите внимание, программа считывает только первые 11 символов (+ нуль-терминатор). Остальные символы остаются во входном потоке.

Один важный нюанс: функция get() не считывает символ новой строки! Например:

```
#include <iostream>

int main()
{
    char strBuf[12];

    // Считываем первые 11 символов
    std::cin.get(strBuf, 12);
    std::cout << strBuf << std::endl;

    // Считываем дополнительно еще 11 символов
    std::cin.get(strBuf, 12);
    std::cout << strBuf << std::endl;
    return 0;
}
```

Если пользователь введет следующее:

```
Hello!
```

То получит:

```
Hello!
```

И программа сразу же завершит свое выполнение! Почему так? Почему не срабатывает второй ввод данных? Дело в том, что первый get() считывает символы до символа новой строки, а затем останавливается. Второй get() видит, что во входном потоке все еще есть данные и пытается их извлечь. Но первый символ, на который он натыкается — символ новой строки, поэтому происходит второй «Стоп!».

Для решения данной проблемы класс `istream` предоставляет **функцию getline()**, которая работает точно так же, как и функция get(), но при этом может считывать символы новой строки:

```
#include <iostream>

int main()
{
    char strBuf[12];

    // Считываем 11 символов
    std::cin.getline(strBuf, 12);
    std::cout << strBuf << std::endl;

    // Считываем дополнительно еще 11 символов
    std::cin.getline(strBuf, 12);
    std::cout << strBuf << std::endl;
    return 0;
}
```

Этот код работает точно так, как ожидается, даже если пользователь введет строку с символом новой строки.

Если вам нужно узнать количество символов, извлеченных последним getline(), используйте **функцию gcount()**:

```
#include <iostream>

int main()
{
    char strBuf[100];
    std::cin.getline(strBuf, 100);
    std::cout << strBuf << std::endl;
    std::cout << std::cin.gcount() << " characters were read" << std::endl;

    return 0;
```

```
11. }
```

Результат:

```
Hello! My name is Anton
24 characters were read
```

Специальная версия функции getline() для std::string

Есть специальная версия функции getline(), которая находится вне класса `istream` и используется для считывания переменных типа std::string. Она не является членом ни `ostream`, ни `istream`, а подключается заголовочным файлом string. Например:

```
1. #include <iostream>
2. #include <string>
3.
4. int main()
5. {
6.     using namespace std;
7.
8.     string strBuf;
9.     getline(cin, strBuf);
10.    cout << strBuf << endl;
11.
12.    return 0;
13. }
```

Еще несколько полезных функций класса istream

Есть еще несколько полезных функций класса `istream`, которые вы можете использовать:

- **функция ignore()** — игнорирует первый символ из потока.
- **функция ignore(int nCount)** — игнорирует первые `nCount` (количество) символов из потока.
- **функция peek()** — считывает символ из потока, при этом не удаляя его из потока.
- **функция unget()** — возвращает последний считанный символ обратно в поток, чтобы его можно было извлечь в следующий раз.
- **функция putback(char ch)** — помещает выбранный вами символ обратно в поток, чтобы его можно было извлечь в следующий раз.

Класс `istream` содержит еще множество других полезный функций и их вариаций, но это уже тема для отдельного туториала.

Урок №217. Функционал классов ostream и ios. Форматирование вывода

На этом уроке мы рассмотрим функционал классов `ostream` и `ios` в языке C++.

> ***Примечание***: Весь функционал объектов, которые работают с потоками ввода/вывода, находится в пространстве имен std. Это означает, что вам нужно либо добавлять префикс `std::` ко всем объектам и функциям ввода/вывода, либо использовать в программе строку `using namespace std;`.

Форматирование вывода

Оператор вставки (вывода) `<<` используется для помещения информации в выходной поток. А как мы уже знаем из урока о потоках, классы `istream` и `ostream` являются дочерними классу `ios`. Одной из задач `ios` (и `ios_base`) является управление параметрами форматирования вывода.

Есть два способа управления параметрами форматирования вывода:

- флаги — это логические переменные, которые можно *включить*/*выключить*;
- **манипуляторы** — это объекты, которые помещаются в поток и влияют на способ ввода/вывода данных.

Для включения флага используйте **функцию setf()** с соответствующим флагом в качестве параметра. Например, по умолчанию C++ не выводит знак `+` перед положительными числами. Однако, используя флаг std::showpos, мы можем это изменить:

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout.setf(std::ios::showpos); // включаем флаг std::showpos
6.     std::cout << 30 << '\n';
7. }
```

Результат:

```
+30
```

Также можно включить сразу несколько флагов, используя побитовый оператор ИЛИ (`|`):

```
1. #include <iostream>
```

```
2.
3.  int main()
4.  {
5.      std::cout.setf(std::ios::showpos | std::ios::uppercase); // включаем флаги
    std::showpos и std::uppercase
6.      std::cout << 30 << '\n';
7.  }
```

Чтобы отключить флаг, используйте функцию **unsetf()**:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      std::cout.setf(std::ios::showpos); // включаем флаг std::showpos
6.      std::cout << 30 << '\n';
7.      std::cout.unsetf(std::ios::showpos); // выключаем флаг std::showpos
8.      std::cout << 31 << '\n';
9.  }
```

Результат:

```
+30
31
```

Многие флаги принадлежат к определенным группам форматирования. **Группа форматирования** — это группа флагов, которые задают аналогичные (иногда взаимоисключающие) параметры форматирования вывода. Например, есть группа форматирования basefield.

Флаги группы форматирования basefield:

- **oct** (от англ. *"**oct**al"* = *"восьмеричный"*) — восьмеричная система счисления;
- **dec** (от англ. *"**dec**imal"* = *"десятичный"*) — десятичная система счисления;
- **hex** (от англ. *"**hex**adecimal"* = *"шестнадцатеричный"*) — шестнадцатеричная система счисления.

Эти флаги управляют выводом целочисленных значений. По умолчанию установлен флаг std::dec, т.е. значения выводятся в десятичной системе счисления. Попробуем сделать следующее:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      std::cout.setf(std::ios::hex); // включаем флаг std::hex
6.      std::cout << 30 << '\n';
7.  }
```

Результат:

```
30
```

Ничего не работает! Почему? Дело в том, что setf() только включает флаги, он не настолько умен, чтобы одновременно отключать другие (взаимоисключающие) флаги. Следовательно, когда мы включаем std::hex, std::dec также включен и у него приоритет выше. Есть два способа решения данной проблемы.

Во-первых, мы можем отключить std::dec, а затем включить std::hex:

```
#include <iostream>

int main()
{
    std::cout.unsetf(std::ios::dec); // выключаем вывод в десятичной системе счисления
    std::cout.setf(std::ios::hex); // включаем вывод в шестнадцатеричной системе счисления
    std::cout << 30 << '\n';
}
```

Теперь уже результат тот, что нужно:

```
1e
```

Второй способ — использовать вариацию функции setf(), которая принимает два параметра:

- первый параметр — это флаг, который нужно включить/выключить;
- второй параметр — группа форматирования, к которой принадлежит флаг.

При использовании этой вариации функции setf() все флаги, принадлежащие группе форматирования, отключаются, а включается только передаваемый флаг.
Например:

```
#include <iostream>

int main()
{
    // Включаем и оставляем включенным единственный флаг (std::hex) группы форматирования std::basefield
    std::cout.setf(std::ios::hex, std::ios::basefield);
    std::cout << 30 << '\n';
}
```

Результат:

```
1e
```

Язык C++ также предоставляет еще один способ изменения параметров форматирования: манипуляторы. Фишка манипуляторов в том, что они достаточно умны, чтобы одновременно включать и выключать соответствующие флаги. Например:

```
#include <iostream>

int main()
{
    std::cout << std::hex << 30 << '\n'; // выводим 30 в шестнадцатеричной системе счисления
    std::cout << 31 << '\n'; // мы все еще находимся в шестнадцатеричной системе счисления
    std::cout << std::dec << 32 << '\n'; // перемещаемся обратно в десятичную систему счисления
}
```

Результат:

```
1e
1f
32
```

В общем, использовать манипуляторы гораздо проще, нежели включать/выключать флаги. Многие параметры форматирования можно изменять как через флаги, так и через манипуляторы, но есть и такие параметры форматирования, которые изменить можно либо только через флаги, либо только через манипуляторы.

Полезные флаги, манипуляторы и методы

Ниже мы рассмотрим список наиболее полезных флагов, манипуляторов и методов. Флаги находятся в классе `ios`, манипуляторы — в пространстве имен std, а методы — в классе `ostream`.

Флаг:

- **boolalpha** — если включен, то логические значения выводятся как `true`/`false`. Если выключен, то логические значения выводятся как `0`/`1`.

Манипуляторы:

- **boolalpha** — логические значения выводятся как `true`/`false`.
- **noboolalpha** — логические значения выводятся как `0`/`1`.

Например:

```
#include <iostream>
```

```
2.
3.  int main()
4.  {
5.      std::cout << true << " " << false << '\n';
6.
7.      std::cout.setf(std::ios::boolalpha);
8.      std::cout << true << " " << false << '\n';
9.
10.     std::cout << std::noboolalpha << true << " " << false << '\n';
11.
12.     std::cout << std::boolalpha << true << " " << false << '\n';
13. }
```

Результат:

```
1 0
true false
1 0
true false
```

Флаг:

- **showpos** — если включен, то перед положительными числами указывается знак `+`.

Манипуляторы:

- **showpos** — перед положительными числами указывается знак `+`.
- **noshowpos** — перед положительными числами не указывается знак `+`.

Например:

```
1.  #include <iostream>
2.
3.  int main()
4.  {
5.      std::cout << 7 << '\n';
6.
7.      std::cout.setf(std::ios::showpos);
8.      std::cout << 7 << '\n';
9.
10.     std::cout << std::noshowpos << 7 << '\n';
11.
12.     std::cout << std::showpos << 7 << '\n';
13. }
```

Результат:

```
7
+7
7
+7
```

Флаг:

- **uppercase** — если включен, то используются заглавные буквы.

Манипуляторы:

- **uppercase** — используются заглавные буквы.
- **nouppercase** — используются строчные буквы.

Например:

```
#include <iostream>

int main()
{
    std::cout << 12345678.9 << '\n';

    std::cout.setf(std::ios::uppercase);
    std::cout << 12345678.9 << '\n';

    std::cout << std::nouppercase << 12345678.9 << '\n';

    std::cout << std::uppercase << 12345678.9 << '\n';
}
```

Результат:

```
1.23457e+007
1.23457E+007
1.23457e+007
1.23457E+007
```

Флаги группы форматирования basefield:

- **dec** — значения выводятся в десятичной системе счисления;
- **hex** — значения выводятся в шестнадцатеричной системе счисления;
- **oct** — значения выводятся в восьмеричной системе счисления.

Манипуляторы:

- **dec** — значения выводятся в десятичной системе счисления;
- **hex** — значения выводятся в шестнадцатеричной системе счисления;
- **oct** — значения выводятся в восьмеричной системе счисления.

Например:

```
#include <iostream>

```

```
3. int main()
4. {
5.     std::cout << 30 << '\n';
6.
7.     std::cout.setf(std::ios::dec, std::ios::basefield);
8.     std::cout << 30 << '\n';
9.
10.    std::cout.setf(std::ios::oct, std::ios::basefield);
11.    std::cout << 30 << '\n';
12.
13.    std::cout.setf(std::ios::hex, std::ios::basefield);
14.    std::cout << 30 << '\n';
15.
16.    std::cout << std::dec << 30 << '\n';
17.    std::cout << std::oct << 30 << '\n';
18.    std::cout << std::hex << 30 << '\n';
19. }
```

Результат:

```
30
30
36
1e
30
36
1e
```

Теперь вы уже должны понимать связь между флагами и манипуляторами.

Точность, запись чисел и десятичная точка

Используя манипуляторы (или флаги), можно изменить точность и формат вывода значений типа с плавающей точкой.

Флаги группы форматирования floatfield:

- **fixed** — используется десятичная запись чисел типа с плавающей точкой;
- **scientific** — используется экспоненциальная запись чисел типа с плавающей точкой;
- **showpoint** — всегда отображается десятичная точка и конечные нули для чисел типа с плавающей точкой.

Манипуляторы:

- **fixed** — используется десятичная запись значений;
- **scientific** — используется экспоненциальная запись значений;

- **showpoint** — отображается десятичная точка и конечные нули чисел типа с плавающей точкой;
- **noshowpoint** — не отображаются десятичная точка и конечные нули чисел типа с плавающей точкой;
- **setprecision(int)** — задаем точность для чисел типа с плавающей точкой.

Методы:

- **precision()** — возвращаем текущую точность для чисел типа с плавающей точкой;
- **precision(int)** — задаем точность для чисел типа с плавающей точкой.

Если используется десятичная или экспоненциальная запись чисел, то точность определяет количество цифр после запятой/точки. Обратите внимание, если точность меньше количества значащих цифр, то число будет округлено. Например:

```
1. #include <iostream>
2. #include <iomanip> // для setprecision()
3.
4. int main()
5. {
6.     std::cout << std::fixed;
7.     std::cout << std::setprecision(3) << 123.456 << '\n';
8.     std::cout << std::setprecision(4) << 123.456 << '\n';
9.     std::cout << std::setprecision(5) << 123.456 << '\n';
10.    std::cout << std::setprecision(6) << 123.456 << '\n';
11.    std::cout << std::setprecision(7) << 123.456 << '\n';
12.
13.    std::cout << std::scientific << '\n';
14.    std::cout << std::setprecision(3) << 123.456 << '\n';
15.    std::cout << std::setprecision(4) << 123.456 << '\n';
16.    std::cout << std::setprecision(5) << 123.456 << '\n';
17.    std::cout << std::setprecision(6) << 123.456 << '\n';
18.    std::cout << std::setprecision(7) << 123.456 << '\n';
19. }
```

Результат:

```
123.456
123.4560
123.45600
123.456000
123.4560000

1.235e+02
1.2346e+02
1.23456e+02
1.234560e+02
1.2345600e+02
```

Если не используются ни десятичная, ни экспоненциальная запись чисел, то точность определяет, сколько значащих цифр будет отображаться. Например:

```
#include <iostream>
#include <iomanip> // для setprecision()

int main()
{
    std::cout << std::setprecision(3) << 123.456 << '\n';
    std::cout << std::setprecision(4) << 123.456 << '\n';
    std::cout << std::setprecision(5) << 123.456 << '\n';
    std::cout << std::setprecision(6) << 123.456 << '\n';
    std::cout << std::setprecision(7) << 123.456 << '\n';
}
```

Результат:

```
123
123.5
123.46
123.456
123.456
```

Используя манипулятор или флаг showpoint, мы можем заставить программу выводить десятичную точку и конечные нули. Например:

```
#include <iostream>
#include <iomanip> // для setprecision()

int main()
{
    std::cout << std::showpoint;
    std::cout << std::setprecision(3) << 123.456 << '\n';
    std::cout << std::setprecision(4) << 123.456 << '\n';
    std::cout << std::setprecision(5) << 123.456 << '\n';
    std::cout << std::setprecision(6) << 123.456 << '\n';
    std::cout << std::setprecision(7) << 123.456 << '\n';
}
```

Результат:

```
123.
123.5
123.46
123.456
123.4560
```

Ширина поля, символы-заполнители и выравнивание

Обычно числа выводятся без учета пространства вокруг них. Тем не менее, числа можно выравнивать. Чтобы это сделать, нужно сначала определить ширину поля (т.е. количество пространства (пробелов) вокруг значений).

Флаги группы форматирования adjustfield:

- **internal** — знак значения выравнивается по левому краю, а само значение - по правому краю;
- **left** — значение и его знак выравниваются по левому краю;
- **right** — значение и его знак выравниваются по правому краю.

Манипуляторы:

- **internal** — знак значения выравнивается по левому краю, а само значение - по правому краю;
- **left** — значение и его знак выравниваются по левому краю;
- **right** — значение и его знак выравниваются по правому краю;
- **setfill(char)** — задаем символ-заполнитель;
- **setw(int)** — задаем ширину поля.

Методы:

- **fill()** — возвращаем текущий символ-заполнитель;
- **fill(char)** — задаем новый символ-заполнитель;
- **width()** — возвращаем текущую ширину поля;
- **width(int)** — задаем ширину поля.

Чтобы использовать любой из вышеперечисленных объектов, нужно сначала установить ширину поля. Это делается с помощью метода `width(int)` или манипулятора setw(). Обратите внимание, по умолчанию при использовании ширины поля значения выравниваются по правому краю. Например:

```
1. #include <iostream>
2. #include <iomanip> // для setw()
3.
4. int main()
5. {
6.     std::cout << -
   12345 << '\n'; // выводим значение без использования ширины поля
7.     std::cout << std::setw(10) << -
   12345 << '\n'; // выводим значение с использованием ширины поля
8.     std::cout << std::setw(10) << std::left << -
   12345 << '\n'; // выравниваем по левому краю
```

```
    std::cout << std::setw(10) << std::right << -12345 << '\n'; // выравниваем по правому краю
    std::cout << std::setw(10) << std::internal << -12345 << '\n'; // знак значения выравнивается по левому краю, а само значение - по правому
}
```

Результат:

```
-12345
    -12345
-12345
    -12345
-    12345
```

Теперь давайте зададим свой собственный символ-заполнитель:

```
#include <iostream>
#include <iomanip> // для setw()

int main()
{
    std::cout.fill('*');
    std::cout << -12345 << '\n'; // выводим значение без использования ширины поля
    std::cout << std::setw(10) << -12345 << '\n'; // выводим значение с использованием ширины поля
    std::cout << std::setw(10) << std::left << -12345 << '\n'; // выравниваем по левому краю
    std::cout << std::setw(10) << std::right << -12345 << '\n'; // выравниваем по правому краю
    std::cout << std::setw(10) << std::internal << -12345 << '\n'; // знак значения выравнивается по левому краю, а само значение - по правому
}
```

Результат:

```
-12345
****-12345
-12345****
****-12345
-****12345
```

Обратите внимание, всё пустое пространство вокруг чисел заполнено `*` (символом-заполнителем).

Класс `ostream` и библиотека iostream содержат и другие полезные функции, флаги и манипуляторы. Но, как и в случае с классом `istream`, рассмотреть их все в рамках данного урока мы не можем. Однако основной функционал и общее представление вы получили.

Урок №218. Потоковые классы и Строки

В Стандартной библиотеке C++ есть отдельный набор классов, которые позволяют использовать уже знакомые нам операторы вставки (<<) и извлечения (>>) со строками.

Потоковые классы

Как и istream с ostream, так и потоковые классы для строк предоставляют буфер для хранения данных. Однако в отличие от cin и cout, эти потоковые классы не подключены к каналу ввода/вывода (т.е. к клавиатуре, монитору и т.д.).

Есть 6 потоковых классов, которые используются для чтения и записи строк:

- класс istringstream (является дочерним классу istream);
- класс ostringstream (является дочерним классу ostream);
- класс stringstream (является дочерним классу iostream);
- класс wistringstream;
- класс wostringstream;
- класс wstringstream.

Чтобы использовать **класс stringstream**, нужно подключить заголовочный файл sstream. Чтобы добавить данные в stringstream, мы можем использовать оператор вставки (<<):

```
1. #include <iostream>
2. #include <sstream> // для stringstream
3.
4. int main()
5. {
6.     std::stringstream myString;
7.     myString << "Lorem ipsum!" << std::endl; // вставляем "Lorem ipsum!" в
   stringstream
8. }
```

Либо **функцию str(string)**:

```
1. #include <iostream>
2. #include <sstream> // для stringstream
3.
4. int main()
5. {
6.     std::stringstream myString;
7.     myString.str("Lorem ipsum!"); // присваиваем буферу stringstream значение
   "Lorem ipsum!"
8. }
```

Аналогично, чтобы получить данные обратно из stringstream, мы можем использовать функцию str():

```
#include <iostream>
#include <sstream> // для stringstream

int main()
{
    std::stringstream myString;
    myString << "336000 12.14" << std::endl;
    std::cout << myString.str();
}
```

Результат:

```
336000 12.14
```

Либо оператор извлечения (>>):

```
#include <iostream>
#include <sstream> // для stringstream

int main()
{
    std::stringstream myString;
    myString << "336000 12.14"; // вставляем (числовую) строку в поток

    std::string part1;
    myString >> part1;

    std::string part2;
    myString >> part2;

    // выводим числа
    std::cout << part1 << " and " << part2 << std::endl;
}
```

Результат:

```
336000 and 12.14
```

Обратите внимание, оператор извлечения (>>) перебирает буфер данных по значению, учитывая пробелы между ними (т.е. одно использование оператора извлечения (>>) равно одному значению из буфера). В то время, как функция str() возвращает все данные из потока (не частично, а полностью), даже если перед ней использовался оператор извлечения.

Конвертация строк в числа и наоборот

Мы можем использовать операторы вставки и извлечения со строками для их конвертации в числа и наоборот.

Например, конвертация чисел в строки:

```
#include <iostream>
#include <sstream> // для stringstream

int main()
{
    std::stringstream myString;

    int nValue = 336000;
    double dValue = 12.14;
    myString << nValue << " " << dValue;

    std::string strValue1, strValue2;
    myString >> strValue1 >> strValue2;

    std::cout << strValue1 << " " << strValue2 << std::endl;
}
```

Результат:

```
336000 12.14
```

А теперь конвертация (числовой) строки обратно в числа:

```
#include <iostream>
#include <sstream> // для stringstream

int main()
{
    std::stringstream myString;
    myString << "336000 12.14"; // вставляем (числовую) строку в поток
    int nValue;
    double dValue;

    myString >> nValue >> dValue;

    std::cout << nValue << " " << dValue << std::endl;
}
```

Результат:

```
336000 12.14
```

Очистка stringstream для повторного использования

Есть несколько способов очистить буфер stringstream:

***Способ №1**:* Использовать функцию str() с пустой строкой C-style:

```
#include <iostream>
#include <sstream> // для stringstream

int main()
{
```

```
6.      std::stringstream myString;
7.      myString << "Hello ";
8.
9.      myString.str(""); // очищаем буфер
10.
11.     myString << "World!";
12.     std::cout << myString.str();
13. }
```

Способ №2: Использовать функцию str() с пустым объектом std::string:

```
1.  #include <iostream>
2.  #include <sstream> // для stringstream
3.
4.  int main()
5.  {
6.      std::stringstream myString;
7.      myString << "Hello ";
8.
9.      myString.str(std::string()); // очищаем буфер
10.
11.     myString << "World!";
12.     std::cout << myString.str();
13. }
```

Результат выполнения вышеприведенных программ:

```
World!
```

При очистке stringstream неплохой идеей является вызов функции clear():

```
1.  #include <iostream>
2.  #include <sstream> // для stringstream
3.
4.  int main()
5.  {
6.      std::stringstream myString;
7.      myString << "Hello ";
8.
9.      myString.str(""); // очищаем буфер
10.     myString.clear(); // сбрасываем все флаги ошибок
11.
12.     myString << "World!";
13.     std::cout << myString.str();
14. }
```

Функция clear() сбрасывает все флаги ошибок, которые были ранее установлены, и возвращает поток обратно в его прежнее (без ошибок) состояние. Мы поговорим подробнее о состояниях потока и флагах ошибок на следующем уроке.

Урок №219. Состояния потока и валидация пользовательского ввода

Класс ios_base содержит следующие **флаги для обозначения состояния потоков**:

- **goodbit** — всё хорошо;
- **badbit** — произошла какая-то фатальная ошибка (например, программа попыталась прочитать данные после конца файла);
- **eofbit** — поток достиг конца файла;
- **failbit** — произошла какая-то НЕ фатальная ошибка (например, пользователь ввел буквы, когда программа ожидала числа).

Хотя эти флаги находятся в `ios_base`, но, поскольку ios является дочерним классом для `ios_base`, доступ к этим флагам также возможен и через `ios` (например, как `std::ios::failbit`).

`ios` также предоставляет **ряд методов для доступа к вышеперечисленным состояниям потока**:

- **good()** — возвращает `true`, если установлен goodbit (значит, что с потоком всё ок);
- **bad()** — возвращает `true`, если установлен badbit (значит, что произошла какая-то фатальная ошибка);
- **eof()** — возвращает `true`, если установлен eofbit (значит, что поток находится в конце файла);
- **fail()** — возвращает `true`, если установлен failbit (значит, что произошла какая-то НЕ фатальная ошибка);
- **clear()** — сбрасывает все текущие флаги состояния потока и задает ему goodbit;
- **clear(state)** — сбрасывает все текущие флаги состояния потока и устанавливает флаг, переданный в качестве параметра;
- **rdstate()** — возвращает текущие установленные флаги;
- **setstate(state)** — устанавливает флаг состояния, переданный в качестве параметра.

Чаще всего мы будем иметь дело с failbit, который срабатывает при некорректном пользовательском вводе. Например:

```
1. #include <iostream>
2.
3. int main()
```

```
4. {
5.     std::cout << "Enter your age: ";
6.     int nAge;
7.     std::cin >> nAge;
8. }
```

Обратите внимание, эта программа ожидает от пользователя ввод целого числа. Однако, если пользователь введет что-либо другое (например, `Tom`), то cin не сможет извлечь что-либо в `nAge`, и для потока будет установлен флаг failbit.

Если же возникает ошибка, и для потока задан какой-либо другой флаг (отличный от goodbit), то дальнейшие операции с этим потоком будут проигнорированы. Это можно исправить, вызвав метод clear().

Валидация пользовательского ввода

Валидация пользовательского ввода — это процесс проверки того, соответствует ли пользовательский ввод заданным критериям. Обычно, валидация ввода бывает числовой и строковой.

Со **строковой валидацией** мы принимаем весь пользовательский ввод в качестве строки, а затем либо принимаем эту строку, либо отклоняем её (в зависимости от критериев проверки). Например, если мы просим пользователя ввести номер телефона, то мы должны убедиться, что этот номер состоит из 10 цифр. В большинстве языков (особенно в скриптовых, таких как Perl и PHP) это можно сделать с помощью регулярных выражений. В языке C++ нет встроенной поддержки регулярных выражений (возможно, это добавят в следующих версиях языка C++), поэтому обычно это делается путем проверки каждого символа строки на соответствие заданным критериям.

С **числовой валидацией** мы обычно заботимся о том, чтобы число, которое ввел пользователь, находилось в определенном диапазоне (например, от 0 до 20). Однако в отличие от строковой валидации, пользователь может ввести данные, которые вообще не являются числами, а нам нужно будет обрабатывать и такие случаи.

Для решения этой проблемы C++ предоставляет ряд полезных функций, которые мы можем использовать для определения того, являются ли конкретные символы цифрами или буквами. Следующие функции находятся в заголовочном файле cctype:

- **функция isalnum(int)** — возвращает ненулевое значение, если параметром является буква или цифра;

- **функция isalpha(int)** — возвращает ненулевое значение, если параметром является буква;
- **функция iscntrl(int)** — возвращает ненулевое значение, если параметром является управляющий символ;
- **функция isdigit(int)** — возвращает ненулевое значение, если параметром является цифра;
- **функция isgraph(int)** — возвращает ненулевое значение, если параметром является выводимый символ (но не пробел);
- **функция isprint(int)** — возвращает ненулевое значение, если параметром является выводимый символ, включая пробел;
- **функция ispunct(int)** — возвращает ненулевое значение, если параметром не являются ни буква, ни цифра, ни пробел;
- **функция isspace(int)** — возвращает ненулевое значение, если параметром является пробел;
- **функция isxdigit(int)** — возвращает ненулевое значение, если параметром является шестнадцатеричная цифра (`0-9`, `a-f`, `A-F`).

Строковая валидация

Давайте выполним простую строковую валидацию, попросив пользователя ввести свое имя. Наши ограничения: имя может содержать только буквы и пробелы. Если пользователь введет что-либо лишнее, то ввод отклоняется. Перебирать и проверять мы будем каждый символ пользовательского ввода:

```
#include <iostream>
#include <cctype>
#include <string>

int main()
{
    while (1)
    {
        // Просим пользователя ввести свое имя
        std::cout << "Enter your name: ";
        std::string strName;
        std::getline(std::cin, strName); // извлекаем целую строку, включая пробелы

        bool bRejected = false;

        // Перебираем каждый символ строки до тех пор, пока не дойдем до конца строки или до отклонения символа
        for (unsigned int nIndex = 0; nIndex < strName.length() && !bRejected; ++nIndex)
        {
            // Если текущий символ является буквой, то всё ок
            if (isalpha(strName[nIndex]))
                continue;

```

```
            // Если пробел, то тоже ок
            if (strName[nIndex] == ' ')
                continue;

            // В противном случае, отклоняем весь пользовательский ввод
            bRejected = true;
        }

        // Если пользовательский ввод был принят, то мы выходим из цикла while, и программа завершает свое выполнение.
        // В противном случае, мы просим пользователя ввести свое имя еще раз
        if (!bRejected)
            break;
    }
}
```

Обратите внимание, этот код не идеален: пользователь может ввести в качестве своего имени `djskbvjdb jdhsbj js` или вообще одни пробелы. Мы можем усилить валидацию, уточнив наши критерии проверки: имя пользователя должно содержать как минимум 1 символ и не более одного пробела.

Теперь рассмотрим другой случай, когда мы просим пользователя ввести свой номер телефона. В отличие от имени пользователя, номер телефона имеет фиксированную длину. Следовательно, мы будем использовать другой подход к валидации пользовательского ввода. Мы напишем функцию, которая будет проверять номер телефона, который ввел пользователь, на соответствие заранее определенному шаблону (такой вот своеобразный аналог регулярным выражениям).

Шаблон будет работать следующим образом:

- `#` — любая цифра в пользовательском вводе;
- `@` — любая буква в пользовательском вводе;
- `_` — любой пробел в пользовательском вводе;
- `?` — вообще любой символ.

Все символы пользовательского ввода и нашего шаблона должны точно совпадать.

Итак, если мы хотим, чтобы пользовательский ввод соответствовал шаблону `(###) ###-####`, то пользователь должен ввести: `(` три цифры `)`, пробел, три цифры, тире и еще четыре цифры. Если что-то из этого не совпадет, то пользовательский ввод будет отклонен, например:

```
#include <iostream>
#include <string>

bool InputMatches(std::string strUserInput, std::string strTemplate)
{
```

```
    if (strTemplate.length() != strUserInput.length())
        return false;

    // Перебираем каждый символ пользовательского ввода
    for (unsigned int nIndex = 0; nIndex < strTemplate.length(); nIndex++)
    {
        switch (strTemplate[nIndex])
        {
        case '#': // = цифра
            if (!isdigit(strUserInput[nIndex]))
                return false;
            break;
        case '_': // = пробел
            if (!isspace(strUserInput[nIndex]))
                return false;
            break;
        case '@': // = буква
            if (!isalpha(strUserInput[nIndex]))
                return false;
            break;
        case '?': // = вообще любой символ
            break;
        default: // = точное совпадение с символом
            if (strUserInput[nIndex] != strTemplate[nIndex])
                return false;
        }
    }

    return true;
}

int main()
{
    std::string strValue;

    while (1)
    {
        std::cout << "Enter a phone number (###) ###-####: ";
        std::getline(std::cin, strValue); // извлекаем целую строку, включая пробелы
        if (InputMatches(strValue, "(###) ###-####"))
            break;
    }

    std::cout << "You entered: " << strValue << std::endl;
}
```

Используя эту функцию, мы можем заставить пользователя ввести свой номер телефона точно по заданному нами шаблону. Но это не панацея на все случаи жизни.

Числовая валидация

При работе с числовым вводом очевидным путем развития событий является использование оператора извлечения для конвертации пользовательского ввода в числовой тип. Проверяя failbit, мы можем сказать, ввел ли пользователь число или нет.

Например:

```
#include <iostream>

int main()
{
    int nAge;

    while (1)
    {
        std::cout << "Enter your age: ";
        std::cin >> nAge;

        if (std::cin.fail()) // если никакого извлечения не произошло
        {
            std::cin.clear(); // то сбрасываем все текущие флаги состояния и устанавливаем goodbit, чтобы иметь возможность использовать функцию ignore()
            std::cin.ignore(32767, '\n'); // очищаем поток от мусора
            continue; // просим пользователя ввести свой возраст еще раз
        }

        if (nAge <= 0) // убеждаемся, что nAge является положительным числом
            continue;

        break;
    }

    std::cout << "You entered: " << nAge << std::endl;
}
```

Если пользователь ввел число, то `cin.fail()` будет `false`, выполнится оператор break, и мы выйдем из цикла while. Если же пользователь ввел букву, то `cin.fail()` будет `true`, и пользователю снова будет предложено ввести свой возраст.

Однако, есть один нюанс: если пользователь введет строку, которая начинается с цифр, но затем содержит буквы (например, `53qwerty74`), то первые цифры (`53`) будут извлечены в `nAge`, а остаток строки (`qwerty74`) останется во входном потоке, и failbit при этом НЕ будет установлен. Это грозит наличием мусора во входном потоке при следующем извлечении.

Давайте решим эту проблему:

```
#include <iostream>

int main()
{
    int nAge;

    while (1)
    {
        std::cout << "Enter your age: ";
        std::cin >> nAge;

        if (std::cin.fail()) // если никакого извлечения не произошло
```

```
13.         {
14.             std::cin.clear(); // то сбрасываем все текущие флаги состояния и
    устанавливаем goodbit, чтобы иметь возможность использовать функцию ignore()
15.             std::cin.ignore(32767, '\n'); // очищаем поток от мусора
16.             continue; // просим пользователя ввести свой возраст еще раз
17.         }
18.
19.         std::cin.ignore(32767, '\n'); // очищаем весь мусор, который остался в
    потоке после извлечения
20.         if (std::cin.gcount() > 1) // если мы очистили более одного символа
21.             continue; // то этот ввод считается некорректным, и мы просим
    пользователя ввести свой возраст еще раз
22.
23.         if (nAge <= 0) // убеждаемся, что nAge является положительным числом
24.             continue;
25.
26.         break;
27.     }
28.
29.     std::cout << "You entered: " << nAge << std::endl;
30. }
```

Числовая валидация с помощью строки

Вышеприведенный пример потребовал немало усилий, чтобы получить одно простое значение! Другой способ обработки числового ввода заключается в том, чтобы прочитать пользовательский ввод как строку, обработать его как строку и, если он пройдет проверку, конвертировать эту строку в числовой тип. Например:

```
1.  #include <iostream>
2.  #include <sstream> // для stringstream
3.
4.  int main()
5.  {
6.      int nAge;
7.
8.      while (1)
9.      {
10.         std::cout << "Enter your age: ";
11.         std::string strAge;
12.         std::cin >> strAge;
13.
14.         // Убеждаемся, что каждый символ является цифрой
15.         bool bValid = true;
16.         for (unsigned int nIndex = 0; nIndex < strAge.length(); nIndex++)
17.             if (!isdigit(strAge[nIndex]))
18.             {
19.                 bValid = false;
20.                 break;
21.             }
22.         if (!bValid)
23.             continue;
24.
25.         // На данный момент у нас есть что-то, что мы можем конвертировать в
    число, поэтому мы используем stringstream для выполнения конвертации
26.         std::stringstream strStream;
27.         strStream << strAge;
28.         strStream >> nAge;
```

```
29.
30.         if (nAge <= 0) // убеждаемся, что nAge является положительным числом
31.             continue;
32.
33.         break;
34.     }
35.
36.     std::cout << "You entered: " << nAge << std::endl;
37. }
```

Будет ли этот вариант более эффективным, нежели прямое числовое извлечение, зависит от ваших параметров валидации и ограничений.

Как вы можете видеть, валидация пользовательского ввода в языке C++ занимает не так уж и мало времени и усилий. К счастью, многие подобные задачи (например, выполнение числовой валидации с помощью строк) можно легко превратить в функции, которые затем можно будет повторно использовать в других программах.

Урок №220. Базовый файловый ввод и вывод

Работа файлового ввода/вывода в языке C++ почти аналогична работе обычных потоков ввода/вывода (но с небольшими нюансами).

Есть **три основных класса файлового ввода/вывода в языке C++**:

- **ifstream** (является дочерним классу istream);
- **ofstream** (является дочерним классу ostream);
- **fstream** (является дочерним классу iostream).

С помощью этих классов можно выполнить однонаправленный файловый ввод, однонаправленный файловый вывод и двунаправленный файловый ввод/вывод. Для их использования нужно всего лишь подключить заголовочный файл fstream.

В отличие от потоков cout, cin, cerr и clog, которые сразу же можно использовать, файловые потоки должны быть явно установлены программистом. То есть, чтобы открыть файл для чтения и/или записи, нужно создать объект соответствующего класса файлового ввода/вывода, указав имя файла в качестве параметра. Затем, с помощью оператора вставки (`<<`) или оператора извлечения (`>>`), можно записывать данные в файл или считывать содержимое файла. После проделывания данных действий нужно закрыть файл — явно вызвать **метод close()** или просто позволить файловой переменной ввода/вывода выйти из области видимости (деструктор файлового класса ввода/вывода закроет этот файл автоматически вместо нас).

Файловый вывод

Для записи в файл используется класс `ofstream`. Например:

```
#include <iostream>
#include <fstream>
#include <cstdlib> // для использования функции exit()

int main()
{
    using namespace std;

    // Класс ofstream используется для записи данных в файл.
    // Создаем файл SomeText.txt
    ofstream outf("SomeText.txt");

    // Если мы не можем открыть этот файл для записи данных в него,
    if (!outf)
    {
        // то выводим сообщение об ошибке и выполняем функцию exit()
```

```
        cerr << "Uh oh, SomeText.txt could not be opened for writing!" << endl;
        exit(1);
    }

    // Записываем в файл следующие две строки
    outf << "See line #1!" << endl;
    outf << "See line #2!" << endl;

    return 0;

    // Когда outf выйдет из области видимости, то деструктор класса ofstream
    автоматически закроет наш файл
}
```

Если вы загляните в каталог вашего проекта (*ПКМ по вкладке с названием вашего файла .cpp в Visual Studio > "Открыть содержащую папку"*), то увидите файл с именем SomeText.txt, в котором находятся следующие строки:

```
See line #1!
See line #2!
```

Обратите внимание, мы также можем использовать **метод put()** для записи одного символа в файл.

Файловый ввод

Теперь мы попытаемся прочитать содержимое файла, который создали в предыдущем примере. Обратите внимание, `ifstream` возвратит `0`, если мы достигли конца файла (это удобно для определения «длины» содержимого файла). Например:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib> // для использования функции exit()

int main()
{
    using namespace std;

    // ifstream используется для чтения содержимого файла.
    // Попытаемся прочитать содержимое файла SomeText.txt
    ifstream inf("SomeText.txt");

    // Если мы не можем открыть этот файл для чтения его содержимого,
    if (!inf)
    {
        // то выводим следующее сообщение об ошибке и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened for reading!" << endl;
        exit(1);
    }

    // Пока есть данные, которые мы можем прочитать,
    while (inf)
```

```
    {
        // то перемещаем эти данные в строку, которую затем выводим на экран
        string strInput;
        inf >> strInput;
        cout << strInput << endl;
    }

    return 0;

    // Когда inf выйдет из области видимости, то деструктор класса ifstream
   автоматически закроет наш файл
}
```

