**Глава 1: Введение в C++ и его история**

**1.1. Что такое C++?**

Представь себе, что C++ — это как конструктор LEGO, но для программ. Он позволяет создавать из простых "кубиков" (инструкций) сложные и мощные приложения. Но, в отличие от LEGO, C++ дает тебе полный контроль над "деталями", что позволяет создавать очень эффективные программы.

В более формальном определении, C++ - это *компилируемый, статически типизированный язык программирования общего назначения*. Давай разберем это определение по частям:

* **Компилируемый:** Это значит, что написанный тобой код на C++ (называемый "исходным кодом") сначала преобразуется специальной программой – компилятором – в машинный код (последовательность инструкций, которые понимает компьютер), а затем этот машинный код выполняется компьютером. Это делает программы на C++ очень быстрыми.
  + **Аналогия:** Представь, что ты пишешь письмо на русском, а компилятор — это переводчик, который переводит это письмо на английский, чтобы его мог прочитать англоговорящий получатель.
* **Статически типизированный:** Это значит, что ты должен явно указать тип данных для каждой переменной (например, число, символ, строка) при ее объявлении. Компилятор проверяет типы данных во время компиляции, чтобы убедиться, что все операции выполняются правильно.
  + **Аналогия:** Представь, что ты сортируешь разные типы игрушек в отдельные коробки. Статическая типизация заставляет тебя четко указать, что эта коробка для машинок, а эта для кукол, чтобы не было путаницы.
* **Язык программирования общего назначения:** Это значит, что C++ можно использовать для создания широкого спектра программ: от игр и операционных систем до научных симуляций и встраиваемых систем. Это как универсальный инструмент, который может пригодиться для многих задач.

**1.2. Краткая история C++**

C++ не возник из ниоткуда. Он имеет свою историю, которая тесно связана с языком C.

* **1970-е: Рождение языка C**
  + Язык C был разработан Деннисом Ритчи в Bell Labs для создания операционной системы Unix.
  + C стал очень популярным благодаря своей простоте, гибкости и высокой производительности.
  + Он позволял программистам иметь прямой доступ к памяти, что было очень важно в то время.
* **1979: Bjarne Stroustrup и "C with Classes"**
  + Бьярн Страуструп, работая в Bell Labs, начал разрабатывать расширение языка C, которое добавило поддержку объектно-ориентированного программирования (ООП).
  + Он назвал свой язык "C with Classes" (С с классами). Классы позволяют создавать "объекты", которые могут иметь свои свойства и функции, что упрощает разработку сложных программ.
* **1983: C++**
  + Язык "C with Classes" был переименован в C++. Оператор ++ в C обозначает инкремент (увеличение на 1). Название C++ намекает на то, что это "улучшенная" версия языка C.
  + В C++ были добавлены новые функции, такие как виртуальные функции, перегрузка операторов, шаблоны и обработка исключений.
  + C++ сохранил обратную совместимость с C, это означало, что можно было использовать старый код на C в C++ программе.
* **1998: Стандартизация C++ (C++98)**
  + Был выпущен первый международный стандарт языка C++ (ISO/IEC 14882:1998), который определил основные правила языка и его стандартную библиотеку.
  + Это гарантировало, что программы на C++ будут работать одинаково на разных платформах.
* **2003, 2011, 2014, 2017, 2020, 2023: Новые стандарты**
  + Позже были выпущены новые стандарты языка (C++03, C++11, C++14, C++17, C++20, C++23), которые добавляли новые возможности и улучшали существующие.
  + Каждый новый стандарт приносил улучшения в производительности, безопасности и удобстве языка.
  + Стандарт C++20, например, добавил такие мощные функции, как модули и корутины.

**1.3. Почему C++ по-прежнему популярен?**

Несмотря на появление новых языков, C++ остается одним из самых популярных языков программирования по нескольким причинам:

* **Производительность:** Он идеально подходит для задач, где важна максимальная скорость и минимальное потребление ресурсов (например, игры, операционные системы, высокопроизводительные вычисления).
* **Контроль:** Он дает разработчику максимальный контроль над памятью и ресурсами. Это позволяет создавать высокооптимизированный код.
* **Универсальность:** Он подходит для самых разных проектов.
* **Большое сообщество:** C++ имеет огромное сообщество разработчиков, что означает, что есть много готовых библиотек, инструментов и ресурсов для обучения.
* **Наследие:** Многие существующие проекты и технологии используют C++, поэтому знание языка является ценным навыком.

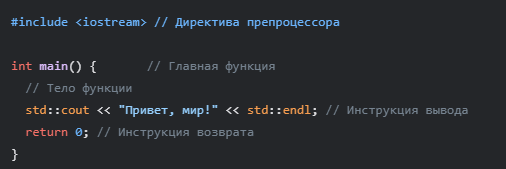
**Глава 2: Структура программы на C++**

В этой главе мы изучим, из чего состоит программа на C++, как она строится и как ее правильно организовывать. Мы рассмотрим следующие ключевые понятия:

* Директивы препроцессора
* Функция main()
* Объявление переменных
* Инструкции и операторы
* Комментарии

**2.1. Базовая структура программы**

Самая простая программа на C++ выглядит примерно так:



Давай разберем эту программу построчно:

* **#include <iostream>**: Это директива препроцессора.
  + **Препроцессор** — это специальная программа, которая обрабатывает исходный код перед тем, как его компилирует компилятор.
  + #include говорит препроцессору вставить содержимое файла iostream в текущий файл.
  + Файл iostream (input-output stream) содержит объявления для ввода и вывода данных, такие как std::cout и std::endl. Мы поговорим о потоках ввода-вывода подробнее в будущем.
* **int main() { ... }**: Это объявление главной функции программы.
  + **Функция** — это блок кода, который выполняет определенную задачу.
  + main — это специальная функция, с которой начинается выполнение любой программы на C++.
  + int — это тип возвращаемого значения функции main. В данном случае это целое число, которое обычно используется для обозначения статуса завершения программы (0 - успешное завершение).
  + { ... } — это тело функции, где помещается весь код, который нужно выполнить.
* **std::cout << "Привет, мир!" << std::endl;**: Это инструкция вывода на экран.
  + std::cout (character output) — это объект, который представляет стандартный поток вывода (обычно консоль).
  + << — это оператор вывода, который помещает данные в поток вывода.
  + "Привет, мир!" — это строковая константа, которую мы хотим вывести на экран.
  + std::endl — это манипулятор, который добавляет перевод строки в поток вывода (переводит курсор на следующую строку).
* **return 0;**: Это инструкция возврата значения из функции.
  + return — это оператор возврата, который возвращает значение из функции.
  + 0 — это целое число, которое означает, что программа завершилась успешно.

**2.2. Директивы препроцессора**

Как мы уже видели, директивы препроцессора начинаются с символа #. Они используются для управления процессом компиляции. Наиболее распространенные директивы:

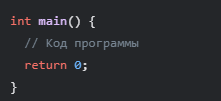
* #include: Включает содержимое указанного файла в текущий исходный файл. Используется для подключения библиотек и других файлов.
* #define: Определяет макрос, который представляет собой замену текста.



* #ifdef, #ifndef, #else, #endif: Используются для условной компиляции, когда некоторые части кода компилируются в зависимости от определенного условия.

**2.3. Функция main()**

Функция main() — это сердце любой программы на C++. Она имеет следующую структуру:

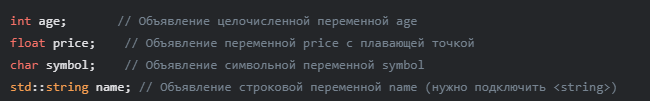


* int указывает на то, что функция возвращает целое число (обычно 0 при успешном завершении).
* main — это имя функции.
* () — это список параметров (в данном случае список пуст).
* { ... } — это тело функции, которое содержит код, который нужно выполнить.

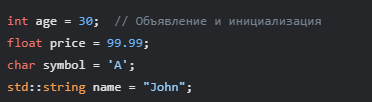
Функция main() может также принимать аргументы командной строки, но об этом мы поговорим позже.

**2.4. Объявление переменных**

Переменные используются для хранения данных. В C++ ты должен объявить переменную, прежде чем ее использовать. Объявление переменной включает в себя указание ее типа и имени.



Ты также можешь инициализировать переменную при объявлении:



**2.5. Инструкции и операторы**

Инструкция — это команда, которая заставляет программу выполнить какое-либо действие. Обычно инструкция заканчивается точкой с запятой ;.

Примеры инструкций:

* Объявление переменной: int age;
* Присваивание значения: age = 30;
* Вывод на экран: std::cout << "Привет!";
* Вызов функции: return 0;

Операторы — это символы, которые выполняют определенные действия над данными (операндами).

Примеры операторов:

* Арифметические операторы: +, -, \*, /, %
* Оператор присваивания: =
* Операторы сравнения: ==, !=, >, <, >=, <=
* Операторы ввода-вывода: <<, >>

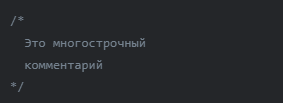
**2.6. Комментарии**

Комментарии — это текст в коде, который игнорируется компилятором. Они используются для пояснения кода и делают его более понятным. В C++ есть два типа комментариев:

* Однострочные комментарии начинаются с //:



* Многострочные комментарии начинаются с /\* и заканчиваются \*/:



**2.7. Пример полной программы**

Вот пример программы, которая демонстрирует большинство элементов, которые мы рассмотрели:



**2.8. Закрепление материала**

1. Что такое директивы препроцессора и зачем они нужны?
2. Какова роль функции main() в программе?
3. Как объявляются переменные в C++?
4. Что такое инструкция и оператор?
5. Для чего нужны комментарии?

Попробуй самостоятельно написать простую программу, которая:

1. Запрашивает имя пользователя.
2. Запрашивает возраст пользователя.
3. Выводит на экран сообщение с приветствием и возрастом пользователя.

**Глава 3: Переменные в C++**

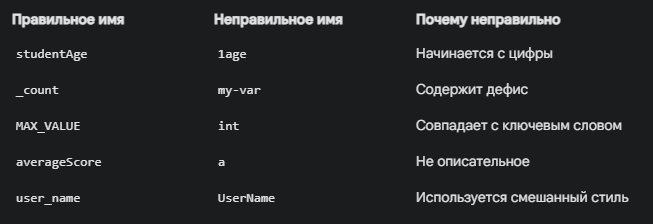
Переменная — это именованная область памяти, в которой можно хранить данные. Представь себе переменную как контейнер, в котором можно положить значение определенного типа (число, символ, строка и т.д.). В C++, как и в других языках программирования, переменные играют ключевую роль в хранении и обработке информации.

**3.1. Именование переменных**

Имена переменных (идентификаторы) в C++ должны соответствовать определенным правилам. Правила хорошего стиля и читаемого кода также играют важную роль.

* **Правила именования:**
  + Имя переменной может состоять из букв (a-z, A-Z), цифр (0-9) и символа подчеркивания \_.
  + Имя переменной должно начинаться с буквы или символа подчеркивания. Цифра в начале недопустима.
  + C++ — регистрозависимый язык, то есть age, Age и AGE — это три разные переменные.
  + Имя переменной не может совпадать с ключевыми словами C++ (например, int, float, return, if и т.д.).
* **Хороший стиль именования:**
  + Имена переменных должны быть *описательными*. Это означает, что имя должно давать представление о том, какая информация хранится в переменной. Например, studentAge лучше, чем sa.
  + Используй *camelCase* или *snake\_case* для составных имен:
    - **camelCase**: firstName, studentGrade, calculateArea
    - **snake\_case**: first\_name, student\_grade, calculate\_area
    - Выбери один стиль и придерживайся его в своем коде.
  + Имена констант (значений, которые не изменяются) обычно пишут заглавными буквами с подчеркиваниями: MAX\_VALUE, PI.
  + Избегай слишком коротких имен (кроме счетчиков в циклах, например, i, j) или слишком длинных имен.
  + Избегай использования имен, которые могут быть легко перепутаны, например, l (маленькая L) и 1 (единица).

**Примеры правильных и неправильных имен переменных:**

****

**3.2. Определение переменной**

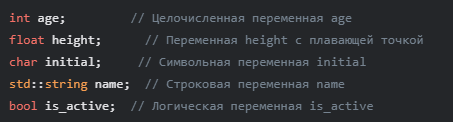
Определение переменной — это процесс, в результате которого выделяется память для хранения данных определенного типа и переменной присваивается имя. Определение переменной включает в себя:

* **Указание типа данных:** Тип данных определяет, какие значения можно хранить в переменной (целые числа, числа с плавающей точкой, символы, строки и т.д.).
* **Имя переменной:** Имя, которое будет использоваться для доступа к переменной.

Синтаксис определения переменной:



Примеры определения переменных:



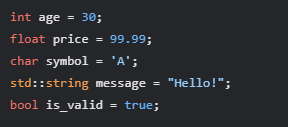
В C++, **каждая переменная должна быть определена до ее использования**. Попытка использования неопределенной переменной приведет к ошибке компиляции.

**3.3. Инициализация переменных**

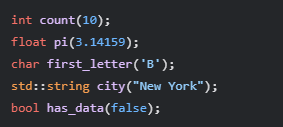
Инициализация переменной — это процесс присваивания ей начального значения при объявлении. Это очень важный шаг, так как использование неинициализированной переменной может привести к непредсказуемому поведению программы.

В C++ есть несколько способов инициализации переменных:

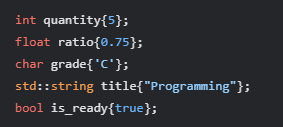
1. **Инициализация с использованием оператора =:**



1. **Инициализация в скобках:**



1. **Инициализация с использованием фигурных скобок (списковая инициализация):**



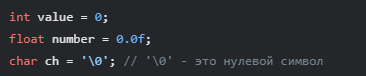
Фигурные скобки имеют дополнительное преимущество: они предотвращают неявные преобразования типов (сужение). Например, если ты попытаешься инициализировать int переменной типа float с использованием фигурных скобок, компилятор выдаст ошибку. Это помогает избежать потенциальных проблем, связанных с потерей данных.