Результат выполнения программы:

```
See
line
#1!
See
line
#2!
```

Хм, это не совсем то, что мы хотели. Как мы уже узнали на предыдущих уроках, оператор извлечения работает с «отформатированными данными», т.е. он игнорирует все пробелы, символы табуляции и символ новой строки. Чтобы прочитать всё содержимое как есть, без его разбивки на части (как в примере, приведенном выше), нам нужно использовать **метод getline()**:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib> // для использования функции exit()

int main()
{
    using namespace std;

    // ifstream используется для чтения содержимого файлов.
    // Мы попытаемся прочитать содержимое файла SomeText.txt
    ifstream inf("SomeText.txt");

    // Если мы не можем открыть файл для чтения его содержимого,
    if (!inf)
    {
        // то выводим следующее сообщение об ошибке и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened for reading!" << endl;
        exit(1);
    }

    // Пока есть, что читать,
    while (inf)
    {
        // то перемещаем то, что можем прочитать, в строку, а затем выводим эту
   строку на экран
        string strInput;
        getline(inf, strInput);
```

```
        cout << strInput << endl;
    }

    return 0;

    // Когда inf выйдет из области видимости, то деструктор класса ifstream
    автоматически закроет наш файл
}
```

Результат выполнения программы:

```
See line #1!
See line #2!
```

Буферизованный вывод

Вывод в языке C++ может быть буферизован. Это означает, что всё, что выводится в файловый поток, не может сразу же быть записанным на диск (в конкретный файл). Это сделано, в первую очередь, по соображениям производительности. Когда данные буфера записываются на диск, то это называется **очисткой буфера**. Одним из способов очистки буфера является закрытие файла. В таком случае всё содержимое буфера будет перемещено на диск, а затем файл будет закрыт.

Буферизация вывода обычно не является проблемой, но при определенных обстоятельствах она может вызвать проблемы у неосторожных новичков. Например, когда в буфере хранятся данные, а программа преждевременно завершает свое выполнение (либо в результате сбоя, либо путем вызова функции exit()). В таких случаях деструкторы классов файлового ввода/вывода не выполняются, файлы никогда не закрываются, буферы не очищаются и наши данные теряются навсегда. Вот почему хорошей идеей является явное закрытие всех открытых файлов перед вызовом функции exit().

Также буфер можно очистить вручную, используя **метод ostream::flush()** или отправив **std::flush** в выходной поток. Любой из этих способов может быть полезен для обеспечения немедленной записи содержимого буфера на диск в случае сбоя программы.

> ***Интересный нюанс***: Поскольку `std::endl;` также очищает выходной поток, то его чрезмерное использование (приводящее к ненужным очисткам буфера) может повлиять на производительность программы (так как очистка буфера в некоторых случаях может быть затратной операцией). По этой причине программисты, которые заботятся о производительности своего кода, часто используют `\n` вместо `std::endl` для вставки символа новой строки в выходной поток, дабы избежать ненужной очистки буфера.

Режимы открытия файлов

Что произойдет, если мы попытаемся записать данные в уже существующий файл? Повторный запуск вышеприведенной программы (самая первая) показывает, что исходный файл полностью перезаписывается при повторном запуске программы. А что, если нам нужно добавить данные в конец файла? Оказывается, конструкторы файлового потока принимают необязательный второй параметр, который позволяет указать программисту способ открытия файла. В качестве этого параметра можно передавать **следующие флаги** (которые находятся в классе `ios`):

- **app** — открывает файл в режиме добавления;
- **ate** — переходит в конец файла перед чтением/записью;
- **binary** — открывает файл в бинарном режиме (вместо текстового режима);
- **in** — открывает файл в режиме чтения (по умолчанию для `ifstream`);
- **out** — открывает файл в режиме записи (по умолчанию для `ofstream`);
- **trunc** — удаляет файл, если он уже существует.

Можно указать сразу несколько флагов путем использования побитового ИЛИ (|).

- `ifstream` по умолчанию работает в режиме ios::in;
- `ofstream` по умолчанию работает в режиме ios::out;
- `fstream` по умолчанию работает в режиме ios::in ИЛИ ios::out, что означает, что вы можете выполнять как чтение содержимого файла, так и запись данных в файл.

Теперь давайте напишем программу, которая добавит две строки в ранее созданный нами файл SomeText.txt:

```
#include <iostream>
#include <cstdlib> // для использования функции exit()
#include <fstream>

int main()
{
    using namespace std;

    // Передаем флаг ios:app, чтобы сообщить fstream, что мы собираемся
    добавить свои данные к уже существующим данным файла.
    // Мы не собираемся перезаписывать файл.
    // Нам не нужно передавать флаг ios::out, поскольку ofstream по умолчанию
    работает в режиме ios::out
    ofstream outf("SomeText.txt", ios::app);

    // Если мы не можем открыть файл для записи данных,
    if (!outf)
    {
        // то выводим следующее сообщение об ошибке и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened for writing!" << endl;
```

```
        exit(1);
    }

    outf << "See line #3!" << endl;
    outf << "See line #4!" << endl;

    return 0;

    // Когда outf выйдет из области видимости, то деструктор класса ofstream
    автоматически закроет наш файл
}
```

Теперь, если мы посмотрим содержимое SomeText.txt (запустим одну из вышеприведенных программ для чтения файла или откроем этот файл в каталоге проекта), то увидим следующее:

```
See line #1!
See line #2!
See line #3!
See line #4!
```

Явное открытие файлов с помощью функции open()

Точно так же, как мы явно закрываем файл с помощью метода close(), мы можем явно открывать файл с помощью **функции open()**. Функция open() работает аналогично конструкторам класса файлового ввода/вывода: принимает имя файла и режим (необязательно), в котором нужно открыть файл, в качестве параметров. Например:

```
#include <fstream>

int main()
{
    using namespace std;

    ofstream outf("SomeText.txt");
    outf << "See line #1!" << endl;
    outf << "See line #2!" << endl;
    outf.close(); // явно закрываем файл

    // Упс, мы кое-что забыли сделать
    outf.open("SomeText.txt", ios::app);
    outf << "See line #3!" << endl;
    outf.close();

    return 0;

    // Когда outf выйдет из области видимости, то деструктор класса ofstream
    автоматически закроет наш файл
}
```

Результат:

```
See line #1!
See line #2!
See line #3!
```

На этом всё. На следующем уроке мы рассмотрим рандомный файловый ввод/вывод.

Урок №221. Рандомный файловый ввод и вывод

Каждый класс файлового ввода/вывода содержит **файловый указатель**, который используется для отслеживания текущей позиции чтения/записи данных в файле. Любая запись в файл или чтение содержимого файла происходит в текущем местоположении файлового указателя. По умолчанию, при открытии файла для чтения или записи, файловый указатель находится в самом начале этого файла. Однако, если файл открывается в режиме добавления, то файловый указатель перемещается в конец файла, чтобы пользователь имел возможность добавить данные в файл, а не перезаписать его.

Рандомный доступ к файлам с помощью функций seekg() и seekp()

До этого момента мы осуществляли последовательный доступ к файлам, т.е. выполняли чтение/запись файла по порядку. Тем не менее, мы можем выполнить и **произвольный** (**рандомный**) **доступ** к файлу (т.е. перемещаться по файлу, как захотим). Это может быть полезно, когда файл имеет обширное содержимое, а нам нужна всего лишь небольшая конкретная запись из всего этого. Вместо последовательного доступа (когда мы переходим до нужной записи начиная с самого начала файла), мы можем осуществить непосредственный доступ к этой записи.

Рандомный доступ к файлу осуществляется путем манипулирования файловым указателем с помощью **функции seekg()** (окончание *"**g**"* = *"**g**et"*, т.е. *«получить/достать»*) — для ввода, и **функции seekp()** (окончание *"**p**"* = *"**p**ut"* (т.е. *«положить/поместить»*) — для вывода.

Функции seekg() и seekp() принимают следующие **два параметра**:

- **первый параметр** — это смещение на которое следует переместить файловый указатель (измеряется в байтах);
- **второй параметр** — это флаг ios, который обозначает место, от которого следует отталкиваться при выполнении смещения.

Флаги ios, которые принимают функции seekg() и seekp() в качестве второго параметра:

- **beg** — смещение относительно начала файла (по умолчанию);
- **cur** — смещение относительно текущего местоположения файлового указателя;

- **end** — смещение относительно конца файла.

Положительное смещение означает перемещение файлового указателя в сторону конца файла, тогда как отрицательное смещение означает перемещение файлового указателя в сторону начала файла. Например:

```
inf.seekg(15, ios::cur); // перемещаемся вперед на 15 байт относительно
   текущего местоположения файлового указателя
inf.seekg(-17, ios::cur); // перемещаемся назад на 17 байт относительно текущего
   местоположения файлового указателя
inf.seekg(24, ios::beg); // перемещаемся к 24-му байту относительно начала файла
inf.seekg(25); // перемещаемся к 25-му байту файла
inf.seekg(-27, ios::end); // перемещаемся к 27-му байту от конца файла
```

Перемещение в начало или в конец файла:

```
inf.seekg(0, ios::beg); // перемещаемся в начало файла
inf.seekg(0, ios::end); // перемещаемся в конец файла
```

Теперь давайте совместим функцию seekg() с файлом SomeText.txt, рассмотренном на предыдущем уроке.

Содержимое файла SomeText.txt:

```
See line #1!
See line #2!
See line #3!
See line #4!
```

Код программы:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib> // для использования функции exit()

int main()
{
    using namespace std;

    ifstream inf("SomeText.txt");

    // Если мы не можем открыть файл для чтения его содержимого,
    if (!inf)
    {
        // то выводим следующую ошибку и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened for reading!" << endl;
        exit(1);
    }

    string strData;

    inf.seekg(6); // перемещаемся к 6-му символу первой строки
    // Получаем остальную часть строки и выводим её на экран
```

```
    getline(inf, strData);
    cout << strData << endl;

    inf.seekg(9, ios::cur); // перемещаемся вперед на 9 байт относительно текущего местоположения файлового указателя
    // Получаем остальную часть строки и выводим её на экран
    getline(inf, strData);
    cout << strData << endl;

    inf.seekg(-14, ios::end); // перемещаемся на 14 байт назад относительно конца файла
    // Получаем остальную часть строки и выводим её на экран
    getline(inf, strData);
    cout << strData << endl;

    return 0;
}
```

Результат выполнения программы:

```
ne #1!
#2!
See line #4!
```

> ***Примечание***: В некоторых компиляторах реализация функций seekg() и tellg() при использовании с текстовыми файлами может иметь ошибки (из-за буферизации данных). Если ваш компилятор является одним из таковых (ваш результат будет отличаться от вышеприведенного результата), то вы можете попробовать открыть файл в бинарном режиме:
>
> ```
> ifstream inf("SomeText.txt", ifstream::binary);
> ```

Есть еще две другие полезные функции — **tellg()** и **tellp()**, которые возвращают абсолютную позицию файлового указателя. Это полезно при определении размера файла:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream inf("SomeText.txt");
    inf.seekg(0, std::ios::end); // перемещаемся в конец файла
    std::cout << inf.tellg();
}
```

Результат:

```
56
```

Это мы получили размер файла SomeText.txt в байтах.

Одновременное чтение и запись в файл с помощью fstream

Класс `fstream` (почти) способен одновременно читать содержимое файла и записывать данные в него! Нюанс заключается в том, что вы не можете произвольно переключаться между чтением и записью файла. Как только начнется чтение или запись файла, то единственным способом переключиться между чтением или записью будет выполнение операции, которая изменит текущее положение файлового указателя (например, поиск данных). Если вы не хотите перемещать файловый указатель (потому что он уже находится в нужном месте), то вы можете просто выполнить поиск текущих данных (на которые указывает файловый указатель):

```
// Предположим, что iofile является объектом класса fstream
iofile.seekg(iofile.tellg(), ios::beg); // перемещаемся к текущей позиции
   файлового указателя
```

Теперь давайте напишем программу, которая откроет файл, прочитает его содержимое и заменит все найденные гласные буквы на символ `#`:

```
#include <iostream>
#include <fstream>
#include <cstdlib> // для использования функции exit()

int main()
{
    using namespace std;

    // Мы должны указать как in, так и out, поскольку используем fstream
    fstream iofile("SomeText.txt", ios::in | ios::out);

    // Если мы не можем открыть iofile,
    if (!iofile)
    {
        // то выводим сообщение об ошибке и выполняем функцию exit()
        cerr << "Uh oh, SomeText.txt could not be opened!" << endl;
        exit(1);
    }

    char chChar;

    // Пока есть данные для обработки
    while (iofile.get(chChar))
    {
        switch (chChar)
        {
        // Если мы нашли гласную букву,
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'A':
        case 'E':
        case 'I':
```

```
        case 'O':
        case 'U':

            // то перемещаемся на один символ назад относительно текущего
    местоположения файлового указателя
            iofile.seekg(-1, ios::cur);

            // Поскольку мы выполнили операцию поиска, то теперь можем
    переключиться на запись данных в файл.
            // Заменим найденную гласную букву символом #
            iofile << '#';

            // Теперь нам нужно вернуться назад в режим чтения файла.
            // Выполняем функцию seekg() к текущей позиции
            iofile.seekg(iofile.tellg(), ios::beg);

            break;
        }
    }

    return 0;
}
```

Результат выполнения программы (содержимое файла SomeText.txt):

```
S## l#n# #1!
S## l#n# #2!
S## l#n# #3!
S## l#n# #4!
```

Другие полезные методы классов файлового ввода/вывода в языке C++:

- **remove()** — удаляет файл;
- **is_open()** — возвращает `true`, если поток в данный момент открыт, и `false` — если закрыт.

Предупреждение о записи указателей в файлы

Хотя записывать переменные в файл достаточно просто, всё становится немного сложнее, когда мы начинаем работать с указателями. Как мы уже знаем, указатель содержит лишь адрес переменной, на которую он указывает. Хотя эти адреса можно записывать в файл и считывать их из файла — это чревато неприятностями, так как адрес одной и той же переменной может отличаться при каждом повторном запуске программы. Следовательно, хотя переменная могла находиться по адресу `003AFCD4`, когда вы записывали этот адрес на диск (в какой-нибудь файл), при повторном запуске программы она уже может находиться по другому адресу!

Например, предположим, что у нас есть переменная `someValue` типа int, которая находится по адресу `003AFCD4`. Мы присваиваем `someValue` значение `7`. Затем

объявляем указатель `*pnValue`, который указывает на `someValue` (адрес `someValue` — `003AFCD4`). Мы записываем значение `7` и значение `pnValue` (`003AFCD4`) в какой-нибудь файл.

Через несколько недель мы снова запускаем эту программу и пытаемся извлечь значения из файла. Мы извлекаем значение `7` в переменную `someValue`, которая в текущей программе уже находится по адресу `0034FD90`. Дальше мы извлекаем адрес `003AFCD4` в указатель `*pnValue`. Поскольку `pnValue` указывает на `003AFCD4`, а `someValue` находится по адресу `0034FD90`, то `pnValue` больше не указывает на `someValue`, и попытка доступа к значению адреса, который хранит `pnValue`, приведет к неприятностям.

Правило: Не сохраняйте адреса переменных в файлах. Переменные, которые изначально были по одним адресам, при повторном запуске программы могут находиться уже по другим адресам.

Статические и динамические библиотеки

Библиотека — это «сборник» кода, который можно многократно использовать в самых разных программах. Как правило, **библиотека в языке C++ состоит из 2-х частей**:

- Заголовочный файл, который объявляет функционал библиотеки.
- Предварительно скомпилированный бинарный файл, содержащий реализацию функционала библиотеки.

Некоторые библиотеки могут быть разбиты на несколько файлов и/или иметь несколько заголовочных файлов.

Типы библиотек

Библиотеки предварительно компилируют по нескольким причинам. Во-первых, их код редко меняется. Было бы напрасной тратой времени повторно компилировать библиотеку каждый раз при её использовании в новой программе. Во-вторых, поскольку весь код предварительно скомпилирован в машинный язык, то это предотвращает получение доступа к исходному коду (и его изменение) сторонними лицами. Этот пункт важен для предприятий/людей, которые не хотят, чтобы результат их труда (исходный код) был доступен всем.

Есть 2 типа библиотек: статические и динамические.

Статическая библиотека (или ***«архив»***) состоит из подпрограмм, которые непосредственно компилируются и линкуются с вашей программой. При компиляции программы, которая использует статическую библиотеку, весь функционал статической библиотеки (тот, что использует ваша программа) становится частью вашего исполняемого файла. В Windows статические библиотеки имеют расширение **.lib** (сокр. от *«**lib**rary»*), тогда как в Linux статические библиотеки имеют расширение **.a** (сокр. от *«**a**rchive»*).

Одним из преимуществ статических библиотек является то, что вам нужно распространить всего лишь 1 (исполняемый) файл, чтобы пользователи могли запустить и использовать вашу программу. Поскольку статические библиотеки становятся частью вашей программы, то вы можете использовать их подобно функционалу своей собственной программы. С другой стороны, поскольку копия библиотеки становится частью каждого вашего исполняемого файла, то это может привести к увеличению размера файла. Также, если вам нужно будет обновить

статическую библиотеку, вам придется перекомпилировать каждый исполняемый файл, который её использует.

Динамическая библиотека (или ***«общая библиотека»***) состоит из подпрограмм, которые подгружаются в вашу программу во время её выполнения. При компиляции программы, которая использует динамическую библиотеку, эта библиотека не становится частью вашего исполняемого файла — она так и остается отдельным модулем. В Windows динамические библиотеки имеют расширение **.dll** (сокр. от *«**d**ynamic **l**ink **l**ibrary»* = *«библиотека динамической компоновки»*), тогда как в Linux динамические библиотеки имеют расширение **.so** (сокр. от *«**s**hared **o**bject»* = *«общий объект»*). Одним из преимуществ динамических библиотек является то, что разные программы могут совместно использовать одну копию динамической библиотеки, что значительно экономит используемое пространство. Еще одним преимуществом динамической библиотеки является то, что её можно обновить до более новой версии без необходимости перекомпиляции всех исполняемых файлов, которые её используют.

Поскольку динамические библиотеки не линкуются непосредственно с вашей программой, то ваши программы, использующие динамические библиотеки, должны явно подключать и взаимодействовать с динамической библиотекой. Этот механизм не всегда может быть понятен для новичков, что может затруднить взаимодействие с динамической библиотекой. Для упрощения этого процесса используют **библиотеки импорта**.

Библиотека импорта (англ. ***"import library"***) — это библиотека, которая автоматизирует процесс подключения и использования динамической библиотеки. В Windows это обычно делается через небольшую статическую библиотеку (.lib) с тем же именем, что и динамическая библиотека (.dll). Статическая библиотека линкуется с вашей программой во время компиляции, и тогда функционал динамической библиотеки может эффективно использоваться в вашей программе, как если бы это была обычная статическая библиотека. В Linux общий объектный файл (с расширением .so) дублируется сразу как динамическая библиотека и библиотека импорта. Большинство линкеров при создании динамической библиотеки автоматически создают к ней библиотеку импорта.

Установка библиотек

Теперь, когда мы уже разобрались с типами библиотек, давайте поговорим о том, как их использовать в наших программах.

Установка библиотеки в языке C++ состоит из 4-х последовательных шагов:

- ***Шаг №1:*** **Получите библиотеку**. Наилучшим вариантом является найти уже предварительно скомпилированный код (если он вообще существует) под вашу операционную систему, чтобы вам не пришлось компилировать библиотеку самостоятельно. В Windows библиотеки обычно распространяются в виде архивов (файлов .zip), а в Linux это пакеты кода (например, пакеты .rpm).
- ***Шаг №2:*** **Установите библиотеку**. В Linux это делается путем вызова менеджера пакетов, а дальше он всё делает сам. В Windows вам придется разархивировать библиотеку самостоятельно в любую выбранную вами папку. Рекомендуется хранить все используемые библиотеки в одном месте для быстрого доступа к ним. Например, создайте папку `Libs` (`C:\Libs`) и выделяйте для каждой (используемой вами) библиотеки свою отдельную подпапку.
- ***Шаг №3:*** **Убедитесь, что компилятор знает, где искать заголовочные файлы библиотеки**. В Windows это обычно подпапка `include` внутри основной папки библиотеки (например, если вы установили библиотеку в `C:\Libs\SDL-1.2.11`, то заголовочные файлы находятся в `C:\Libs\SDL-1.2.11\include`). В Linux библиотеки обычно устанавливаются в `/usr/include`. Однако, если файлы находятся в другом месте, вам нужно будет сообщить компилятору, где именно.
- ***Шаг №4:*** **Сообщите линкеру, где искать файлы с реализацией функционала библиотеки**. Аналогично с предыдущим шагом, вам нужно указать линкеру место, где находятся файлы с реализацией библиотеки. В Windows это обычно подпапка `\lib` внутри основной папки библиотеки (`C:\Libs\SDL-1.2.11\lib`), а в Linux это обычно `/usr/lib`.

Использование библиотек

Как только библиотека установлена и ваша IDE знает, где искать её файлы, то для того, чтобы вы могли использовать эту библиотеку в ваших проектах, вам необходимо выполнить следующие 3 шага:

- ***Шаг №5:*** **Если вы используете статические библиотеки или библиотеки импорта, сообщите линкеру, какие файлы библиотеки нужно связать с вашей программой**.
- ***Шаг №6:*** **Подключите заголовочные файлы библиотеки к вашей программе.**

- ***Шаг №7:*** **Если вы используете динамические библиотеки, то убедитесь, что исполняемые файлы будут иметь доступ к файлам библиотеки**. Самый простой способ использовать .dll — это скопировать .dll в папку с исполняемым файлом. Поскольку .dll-ка обычно распространяется вместе с исполняемым файлом, то это не составит труда.

Заключение

Шаги №3-№5 включают настройку вашей IDE. К счастью, почти все IDE работают одинаково, когда дело доходит до выполнения подобных задач. На следующем уроке мы рассмотрим, как выполнить данные шаги в Visual Studio.

Подключение и использование библиотек в Visual Studio

В качестве примера мы рассмотрим **подключение библиотеки SDL к нашему проекту в Visual Studio 2017 (работать будет и с более новыми версиями Visual Studio)**.

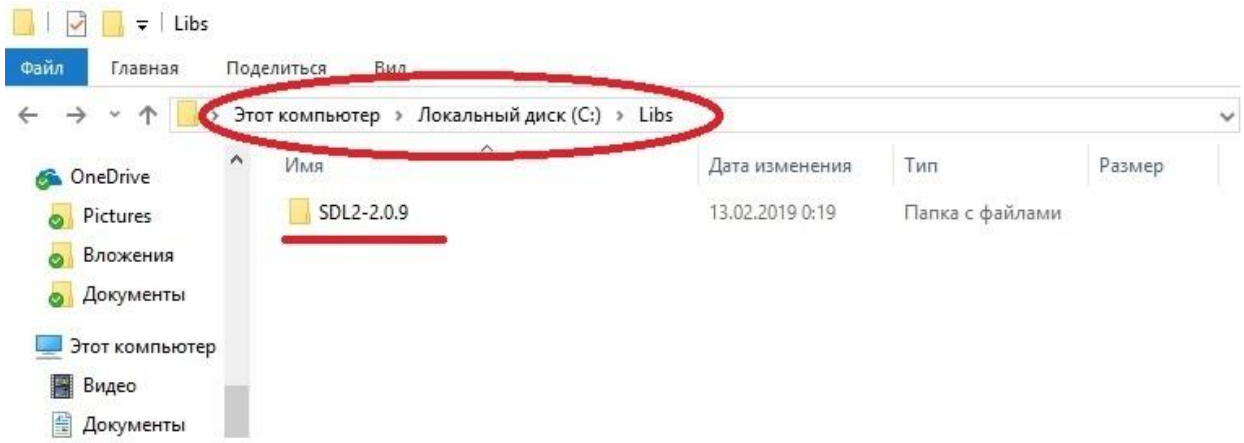
Шаг №1: Создаем папку для хранения библиотеки

Создаем папку `Libs` на диске C (`C:\Libs`).

Шаг №2: Скачиваем и устанавливаем библиотеку

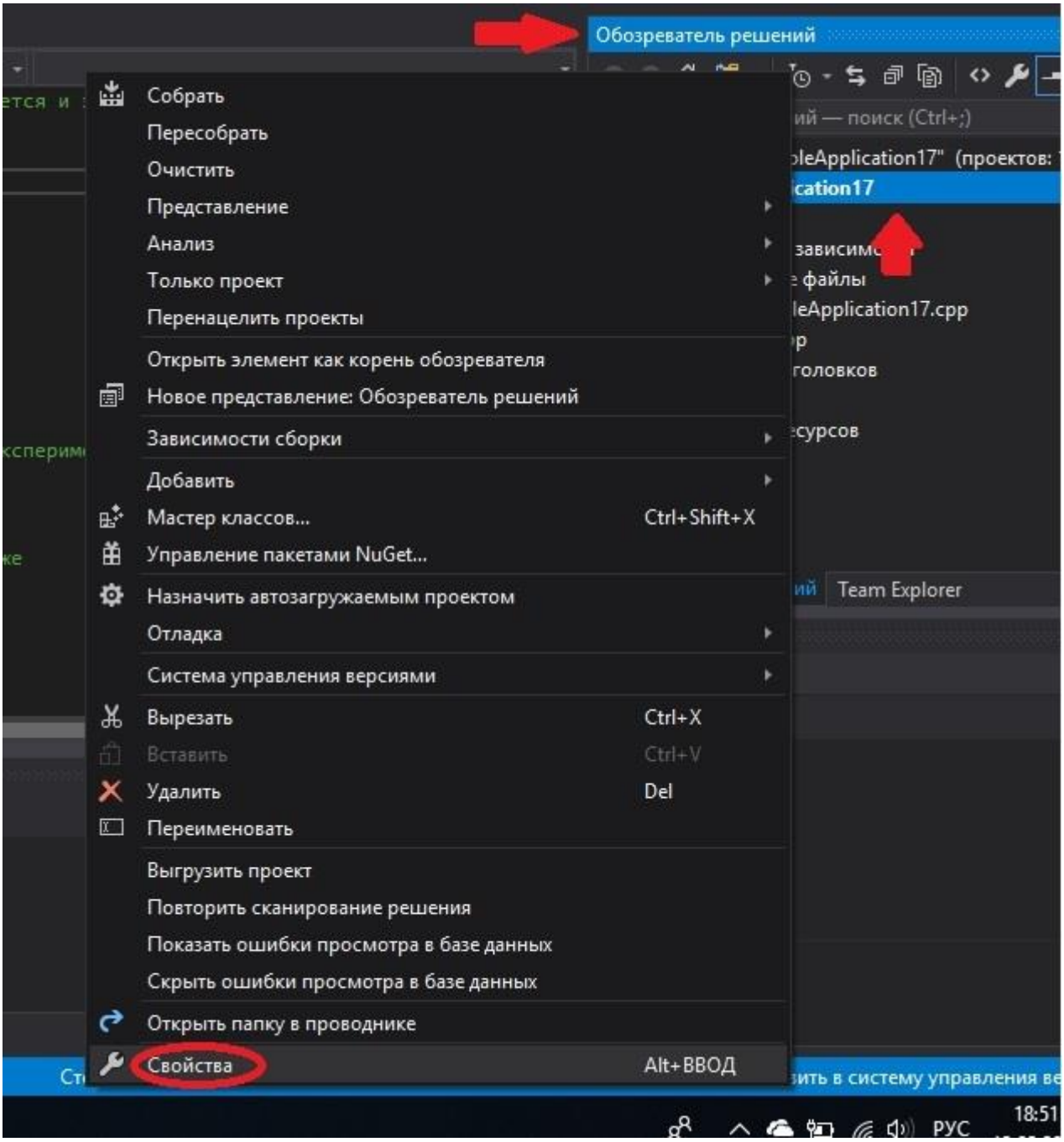
Заходим на сайт **https://www.libsdl.org/download-2.0.php**, пролистываем вниз до *"Development Libraries"* и скачиваем *SDL2-devel-2.0.9-VC.zip (Visual C++ 32/64-bit)*. После успешного скачивания нужно разархивировать этот архив в папку `Libs`.

Конечный результат:

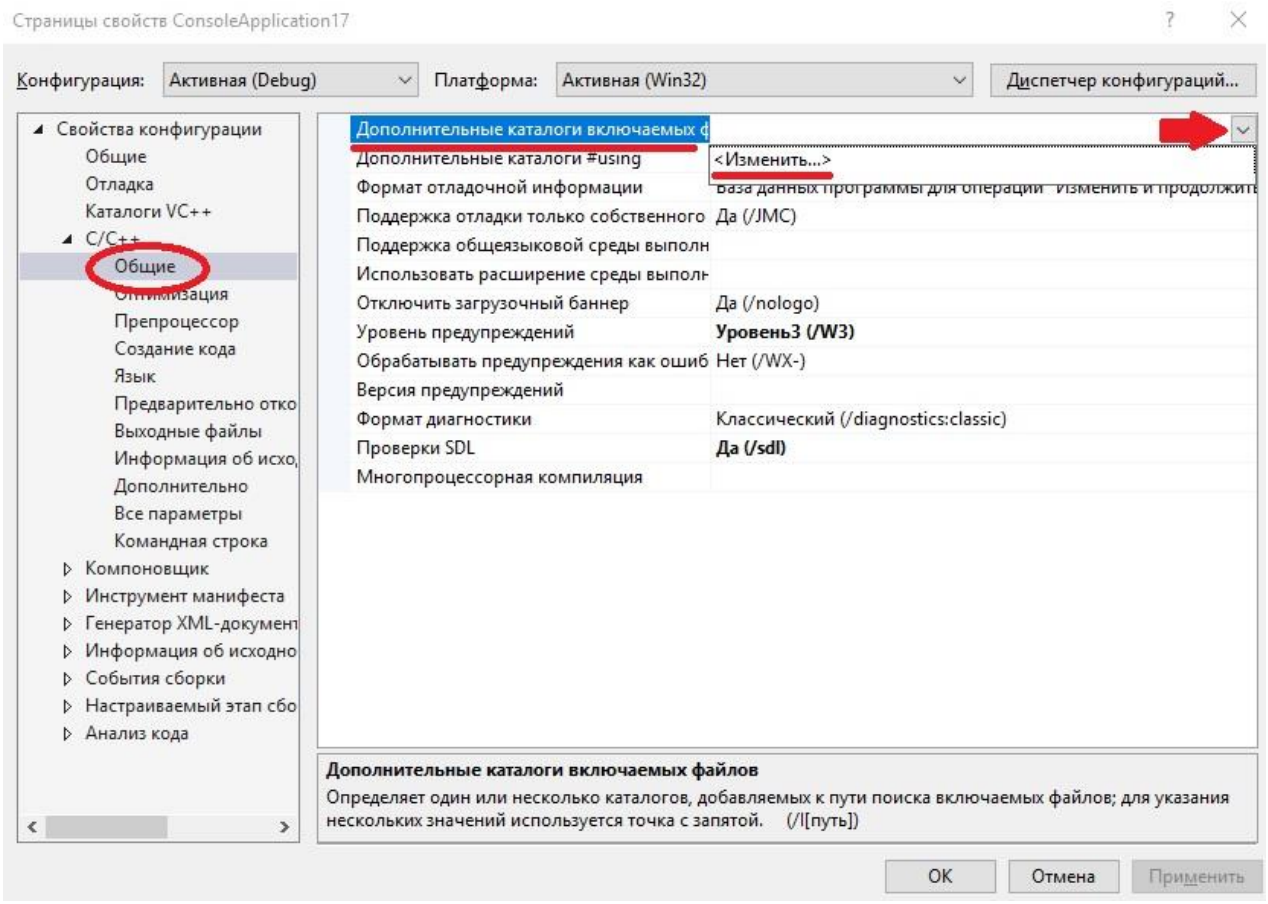


Шаг №3: Указываем путь к заголовочным файлам библиотеки

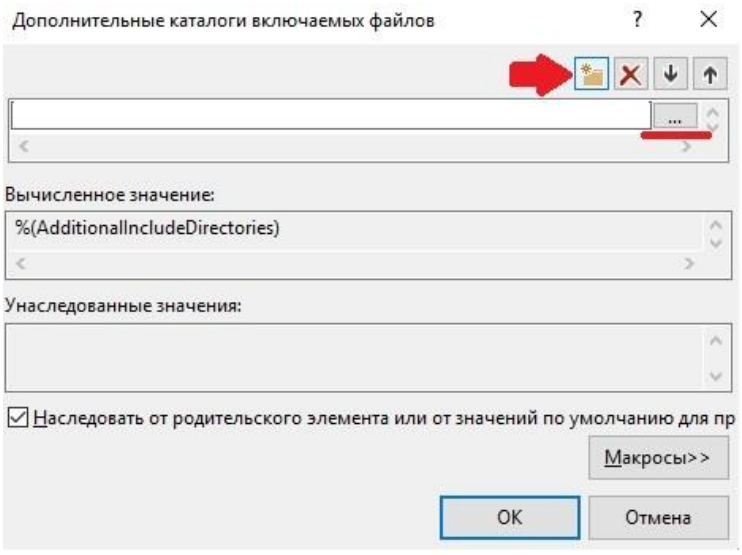
Открываем свой любой проект в Visual Studio или создаем новый, переходим в `"Обозреватель решений" > кликаем правой кнопкой мыши (ПКМ) по названию нашего проекта > "Свойства"`:



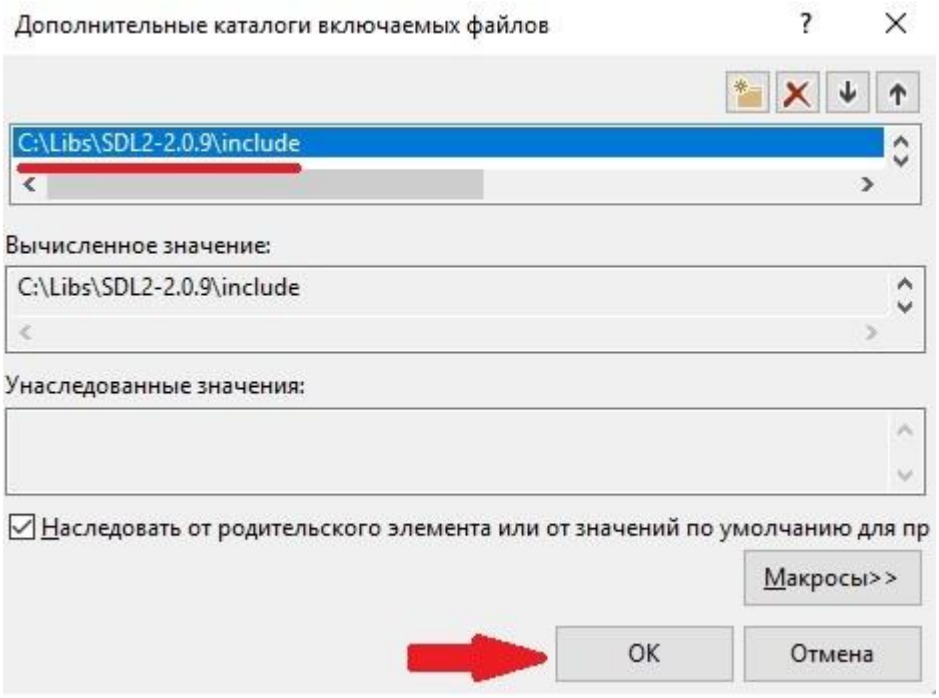
В `"Свойства конфигурации"` ищем вкладку `"C/C++" > "Общие"`. Затем выбираем пункт `"Дополнительные каталоги включаемых файлов" > нажимаем на стрелочку в конце > "Изменить"`:



В появившемся окне кликаем на иконку с изображением папки, а затем на появившееся троеточие:

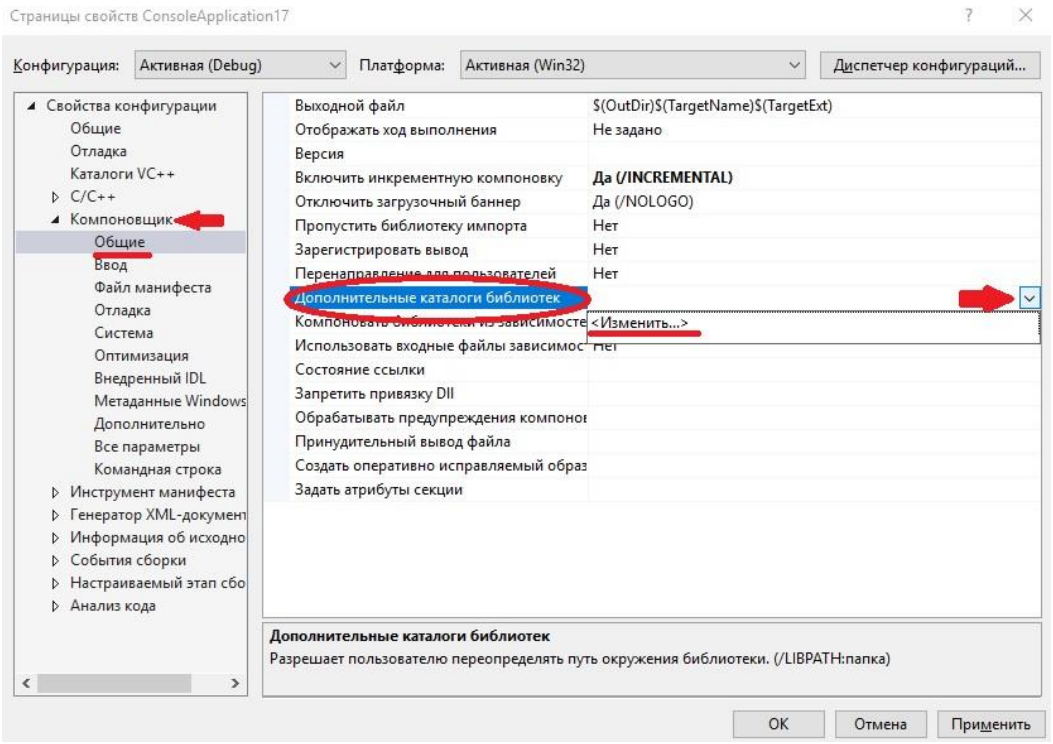


Заголовочные файлы находятся в папке `include` внутри нашей библиотеки, поэтому переходим в нее (`C:\Libs\SDL2-2.0.9\include`) и нажимаем `"Выбор папки"`, а затем `"OK"`:

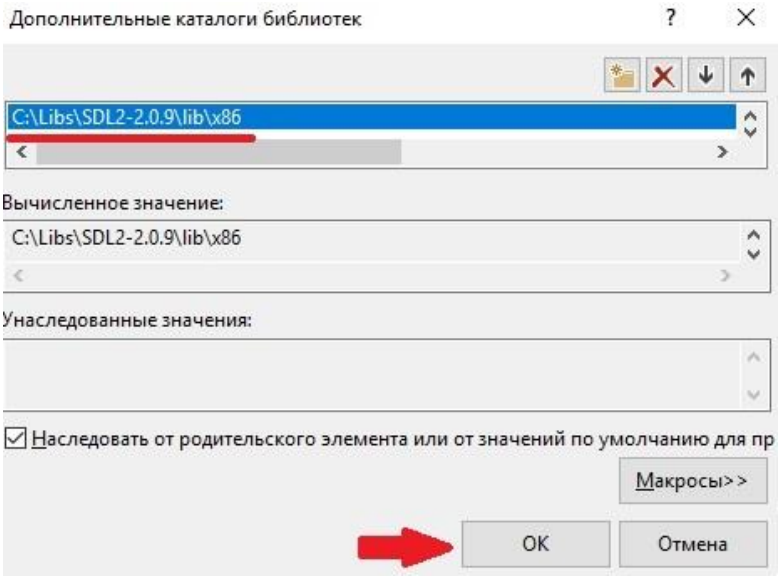


Шаг №4: Указываем путь к файлам с реализацией библиотеки

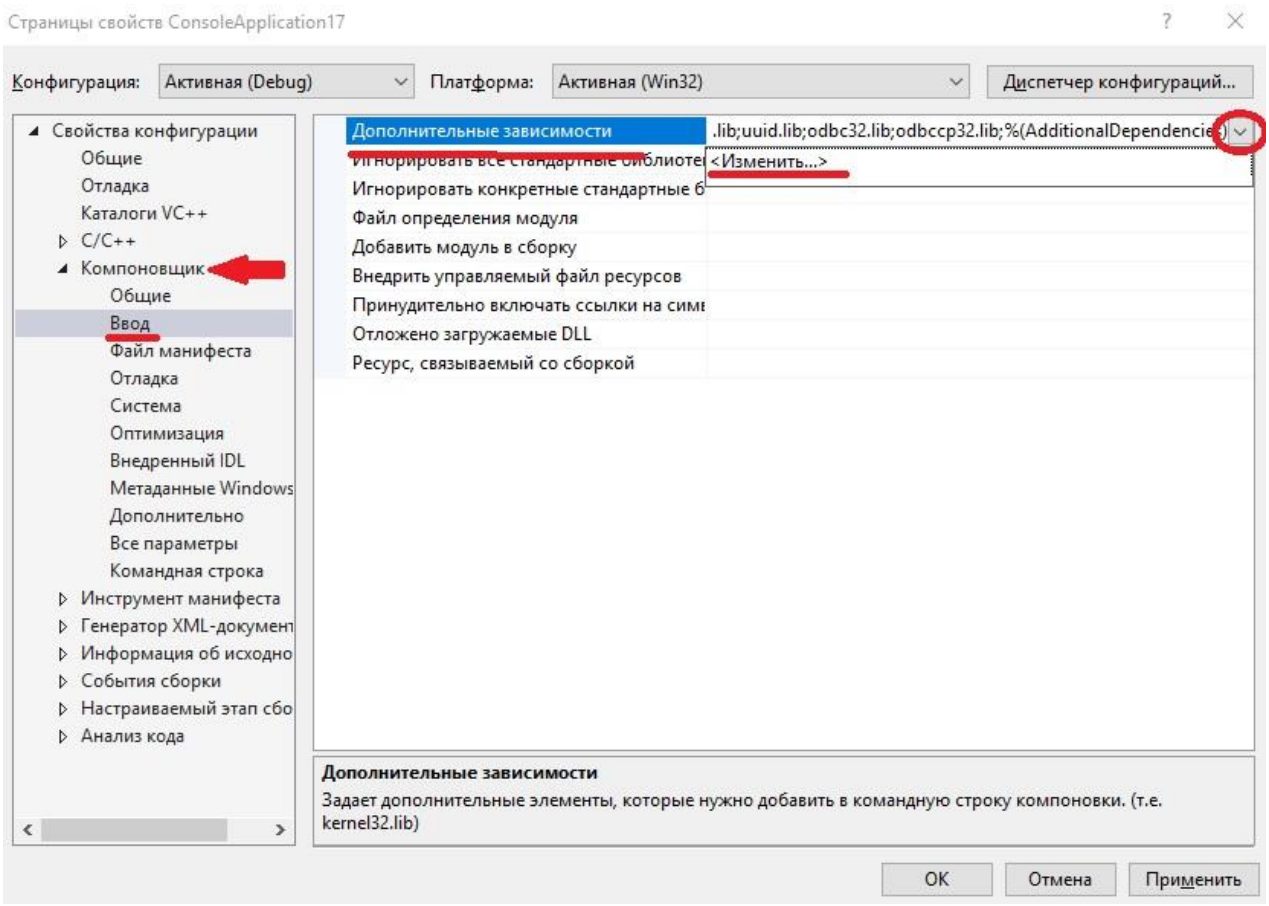
Переходим на вкладку `"Компоновщик" > "Общие"`. Ищем пункт `"Дополнительные каталоги библиотек" > стрелочку в конце > "Изменить"`:



Опять же, нажимаем на иконку с папкой, а затем на появившееся троеточие. Нам нужно указать следующий путь: `C:\Libs\SDL2-2.0.9\lib\x86`. Будьте внимательны, в папке `lib` находятся две папки: `x64` и `x86`. Даже если у вас Windows разрядности x64, указывать нужно папку `x86`. Затем `"Выбор папки"` и `"OK"`:



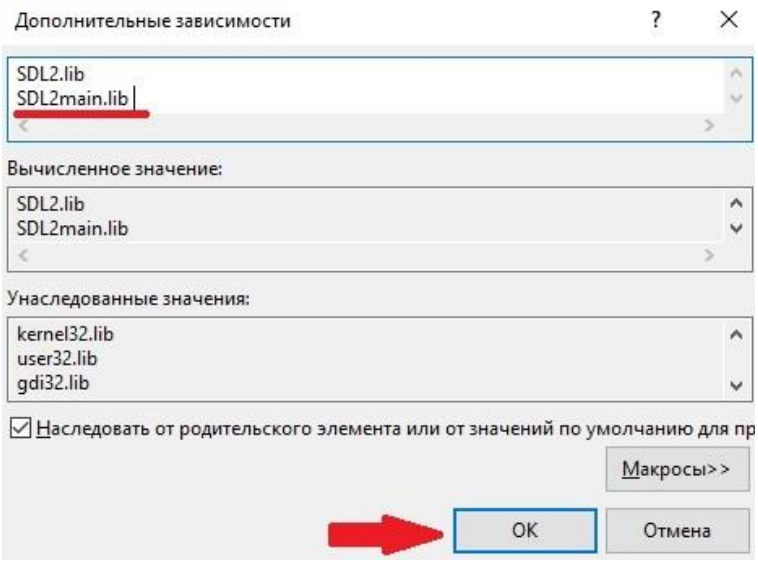
После этого переходим в `"Компоновщик" > "Ввод"`. Затем `"Дополнительные зависимости" > стрелочку вниз > "Изменить"`:



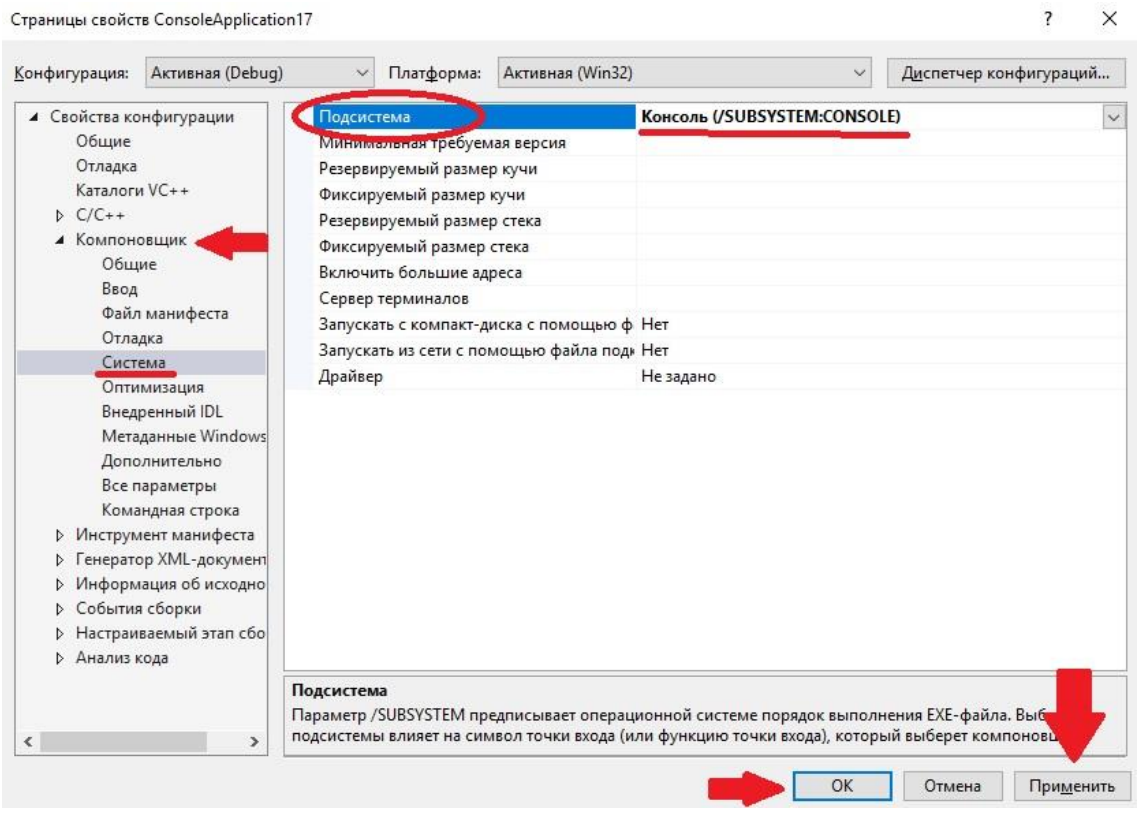
В появившемся текстовом блоке вставляем:

```
SDL2.lib
SDL2main.lib
```

И нажимаем `"OK"`:

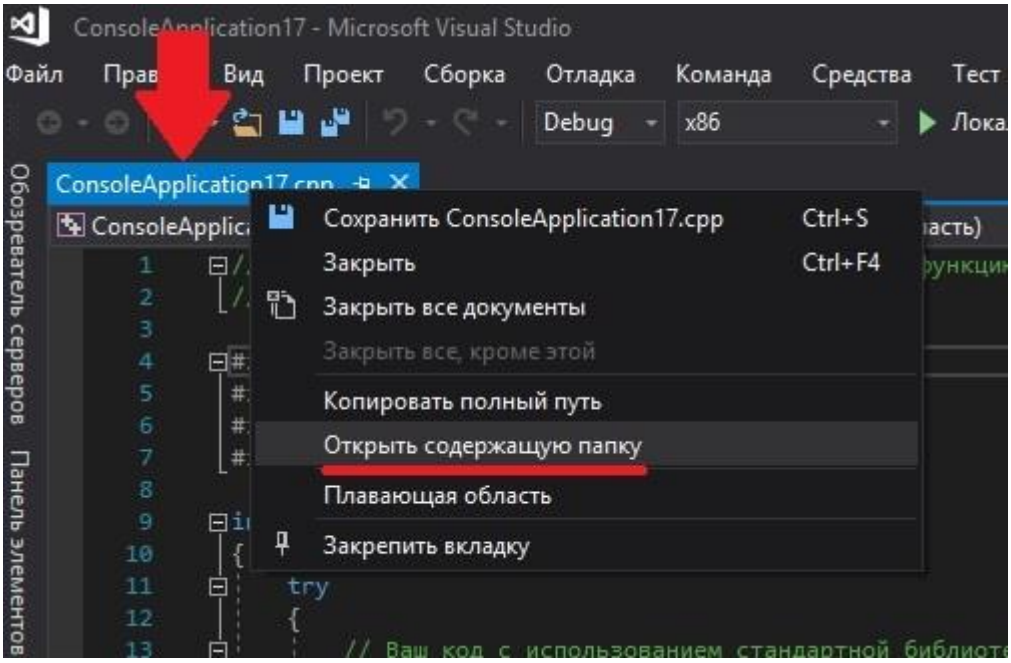


Затем переходим в `"Компоновщик" > "Система"`. После этого `"Подсистема" > стрелочку вниз > выбираем "Консоль (/SUBSYSTEM:CONSOLE)" > "Применить" > "OK"`:

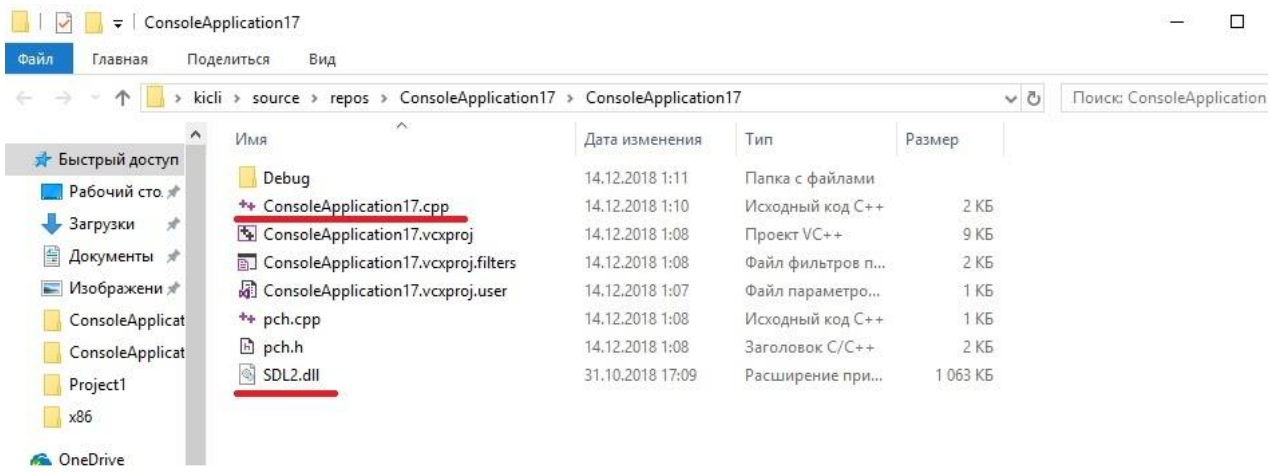


Шаг №5: Копируем dll-ку в папку с проектом

Переходим в папку `x86` (`C:\Libs\SDL2-2.0.9\lib\x86`), копируем SDL2.dll и вставляем в папку с вашим проектом в Visual Studio. Чтобы просмотреть папку вашего проекта в Visual Studio, `нажмите ПКМ по названию вашего проекта > "Открыть содержащую папку"`:



Затем вставляем скопированный файл (SDL2.dll) в папку с проектом (где находится рабочий файл .cpp):



Всё!

Шаг №6: Тестируем

Теперь, чтобы проверить, всё ли верно мы сделали – копируем и запускаем следующий код:

```
#include <iostream>
#include <SDL.h>

int main(int argc, char * argv[])
{

    if (SDL_Init(SDL_INIT_EVERYTHING) < 0)
    {
        std::cout << "SDL initialization failed. SDL Error: " << SDL_GetError();
    }
    else
    {
        std::cout << "SDL initialization succeeded!";
    }

    std::cin.get();
    return 0;
}
```

Если результат следующий:

```
SDL initialization succeeded!
```

Значит мы успешно подключили библиотеку SDL к нашему проекту!

Если вы получили какую-либо ошибку, то внимательно повторите все вышеприведенные действия, но уже с новым проектом. Скорее всего вы что-то пропустили или указали неверные пути к папкам.

C++11. Нововведения

12 августа 2011 года **Международная организация по стандартизации** (англ. ***"ISO"***) утвердила новую версию языка C++ под названием C++11. Было добавлено много нового функционала, использовать который не обязательно, но который оказался очень даже полезным в определенных случаях.

Зачем нужен C++11?

Бьёрн Страуструп охарактеризовал цели C++11 следующим образом:

- Укрепить сильные стороны языка C++, нежели пытаться захватить те области, в которых язык C++ слаб (например, приложения Windows с тяжелым графическим интерфейсом). Сосредоточиться на том, чтобы заставить язык C++ делать еще лучше то, что у него и так хорошо получается.
- Сделать язык C++ легче в изучении, использовании и обучении, предоставить функционал, который сделает язык более простым в использовании.

Руководствуясь этими целями, комитет, который согласовывал версию C++11, постарался выполнить следующее:

- Поддерживать стабильность и совместимость со старыми версиями языков C++ и Си (насколько это возможно). Программы, которые работали под C++03 должны работать и под C++11.
- Свести к минимуму количество основных расширений языка C++. Внести большую часть изменений в Стандартную библиотеку C++ (эта цель не была достигнута в полной мере).
- Сосредоточиться на улучшении механизмов абстракции (классы, шаблоны), нежели на добавлении механизмов обработки конкретных случаев (которые случаются нечасто).
- Повысить безопасность типов данных.
- Повысить производительность и позволить языку C++ напрямую работать с оборудованием.
- Рассмотреть вопросы юзабилити и экосистемы. Язык C++ должен хорошо работать с другими инструментами, быть простым в использовании, обучении и т.д.

C++11 не слишком далеко ушел от C++ 03, но он действительно привнес много нового функционала.

Что нового в C++11?

Вот **список основного функционала, который добавили в C++11** (это не полный список всего, что добавили, а основная его часть, представляющая интерес):

- auto;
- char16_t, char_32t и новые литералы для их поддержки;
- constexpr;
- decltype;
- спецификатор default;
- делегирующие конструкторы;
- ключевое слово delete;
- классы enum;
- внешние шаблоны;
- лямбда-выражения;
- long long int;
- конструктор перемещения и оператор присваивания перемещением;
- спецификатор noexcept;
- nullptr;
- модификаторы override и final;
- цикл foreach;
- ссылки r-value;
- static_assert;
- std::initializer_list;
- псевдонимы типов;
- uniform-инициализация;
- пользовательские литералы;
- вариационные шаблоны.

В C++11 также было добавлено много новых классов, доступных к использованию, и улучшены старые:

- лучшая поддержка многопоточности и локальное хранилище потоков;
- хеш-таблицы;
- улучшенная генерация случайных чисел;
- std::reference_wrapper;
- регулярные выражения;
- std::tuple;
- std::unique_ptr.

C++14. Нововведения

18 августа 2014 года **Международная организация по стандартизации** (англ. ***"ISO"***) утвердила новую версию языка C++ под названием C++14. В отличие от версии C++11, в которой добавилось относительно много нового функционала, в C++14 произошло лишь небольшое обновление — в основном исправления ошибок и небольшие улучшения.

Что нового в C++14?

Вот список основных улучшений, добавленных в C++14 (это не полный список всего добавленного, а ключевые улучшения, представляющие интерес):

- бинарные литералы;
- атрибут deprecated;
- цифровые разделители;
- автоматическое определение возвращаемого типа функции – вывод типов;
- relaxed constexpr функции;
- шаблоны переменных;
- стандартные пользовательские литералы;
- std::make_unique().

C++17. Нововведения

В декабре 2017 года **Международная организация по стандартизации** (англ. ***"ISO"***) утвердила новую версию языка C++ под названием C++17.

Что нового в C++17?

Вот **список основных улучшений, добавленных в C++17** (это не полный список всего добавленного, а ключевые улучшения, представляющие интерес):

- идентификатор препроцессора `__has_include` для проверки доступности дополнительных заголовочных файлов;
- стейтменты if, которые обрабатываются во время компиляции;
- инициализаторы в стейтментах if и switch;
- встроенные переменные;
- fold-выражения;
- вложенные пространства имен теперь можно определять как пространство имен `X::Y`;
- удаление std::auto_ptr и других устаревших типов;
- static_assert больше не требует параметра в виде текстового сообщения;
- std::any;
- std::byte;
- std::filesystem;
- std::optional;
- std::shared_ptr теперь может управлять массивами C-style (но через std::make_shared() их по-прежнему нельзя создавать);
- std::size;
- триграфы были удалены;
- UTF-8 (u8) символьные литералы.

Спецификации исключений и спецификатор noexcept

В языке C++ все функции условно можно разделить на 2 типа:

- функции, **не выбрасывающие исключений**;
- функции, которые **потенциально могут выбросить** исключения.

Рассмотрим следующее объявление функции:

```
int doSomething(); // может ли эта функция выбросить исключение или нет?
```

Глядя на типичное объявление функции, не представляется возможным определить, может ли функция выбросить исключение или нет. Несмотря на то, что комментарии могут помочь обозначить, выбрасывает ли функция исключения или нет (и если да, то какие именно исключения), у нас нет никакого специального компилятора для комментариев, а документация может устареть.

Спецификации исключений — это механизм языка C++, который изначально был разработан для документирования в пределах объявления функции того, какие исключения она может выбрасывать. Хотя большинство *спецификаций исключений* теперь устарели или удалены из стандарта, но в качестве замены была добавлена одна полезная спецификация исключений, которую мы рассмотрим на данном уроке.

Спецификатор noexcept

Спецификатор noexcept определяет функцию как *не выбрасывающую* исключений. Чтобы определить функцию как *не выбрасывающую*, мы можем использовать спецификатор noexcept в объявлении функции, поместив его справа от списка параметров функции:

```
void doSomething() noexcept; // эта функция является не выбрасывающей
    исключений функцией
```

Обратите внимание, что noexcept на самом деле не запрещает функции выбрасывать исключения или вызывать другие функции, которые *потенциально могут выбросить исключения*. Скорее всего, при возникновении исключения, если оно происходит из noexcept-функции, будет вызвана функция std::terminate(). И обратите внимание, что если std::terminate() вызывается внутри noexcept-функции, то раскручивание стека может происходить, а может и не происходить (в зависимости от реализации и оптимизации). А это означает, что ваши объекты могут

быть уничтожены должным образом до завершения работы, а может и не произойти этого уничтожения.

Подобно тому, как функции, отличающиеся только своими возвращаемыми значениями, не могут быть перегружены, функции, отличающиеся только своей спецификацией исключений, также не могут быть перегружены.

Спецификатор noexcept с параметром типа bool

Спецификатор noexcept имеет необязательный параметр типа bool:

- `noexcept(true)` равносильно noexcept, что означает, что функция не является выбрасывающей;
- `noexcept(false)` означает, что функция относится к классу потенциально выбрасывающих исключения функций.

Эти параметры обычно используются только в шаблонных функциях, так что шаблонная функция может быть динамически создана как *не выбрасывающая* или *потенциально выбрасывающая* исключения на основе некоторого параметризованного значения.

Не выбрасывающие и потенциально выбрасывающие исключения функции

Функции, которые по умолчанию являются не выбрасывающими исключения:

- конструкторы, заданные по умолчанию;
- конструкторы копирования;
- конструкторы перемещения;
- деструкторы;
- операторы присваивания копированием;
- операторы присваивания перемещением.

Однако, если какая-либо из вышеперечисленных функций вызывает (явно или неявно) другую функцию, которая может выбросить исключение, то вызывающая функция также будет рассматриваться как потенциально выбрасывающая исключения. Например, если класс имеет элемент данных с потенциально выбрасывающим исключение конструктором, то конструкторы класса также будут рассматриваться как потенциально выбрасывающие исключения. В качестве другого примера, если оператор присваивания копированием вызывает потенциально выбрасывающий исключение оператор присваивания, то оператор

присваивания копированием также будет рассматриваться как оператор, потенциально выбрасывающий исключения.

> ***Совет:*** Если вы хотите, чтобы какая-либо из вышеперечисленных функций не выбрасывала исключений, то явно пометьте её как noexcept (даже если она является таковой по умолчанию), чтобы случайно данная функция не стала функцией, потенциально выбрасывающей исключение.

По умолчанию, потенциально выбрасывающими исключение функциями являются следующие объекты:

- обычные функции;
- пользовательские конструкторы;
- некоторые операторы, такие как оператор new.

Оператор noexcept

Оператор noexcept может использоваться внутри функций. Он принимает в качестве аргумента выражение и возвращает `true` или `false`, если компилятор считает, что выражение не может или может выбросить исключение. Оператор noexcept проверяется статически во время компиляции и фактически не вычисляет входное выражение.

```
void foo() {throw -1;}
void boo() {};
void goo() noexcept {};
struct S{};

constexpr bool b1{ noexcept(5 + 3) }; // true; целочисленные значения не
    выбрасывают исключений
constexpr bool b2{ noexcept(foo()) }; // false; функция foo() выбрасывает
    исключение
constexpr bool b3{ noexcept(boo()) }; // false; функция boo() – неявное
    noexcept(false)
constexpr bool b4{ noexcept(goo()) }; // true; функция goo() – явное
    noexcept(true)
constexpr bool b5{ noexcept(S{}) };   // true; заданный по умолчанию
    конструктор структуры является по умолчанию noexcept
```

Оператор noexcept может использоваться для задания условия выполнения кода: является ли фрагмент кода потенциально выбрасывающим исключения или нет. Это необходимо для осуществления определенных гарантий безопасности исключений, о которых мы поговорим дальше.

Гарантии безопасности исключений

Гарантии безопасности исключений — это договоренности о том, как функции или классы будут вести себя в случае возникновения исключений.

Существуют 4 уровня безопасности исключений:

- **Нет никаких гарантий** — нет никаких гарантий относительно того, что произойдет, если возникнет исключение (например, класс может быть оставлен в непригодном для использования состоянии).
- **Базовая гарантия** — если возникнет исключение, то утечки памяти не произойдет (все ресурсы будут освобождены корректно), и объект все еще будет использоваться, но программа может быть оставлена в измененном состоянии.
- **Строгая гарантия** — если возникнет исключение, то утечки памяти не произойдет (все ресурсы будут освобождены корректно), состояние программы не будет изменено. Это означает, что функция должна корректно завершить свою работу, либо не иметь побочных эффектов в случае, если функция аварийно завершила свою работу. Простыми словами — если при выполнении операции возникнет исключение, то программа останется в том же состоянии, которое было до начала выполнения операции.
- **Гарантия отсутствия исключений/сбоев** — работа функции всегда завершается успешно (без сбоев) или завершается аварийно, но без выбрасывания исключений.

Давайте рассмотрим пункт *«Гарантия отсутствия исключений/сбоев»* более подробно.

Гарантия отсутствия исключений: Если функция завершается аварийно, то она не будет выбрасывать исключение. Вместо этого она вернет код ошибки или проигнорирует проблему. Гарантии отсутствия исключений требуются во время раскручивания стека, когда исключение уже обрабатывается; например, все деструкторы должны иметь гарантию отсутствия исключения (как и любые функции, вызываемые этими деструкторами). Примеры кода, который относится к коду с отсутствием исключений:

- деструкторы и функции освобождения/очистки памяти;
- функции, которые вызывают другие функции, имеющие гарантии отсутствия исключений.

Гарантия отсутствия сбоев: Функция всегда успешно выполняет свою работу (и, следовательно, у нее никогда не будет необходимости выбрасывать исключения. Таким образом, гарантия отсутствия сбоя — это немного более сильная форма гарантии отсутствия исключения).

Примеры кода, который относится к коду с отсутствием сбоев:

- конструкторы перемещения и оператор присваивания перемещением;
- swap-функции;
- контейнерные функции clear()/erase()/reset();
- операции с std::unique_ptr;
- функции, которые должны вызывать другие функции, имеющие гарантии отсутствия сбоев.

Когда следует использовать спецификатор noexcept

Из того факта, что ваш код явно не выбрасывает никаких исключений, еще не следует, что вы должны начать указывать спецификатор noexcept на все подряд функции. По умолчанию, большинство функций потенциально способны выбросить исключение, поэтому, если ваша функция вызывает другие функции, есть вероятность того, что она вызывает функцию, которая потенциально способна выбросить исключение, и, следовательно, сама вызывающая функция будет относиться к потенциально выбрасывающим исключение функциям.

Принципы Стандартной библиотеки C++ состоят в том, чтобы использовать спецификатор noexcept только для тех функций, которые НЕ ДОЛЖНЫ выбрасывать исключения или аварийно завершать свою работу. Функции, которые потенциально способны выбросить исключения, но по факту не генерируют исключения (из-за реализации), как правило, не помечаются как noexcept-функции.

> Советы:
>
> - Используйте спецификатор noexcept лишь в конкретных случаях, когда вы хотите явно указать на гарантию отсутствия сбоя или отсутствие выбрасывания исключения.
> - Если вы не уверены, должна ли функция иметь гарантию отсутствия сбоя/исключения, то лучше перестраховаться и не отмечать её с помощью noexcept. Решение использовать в таких случаях спецификатор noexcept нарушает обязательство взаимодействия с пользователем относительно поведения функции. Гораздо лучше потом при необходимости ужесточить гарантии безопасности, добавив спецификатор noexcept.

Почему полезно отмечать функции, как не выбрасывающие исключений функции

Есть пара веских причин, почему стоит помечать функции спецификатором noexcept:

- Функции, которые являются noexcept, могут позволить компилятору выполнять некоторые оптимизации, которые в противном случае были бы недоступны. Поскольку noexcept-функция не может выбрасывать исключения, то компилятору не нужно беспокоиться о сохранении стека времени выполнения в состоянии произвести раскручивание, что может позволить компилятору создавать более быстрый код.
- Есть также несколько случаев, когда знание, что функция относится к noexcept, позволяет нам создавать более эффективные реализации в нашем собственном коде: стандартные библиотечные контейнеры (такие как std::vector) знают о noexcept и будут использовать его для определения того, следует ли использовать семантику перемещения (быстрее) или семантику копирования (медленнее) в определенных местах.

Динамические спецификации исключений

До C++11 и даже до C++17, динамические спецификации исключений использовались вместо noexcept. Синтаксис **динамических спецификаций исключений** использует ключевое слово throw для перечисления типов исключений, которые функция может прямо или косвенно генерировать:

```
int doSomething() throw(); // не выбрасывает исключений
int doSomething() throw(std::out_of_range, int*); // может выбросить либо
    исключение типа std::out_of_range, либо указатель на целочисленное значение
int doSomething() throw(...); // может выбросить что угодно
```

Из-за таких факторов, как неполная реализация компилятора, некоторая несовместимость с шаблонными функциями, распространенное непонимание того, как они работают, и тот факт, что Стандартная библиотека C++ в основном не использует их, динамические спецификации исключений были объявлены устаревшими в C++11 и удалены из языка в C++17 и C++20. Более детально об этом читайте **здесь**.

Функция std::move_if_noexcept()

Этот урок основан на уроке о спецификациях исключений и спецификаторе noexcept, где мы рассматривали строгую гарантию исключений, которая гарантирует то, что если работа функции прерывается исключением, то не произойдет утечки памяти и состояние программы не будет изменено. В частности, все конструкторы должны придерживаться строгой гарантии исключений, чтобы остальная часть программы не оставалась в измененном состоянии, если создание объекта завершится неудачей.

Проблемы с исключениями при их совместном использовании с конструкторами перемещения

Рассмотрим случай, когда мы копируем какой-то объект, и копирование по какой-то причине дает сбой (например, не хватает памяти). В таком случае копируемому объекту не причиняется никакого вреда, поскольку исходный объект не нуждается в модификации для создания копии. Мы можем отбросить неудавшуюся копию и двигаться дальше. Строгая гарантия исключений сохраняется.

Теперь рассмотрим случай, когда мы вместо копирования перемещаем объект. Операция перемещения передает право собственности на используемый ресурс от источника к целевому объекту. Если операция перемещения прерывается исключением после того, как происходит передача права собственности, то наш исходный объект останется в измененном состоянии. Это не проблема, если исходный объект является временным объектом и будет в любом случае отброшен после перемещения, но для объектов, не являющихся временными, это — проблема, т.к. мы повредили исходный объект. Для соблюдения правил строгой гарантии исключений, нам нужно было бы переместить используемый ресурс обратно в исходный объект, но если перемещение не удалось в первый раз, то нет никакой гарантии, что и обратное перемещение будет успешным.

Каким образом для конструкторов перемещения можно обеспечить выполнение строгой гарантии исключений? Для этого достаточно избегать создания исключений в теле конструктора перемещения. Но конструктор перемещения может вызывать другие конструкторы, которые потенциально способны генерировать исключения. Возьмем, к примеру, конструктор перемещения для `std::pair`, который должен попытаться переместить каждый подобъект в исходной паре в новый объект:

```
// Пример определения конструктора перемещения для std::pair.
// Берем «старую» пару объектов, и при помощи конструктора перемещения создаем
   новую пару объектов, в которой «первый» и «второй» объекты получены из «старых»
```

```
template <typename T1, typename T2>
pair<T1,T2>::pair(pair&& old)
  : first(std::move(old.first)),
    second(std::move(old.second))
{}
```

Мы будем использовать два класса: MoveClass и CopyClass, из которых создадим пару, чтобы продемонстрировать проблему строгой гарантии исключений при работе с конструкторами перемещения:

```
#include <iostream>
#include <utility> // для работы std::pair, std::make_pair, std::move,
   std::move_if_noexcept
#include <stdexcept> // std::runtime_error
#include <string>
#include <string_view>

class MoveClass
{
private:
  int* m_resource{};

public:
  MoveClass() = default;

  MoveClass(int resource)
    : m_resource{ new int{ resource } }
  {}

  // Конструктор копирования
  MoveClass(const MoveClass& that)
  {
    // Глубокое копирование
    if (that.m_resource != nullptr)
    {
      m_resource = new int{ *that.m_resource };
    }
  }

  // Конструктор перемещения
  MoveClass(MoveClass&& that)
    : m_resource{ that.m_resource }
  {
    that.m_resource = nullptr;
  }

  ~MoveClass()
  {
    std::cout << "destroying " << *this << '\n';

    delete m_resource;
  }

  friend std::ostream& operator<<(std::ostream& out, const MoveClass& moveClass)
  {
    out << "MoveClass(";

    if (moveClass.m_resource == nullptr)
    {
      out << "empty";
```

```
    }
    else
    {
      out << *moveClass.m_resource;
    }

    out << ')';

    return out;
  }
};


class CopyClass
{
public:
  bool m_throw{};

  CopyClass() = default;

  // Конструктор копирования выбрасывает исключение при выполнении копирования из
   объекта CopyClass, где его переменная m_throw имеет значение 'true'
  CopyClass(const CopyClass& that)
    : m_throw{ that.m_throw }
  {
    if (m_throw)
    {
      throw std::runtime_error{ "abort!" };
    }
  }
};

int main()
{
  // Мы можем создать объект std::pair без каких-либо проблем
  std::pair my_pair{ MoveClass{ 13 }, CopyClass{} };

  std::cout << "my_pair.first: " << my_pair.first << '\n';

  // Но проблемы начинают появляться, когда мы пытаемся переместить объекты
   одной пары в другую пару
  try
  {
    my_pair.second.m_throw = true; // чтобы спровоцировать генерацию исключения
   конструктором копирования

    // Следующая строка выбросит исключение
    std::pair moved_pair{ std::move(my_pair) }; // мы закомментируем эту строку
   чуть позже
    // std::pair moved_pair{std::move_if_noexcept(my_pair)}; // мы
   раскомментируем эту строку чуть позже

    std::cout << "moved pair exists\n"; // никогда не выведется
  }
  catch (const std::exception& ex)
  {
      std::cerr << "Error found: " << ex.what() << '\n';
  }

  std::cout << "my_pair.first: " << my_pair.first << '\n';

  return 0;
```

```
}
```

Результат выполнения программы:

```
destroying MoveClass(empty)
my_pair.first: MoveClass(13)
destroying MoveClass(13)
Error found: abort!
my_pair.first: MoveClass(empty)
destroying MoveClass(empty)
```

Давайте разберем результат вывода построчно:

- *Строка №1:* Здесь мы видим, что временный объект класса MoveClass, используемый для инициализации `my_pair`, уничтожается сразу же после выполнения стейтмента создания экземпляра `my_pair`. Он пуст (`empty`), так как подобъект класса MoveClass в `my_pair` был создан из него перемещением, о чем свидетельствует следующая строка вывода.
- *Строка №2:* Здесь мы видим, что `my_pair.first` содержит объект класса MoveClass со значением `13`.
- *Строка №3:* Мы создали `moved_pair`, копируя его подобъект класса CopyClass (у него нет конструктора перемещения), но эта конструкция копирования вызвала исключение, так как мы изменили логический флаг. Построение объекта `moved_pair` было прервано генерацией исключения, и его уже созданные члены были уничтожены. В этом случае член класса MoveClass был уничтожен, подтверждением чего является вывод строки `destroying MoveClass(13)`.
- *Строка №4:* Далее мы видим сообщение `Error found: abort!` из функции main().
- *Строка №5:* Когда мы опять пытаемся вывести на экран `my_pair.first`, мы снова видим, что член класса MoveClass пуст (`empty`). Поскольку объект `moved_pair` был инициализирован с помощью функции std::move(), то член класса MoveClass (который имеет конструктор перемещения) был перемещен для создания `moved_pair`, и объект `my_pair.first` был обнулен.
- *Строка №6:* В завершении, объект `my_pair` уничтожается в конце функции main().

Итак, подводя итог вышеописанных результатов, можно заключить, что конструктор перемещения std::pair использовал выбрасывающий исключение конструктор копирования класса CopyClass. Этот конструктор копирования выбрасывает исключение, вызывающее прерывание создания объекта `moved_pair`, из-за чего

объект `my_pair.first` постоянно повреждается. Строгая гарантия исключений не выполняется.

На помощь спешит функция std::move_if_noexcept()

Обратите внимание, что вышеописанной проблемы можно было бы избежать, если бы std::pair попытался сделать копию вместо перемещения. В таком случае объект `moved_pair` не удалось бы создать, но объект `my_pair` не был бы изменен.

Но за копирование вместо перемещения приходится расплачиваться производительностью, даже для тех объектов, которые этого и не требуют. В идеале мы бы хотели выполнить перемещение, если это возможно сделать безопасно, и копирование — в противном случае.

К счастью, в C++ есть два механизма, которые при совместном использовании позволяют нам это выполнить. Во-первых, поскольку noexcept-функции являются функциями без исключений/сбоев, то они неявно удовлетворяют критериям строгой гарантии исключений. Таким образом, noexcept-конструктор перемещения гарантированно завершит свою работу успешно.

Во-вторых, мы можем использовать **функцию std::move_if_noexcept()** из Стандартной библиотеки C++, чтобы определить, следует ли выполнять перемещение или копирование. Функция std::move_if_noexcept() является аналогом функции std::move() и используется таким же образом.

Если компилятор может определить, что объект, переданный в качестве аргумента для std::move_if_noexcept(), не будет выбрасывать исключение при применении конструктора перемещения (или если объект является только перемещаемым и не имеет конструктора копирования), то std::move_if_noexcept() будет работать идентично std::move() (и вернет объект, преобразованный в r-value). В противном случае, std::move_if_noexcept() вернет обычную l-value ссылку на объект.

> ***Ключевой момент:*** Функция std::move_if_noexcept() вернет перемещаемый r-value объект в том случае, если объект имеет noexcept-конструктор перемещения, в противном случае он вернет копируемый l-value объект. Мы можем задействовать спецификатор noexcept в сочетании с std::move_if_noexcept() для использования семантики перемещения только тогда, когда выполняется строгая гарантия исключений (и использовать семантику копирования в противном случае).

Давайте обновим код из предыдущего примера следующим образом:

```
//std::pair moved_pair{std::move(my_pair)}; // закомментируем эту строку
std::pair moved_pair{std::move_if_noexcept(my_pair)}; // и раскомментируем
    эту строку
```

Запуск программы покажет нам следующий результат:

```
destroying MoveClass(empty)
my_pair.first: MoveClass(13)
destroying MoveClass(13)
Error found: abort!
my_pair.first: MoveClass(13)
destroying MoveClass(13)
```

Как вы можете видеть, после того, как было выброшено исключение, подобъект `my_pair.first` по-прежнему указывает на значение `13`.

Класс CopyClass не имеет noexcept-конструктора перемещения, поэтому std::move_if_noexcept() возвращает `my_pair` в качестве l-value ссылки. Это приводит к тому, что `moved_pair` создается с помощью конструктора копирования (а не конструктора перемещения). Конструктор копирования может смело выбрасывать исключения, поскольку он не изменяет исходный объект.

Стандартная библиотека C++ часто использует std::move_if_noexcept() для оптимизации функций, которые являются noexcept. Например, std::vector::resize() будет использовать семантику перемещения, если тип элемента имеет noexcept-конструктор перемещения, и семантику копирования в противном случае. Это означает, что std::vector обычно будет работать быстрее с объектами, имеющими noexcept-конструктор перемещения (*напоминание:* конструкторы перемещения по умолчанию являются noexcept, если только они не вызывают функцию, которая является `noexcept(false)`).

> ***Предупреждение:*** Если тип имеет как потенциально вызывающую исключения семантику перемещения, так и семантику удаленного копирования (конструктор копирования и оператор присваивания копированием недоступны), то std::move_if_noexcept() откажется от строгой гарантии исключений и вызовет семантику перемещения. Этот условный отказ от строгой гарантии повсеместно встречается в стандартных библиотечных контейнерных классах, поскольку они часто используют std::move_if_noexcept().

Конец? Что дальше?

Примите мои поздравления! Вы полностью прошли основную часть туториала по языку C++! Вы красавчики! Вы получили необходимую базу/фундамент в программировании и, в частности, в программировании на языке C++. Теперь вы уже должны решить самостоятельно, в чем именно вы хотите развиваться дальше. Направлений есть много, мы рассмотрим основные из них.

Структуры данных и алгоритмы

Структуры данных — это набор данных и методов для доступа и манипулирования этими данными. Наиболее распространенной структурой данных, используемой в программировании, является массив, который содержит ряд последовательных элементов одного типа. Вы можете манипулировать этими данными, используя индексацию массива для получения прямого доступа к элементам массива (и их последующего изменения). Мы уже ранее рассматривали стек, как структуру данных, а также методы push() и pop() для манипулирования этими данными.

Алгоритм — это последовательность операций для манипулирования данными с целью получения определенного результата. Например, когда вы просматриваете массив, чтобы найти среднее значение, вы выполняете алгоритм. Бинарный поиск — это алгоритм, позволяющий определить, существует ли заданное значение в отсортированном массиве. Методы сортировки (такие как сортировка методом выбора или пузырьковая сортировка) — это алгоритмы, которые сортируют наборы данных.

В программировании структуры данных и алгоритмы служат одной цели: они являются инструментами, позволяющими в разы увеличить скорость написания и эффективность выполнения вашего кода (при условии, что вы научились пользоваться этими инструментами).

Хорошей новостью является то, что многие из структур данных и алгоритмов, которые вам будут нужны, уже реализованы в Стандартной библиотеке C++ (std::array, std::vector, std::stack, std::string, std::sort() и т.д.). Ваша цель — научиться эффективно их использовать. Кроме того, вы можете попробовать реализовать их самостоятельно с нуля.

Стандартная библиотека C++

Основная часть Стандартной библиотеки C++ — это структуры данных и алгоритмы. Однако есть и другой функционал, который вы можете использовать: числовые (математические) библиотеки, подпрограммы ввода/вывода, функции для управления локализацией, регулярные выражения, многопоточность и т.д. Каждый новый релиз версии языка C++ (который происходит каждые 3 года) добавляет новый функционал в Стандартную библиотеку C++. Вам не обязательно знать, как это всё работает, но вам нужно хотя бы знать, что уже есть, тогда, в случае надобности, не придется придумывать свои велосипеды.

Приложения с графическим интерфейсом

Мы имели дело только с консольными приложениями, поскольку они просты, кроссплатформенны и не требуют установки дополнительного программного обеспечения. В отличие от многих современных языков программирования, язык C++ не имеет встроенного функционала для создания приложений с графическим интерфейсом. Для этого вам нужно будет подключать дополнительные библиотеки. Популярными вариантами являются Qt, WxWidgets, SDL и SFML. Если вы хотите работать с 3D-графикой, то вам нужно будет разбираться с OpenGL.

Графические приложения работают не так, как консольные. В консольном приложении выполнение кода начинается последовательно с первой строки функции main(), обычно останавливаясь только для пользовательского ввода. В графическом приложении выполнение кода тоже начинается с первой строки функции main(): создание рабочего окна, заполнение его графическими объектами, виджетами, а затем бесконечный цикл с ожиданием взаимодействия пользователя с окном (через нажатия на кнопку мыши или на клавишу клавиатуры). Этот бесконечный цикл называется **циклом событий**, а когда происходит клик мыши или нажатие на клавишу клавиатуры, то произошедшее событие направляется в функцию(и), которая обрабатывает этот тип события. Это называется **обработкой событий**. Как только событие обработано, цикл событий продолжает свое выполнение, ожидая следующего пользовательского ввода.

TCP/IP. Сетевое программирование

Сейчас уже большинство всех программ подключаются к сети Интернет, к внешнему серверу или к облаку. Любая программа, которая требует наличия учетной записи и входа в систему, подключается к серверу и аутентифицирует пользователя. Многие программы подключаются к службе обновлений (для проверки того, доступно ли

обновление). Социальные сети используют Интернет, чтобы пользователи могли общаться друг с другом. И таких примеров десятки.

Сеть (в широком смысле этого слова) — это подключение вашей программы к другим программам на вашем компьютере или на других компьютерах, подключенных к сети, для обмена информацией. Работа в сети является мощным инструментом. В прошлом, если вы хотели изменить работу вашего приложения, вам приходилось выпускать и распространять обновление приложения. Теперь же вы можете просто обновить информацию где-нибудь на сервере, и все экземпляры вашей программы смогут автоматически использовать это изменение.

В языке C++ есть отдельные библиотеки для работы с сетевым программированием (например, библиотека Asio).

Многопоточность

К этому моменту все программы, которые мы рассматривали, выполнялись последовательно. Только после завершения выполнения одной задачи, начинала выполняться следующая. Если по каким-либо причинам (например, если от пользователя требовался ввод, а он ничего не вводил) выполнение задачи тормозилось, то и выполнение целой программы также приостанавливалось. В теории, для небольших тривиальных программ - это еще ок, но на практике (с реальными приложениями) — это почти всегда плохо. Представьте, что ваша программа не может обработать пользовательский ввод, потому что она занята выводом чего-либо на экран, или, например, выполнение программы приостановилось из-за передачи данных из одного узла программы в другой.

К счастью, есть такая вещь, как **многопоточность**, позволяющая программам выполнять сразу несколько задач одновременно. Аналогично тому, как вы можете идти и разговаривать по телефону, так и многопоточность позволяет программе «разделить» свое внимание на выполнение сразу нескольких задач параллельно друг другу.

Например, некоторые графические приложения (такие, как веб-браузеры) помещают рендеринг картинки в отдельный поток, чтобы обновление экрана не блокировало другие вещи (например, обработку пользовательского ввода).

Многопоточность — вещь мощная, но переходить к её изучению стоит хотя бы с минимальным опытом работы с сетевым программированием или с графическими приложениями.

Ravesli. Что дальше?

230+ уроков, сотни тысяч слов, миллионы символов. Первый урок датируется 2 апреля 2016 года, последний — 17 февраля 2019 года. Почти 3 года у меня ушло на перевод данных уроков. Отдельное спасибо автору **LearnCpp.com** — это его уроки, а не мои (если кто не знал). Я выполнил лишь их адаптированный перевод.

Вы получили необходимый старт для вашего развития в программировании, теперь я предлагаю вам прокачать ваш уровень еще больше с помощью следующих материалов на Ravesli:

- **Задания по C++**. 70+ упражнений с разными уровнями сложности для выполнения. Здесь вы сможете хорошенько попрактиковаться, начиная уже с **первой части**.
- **Пошаговое создание графической игры на C++ с помощью библиотеки MFC**. Начинаем на **первом уроке** и продолжаем создавать с нуля игрушку "Same Game".
- **Уроки по графической библиотеке SFML**. Разбираемся с тонкостями работы с графикой в языке C++, начиная со **вступления и установки SFML**.
- **Уроки по Qt5**. Рассматриваем кроссплатформенный фреймворк Qt5, его функционал и возможности. Начинаем с **установки QtCreator**.
- **Уроки по OpenGL**. Рассматриваем функционал OpenGL простым для понимания способом с наглядными примерами, а также предоставляем бэкграунд для ваших последующих исследований/углублений в эту тему. Начинаем с **введения в OpenGL**.

Как видите, есть куда копать. На уроках, перечисленных выше, рассматриваются конкретные фреймворки и библиотеки, а также создаются полноценные проекты/игры. Изучив эти уроки и повторив всё, что на них рассматривается, вы получите не только теорию со знаниями, но и практический опыт с проектами в портфолио. Всё в ваших руках!

Спасибо Вам!

>ОтветыНаТесты(); _

ravesli.com

Ответы: Урок №11

Ответ №1

Стейтмент — это "полное предложение", которое сообщает компилятору, что ему нужно выполнить определенное задание. Выражение всегда имеет результат (исключение — деление на ноль) и является частью стейтмента.

Ответ №2

Функция — это последовательность стейтментов для выполнения определенного задания. Библиотека — это последовательность функций, которые могут повторно использоваться в других программах.

Ответ №3

Точкой с запятой (`;`).

Ответ №4

Синтаксическая ошибка — это ошибка, указывающая на нарушение правил грамматики языка C++.

Ответы: Урок №13

Ответ №1

Программа выведет `3`: `a - 3 = 3`, что и присваивается переменной `a`.

Ответ №2

Программа выведет `3`: переменной `b` присваивается значение переменной `a` (`3`).

Ответ №3

Программа выведет `6`: `a + b = 6`. Здесь не используется операция присваивания.

Ответ №4

Программа выведет `3`: значением переменной `a` до сих пор является `3`.

Ответ №5

Результатом будет ошибка, так как `c` — это неинициализированная переменная.

Ответы: Урок №15

Ответ №1

Скомпилируется, результатом выполнения программы будет значение `13`.

Ответ №2

Эта программа не скомпилируется. Вложенные функции запрещены.

Ответ №3

Эта программа скомпилируется, но не будет никакого вывода. Возвращаемые значения из функций не используются в main() и, таким образом, игнорируются.

Ответ №4

Эта программа не скомпилируется, так как тип возврата функции printO() — void, а мы отправляем несуществующее возвращаемое значение на вывод. Результат — ошибка компиляции.

Ответ №5

Результатом выполнения этой программы будет:

```
6
6
```

Оба раза, когда вызывается функция getNumbers(), возвращается значение `6`. Компилятор, встречая первый return, сразу же выполняет возврат этого значения, и всё, что находится за первым return-ом, — игнорируется. Строка `return 8;` никогда не выполнится.

Ответ №6

Эта программа не скомпилируется из-за недопустимого имени функции.

Ответ №7

Эта программа скомпилируется, но функция не будет вызвана, так как в её вызове отсутствуют круглые скобки. Результат вывода зависит от компилятора.

Ответы: Урок №16

Ответ №1

Функция multiply() имеет тип возврата void, что означает, что эта функция не возвращает значения. Но, так как она все равно пытается возвратить значение с помощью оператора return, мы получим ошибку от компилятора. Функция должна иметь тип возврата int.

Ответ №2

Проблема №1: main() передает один аргумент в multiply(), но multiply() имеет два параметра.

Проблема №2: multiply() вычисляет результат и присваивает его локальной переменной, которую не возвращает обратно в main(). А поскольку тип возврата функции multiply() — int, то мы получим ошибку (в некоторых компиляторах) или неожиданные результаты (в остальных компиляторах).

Ответ №3

Функция multiply() принимает следующие параметры: `a = add(3, 4, 5)` и `b = 5`. Сначала процессор обрабатывает `a = add(3, 4, 5)`, т.е. `3 + 4 + 5 = 12`. Затем уже выполняет операцию умножения, результатом которой является `60`. Ответ: 60.

Ответ №4

```
int doubleNumber(int a)
{
   return 2 * a;
}
```

Ответ №5

```
#include <iostream>

int doubleNumber(int a)
{
   return 2 * a;
}

int main()
{
   int a;
   std::cout<<"Enter a number: ";
   std::cin >> a;
```

```
    std::cout << doubleNumber(a) << std::endl;
    return 0;
}

/*
// Следующее решение является альтернативным:
int main()
{
    int a;
    std::cout<<"Enter a number: ";
    std::cin >> a;
    a = doubleNumber(a);
    std::cout << a << std::endl;
    return 0;
}
*/
```

Примечание: У вас могут быть и другие решения заданий №4 и №5 — это ок. В программировании есть много случаев, когда одну задачу можно решить несколькими способами.

Ответы: Урок №18

Ответ

Результат выполнения программы:

```
main: a = 6 and b = 7
doMath: a = 6 and b = 5
doMath: a = 4 and b = 5
main: a = 6 and b = 7
```

Вот ход выполнения этой программы:

- выполнение начинается с функции main();
- создается переменная `a` в функции main(), ей присваивается значение `6`;
- создается переменная `b` в функции main(), ей присваивается значение `7`;
- cout выводит `main: a = 6 and b = 7`;
- вызывается doMath() с аргументом `6`;
- создается переменная `a` в функции doMath(), ей присваивается значение `6`;
- выполняется инициализация переменной `b` функции doMath() значением `5`;
- cout выводит `doMath: a = 6 and b = 5`;
- переменной `a` функции doMath() присваивается значение `4`;
- cout выводит `doMath: a = 4 and b = 5`;
- переменные `a` и `b` функции doMath() уничтожаются;
- cout выводит `main: a = 6 and b = 7`;
- функция main() возвращает `0` в операционную систему;
- переменные `a` и `b` функции main() уничтожаются.

Обратите внимание, даже когда мы присвоили значения переменным `a` и `b` внутри функции doMath(), на переменные внутри функции main() это никак не повлияло.

Ответы: Урок №19

Ответ

- `int result`. Всё ок.
- `int _oranges`. Имена переменных не должны начинаться с символов подчёркивания.
- `int NUMBER`. Имена переменных должны начинаться с буквы в нижнем регистре.
- `int the name of a variable`. Имена переменных не могут содержать пробелы.
- `int TotalCustomers`. Имена переменных должны начинаться с буквы в нижнем регистре.
- `int void`. `void` — это ключевое слово и его нельзя использовать в качестве идентификатора для своих переменных или функций.
- `int countFruit`. Всё ок.
- `int 4orYou`. Имена переменных не могут начинаться с цифр.
- `int kilograms_of_pipe`. Всё ок.

Ответы: Урок №22

Ответ №1

Прототип функции (полноценный) — это стейтмент объявления функции, который состоит из типа возврата функции, её имени и параметров (тип + имя параметра). В кратком прототипе отсутствуют имена параметров функции. Тело функции не записывается.

Предварительное объявление сообщает компилятору о существовании идентификатора до его фактического определения.

Для функций прототип является предварительным объявлением.

Ответ №2

```
// Любой из этих прототипов является правильным
// Не забывайте указывать точку с запятой в конце

int doMath(int first, int second, int third, int fourth); // лучшее решение
int doMath(int, int, int, int); // альтернативное решение
```

Ответ №3

Программа №1: Не скомпилируется. Компилятор будет жаловаться, что слишком много аргументов в вызове функции add().

Программа №2: Не скомпилируется. Компилятор будет жаловаться на то, что вызов функции add() не может принять столько аргументов.

Программа №3: Провал на этапе линкинга. Функция add(), которая принимает два параметра, не была определена (мы определили функцию, которая принимает 3 параметра).

Программа №4: Успешная компиляция и линкинг. Вызов функции add() соответствует и прототипу, который был объявлен, и определению функции.

Ответы: Урок №23

Ответ

input.cpp:

```
#include <iostream>

int getInteger()
{
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;
    return x;
}
```

main.cpp:

```
#include <iostream>

int getInteger(); // предварительное объявление функции getInteger()

int main()
{
    int x = getInteger();
    int y = getInteger();

    std::cout << x << " + " << y << " is " << x + y << '\n';
    return 0;
}
```

Ответы: Урок №26

Ответ

```
#ifndef ADD_H
#define ADD_H

int add(int x, int y);

#endif
```

Ответы: Глава №1. Итоговый тест

Ответ №1

main.cpp:

```
#include <iostream>

int readNumber()
{
    std::cout << "Enter a number: ";
    int x;
    std::cin >> x;
    return x;
}

void writeAnswer(int x)
{
    std::cout << "The answer is " << x << std::endl;
}

int main()
{
    int x = readNumber();
    int y = readNumber();
    writeAnswer(x + y); // передаем результат в функцию writeAnswer()
    return 0;
}
```

Ответ №2

io.cpp:

```
#include <iostream>

int readNumber()
{
    std::cout << "Enter a number: ";
    int x;
    std::cin >> x;
    return x;
}

void writeAnswer(int x)
{
    std::cout << "The answer is " << x << std::endl;
}
```

main.cpp:

```
// Это предварительные объявления функций, которые находятся в файле io.cpp
int readNumber();
void writeAnswer(int x);

int main()
```

```
{
    int x = readNumber();
    int y = readNumber();
    writeAnswer(x+y);
    return 0;
}
```

Ответ №3

io.cpp:

```
#include <iostream>

int readNumber()
{
    std::cout << "Enter a number: ";
    int x;
    std::cin >> x;
    return x;
}

void writeAnswer(int x)
{
    std::cout << "The answer is " << x << std::endl;
}
```

io.h:

```
#ifndef IO_H
#define IO_H

int readNumber();
void writeAnswer(int x);

#endif
```

main.cpp:

```
#include "io.h"

int main()
{
    int x = readNumber();
    int y = readNumber();
    writeAnswer(x+y);
    return 0;
}
```

Ответы: Урок №36

Ответ

- `3.450e1` (4 значащие цифры).
- `4.000e-3` (4 значащие цифры).
- `1.23005e2` (6 значащих цифр).
- `1.46e5` (3 значащие цифры).
- `1.46000001e5` (9 значащих цифр).
- `8e-10` (1 значащая цифра). Здесь мантисса не `8.0`, а `8`, поэтому число и имеет только 1 значащую цифру.
- `3.45000e4` (6 значащих цифр). Здесь конечные нули не игнорируются, так как в исходном числе есть точка, которая разделяет целую и дробную части. Хотя эта точка никак не влияет на само число, она влияет на его точность. Если бы исходное число было указано, как `34500`, то ответ равнялся бы `3.45e4`.

Ответы: Урок №37

Ответ

```
#include <iostream>

bool isPrime(int x)
{
    if (x == 2)
        return true;
    if (x == 3)
        return true;
    if (x == 5)
        return true;
    if (x == 7)
        return true;
    return false;
}

int main()
{
    std::cout << "Enter a single digit integer: ";
    int x;
    std::cin >> x;

    bool prime = isPrime(x);

    if (prime)
        std::cout << "The digit is prime" << std::endl;
    else
        std::cout << "The digit is not prime" << std::endl;

    return 0;
}
```

Ответы: Глава №2. Итоговый тест

Ответ №1

Использование литеральных констант (магических чисел) не только усложняет программу, но и затрудняет внесение в нее изменений. Символьные константы лучше, так как они дают понимание того, зачем и почему они используются, а также, если вам еще понадобится внести изменения — достаточно будет внести правки только в объявлении константы, а не искать их по всему коду. Значения констант, объявленных с помощью директивы #define, не отображаются в отладчике, вероятность возникновения конфликта имен у таких констант выше.

Ответ №2

- int8_t не сможет хранить возраст человека, старше 127. Несмотря на то, что таких случаев в мире единицы (если есть вообще), в будущем это может быть вполне возможным. Так что int16_t здесь подойдет лучше.
- bool.
- const double.
- Так как большинство книг имеют больше 255 страниц, но меньше чем 32767, то int16_t — здесь наилучший вариант.
- float.
- int32_t.
- char.

Ответ №3

```
#include <iostream>

double getDouble()
{
    std::cout << "Enter a double value: ";
    double x;
    std::cin >> x;
    return x;
}

char getOperator()
{
    std::cout << "Enter one of the following: +, -, *, or / ";
    char op;
    std::cin >> op;
    return op;
}

void printResult(double x, char op, double y)
{
```

```
    if (op == '+')
        std::cout << x << " + " << y << " = " << x + y << '\n';
    else if (op == '-')
        std::cout << x << " - " << y << " = " << x - y << '\n';
    else if (op == '*')
        std::cout << x << " * " << y << " = " << x * y << '\n';
    else if (op == '/')
        std::cout << x << " / " << y << " = " << x / y << '\n';
}

int main()
{
    double x = getDouble();
    double y = getDouble();

    char op = getOperator();

    printResult(x, op, y);

    return 0;
}
```

Ответ №4

Файл constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace myConstants
{
    const double gravity(9.8);
}
#endif
```

Файл main.cpp:

```
#include <iostream>
#include "constants.h"

// Получаем начальную высоту от пользователя и возвращаем её
double getInitialHeight()
{
    std::cout << "Enter the initial height of the tower in meters: ";
    double initialHeight;
    std::cin >> initialHeight;
    return initialHeight;
}

// Возвращаем расстояние от земли после "..." секунд падения
double calculateHeight(double initialHeight, int seconds)
{
    // Используем формулу: [ s = u * t + (a * t^2) / 2 ], где u(начальная
   скорость) = 0
    double distanceFallen = (myConstants::gravity * seconds * seconds) / 2;
    double currentHeight = initialHeight - distanceFallen;

    return currentHeight;
}
```

```
// Выводим высоту, на которой находится мячик после каждой секунды падения
void printHeight(double height, int seconds)
{
    if (height > 0.0)
        std::cout << "At " << seconds << " seconds, the ball is at height: " <<
   height << " meters\n";
    else
        std::cout << "At " << seconds << " seconds, the ball is on the ground.\n";
}

void calculateAndPrintHeight(double initialHeight, int seconds)
{
    double height = calculateHeight(initialHeight, seconds);
    printHeight(height, seconds);
}

int main()
{
    const double initialHeight = getInitialHeight();

    calculateAndPrintHeight(initialHeight, 0);
    calculateAndPrintHeight(initialHeight, 1);
    calculateAndPrintHeight(initialHeight, 2);
    calculateAndPrintHeight(initialHeight, 3);
    calculateAndPrintHeight(initialHeight, 4);
    calculateAndPrintHeight(initialHeight, 5);

    return 0;
}
```

Обратите внимание, функция calculateHeight() не выводит высоту на экран (помните о «правиле одного задания»). Мы используем отдельную функцию для вывода.

Ответы: Урок №41

Ответ

Выражение №1: `x = 3 + 4 + 5`

Уровень приоритета бинарного оператора `+` выше, чем оператора `=`, поэтому: `x = (3 + 4 + 5)`. Ассоциативность бинарного оператора `+` слева направо, поэтому **ответ**: `x = ((3 + 4) + 5)`.

Выражение №2: `x = y = z`

Ассоциативность бинарного оператора `=` справа налево, поэтому **ответ**: `x = (y = z)`.

Выражение №3: `z *= ++y + 5`

Унарный оператор `++` имеет наивысший приоритет, поэтому `z *= (++y) + 5`. Затем идет бинарный оператор `+`, поэтому **ответ**: `z *= ((++y) + 5)`.

Выражение №4: `a || b && c || d`

Бинарный оператор `&&` имеет приоритет выше, чем `||`, поэтому `a || (b && c) || d`. Ассоциативность бинарного оператора `||` слева направо, поэтому **ответ**: `(a || (b && c)) || d`.

Ответы: Урок №42

Ответ №1

Поскольку операторы `*` и `%` имеют более высокий приоритет, чем оператор `+`, то оператор `+` будет выполняться последним. Мы можем переписать наше выражение следующим образом: `6 + (5 * 4 % 3)`. Операторы `*` и `%` имеют одинаковый приоритет, но их ассоциативность слева направо, так что левый оператор будет выполняться первым. Получается `6 + ((5 * 4) % 3)`.

`6 + ((5 * 4) % 3) = 6 + (20 % 3) = 6 + 2 = 8`

Ответ: 8.

Ответ №2

```
#include <iostream>

bool isEven(int x)
{
    // Если x % 2 == 0, то x - это четное число
    return (x % 2) == 0;
}

int main()
{
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;

    if (isEven(x))
        std::cout << x << " is even\n";
    else
        std::cout << x << " is odd\n";

    return 0;
}
```

Возможно, вы хотели написать или написали функцию isEven() следующим образом:

```
bool isEven(int x)
{
    if ((x % 2) == 0)
        return true;
    else
        return false;
}
```

Хотя этот способ тоже рабочий, но он сложнее. Посмотрим, как его можно упростить. Во-первых, давайте вытащим условие if и присвоим его отдельной переменной типа bool:

```
bool isEven(int x)
{
    bool isEven = (x % 2) == 0;
    if (isEven) // isEven - true
        return true;
    else // isEven - false
        return false;
}
```

В коде, приведенном выше, если переменная `isEven` имеет значение `true`, то возвращаем `true`, в противном случае (если `isEven` имеет значение `false`) — возвращаем `false`. Мы же можем сразу возвращать `isEven`:

```
bool isEven(int x)
{
    bool isEven = (x % 2) == 0;
    return isEven;
}
```

Так как переменная `isEven` используется только один раз, то мы можем её вообще исключить:

```
bool isEven(int x)
{
    return (x % 2) == 0;
}
```

Ответы: Урок №46

Ответ

> ***Примечание:*** В ответах объяснение выполняется с помощью стрелочки (=>). Например, (`true || false`) => `true` означает, что результатом выражения (`true || false`) является `true`.

- *Выражение №1:* `(true && true) || false` => `true || false` => `true`
- *Выражение №2:* `(false && true) || true` => `false || true` => `true`
- *Выражение №3:* `(false && true) || false || true` => `false || false || true` => `false || true` => `true`
- *Выражение №4:* `(5 > 6 || 4 > 3) && (7 > 8)` => `(false || true) && false` => `true && false` => `false`
- *Выражение №5:* `!(7 > 6 || 3 > 4)` => `!(true || false)` => `!true` => `false`

Ответы: Урок №47

Ответ №1

Двоичный символ	0	1	0	0	1	1	0	1
* Значение символа	128	64	32	16	8	4	2	1
= Результат (77)	0	64	0	0	8	4	0	1

Ответ: 77.

Ответ №2

Используя способ №1:

```
93 / 2 = 46 r1
46 / 2 = 23 r0
23 / 2 = 11 r1
11 / 2 = 5 r1
5 / 2 = 2 r1
2 / 2 = 1 r0
1 / 2 = 0 r1
```

Остатки читаем снизу вверх и записываем в одну строку: 101 1101.

Ответ: 0101 1101.

Используя способ №2:

Наибольшее число, умноженное на 2, но которое меньше 93 — это 64.

```
93 >= 64? Да, 64-й бит равен 1. 93 - 64 = 29.
29 >= 32? Нет, 32-й бит равен 0.
29 >= 16? Да, 16-й бит равен 1. 29 - 16 = 13.
13 >= 8? Да, 8-й бит равен 1. 13 - 8 = 5.
5 >= 4? Да, 4-й бит равен 1. 5 - 4 = 1.
1 >= 2? Нет, 2-й бит равен 0.
1 >= 1? Да, 1-й бит равен 1.
```

Ответ: 0101 1101.

Ответ №3

```
Мы уже знаем из предыдущего примера, что 93 — это 0101 1101
Поэтому инвертируем биты: 1010 0010
И добавляем единицу: 1010 0011
```

Ответ: 1010 0011.

Ответ №4

Работая справа налево:

```
1010 0010 = (0 * 1) + (1 * 2) + (0 * 4) + (0 * 8) + (0 * 16) + (1 * 32) + (0 * 64) + (1 * 128) = 2 + 32 + 128 = 162
```

Ответ: 162.

Ответ №5

```
Имеем: 1010 0010
Инвертируем биты: 0101 1101
Добавляем единицу: 0101 1110
Конвертируем в десятичную систему счисления: 16 + 64 + 8 + 4 + 2 = 94
```

Так как здесь используется метод «two’s complement», а знаковый бит является отрицательным, то результат: -94

Ответ: -94.

Ответ №6

```
#include <iostream>

// x - это число, которое мы будем тестировать.
// pow - это множитель 2 (например, 128, 64, 32 и т.д.)
int printandDecrementBit(int x, int pow)
{
    // Проверяем, является ли x больше определенного числа, умноженного на 2 и выводим бит
    if (x >= pow)
        std::cout << "1";
    else
        std::cout << "0";

    // Если x больше, чем число, умноженное на 2, то вычитаем его из значения
    if (x >= pow)
        return x - pow;
    else
        return x;
```

```
}

int main()
{
    std::cout << "Enter an integer between 0 and 255: ";
    int x;
    std::cin >> x;

    x = printandDecrementBit(x, 128);
    x = printandDecrementBit(x, 64);
    x = printandDecrementBit(x, 32);
    x = printandDecrementBit(x, 16);

    std::cout << " ";

    x = printandDecrementBit(x, 8);
    x = printandDecrementBit(x, 4);
    x = printandDecrementBit(x, 2);
    x = printandDecrementBit(x, 1);

    return 0;
}
```

Ответы: Урок №48

Ответ №1

Результатом `0110 >> 2` является двоичное число `0001`.

Ответ №2

Выражение `5 | 12`:

```
0 1 0 1
1 1 0 0
--------
1 1 0 1 // 13 (десятичное)
```

Ответ №3

Выражение `5 & 12`:

```
0 1 0 1
1 1 0 0
--------
0 1 0 0 // 4 (десятичное)
```

Ответ №4

Выражение `5 ^ 12`:

```
0 1 0 1
1 1 0 0
--------
1 0 0 1 // 9 (десятичное)
```

Ответы: Урок №49

Ответ №1

```
myArticleFlags |= option_viewed;
```

Ответ №2

```
if (myArticleFlags & option_deleted) …
```

Ответ №3

```
myArticleFlags &= ~option_favorited;
```

Ответ №4

Законы Де Моргана гласят, что если мы используем побитовое НЕ, то операторы И и ИЛИ меняются местами. Следовательно, `~(option4 | option5)` становится `~option4 & ~option5`.

Ответы: Глава №3. Итоговый тест

Ответ №1

- `(5 > 3 && 4 < 8)` => `(true && true)`. Результат: true.
- `(4 > 6 && true)` => `(false && true)`. Результат: false.
- `(3 >= 3 || false)` => `(true || false)`. Результат: true.
- `(true || false) ? 4 : 5` => `(true ? 4 : 5)`. Результат: 4.

Ответ №2

- `7 / 4 = 1` с остатком `3`. Результат: 1.
- `14 % 5 = 2` с остатком `4`. Результат: 4.

Ответ №3

- 1101: `((1 * 8) + (1 * 4) + (0 * 2) + (1 * 1)) = 8 + 4 + 1 = 13`
- 101110: `((1 * 32) + (0 * 16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1)) = 32 + 8 + 4 + 2 = 46`

Ответ №4

Десятичное 15:

Используя метод №1:

```
15 / 2 = 7 r1
7 / 2 = 3 r1
3 / 2 = 1 r1
1 / 2 = 0 r1
```

Смотрим на остатки (снизу вверх): 1111.

Используя метод №2:

```
15 >= 8? Да, 8-й бит равен 1. Остается 7.
7 >= 4? Да, 4-й бит равен 1. Остается 3.
3 >= 2? Да, 2-й бит равен 1. Остается 1.
1 >= 1? Да, 1-й бит равен 1.
```

Результат: 1111.

Десятичное 53:

Используя метод №1:

```
53 / 2 = 26 r1
26 / 2 = 13 r0
13 / 2 = 6 r1
6 / 2 = 3 r0
3 / 2 = 1 r1
1 / 2 = 0 r1
```

Смотрим на остатки (снизу вверх): 110101.

Используя метод №2:

```
53 >= 32? Да, 32-й бит равен 1. Остается 21.
21 >= 16? Да, 16-й бит равен 1. Остается 5.
5 >= 8? Нет, 8-й бит равен 0.
5 >= 4? Да, 4-й бит равен 1. Остается 1.
1 > 2? Нет, 2-й бит равен 0.
1 >=1? Да, 1-й бит равен 1.
```

Таким образом, десятичное число 53 равно двоичному 110101.

Ответ №5

- Поскольку оператор `++` создает побочный эффект аргументу `x`, то мы не должны использовать переменную `x` дважды в этом выражении. Параметры функции foo() могут обрабатываться в любом порядке и нельзя определить, что будет первым (`x` или `++x`). Поскольку `++x` изменяет значение `x`, то непонятно, какие значения будут переданы в функцию.
- До C++11 непонятно, округлит ли компилятор это значение до `-3` или до `-4`.
- До C++11 непонятно, будет ли результатом `1` или `-1`.
- Ошибки округления со значениями типа с плавающей точкой приведут к результату `false`, хоть и кажется, что должно быть `true`.
- Деление на 0 приведет к сбою в программе.

Ответы: Урок №51

Ответ №1

```
#include <iostream>

int main()
{
    // Используем кириллицу
    setlocale(LC_ALL, "rus");

    std::cout << "Введите число: ";
    int smaller;
    std::cin >> smaller;

    std::cout << "Введите большее число: ";
    int larger;
    std::cin >> larger;

    // Если пользователь ввел числа не так, как нужно
    if (smaller > larger)
    {
        // То меняем местами эти значения
        std::cout << "Меняем значения местами\n";

        int temp = larger;
        larger = smaller;
        smaller = temp;
    } // temp уничтожается здесь

    std::cout << "Меньшее число: " << smaller << "\n";
    std::cout << "Большее число: " << larger << "\n";

    return 0;
} // smaller и larger уничтожаются здесь
```

Ответ №2

Область видимости определяет, где переменная доступна для использования. Продолжительность жизни переменной определяет, когда переменная создается и когда уничтожается.

Локальные переменные имеют локальную (блочную) область видимости, доступ к ним осуществляется только внутри блока, в котором они определены.

Локальные переменные имеют автоматическую продолжительность жизни, что означает, что они создаются в точке определения и уничтожаются в конце блока, в котором определены.

Ответы: Урок №52

Ответ

- Область видимости определяет, где переменная доступна для использования. Продолжительность жизни определяет, где переменная создается и где уничтожается. Связь определяет, может ли переменная использоваться в другом файле или нет.
- Глобальные переменные имеют глобальную область видимости (или «файловую область видимости»), что означает, что они доступны из точки объявления до конца файла, в котором объявлены.
- Глобальные переменные имеют статическую продолжительность жизни, что означает, что они создаются при запуске программы и уничтожаются при её завершении.
- Глобальные переменные могут иметь либо внутреннюю, либо внешнюю связь (это можно изменить через использование ключевых слов static и extern, соответственно).

Ответы: Урок №54

Ответ

Добавляя ключевое слово static к глобальной переменной, мы определяем её как внутреннюю, то есть такую, которую нельзя экспортировать и использовать в других файлах.

В случае с локальной переменной, добавление static определяет её как статическую, то есть она создается и инициализируется только один раз, и не уничтожается до самого конца программы.

Ответы: Урок №58

Ответ

Неявное преобразование происходит, когда компилятор ожидает значение одного типа, но получает значение другого типа.

Явное преобразование происходит, когда программист использует оператор явного преобразования для конвертации значения из одного типа данных в другой.

Ответы: Урок №60

Ответ

```
#include <iostream>
#include <string>

int main()
{
    std::cout << "Enter your full name: ";
    std::string myName;
    std::getline(std::cin, myName); // извлекаем целую строку из входного потока в переменную myName

    std::cout << "Enter your age: ";
    int myAge; // переменная myAge должна быть типа int, а не типа string, чтобы мы могли проводить с ней арифметические операции
    std::cin >> myAge;

    int letters = myName.length(); // вычисляем длину переменной myName (учитывая пробелы)
    double agePerLetter = static_cast<double>(myAge) / letters; // используем оператор static_cast, чтобы изменить тип переменной myAge на double, дабы сохранить дробную часть при целочисленном делении
    std::cout << "You've lived " << agePerLetter << " years for each letter in your name.\n";

    return 0;
}
```

Ответы: Урок №61

Ответ №1

```
enum MonsterType
{
    MONSTER_OGRE,
    MONSTER_GOBLIN,
    MONSTER_SKELETON,
    MONSTER_ORC,
    MONSTER_TROLL
};
```

Ответ №2

```
MonsterType eMonsterType = MONSTER_OGRE;
```

Ответ №3

Перечислителям можно:

- Правда.
- Правда. Перечислителю без значения будет неявно присвоено целочисленное значение предыдущего перечислителя +1. Если предыдущего перечислителя нет, то тогда присвоится значение `0`.
- Ложь.
- Правда. Поскольку значениями перечислителей являются целые числа, а целые числа можно присвоить перечислителям, то одни перечислители могут быть присвоены другим перечислителям (хотя этого лучше избегать).