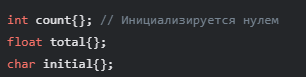
**3.4. Инициализация нулем**

Иногда полезно инициализировать переменные нулевым значением при объявлении. Это можно сделать следующими способами:

1. **Явная инициализация нулем:**



1. **Инициализация в фигурных скобках:**



Если ты используешь фигурные скобки без начального значения, переменная будет инициализирована нулем (или значением по умолчанию для соответствующего типа).

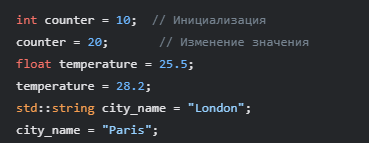
**3.5. Инициализация по умолчанию**

Если переменная не инициализируется явно, то ее значение будет *неопределенным*. Это значит, что переменная может содержать случайное значение из области памяти, где она была выделена.

**Важно:** Всегда явно инициализируй переменные при объявлении, чтобы избежать непредсказуемого поведения программы.

**3.6. Изменение значения переменной**

После инициализации значение переменной можно изменять с помощью оператора присваивания =:



Оператор присваивания копирует значение справа от него в переменную, находящуюся слева.

**3.7. Закрепление материала**

1. Перечисли правила именования переменных в C++.
2. Почему важно выбирать описательные имена для переменных?
3. Что такое определение переменной?
4. Почему важно инициализировать переменные?
5. Какие способы инициализации переменных ты знаешь?
6. В чем отличие инициализации с помощью фигурных скобок от инициализации с помощью оператора =?
7. Что происходит, если не инициализировать переменную?
8. Как изменить значение переменной после ее инициализации?

**Практическое задание:**

1. Создай несколько переменных разных типов (int, float, char, string, bool).
2. Присвой им начальные значения различными способами (с помощью =, в скобках, в фигурных скобках).
3. Измени значения этих переменных.
4. Выведи значения переменных на экран, чтобы проверить, что они действительно изменились.
5. Попробуй использовать разные стили именования переменных (camelCase, snake\_case) и выбери тот, который тебе больше нравится.

**Глава 4: Константы в C++**

Константа – это именованная область памяти, которая содержит значение, которое не может быть изменено во время выполнения программы. Константы играют важную роль в обеспечении безопасности, надежности и читаемости кода.

**4.1. Зачем нужны константы?**

Константы используются для:

* **Хранения неизменяемых значений:** Например, математические константы (число π), физические константы (скорость света), значения конфигурации (максимальный размер массива) и т.д.
* **Улучшения читаемости кода:** Использование констант с описательными именами делает код более понятным и легким для восприятия.
* **Предотвращения ошибок:** Компилятор гарантирует, что значение константы не будет случайно изменено. Это уменьшает вероятность ошибок в программе.
* **Упрощения сопровождения кода:** Если необходимо изменить значение, которое используется в нескольких местах программы, то достаточно изменить значение константы в одном месте, а не искать и менять его по всему коду.

**4.2. Способы объявления констант**

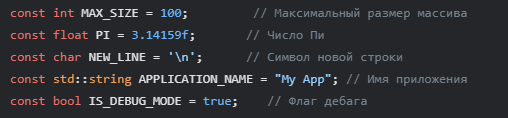
В C++ есть два основных способа объявления констант:

1. **С использованием ключевого слова const:**  
   Этот способ является наиболее распространенным и рекомендуется для большинства случаев.



* const — ключевое слово, которое указывает, что это константа.
* тип\_данных — тип данных константы (например, int, float, char).
* имя\_константы — имя константы, которое должно соответствовать правилам именования переменных (но обычно используется UPPER\_SNAKE\_CASE).
* значение — начальное значение константы.

**Примеры:**

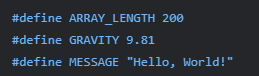


1. **С использованием директивы препроцессора #define:**  
   Этот способ, унаследованный от C, является менее предпочтительным в C++ и лучше избегать его использования, если есть возможность использовать const.



* #define — директива препроцессора, которая определяет макрос.
* имя\_константы — имя константы (обычно пишется заглавными буквами).
* значение — значение константы.

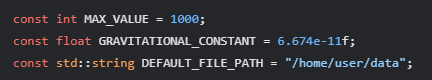
**Примеры:**



1. **Почему #define не рекомендуется?**
   * #define работает на уровне препроцессора, а не на уровне компилятора, поэтому у него нет типовой безопасности.
   * Ошибки, связанные с #define, могут быть сложнее в отладке.
   * Константы, объявленные с помощью const, могут иметь тип данных, что позволяет компилятору проверять правильность их использования.

**4.3. Правила именования констант**

По соглашению, имена констант обычно пишутся заглавными буквами, а слова разделяются подчеркиванием (UPPER\_SNAKE\_CASE). Это делает их легко узнаваемыми в коде.



**4.4. Инициализация констант**

Константы должны быть инициализированы при объявлении. После инициализации значение константы не может быть изменено. Попытка изменить значение константы приведет к ошибке компиляции.

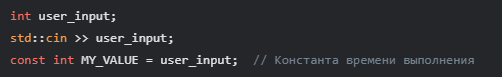


**4.5. Константы времени компиляции и времени выполнения**

* **Константы времени компиляции:** Значения констант const, объявленных с использованием литералов, вычисляются во время компиляции. Это означает, что их значения известны на этапе компиляции.



* **Константы времени выполнения:** Значение константы const может быть получено во время выполнения программы, например, из аргументов командной строки, ввода пользователя или из других переменных.



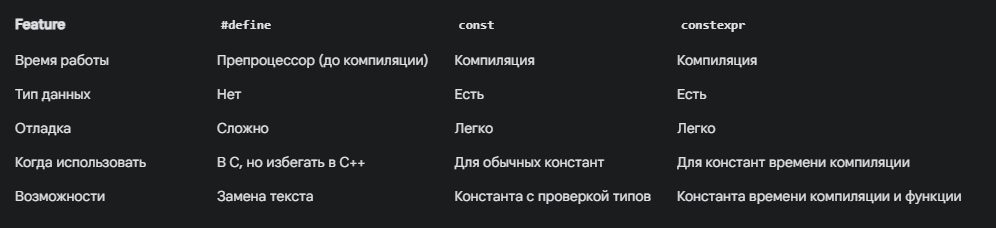
**4.6. constexpr (константы, вычисляемые на этапе компиляции)**

В C++11 появился ключевое слово constexpr, которое позволяет объявлять константы, значения которых должны быть вычислены на этапе компиляции. constexpr константы могут использоваться там, где требуются константы времени компиляции (например, при определении размеров массивов).



constexpr может использоваться не только для простых значений, но и для функций, которые возвращают константное значение на этапе компиляции.

**4.7. Различия между #define, const и constexpr**



**4.8. Закрепление материала**

1. Зачем нужны константы в программировании?
2. Какие способы объявления констант в C++ ты знаешь?
3. В чем разница между const и #define?
4. Почему const предпочтительнее, чем #define в C++?
5. Как и когда использовать constexpr?
6. Что такое константы времени компиляции и времени выполнения?

**Практическое задание:**

1. Объяви несколько констант разных типов (int, float, char, string, bool), используя const.
2. Попробуй объявить константы с помощью #define (но помни, что это не рекомендуется).
3. Попытайся изменить значение константы и посмотри, как отреагирует компилятор.
4. Попробуй использовать constexpr для константы и для функции.

**Глава 5: Ввод и вывод в консоль в C++**

Ввод и вывод — это способы, которыми программа может получать данные от пользователя (ввод) и отображать результаты своей работы (вывод). Консоль — это стандартное текстовое окно, через которое осуществляется это взаимодействие.

В C++ для ввода и вывода используются объекты из библиотеки iostream, в частности, std::cin (стандартный ввод) и std::cout (стандартный вывод).

**5.1. Вывод в консоль (std::cout)**

std::cout — это объект, представляющий стандартный поток вывода (обычно это консоль). Для вывода данных на экран используется оператор << (оператор вставки в поток).

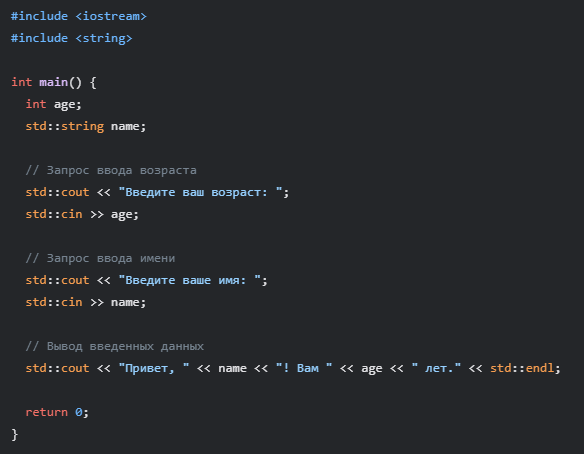


**Основные моменты:**

* std::endl: добавляет символ новой строки, перемещая курсор на следующую строку.
* Можно выводить текст (строковые литералы), значения переменных и выражения.
* Можно объединять несколько выводов в одну инструкцию, используя оператор <<.

**5.2. Ввод из консоли (std::cin)**

std::cin — это объект, представляющий стандартный поток ввода (обычно это клавиатура). Для получения данных от пользователя используется оператор >> (оператор извлечения из потока).

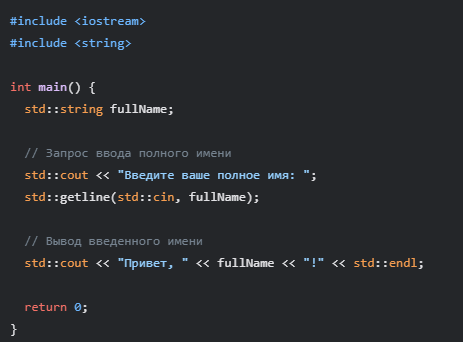


**Основные моменты:**

* std::cin >> variable; извлекает значение из потока ввода и сохраняет его в переменную variable.
* Тип переменной variable должен соответствовать типу вводимых данных.
* Оператор >> пропускает пробелы и переводы строк, и считывает данные до ближайшего пробела или перевода строки.
* Для ввода строк с пробелами нужно использовать std::getline().

**5.3. Ввод строк с пробелами (std::getline())**

Для ввода строк, которые могут содержать пробелы, используется функция std::getline(). Она считывает всю строку до символа новой строки (\n).



**5.4. Комбинирование ввода и вывода**

Можно комбинировать ввод и вывод в одной программе, чтобы создавать интерактивные приложения.



**Обрати внимание:**

* std::ws используется для пропуска начальных пробелов перед считыванием строки через std::getline. Без него предыдущий ввод (std::cin >> num1 >> num2;) может оставить перенос строки, который затем будет считан getline.

**5.5. Закрепление материала**

1. Для чего используется std::cout?
2. Для чего используется std::cin?
3. Как вывести на экран текст и значение переменной?
4. Как ввести значение с клавиатуры и сохранить его в переменную?
5. Как ввести строку с пробелами?
6. Напиши программу, которая запрашивает у пользователя два числа, а затем выводит их сумму.
7. Напиши программу, которая запрашивает у пользователя его имя и фамилию, а затем выводит их на экран в формате "Фамилия, Имя".

**Глава 6: Пространства имен и псевдонимы (using) в C++**

**6.1. Пространства имен (Namespaces)**

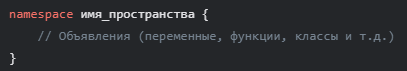
Пространства имен (namespaces) — это механизм для организации имен в C++, предотвращая конфликты имен. Представь, что пространство имен — это как папка, в которой хранятся имена переменных, функций, классов и других элементов программы. В разных папках могут быть файлы с одинаковыми именами, и это не вызовет конфликта.

**Зачем нужны пространства имен?**

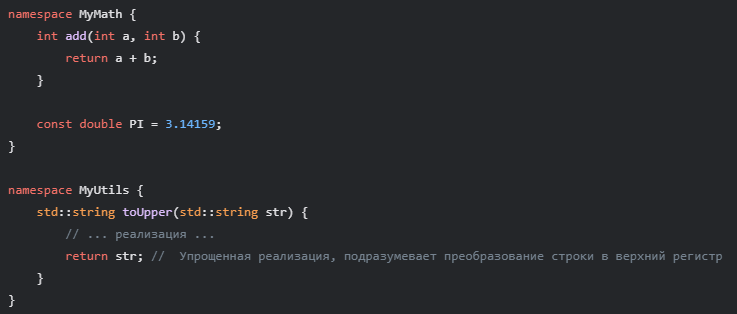
* **Предотвращение конфликтов имен:** Когда два разных разработчика или библиотеки используют одно и то же имя для разных целей, это может привести к конфликту. Пространства имен позволяют разделить эти имена, помещая их в разные области видимости.
* **Организация кода:** Пространства имен позволяют логически группировать связанные имена, что делает код более читаемым и структурированным.

**6.2. Объявление пространства имен**

Для объявления пространства имен используется ключевое слово namespace:



**Пример:**



**6.3. Доступ к элементам пространства имен**

Для доступа к элементам (переменным, функциям, классам) из пространства имен используется оператор разрешения области видимости ::  (scope resolution operator).



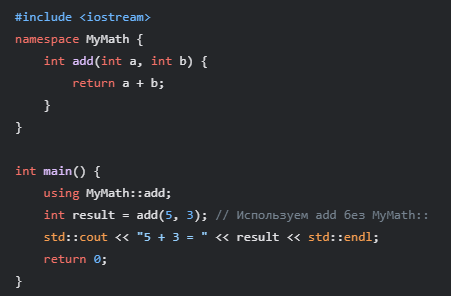
**6.4. Директива using**

Использование оператора :: для доступа к элементам пространства имен может сделать код громоздким. Для упрощения доступа можно использовать директиву using. Директива using бывает двух видов:

1. **using для конкретного элемента:**



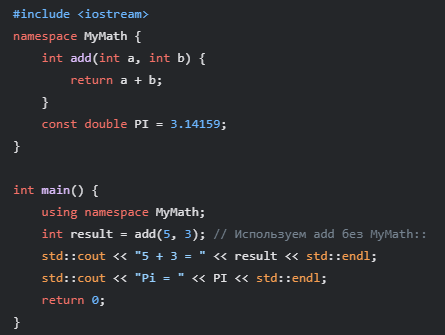
Это позволяет использовать элемент непосредственно, без указания имени пространства имен.