Перечислители могут быть:

- Правда.
- Правда.

Ответы: Урок №63

Ответ №1

```
typedef int status_t;
status_t editData();
```

Ответ №2

```
using status_t = int;
status_t editData();
```

Ответы: Урок №64

Ответ №1

```
#include <iostream>

// Сначала объявляем структуру Advertising
struct Advertising
{
    int adsShown;
    double clickThroughRatePercentage;
    double averageEarningsPerClick;
};

void printAdvertising(Advertising ad)
{
    using namespace std;
    cout << "Number of ads shown: " << ad.adsShown << endl;
    cout << "Click through rate: " << ad.clickThroughRatePercentage << endl;
    cout << "Average earnings per click: $" << ad.averageEarningsPerClick <<
   endl;

    // Следующая строка кода разбита из-за своей длины.
    // Мы делим ad.clickThroughRatePercentage на 100, так как пользователь
   указывает проценты, а не готовое число
    cout << "Total Earnings: $" <<
        (ad.adsShown * ad.clickThroughRatePercentage / 100 *
   ad.averageEarningsPerClick) << endl;
}

int main()
{
    using namespace std;
    // Объявляем переменную структуры Advertising
    Advertising ad;

    cout << "How many ads were shown today? ";
    cin >> ad.adsShown;
    cout << "What percentage of users clicked on the ads? ";
    cin >> ad.clickThroughRatePercentage;
    cout << "What was the average earnings per click? ";
    cin >> ad.averageEarningsPerClick;

    printAdvertising(ad);

    return 0;
}
```

Ответ №2

```
#include <iostream>

struct Drob
{
    int chislitel;
    int znamenatel;
};
```

```
void multiply(Drob d1, Drob d2)
{
    using namespace std;

    // Не забываем об операторе static_cast, иначе компилятор выполнит
   целочисленное деление!
    cout << static_cast<float>(d1.chislitel* d2.chislitel) /
        (d1.znamenatel* d2.znamenatel);
}

int main()
{
    using namespace std;

    // Определяем первую переменную-дробь
    Drob d1;
    cout << "Input the first chislitel: ";
    cin >> d1.chislitel;
    cout << "Input the first znamenatel: ";
    cin >> d1.znamenatel;

    // Определяем вторую переменную-дробь
    Drob d2;
    cout << "Input the second chislitel: ";
    cin >> d2.chislitel;
    cout << "Input the second znamenatel: ";
    cin >> d2.znamenatel;

    multiply(d1, d2);

    return 0;
}
```

Ответы: Глава №4. Итоговый тест

Ответ C++11

```
#include <iostream>
#include <string>

// Определяем класс enum с типами монстров
enum class MonsterType
{
    OGRE,
    GOBLIN,
    SKELETON,
    ORC,
    TROLL
};

// Наша структура представляет одного монстра
struct Monster
{
    MonsterType type;
    std::string name;
    int health;
};

// Возвращаем тип монстра в виде строки
std::string getMonsterTypeString(Monster monster)
{
    if (monster.type == MonsterType::OGRE)
        return "Ogre";
    if (monster.type == MonsterType::GOBLIN)
        return "Goblin";
    if (monster.type == MonsterType::SKELETON)
        return "Skeleton";
    if (monster.type == MonsterType::ORC)
        return "Orc";
    if (monster.type == MonsterType::TROLL)
        return "Troll";

    return "Unknown";
}

// Выводим информацию о монстре
void printMonster(Monster monster)
{
    std::cout << "This " << getMonsterTypeString(monster);
    std::cout << " is named " << monster.name << " and has " << monster.health
   << " health.\n";
}

int main()
{
    Monster goblin = { MonsterType::GOBLIN, "John", 170 };
    Monster orc = { MonsterType::ORC, "James", 35 };

    printMonster(goblin);
    printMonster(orc);

    return 0;
```

```
}
```

Ответ до C++11

```
#include <iostream>
#include <string>

// Определяем перечисление с типами монстров
enum MonsterType
{
    MONSTER_OGRE,
    MONSTER_GOBLIN,
    MONSTER_SKELETON,
    MONSTER_ORC,
    MONSTER_TROLL
};

// Наша структура представляет одного монстра
struct Monster
{
    MonsterType type;
    std::string name;
    int health;
};

// Возвращаем тип монстра в виде строки
std::string getMonsterTypeString(Monster monster)
{
    if (monster.type == MONSTER_OGRE)
        return "Ogre";
    if (monster.type == MONSTER_GOBLIN)
        return "Goblin";
    if (monster.type == MONSTER_SKELETON)
        return "Skeleton";
    if (monster.type == MONSTER_ORC)
        return "Orc";
    if (monster.type == MONSTER_TROLL)
        return "Troll";

    return "Unknown";
}

// Выводим информацию о монстре
void printMonster(Monster monster)
{
    std::cout << "This " << getMonsterTypeString(monster);
    std::cout << " is named " << monster.name << " and has " << monster.health
   << " health.\n";
}

int main()
{
    Monster goblin = { MONSTER_GOBLIN, "John", 170};
    Monster orc = { MONSTER_ORC, "James", 35};

    printMonster(goblin);
    printMonster(orc);

    return 0;
}
```

Ответы: Урок №68

Ответ №1

```
#include <iostream>

int calculate(int x, int y, char op)
{
    switch (op)
    {
        case '+':
            return x + y;
        case '-':
            return x - y;
        case '*':
            return x * y;
        case '/':
            return x / y;
        case '%':
            return x % y;
        default:
            std::cout << "calculate(): Unhandled case\n";
            return 0;
    }
}

int main()
{
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;

    std::cout << "Enter another integer: ";
    int y;
    std::cin >> y;

    std::cout << "Enter a mathematical operator (+, -, *, /, or %): ";
    char op;
    std::cin >> op;

    std::cout << x << " " << op << " " << y << " is " << calculate(x, y, op) <<
   "\n";

    return 0;
}
```

Ответ №2

```
#include <iostream>
#include <string>

enum Animal
{
    ANIMAL_PIG,
    ANIMAL_CHICKEN,
    ANIMAL_GOAT,
    ANIMAL_CAT,
    ANIMAL_DOG,
```

```
    ANIMAL_OSTRICH
};

std::string getAnimalName(Animal animal)
{
    switch (animal)
    {
        case ANIMAL_CHICKEN:
            return "chicken";
        case ANIMAL_OSTRICH:
            return "ostrich";
        case ANIMAL_PIG:
            return "pig";
        case ANIMAL_GOAT:
            return "goat";
        case ANIMAL_CAT:
            return "cat";
        case ANIMAL_DOG:
            return "dog";

        default:
            return "getAnimalName(): Unhandled enumerator";
    }
}

void printNumberOfLegs(Animal animal)
{
    std::cout << "A " << getAnimalName(animal) << " has ";

    switch (animal)
    {
        case ANIMAL_CHICKEN:
        case ANIMAL_OSTRICH:
            std::cout << "2";
            break;

        case ANIMAL_PIG:
        case ANIMAL_GOAT:
        case ANIMAL_CAT:
        case ANIMAL_DOG:
            std::cout << "4";
            break;

        default:
            std::cout << "printNumberOfLegs(): Unhandled enumerator";
            break;
    }

    std::cout << " legs.\n";
}

int main()
{
    printNumberOfLegs(ANIMAL_CAT);
    printNumberOfLegs(ANIMAL_CHICKEN);

    return 0;
}
```

Ответы: Урок №70

Ответ №1

Переменная `inner` объявлена внутри блока while так, чтобы она была восстановлена (и повторно инициализирована значением `1`) каждый раз, когда выполняется внешний цикл. Если бы переменная `inner` была объявлена вне внешнего цикла while, то её значение никогда не было бы сброшено до `1`, или нам бы пришлось это сделать самостоятельно с помощью операции присваивания. Кроме того, поскольку переменная `inner` используется только внутри внешнего цикла while, то имеет смысл объявить её именно там. Помните, что переменные нужно объявлять максимально близко к их первому использованию!

Ответ №2

```
#include <iostream>

int main()
{
    char mychar = 'a';
    while (mychar <= 'z')
    {
        std::cout << mychar  << " " << static_cast<int>(mychar) << "\n";
        ++mychar;
    }

    return 0;
}
```

Ответ №3

```
#include <iostream>

int main()
{
    int outer = 5;
    while (outer >= 1)
    {
        int inner = outer;
        while (inner >= 1)
            std::cout << inner-- << " ";

        // Вставляем символ новой строки в конце каждого ряда
        std::cout << "\n";
        --outer;
        }

    return 0;
}
```

Ответ №4

```
#include <iostream>

int main()
{
    // Цикл с 1 до 5
    int outer = 1;

    while (outer <= 5)
    {
        // Числа в рядах появляются в порядке убывания, поэтому цикл начинаем
   с 5 и до 1
        int inner = 5;

        while (inner >= 1)
        {
            // Первое число в любом ряде совпадает с номером этого ряда,
   поэтому числа должны выводиться только если <= номера ряда (в противном
   случае, выводится пробел)
            if (inner <= outer)
                std::cout << inner << " ";
            else
                std::cout << "  "; // вставляем дополнительные пробелы

            --inner;
        }

        // Этот ряд вывели, переходим к следующему
        std::cout << "\n";

        ++outer;
    }
}
```

Ответы: Урок №72

Ответ №1

```
#include <iostream>

int main()
{
    for (int count = 0; count <= 20; count += 2)
        std::cout << count << std::endl;

    return 0;
}
```

Ответ №2

```
int sumTo(int value)
{
    int total(0);
    for (int count=1; count <= value; ++count)
        total += count;

    return total;
}
```

Ответ №3

Этот цикл for выполняется до тех пор, пока `count >= 0`. Другими словами, он работает до тех пор, пока переменная `count` не станет отрицательным числом. Однако, поскольку `count` является типа unsigned, то эта переменная никогда не сможет быть отрицательной. Следовательно, этот цикл бесконечный! Как правило, рекомендуется избегать использования типов unsigned в цикле, если на это нет веских причин.

Ответы: Урок №76

Ответ №1

Сразу, как только написали нетривиальную функцию.

Ответ №2

4-х тестов будет достаточно:

- Один для проверки случаев `a`/`e`/`i`/`o`/`u`.
- Один для проверки случая по умолчанию.
- Один для тестирования `isLowerVowel('y', true)`.
- И один для тестирования `isLowerVowel('y', false)`.

Ответы: Глава №5. Итоговый тест

Ответ №1

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace myConstants
{
    const double gravity(9.8);
}
#endif
```

Основной файл:

```
#include <iostream>
#include "constants.h"

// Получаем начальную высоту от пользователя и возвращаем её
double getInitialHeight()
{
    std::cout << "Enter the initial height of the tower in meters: ";
    double initialHeight;
    std::cin >> initialHeight;
    return initialHeight;
}

// Возвращаем расстояние от земли после "..." секунд падения
double calculateHeight(double initialHeight, int seconds)
{
    // Используем формулу: [ s = u * t + (a * t^2) / 2 ], где u (начальная
   скорость) = 0
    double distanceFallen = (myConstants::gravity * seconds * seconds) / 2;
    double currentHeight = initialHeight - distanceFallen;

    return currentHeight;
}

// Выводим высоту, на которой находится мячик после каждой секунды падения
void printHeight(double height, int seconds)
{
    if (height > 0.0)
        std::cout << "At " << seconds << " seconds, the ball is at height: "
   << height << " meters\n";
    else
        std::cout << "At " << seconds << " seconds, the ball is on the
   ground.\n";
}

int main()
{
    const double initialHeight = getInitialHeight();

    int seconds = 0;
    double height;
```

```
    do
    {
        height = calculateHeight(initialHeight, seconds);
        printHeight(height, seconds);
        ++seconds;
    } while (height > 0.0);

    return 0;
}
```

Ответ №2

```
#include <iostream>
#include <cstdlib> // для функций srand() и rand()
#include <ctime> // для функции time()

// Генерируем случайное число между min и max.
// Предполагается, что srand() уже вызывали
int getRandomNumber(int min, int max)
{
    static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);
    // Равномерно распределяем выбор случайного числа в диапазоне
    return static_cast<int>(rand() * fraction * (max - min + 1) + min);
}

// Возвращаем true, если пользователь выиграл, false - если проиграл
bool playGame(int guesses, int number)
{
    // Цикл работы с догадками пользователя
    for (int count = 1; count <= guesses; ++count)
    {
        std::cout << "Guess #" << count << ": ";
        int guess;
        std::cin >> guess;

        if (guess > number)
            std::cout << "Your guess is too high.\n";
        else if (guess < number)
            std::cout << "Your guess is too low.\n";
        else // догадка == число
            return true;
    }

    return false;
}

bool playAgain()
{
    // Продолжаем спрашивать у пользователя, хочет ли он сыграть еще раз до
   тех пор, пока он не нажмет 'y' или 'n'
    char ch;
    do
    {
        std::cout << "Would you like to play again (y/n)? ";
        std::cin >> ch;
    } while (ch != 'y' && ch != 'n');

    return (ch == 'y');
}
```

```
int main()
{
    srand(static_cast<unsigned int>(time(0))); // в качестве стартового числа используем системные часы
    rand(); // сбрасываем первый результат, так как функция rand() не особо хорошо рандомизирует первое случайное число в Visual Studio

    const int guesses = 7; // у пользователя есть 7 попыток

    do
    {
        int number = getRandomNumber(1, 100); // число, которое пользователь должен угадать

        std::cout << "Let's play a game.  I'm thinking of a number.  You have " << guesses << " tries to guess what it is.\n";

        bool won = playGame(guesses, number);
        if (won)
            std::cout << "Correct!  You win!\n";
        else
            std::cout << "Sorry, you lose.  The correct number was " << number << "\n";

    }
    while (playAgain());

    std::cout << "Thank you for playing.\n";
    return 0;
}
```

Ответы: Урок №78

Ответ №1

> ***Примечание:*** Если размер не является ограничением, то вместо типа float лучше использовать тип double.

```
double temperature[365] = { 0.0 };
```

Ответ №2

```
#include <iostream>

namespace Animals
{
    enum Animals
    {
        CHICKEN,
        LION,
        GIRAFFE,
        ELEPHANT,
        DUCK,
        SNAKE,
        MAX_ANIMALS
    };
}

int main()
{
    int legs[Animals::MAX_ANIMALS] = { 2, 4, 4, 4, 2, 0 };

    std::cout << "An elephant has " << legs[Animals::ELEPHANT] << " legs.\n";

    return 0;
}
```

Ответы: Урок №79

Ответ №1

```
#include <iostream>

int main()
{
    int array[] = { 7, 5, 6, 4, 9, 8, 2, 1, 3 };
    const int arrayLength = sizeof(array) / sizeof(array[0]);

    for (int index=0; index < arrayLength; ++index)
        std::cout << array[index] << " ";
    return 0;
}
```

Ответ №2

```
#include <iostream>

int main()
{
    // Сначала принимаем корректный пользовательский ввод
    int number = 0;
    do
    {
        std::cout << "Enter a number between 1 and 9: ";
        std::cin >> number;

        // Если пользователь ввел некорректное значение
        if (std::cin.fail())
            std::cin.clear();

        std::cin.ignore(32767, '\n');

    } while (number < 1 || number > 9);

    // Дальше выводим массив на экран
    int array[] = { 7, 5, 6, 4, 9, 8, 2, 1, 3 };
    const int arrayLength = sizeof(array) / sizeof(array[0]);

    for (int index=0; index < arrayLength; ++index)
        std::cout << array[index] << " ";

    std::cout << "\n";

    // Затем ищем в массиве число, которое ввел пользователь и выводим его
индекс
    for (int index=0; index < arrayLength; ++index)
    {
        if (array[index] == number)
        {
            std::cout <<  "The number " << number << " has index " <<  index
<< "\n";
            break; // так как каждый элемент в массиве уникальный, то нет
надобности продолжать перебирать элементы дальше
        }
    }
```

```
    return 0;
}
```

Ответ №3

```
#include <iostream>

int main()
{
    int students[] = { 73, 85, 84, 44, 78};
    const int numStudents = sizeof(students) / sizeof(students[0]);

    int maxIndex = 0; // отслеживаем индекс самого большого значения

    for (int person = 0; person < numStudents; ++person)
        if (students[person] > students[maxIndex])
            maxIndex = person;

    std::cout << "The best score was " << students[maxIndex] << '\n';

    return 0;
}
```

Ответы: Урок №80

Ответ №1

```
30 60 20 50 40 10
10 60 20 50 40 30
10 20 60 50 40 30
10 20 30 50 40 60
10 20 30 40 50 60
10 20 30 40 50 60 (самозамена)
10 20 30 40 50 60 (самозамена)
```

Ответ №2

Просто измените:

```
if (array[currentIndex] < array[smallestIndex])
```

на

```
if (array[currentIndex] > array[smallestIndex])
```

Также `smallestIndex` следует переименовать в `largestIndex`:

```
#include <iostream>
#include <algorithm> // для std::swap. В C++11 используйте заголовок <utility>

int main()
{
    const int length= 5;
    int array[length] = { 30, 50, 20, 10, 40 };

    // Перебираем каждый элемент массива, кроме последнего
    for (int startIndex = 0; startIndex < length - 1; ++startIndex)
    {
        // largestIndex - это индекс наибольшего элемента, который мы
    обнаружили до сих пор
        int largestIndex = startIndex;

        // Перебираем каждый элемент массива, начиная со startIndex + 1
        for (int currentIndex = startIndex + 1; currentIndex < length;
    ++currentIndex)
        {
            // Если текущий элемент больше нашего наибольшего элемента,
            if (array[currentIndex] > array[largestIndex])
                // то это новый наибольший элемент в этой итерации
                largestIndex = currentIndex;
        }

        // Меняем местами наше стартовое число с обнаруженным наибольшим
    элементом
        std::swap(array[startIndex], array[largestIndex]);
    }
```

```
    // Выводим отсортированный массив на экран
    for (int index = 0; index < length; ++index)
        std::cout << array[index] << ' ';

    return 0;
}
```

Ответ №3

```
#include <iostream>
#include <algorithm> // для std::swap. В C++11 используйте заголовок <utility>

int main()
{
    const int length(9);
    int array[length] = { 7, 5, 6, 4, 9, 8, 2, 1, 3 };

    for (int iteration = 0; iteration < length-1; ++iteration)
    {
        // Перебираем каждый элемент массива до последнего элемента (не
    включительно).
        // Последний элемент не имеет пары для сравнения
        for (int currentIndex = 0; currentIndex < length - 1; ++currentIndex)
        {
            // Если текущий элемент больше элемента, следующего за ним, то
    меняем их местами
            if (array[currentIndex] > array[currentIndex+1])
                std::swap(array[currentIndex], array[currentIndex + 1]);
        }
    }

    // Выводим отсортированный массив на экран
    for (int index = 0; index < length; ++index)
        std::cout << array[index] << ' ';

    return 0;
}
```

Ответ №4

```
#include <iostream>
#include <algorithm> // для std::swap. В C++11 используйте заголовок <utility>

int main()
{
    const int length(9);
    int array[length] = { 7, 5, 6, 4, 9, 8, 2, 1, 3 };

    for (int iteration = 0; iteration < length-1; ++iteration)
    {
        // Помните о том, что последний элемент будет отсортирован и в каждой
    последующей итерации цикла, поэтому наша сортировка «заканчивается» на один
    элемент раньше
        int endOfArrayIndex(length - iteration);

        bool swapped(false); // отслеживаем, были ли выполнены замены в этой
    итерации

        // Перебираем каждый элемент массива до последнего (не включительно).
```

```
        // Последний элемент не имеет пары для сравнения
        for (int currentIndex = 0; currentIndex < endOfArrayIndex - 1; ++currentIn
    dex)
        {
            // Если текущий элемент больше элемента, следующего за ним,
            if (array[currentIndex] > array[currentIndex + 1])
            {
                // то выполняем замену
                std::swap(array[currentIndex], array[currentIndex + 1]);
                swapped = true;
            }
        }

        // Если в этой итерации не выполнилось ни одной замены, то цикл можно
    завершать
        if (!swapped)
        {
            // Выполнение начинается с 0-й итерации, но мы привыкли
    считать, начиная с 1, поэтому для подсчета количества итераций добавляем
    единицу
            std::cout << "Early termination on iteration: " << iteration+1 <<
    '\n';
            break;
        }
    }

    // Выводим отсортированный массив на экран
    for (int index = 0; index < length; ++index)
        std::cout << array[index] << ' ';

    return 0;
}
```

Ответы: Урок №84

Ответ №1

Значения:

```
0012FF60
7
0012FF60
7

0012FF60
9
0012FF60
9

0012FF54
3
0012FF54
3

4
2
```

Краткое объяснение по поводу последней пары: `4` и `2`. 32-битное устройство означает, что размер указателя составляет 32 бита, но оператор sizeof всегда выводит размер в байтах: 32 бита = 4 байта. Таким образом, `sizeof(ptr)` равен `4`. Поскольку `ptr` является указателем на значение типа short, то `*ptr` является типа short. Размер short в этом примере составляет 2 байта. Таким образом, `sizeof(*ptr)` равен `2`.

Ответ №2

Последняя строка не скомпилируется. Рассмотрим эту программу детально.

В первой строке находится стандартное определение переменной вместе с инициализируемым значением. Здесь ничего особенного.

Во второй строке мы определяем новый указатель с именем `ptr` и присваиваем ему адрес переменной `value`. Помним, что в этом контексте звёздочка является частью синтаксиса объявления указателя, а не оператором разыменования. Так что и в этой строке всё нормально.

В третьей строке звёздочка уже является оператором разыменования, и используется для вытаскивания значения, на которое указывает указатель. Таким образом, эта строка говорит: «Вытаскиваем значение, на которое указывает `ptr` (целочисленное значение), и переписываем его на адрес этого же значения». А это уже какая-то чепуха — вы не можете присвоить адрес целочисленному значению!

Третья строка должна быть:

```
ptr = &value;
```

В вышеприведенной строке мы корректно присваиваем указателю адрес значения переменной.

Ответы: Урок №90

Ответ

```
#include <iostream>
#include <string>
#include <utility> // для std::swap(). Если у вас не поддерживается C++11, то тогда #include <algorithm>

void sortArray(std::string *array, int length)
{
    // Перебираем каждый элемент массива
    for (int startIndex = 0; startIndex < length; ++startIndex)
    {
        // smallestIndex - индекс наименьшего элемента, с которым мы столкнулись
        int smallestIndex = startIndex;

        // Ищем наименьший элемент, который остался в массиве (начиная со startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < length; ++currentIndex)
        {
            // Если текущий элемент меньше нашего ранее найденного наименьшего элемента,
            if (array[currentIndex] < array[smallestIndex])
                // то тогда это новое наименьшее значение в этой итерации
                smallestIndex = currentIndex;
        }

        // Меняем местами наш начальный элемент с найденным наименьшим элементом массива
        std::swap(array[startIndex], array[smallestIndex]);
    }
}

int main()
{
    std::cout << "How many names would you like to enter? ";
    int length;
    std::cin >> length;

    // Выделяем массив для хранения имен
    std::string *names = new std::string[length];

    // Просим пользователя ввести все имена
    for (int i = 0; i < length; ++i)
    {
        std::cout << "Enter name #" << i + 1 << ": ";
        std::cin >> names[i];
    }

    // Сортируем массив
    sortArray(names, length);

    std::cout << "\nHere is your sorted list:\n";
    // Выводим отсортированный массив
    for (int i = 0; i < length; ++i)
        std::cout << "Name #" << i + 1 << ": " << names[i] << '\n';
```

```
    delete[] names; // не забываем использовать оператор delete[] для
    освобождения памяти
        names = nullptr; // используйте 0, если не поддерживается C++11

    return 0;
}
```

Ответы: Урок №95

Ответ

```
#include <iostream>
#include <string>

int main()
{
    const std::string names[] = { "Sasha", "Ivan", "John", "Orlando",
   "Leonardo", "Nina", "Anton", "Molly" };

    std::cout << "Enter a name: ";
    std::string username;
    std::cin >> username;

    bool found(false);
    for (const auto &name : names)
        if (name == username)
        {
            found = true;
            break;
        }

    if (found)
        std::cout << username << " was found.\n";
    else
        std::cout << username << " was not found.\n";

    return 0;
}
```

Ответы: Урок №96

Ответ

Указатель типа void — это указатель, который может указывать на объект любого типа данных, но он сам не знает, какой это будет тип. Для разыменования указатель типа void должен быть явно преобразован с помощью оператора static_cast в другой тип данных. Нулевой указатель — это указатель, который не указывает на адрес. Указатель типа void может быть нулевым указателем.

Ответы: Глава №6. Итоговый тест

Ответ №1

```
#include <iostream>

enum ItemTypes
{
    ITEM_HEALTH_POTION,
    ITEM_TORCH,
    ITEM_ARROW,
    MAX_ITEMS
};

int countTotalItems(int *items) // нам здесь не нужно передавать длину массива, так как она уже указана членом MAX_ITEMS перечисления ItemTypes
{
    int totalItems = 0;
    for (int index = 0; index < MAX_ITEMS; ++index)
        totalItems += items[index];

    return totalItems;
}

int main()
{
    int items[MAX_ITEMS]{ 3, 6, 12 }; // используем uniform-инициализацию для указания стартового количества предметов, которые имеет при себе игрок (C++11)
//  int items[MAX_ITEMS] = { 3, 6, 12 }; // используйте список инициализаторов, если у вас не поддерживается C++11

    std::cout << "The player has " << countTotalItems(items) << " items in total.\n";

    return 0;
}
```

Ответ №2

```
#include <iostream>
#include <string>
#include <utility> // include <algorithm>, если не поддерживается C++11

struct Student
{
    std::string name;
    int grade;
};

// Функция сортировки студентов. Поскольку students - это указатель на массив, и он не знает длину массива (на который он указывает), то мы передаем длину явно, добавив параметр length
void sortNames(Student *students, int length)
{
    // Перебираем каждый элемент массива
    for (int startIndex = 0; startIndex < length; ++startIndex)
    {
```

```
        // largestIndex - это индекс наибольшего элемента, который мы обнаружили до сих пор
        int largestIndex = startIndex;

        // Ищем наибольший элемент среди оставшихся элементов массива (начиная со startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < length; ++currentIndex)
        {
            // Если текущий элемент больше нашего предыдущего наибольшего элемента,
            if (students[currentIndex].grade > students[largestIndex].grade)
                // то тогда это наш новый наибольший элемент в текущей итерации
                largestIndex = currentIndex;
        }

        // Меняем местами наш стартовый элемент с найденным наибольшим элементом
        std::swap(students[startIndex], students[largestIndex]);
    }
}

int main()
{
    int numStudents = 0;
    do
    {
        std::cout << "How many students do you want to enter? ";
        std::cin >> numStudents;
    } while (numStudents <= 1);

    // Динамически выделяем массив для хранения студентов
    Student *students = new Student[numStudents];

    // Записываем имя и оценку каждого студента
    for (int index = 0; index < numStudents; ++index)
    {
        std::cout << "Enter name #" << index + 1 << ": ";
        std::cin >> students[index].name;
        std::cout << "Enter grade #" << index + 1 << ": ";
        std::cin >> students[index].grade;
    }

    // Сортируем студентов
    sortNames(students, numStudents);

    // Выводим имена студентов и их оценки
    for (int index = 0; index < numStudents; ++index)
        std::cout << students[index].name << " got a grade of " << students[index].grade << "\n";

    // Не забываем об освобождении памяти
    delete[] students;

    return 0;
}
```

Ответ №3

```
#include <iostream>
```

```
// Используем в качестве параметров ссылки, чтобы иметь возможность изменить
   значения исходных аргументов
void swap(int &x, int &y)
{
    // Временно сохраняем значение переменной x в temp
    int temp = x;
    // Помещаем значение y в x
    x = y;
    // Помещаем предыдущее значение x в y
    y = temp;
}

int main()
{
    int x = 5;
    int y = 7;
    swap(x, y);

    if (x == 7 && y == 5)
        std::cout << "It works!";
    else
        std::cout << "It's broken!";

    return 0;
}
```

Ответ №4

```
#include <iostream>

// str указывает на первый символ строки C-style.
// Обратите внимание, str указывает на const char и мы не можем изменить это
   значение.
// Однако, мы можем заставить str указывать на что-либо другое. Это не приведет
   к изменению значения исходного аргумента
void printCString(const char *str)
{
    // Пока мы не встретили нуль-терминатор
    while (*str != '\0')
    {
        // Выводим текущий символ
        std::cout << *str;

        // И переводим указатель str на следующий символ
        ++str;
    }
}

int main()
{
    printCString("Hello, world!");

    return 0;
}
```

Ответ №5.a)

Цикл for имеет ошибку «неучтенной единицы» и пытается получить доступ к элементу массива под индексом 6, которого не существует. В условии цикла for нужно использовать оператор `<` вместо оператора `<=`.

Ответ №5.b)

`ptr` — это указатель на const int. Мы не можем присвоить ему значение `7`.

Ответ №5.c)

`array` распадается в указатель при передаче в функцию printArray(). Цикл foreach не работает с указателем на массив, так как указателю неизвестна длина массива, на который он указывает. Первое из решений — добавить параметр `length` в функцию printArray() и использовать обычный цикл for. Второе решение — использовать std::array вместо стандартных фиксированных массивов.

Ответ №5.d)

Мы не можем присвоить указателю типа int переменную не типа int. `ptr` должен быть типа double*.

Ответ №6.a)

```
enum CardSuit
{
    SUIT_TREFU,
    SUIT_BYBNU,
    SUIT_CHERVU,
    SUIT_PIKI,
    MAX_SUITS
};

enum CardRank
{
    RANK_2,
    RANK_3,
    RANK_4,
    RANK_5,
    RANK_6,
    RANK_7,
    RANK_8,
    RANK_9,
    RANK_10,
    RANK_VALET,
    RANK_DAMA,
    RANK_KOROL,
    RANK_TYZ,
```

```
    MAX_RANKS
};
```

Ответ №6.b)

```
struct Card
{
    CardRank rank;
    CardSuit suit;
};
```

Ответ №6.c)

```
void printCard(const Card &card)
{
    switch (card.rank)
    {
        case RANK_2:        std::cout << "2"; break;
        case RANK_3:        std::cout << "3"; break;
        case RANK_4:        std::cout << "4"; break;
        case RANK_5:        std::cout << "5"; break;
        case RANK_6:        std::cout << "6"; break;
        case RANK_7:        std::cout << "7"; break;
        case RANK_8:        std::cout << "8"; break;
        case RANK_9:        std::cout << "9"; break;
        case RANK_10:       std::cout << "T"; break;
        case RANK_VALET:    std::cout << "V"; break;
        case RANK_DAMA:     std::cout << "D"; break;
        case RANK_KOROL:    std::cout << "K"; break;
        case RANK_TYZ:      std::cout << "T"; break;
    }

    switch (card.suit)
    {
        case SUIT_TREFU:    std::cout << "TR"; break;
        case SUIT_BYBNU:    std::cout << "B"; break;
        case SUIT_CHERVU:   std::cout << "CH"; break;
        case SUIT_PIKI:     std::cout << "P"; break;
    }
}
```

Ответ №6.d)

```
int main()
{
    std::array<Card, 52> deck;

    int card = 0;
    for (int suit = 0; suit < MAX_SUITS; ++suit)
    for (int rank = 0; rank < MAX_RANKS; ++rank)
    {
        deck[card].suit = static_cast<CardSuit>(suit);
        deck[card].rank = static_cast<CardRank>(rank);
        ++card;
    }

    return 0;
}
```

Ответ №6.e)

```
void printDeck(const std::array<Card, 52> &deck)
{
    for (const auto &card : deck)
    {
        printCard(card);
        std::cout << ' ';
    }

    std::cout << '\n';
}
```

Ответ №6.f)

```
void swapCard(Card &a, Card &b)
{
    Card temp = a;
    a = b;
    b = temp;
}
```

Ответ №6.g)

```
#include <ctime> // для time()
#include <cstdlib> // для rand() и srand()

// Генерируем случайное число между min и max (предполагается, что функция
   srand() уже была вызвана)
int getRandomNumber(int min, int max)
{
    static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);
    // Равномерно распределяем генерацию случайного числа в диапазоне значений
    return static_cast<int>(rand() * fraction * (max - min + 1) + min);
}

void shuffleDeck(std::array<Card, 52> &deck)
{
    // Перебираем каждую карту в колоде
    for (int index = 0; index < 52; ++index)
    {
        // Выбираем любую случайную карту
        int swapIndex = getRandomNumber(0, 51);
        // Меняем местами с нашей текущей картой
        swapCard(deck[index], deck[swapIndex]);
    }
}

int main()
{
    srand(static_cast<unsigned int>(time(0))); // устанавливаем значение
   системных часов в качестве стартового числа
    rand(); // если используете Visual Studio, сбрасываем первое
   сгенерированное рандомное число

    std::array<Card, 52> deck;

    // Можно было бы вручную (по отдельности) инициализировать каждую карту, но мы
    ведь программисты! Цикл нам в помощь!
```

```
    int card = 0;
    for (int suit = 0; suit < MAX_SUITS; ++suit)
    for (int rank = 0; rank < MAX_RANKS; ++rank)
    {
        deck[card].suit = static_cast<CardSuit>(suit);
        deck[card].rank = static_cast<CardRank>(rank);
        ++card;
    }

    printDeck(deck);

    shuffleDeck(deck);

    printDeck(deck);

    return 0;
}
```

Ответ №6.h)

```
int getCardValue(const Card &card)
{
    switch (card.rank)
    {
    case RANK_2:         return 2;
    case RANK_3:         return 3;
    case RANK_4:         return 4;
    case RANK_5:         return 5;
    case RANK_6:         return 6;
    case RANK_7:         return 7;
    case RANK_8:         return 8;
    case RANK_9:         return 9;
    case RANK_10:        return 10;
    case RANK_VALET:     return 10;
    case RANK_DAMA:      return 10;
    case RANK_KOROL:     return 10;
    case RANK_TYZ:       return 11;
    }

    return 0;
}
```

Ответ №7

```
#include <iostream>
#include <array>
#include <ctime> // для time()
#include <cstdlib> // для rand() и srand()

enum CardSuit
{
    SUIT_TREFU,
    SUIT_BYBNU,
    SUIT_CHERVU,
    SUIT_PIKI,
    MAX_SUITS
};

enum CardRank
{
```

```
    RANK_2,
    RANK_3,
    RANK_4,
    RANK_5,
    RANK_6,
    RANK_7,
    RANK_8,
    RANK_9,
    RANK_10,
    RANK_VALET,
    RANK_DAMA,
    RANK_KOROL,
    RANK_TYZ,
    MAX_RANKS
};

struct Card
{
    CardRank rank;
    CardSuit suit;
};

void printCard(const Card &card)
{
    switch (card.rank)
    {
        case RANK_2:        std::cout << "2"; break;
        case RANK_3:        std::cout << "3"; break;
        case RANK_4:        std::cout << "4"; break;
        case RANK_5:        std::cout << "5"; break;
        case RANK_6:        std::cout << "6"; break;
        case RANK_7:        std::cout << "7"; break;
        case RANK_8:        std::cout << "8"; break;
        case RANK_9:        std::cout << "9"; break;
        case RANK_10:       std::cout << "T"; break;
        case RANK_VALET:    std::cout << "V"; break;
        case RANK_DAMA:         std::cout << "D"; break;
        case RANK_KOROL:    std::cout << "K"; break;
        case RANK_TYZ:      std::cout << "T"; break;
    }

    switch (card.suit)
    {
        case SUIT_TREFU:    std::cout << "TR"; break;
        case SUIT_BYBNU:    std::cout << "B"; break;
        case SUIT_CHERVU:   std::cout << "CH"; break;
        case SUIT_PIKI:         std::cout << "P"; break;
    }
}

void printDeck(const std::array<Card, 52> &deck)
{
    for (const auto &card : deck)
    {
        printCard(card);
        std::cout << ' ';
    }

    std::cout << '\n';
}

void swapCard(Card &a, Card &b)
```

```
{
    Card temp = a;
    a = b;
    b = temp;
}

// Генерируем случайное число между min и max (предполагается, что функция
   srand() уже была вызвана)
int getRandomNumber(int min, int max)
{
    static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);
    // Равномерно распределяем генерацию случайного числа в диапазоне значений
    return static_cast<int>(rand() * fraction * (max - min + 1) + min);
}

void shuffleDeck(std::array<Card, 52> &deck)
{
    // Перебираем каждую карту в колоде
    for (int index = 0; index < 52; ++index)
    {
        // Выбираем любую случайную карту
        int swapIndex = getRandomNumber(0, 51);
        // Меняем местами с нашей текущей картой
        swapCard(deck[index], deck[swapIndex]);
    }
}

int getCardValue(const Card &card)
{
    switch (card.rank)
    {
    case RANK_2:         return 2;
    case RANK_3:         return 3;
    case RANK_4:         return 4;
    case RANK_5:         return 5;
    case RANK_6:         return 6;
    case RANK_7:         return 7;
    case RANK_8:         return 8;
    case RANK_9:         return 9;
    case RANK_10:        return 10;
    case RANK_VALET:     return 10;
    case RANK_DAMA:          return 10;
    case RANK_KOROL:     return 10;
    case RANK_TYZ:       return 11;
    }

    return 0;
}

char getPlayerChoice()
{
    std::cout << "(h) to hit, or (s) to stand: ";
    char choice;
    do
    {
        std::cin >> choice;
    } while (choice != 'h' && choice != 's');

    return choice;
}

bool playBlackjack(const std::array<Card, 52> &deck)
```

```
{
    // Настраиваем стартовый режим игры
    const Card *cardPtr = &deck[0];

    int playerTotal = 0;
    int dealerTotal = 0;

    // Дилер получает одну карту
    dealerTotal += getCardValue(*cardPtr++);
    std::cout << "The dealer is showing: " << dealerTotal << '\n';

    // Игрок получает две карты
    playerTotal += getCardValue(*cardPtr++);
    playerTotal += getCardValue(*cardPtr++);

    // Игрок начинает
    while (1)
    {
        std::cout << "You have: " << playerTotal << '\n';

        // Смотрим, не больше ли 21 очка у игрока
        if (playerTotal > 21)
            return false;

        char choice = getPlayerChoice();
        if (choice == 's')
            break;

        playerTotal += getCardValue(*cardPtr++);
    }

    // Если игрок не проиграл и у него не больше 21 очка, то тогда
   дилер получает карты до тех пор, пока у него не получится в сумме 17 очков
    while (dealerTotal < 17)
    {
        dealerTotal += getCardValue(*cardPtr++);
        std::cout << "The dealer now has: " << dealerTotal << '\n';
    }

    // Если у дилера больше 21 очка, то игрок победил
    if (dealerTotal > 21)
        return true;

    return (playerTotal > dealerTotal);
}

int main()
{
    srand(static_cast<unsigned int>(time(0))); // устанавливаем значение системных
    часов в качестве стартового числа
    rand(); // если используете Visual Studio, сбрасываем первое
   сгенерированное рандомное число

    std::array<Card, 52> deck;

    // Можно было бы вручную (по отдельности) инициализировать каждую
   карту, но мы ведь программисты! Цикл нам в помощь!
    int card = 0;
    for (int suit = 0; suit < MAX_SUITS; ++suit)
    for (int rank = 0; rank < MAX_RANKS; ++rank)
    {
        deck[card].suit = static_cast<CardSuit>(suit);
```

```
            deck[card].rank = static_cast<CardRank>(rank);
            ++card;
        }

        shuffleDeck(deck);

        if (playBlackjack(deck))
            std::cout << "You win!\n";
        else
            std::cout << "You lose!\n";

        return 0;
}
```

Дополнительные задания

Ответ a)

Можно было бы отслеживать, сколько тузов игрок и дилер получили в отдельной целочисленной переменной-счетчике. Если у игрока или дилера результат превысил 21 очко, и их счетчик тузов больше нуля, то тогда уменьшается результат игрока или дилера на 10 (конвертируем туз из 11 очков в 1) и удаляется 1 из счетчика тузов. Продолжаться это будет до тех пор, пока счетчик тузов не достигнет нуля.

Ответ b)

Функция playBlackjack() сейчас возвращает `true`, если игрок выигрывает, и `false` — если проигрывает. Нужно обновить эту функцию, чтобы было три возможных исхода: победа дилера, победа игрока или ничья. Лучший способ это сделать — создать перечисление для этих 3-х вариантов + чтобы функция возвращала соответствующее значение из этого перечисления:

```
enum BlackjackResult
{
    BLACKJACK_PLAYER_WIN,
    BLACKJACK_DEALER_WIN,
    BLACKJACK_NICHIA
};

BlackjackResult playBlackjack(const std::array<Card, 52> &deck);
```

Ответы: Урок №106

Ответ №1

```
int sumTo(const int value);
```

Ответ №2

```
void printAnimalName(const Animal &animal);
```

Ответ №3

```
void minmax(const int a, const int b, int &minOut, int &maxOut);
```

Ответ №4

```
int getIndexOfLargestValue(const int *array, const int length);
```

Ответ №5

```
const int& getElement(const int *array, const int index);
```

Ответы: Урок №110

Ответ №1.a)

```
#include <iostream>

int getInteger()
{
    std::cout << "Enter an integer: ";
    int a;
    std::cin >> a;
    return a;
}

char getOperation()
{
    char op;

    do
    {
        std::cout << "Enter an operation ('+', '-', '*', '/'): ";
        std::cin >> op;
    }
    while (op!='+' && op!='-' && op!='*' && op!='/');

    return op;
}

int main()
{
    int a = getInteger();
    char op = getOperation();
    int b = getInteger();

    return 0;
}
```

Ответ №1.b)

```
int add(int a, int b)
{
    return a + b;
}

int subtract(int a, int b)
{
    return a - b;
}

int multiply(int a, int b)
{
    return a * b;
}

int divide(int a, int b)
{
    return a / b;
}
```

Ответ №1.c)

```
typedef int (*arithmeticFcn)(int, int);
```

Ответ №1.d)

```
arithmeticFcn getArithmeticFcn(char op)
{
    switch (op)
    {
    default: // функцией по умолчанию будет add()
    case '+': return add;
    case '-': return subtract;
    case '*': return multiply;
    case '/': return divide;
    }
}
```

Ответ №1.e)

```
#include <iostream>

int main()
{
    int a = getInteger();
    char op = getOperation();
    int b = getInteger();

    arithmeticFcn fcn = getArithmeticFcn(op);
    std::cout << a << ' ' << op << ' ' << b << " = " << fcn(a, b) << '\n';

    return 0;
}
```

Полная программа

```
#include <iostream>

int getInteger()
{
    std::cout << "Enter an integer: ";
    int a;
    std::cin >> a;
    return a;
}

char getOperation()
{
    char op;

    do
    {
        std::cout << "Enter an operation ('+', '-', '*', '/'): ";
        std::cin >> op;
    }
    while (op!='+' && op!='-' && op!='*' && op!='/');

    return op;
```

```
}

int add(int a, int b)
{
    return a + b;
}

int subtract(int a, int b)
{
    return a - b;
}

int multiply(int a, int b)
{
    return a * b;
}

int divide(int a, int b)
{
    return a / b;
}

typedef int (*arithmeticFcn)(int, int);

arithmeticFcn getArithmeticFcn(char op)
{
    switch (op)
    {
    default: // функцией по умолчанию будет add()
    case '+': return add;
    case '-': return subtract;
    case '*': return multiply;
    case '/': return divide;
    }
}

int main()
{
    int a = getInteger();
    char op = getOperation();
    int b = getInteger();

    arithmeticFcn fcn = getArithmeticFcn(op);
    std::cout << a << ' ' << op << ' ' << b << " = " << fcn(a, b) << '\n';

    return 0;
}
```

Ответ №2.a)

```
struct arithmeticStruct
{
    char op;
    arithmeticFcn fcn;
};
```

Ответ №2.b)

```
// Версия до C++11:
static arithmeticStruct arithmeticArray[] = {
```

```
    { '+', add },
    { '-', subtract },
    { '*', multiply },
    { '/', divide }
};

// Версия C++11 с использованием uniform-инициализации
static arithmeticStruct arithmeticArray[] {
    { '+', add },
    { '-', subtract },
    { '*', multiply },
    { '/', divide }
};
```

Ответ №2.c)

```
arithmeticFcn getArithmeticFcn(char op)
{
    for (auto &arith : arithmeticArray)
    {
        if (arith.op == op)
            return arith.fcn;
    }

    return add; // функцией по умолчанию будет add()
}
```

Полная программа

```
#include <iostream>

int getInteger()
{
    std::cout << "Enter an integer: ";
    int a;
    std::cin >> a;
    return a;
}

char getOperation()
{
    char op;

    do
    {
        std::cout << "Enter an operation ('+', '-', '*', '/'): ";
        std::cin >> op;
    } while (op != '+' && op != '-' && op != '*' && op != '/');

    return op;
}

int add(int a, int b)
{
    return a + b;
}

int subtract(int a, int b)
{
    return a - b;
```

```
}

int multiply(int a, int b)
{
    return a * b;
}

int divide(int a, int b)
{
    return a / b;
}

typedef int(*arithmeticFcn)(int, int);

struct arithmeticStruct
{
    char op;
    arithmeticFcn fcn;
};

static arithmeticStruct arithmeticArray[] {
    { '+', add },
    { '-', subtract },
    { '*', multiply },
    { '/', divide }
};


arithmeticFcn getArithmeticFcn(char op)
{
    for (auto &arith : arithmeticArray)
    {
        if (arith.op == op)
            return arith.fcn;
    }

    return add; // функцией по умолчанию будет add()
}

int main()
{
    int a = getInteger();
    char op = getOperation();
    int b = getInteger();

    arithmeticFcn fcn = getArithmeticFcn(op);
    std::cout << a << ' ' << op << ' ' << b << " = " << fcn(a, b) << '\n';

    return 0;
}
```

Ответы: Урок №113

Ответ №1

```
#include <iostream>

int factorial(int n)
{
    if (n < 1)
        return 1;
    else
        return factorial(n - 1) * n;
}

int main()
{
    for (int count = 0; count < 8; ++count)
        std::cout << factorial(count) << '\n';
}
```

Ответ №2

```
#include <iostream>

int sumNumbers(int x)
{
    if (x < 10)
        return x;
    else
        return sumNumbers(x / 10) + x % 10;
}

int main()
{
    std::cout << sumNumbers(83569) << std::endl;
}
```

Ответ №3

```
#include <iostream>

void printBinary(int x)
{
        // Условие завершения
        if (x == 0)
            return;

    // Рекурсия к следующему биту
    printBinary(x / 2);

    // Выводим остаток (в обратном порядке)
    std::cout << x % 2;
}

int main()
{
    int x;
```

```
    std::cout << "Enter an integer: ";
    std::cin >> x;

    printBinary(x);
}
```

Ответ №4

```
#include <iostream>

void printBinaryDigits(unsigned int n)
{
    // Условие завершения
    if (n == 0)
        return;

    printBinaryDigits(n / 2);

    std::cout << n % 2;
}

void printBinary(int n)
{
    if (n == 0)
        std::cout << '0'; // выводим "0", если n == 0
    else
        printBinaryDigits(static_cast<unsigned int>(n));
}

int main()
{
    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;

    printBinary(x);
}
```

Ответы: Урок №118

Ответ №1

```
#include <array>
#include <iostream>
#include <string>

struct Student
{
  std::string name{};
  int points{};
};

int main()
{
  std::array<Student, 8> arr{
    { { "Albert", 3 },
      { "Ben", 5 },
      { "Christine", 2 },
      { "Dan", 8 },
      { "Enchilada", 4 },
      { "Francis", 1 },
      { "Greg", 3 },
      { "Hagrid", 5 } }
  };

  auto best{
    std::max_element(arr.begin(), arr.end(), [](const auto& a, const auto& b) {
      return (a.points < b.points);
    })
  };

  std::cout << best->name << " is the best student\n";

  return 0;
}
```

Ответ №2

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

struct Season
{
  std::string_view name{};
  double averageTemperature{};
};

int main()
{
  std::array<Season, 4> seasons{
    { { "Spring", 285.0 },
      { "Summer", 296.0 },
      { "Fall", 288.0 },
      { "Winter", 263.0 } }
```

```
  };

  // Лямбде не нужно даже ничего захватывать, потому что порядок зависит только от
     элементов массива, которые мы получаем в качестве параметров.
  // Мы можем сравнить averageTemperature двух аргументов для сортировки массива
  std::sort(seasons.begin(), seasons.end(),
            [](const auto& a, const auto& b) {
              return (a.averageTemperature < b.averageTemperature);
            });

  for (const auto& season : seasons)
  {
    std::cout << season.name << '\n';
  }

  return 0;
}
```

Ответы: Урок №119

Ответ №1

Переменная	Использование без явного захвата
a	Нет. Переменная a имеет автоматическую продолжительность.
b	Да. Переменная b используется в константном выражении.
c	Да. Переменная c имеет статическую продолжительность.
d	Да.
e	Да. Переменная e используется в константном выражении.
f	Нет. Значение переменной f зависит от getValue(), что может потребовать запуска программы.
g	Да.
h	Да. Переменная h имеет статическую продолжительность.
i	Да. Переменная i является глобальной переменной.
j	Да. Переменная j доступна в целом файле.

Ответ №2

Результат выполнения программы:

```
I like grapes
```

Лямбда `printFavoriteFruit` захватывает `favoriteFruit` по значению. Изменение переменной `favoriteFruit` в функции main() никак не влияет на изменение переменной `favoriteFruit` в лямбде.

Ответ №3

```
#include <algorithm> // для std::generate(), std::find() и std::min_element()
#include <cmath> // для std::abs()
#include <ctime>
#include <iostream>
#include <random>
#include <vector>

using list_type = std::vector<int>;

namespace config
{
  constexpr int multiplierMin{ 2 };
  constexpr int multiplierMax{ 4 };
  constexpr int maximumWrongAnswer{ 4 };
}

int getRandomInt(int min, int max)
{
  static std::mt19937 mt{ static_cast<std::mt19937::result_type>(std::time(nullptr
   )) };

  return std::uniform_int_distribution{ min, max }(mt);
}

// Генерируем количество чисел, указанное в count, начиная со start*start, и
   умножаем каждое число в квадрате на умножитель (multiplier)
list_type generateNumbers(int start, int count, int multiplier)
{
  list_type numbers(static_cast<list_type::size_type>(count));

  int i{ start };

  for (auto& number : numbers)
  {
    number = ((i * i) * multiplier);
    ++i;
  }

  return numbers;
}

// Просим пользователя ввести стартовое число и общее количество чисел, а затем
   вызываем generateNumbers()
list_type generateUserNumbers(int multiplier)
{
  int start{};
  int count{};

  std::cout << "Start where? ";
  std::cin >> start;

  std::cout << "How many? ";
  std::cin >> count;

  // Здесь пропущена проверка пользовательского ввода. Все функции
   подразумевают корректный пользовательский ввод

  return generateNumbers(start, count, multiplier);
}
```

```
int getUserGuess()
{
  int guess{};

  std::cout << "> ";
  std::cin >> guess;

  return guess;
}

// Ищем значение guess в numbers и удаляем его.
// Возвращаем true, если значение было найдено, в противном случае – возвращаем
   false
bool findAndRemove(list_type& numbers, int guess)
{
  if (auto found{ std::find(numbers.begin(), numbers.end(), guess) };
      found == numbers.end())
  {
    return false;
  }
  else
  {
    numbers.erase(found);
    return true;
  }
}

// Находим значение в numbers, которое ближе всего к guess
int findClosestNumber(const list_type& numbers, int guess)
{
  return *std::min_element(numbers.begin(), numbers.end(), [=](int a, int b) {
    return (std::abs(a - guess) < std::abs(b - guess));
  });
}

void printTask(list_type::size_type count, int multiplier)
{
  std::cout << "I generated " << count
            << " square numbers. Do you know what each number is after
   multiplying it by "
            << multiplier << "?\n";
}

// Вызывается, когда пользователь правильно угадывает число
void printSuccess(list_type::size_type numbersLeft)
{
  std::cout << "Nice! ";

  if (numbersLeft == 0)
  {
    std::cout << "You found all numbers, good job!\n";
  }
  else
  {
    std::cout << numbersLeft << " number(s) left.\n";
  }
}

// Вызывается, когда пользователь указывает число, которого нет в numbers
void printFailure(const list_type& numbers, int guess)
{
```

```
  int closest{ findClosestNumber(numbers, guess) };

  std::cout << guess << " is wrong!";

  if (std::abs(closest - guess) <= config::maximumWrongAnswer)
  {
    std::cout << " Try " << closest << " next time.\n";
  }
  else
  {
    std::cout << '\n';
  }
}

// Возвращаем false, если игра окончена, в противном случае - возвращаем true
bool playRound(list_type& numbers)
{
  int guess{ getUserGuess() };

  if (findAndRemove(numbers, guess))
  {
    printSuccess(numbers.size());

    return !numbers.empty();
  }
  else
  {
    printFailure(numbers, guess);
    return false;
  }
}

int main()
{
  int multiplier{ getRandomInt(config::multiplierMin, config::multiplierMax) };
  list_type numbers{ generateUserNumbers(multiplier) };

  printTask(numbers.size(), multiplier);

  while (playRound(numbers));

  return 0;
}
```

Ответы: Глава №7. Итоговый тест

Ответ №1.a)

```
double max(double a, double b);
```

Ответ №1.b)

```
void swap(int &a, int &b);
```

Ответ №1.c)

```
// Примечание: array здесь не может быть константным, так как возврат
   неконстантной ссылки на константный элемент является нарушением

int& getLargestElement(int *array, int length);
```

Ответ №2.a)

Функция doSomething() возвращает ссылку на локальную переменную, которая будет уничтожена при завершении выполнения doSomething().

Ответ №2.b)

Рекурсивная функция sumTo() не имеет условия завершения. Значение переменной `value` в конечном итоге станет отрицательным, и функция будет выполняться бесконечно, вплоть до возникновения переполнения стека.

Ответ №2.c)

Две функции divide() не отличаются друг от друга никак, так как имеют одно и то же имя и одинаковые параметры.

Ответ №2.d)

Массив слишком велик для размещения в стеке. Его следует выделять динамически.

Ответ №2.e)

`argv[1]` может и не существовать вообще. Если же он существует, то это будет строка C-style, которая не может быть конвертирована в целое число через операцию присваивания.

Ответ №3.a)

```
// array - это массив, в котором мы проводим поиски.
// target - это искомое значение.
// min - это индекс минимальной границы массива, в котором мы осуществляем
   поиск.
// max - это индекс максимальной границы массива, в котором мы осуществляем
   поиск.
// Функция binarySearch() должна возвращать индекс искомого значения, если он
   обнаружен. В противном случае, возвращаем -1
int binarySearch(int *array, int target, int min, int max)
{
    while (min <= max)
    {
        // Итеративная реализация
        int midpoint = min + ((max-
   min) / 2); // такой способ вычисления середины массива избегает вероятность
   возникновения переполнения

        if (array[midpoint] > target)
        {
            // Если array[midpoint] > target, то тогда понимаем, что искомое
   число не находится в правой половине массива.
            // Мы можем использовать midpoint - 1 в качестве индекса
   максимальной границы, так как в следующей итерации этот индекс вычислять не
   нужно
            max = midpoint - 1;
        }
        else if (array[midpoint] < target)
        {
            // Если array[midpoint] < target, то тогда понимаем, что искомое
   число не находится в левой половине массива.
            // Мы можем использовать midpoint + 1 в качестве индекса
   минимальной границы, так как в следующей итерации этот индекс вычислять не
   нужно
            min = midpoint + 1;
        }
        else
            return midpoint;
    }

    return -1;
}
```

Ответ №3.b)

```
// array - это массив, в котором мы проводим поиски.
// target - это искомое значение.
// min - это индекс минимальной границы массива, в котором мы осуществляем
   поиск.
// max - это индекс максимальной границы массива, в котором мы осуществляем
   поиск.
// Функция binarySearch() должна возвращать индекс искомого значения, если он
   обнаружен. В противном случае, возвращаем -1
int binarySearch(int *array, int target, int min, int max)
{
    // Рекурсивная реализация

    if (min > max)
        return -1;
```

```
    int midpoint = min + ((max-
    min) / 2); // такой способ вычисления середины массива избегает вероятность
    возникновения переполнения

    if (array[midpoint] > target)
    {
        return binarySearch(array, target, min, midpoint - 1);
    }
    else if (array[midpoint] < target)
    {
        return binarySearch(array, target, midpoint + 1, max);
    }
    else
        return midpoint;
}
```

Ответы: Урок №121

Ответ №1

```
#include <iostream>

class Numbers
{
public:
    int m_first;
    int m_second;

    void set(int first, int second)
    {
        m_first = first;
        m_second = second;
    }
    void print()
    {
        std::cout << "Numbers(" << m_first << ", " << m_second << ")\n";
    }
};

int main()
{
    Numbers n1;
    n1.set(3, 3);

    Numbers n2{ 4, 4 };

    n1.print();
    n2.print();

    return 0;
}
```

Ответ №2

Класс Numbers содержит как переменные-члены, так и методы, поэтому мы должны использовать класс. Мы не должны использовать структуры с объектами, которые имеют методы.

Ответы: Урок №122

Ответ №1.a)

Открытый член — это член класса, доступ к которому имеют объекты как внутри, так и извне класса.

Ответ №1.b)

Закрытый член — это член класса, доступ к которому имеют только другие члены этого же класса.

Ответ №1.c)

Спецификатор доступа определяет, кто имеет доступ к членам этого же спецификатора.

Ответ №1.d)

В языке C++ есть 3 спецификатора доступа:

- public;
- private;
- protected.

Ответ №2.a)

```
#include <iostream>

class Numbers
{
private:
    double m_a, m_b, m_c;

public:
    void setValues(double a, double b, double c)
    {
        m_a = a;
        m_b = b;
        m_c = c;
    }

    void print()
    {
        std::cout << "<" << m_a << ", " << m_b << ", " << m_c << ">";
    }
};

int main()
```

```
{
    Numbers point;
    point.setValues(3.0, 4.0, 5.0);

    point.print();

    return 0;
}
```

Ответ №2.b)

```
#include <iostream>

class Numbers
{
private:
    double m_a, m_b, m_c;

public:
    void setValues(double a, double b, double c)
    {
        m_a = a;
        m_b = b;
        m_c = c;
    }

    void print()
    {
        std::cout << "<" << m_a << ", " << m_b << ", " << m_c << ">";
    }

    // Здесь мы можем использовать тот факт, что контроль доступа
    осуществляется на основе класса для того, чтобы получить доступ напрямую к
    закрытым членам объекта d класса Numbers
    bool isEqual(const Numbers &d)
    {
        return (m_a == d.m_a && m_b == d.m_b && m_c == d.m_c);
    }
};

int main()
{
    Numbers point1;
    point1.setValues(3.0, 4.0, 5.0);

    Numbers point2;
    point2.setValues(3.0, 4.0, 5.0);

    if (point1.isEqual(point2))
        std::cout << "point1 and point2 are equal\n";
    else
        std::cout << "point1 and point2 are not equal\n";

    Numbers point3;
    point3.setValues(7.0, 8.0, 9.0);

    if (point1.isEqual(point3))
        std::cout << "point1 and point3 are equal\n";
    else
        std::cout << "point1 and point3 are not equal\n";
```

```
    return 0;
}
```

Ответ №3

```
#include <iostream>
#include <cassert>

class Stack
{
private:
    int m_array[10]; // это будут данные нашего стека
    int m_next; // это будет индексом следующего свободного элемента стека

public:

    void reset()
    {
        m_next = 0;
        for (int i = 0; i < 10; ++i)
            m_array[i] = 0;
    }

    bool push(int value)
    {
        // Если стек уже заполнен, то возвращаем false
        if (m_next == 10)
            return false;

        m_array[m_next++] = value; // присваиваем следующему свободному
   элементу значение value, а затем увеличиваем m_next
        return true;
    }

    int pop()
    {
        // Если элементов в стеке нет, то выводим стейтмент assert
        assert (m_next > 0);

                // m_next указывает на следующий свободный элемент, поэтому
   последний элемент со значением - это m_next-1.
                // Мы хотим сделать следующее:
                // int val = m_array[m_next-
   1]; // получаем последний элемент со значением
                // --
   m_next; // m_next теперь на единицу меньше, так как мы только что вытянули
   верхний элемент стека
                // return val; // возвращаем элемент
                // Весь вышеприведенный код можно заменить следующей (одной)
   строкой кода
        return m_array[--m_next];
    }

    void print()
    {
        std::cout << "( ";
        for (int i = 0; i < m_next; ++i)
            std::cout << m_array[i] << ' ';
        std::cout << ")\n";
    }
};
```

```
int main()
{
    Stack stack;
    stack.reset();

    stack.print();

    stack.push(3);
    stack.push(7);
    stack.push(5);
    stack.print();

    stack.pop();
    stack.print();

    stack.pop();
    stack.pop();

    stack.print();

    return 0;
}
```

Ответы: Урок №124

Ответ №1.а)

```
#include <iostream>
#include <string>

class Ball
{
private:
    std::string m_color;
    double m_radius;

public:
        // Конструктор по умолчанию без параметров
        Ball()
        {
        m_color = "red";
        m_radius = 20.0;
        }

        // Конструктор с параметром color (для radius предоставлено значение по
   умолчанию)
    Ball(const std::string &color)
    {
        m_color = color;
        m_radius = 20.0;
    }

        // Конструктор с параметром radius (для color предоставлено значение по
   умолчанию)
    Ball(double radius)
    {
        m_color = "red";
        m_radius = radius;
    }

        // Конструктор с параметрами color и radius
    Ball(const std::string &color, double radius)
    {
        m_color = color;
        m_radius = radius;
    }

    void print()
    {
        std::cout << "color: " << m_color << ", radius: " << m_radius << '\n';
    }
};

int main()
{
    Ball def;
    def.print();

    Ball black("black");
    black.print();

    Ball thirty(30.0);
```

```
    thirty.print();

    Ball blackThirty("black", 30.0);
    blackThirty.print();

    return 0;
}
```

Ответ №1.b)

```
#include <iostream>
#include <string>

class Ball
{
private:
    std::string m_color;
    double m_radius;

public:
        // Конструктор с параметром radius (для color предоставлено значение по
   умолчанию)
    Ball(double radius)
    {
        m_color = "red";
        m_radius = radius;
    }

        // Конструктор с параметрами color и radius.
        // Обрабатываются такие случаи, как не предоставлено никаких
   параметров, только color, color + radius
    Ball(const std::string &color="red", double radius=20.0)
    {
        m_color = color;
        m_radius = radius;
    }

    void print()
    {
        std::cout << "color: " << m_color << ", radius: " << m_radius << '\n';
    }
};

int main()
{
    Ball def;
    def.print();

    Ball black("black");
    black.print();

    Ball thirty(30.0);
    thirty.print();

    Ball blackThirty("black", 30.0);
    blackThirty.print();

    return 0;
}
```

Ответ №2

Если не определить каких-либо конструкторов, то компилятор автоматически создаст пустой открытый конструктор по умолчанию. Если определить хотя бы один любой конструктор (по умолчанию или другой), то компилятор не создаст пустой конструктор по умолчанию.

Ответы: Урок №125

Ответ

```
#include <iostream>
#include <cstdint> // для std::uint8

class RGBA
{
private:
    std::uint8_t m_red;
    std::uint8_t m_green;
    std::uint8_t m_blue;
    std::uint8_t m_alpha;

public:
    RGBA(std::uint8_t red=0, std::uint8_t green=0, std::uint8_t blue=0,
   std::uint8_t alpha=255) :
        m_red(red), m_green(green), m_blue(blue), m_alpha(alpha)
    {
    }

    void print()
    {
        std::cout << "r=" << static_cast<int>(m_red) <<
            " g=" << static_cast<int>(m_green) <<
            " b=" << static_cast<int>(m_blue) <<
            " a=" << static_cast<int>(m_alpha) << '\n';
    }
};

int main()
{
    RGBA color(0, 135, 135);
    color.print();

    return 0;
}
```

Ответы: Урок №126

Ответ №1

```
#include <iostream>
#include <string>

class Thing
{
private:
    std::string m_color = "blue";
    double m_radius = 20.0;

public:
        // Конструктор по умолчанию без параметров (color и radius используют
   значения по умолчанию)
    Thing()
    {
    }

        // Конструктор с параметром color (для radius предоставлено значение по
   умолчанию)
    Thing(const std::string &color):
        m_color(color)
    {
    }

        // Конструктор с параметром radius (для color предоставлено значение по
   умолчанию)
    Thing(double radius):
        m_radius(radius)
    {
    }

        // Конструктор с параметрами color и radius
    Thing(const std::string &color, double radius):
        m_color(color), m_radius(radius)
    {
    }

    void print()
    {
        std::cout << "color: " << m_color << " and radius: " << m_radius <<
   '\n';
    }
};

int main()
{
    Thing defl;
    defl.print();

    Thing red("red");
    red.print();

    Thing thirty(30.0);
    thirty.print();

    Thing redThirty("red", 30.0);
```

```
    redThirty.print();

    return 0;
}
```

Ответ №2

Объект `defl` класса Thing будет искать конструктор по умолчанию. Если его не будет, то компилятор выдаст ошибку.

Ответы: Урок №134

Ответ а)

```
#include <iostream>

class Vector3D
{
private:
    double m_x, m_y, m_z;

public:
    Vector3D(double x = 0.0, double y = 0.0, double z = 0.0)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    void print()
    {
        std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";
    }

    friend class Point3D; // Point3D теперь является другом класса Vector3D
};


class Point3D
{
private:
    double m_x, m_y, m_z;

public:
    Point3D(double x = 0.0, double y = 0.0, double z = 0.0)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    void print()
    {
        std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z << ")\n";
    }


    void moveByVector(const Vector3D &v)
    {
        m_x += v.m_x;
        m_y += v.m_y;
        m_z += v.m_z;
    }
};


int main()
{
    Point3D p(3.0, 4.0, 5.0);
    Vector3D v(3.0, 3.0, -2.0);
```

```
    p.print();
    p.moveByVector(v);
    p.print();

    return 0;
}
```

Ответ b)

```
class Vector3D; // сначала говорим компилятору, что класс с именем Vector3D существует

class Point3D
{
private:
    double m_x, m_y, m_z;

public:
    Point3D(double x = 0.0, double y = 0.0, double z = 0.0)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    void print()
    {
        std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z << ")\n";
    }

    void moveByVector(const Vector3D &v); // чтобы мы могли использовать Vector3D здесь
       // Примечание: Мы не можем определить эту функцию здесь, так как Vector3D еще не был определен (компилятор увидит только его предварительное объявление)
};

class Vector3D
{
private:
    double m_x, m_y, m_z;

public:
    Vector3D(double x = 0.0, double y = 0.0, double z = 0.0)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    void print()
    {
        std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";
    }

    friend void Point3D::moveByVector(const Vector3D &v); // Point3D::moveByVector() теперь является другом класса Vector3D
};

// Теперь, когда Vector3D был определен, мы можем определить функцию Point3D::moveByVector()
void Point3D::moveByVector(const Vector3D &v)
{
```

```
    m_x += v.m_x;
    m_y += v.m_y;
    m_z += v.m_z;
}

int main()
{
    Point3D p(3.0, 4.0, 5.0);
    Vector3D v(3.0, 3.0, -2.0);

    p.print();
    p.moveByVector(v);
    p.print();

    return 0;
}
```

Ответ c)

Point3D.h:

```
// Определение класса Point3D

#ifndef POINT3D_H
#define POINT3D_H

class Vector3D; // предварительное объявление класса Vector3D для функции
   moveByVector()

class Point3D
{
    private:
        double m_x;
        double m_y;
        double m_z;

    public:
        Point3D(double x = 0.0, double y = 0.0, double z = 0.0) : m_x(x),
   m_y(y), m_z(z) {}

        void print();
        void moveByVector(const Vector3D &v); // предварительное объявление,
   приведенное выше, нужно для выполнения этой строки
};

#endif
```

Point3D.cpp:

```
// Определение методов класса Point3D

#include <iostream> // для std::cout
#include "Point3D.h" // класс Point3D определен здесь
#include "Vector3D.h" // для параметра функции moveByVector()

void Point3D::moveByVector(const Vector3D &v)
{
    // Добавляем координаты вектора к соответствующим координатам точки
    m_x += v.m_x;
```

```
    m_y += v.m_y;
    m_z += v.m_z;
}

void Point3D::print()
{
    std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z << ")\n";
}
```

Vector3D.h:

```
// Определение класса Vector3D

#ifndef VECTOR3D_H
#define VECTOR3D_H

#include "Point3D.h"

class Vector3D
{
    private:
        double m_x;
        double m_y;
        double m_z;

    public:
        Vector3D(double x = 0.0, double y = 0.0, double z = 0.0) : m_x(x),
   m_y(y), m_z(z) {}

        void print();
        friend void Point3D::moveByVector(const Vector3D &v);
};

#endif
```

Vector3D.cpp:

```
// Определение методов класса Vector3D

#include <iostream>
#include "Vector3D.h" // класс Vector3D определен в этом файле

void Vector3D::print()
{
    std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";
}
```

main.cpp:

```
#include "Vector3D.h" // для создания объекта класса Vector3D
#include "Point3D.h" // для создания объекта класса Point3D

int main()
{
    Point3D p(3.0, 4.0, 5.0);
    Vector3D v(3.0, 3.0, -2.0);

    p.print();
    p.moveByVector(v);
```

```
    p.print();

    return 0;
}
```

Ответы: Глава №8. Итоговый тест

Ответ №1.a)

```
#include <iostream>

class Point
{
private:
    double m_a;
    double m_b;

public:
    Point(double a = 0.0, double b = 0.0)
        : m_a(a), m_b(b)
    {
    }

    void print() const
    {
        std::cout << "Point(" << m_a << ", " << m_b << ")\n";
    }
};


int main()
{
   Point first;
   Point second(2.0, 5.0);
   first.print();
   second.print();

   return 0;
}
```

Ответ №1.b)

```
#include <iostream>
#include <cmath>

class Point
{
private:
    double m_a;
    double m_b;

public:
    Point(double a = 0.0, double b = 0.0)
        : m_a(a), m_b(b)
    {
    }

    void print() const
    {
        std::cout << "Point(" << m_a << ", " << m_b << ")\n";
    }

    double distanceTo(const Point & other) const
```

```
    {
        return sqrt((m_a - other.m_a)*(m_a - other.m_a) + (m_b - other.m_b)*(m_b -
    other.m_b));
    }
};

int main()
{
    Point first;
    Point second(2.0, 5.0);
    first.print();
    second.print();
    std::cout << "Distance between two points: " << first.distanceTo(second)
   << '\n';

    return 0;
}
```

Ответ №1.c)

```
#include <iostream>
#include <cmath>

class Point
{
private:
    double m_a;
    double m_b;

public:
    Point(double a = 0.0, double b = 0.0)
        : m_a(a), m_b(b)
    {
    }

    void print() const
    {
        std::cout << "Point(" << m_a << ", " << m_b << ")\n";
    }

    friend double distanceFrom(const Point &a, const Point &b);

};

double distanceFrom(const Point &a, const Point &b)
{
    return sqrt((a.m_a - b.m_a)*(a.m_a - b.m_a) + (a.m_b - b.m_b)*(a.m_b -
   b.m_b));
}

int main()
{
    Point first;
    Point second(2.0, 5.0);
    first.print();
    second.print();
    std::cout << "Distance between two points: " << distanceFrom(first, second)
   << '\n';

    return 0;
}
```

Ответ №2

```
#include <iostream>

class Welcome
{
private:
    char *m_data;

public:
    Welcome()
    {
        m_data = new char[14];
        const char *init = "Hello, World!";
        for (int i = 0; i < 14; ++i)
            m_data[i] = init[i];
    }

    ~Welcome()
    {
        delete[] m_data;
    }

    void print() const
    {
        std::cout << m_data;
    }

};

int main()
{
    Welcome hello;
    hello.print();

    return 0;
}
```

Ответ №3.a)

```
enum MonsterType
{
    Dragon,
    Goblin,
    Ogre,
    Orc,
    Skeleton,
    Troll,
    Vampire,
    Zombie,
    MAX_MONSTER_TYPES
};
```

Ответ №3.b)

```
#include <string>

enum MonsterType
{
```

```
    Dragon,
    Goblin,
    Ogre,
    Orc,
    Skeleton,
    Troll,
    Vampire,
    Zombie,
    MAX_MONSTER_TYPES
};

class Monster
{
private:

    MonsterType m_type;
    std::string m_name;
    int m_health;
};
```

Ответ №3.c)

```
#include <string>

class Monster
{
public:
    enum MonsterType
    {
        Dragon,
        Goblin,
        Ogre,
        Orc,
        Skeleton,
        Troll,
        Vampire,
        Zombie,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    int m_health;
};
```

Ответ №3.d)

```
#include <string>

class Monster
{
public:
    enum MonsterType
    {
        Dragon,
        Goblin,
        Ogre,
        Orc,
```

```
        Skeleton,
        Troll,
        Vampire,
        Zombie,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    int m_health;

public:
    Monster(MonsterType type, std::string name, int health)
        : m_type(type), m_name(name), m_health(health)
    {

    }
};

int main()
{
    Monster jack(Monster::Orc, "Jack", 90);

    return 0;
}
```

Ответ №3.e)

```
#include <iostream>
#include <string>

class Monster
{
public:
    enum MonsterType
    {
        Dragon,
        Goblin,
        Ogre,
        Orc,
        Skeleton,
        Troll,
        Vampire,
        Zombie,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    int m_health;

public:
    Monster(MonsterType type, std::string name, int health)
        : m_type(type), m_name(name), m_health(health)
    {

    }
```

```
    std::string getTypeString() const
    {
        switch (m_type)
        {
        case Dragon: return "dragon";
        case Goblin: return "goblin";
        case Ogre: return "ogre";
        case Orc: return "orc";
        case Skeleton: return "skeleton";
        case Troll: return "troll";
        case Vampire: return "vampire";
        case Zombie: return "zombie";
        }

        return "Error!";
    }

    void print() const
    {
        std::cout << m_name << " is the " << getTypeString() << " that has "
   << m_health << " health points."<< '\n';
    }
};

int main()
{
    Monster jack(Monster::Orc, "Jack", 90);
    jack.print();

    return 0;
}
```

Ответ №3.f)

```
#include <iostream>
#include <string>

class Monster
{
public:
    enum MonsterType
    {
        Dragon,
        Goblin,
        Ogre,
        Orc,
        Skeleton,
        Troll,
        Vampire,
        Zombie,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    int m_health;

public:
```

```
    Monster(MonsterType type, std::string name, int health)
        : m_type(type), m_name(name), m_health(health)
    {

    }

    std::string getTypeString() const
    {
        switch (m_type)
        {
        case Dragon: return "dragon";
        case Goblin: return "goblin";
        case Ogre: return "ogre";
        case Orc: return "orc";
        case Skeleton: return "skeleton";
        case Troll: return "troll";
        case Vampire: return "vampire";
        case Zombie: return "zombie";
        }

        return "Error!";
    }

    void print() const
    {
        std::cout << m_name << " is the " << getTypeString() << " that has "
   << m_health << " health points." << '\n';
    }
};

class MonsterGenerator
{
public:
    static Monster generateMonster()
    {
        return Monster(Monster::Orc, "Jack", 90);
    }
};

int main()
{
    Monster m = MonsterGenerator::generateMonster();
    m.print();

    return 0;
}
```

Ответ №3.g)

```
class MonsterGenerator
{
public:
    // Генерируем случайное число между min и max (включительно).
    // Предполагается, что srand() уже был вызван
    static int getRandomNumber(int min, int max)
    {
        static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0)
    ; // используем static, так как это значение нужно вычислить единожды
        // Равномерно распределяем вычисление значения из нашего диапазона
        return static_cast<int>(rand() * fraction * (max - min + 1) + min);
    }
```

```
    static Monster generateMonster()
    {
        return Monster(Monster::Orc, "Jack", 90);
    }
};
```

Ответ №3.h)

```
#include <iostream>
#include <ctime> // для time()
#include <cstdlib> // для rand() и srand()
#include <string>

class Monster
{
public:
    enum MonsterType
    {
        Dragon,
        Goblin,
        Ogre,
        Orc,
        Skeleton,
        Troll,
        Vampire,
        Zombie,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    int m_health;

public:
    Monster(MonsterType type, std::string name, int health)
        : m_type(type), m_name(name), m_health(health)
    {

    }

    std::string getTypeString() const
    {
        switch (m_type)
        {
        case Dragon: return "dragon";
        case Goblin: return "goblin";
        case Ogre: return "ogre";
        case Orc: return "orc";
        case Skeleton: return "skeleton";
        case Troll: return "troll";
        case Vampire: return "vampire";
        case Zombie: return "zombie";
        }

        return "Error!";
    }

    void print() const
```

```
    {
        std::cout << m_name << " is the " << getTypeString() << " that has "
    << m_health << " health points." << '\n';
    }
};

class MonsterGenerator
{
public:
    // Генерируем случайное число между min и max (включительно).
    // Предполагается, что srand() уже был вызван
    static int getRandomNumber(int min, int max)
    {
        static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) +
    1.0); // используем static, так как это значение нужно вычислить единожды
        // Равномерно распределяем вычисление значения из нашего диапазона
        return static_cast<int>(rand() * fraction * (max - min + 1) + min);
    }

    static Monster generateMonster()
    {
        Monster::MonsterType type = static_cast<Monster::MonsterType>(getRandomNum
    ber(0, Monster::MAX_MONSTER_TYPES - 1));
        int health = getRandomNumber(1, 100);

        static std::string s_names[6]{ "John", "Brad", "Alex", "Thor", "Hulk",
    "Asnee" };
        return Monster(type, s_names[getRandomNumber(0, 5)], health);
    }
};

int main()
{
    srand(static_cast<unsigned int>(time(0))); // используем системные часы в
    качестве стартового значения
    rand(); // пользователям Visual Studio: делаем сброс первого случайного
    числа

    Monster m = MonsterGenerator::generateMonster();
    m.print();

    return 0;
}
```

Ответ №3.i)

Мы объявили `s_names` статическим, так как инициализировать его нужно только один раз. В противном случае, он повторно инициализировался бы каждый раз при вызове generateMonster().