1. **using для всего пространства имен:**



Это позволяет использовать все элементы из указанного пространства имен без явного указания имени пространства.



1. **Важно:** Использование using namespace для всего пространства имен может привести к конфликтам имен, особенно в больших проектах. Поэтому, предпочтительно использовать using для конкретных элементов, или использовать using namespace в небольших файлах.

**6.5. Псевдонимы (Aliases)**

Иногда имена пространств имен или типов данных могут быть очень длинными. Для упрощения их использования можно определить псевдонимы (alias). Для этого используется ключевое слово using.



**Пример:**



**6.6. using namespace std;**

Ты наверняка заметил строку using namespace std; во многих примерах кода. std — это пространство имен стандартной библиотеки C++, в которой определены такие объекты как cout, cin, endl, string и многие другие.

Использование using namespace std; упрощает доступ ко всем элементам из std без явного указания std::. **В небольших программах это удобно, но в больших проектах это может привести к проблемам**, поэтому лучше избегать использование using namespace std; в глобальной области видимости.

**6.7. Закрепление материала**

1. Что такое пространства имен и для чего они нужны?
2. Как объявить пространство имен?
3. Как получить доступ к элементам пространства имен?
4. Для чего используется директива using?
5. В чем разница между using для конкретного элемента и using namespace?
6. Когда лучше использовать псевдонимы?
7. Почему не рекомендуется использовать using namespace std; в глобальной области видимости?

**Практическое задание:**

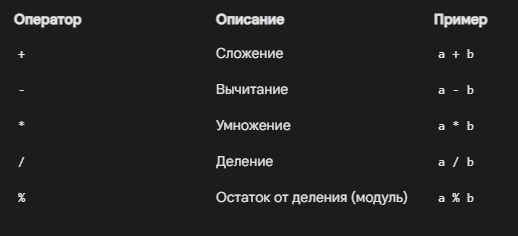
1. Создай свое собственное пространство имен MyProject.
2. Помести в него несколько функций и переменных.
3. Используй элементы из твоего пространства имен в функции main(), как с помощью ::, так и с помощью директивы using.
4. Создай псевдонимы для нескольких типов данных и используй их в своей программе.

**Глава 7: Арифметические операции в C++**

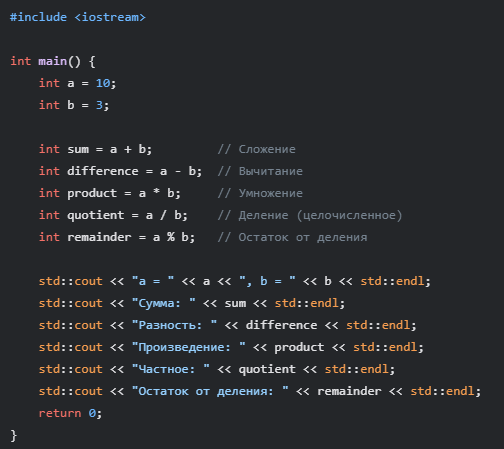
Арифметические операции позволяют выполнять математические вычисления над числовыми значениями. C++ предоставляет широкий набор арифметических операторов, которые позволяют выполнять сложение, вычитание, умножение, деление и другие операции.

**7.1. Основные арифметические операторы**

C++ поддерживает следующие основные арифметические операторы:



**Пример использования:**



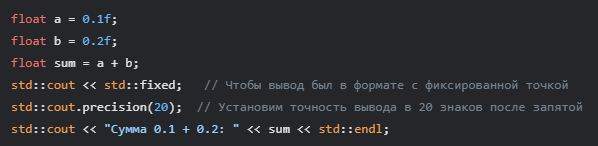
**Обратите внимание:**

* Деление целых чисел (int) в C++ является целочисленным. Это означает, что дробная часть результата отбрасывается. Например, 10 / 3 равно 3, а не 3.33.
* Оператор % (модуль) возвращает остаток от деления. Он работает только с целыми числами.

**7.2. Некоторые особенности при работе с числами с плавающей точкой**

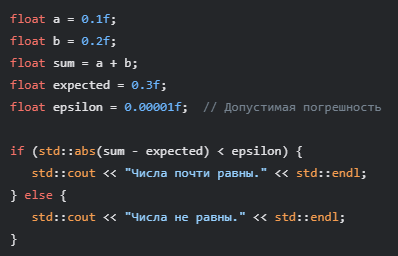
Числа с плавающей точкой (float и double) имеют особенности, которые нужно учитывать при выполнении арифметических операций:

* **Неточность представления:** Числа с плавающей точкой представляются в памяти компьютера с ограниченной точностью. Это может привести к небольшим ошибкам округления при выполнении вычислений.



Результат может быть не точно 0.3, а, например, 0.30000001192092896.

* **Сравнение чисел с плавающей точкой:** Из-за неточности представления, не рекомендуется сравнивать числа с плавающей точкой на равенство (==). Вместо этого лучше использовать сравнение с некоторой погрешностью (эпсилон).



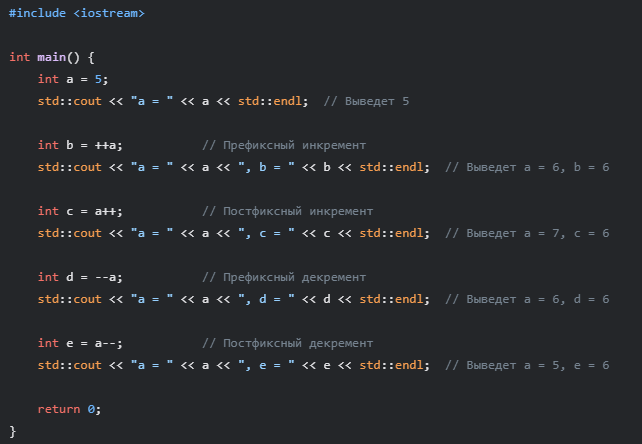
std::abs() - функция для вычисления модуля числа.

* **Деление на ноль:** Деление на ноль (/ 0.0f или / 0.0) для чисел с плавающей точкой не вызывает ошибку во время выполнения (как при целочисленном делении). В результате деления на ноль получается специальное значение "бесконечность" (inf) или "не число" (NaN).

**7.3. Инкремент и декремент**

Операторы инкремента (++) и декремента (--) используются для увеличения и уменьшения значения переменной на 1, соответственно. Они бывают двух видов:

* **Префиксный (++a, --a):** Сначала изменяет значение переменной, затем возвращает новое значение.
* **Постфиксный (a++, a--):** Сначала возвращает старое значение, затем изменяет значение переменной.

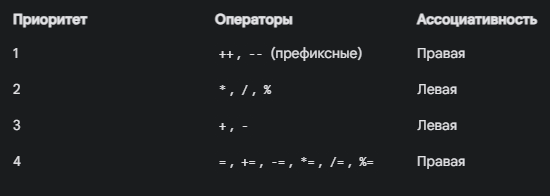


**7.4. Приоритет и ассоциативность операторов**

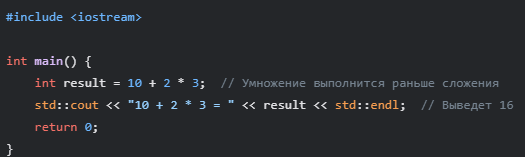
Приоритет и ассоциативность операторов определяют порядок выполнения операций в выражении.

* **Приоритет:** Операторы с более высоким приоритетом выполняются раньше операторов с более низким приоритетом. Например, умножение и деление имеют более высокий приоритет, чем сложение и вычитание.
* **Ассоциативность:** Определяет порядок выполнения операторов с одинаковым приоритетом. Операторы могут быть левоассоциативными (выполняются слева направо) или правоассоциативными (выполняются справа налево).

Вот таблица приоритетов и ассоциативности некоторых операторов (от высшего к низшему):

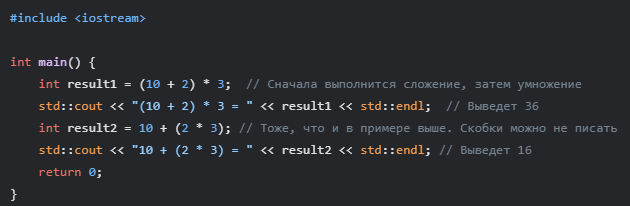


**Пример:**



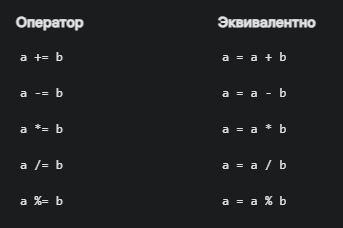
**7.5. Переопределение порядка операций**

Для переопределения порядка операций можно использовать круглые скобки (). Выражения в скобках всегда выполняются раньше операций вне скобок.

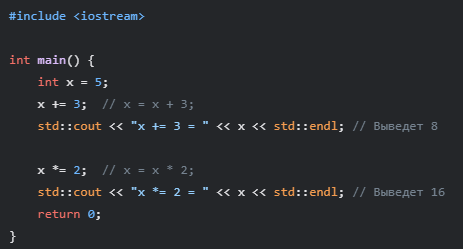


**7.6. Составные операторы присваивания**

Для удобства и краткости можно использовать составные операторы присваивания, которые выполняют операцию над переменной и присваивают ей результат:



**Пример:**



**7.7. Закрепление материала**

1. Перечисли основные арифметические операторы в C++.
2. В чем отличие целочисленного деления от деления чисел с плавающей точкой?
3. Почему при сравнении чисел с плавающей точкой не рекомендуется использовать оператор ==?
4. Чем отличается префиксный инкремент от постфиксного?
5. Что такое приоритет и ассоциативность операторов?
6. Как переопределить порядок выполнения операций?
7. Для чего используются составные операторы присваивания?

**Практическое задание:**

1. Напиши программу, которая запрашивает у пользователя два числа и выполняет все основные арифметические операции.
2. Напиши программу, которая вычисляет площадь и периметр круга, используя константу PI.
3. Поэкспериментируй с числами с плавающей точкой и посмотри, как работают особенности их представления.
4. Используй инкремент и декремент в разных ситуациях и посмотри, как это влияет на результат.
5. Напиши программу, где нужно переопределить порядок выполнения операций, используя скобки.

**Глава 8: Статическая типизация и преобразования типов в C++**

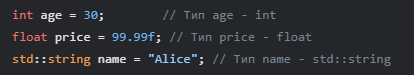
C++ — это язык со статической типизацией. Это означает, что тип каждой переменной должен быть известен на этапе компиляции. Компилятор проверяет типы данных во время компиляции и выдает ошибку, если обнаруживает несоответствие типов.

**8.1. Что такое статическая типизация?**

Статическая типизация обеспечивает следующие преимущества:

* **Безопасность:** Компилятор проверяет типы данных, что позволяет обнаружить ошибки, связанные с неправильным использованием типов, на этапе компиляции, а не во время выполнения программы.
* **Производительность:** Компилятор может оптимизировать код, основываясь на информации о типах данных.
* **Читаемость и понятность:** Явно указанные типы данных делают код более понятным и легким для восприятия.

В отличие от языков с динамической типизацией (например, Python), где тип переменной определяется во время выполнения, в C++ тип переменной объявляется явно при ее создании.



**8.2. Преобразования типов (Type Conversions)**

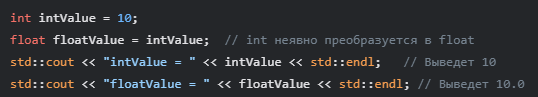
Преобразование типов — это процесс изменения типа данных значения. Иногда возникает необходимость преобразовать значение из одного типа в другой. В C++ существуют два основных вида преобразования типов:

1. **Неявные преобразования (Implicit conversions)**:  
   Это преобразования, которые компилятор выполняет автоматически без явного указания со стороны программиста. Неявные преобразования обычно выполняются, когда значение одного типа используется в контексте, где ожидается значение другого типа.
2. **Явные преобразования (Explicit conversions)**:  
   Это преобразования, которые программист должен явно указать в коде. Явные преобразования используются, когда компилятор не может автоматически выполнить преобразование или когда нужно явно контролировать процесс преобразования.

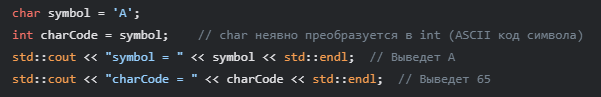
**8.3. Неявные преобразования**

Неявные преобразования выполняются автоматически, когда компилятор считает, что это безопасно. Примеры неявных преобразований:

* **Преобразование целых чисел в числа с плавающей точкой:**



* **Преобразование символов в целые числа:**

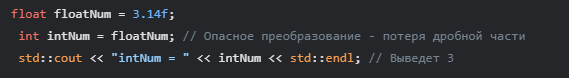


* **Преобразования в арифметических операциях:**  
  При выполнении арифметических операций с операндами разных типов, компилятор автоматически выполняет преобразование типа с меньшей точностью в тип с большей точностью.
* Правило преобразования типов при арифметических операциях: bool -> char -> short -> int -> long -> long long -> float -> double -> long double

**8.4. Опасные и безопасные преобразования**

Некоторые преобразования типов являются безопасными (то есть не приводят к потере данных), а другие — опасными (могут привести к потере данных или непредсказуемым результатам).

* **Безопасные преобразования:**
  + Преобразование целого числа в число с плавающей точкой.
  + Преобразование типа с меньшей точностью в тип с большей точностью (например, int в double).
* **Опасные преобразования:**
  + Преобразование числа с плавающей точкой в целое число (дробная часть отбрасывается).



* + Преобразование типа с большей точностью в тип с меньшей точностью (например, double в int).
  + Преобразование целых чисел в символы, когда значение целого числа выходит за пределы диапазона символов.

**8.5. Явные преобразования типов (Type Casting)**

Явные преобразования типов позволяют программисту явно контролировать процесс преобразования, в тех случаях когда неявные преобразования не подходят. В C++ существует несколько способов выполнить явное преобразование типов:

1. **C-style cast (преобразование в стиле C):**  
   Этот способ унаследован от языка C, его лучше избегать из-за его опасности.



Пример:



1. **static\_cast:**  
   Этот оператор следует использовать для безопасных и логичных преобразований.



Пример:



1. **dynamic\_cast:**  
   Этот оператор используется для преобразования указателей и ссылок на объекты классов в иерархии наследования. Он проверяет корректность преобразования во время выполнения. Мы обсудим dynamic\_cast позже при изучении ООП.
2. **const\_cast:**  
   Этот оператор используется для изменения const или volatile квалификаторов. Необходимо применять с осторожностью.
3. **reinterpret\_cast:**  
   Этот оператор используется для низкоуровневых преобразований, которые могут привести к непредсказуемым результатам. Необходимо использовать с осторожностью.

**Примеры явных преобразований:**



**Важно:**

* Используйте static\_cast для большинства преобразований типов, так как он безопаснее C-style cast.
* dynamic\_cast используется для преобразования типов в иерархии наследования.
* const\_cast используется для изменения const и volatile квалификаторов (используйте с осторожностью).
* reinterpret\_cast используется для низкоуровневых преобразований (используйте с осторожностью).

**8.6. Закрепление материала**

1. Что такое статическая типизация?
2. В чем ее преимущества?
3. Какие типы преобразований ты знаешь?
4. Что такое неявные преобразования и когда они выполняются?
5. Какие преобразования типов считаются безопасными, а какие — опасными?
6. Какие явные преобразования типов в C++ ты знаешь и когда их нужно применять?
7. Почему рекомендуется использовать static\_cast вместо C-style cast?

**Практическое задание:**

1. Напиши программу, которая демонстрирует неявные преобразования при арифметических операциях.
2. Попробуй разные явные преобразования типов (безопасные и опасные).
3. Исследуй, как изменяются значения при разных преобразованиях.
4. Напиши программу, которая запрашивает у пользователя число с плавающей точкой, а затем выводит его целочисленное значение (с отбрасыванием дробной части) и округленное значение (используя функцию std::round).

**Глава 9: Поразрядные операции в C++**

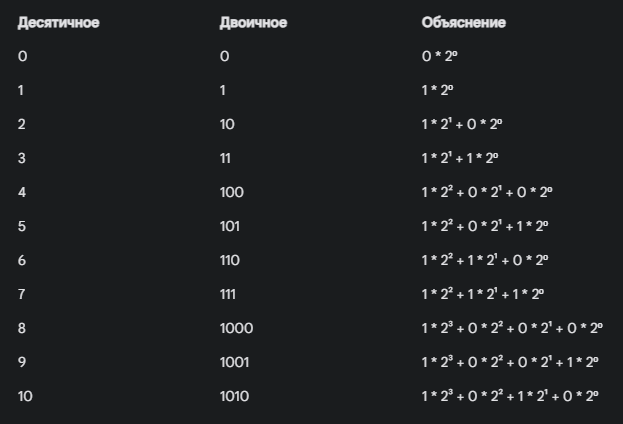
Поразрядные операции позволяют выполнять манипуляции с битами данных, то есть с отдельными нулями и единицами, составляющими двоичное представление чисел.

**9.1. Двоичное представление чисел**

В компьютере вся информация, включая числа, хранится в двоичной системе счисления, которая использует только два символа: 0 и 1. Каждый такой символ называется *битом*.

* **Десятичная система:** Основание 10, использует цифры от 0 до 9.
* **Двоичная система:** Основание 2, использует цифры 0 и 1.

Чтобы понять, как числа представляются в двоичной системе, рассмотрим примеры:



Для того чтобы представить более крупные числа в двоичном виде, добавляются новые разряды (биты).

**9.2. Представление отрицательных чисел**

Для представления отрицательных чисел обычно используется *дополнительный код* (two's complement).

1. **Инвертирование битов:** Сначала все биты положительного числа инвертируются (0 заменяются на 1, а 1 на 0).
2. **Прибавление 1:** Затем к результату прибавляется 1.

Например, представим число -5 в 8-битном формате:

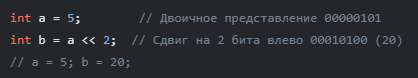
1. Положительное число 5: 00000101
2. Инвертирование битов: 11111010
3. Прибавление 1: 11111011

Таким образом, -5 представляется как 11111011.

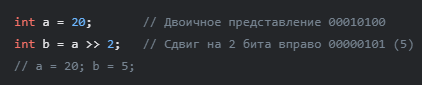
**9.3. Операции сдвига**

Операции сдвига перемещают биты числа влево или вправо:

* **Сдвиг влево (<<):** Сдвигает биты числа влево на заданное количество позиций. В освободившиеся биты справа добавляются нули. Эквивалентно умножению на 2 в степени сдвига.

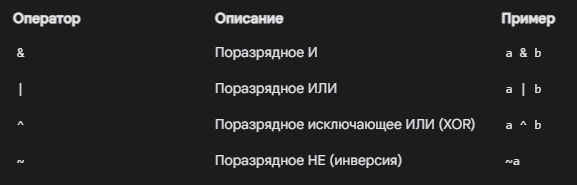


* **Сдвиг вправо (>>):** Сдвигает биты числа вправо на заданное количество позиций. В освободившиеся биты слева добавляются нули (для беззнаковых чисел) или сохраняется знаковый бит (для знаковых чисел). Эквивалентно целочисленному делению на 2 в степени сдвига.

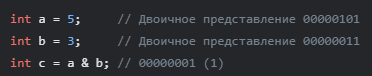


**9.4. Поразрядные операции**

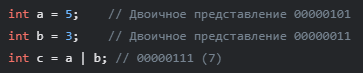
C++ поддерживает следующие поразрядные операторы:



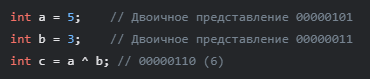
* **Поразрядное И (&):** Возвращает 1 в том бите, если оба соответствующих бита операндов равны 1, иначе 0.



* **Поразрядное ИЛИ (|):** Возвращает 1 в том бите, если хотя бы один из соответствующих битов операндов равен 1, иначе 0.



* **Поразрядное исключающее ИЛИ (^):** Возвращает 1 в том бите, если соответствующие биты операндов отличаются, иначе 0.



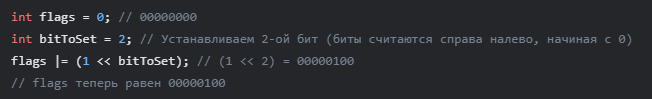
* **Поразрядное НЕ (~):** Инвертирует все биты числа (0 становится 1, а 1 становится 0).



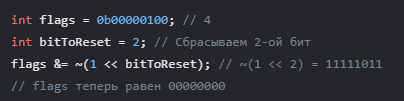
**9.5. Пример практического применения операций**

Давай рассмотрим несколько примеров, где поразрядные операции могут быть полезными:

* **Установка бита:** Чтобы установить определённый бит в 1, можно использовать поразрядное ИЛИ.



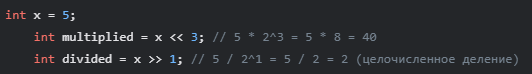
* **Сброс бита:** Чтобы сбросить определённый бит в 0, можно использовать поразрядное И с инвертированным значением.



* **Проверка состояния бита:** Чтобы проверить, установлен ли бит в 1, можно использовать поразрядное И.



* **Быстрое умножение и деление на степени 2:** Сдвиг влево и вправо могут быть использованы для быстрого умножения и деления на степени 2, но используйте это с осторожностью, так как при сдвиге может произойти переполнение.



**9.6. Закрепление материала**

1. Как числа представляются в двоичной системе?
2. Как представляются отрицательные числа с помощью дополнительного кода?
3. Что делают операторы сдвига влево и вправо?
4. Перечисли поразрядные операторы и опиши, что они делают.
5. Приведи примеры практического применения поразрядных операций.
6. Напиши функцию, которая устанавливает заданный бит в числе.
7. Напиши функцию, которая проверяет, установлен ли заданный бит в числе.

**Практическое задание:**

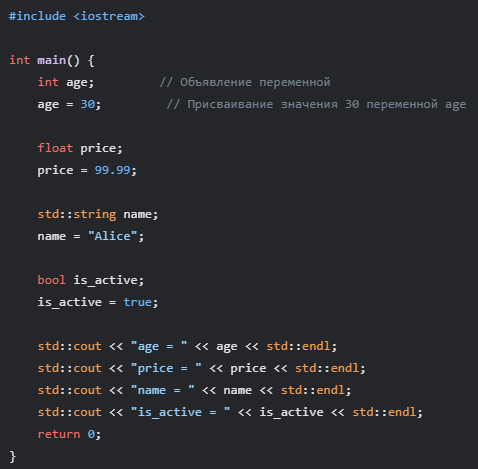
1. Напиши программу, которая преобразует десятичное число в двоичное представление.
2. Напиши программу, которая выполняет поразрядные операции (&, |, ^, ~) над двумя введенными числами.
3. Напиши программу, которая меняет местами значения двух целых чисел, используя поразрядное исключающее ИЛИ.
4. Попробуй использовать поразрядные операции для решения других задач, например, проверки чётности числа или выделения определенных битов из числа.

**Глава 10: Операции присваивания в C++**

Операция присваивания – это способ присвоить значение переменной. В C++ используется оператор =, который копирует значение, расположенное справа от него, в переменную, расположенную слева.

**10.1. Простой оператор присваивания =**

Простой оператор присваивания = является основным инструментом для присваивания значений переменным.



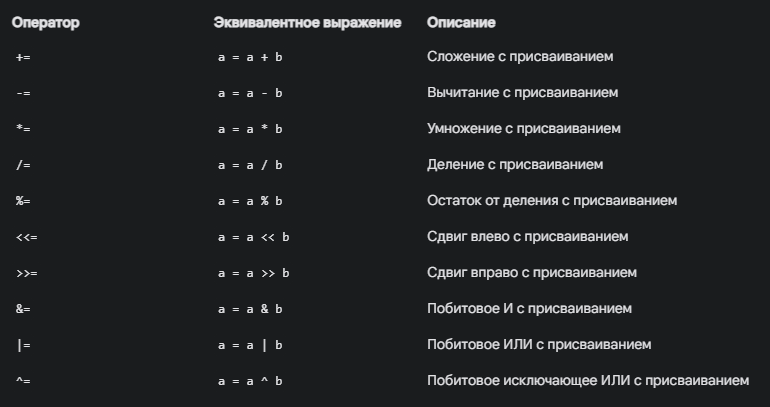
**Основные моменты:**

* Оператор присваивания = копирует значение из правой части в левую.
* Тип значения в правой части должен быть совместим с типом переменной в левой части.
* Оператор присваивания можно использовать для инициализации переменной при объявлении:



**10.2. Составные операторы присваивания**

C++ предоставляет составные операторы присваивания, которые сочетают в себе арифметическую операцию и присваивание. Это позволяет сократить запись кода и сделать его более читаемым.



**Примеры использования:**

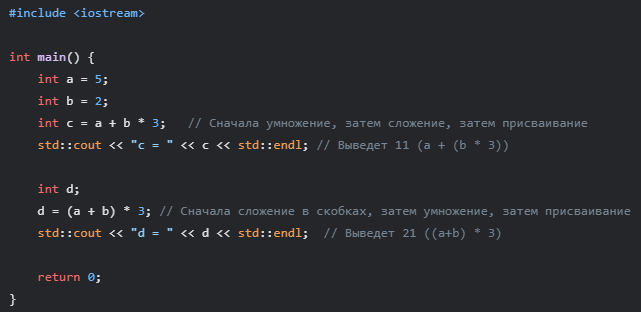


**Основные моменты:**

* Составные операторы присваивания позволяют сократить запись кода, выполняя операцию над переменной и присваивая результат ей же.
* Они делают код более читаемым и понятным.

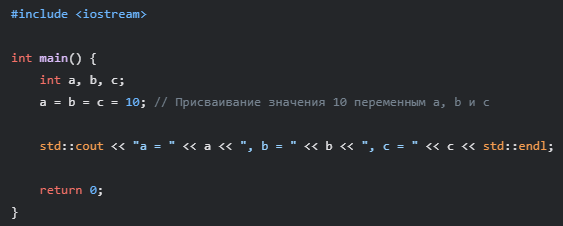
**10.3. Присваивание и порядок выполнения операций**

Оператор присваивания имеет самый низкий приоритет среди всех операторов. Это означает, что все вычисления в правой части выполняются до того, как значение будет присвоено переменной в левой части.



**10.4. Присваивание значений в цепочке**

В C++ возможно присваивать значения в цепочке:



В этом случае присваивание выполняется справа налево. Сначала 10 присваивается переменной c, затем значение c присваивается переменной b, затем значение b присваивается переменной a.

**10.5. Инициализация и присваивание**

Важно отличать инициализацию от присваивания.

* **Инициализация** выполняется при объявлении переменной и задает ей начальное значение.



* **Присваивание** выполняется после объявления переменной и изменяет ее значение.



**10.6. Закрепление материала**

1. Что делает оператор присваивания =?
2. Перечисли составные операторы присваивания.
3. В чем преимущество составных операторов присваивания?
4. Какой приоритет имеет оператор присваивания?
5. Как можно присвоить значения в цепочке?
6. В чем разница между инициализацией и присваиванием?

**Практическое задание:**

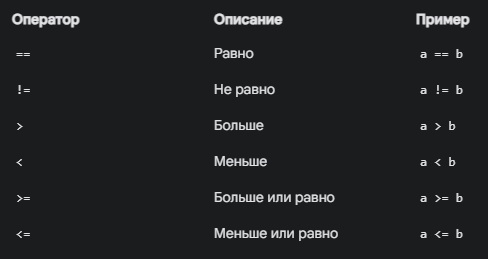
1. Напиши программу, которая использует все составные операторы присваивания.
2. Напиши программу, которая запрашивает у пользователя начальное значение и затем несколько раз изменяет его с помощью разных составных операторов присваивания.
3. Напиши программу, которая показывает порядок выполнения операций при присваивании.
4. Разберись с примерами с побитовыми операциями, как они работают с присваиваниями.

**Глава 11: Условные выражения в C++**

Условные выражения используются для сравнения значений и принятия решений на основе результатов этих сравнений. Они являются основой для управления потоком выполнения программы, позволяя ей выполнять различные действия в зависимости от условий.

**11.1. Операции сравнения**

Операции сравнения используются для сравнения двух значений и возвращают логическое значение true или false в зависимости от результата сравнения.