Ответ №4.a)

```
#include <iostream>

class Card
{
public:
    enum CardSuit
```

```
    {
        SUIT_CLUB,
        SUIT_DIAMOND,
        SUIT_HEART,
        SUIT_SPADE,
        MAX_SUITS
    };

    enum CardRank
    {
        RANK_2,
        RANK_3,
        RANK_4,
        RANK_5,
        RANK_6,
        RANK_7,
        RANK_8,
        RANK_9,
        RANK_10,
        RANK_JACK,
        RANK_QUEEN,
        RANK_KING,
        RANK_ACE,
        MAX_RANKS
    };

private:
    CardRank m_rank;
    CardSuit m_suit;

public:
    Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
        m_rank(rank), m_suit(suit)
    {

    }

    void printCard() const
    {
        switch (m_rank)
        {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
        }

        switch (m_suit)
        {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
```

```
        }
    }

    int getCardValue() const
    {
        switch (m_rank)
        {
        case RANK_2:         return 2;
        case RANK_3:         return 3;
        case RANK_4:         return 4;
        case RANK_5:         return 5;
        case RANK_6:         return 6;
        case RANK_7:         return 7;
        case RANK_8:         return 8;
        case RANK_9:         return 9;
        case RANK_10:        return 10;
        case RANK_JACK:      return 10;
        case RANK_QUEEN:     return 10;
        case RANK_KING:      return 10;
        case RANK_ACE:       return 11;
        }

        return 0;
    }
};

int main()
{
    const Card cardQueenHearts(Card::RANK_QUEEN, Card::SUIT_HEART);
    cardQueenHearts.printCard();
    std::cout << " has the value " << cardQueenHearts.getCardValue() << '\n';

    return 0;
}
```

Ответ №4.b)

```
#include <iostream>
#include <array>
#include <ctime> // для time()
#include <cstdlib> // для rand() и srand()

class Card
{
public:
    enum CardSuit
    {
        SUIT_CLUB,
        SUIT_DIAMOND,
        SUIT_HEART,
        SUIT_SPADE,
        MAX_SUITS
    };

    enum CardRank
    {
        RANK_2,
        RANK_3,
        RANK_4,
        RANK_5,
        RANK_6,
```

```
        RANK_7,
        RANK_8,
        RANK_9,
        RANK_10,
        RANK_JACK,
        RANK_QUEEN,
        RANK_KING,
        RANK_ACE,
        MAX_RANKS
    };

private:
    CardRank m_rank;
    CardSuit m_suit;

public:

    Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
        m_rank(rank), m_suit(suit)
    {

    }

    void printCard() const
    {
        switch (m_rank)
        {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
        }

        switch (m_suit)
        {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
        }
    }

    int getCardValue() const
    {
        switch (m_rank)
        {
        case RANK_2:        return 2;
        case RANK_3:        return 3;
        case RANK_4:        return 4;
        case RANK_5:        return 5;
        case RANK_6:        return 6;
        case RANK_7:        return 7;
        case RANK_8:        return 8;
```

```
        case RANK_9:        return 9;
        case RANK_10:       return 10;
        case RANK_JACK:     return 10;
        case RANK_QUEEN:    return 10;
        case RANK_KING:     return 10;
        case RANK_ACE:      return 11;
        }

        return 0;
    }
};

class Deck
{
private:
    std::array<Card, 52> m_deck;

    // Генерируем случайное число между min и max (включительно).
    // Предполагается, что srand() уже был вызван
    static int getRandomNumber(int min, int max)
    {
        static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0)
    ; // используем static, так как это значение нужно вычислить единожды
        // Равномерно распределяем вычисление значения из нашего диапазона
        return static_cast<int>(rand() * fraction * (max - min + 1) + min);
    }

    static void swapCard(Card &a, Card &b)
    {
        Card temp = a;
        a = b;
        b = temp;
    }

public:
    Deck()
    {
        int card = 0;
        for (int suit = 0; suit < Card::MAX_SUITS; ++suit)
            for (int rank = 0; rank < Card::MAX_RANKS; ++rank)
            {
                m_deck[card] = Card(static_cast<Card::CardRank>(rank),
    static_cast<Card::CardSuit>(suit));
                ++card;
            }
    }

    void printDeck() const
    {
        for (const auto &card : m_deck)
        {
            card.printCard();
            std::cout << ' ';
        }

        std::cout << '\n';
    }


    void shuffleDeck()
    {
        // Перебираем каждую карту в колоде
```

```
        for (int index = 0; index < 52; ++index)
        {
            // Выбираем любую случайную карту
            int swapIndex = getRandomNumber(0, 51);
            // Меняем местами с нашей текущей картой
            swapCard(m_deck[index], m_deck[swapIndex]);
        }
    }
};

int main()
{
    srand(static_cast<unsigned int>(time(0))); // используем системные
   часы в качестве стартового значения
    rand(); // пользователям Visual Studio: делаем сброс первого
   случайного числа

    Deck deck;
    deck.printDeck();
    deck.shuffleDeck();
    deck.printDeck();

    return 0;
}
```

Ответ №4.c)

```
#include <iostream>
#include <array>
#include <ctime> // для time()
#include <cstdlib> // для rand() и srand()
#include <cassert> // для assert()

class Card
{
public:
    enum CardSuit
    {
        SUIT_CLUB,
        SUIT_DIAMOND,
        SUIT_HEART,
        SUIT_SPADE,
        MAX_SUITS
    };

    enum CardRank
    {
        RANK_2,
        RANK_3,
        RANK_4,
        RANK_5,
        RANK_6,
        RANK_7,
        RANK_8,
        RANK_9,
        RANK_10,
        RANK_JACK,
        RANK_QUEEN,
        RANK_KING,
        RANK_ACE,
        MAX_RANKS
```

```
    };

private:
    CardRank m_rank;
    CardSuit m_suit;

public:

    Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
        m_rank(rank), m_suit(suit)
    {

    }

    void printCard() const
    {
        switch (m_rank)
        {
        case RANK_2:         std::cout << '2'; break;
        case RANK_3:         std::cout << '3'; break;
        case RANK_4:         std::cout << '4'; break;
        case RANK_5:         std::cout << '5'; break;
        case RANK_6:         std::cout << '6'; break;
        case RANK_7:         std::cout << '7'; break;
        case RANK_8:         std::cout << '8'; break;
        case RANK_9:         std::cout << '9'; break;
        case RANK_10:        std::cout << 'T'; break;
        case RANK_JACK:      std::cout << 'J'; break;
        case RANK_QUEEN:     std::cout << 'Q'; break;
        case RANK_KING:      std::cout << 'K'; break;
        case RANK_ACE:       std::cout << 'A'; break;
        }

        switch (m_suit)
        {
        case SUIT_CLUB:      std::cout << 'C'; break;
        case SUIT_DIAMOND:   std::cout << 'D'; break;
        case SUIT_HEART:     std::cout << 'H'; break;
        case SUIT_SPADE:     std::cout << 'S'; break;
        }
    }

    int getCardValue() const
    {
        switch (m_rank)
        {
        case RANK_2:         return 2;
        case RANK_3:         return 3;
        case RANK_4:         return 4;
        case RANK_5:         return 5;
        case RANK_6:         return 6;
        case RANK_7:         return 7;
        case RANK_8:         return 8;
        case RANK_9:         return 9;
        case RANK_10:        return 10;
        case RANK_JACK:      return 10;
        case RANK_QUEEN:     return 10;
        case RANK_KING:      return 10;
        case RANK_ACE:       return 11;
        }

        return 0;
```

```
    }
};

class Deck
{
private:
    std::array<Card, 52> m_deck;
    int m_cardIndex = 0;

    // Генерируем случайное число между min и max (включительно).
    // Предполагается, что srand() уже был вызван
    static int getRandomNumber(int min, int max)
    {
        static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0)
    ; // используем static, так как это значение нужно вычислить единожды
        // Равномерно распределяем вычисление значения из нашего диапазона
        return static_cast<int>(rand() * fraction * (max - min + 1) + min);
    }

    static void swapCard(Card &a, Card &b)
    {
        Card temp = a;
        a = b;
        b = temp;
    }

public:
    Deck()
    {
        int card = 0;
        for (int suit = 0; suit < Card::MAX_SUITS; ++suit)
            for (int rank = 0; rank < Card::MAX_RANKS; ++rank)
            {
                m_deck[card] = Card(static_cast<Card::CardRank>(rank),
    static_cast<Card::CardSuit>(suit));
                ++card;
            }
    }

    void printDeck() const
    {
        for (const auto &card : m_deck)
        {
            card.printCard();
            std::cout << ' ';
        }

        std::cout << '\n';
    }


    void shuffleDeck()
    {
        // Перебираем каждую карту в колоде
        for (int index = 0; index < 52; ++index)
        {
            // Выбираем любую случайную карту
            int swapIndex = getRandomNumber(0, 51);
            // Меняем местами с нашей текущей картой
            swapCard(m_deck[index], m_deck[swapIndex]);
        }
```

```
    m_cardIndex = 0; // начинаем новую раздачу карт
    }

    const Card& dealCard()
    {
        assert (m_cardIndex < 52);
        return m_deck[m_cardIndex++];
    }
};

int main()
{
    srand(static_cast<unsigned int>(time(0))); // используем системные
   часы в качестве стартового значения
    rand(); // пользователям Visual Studio: делаем сброс первого
   случайного числа

    Deck deck;
    deck.shuffleDeck();
    deck.printDeck();
    std::cout << "The first card has value: " << deck.dealCard().getCardValue()
   << '\n';
    std::cout << "The second card has value: " << deck.dealCard().getCardValue()
   << '\n';

    return 0;
}
```

Ответ №4.d)

```
#include <iostream>
#include <array>
#include <ctime> // для time()
#include <cstdlib> // для rand() и srand()
#include <cassert> // для assert()

class Card
{
public:
    enum CardSuit
    {
        SUIT_CLUB,
        SUIT_DIAMOND,
        SUIT_HEART,
        SUIT_SPADE,
        MAX_SUITS
    };

    enum CardRank
    {
        RANK_2,
        RANK_3,
        RANK_4,
        RANK_5,
        RANK_6,
        RANK_7,
        RANK_8,
        RANK_9,
        RANK_10,
        RANK_JACK,
        RANK_QUEEN,
```

```
        RANK_KING,
        RANK_ACE,
        MAX_RANKS
    };

private:
    CardRank m_rank;
    CardSuit m_suit;

public:

    Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
        m_rank(rank), m_suit(suit)
    {

    }

    void printCard() const
    {
        switch (m_rank)
        {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
        }

        switch (m_suit)
        {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
        }
    }

    int getCardValue() const
    {
        switch (m_rank)
        {
        case RANK_2:        return 2;
        case RANK_3:        return 3;
        case RANK_4:        return 4;
        case RANK_5:        return 5;
        case RANK_6:        return 6;
        case RANK_7:        return 7;
        case RANK_8:        return 8;
        case RANK_9:        return 9;
        case RANK_10:       return 10;
        case RANK_JACK:     return 10;
        case RANK_QUEEN:    return 10;
        case RANK_KING:     return 10;
        case RANK_ACE:      return 11;
```

```
        }

        return 0;
    }
};

class Deck
{
private:
    std::array<Card, 52> m_deck;
    int m_cardIndex = 0;

    // Генерируем случайное число между min и max (включительно).
    // Предполагается, что srand() уже был вызван
    static int getRandomNumber(int min, int max)
    {
        static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0)
    ; // используем static, так как это значение нужно вычислить единожды
        // Равномерно распределяем вычисление значения из нашего диапазона
        return static_cast<int>(rand() * fraction * (max - min + 1) + min);
    }

    static void swapCard(Card &a, Card &b)
    {
        Card temp = a;
        a = b;
        b = temp;
    }

public:
    Deck()
    {
        int card = 0;
        for (int suit = 0; suit < Card::MAX_SUITS; ++suit)
            for (int rank = 0; rank < Card::MAX_RANKS; ++rank)
            {
                m_deck[card] = Card(static_cast<Card::CardRank>(rank),
    static_cast<Card::CardSuit>(suit));
                ++card;
            }
    }

    void printDeck() const
    {
        for (const auto &card : m_deck)
        {
            card.printCard();
            std::cout << ' ';
        }

        std::cout << '\n';
    }


    void shuffleDeck()
    {
        // Перебираем каждую карту в колоде
        for (int index = 0; index < 52; ++index)
        {
            // Выбираем любую случайную карту
            int swapIndex = getRandomNumber(0, 51);
            // Меняем местами с нашей текущей картой
```

```
            swapCard(m_deck[index], m_deck[swapIndex]);
        }

    m_cardIndex = 0; // начинаем новую раздачу карт
    }

    const Card& dealCard()
    {
        assert (m_cardIndex < 52);
        return m_deck[m_cardIndex++];
    }
};


char getPlayerChoice()
{
    std::cout << "(h) to hit, or (s) to stand: ";
    char choice;
    do
    {
        std::cin >> choice;
    } while (choice != 'h' && choice != 's');

    return choice;
}

bool playBlackjack(Deck &deck)
{

    int playerTotal = 0;
    int dealerTotal = 0;

    // Дилер получает одну карту
    dealerTotal += deck.dealCard().getCardValue();
    std::cout << "The dealer is showing: " << dealerTotal << '\n';

    // Игрок получает две карты
    playerTotal += deck.dealCard().getCardValue();
    playerTotal += deck.dealCard().getCardValue();

    // Игрок начинает
    while (1)
    {
        std::cout << "You have: " << playerTotal << '\n';
        char choice = getPlayerChoice();
        if (choice == 's')
            break;

        playerTotal += deck.dealCard().getCardValue();

        // Смотрим, не проиграл ли игрок
        if (playerTotal > 21)
            return false;
    }

    // Если игрок не проиграл (у него не больше 21 очка), тогда дилер
   получает карты до тех пор, пока у него будет не меньше 17 очков
    while (dealerTotal < 17)
    {
        dealerTotal += deck.dealCard().getCardValue();
        std::cout << "The dealer now has: " << dealerTotal << '\n';
    }
```

```
    // Если дилер проиграл, то игрок выиграл
    if (dealerTotal > 21)
        return true;

    return (playerTotal > dealerTotal);
}

int main()
{
    srand(static_cast<unsigned int>(time(0))); // используем системные
   часы в качестве стартового значения
    rand(); // пользователям Visual Studio: делаем сброс первого
   случайного числа

    Deck deck;
    deck.shuffleDeck();

    if (playBlackjack(deck))
        std::cout << "You win!\n";
    else
        std::cout << "You lose!\n";

    return 0;
}
```

Ответы: Урок №139

Ответ a)

```
#include <iostream>

class Fraction
{
private:
    int m_numerator = 0;
    int m_denominator = 1;

public:
    Fraction(int numerator=0, int denominator=1):
        m_numerator(numerator), m_denominator(denominator)
    {
    }

    void print()
    {
        std::cout << m_numerator << "/" << m_denominator << "\n";
    }
};

int main()
{
    Fraction f1(1, 4);
    f1.print();

    Fraction f2(1, 2);
    f2.print();

    return 0;
}
```

Ответ b)

```
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction(int numerator=0, int denominator=1):
        m_numerator(numerator), m_denominator(denominator)
    {
    }

    friend Fraction operator*(const Fraction &f1, const Fraction &f2);
    friend Fraction operator*(const Fraction &f1, int value);
    friend Fraction operator*(int value, const Fraction &f1);

    void print()
    {
        std::cout << m_numerator << "/" << m_denominator << "\n";
```

```
    }
};

Fraction operator*(const Fraction &f1, const Fraction &f2)
{
    return Fraction(f1.m_numerator * f2.m_numerator, f1.m_denominator *
   f2.m_denominator);
}

Fraction operator*(const Fraction &f1, int value)
{
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}

Fraction operator*(int value, const Fraction &f1)
{
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}

int main()
{
    Fraction f1(3, 4);
    f1.print();

    Fraction f2(2, 7);
    f2.print();

    Fraction f3 = f1 * f2;
    f3.print();

    Fraction f4 = f1 * 3;
    f4.print();

    Fraction f5 = 3 * f2;
    f5.print();

    Fraction f6 = Fraction(1, 2) * Fraction(2, 3) * Fraction(3, 4);
    f6.print();

    return 0;
}
```

Ответ c)

```
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction(int numerator=0, int denominator=1):
        m_numerator(numerator), m_denominator(denominator)
    {
        // Мы поместили метод reduce() в конструктор, чтобы убедиться, что все
   дроби, которые у нас есть, будут уменьшены!
        // Поскольку выполнение всех перегруженных операторов осуществляется
   вместе с созданием новых объектов класса Fraction, то мы можем гарантировать,
   что эта функция вызовется для каждой дроби
```

```
        reduce();
    }

        // Делаем функцию nod статической, чтобы она могла быть частью класса
    Fraction и, при этом, для её использования нам не нужно было бы создавать
    объект класса Fraction
    static int nod(int a, int b)
    {
        return (b == 0) ? (a > 0 ? a : -a) : nod(b, a % b);
    }

    void reduce()
    {
        int nod = Fraction::nod(m_numerator, m_denominator);
        m_numerator /= nod;
        m_denominator /= nod;
    }

    friend Fraction operator*(const Fraction &f1, const Fraction &f2);
    friend Fraction operator*(const Fraction &f1, int value);
    friend Fraction operator*(int value, const Fraction &f1);

    void print()
    {
        std::cout << m_numerator << "/" << m_denominator << "\n";
    }
};

Fraction operator*(const Fraction &f1, const Fraction &f2)
{
    return Fraction(f1.m_numerator * f2.m_numerator, f1.m_denominator *
    f2.m_denominator);
}

Fraction operator*(const Fraction &f1, int value)
{
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}

Fraction operator*(int value, const Fraction &f1)
{
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}

int main()
{
    Fraction f1(3, 4);
    f1.print();

    Fraction f2(2, 7);
    f2.print();

    Fraction f3 = f1 * f2;
    f3.print();

    Fraction f4 = f1 * 3;
    f4.print();

    Fraction f5 = 3 * f2;
    f5.print();

    Fraction f6 = Fraction(1, 2) * Fraction(2, 3) * Fraction(3, 4);
```

```
    f6.print();

    return 0;
}
```

Ответы: Урок №141

Ответ

```
#include <iostream>

class Fraction
{
private:
    int m_numerator = 0;
    int m_denominator = 1;

public:
    Fraction(int numerator=0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        // Мы поместили метод reduce() в конструктор, чтобы убедиться, что все дроби, которые у нас есть, будут уменьшены!
        // Любые дроби, которые перезаписаны, должны быть повторно уменьшены
        reduce();
    }

    static int nod(int a, int b)
    {
        return b == 0 ? a : nod(b, a % b);
    }

    void reduce()
    {
        int nod = Fraction::nod(m_numerator, m_denominator);
        m_numerator /= nod;
        m_denominator /= nod;
    }

    friend Fraction operator*(const Fraction &f1, const Fraction &f2);
    friend Fraction operator*(const Fraction &f1, int value);
    friend Fraction operator*(int value, const Fraction &f1);

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);
    friend std::istream& operator>>(std::istream& in, Fraction &f1);

    void print()
    {
        std::cout << m_numerator << "/" << m_denominator << "\n";
    }
};

Fraction operator*(const Fraction &f1, const Fraction &f2)
{
    return Fraction(f1.m_numerator * f2.m_numerator, f1.m_denominator * f2.m_denominator);
}

Fraction operator*(const Fraction &f1, int value)
{
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}

Fraction operator*(int value, const Fraction &f1)
```

```
{
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

std::istream& operator>>(std::istream& in, Fraction &f1)
{
    char c;

    // Перезаписываем значения объекта f1
    in >> f1.m_numerator;
    in >> c; // игнорируем разделитель '/'
    in >> f1.m_denominator;

    // Поскольку мы перезаписали существующий f1, то нам нужно повторно
    выполнить уменьшение дроби
    f1.reduce();

    return in;
}

int main()
{

    Fraction f1;
    std::cout << "Enter fraction 1: ";
    std::cin >> f1;

    Fraction f2;
    std::cout << "Enter fraction 2: ";
    std::cin >> f2;

    std::cout << f1 << " * " << f2 << " is " << f1 * f2 << '\n';

    return 0;
}
```

Ответы: Урок №143

Ответ

Есть два решения.

Решение №1:

```
Something Something::operator+ () const
{
    return Something(m_a, m_b, m_c);
}
```

Решение №2:

```
Something Something::operator+ () const
{
    return *this;
}
```

Это работает, так как Something, который мы возвращаем, является текущим объектом.

Ответы: Урок №144

Ответ №1

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    friend bool operator> (const Dollars &d1, const Dollars &d2);
    friend bool operator<= (const Dollars &d1, const Dollars &d2);

    friend bool operator< (const Dollars &d1, const Dollars &d2);
    friend bool operator>= (const Dollars &d1, const Dollars &d2);
};

bool operator> (const Dollars &d1, const Dollars &d2)
{
    return d1.m_dollars > d2.m_dollars;
}

bool operator>= (const Dollars &d1, const Dollars &d2)
{
    return d1.m_dollars >= d2.m_dollars;
}

// Логической противоположностью оператора < является >=, поэтому мы можем
   просто инвертировать результат выполнения >=
bool operator< (const Dollars &d1, const Dollars &d2)
{
    return !(d1 >= d2);
}

// Логической противоположностью оператора <= является >, поэтому мы можем
   просто инвертировать результат выполнения >
bool operator<= (const Dollars &d1, const Dollars &d2)
{
    return !(d1 > d2);
}

int main()
{
    Dollars ten(10);
    Dollars seven(7);

    if (ten > seven)
        std::cout << "Ten dollars are greater than seven dollars.\n";
    if (ten >= seven)
        std::cout << "Ten dollars are greater than or equal to seven
   dollars.\n";
    if (ten < seven)
        std::cout << "Seven dollars are greater than ten dollars.\n";
    if (ten <= seven)
```

```
        std::cout << "Seven dollars are greater than or equal to ten
    dollars.\n";

    return 0;
}
```

Ответ №2

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

class Car
{
private:
    std::string m_company;
    std::string m_model;

public:
    Car(std::string company, std::string model)
        : m_company(company), m_model(model)
    {
    }

    friend bool operator== (const Car &c1, const Car &c2);
    friend bool operator!= (const Car &c1, const Car &c2);
    friend std::ostream& operator<< (std::ostream& out, const Car & c)
    {
        out << '(' << c.m_company << ", " << c.m_model << ')';
        return out;
    }

    friend bool operator<(const Car &c1, const Car &c2)
    {
        if (c1.m_company < c2.m_company)
            return true;
        if (c1.m_company > c2.m_company)
            return false;
        if (c1.m_model < c2.m_model)
            return true;
        if (c1.m_model > c2.m_model)
            return false;

        return false;
    }
};

bool operator== (const Car &c1, const Car &c2)
{
    return (c1.m_company == c2.m_company &&
        c1.m_model == c2.m_model);
}

bool operator!= (const Car &c1, const Car &c2)
{
    return !(c1 == c2);
}

int main()
{
```

```
    std::vector<Car> v;
    v.push_back(Car("Ford", "Mustang"));
    v.push_back(Car("Renault", "Logan"));
    v.push_back(Car("Ford", "Ranger"));
    v.push_back(Car("Renault", "Duster"));

    std::sort(v.begin(), v.end()); // требуется перегрузка оператора < для
   класса Car

    for (auto &car : v)
        std::cout << car << '\n'; // требуется перегрузка оператора << для
   класса Car

    return 0;
}
```

Ответы: Урок №146

Ответ №1.a)

```
#include <string>

struct StudentGrade
{
    std::string name;
    char grade;
};
```

Ответ №1.b)

```
#include <string>
#include <vector>

struct StudentGrade
{
    std::string name;
    char grade;
};

class GradeMap
{
private:
    std::vector<StudentGrade> m_map;

public:
    GradeMap()
    {
    }
};
```

Ответ №1.c)

```
#include <iostream>
#include <string>
#include <vector>

struct StudentGrade
{
    std::string name;
    char grade;
};

class GradeMap
{
private:
    std::vector<StudentGrade> m_map;

public:
    GradeMap()
    {
    }

    char& operator[](const std::string &name);
```

```
};

char& GradeMap::operator[](const std::string &name)
{
    // Смотрим, найдем ли мы имя ученика в векторе
    for (auto &ref : m_map)
    {
        // Если нашли, то возвращаем ссылку на его оценку
        if (ref.name == name)
            return ref.grade;
    }

    // В противном случае, создаем новый StudentGrade для нового ученика
    StudentGrade temp { name, ' ' };

    // Помещаем его в конец вектора
    m_map.push_back(temp);

    // И возвращаем ссылку на его оценку
    return m_map.back().grade;
}

int main()
{
    GradeMap grades;
    grades["John"] = 'A';
    grades["Martin"] = 'B';
    std::cout << "John has a grade of " << grades["John"] << '\n';
    std::cout << "Martin has a grade of " << grades["Martin"] << '\n';

    return 0;
}
```

Ответ №2

std::vector не является изначально отсортированным. Это означает, что каждый раз, при вызове operator[](), мы будем перебирать весь std::vector для поиска элемента. С несколькими элементами это не является проблемой, но, по мере того как их количество будет увеличиваться, процесс поиска элемента будет становиться все медленнее и медленнее. Мы могли бы это оптимизировать, сделав `m_map` отсортированным и используя бинарный поиск. Таким образом, количество элементов, которые будут использоваться при просмотре во время поиска одного элемента, уменьшится в разы.

Ответ №3

При добавлении `Martin`, std::vector должен увеличить свой размер. А для этого потребуется динамическое выделение нового блока памяти, копирование элементов массива в этот новый блок и удаление старого блока. Когда это произойдет, то любые ссылки на существующие элементы в std::vector пропадут! Другими словами, после того, как

выполнится `push_back("Martin")`, `gradeJohn` останется ссылкой на удаленную память. Это и приведет к неопределенным результатам.

Ответы: Урок №147

Ответ

```
#include <iostream>
#include <string>

class Mystring
{
private:
    std::string m_string;

public:
    Mystring(const std::string string="")
        :m_string(string)
    {
    }

    std::string operator()(int index1, int length)
    {
        std::string ret;
        for (int count = 0; count < length; ++count)
            ret += m_string[index1 + count];
        return ret;
    }
};

int main()
{
    Mystring string("Hello, world!");
    std::cout << string(7, 6); // начинаем с 7-го символа (индекса) и возвращаем
   следующие 6 символов

    return 0;
}
```

Ответы: Глава №9. Итоговый тест

Ответ №1

- Перегрузку бинарного оператора `+` лучше всего выполнять через обычную/дружественную функцию.
- Перегрузку унарного оператора `-` лучше всего выполнять через метод класса.
- Перегрузка оператора `<<` должна выполняться через обычную/дружественную функцию.
- Перегрузка оператора `=` должна выполняться через метод класса.

Ответ №2.а)

```
#include <iostream>
#include <cstdint> // для целочисленных значений фиксированного размера

class Average
{
private:
    int32_t m_total = 0; // сумма всех полученных значений
    int8_t m_numbers = 0; // количество всех полученных значений

public:
    Average()
    {
    }

    friend std::ostream& operator<<(std::ostream &out, const Average &average)
    {
        // Среднее значение = сумма всех полученных значений / количество всех
   полученных значений.
        // Следует помнить, что здесь должно выполняться деление типа с
   плавающей точкой (а не типа int)
        out << static_cast<double>(average.m_total) / average.m_numbers;

        return out;
    }

    // Поскольку operator+=() изменяет свой левый операнд, то перегрузку
   следует выполнять через метод класса
    Average& operator+=(int num)
    {
        // Увеличиваем сумму всех полученных значений новым значением
        m_total += num;
        // И добавляем единицу к общему количеству полученных чисел
        ++m_numbers;

        // Возвращаем текущий объект, чтобы иметь возможность выполнять цепочку
   операций с +=
        return *this;
    }
};
```

```
int main()
{
    Average avg;

    avg += 5;
    std::cout << avg << '\n'; // 5 / 1 = 5

    avg += 9;
    std::cout << avg << '\n'; // (5 + 9) / 2 = 7

    avg += 19;
    std::cout << avg << '\n'; // (5 + 9 + 19) / 3 = 11

    avg += -9;
    std::cout << avg << '\n'; // (5 + 9 + 19 - 9) / 4 = 6

    (avg += 7) += 11; // выполнение цепочки операций
    std::cout << avg << '\n'; // (5 + 9 + 19 - 9 + 7 + 11) / 6 = 7

    Average copy = avg;
    std::cout << copy << '\n';

    return 0;
}
```

Ответ №2.b)

Нет. Использование конструктора копирования и перегруженного оператора присваивания, предоставляемых компилятором по умолчанию, здесь будет достаточно.

Ответ №3

```
#include <iostream>
#include <cassert> // для стейтментов assert

class IntArray
{
private:
    int m_length = 0;
    int *m_array = nullptr;

public:
    IntArray(int length):
        m_length(length)
    {
        assert(length > 0 && "IntArray length should be a positive integer");

        m_array = new int[m_length] { 0 };
    }

    // Конструктор копирования, который выполняет глубокое копирование
    IntArray(const IntArray &array):
        m_length(array.m_length)
    {
        // Выделяем новый массив
        m_array = new int[m_length];
```

```
        // Копируем элементы из исходного массива в наш только что выделенный
    массив
        for (int count = 0; count < array.m_length; ++count)
            m_array[count] = array.m_array[count];
    }

    ~IntArray()
    {
        delete[] m_array;
    }

    // Функция перегрузки оператора <<
    friend std::ostream& operator<<(std::ostream &out, const IntArray &array)
    {
        for (int count = 0; count < array.m_length; ++count)
        {
            out << array.m_array[count] << ' ';
        }
        return out;
    }

    int& operator[] (const int index)
    {
        assert(index >= 0);
        assert(index < m_length);
        return m_array[index];
    }

    // Перегрузка оператора присваивания с выполнением глубокого копирования
    IntArray& operator= (const IntArray &array)
    {
        // Проверка на самоприсваивание
        if (this == &array)
            return *this;

        // Если массив уже существует, то удаляем его, дабы не произошла утечка
    памяти
        delete[] m_array;

        m_length = array.m_length;

        // Выделяем новый массив
        m_array = new int[m_length];

        // Копируем элементы из исходного массива в наш только что выделенный
    массив
        for (int count = 0; count < array.m_length; ++count)
            m_array[count] = array.m_array[count];

        return *this;
    }

};

IntArray fillArray()
{
    IntArray a(6);
    a[0] = 6;
    a[1] = 7;
    a[2] = 3;
    a[3] = 4;
    a[4] = 5;
```

```
        a[5] = 8;

    return a;
}

int main()
{
    IntArray a = fillArray();

    // Если у вас здесь получается какая-то чепуха, то, скорее всего, вы
    забыли выполнить глубокое копирование в вашем конструкторе копирования
    std::cout << a << '\n';

    IntArray b(1);
    a = a;
    b = a;

    // Если у вас здесь получается какая-то чепуха, то, скорее
    всего, вы забыли выполнить глубокое копирование в своей функции перегрузки
    оператора присваивания, либо забыли о проверке на самоприсваивание
    std::cout << b << '\n';

    return 0;
}
```

Ответ №4.a)

Существует несколько способов реализации значений типа с фиксированной точкой. Поскольку это тот же тип с плавающей точкой (кроме того, что количество цифр после точки является фиксированным), то использование типа float или типа double может показаться очевидным решением. Но значения типа с плавающей точкой имеют проблемы с точностью. С фиксированной точкой мы можем перебрать все возможные числа, которые могут находиться в дробной части значения (в нашем случае, от `.00` до `.99`), поэтому использование типа данных с ошибками в точности не является хорошим выбором.

Лучшее решение: Использовать тип int16_t signed для хранения целой части значения и int8_t signed для хранения дробной части значения.

Ответ №4.b)

```
#include <iostream>
#include <cstdint> // для целочисленных значений фиксированного размера

class FixedPoint
{
private:
    std::int16_t m_base; // это целая часть значения
    std::int8_t m_decimal; // это дробная часть значения

public:
    FixedPoint(std::int16_t base = 0, std::int8_t decimal = 0)
        : m_base(base), m_decimal(decimal)
    {
```

```
        // Здесь нам нужно обработать случай, когда дробная часть > 99 или < -
    99, но это вы должны будете сделать самостоятельно

        // Если целая или дробная части отрицательные
        if (m_base < 0.0 || m_decimal < 0.0)
        {
            // Проверяем целую часть
            if (m_base > 0.0)
                m_base = -m_base;
            // Проверяем дробную часть
            if (m_decimal > 0.0)
                m_decimal = -m_decimal;
        }
    }

    operator double() const
    {
        return m_base + static_cast<double>(m_decimal) / 100;
    }

    friend std::ostream& operator<<(std::ostream &out, const FixedPoint &fp)
    {
        out << static_cast<double>(fp);
        return out;
    }
};

int main()
{
    FixedPoint a(37, 58);
    std::cout << a << '\n';

    FixedPoint b(-3, 9);
    std::cout << b << '\n';

    FixedPoint c(4, -7);
    std::cout << c << '\n';

    FixedPoint d(-5, -7);
    std::cout << d << '\n';

    FixedPoint e(0, -3);
    std::cout << e << '\n';

    std::cout << static_cast<double>(e) << '\n';

    return 0;
}
```

Ответ №4.c)

```
#include <iostream>
#include <cstdint> // для целочисленных значений фиксированного размера
#include <cmath> // для функции round()

class FixedPoint
{
private:
    std::int16_t m_base; // это целая часть нашего значения
    std::int8_t m_decimal; // это дробная часть нашего значения
```

```
public:
    FixedPoint(std::int16_t base = 0, std::int8_t decimal = 0)
        : m_base(base), m_decimal(decimal)
    {
        // Здесь нам нужно обработать случай, когда дробная часть > 99 или < -
   99, но это вы должны будете сделать самостоятельно

        // Если целая или дробная части отрицательные
        if (m_base < 0.0 || m_decimal < 0.0)
        {
            // Проверяем целую часть
            if (m_base > 0.0)
                m_base = -m_base;
            // Проверяем дробную часть
            if (m_decimal > 0.0)
                m_decimal = -m_decimal;
        }
    }

    FixedPoint(double d)
    {
        // Сначала нам нужно получить целую часть значения.
        // Мы можем сделать это, конвертируя наше число типа double в число
   типа int
        m_base = static_cast<int16_t>(d); // отбрасывается дробная часть

        // Теперь нам нужно получить дробную часть нашего значения:
        // 1) d - m_base оставляет только дробную часть,
        // 2) которую затем мы можем умножить на 100, переместив две цифры из
   дробной части в целую часть значения
        // 3) теперь мы можем это дело округлить
        // 4) и, наконец, конвертировать в тип int, чтобы отбросить любую
   дополнительную дробь
        m_decimal = static_cast<std::int8_t>(round((d - m_base) * 100));
    }

    operator double() const
    {
        return m_base + static_cast<double>(m_decimal) / 100;
    }

    friend std::ostream& operator<<(std::ostream &out, const FixedPoint &fp)
    {
        out << static_cast<double>(fp);
        return out;
    }
};

int main()
{
    FixedPoint a(0.03);
    std::cout << a << '\n';

    FixedPoint b(-0.03);
    std::cout << b << '\n';

    FixedPoint c(4.01); // сохранится, как 4.0099999..., поэтому нам нужно это всё
    округлить
    std::cout << c << '\n';

    FixedPoint d(-4.01); // сохранится, как -4.0099999…, поэтому нам нужно это
   всё округлить
```

```
    std::cout << d << '\n';

    return 0;
}
```

Ответ №4.d)

```
#include <iostream>
#include <cstdint> // для целочисленных значений фиксированного размера
#include <cmath> // для функции round()

class FixedPoint
{
private:
    std::int16_t m_base; // это целая часть нашего значения
    std::int8_t m_decimal; // это дробная часть нашего значения

public:
    FixedPoint(std::int16_t base = 0, std::int8_t decimal = 0)
        : m_base(base), m_decimal(decimal)
    {
        // Здесь нам нужно обработать случай, когда дробная часть > 99 или < -
   99, но это вы должны будете сделать самостоятельно

        // Если дробная или целая части значения отрицательные
        if (m_base < 0.0 || m_decimal < 0.0)
        {
            // Проверяем целую часть
            if (m_base > 0.0)
                m_base = -m_base;
            // Проверяем дробную часть
            if (m_decimal > 0.0)
                m_decimal = -m_decimal;
        }
    }

    FixedPoint(double d)
    {
        // Сначала нам нужно получить целую часть значения.
        // Мы можем сделать это, конвертируя наше число типа double в число
   типа int
        m_base = static_cast<int16_t>(d); // отбрасывается дробная часть

        // Теперь нам нужно получить дробную часть нашего значения:
        // 1) d - m_base оставляет только дробную часть,
        // 2) которую затем мы можем умножить на 100, переместив две цифры из
   дробной части в целую часть значения
        // 3) теперь мы можем это дело округлить
        // 4) и, наконец, конвертировать в тип int, чтобы отбросить любую
   дополнительную дробь
        m_decimal = static_cast<std::int8_t>(round((d - m_base) * 100));
    }

    operator double() const
    {
        return m_base + static_cast<double>(m_decimal) / 100;
    }

    friend bool operator==(const FixedPoint &fp1, const FixedPoint &fp2)
    {
        return (fp1.m_base == fp2.m_base && fp1.m_decimal == fp2.m_decimal);
```

```
    }

    friend std::ostream& operator<<(std::ostream &out, const FixedPoint &fp)
    {
        out << static_cast<double>(fp);
        return out;
    }

    friend std::istream& operator >> (std::istream &in, FixedPoint &fp)
    {
        double d;
        in >> d;
        fp = FixedPoint(d);

        return in;
    }

    friend FixedPoint operator+(const FixedPoint &fp1, const FixedPoint &fp2)
    {
        return FixedPoint(static_cast<double>(fp1) + static_cast<double>(fp2));
    }

    FixedPoint operator-()
    {
        return FixedPoint(-m_base, -m_decimal);
    }

};

void SomeTest()
{
    std::cout << std::boolalpha;
    std::cout << (FixedPoint(0.75) + FixedPoint(1.23) == FixedPoint(1.98)) <<
   '\n'; // оба значения положительные, никакого переполнения
    std::cout << (FixedPoint(0.75) + FixedPoint(1.50) == FixedPoint(2.25)) <<
   '\n'; // оба значения положительные, переполнение
    std::cout << (FixedPoint(-0.75) + FixedPoint(-1.23) == FixedPoint(-
   1.98)) << '\n'; // оба значения отрицательные, никакого переполнения
    std::cout << (FixedPoint(-0.75) + FixedPoint(-1.50) == FixedPoint(-
   2.25)) << '\n'; // оба значения отрицательные, переполнение
    std::cout << (FixedPoint(0.75) + FixedPoint(-1.23) == FixedPoint(-
   0.48)) << '\n'; // второе значение отрицательное, никакого переполнения
    std::cout << (FixedPoint(0.75) + FixedPoint(-1.50) == FixedPoint(-
   0.75)) << '\n'; // второе значение отрицательное, возможно переполнение
    std::cout << (FixedPoint(-
   0.75) + FixedPoint(1.23) == FixedPoint(0.48)) << '\n'; // первое значение
   отрицательное, никакого переполнения
    std::cout << (FixedPoint(-
   0.75) + FixedPoint(1.50) == FixedPoint(0.75)) << '\n'; // первое значение
   отрицательное, возможно переполнение
}

int main()
{
    SomeTest();

    FixedPoint a(-0.48);
    std::cout << a << '\n';

    std::cout << -a << '\n';

    std::cout << "Enter a number: "; // введите 5.678
```

```
    std::cin >> a;

    std::cout << "You entered: " << a << '\n';

    return 0;
}
```

Ответы: Урок №156

Ответ №1

- Композиция: Цвет является неотъемлемым свойством шара.
- Агрегация: Работодатель в начале не имеет никаких работников и, надеемся, не уничтожит всех своих сотрудников, когда обанкротится.
- Композиция: Факультеты не могут существовать отдельно от университета.
- Композиция: Ваш возраст является неотъемлемым Вашим свойством.
- Агрегация: Мешок и шарики внутри являются независимыми объектами и могут существовать отдельно.