**Примеры использования:**

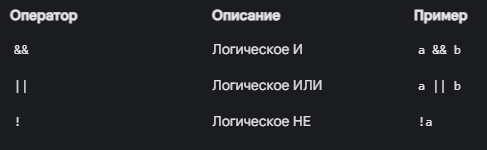


**Основные моменты:**

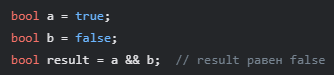
* Операции сравнения возвращают логические значения (true или false).
* Необходимо отличать оператор присваивания = от оператора сравнения ==.
* При сравнении чисел с плавающей точкой нужно учитывать их неточность представления. Лучше сравнивать с погрешностью.
* std::boolalpha — это манипулятор вывода, который выводит логические значения как true и false вместо 1 и 0.

**11.2. Логические операции**

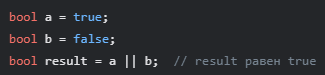
Логические операторы используются для объединения нескольких условий в более сложные выражения. C++ поддерживает следующие логические операторы:



* **Логическое И (&&):** Возвращает true только в том случае, если оба операнда true, иначе false.



* **Логическое ИЛИ (||):** Возвращает true, если хотя бы один из операндов true, иначе false.



* **Логическое НЕ (!):** Инвертирует значение операнда. Если операнд true, то результат false, и наоборот.



**Примеры использования:**



**11.3. Приоритет и ассоциативность операторов**

При выполнении условных выражений необходимо учитывать приоритет и ассоциативность операторов:



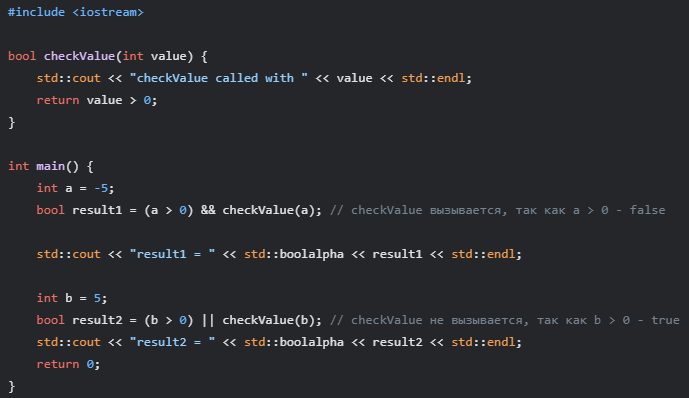
Оператор присваивания имеет самый низкий приоритет, поэтому сначала вычисляются условные выражения.

**11.4. Короткое вычисление**

Логические операторы && и || используют *короткое вычисление* (short-circuit evaluation). Это означает, что второй операнд вычисляется только в том случае, если это необходимо для определения результата всего выражения.

* При использовании && если первый операнд false, то второй операнд не вычисляется, так как результат выражения будет false независимо от значения второго операнда.
* При использовании || если первый операнд true, то второй операнд не вычисляется, так как результат выражения будет true независимо от значения второго операнда.

Это может быть полезно для оптимизации кода и предотвращения ошибок.



**11.5. Закрепление материала**

1. Перечисли операторы сравнения и опиши, что они делают.
2. Какие логические операторы есть в C++?
3. Что такое короткое вычисление и как оно работает?
4. Каков приоритет логических операторов?
5. В чем разница между операторами = и ==?

**Практическое задание:**

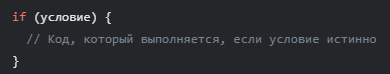
1. Напиши программу, которая запрашивает у пользователя два числа и проверяет, равны ли они, и какое из них больше.
2. Напиши программу, которая определяет, является ли введенное число положительным, отрицательным или нулем.
3. Напиши программу, которая определяет, является ли введенный год високосным. (Год является високосным, если он делится на 4, но не делится на 100, либо делится на 400.)
4. Напиши программу, которая определяет, попадает ли введенное число в заданный диапазон (например, от 10 до 100).
5. Используй логические операторы для создания сложных условий.

**Глава 12: Конструкция if-else и тернарный оператор в C++**

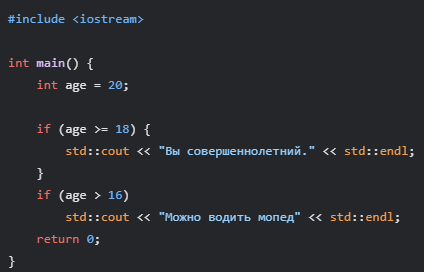
В этой главе мы подробно рассмотрим оператор if, оператор if-else, вложенные конструкции if-else и тернарный оператор, и узнаем, как их правильно использовать в разных ситуациях.

**12.1. Оператор if**

Оператор if позволяет выполнить блок кода только в том случае, если заданное условие истинно (true).

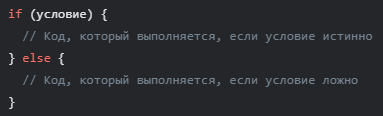


* условие: Выражение, которое возвращает логическое значение (true или false).
* { ... }: Блок кода, который выполняется, если условие true. Если блок состоит из одной инструкции, то фигурные скобки можно опустить.

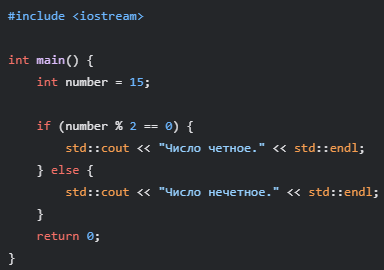


**12.2. Оператор if-else**

Оператор if-else позволяет выполнить один блок кода, если условие истинно (true), и другой блок кода, если условие ложно (false).

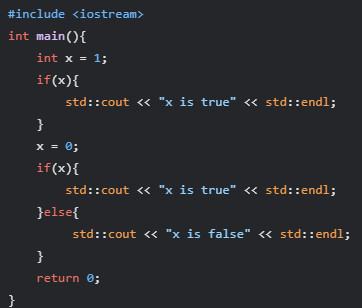


* условие: Выражение, которое возвращает логическое значение (true или false).
* { ... }: Блок кода, который выполняется, если условие true (после if).
* { ... }: Блок кода, который выполняется, если условие false (после else).



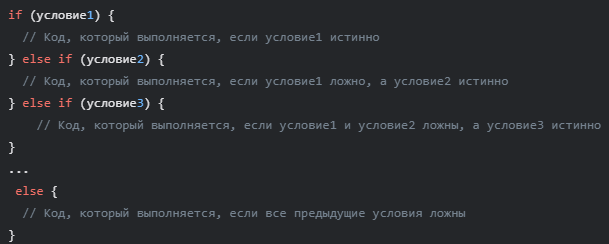
**12.3. Целочисленные условия**

В С++ любое целое число можно использовать в качестве условия. При этом 0 рассматривается как false, а любое другое целое число рассматривается как true.



**12.4. Конструкция if-else if-else**

Конструкция if-else if-else позволяет проверять несколько условий последовательно. Если одно из условий выполняется, то соответствующий блок кода выполняется, а остальные условия не проверяются.

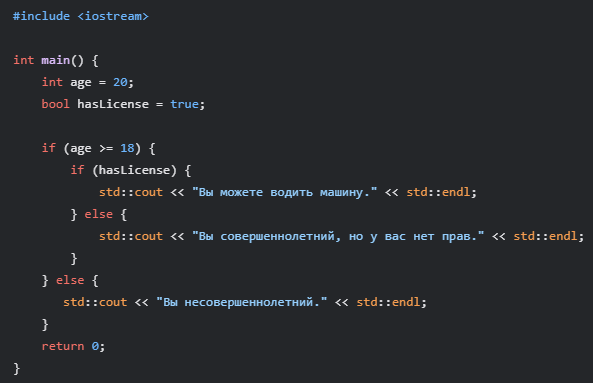


* Можно использовать любое количество блоков else if.
* Блок else является необязательным.



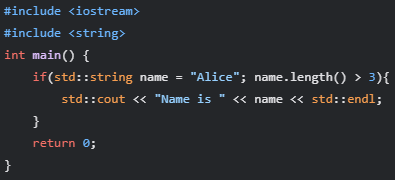
**12.5. Вложенные конструкции if-else**

Можно вкладывать операторы if-else друг в друга для создания более сложных условий.



**12.6. Блок if с инициализацией переменной**

Начиная со стандарта C++17, в блоке if можно не только задать условие, но и объявить переменную, видимую только внутри этого блока.

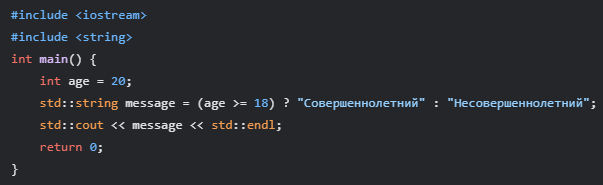


**12.7. Тернарный оператор**

Тернарный оператор ? : представляет собой сокращенную форму оператора if-else для простых условных выражений.



* Если условие истинно, то возвращается значение выражение1, иначе возвращается значение выражение2.



**Основные моменты:**

* Тернарный оператор подходит только для простых выражений.
* Значения выражение1 и выражение2 должны быть совместимы по типам.

**12.8. Закрепление материала**

1. Как работает оператор if?
2. Как работает оператор if-else?
3. Для чего нужна конструкция if-else if-else?
4. Можно ли вкладывать операторы if-else друг в друга?
5. Для чего используется тернарный оператор?
6. Когда лучше использовать тернарный оператор, а когда лучше использовать if-else?

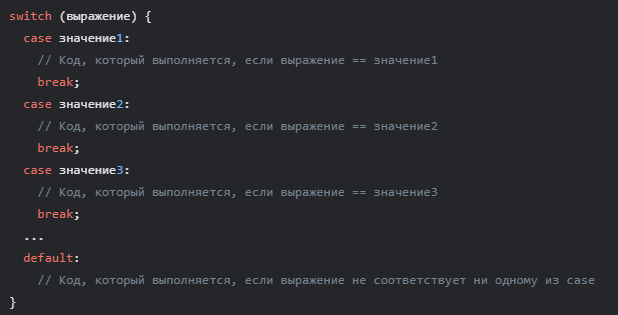
**Практическое задание:**

1. Напиши программу, которая запрашивает у пользователя число и проверяет, является ли оно положительным, отрицательным или нулем. Используй операторы if-else if-else.
2. Напиши программу, которая определяет максимальное из трех введенных чисел. Используй вложенные if-else.
3. Напиши программу, которая определяет, является ли введенный символ гласной или согласной буквой (в английском алфавите).
4. Напиши программу, которая использует тернарный оператор для определения четности введенного числа.
5. Напиши программу, которая имитирует простую систему авторизации.
6. Попробуй использование блока if с инициализацией переменной.

**Глава 13: Конструкция switch-case в C++**

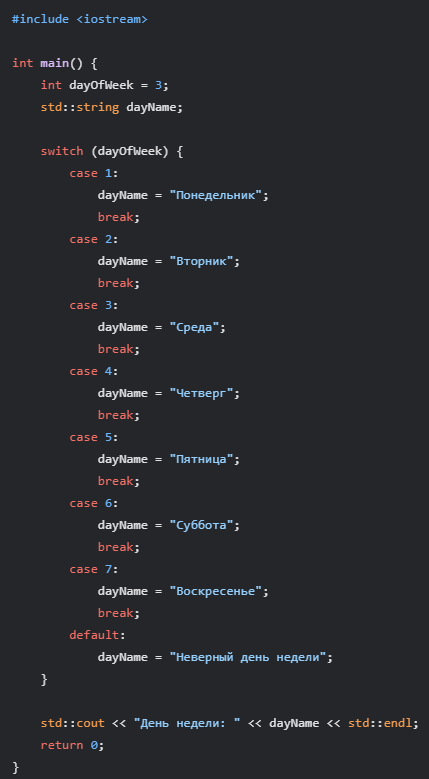
Оператор switch-case используется для выбора одного из нескольких блоков кода для выполнения в зависимости от значения целочисленного выражения или выражения, приводимого к целочисленному типу.

**13.1. Базовая структура оператора switch-case**



* switch (выражение): Выражение должно быть целочисленным типом (например, int, char, enum), или типом, приводимым к целочисленному.
* case значение: Константа или константное выражение целочисленного типа.
* break;: Оператор break завершает выполнение switch и передает управление следующей инструкции после switch.
* default:: Необязательный блок кода, который выполняется, если ни один из case не соответствует значению выражения.

**Пример использования:**



**Основные моменты:**

* Выполнение начинается с case, значение которого совпадает со значением выражения в switch.
* Если break не используется, выполнение "проваливается" к следующему case (или default), что может привести к непредсказуемому поведению.
* Блок default выполняется, если ни один из case не подходит.

**13.2. Совмещение условий**

Можно объединить несколько case для выполнения одного и того же блока кода, убрав оператор break между ними.



**Основные моменты:**

* break нужно ставить только после того, как мы хотим завершить выполнение блока кода.

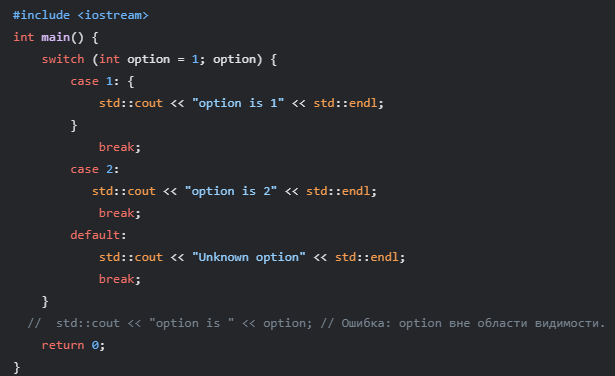
**13.3. Переменные в блоках case**

Внутри каждого блока case можно объявлять переменные, но их область видимости ограничена только этим блоком. Если вам нужна переменная, доступная во всех блоках case, объявите ее перед switch.

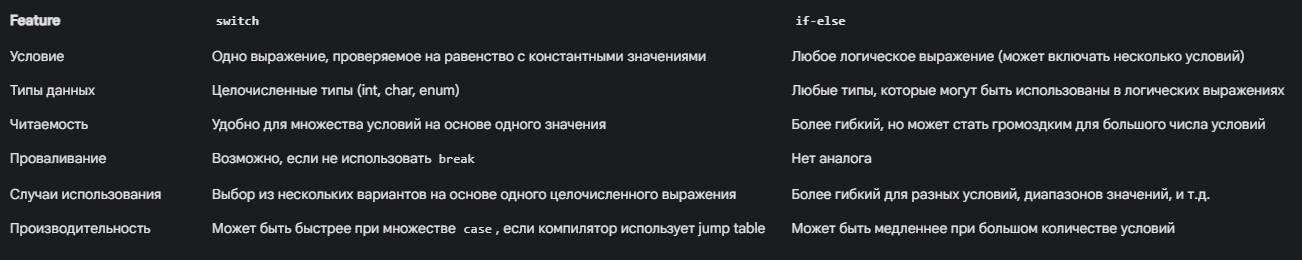


**13.4. switch с инициализацией переменной**

Начиная со стандарта C++17, в операторе switch можно объявлять переменную, которая будет видна только внутри этого оператора.



**13.5. Сравнение switch и if-else**



**13.6. Закрепление материала**

1. Как работает оператор switch-case?
2. Для чего нужен оператор break в switch-case?
3. Можно ли объединять несколько case в switch-case?
4. Где можно объявлять переменные в блоках case и в чем особенность их области видимости?
5. Что такое jump table?
6. В чем отличие switch-case от if-else if-else?
7. Когда лучше использовать switch-case, а когда if-else?

**Практическое задание:**

1. Напиши программу, которая запрашивает у пользователя номер месяца и выводит название месяца.
2. Напиши программу, которая имитирует простой калькулятор. Пользователь вводит два числа и символ операции (например, +, -, \*, /), а программа выводит результат.
3. Напиши программу, которая преобразует оценку в виде буквы (A, B, C, D, F) в ее словесное описание (например, "отлично", "хорошо" и т.д.).
4. Попробуй создать блок switch с инициализацией переменной.

**Глава 14: Циклы в C++**

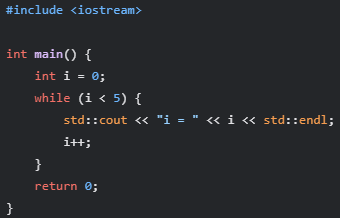
Циклы используются для многократного выполнения одного и того же блока кода. В C++ есть три основных вида циклов: while, for и do...while.

**14.1. Цикл while**

Цикл while выполняет блок кода до тех пор, пока заданное условие истинно.



* условие: Выражение, которое возвращает логическое значение (true или false).
* { ... }: Блок кода, который выполняется, пока условие true.

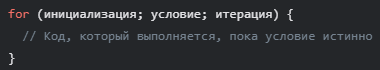


**Основные моменты:**

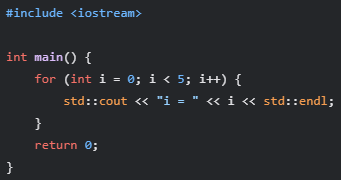
* Перед выполнением тела цикла всегда проверяется условие.
* Если условие ложно (false) с самого начала, тело цикла не выполнится ни разу.
* Внутри тела цикла необходимо изменять переменные, от которых зависит условие, чтобы цикл в конечном итоге завершился.

**14.2. Цикл for**

Цикл for используется для повторяющегося выполнения блока кода определенное количество раз.



* инициализация: Выполняется один раз перед началом цикла. Обычно используется для объявления и инициализации счетчика.
* условие: Выражение, которое возвращает логическое значение (true или false). Цикл выполняется, пока условие true.
* итерация: Выполняется после каждой итерации цикла. Обычно используется для изменения счетчика.
* { ... }: Блок кода, который выполняется, пока условие true.

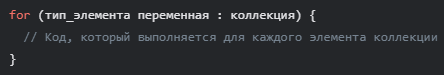


**Основные моменты:**

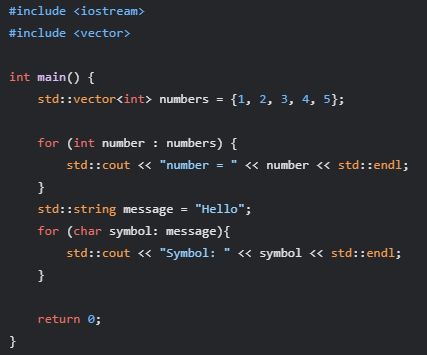
* Цикл for удобен, когда известно количество итераций.
* Все три части цикла (инициализация, условие, итерация) могут быть пустыми, но точки с запятой (;) остаются.
* Область видимости переменной, объявленной в блоке инициализации цикла for, ограничена телом цикла.

**14.3. Перебор значений в стиле for-each (цикл на основе диапазона)**

Начиная с C++11, появился цикл for, который позволяет перебирать элементы коллекций, массивов и строк.

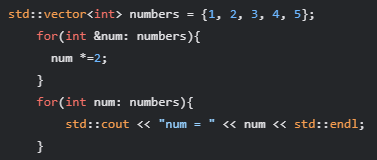


* тип\_элемента: Тип элементов в коллекции. Можно использовать auto для автоматического определения типа.
* переменная: Переменная, которая будет содержать текущий элемент коллекции.
* коллекция: Коллекция (например, массив, std::vector, строка), элементы которой нужно перебрать.



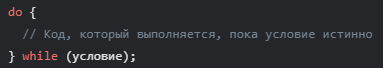
**Основные моменты:**

* Цикл for-each удобен для перебора всех элементов коллекции.
* Он более лаконичен и читаемый, чем обычный цикл for для перебора элементов коллекций.
* Можно использовать ссылку & \* для изменения значений элементов:

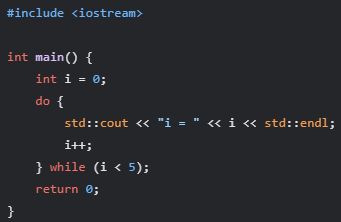


**14.4. Цикл do...while**

Цикл do...while похож на цикл while, но проверяет условие после выполнения блока кода. Это гарантирует, что блок кода будет выполнен хотя бы один раз.



* { ... }: Блок кода, который выполняется.
* условие: Выражение, которое возвращает логическое значение (true или false).

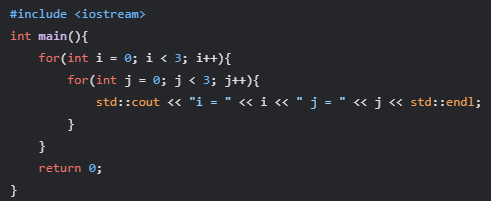


**Основные моменты:**

* Тело цикла do...while выполняется хотя бы один раз.
* Условие проверяется после выполнения тела цикла.

**14.5. Вложенные циклы for**

Можно вкладывать циклы друг в друга. Это позволяет, например, перебирать элементы двумерных массивов.



**14.6. Операторы continue и break**

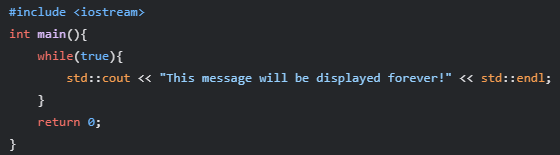
Операторы continue и break позволяют управлять выполнением циклов:

* continue: Завершает текущую итерацию цикла и переходит к следующей итерации.
* break: Прерывает выполнение цикла и передает управление следующей инструкции после цикла.



**14.7. Бесконечные циклы**

Цикл, условие которого всегда истинно, называется *бесконечным*. Бесконечные циклы часто используются в играх, операционных системах и других программах, где требуется непрерывное выполнение.



* Для того, что прервать работу бесконечного цикла обычно нужно использовать break внутри тела цикла.

**14.8. Закрепление материала**

1. Какие виды циклов есть в C++?
2. В чем отличие цикла while от цикла do...while?
3. Когда лучше использовать цикл for, а когда while или do...while?
4. Что делает цикл for-each?
5. Для чего нужны операторы continue и break?
6. Что такое бесконечный цикл?

**Практическое задание:**

1. Напиши программу, которая выводит все числа от 1 до 100, используя циклы while и for.
2. Напиши программу, которая запрашивает у пользователя число и выводит таблицу умножения для этого числа.
3. Напиши программу, которая вычисляет факториал введенного числа с помощью цикла for.
4. Напиши программу, которая запрашивает у пользователя числа и суммирует их до тех пор, пока пользователь не введет 0. Используй цикл do...while.
5. Напиши программу, которая перебирает элементы массива и выводит только четные числа, используя цикл for-each.
6. Напиши программу, которая выводит на экран "ёлочку" из символов, используя вложенные циклы for.
7. Напиши бесконечный цикл с выходом по условию.

**Глава 15: Ссылки в C++**

Ссылка — это альтернативное имя для существующей переменной. Представьте себе ссылку как псевдоним или другое имя, которое указывает на ту же область памяти, что и исходная переменная.

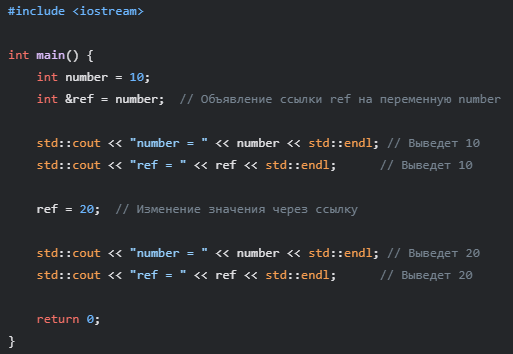
**15.1. Объявление ссылки**

Для объявления ссылки используется символ & после типа данных:



* тип\_данных: Тип данных переменной, на которую ссылается ссылка.
* &: Символ, который указывает, что это ссылка.
* имя\_ссылки: Имя ссылки.
* имя\_переменной: Имя существующей переменной, на которую ссылается ссылка.

**Пример:**



**Основные моменты:**

* Ссылка должна быть инициализирована при объявлении.
* Ссылка не является отдельной переменной. Она является другим именем для существующей переменной.
* Изменение значения через ссылку влияет на значение исходной переменной.
* После инициализации ссылку нельзя перенаправить на другую переменную.
* Ссылки не могут быть nullptr (null pointer).
* Ссылки могут быть только на левосторонние значения (lvalue).

**15.2. Ссылки как параметры функций**

Ссылки часто используются в качестве параметров функций для передачи аргументов по ссылке, а не по значению. Это позволяет функции изменять значения переменных, переданных в качестве аргументов.

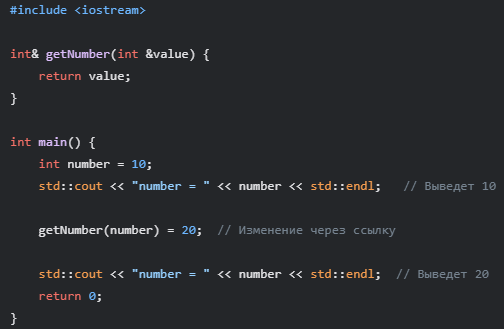


**Основные моменты:**

* Когда функция принимает аргумент по ссылке, функция работает с исходной переменной, а не с ее копией.
* Изменения параметра-ссылки в функции повлияют на исходную переменную.
* Это позволяет функциям возвращать более одного значения (через изменение параметров, переданных по ссылке).

**15.3. Ссылки как возвращаемое значение функции**

Функция может возвращать ссылку. Это позволяет избежать создания копии возвращаемого значения, но это нужно использовать с осторожностью.

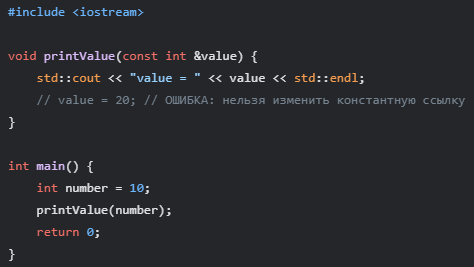


**Основные моменты:**

* Функция возвращает ссылку на исходную переменную, а не ее копию.
* Это позволяет изменять исходную переменную через возвращаемую ссылку.
* Следует избегать возвращения ссылок на локальные переменные из функции, так как они перестанут существовать после выхода из функции, что приведет к неопределенному поведению.

**15.4. Константные ссылки**

Константная ссылка (const &) позволяет ссылаться на переменную, не изменяя ее значение. Это полезно для передачи аргументов в функции, если функция не должна изменять их значения.

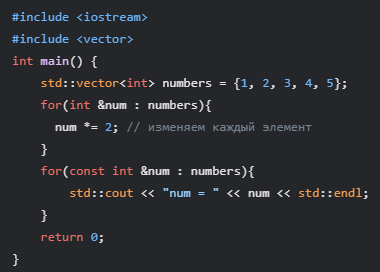


**Основные моменты:**

* Константная ссылка может ссылаться как на константную, так и на не константную переменную.
* Значение переменной, на которую ссылается константная ссылка, не может быть изменено через эту ссылку.

**15.5. Ссылки в цикле for**

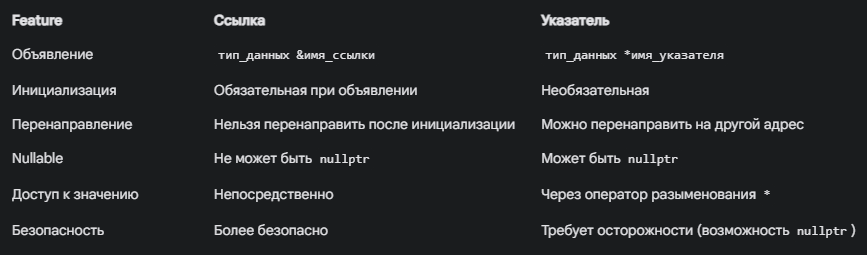
Ссылки особенно полезны в циклах for-each при переборе элементов коллекций или массивов. Использование ссылок позволяет изменять элементы коллекции, а также избежать лишнего копирования элементов при переборе.



* Если нет необходимости изменять значение элемента в цикле, лучше использовать const & для предотвращения случайного изменения.

**15.6. Различия между ссылками и указателями**

Ссылки и указатели похожи в том, что они оба позволяют косвенно обращаться к данным. Однако между ними есть важные отличия:



**15.7. Закрепление материала**

1. Что такое ссылка в C++?
2. Как объявить ссылку?
3. Почему ссылка должна быть инициализирована при объявлении?
4. Чем ссылка отличается от переменной?
5. Как использовать ссылки в качестве параметров функций?
6. Как использовать ссылки в качестве возвращаемых значений функций?
7. Что такое константная ссылка и когда ее нужно использовать?
8. Чем отличаются ссылки от указателей?
9. Как использовать ссылки в цикле for-each?