Ответ №2

```
#include <iostream>
#include <string>
#include <vector>

class Worker
{
private:
    std::string m_name;

public:
    Worker(std::string name)
        : m_name(name)
    {
    }

    std::string getName() { return m_name; }
};

class Department
{
private:
    std::vector<Worker*> m_worker;

public:
    Department()
    {
    }

    void add(Worker *worker)
    {
        m_worker.push_back(worker);
    }

    friend std::ostream& operator<<(std::ostream &out, const Department
   &department)
    {
        out << "Department: ";
        for (unsigned int count = 0; count < department.m_worker.size();
   ++count)
            out << department.m_worker[count]->getName() << ' ';
```

```
        out << '\n';

        return out;
    }
};


int main()
{
    // Создаем Работников вне области видимости класса Department
    Worker *w1 = new Worker("Anton");
    Worker *w2 = new Worker("Ivan");
    Worker *w3 = new Worker("Max");

    {
        // Создаем Отдел и передаем Работников в качестве параметров
   конструктора
        Department department; // создаем пустой Отдел
        department.add(w1);
        department.add(w2);
        department.add(w3);

        std::cout << department;

    } // department выходит из области видимости и уничтожается здесь

    std::cout << w1->getName() << " still exists!\n";
    std::cout << w2->getName() << " still exists!\n";
    std::cout << w3->getName() << " still exists!\n";

    delete w1;
    delete w2;
    delete w3;

    return 0;
}
```

Ответы: Урок №160

Ответ

```
#include <iostream>
#include <cassert> // для assert()
#include <initializer_list> // для std::initializer_list

class ArrayInt
{
private:
    int m_length;
    int *m_data;

public:
    ArrayInt() :
        m_length(0), m_data(nullptr)
    {
    }

    ArrayInt(int length) :
        m_length(length)
    {
        m_data = new int[length];
    }

    ArrayInt(const std::initializer_list<int> &list) : // позволяем инициализацию ArrayInt через список инициализации
        ArrayInt(list.size()) // используем концепцию делегирования конструкторов для создания начального массива, в который будет выполняться копирование элементов
    {
        // Инициализируем наш начальный массив значениями из списка
        int count = 0;
        for (auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }
    }

    ~ArrayInt()
    {
        delete[] m_data;
        // Нам не нужно здесь присваивать значение null для m_data или выполнять m_length = 0, так как объект будет уничтожен сразу же после выполнения этой функции
    }

    ArrayInt& operator=(const std::initializer_list<int> &list)
    {
        // Если новый список имеет другой размер, то перевыделяем его
        if (list.size() != static_cast<size_t>(m_length))
        {
            // Удаляем все существующие элементы
            delete[] m_data;

            // Перевыделяем массив
            m_length = list.size();
```

```
            m_data = new int[m_length];
        }

        // Теперь инициализируем наш массив значениями из списка
        int count = 0;
        for (auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }

        return *this;
    }

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() { return m_length; }
};

int main()
{
    ArrayInt array { 7, 6, 5, 4, 3, 2, 1 }; // список инициализации
    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    std::cout << '\n';

    array = { 1, 4, 9, 12, 15, 17, 19, 21 };

    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

Ответы: Глава №10. Итоговый тест

Ответ №1.a)

Композиция. Тип Животного и его Имя не используются вне класса Животное.

Ответ №1.b)

Зависимость. Класс TextEditor использует объект `file` для выполнения определенного задания — сохранения объекта на диск.

Ответ №1.c)

Агрегация. Когда Предметы связаны с Авантюристом, то Авантюрист «имеет» эти предметы. Меч, используемый Авантюристом, не может одновременно быть использован другим Авантюристом. Но Авантюрист не управляет существованием самих Предметов.

Ответ №1.d)

Ассоциация. Программист и Компьютер не связаны между собой, за исключением случаев, когда Программист использует Компьютер для просмотра видео с котами.

Ответ №1.e)

Агрегация. Компьютер имеет Процессор, но не управляет его существованием.

Ответ №2

Композиция.

Ответы: Урок №164

Ответ

```
#include <iostream>
#include <string>

class Fruit
{
private:
    std::string m_name;
    std::string m_color;

public:
    Fruit(std::string name, std::string color)
        : m_name(name), m_color(color)
    {
    }

    std::string getName() const { return m_name; }
    std::string getColor() const { return m_color; }

};

class Apple : public Fruit
{
private:
    double m_fiber;

public:
    Apple(std::string name, std::string color, double fiber)
        :Fruit(name, color), m_fiber(fiber)
    {
    }

    double getFiber() const { return m_fiber; }

    friend std::ostream& operator<<(std::ostream &out, const Apple &a)
    {
        out << "Apple (" << a.getName() << ", " << a.getColor() << ", " <<
   a.getFiber() << ")\n";
        return out;
    }
};

class Banana : public Fruit
{
public:
    Banana(std::string name, std::string color)
        :Fruit(name, color)
    {
    }

    friend std::ostream& operator<<(std::ostream &out, const Banana &b)
    {
        out << "Banana (" << b.getName() << ", " << b.getColor() << ")\n";
        return out;
    }

```

```
};

int main()
{
    const Apple a("Red delicious", "red", 7.3);
    std::cout << a;

    const Banana b("Cavendish", "yellow");
    std::cout << b;

    return 0;
}
```

Ответы: Глава №11. Итоговый тест

Ответ №1.a)

Сначала инициализируется родительская часть объекта, а затем уже дочерняя. Уничтожение происходит в обратном порядке.

```
Parent()
Child()
~Child()
~Parent()
```

Ответ №1.b)

Сначала выполняется построение `ch`:

```
Parent()
Child()
```

Затем построение `p`:

```
Parent()
```

Затем уничтожение `p`:

```
~Parent()
```

Затем уничтожение `ch`:

```
~Child()
~Parent()
```

Ответ №1.c)

Не скомпилируется. Метод Child::print() не имеет доступа к закрытому члену `m_x`.

Ответ №1.d)

Результат:

```
Parent()
Child()
Child: 7
```

```
~Child()
~Parent()
```

Ответ №1.e)

Результат:

```
Parent()
Child()
D2()
Child: 7
~D2()
~Child()
~Parent()
```

Ответ №2.a)

```
#include <iostream>
#include <string>

class Fruit
{
private:
    std::string m_name;
    std::string m_color;

public:
    Fruit(std::string name, std::string color)
        : m_name(name), m_color(color)
    {

    }

    std::string getName() { return m_name; }
    std::string getColor() { return m_color; }
};

class Apple: public Fruit
{
public:
    Apple(std::string color="red")
        : Fruit("apple", color)
    {
    }
};

class Banana : public Fruit
{
public:
    Banana()
        : Fruit("banana", "yellow")
    {

    }
};
```

```
int main()
{
    Apple a("red");
    Banana b;

    std::cout << "My " << a.getName() << " is " << a.getColor() << ".\n";
    std::cout << "My " << b.getName() << " is " << b.getColor() << ".\n";

    return 0;
}
```

Ответ №2.b)

```
#include <iostream>
#include <string>

class Fruit
{
private:
    std::string m_name;
    std::string m_color;

public:
    Fruit(std::string name, std::string color)
        : m_name(name), m_color(color)
    {

    }

    std::string getName() { return m_name; }
    std::string getColor() { return m_color; }
};

class Apple: public Fruit
{
// Предыдущий конструктор, который мы использовали для Apple, имел
   фиксированное имя ("apple").
// Нам нужен новый конструктор для GrannySmith, чтобы иметь возможность
   задавать имя для фрукта
protected:
    Apple(std::string name, std::string color)
        : Fruit(name, color)
    {
    }

public:
    Apple(std::string color="red")
        : Fruit("apple", color)
    {
    }
};

class Banana : public Fruit
{
public:
    Banana()
        : Fruit("banana", "yellow")
    {

    }
};
```

```
class GrannySmith : public Apple
{
public:
    GrannySmith()
        : Apple("Granny Smith apple", "green")
    {

    }
};

int main()
{
    Apple a("red");
    Banana b;
    GrannySmith c;

    std::cout << "My " << a.getName() << " is " << a.getColor() << ".\n";
    std::cout << "My " << b.getName() << " is " << b.getColor() << ".\n";
    std::cout << "My " << c.getName() << " is " << c.getColor() << ".\n";

    return 0;
}
```

Ответ №3.a)

```
#include <iostream>
#include <string>

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
    int m_health;
    int m_damage;
    int m_gold;

public:
    Creature(std::string name, char symbol, int health, int damage, int gold)
   : m_name(name), m_symbol(symbol), m_health(health), m_damage(damage),
   m_gold(gold)
    {
    }

    const std::string& getName() { return m_name; }
    char getSymbol() { return m_symbol; }
    int getHealth() { return m_health; }
    int getDamage() { return m_damage; }
    int getGold() { return m_gold; }

    void reduceHealth(int health) { m_health -= health; }
    bool isDead() { return m_health <= 0; }
    void addGold(int gold) { m_gold += gold; }
};

int main()
{
    Creature o("orc", 'o', 4, 2, 10);
    o.addGold(5);
    o.reduceHealth(1);
```

```
    std::cout << "The " << o.getName() << " has " << o.getHealth() <<
   " health and is carrying " << o.getGold() << " gold.";

    return 0;
}
```

Ответ №3.b)

```
#include <iostream>
#include <string>

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
    int m_health;
    int m_damage;
    int m_gold;

public:
    Creature(std::string name, char symbol, int health, int damage, int gold) :
        m_name(name), m_symbol(symbol), m_health(health), m_damage(damage),
   m_gold(gold)
    {
    }

    const std::string& getName() { return m_name; }
    char getSymbol() { return m_symbol; }
    int getHealth() { return m_health; }
    int getDamage() { return m_damage; }
    int getGold() { return m_gold; }

    void reduceHealth(int health) { m_health -= health; }
    bool isDead() { return m_health <= 0; }
    void addGold(int gold) { m_gold += gold; }
};

class Player : public Creature
{
    int m_level = 1;

public:
    Player(std::string name)
        : Creature(name, '@', 10, 1, 0)
    {
    }

    void levelUp()
    {
        ++m_level;
        ++m_damage;
    }

    int getLevel() { return m_level; }
    bool hasWon() { return m_level >= 20; }
};

int main()
{
    std::cout << "Enter your name: ";
```

```
    std::string playerName;
    std::cin >> playerName;

    Player p(playerName);
    std::cout << "Welcome, " << p.getName() << ".\n";

    std::cout << "You have " << p.getHealth() << " health and are carrying "
   << p.getGold() << " gold.";

    return 0;
}
```

Ответ №3.c)

```
class Monster : public Creature
{
public:
    enum Type
    {
        DRAGON,
        ORC,
        SLIME,
        MAX_TYPES
    };
};
```

Ответ №3.d)

```
#include <iostream>
#include <string>

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
    int m_health;
    int m_damage;
    int m_gold;

public:
    Creature(std::string name, char symbol, int health, int damage, int gold) :
        m_name(name), m_symbol(symbol), m_health(health), m_damage(damage),
   m_gold(gold)
    {
    }

    const std::string& getName() { return m_name; }
    char getSymbol() { return m_symbol; }
    int getHealth() { return m_health; }
    int getDamage() { return m_damage; }
    int getGold() { return m_gold; }

    void reduceHealth(int health) { m_health -= health; }
    bool isDead() { return m_health <= 0; }
    void addGold(int gold) { m_gold += gold; }
};

class Player : public Creature
{
```

```
    int m_level = 1;

public:
    Player(std::string name)
        : Creature(name, '@', 10, 1, 0)
    {
    }

    void levelUp()
    {
        ++m_level;
        ++m_damage;
    }

    int getLevel() { return m_level; }
};

class Monster : public Creature
{
public:
    enum Type
    {
        DRAGON,
        ORC,
        SLIME,
        MAX_TYPES
    };

    struct MonsterData
    {
        const char* name;
        char symbol;
        int health;
        int damage;
        int gold;
    };

    static MonsterData monsterData[MAX_TYPES];

    Monster(Type type)
        : Creature(monsterData[type].name, monsterData[type].symbol,
   monsterData[type].health, monsterData[type].damage, monsterData[type].gold)
    {
    }
};

Monster::MonsterData Monster::monsterData[Monster::MAX_TYPES]
{
    { "dragon", 'D', 20, 4, 100 },
    { "orc", 'o', 4, 2, 25 },
    { "slime", 's', 1, 1, 10 }
};

int main()
{
    Monster m(Monster::ORC);
    std::cout << "A " << m.getName() << " (" << m.getSymbol() << ") was
   created.\n";

    return 0;
}
```

Ответ №3.e)

```
#include <iostream>
#include <string>
#include <cstdlib> // для rand() и srand()
#include <ctime> // для time()

// Генерируем рандомное число между min и max
int getRandomNumber(int min, int max)
{
    static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);
    return static_cast<int>(rand() * fraction * (max - min + 1) + min);
}

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
    int m_health;
    int m_damage;
    int m_gold;

public:
    Creature(std::string name, char symbol, int health, int damage, int gold) :
        m_name(name), m_symbol(symbol), m_health(health), m_damage(damage),
   m_gold(gold)
    {
    }

    char getSymbol() { return m_symbol; }
    const std::string& getName() { return m_name; }
    bool isDead() { return m_health <= 0; }
    int getGold() { return m_gold; }
    void addGold(int gold) { m_gold += gold;  }
    void reduceHealth(int health) { m_health -= health;  }
    int getHealth() { return m_health; }
    int getDamage() { return m_damage;  }
};

class Player : public Creature
{
    int m_level = 1;

public:
    Player(std::string name)
        : Creature(name, '@', 10, 1, 0)
    {
    }

    void levelUp()
    {
        ++m_level;
        ++m_damage;
    }

    int getLevel() { return m_level; }
    bool hasWon() { return m_level >= 20; }
};

class Monster : public Creature
```

```
{
public:
    enum Type
    {
        DRAGON,
        ORC,
        SLIME,
        MAX_TYPES
    };

    struct MonsterData
    {
        const char* name;
        char symbol;
        int health;
        int damage;
        int gold;
    };

    static MonsterData monsterData[MAX_TYPES];

    Monster(Type type)
        : Creature(monsterData[type].name, monsterData[type].symbol,
   monsterData[type].health, monsterData[type].damage, monsterData[type].gold)
    {
    }

    static Monster getRandomMonster()
    {
        int num = getRandomNumber(0, MAX_TYPES - 1);
        return Monster(static_cast<Type>(num));
    }
};

Monster::MonsterData Monster::monsterData[Monster::MAX_TYPES]
{
    { "dragon", 'D', 20, 4, 100 },
    { "orc", 'o', 4, 2, 25 },
    { "slime", 's', 1, 1, 10 }
};

int main()
{
    srand(static_cast<unsigned int>(time(0))); // устанавливаем значение системных
    часов в качестве стартового числа
    rand(); // сбрасываем первый результат

    for (int i = 0; i < 10; ++i)
    {
        Monster m = Monster::getRandomMonster();
        std::cout << "A " << m.getName() << " (" << m.getSymbol() <<
   ") was created.\n";
    }

    return 0;
}
```

Ответ №3.f)

```
#include <iostream>
#include <string>
```

```
#include <cstdlib> // для rand() и srand()
#include <ctime> // для time()

// Генерируем рандомное число между min и max
int getRandomNumber(int min, int max)
{
    static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);
    return static_cast<int>(rand() * fraction * (max - min + 1) + min);
}

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
    int m_health;
    int m_damage;
    int m_gold;

public:
    Creature(std::string name, char symbol, int health, int damage, int gold) :
        m_name(name), m_symbol(symbol), m_health(health), m_damage(damage),
   m_gold(gold)
    {
    }

    char getSymbol() { return m_symbol; }
    const std::string& getName() { return m_name; }
    bool isDead() { return m_health <= 0; }
    int getGold() { return m_gold; }
    void addGold(int gold) { m_gold += gold;  }
    void reduceHealth(int health) { m_health -= health;  }
    int getHealth() { return m_health; }
    int getDamage() { return m_damage;  }
};

class Player : public Creature
{
    int m_level = 1;

public:
    Player(std::string name)
        : Creature(name, '@', 10, 1, 0)
    {
    }

    void levelUp()
    {
        ++m_level;
        ++m_damage;
    }

    int getLevel() { return m_level; }
    bool hasWon() { return m_level >= 20; }
};



class Monster : public Creature
{
public:
    enum Type
```

```
    {
        DRAGON,
        ORC,
        SLIME,
        MAX_TYPES
    };

    struct MonsterData
    {
        const char* name;
        char symbol;
        int health;
        int damage;
        int gold;
    };

    static MonsterData monsterData[MAX_TYPES];
    Type m_type;

    Monster(Type type)
        : Creature(monsterData[type].name, monsterData[type].symbol,
   monsterData[type].health, monsterData[type].damage, monsterData[type].gold),
   m_type(type)
    {
    }

    static Monster getRandomMonster()
    {
        int num = getRandomNumber(0, MAX_TYPES - 1);
        return Monster(static_cast<Type>(num));
    }
};

Monster::MonsterData Monster::monsterData[Monster::MAX_TYPES]
{
    { "dragon", 'D', 20, 4, 100 },
    { "orc", 'o', 6, 2, 25 },
    { "slime", 's', 1, 1, 10 }
};

// Этот метод обрабатывает атаку монстра игроком
void attackMonster(Player &p, Monster &m)
{
    // Если игрок мертв, то он не может атаковать монстра
    if (p.isDead())
        return;

    std::cout << "You hit the " << m.getName() << " for " <<
   p.getDamage() << " damage.\n";

    // Здоровье монстра уменьшается от урона игрока
    m.reduceHealth(p.getDamage());

    // Если монстр мертв, то увеличиваем level игрока
    if (m.isDead())
    {
        std::cout << "You killed the " << m.getName() << ".\n";
        p.levelUp();
        std::cout << "You are now level " << p.getLevel() << ".\n";
        std::cout << "You found " << m.getGold() << " gold.\n";
        p.addGold(m.getGold());
    }
```

```
}

// Этот метод обрабатывает атаку игрока монстром
void attackPlayer(Monster &m, Player &p)
{
    // Если монстр мертв, то он не может атаковать игрока
    if (m.isDead())
        return;

    // Здоровье игрока уменьшается от урона монстра
    p.reduceHealth(m.getDamage());
    std::cout << "The " << m.getName() << " hit you for " <<
   m.getDamage() << " damage.\n";
}

// Этот метод обрабатывает весь бой между игроком и рандомным монстром
void fightMonster(Player &p)
{
    // Сначала генерируем рандомного монстра
    Monster m = Monster::getRandomMonster();
    std::cout << "You have encountered a " << m.getName() << " (" <<
   m.getSymbol() << ").\n";

    // Пока монстр или игрок не мертв, то бой продолжается
    while (!m.isDead() && !p.isDead())
    {
        std::cout << "(R)un or (F)ight: ";
        char input;
        std::cin >> input;
        if (input == 'R' || input == 'r')
        {
            // 50/50 шанс удачного побега
            if (getRandomNumber(1, 2) == 1)
            {
                std::cout << "You successfully fled.\n";
                return; // встречу с монстром удалось избежать
            }
            else
            {
                // Неудачная попытка побега дает монстру право атаковать
                std::cout << "You failed to flee.\n";
                attackPlayer(m, p);
                continue;
            }
        }

        if (input == 'F' || input == 'f')
        {
            // Сначала атакует игрок, затем монстр
            attackMonster(p, m);
            attackPlayer(m, p);
        }
    }
}

int main()
{
    srand(static_cast<unsigned int>(time(0))); // устанавливаем значение системных
    часов в качестве стартового числа
    rand(); // сбрасываем первый результат

    std::cout << "Enter your name: ";
```

```
    std::string playerName;
    std::cin >> playerName;

    Player p(playerName);
    std::cout << "Welcome, " << p.getName() << '\n';

    // Если игрок не мертв и еще не победил, то игра продолжается
    while (!p.isDead() && !p.hasWon())
        fightMonster(p);

    // К этому моменту игрок либо мертв, либо победил
    if (p.isDead())
    {
        std::cout << "You died at level " << p.getLevel() << " and with "
   << p.getGold() << " gold.\n";
        std::cout << "Too bad you can't take it with you!\n";
    }
    else
    {
        std::cout << "You won the game with " << p.getGold() << " gold!\n";
    }

    return 0;
}
```

Ответы: Урок №170

Ответ

```
#include <iostream>
#include <string>

class Animal
{
protected:
    std::string m_name;
    const char* m_speak;

    // Мы делаем этот конструктор protected так как не хотим, чтобы
   пользователи могли создавать объекты класса Animal напрямую, но хотим, чтобы у
   дочерних классов доступ был открыт
    Animal(std::string name, const char* speak)
        : m_name(name), m_speak(speak)
    {
    }

public:
    std::string getName() { return m_name; }
    const char* speak() { return m_speak; }
};

class Cat: public Animal
{
public:
    Cat(std::string name)
        : Animal(name, "Meow")
    {
    }
};

class Dog: public Animal
{
public:
    Dog(std::string name)
        : Animal(name, "Woof")
    {
    }
};

int main()
{
    Cat matros("Matros"), ivan("Ivan"), martun("Martun");
    Dog barsik("Barsik"), tolik("Tolik"), tyzik("Tyzik");

    // Создаем массив указателей на наши объекты Cat и Dog
    Animal *animals[] = { &matros, &ivan, &martun, &barsik, &tolik, &tyzik};
    for (int iii=0; iii < 6; iii++)
        std::cout << animals[iii]->getName() << " says " << animals[iii]-
   >speak() << '\n';

    return 0;
}
```

> ***Примечание:*** Вы также можете сделать `m_speak` типа std::string, но недостатком будет то, что каждый объект класса Animal будет содержать лишнюю копию строки `speak`, а построение объектов Animal займет больше времени, так как глубокое копирование std::string будет выполняться медленнее, нежели копирование указателя, указывающего на константную строку C-style.

Ответы: Урок №171

Ответ a)

Результат:

```
B
```

`rParent` — это ссылка класса A на объект `c`. `rParent.getName()` вызывает A::getName(), но, поскольку A::getName() является виртуальной функцией, вызываться будет наиболее дочерний метод между классами A и C. А это B::getName(), так как в классе C метода getName() нет.

Ответ b)

Результат:

```
C
```

Всё довольно просто, C::getName() — это наиболее дочерний метод между классами B и C.

Ответ c)

Результат:

```
A
```

Поскольку getName() класса A не является виртуальным методом, то при обработке `rParent.getName()` вызовется A::getName().

Ответ d)

Результат:

```
C
```

Хотя B и C не являются виртуальными функциями, но A::getName() является виртуальной функцией, а B::getName() и C::getName() являются переопределениями. Следовательно, B::getName() и C::getName() считаются неявно виртуальными, и поэтому вызов rParent.getName() вызовет C::getName().

Ответ e)

Результат:

```
A
```

Это уже несколько сложнее. `rParent` — это ссылка класса A на объект `c`, поэтому `rParent.getName()` вызывает A::getName(). Но, поскольку A::getName() является виртуальной функцией, вызывается наиболее дочерний метод между A и C. И это A::getName(). Поскольку B::getName() и C::getName() не являются const, то они не считаются переопределениями!

Ответ f)

Результат:

```
A
```

Еще одно хитрое задание. При создании объекта `c`, сначала выполняется построение родительской части A. Для этого вызывается конструктор A, а он, в свою очередь, вызывает виртуальную функцию getName(). Поскольку части классов B и C еще не созданы, то выполняется A::getName().

Ответы: Урок №176

Ответ

Абстрактный класс может иметь переменные-члены и имеет как минимум одну чистую виртуальную функцию, в то время как интерфейс не имеет переменных-членов, и *все* его методы должны быть чистыми виртуальными функциями.

Ответы: Глава №12. Итоговый тест

Ответ №1.a)

Parent::getName() не является виртуальной функцией, поэтому `p.getName()` не вызовет Child::getName().

Ответ №1.b)

Хотя метод Child::getName() является константным, метод Parent::getName() не является константным, поэтому Child::getName() не считается переопределением и, следовательно, не вызывается.

Ответ №1.c)

Объект `ch` присваивается объекту `p` по значению (а не по ссылке), что приводит к обрезке объекта `ch`.

Ответ №1.d)

Класс Parent был объявлен как final, поэтому класс Child не может наследовать класс Parent. Результат — ошибка компиляции.

Ответ №1.e)

Child::getName() является абстрактной функцией, хотя имеет тело (записанное отдельно), поэтому класс Child является абстрактным, а объекты абстрактного класса создавать нельзя.

Ответ №1.f)

Эта программа выводит верный результат, но имеет другую проблему. В конце функции main() мы удаляем `p`, который является указателем класса Parent, но у нас нет виртуального деструктора в классе Parent. Следовательно, удаляется только часть Parent объекта класса `ch`, а часть Child объекта `ch` остается в виде утечки памяти.

Ответ №2.a)

```
class Shape
{
public:
    virtual std::ostream& print(std::ostream &out) const = 0;
```

```
    friend std::ostream& operator<<(std::ostream &out, const Shape &p)
    {
        return p.print(out);
    }
    virtual ~Shape() {}
};
```

Ответ №2.b)

```
#include <iostream>

class Point
{
private:
    int m_x = 0;
    int m_y = 0;
    int m_z = 0;

public:
    Point(int x, int y, int z)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    friend std::ostream& operator<<(std::ostream &out, const Point &p)
    {
        out << "Point(" << p.m_x << ", " << p.m_y << ", " << p.m_z << ")";
        return out;
    }
};

class Shape
{
public:
    virtual std::ostream& print(std::ostream &out) const = 0;

    friend std::ostream& operator<<(std::ostream &out, const Shape &p)
    {
        return p.print(out);
    }
    virtual ~Shape() {}
};

class Triangle : public Shape
{
private:
    Point m_p1;
    Point m_p2;
    Point m_p3;

public:
    Triangle(const Point &p1, const Point &p2, const Point &p3)
        : m_p1(p1), m_p2(p2), m_p3(p3)
    {
    }

    virtual std::ostream& print(std::ostream &out) const override
    {
        out << "Triangle(" << m_p1 << ", " << m_p2 << ", " << m_p3 << ")";
```

```
        return out;
    }
};

class Circle: public Shape
{
private:
    Point m_center;
    int m_radius;

public:
    Circle(const Point ¢er, int radius)
        : m_center(center), m_radius(radius)
    {
    }

    virtual std::ostream& print(std::ostream &out) const override
    {
        out << "Circle(" << m_center << ", radius " << m_radius << ")";
        return out;
    }
};

int main()
{
    Circle c(Point(1, 2, 3), 7);
    std::cout << c << '\n';

    Triangle t(Point(1, 2, 3), Point(4, 5, 6), Point(7, 8, 9));
    std::cout << t << '\n';

    return 0;
}
```

Ответ №2.c)

```
#include <iostream>
#include <vector>

class Point
{
private:
    int m_x = 0;
    int m_y = 0;
    int m_z = 0;

public:
    Point(int x, int y, int z)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    friend std::ostream& operator<<(std::ostream &out, const Point &p)
    {
        out << "Point(" << p.m_x << ", " << p.m_y << ", " << p.m_z << ")";
        return out;
    }
};

class Shape
```

```
{
public:
    virtual std::ostream& print(std::ostream &out) const = 0;

    friend std::ostream& operator<<(std::ostream &out, const Shape &p)
    {
        return p.print(out);
    }
    virtual ~Shape() {}
};

class Triangle : public Shape
{
private:
    Point m_p1;
    Point m_p2;
    Point m_p3;

public:
    Triangle(const Point &p1, const Point &p2, const Point &p3)
        : m_p1(p1), m_p2(p2), m_p3(p3)
    {
    }

    virtual std::ostream& print(std::ostream &out) const override
    {
        out << "Triangle(" << m_p1 << ", " << m_p2 << ", " << m_p3 << ")";
        return out;
    }
};


class Circle: public Shape
{
private:
    Point m_center;
    int m_radius;

public:
    Circle(const Point ¢er, int radius)
        : m_center(center), m_radius(radius)
    {
    }

    virtual std::ostream& print(std::ostream &out) const override
    {
        out << "Circle(" << m_center << ", radius " << m_radius << ")";
        return out;
    }

    int getRadius() { return m_radius;  }
};

int getLargestRadius(const std::vector<Shape*> &v)
{
    int largestRadius { 0 };

    // Перебираем каждый элемент вектора
    for (auto const &element : v)
    {
        // Выполняем проверку на нулевой указатель результата динамического
   приведения
```

```
            if (Circle *c = dynamic_cast<Circle*>(element))
        {
            if (c->getRadius() > largestRadius)
                largestRadius = c->getRadius();
        }
    }

    return largestRadius;
}
int main()
{
    std::vector<Shape*> v;
    v.push_back(new Circle(Point(1, 2, 3), 7));
    v.push_back(new Triangle(Point(1, 2, 3), Point(4, 5, 6), Point(7, 8, 9)));
    v.push_back(new Circle(Point(4, 5, 6), 3));

    for (auto const &element : v)
        std::cout << *element << '\n';

    std::cout << "The largest radius is: " << getLargestRadius(v) << '\n';

    for (auto const &element : v)
        delete element;

    return 0;
}
```

Ответы: Глава №13. Итоговый тест

Ответ №1

```
#include <iostream>

template <class T>
class Pair1
{
private:
    T m_a;
    T m_b;

public:
    Pair1(const T& a, const T& b)
        : m_a(a), m_b(b)
    {
    }

    T& first() { return m_a;  }
    const T& first() const { return m_a; }
    T& second() { return m_b; }
    const T& second() const { return m_b; }
};

int main()
{
    Pair1<int> p1(6, 9);
    std::cout << "Pair: " << p1.first() << ' ' << p1.second() << '\n';

    const Pair1<double> p2(3.4, 7.8);
    std::cout << "Pair: " << p2.first() << ' ' << p2.second() << '\n';

    return 0;
}
```

Ответ №2

```
#include <iostream>

template <class T, class S>
class Pair
{
private:
    T m_a;
    S m_b;

public:
    Pair(const T& a, const S& b)
        : m_a(a), m_b(b)
    {
    }

    T& first() { return m_a; }
    const T& first() const { return m_a; }
    S& second() { return m_b; }
    const S& second() const { return m_b; }
};
```

```
int main()
{
    Pair<int, double> p1(6, 7.8);
    std::cout << "Pair: " << p1.first() << ' ' << p1.second() << '\n';

    const Pair<double, int> p2(3.4, 5);
    std::cout << "Pair: " << p2.first() << ' ' << p2.second() << '\n';

    return 0;
}
```

Ответ №3

```
#include <iostream>
#include <string>

template <class T, class S>
class Pair
{
private:
    T m_a;
    S m_b;

public:
    Pair(const T& a, const S& b)
        : m_a(a), m_b(b)
    {
    }

    T& first() { return m_a; }
    const T& first() const { return m_a; }
    S& second() { return m_b; }
    const S& second() const { return m_b; }
};

template <class S>
class StringValuePair : public Pair<std::string, S>
{
public:
    StringValuePair(const std::string& key, const S& value)
        : Pair<std::string, S>(key, value)
    {
    }
};

int main()
{
    StringValuePair<int> svp("Amazing", 7);
    std::cout << "Pair: " << svp.first() << ' ' << svp.second() << '\n';

    return 0;
}
```

Ответы: Глава №14. Итоговый тест

Ответ

```
#include <iostream>
#include <stdexcept> // для std::runtime_error

class Fraction
{
private:
    int m_numerator = 0;
    int m_denominator = 1;

public:
    Fraction(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        if (m_denominator == 0)
            throw std::runtime_error("Invalid denominator");
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

int main()
{
    std::cout << "Enter the numerator: ";
    int numerator;
    std::cin >> numerator;

    std::cout << "Enter the denominator: ";
    int denominator;
    std::cin >> denominator;

    try
    {
        Fraction f(numerator, denominator);
        std::cout << "Your fraction is: " << f << '\n';
    }
    catch (std::exception&)
    {
        std::cout << "Your fraction has an invalid denominator.\n";
    }

    return 0;
}
```

Ответы: Урок №198

Ответ

`B`, `E` и `G` не скомпилируются.

Ответы: Урок №201

Ответ №1

Умные указатели в качестве членов класса удаляют свой ресурс только в том случае, если объект класса выходит из области видимости. Если вы выделите объект класса динамически и не удалите его должным образом, то объект класса никогда не выйдет из области видимости, и умный указатель не сможет очистить ресурс, который он хранит.

Ответ №2

```
#include <iostream>
#include <memory> // для std::unique_ptr

class Fraction
{
private:
    int m_numerator = 0;
    int m_denominator = 1;

public:
    Fraction(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
    {
        out << f1.m_numerator << "/" << f1.m_denominator;
        return out;
    }
};

// Эта функция использует объект класса Fraction, поэтому мы только его
   передаем.
// Таким образом мы можем не беспокоиться о том, какой умный указатель
   использует caller (если вообще использует)
void printFraction(const Fraction* const ptr)
{
    if (ptr)
        std::cout << *ptr;
}

int main()
{
    auto ptr = std::make_unique<Fraction>(7, 9);

    printFraction(ptr.get());

    return 0;
}
```

Ответы: Урок №203

Ответ

```
#include <iostream>
#include <memory> // для std::shared_ptr и std::weak_ptr

class Item
{
public:
    std::weak_ptr<Item> m_ptr; // используем std::weak_ptr, чтобы m_ptr не поддерживал Item-а

    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
};

int main()
{
    auto ptr1 = std::make_shared<Item>();

    ptr1->m_ptr = ptr1; // m_ptr теперь является владельцем Item-а, членом которого он является сам

    return 0;
}
```

Ответы: Глава №15. Итоговый тест

Ответ №1.a)

Умный указатель std::unique_ptr следует использовать, когда нужно, чтобы умный указатель единолично владел динамически выделенным ресурсом.

Ответ №1.b)

Умный указатель std::shared_ptr следует использовать, когда нужно, чтобы сразу несколько умных указателей владело одним динамически выделенным ресурсом. Ресурс не будет уничтожен до тех пор, пока не будут уничтожены все std::shared_ptr, владеющие им.

Ответ №1.c)

Умный указатель std::weak_ptr следует использовать, когда нужно иметь доступ к ресурсу, которым управляет другой умный указатель std::shared_ptr, но продолжительности жизни этих двух умных указателей не должны быть связаны между собой.

Ответ №1.d)

Умный указатель std::auto_ptr устарел, и его использования следует избегать.

Ответ №2

Поскольку значения r-values являются временными, то мы знаем, что они будут уничтожены сразу же после их использования. При передаче или возврате r-value по значению менее эффективным будет выполнять копирование, а затем уничтожать оригинал. Вместо этого гораздо эффективнее будет просто переместить (передать) ресурсы r-value в нужный объект.

Ответ №3.a)

`ptr2` был создан из `item`-а, а не из `ptr1`. Это означает, что теперь у нас есть два отдельных умных указателя std::shared_ptr, каждый из которых пытается независимо управлять Item-ом (они не знают о существовании друг друга). Когда один из этих умных указателей выйдет из области видимости, то другой останется висячим указателем.

`ptr2` должен быть скопирован из `ptr1`, и для создания std::shared_ptr следует использовать функцию std::make_shared():

```
#include <iostream>
#include <memory> // для std::shared_ptr

class Item
{
public:
    Item() { std::cout << "Item acquired\n"; }
    ~Item() { std::cout << "Item destroyed\n"; }
};

int main()
{
    auto ptr1 = std::make_shared<Item>();
    auto ptr2(ptr1);

    return 0;
}
```

Ответ №3.b)

Если конструктор класса Something выбросит исключение, то один из объектов класса Something, вполне вероятно, не будет уничтожен должным образом.

Решение заключается в использовании функции std::make_shared():

```
#include <iostream>
#include <memory> // для std::shared_ptr

class Something; // предположим, что Something - это класс, который может
   выбросить исключение

int main()
{
    doSomething(std::make_shared<Something>(), std::make_shared<Something>());

    return 0;
}
```