**Практическое задание:**

1. Напиши программу, которая демонстрирует работу с ссылками и их влияние на исходные переменные.
2. Напиши функцию, которая меняет местами значения двух целых чисел, используя ссылки.
3. Напиши функцию, которая принимает массив чисел и удваивает их значения, используя ссылки.
4. Напиши функцию, которая принимает строку и выводит ее на экран, используя константную ссылку.
5. Напиши программу, которая использует ссылки в цикле for-each для перебора элементов вектора.

**Глава 16: Массивы в C++**

Массив — это последовательная область памяти, которая хранит коллекцию элементов одного и того же типа. Каждый элемент массива имеет свой индекс, начиная с 0, который используется для доступа к этому элементу.

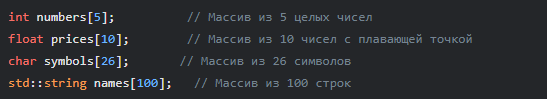
**16.1. Объявление массива**

Для объявления массива в C++ используется следующий синтаксис:



* тип\_данных: Тип данных элементов массива (например, int, float, char, std::string).
* имя\_массива: Имя, которое будет использоваться для доступа к массиву.
* размер\_массива: Количество элементов, которое может хранить массив (должно быть константным выражением, известным на этапе компиляции).

**Примеры:**



**16.2. Инициализация массива**

Массив можно инициализировать при объявлении:

1. **Инициализация списком значений:**
2. **Частичная инициализация:**

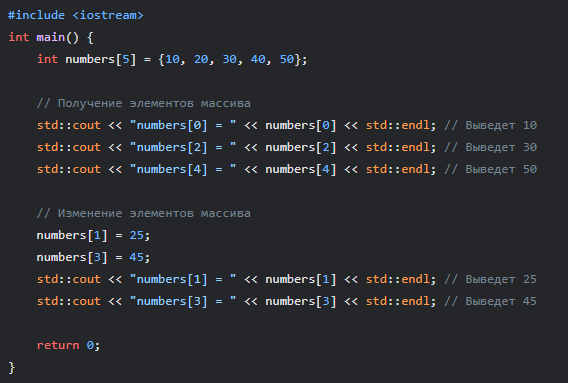
Если инициализируется только часть массива, оставшиеся элементы будут инициализированы нулями или значениями по умолчанию.



1. **Инициализация без указания размера (только при инициализации):**  
   Компилятор сам определит размер массива по количеству инициализированных элементов.

**16.3. Индексы массива**

Доступ к элементам массива осуществляется по их индексу. Индексация начинается с 0. Таким образом, первый элемент массива имеет индекс 0, второй — индекс 1, и так далее.



**Важно:**

* Обращение к массиву за пределами допустимого индекса (например, numbers[5] для массива из 5 элементов) приводит к непредсказуемому поведению программы и ошибкам.

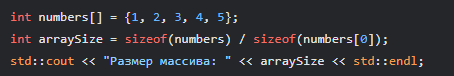
**16.4. Константные массивы**

Массив можно объявить как константный, используя ключевое слово const. Это означает, что значения элементов массива не могут быть изменены после его инициализации.



**16.5. Длина массива**

В C++ нет встроенного механизма для определения длины массива. Размер массива должен быть известен на этапе компиляции. Чтобы получить размер массива, можно использовать оператор sizeof:



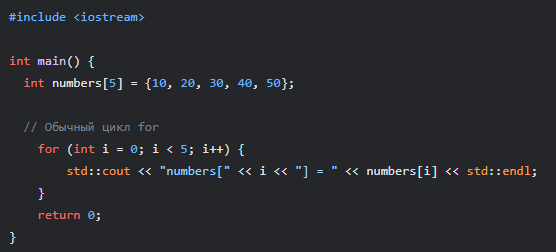
* sizeof(numbers): Возвращает размер всего массива в байтах.
* sizeof(numbers[0]): Возвращает размер одного элемента массива в байтах.

**Важно:**

* Этот метод работает только в том месте кода, где массив был объявлен.
* Для массивов, переданных в функцию, этот метод не работает, так как массив превращается в указатель на свой первый элемент.

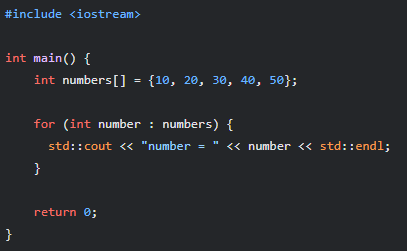
**16.6. Перебор элементов массива**

Для перебора элементов массива обычно используются циклы for:

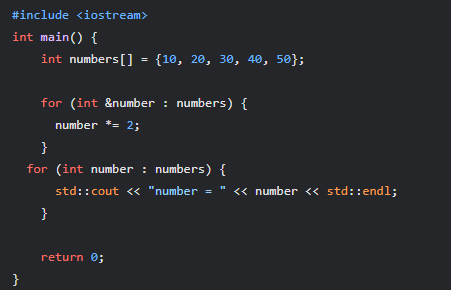


**16.7. Перебор элементов массива в стиле for-each**

В C++11 появился цикл for-each (range-based for loop), который упрощает перебор элементов массива:

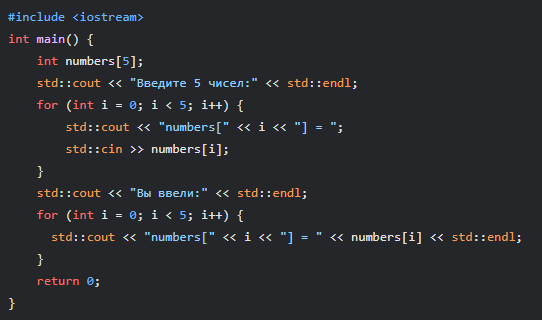


* Можно перебирать массив по ссылке.



**16.8. Ввод значений массива с консоли**

Значения элементов массива можно вводить с консоли, используя цикл:



**16.9. Закрепление материала**

1. Что такое массив?
2. Как объявляются массивы в C++?
3. Как инициализировать массив?
4. Как получить доступ к элементам массива?
5. Что такое константный массив?
6. Как определить длину массива?
7. Какие способы перебора элементов массива ты знаешь?
8. Как ввести значения массива с консоли?

**Практическое задание:**

1. Объявите массив целых чисел размером 10.
2. Заполните массив случайными числами.
3. Выведите все элементы массива на экран.
4. Найдите сумму и среднее арифметическое всех элементов массива.
5. Найдите минимальный и максимальный элементы массива.
6. Создайте константный массив, который содержит 12 значений (например, кол-во дней в месяце).
7. Напишите программу, которая позволяет пользователю ввести значения массива с консоли и затем выводит их на экран.

**Глава 17: Многомерные массивы в C++**

Многомерный массив — это массив, каждый элемент которого является другим массивом. Наиболее часто используемыми многомерными массивами являются двумерные массивы, которые представляют собой таблицы или матрицы.

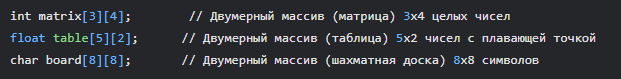
**17.1. Объявление многомерного массива**

Для объявления двумерного массива в C++ используется следующий синтаксис:



* тип\_данных: Тип данных элементов массива (например, int, float, char).
* имя\_массива: Имя массива.
* количество\_строк: Количество строк в массиве.
* количество\_столбцов: Количество столбцов в массиве.

**Примеры:**

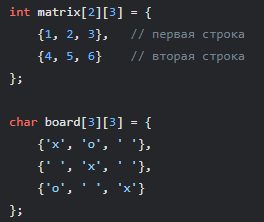


В C++ можно объявлять массивы с любым количеством измерений, но двумерные массивы наиболее распространены.

**17.2. Инициализация многомерных массивов**

Многомерные массивы можно инициализировать при объявлении несколькими способами:

1. **Инициализация списком значений:**

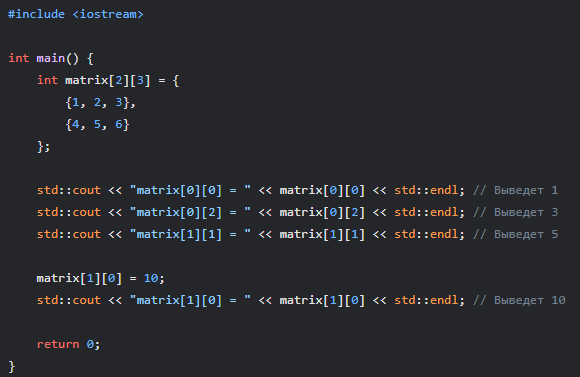


1. **Частичная инициализация:**  
   Если инициализируется только часть массива, то остальные элементы будут инициализированы нулями или значениями по умолчанию.



**17.3. Обращение к элементам многомерного массива**

Доступ к элементам многомерного массива осуществляется через индексы для каждого измерения. Индексация начинается с 0 для каждого измерения.

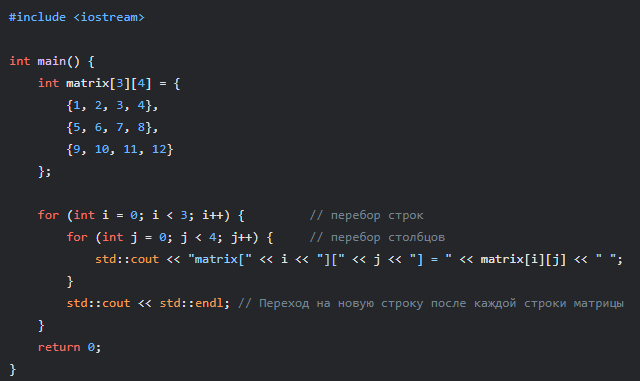


**Важно:**

* Обращение к элементам массива за пределами его границ является ошибкой.
* Первый индекс обращается к строке, второй - к столбцу.

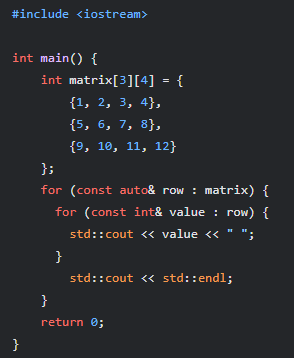
**17.4. Перебор элементов многомерного массива**

Для перебора элементов многомерного массива обычно используются вложенные циклы for:



**17.5. Перебор многомерного массива в стиле for-each (range-based for loop)**

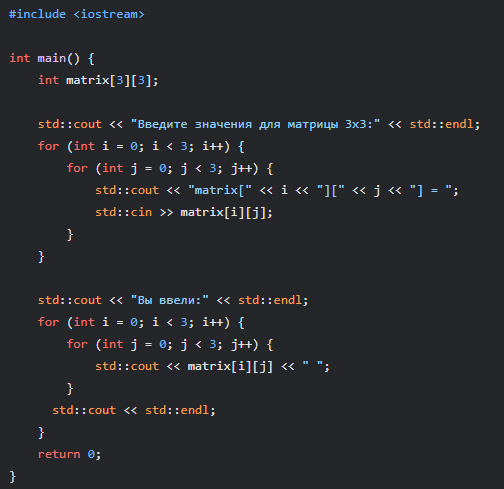
Можно использовать range-based for loop, но нужно помнить, что такой цикл может перебирать только одномерные массивы, следовательно для перебора двухмерного массива придется использовать вложенные for-each:



* Внешний цикл for(const auto& row: matrix) перебирает строки массива matrix.
* Внутренний цикл for (const int& value : row) перебирает элементы каждой строки.
* Можно использовать ссылки & для изменения значений.

**17.6. Ввод значений многомерного массива с консоли**

Значения элементов многомерного массива можно вводить с консоли, используя вложенные циклы:



**17.7. Закрепление материала**

1. Что такое многомерный массив?
2. Как объявить двумерный массив?
3. Как инициализировать многомерный массив?
4. Как получить доступ к элементам многомерного массива?
5. Как перебрать элементы многомерного массива с помощью вложенных циклов for?
6. Как перебрать элементы многомерного массива в стиле for-each?
7. Как ввести значения многомерного массива с консоли?

**Практические задания:**

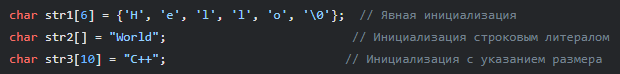
1. Объявите двумерный массив целых чисел размером 4x5.
2. Заполните массив случайными числами.
3. Выведите все элементы массива в виде таблицы.
4. Найдите сумму всех элементов массива.
5. Найдите сумму элементов в каждой строке массива.
6. Найдите сумму элементов в каждом столбце массива.
7. Напишите программу, которая позволяет пользователю ввести значения для матрицы 3x3 и затем выводит их на экран.
8. Реализуйте функцию, которая транспонирует двумерный массив (меняет местами строки и столбцы).

**Глава 18: Массивы символов в C++**

Массив символов (char array) — это последовательность символов, хранящихся в памяти. В C++ строки часто представляются именно как массивы символов, хотя также существует более удобный класс std::string.

**18.1. Объявление и инициализация массивов символов**

Массивы символов объявляются и инициализируются так же, как и массивы других типов данных:



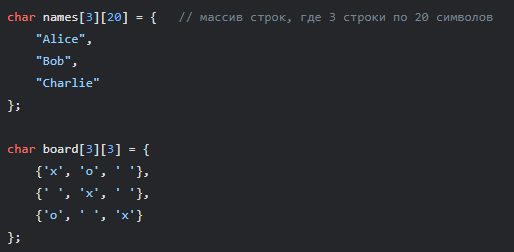
* char str1[6]: объявляет массив символов с размером 6. Последний элемент \0 (нулевой символ) отмечает конец строки. Это очень важно для корректной работы со строками в стиле C.
* char str2[] = "World";: объявляет массив символов, размер которого автоматически определяется по длине строкового литерала. Компилятор автоматически добавит в конец нулевой символ.
* char str3[10] = "C++";: объявляет массив символов с размером 10, инициализированный строкой "C++".

**Важно:**

* В C-строках используется нулевой символ \0 для обозначения конца строки.
* Массив символов должен иметь достаточно места для хранения всех символов строки и нулевого символа.

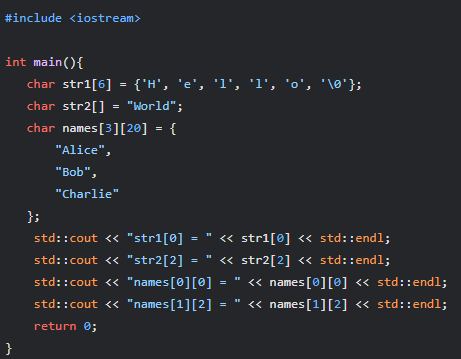
**18.2. Двухмерные массивы символов**

Двухмерные массивы символов используются для хранения набора строк, как таблицы, где каждая строка является массивом символов.



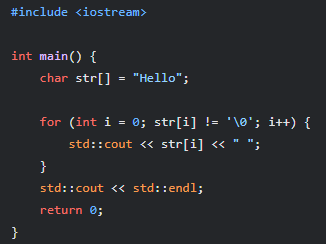
* char names[3][20]: объявляет массив, где можно хранить 3 строки, каждая длиной до 19 символов + нулевой символ.
* char board[3][3]: объявляет двумерный массив символов 3х3.

**18.3. Обращение к элементам массива символов**  
Доступ к элементам массива символов осуществляется по индексу, как и в обычных массивах.



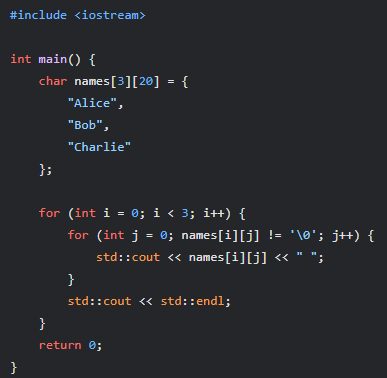
**18.4. Перебор массивов символов**

Для перебора массивов символов можно использовать цикл for и использовать индекс для доступа к символам.



**18.5. Перебор двухмерных массивов символов**

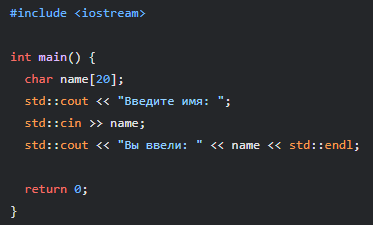
Для перебора двухмерных массивов символов можно использовать вложенные циклы for.



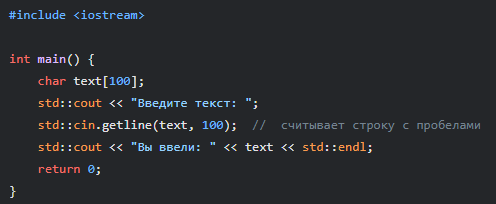
**18.6. Ввод символьных массивов / строк с консоли**

Для ввода массивов символов / строк с консоли можно использовать std::cin или std::cin.getline():

1. **Использование std::cin:**  
   std::cin считывает ввод до первого пробела.



1. **Использование std::cin.getline():**  
   std::cin.getline() считывает строку до символа новой строки или до достижения максимального размера, указанного при объявлении массива.



* + Первый аргумент text — это массив символов.
  + Второй аргумент 100 — максимальный размер строки, которую нужно считать (не включая нулевой символ).

**Важно:**

* + Функция getline считывает символы до тех пор пока не встретится символ новой строки, а не пробел как в случае с cin >>.
  + Функция std::cin.getline() может привести к переполнению буфера, если пользователь введет строку, длиннее чем размер массива. Для избежания этой проблемы, следует использовать класс std::string.

**18.7. Закрепление материала**

1. Что такое массив символов?
2. Как объявляются и инициализируются массивы символов?
3. Для чего нужен нулевой символ \0 в массивах символов?
4. Как объявить и инициализировать двухмерный массив символов?
5. Как перебрать массив символов?
6. Как перебрать двухмерный массив символов?
7. Как ввести строку с консоли, используя std::cin?
8. Как ввести строку с консоли, используя std::cin.getline()?

**Практические задания:**

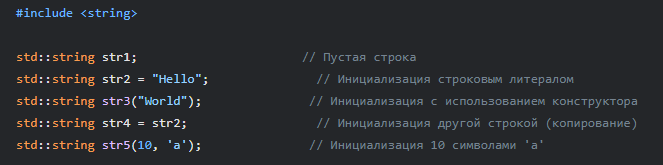
1. Объявите массив символов и инициализируйте его строкой "Programming".
2. Выведите каждый символ массива на экран.
3. Объявите двумерный массив символов 3x10 и инициализируйте его тремя именами.
4. Выведите все имена из двумерного массива на экран.
5. Напишите программу, которая запрашивает у пользователя строку, выводит ее на экран, а затем выводит ее в обратном порядке.
6. Напишите программу, которая запрашивает у пользователя несколько строк, сохраняет их в двумерном массиве, а затем выводит их на экран.
7. Создайте программу, которая запрашивает у пользователя текст и находит в нём количество гласных букв (a, e, i, o, u).

**Глава 19: Строки в C++ (std::string)**

std::string — это класс в C++, который предоставляет удобный и безопасный способ работы со строками. Он автоматически управляет памятью, выделяемой для строки, и предоставляет множество методов для работы с текстом. Для его использования нужно подключить библиотеку string (#include <string>).

**19.1. Объявление и инициализация строк**

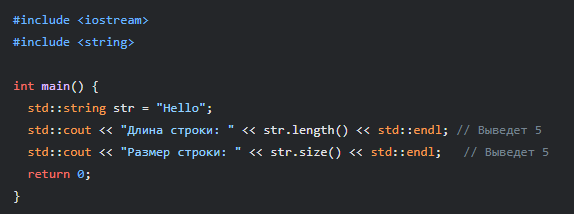
Для объявления и инициализации строки используется следующий синтаксис:



* std::string str1: объявляет пустую строку.
* std::string str2 = "Hello";: объявляет строку и инициализирует ее строковым литералом "Hello".
* std::string str3("World"): объявляет строку и инициализирует ее строковым литералом "World" с использованием конструктора.
* std::string str4 = str2: создает копию строки str2.
* std::string str5(10, 'a'): создает строку, состоящую из 10 символов 'a'.

**19.2. Получение длины строки**

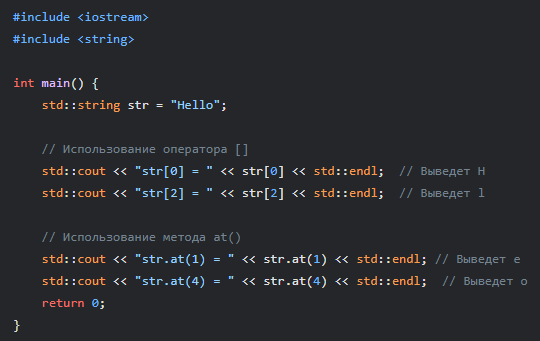
Для получения длины строки используется метод length() или size():



* Методы length() и size() возвращают количество символов в строке.
* Оба метода делают одно и тоже и их можно использовать взаимозаменяемо.

**19.3. Получение символов строки**

Доступ к символам строки можно получить через оператор [] или метод at():

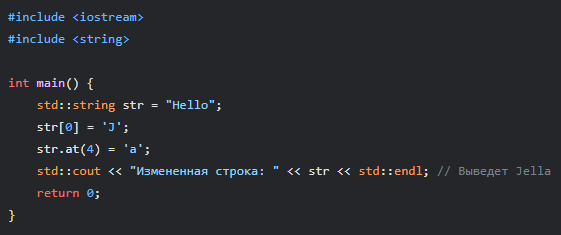


**Важно:**

* Оператор [] не выполняет проверку границ, поэтому если обратиться по индексу, выходящему за пределы строки, может произойти ошибка.
* Метод at() выполняет проверку границ и выбрасывает исключение, если индекс недействителен.

**19.4. Изменение символов строки**

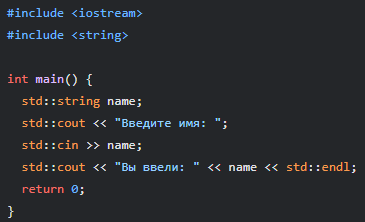
Символы строки можно изменять, используя оператор [] или метод at():



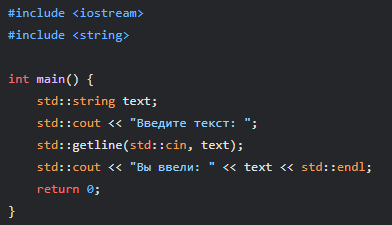
**19.5. Чтение строки с консоли**

Для считывания строки с консоли можно использовать std::cin или std::getline():

1. **Использование std::cin:**  
   std::cin считывает ввод до первого пробела.



1. **Использование std::getline():**  
   std::getline() считывает целую строку, включая пробелы, до символа новой строки.



**19.6. Добавление символов и строк**

Класс std::string предоставляет методы для добавления символов и строк:



**19.7. Закрепление материала**

1. Что такое std::string?
2. Как объявить и инициализировать строку?
3. Как получить длину строки?
4. Как получить доступ к символам строки?
5. Как изменить символы строки?
6. Как считать строку с консоли, используя std::cin и std::getline()?
7. Как добавить символы и строки к существующей строке?

**Практические задания:**

1. Объявите строку и инициализируйте ее значением "Hello, C++!".
2. Выведите длину этой строки.
3. Выведите первый и последний символы этой строки.
4. Измените несколько символов в строке.
5. Напишите программу, которая запрашивает у пользователя имя и фамилию, а затем выводит их в формате "Фамилия, Имя".
6. Напишите программу, которая запрашивает у пользователя предложение и выводит его, заменив все пробелы на дефисы.
7. Напишите программу, которая запрашивает у пользователя два слова и соединяет их в одну строку.

**Глава 20: Указатели в C++**

Указатель — это переменная, которая хранит адрес в памяти другой переменной. Представьте себе указатель как стрелку, которая указывает на место в памяти, где хранится какое-либо значение.

**20.1. Что такое указатель?**

* **Переменные:** Когда мы объявляем переменную, например int x = 10;, компилятор выделяет область памяти для хранения значения 10 и связывает эту область с именем x.
* **Указатели:** Указатель — это переменная, которая хранит адрес в памяти этой области. То есть, указатель "указывает" на то место, где хранится значение переменной.

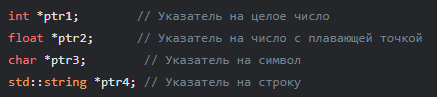
**20.2. Определение указателя**

Для объявления указателя используется символ \* после типа данных:



* тип\_данных: Тип данных, на который будет указывать указатель (например, int, float, char, std::string).
* \*: Символ, указывающий, что это указатель.
* имя\_указателя: Имя указателя.

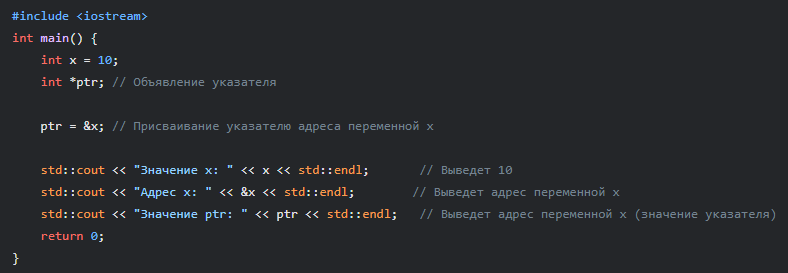
**Примеры:**



**Важно:** Тип указателя должен соответствовать типу данных переменной, на которую он указывает. Указатель типа int\* должен указывать на переменную типа int.

**20.3. Получение адреса и оператор &**

Для получения адреса переменной используется оператор & (оператор взятия адреса).

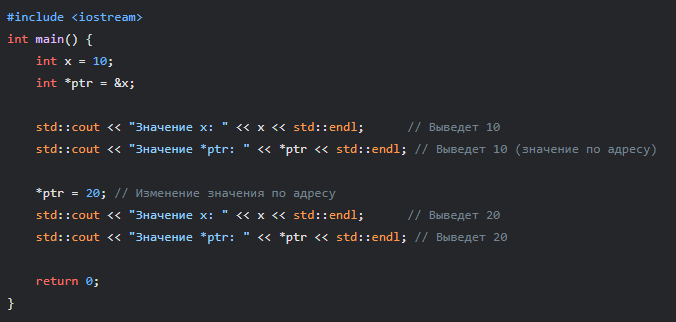


**Основные моменты:**

* &x возвращает адрес переменной x в памяти.
* Указатель ptr теперь хранит адрес переменной x.

**20.4. Получение значения по адресу (разыменование указателя) и оператор \***

Для получения значения переменной, на которую указывает указатель, используется оператор \* (оператор разыменования).

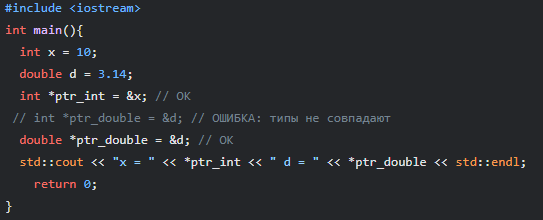


**Основные моменты:**

* \*ptr возвращает значение, хранящееся по адресу, который хранится в указателе ptr.
* Изменение значения через указатель меняет значение исходной переменной.
* \*ptr — это «разыменованный» указатель, то есть обращение к значению по адресу.

**20.5. Указатели и типы данных**

Тип указателя должен соответствовать типу данных переменной, на которую он указывает.



**20.6. Неинициализированные указатели**

Если указатель не инициализирован, то он будет содержать случайный адрес в памяти. Работа с неинициализированными указателями может привести к ошибкам.

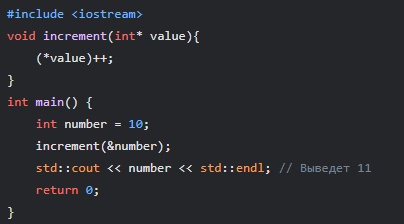


**Важно:**

* Всегда инициализируйте указатель перед его использованием.

**20.7. Указатели как параметры функций**

Указатели часто используются для передачи параметров в функции, чтобы функция могла изменять значение переменной, переданной в качестве аргумента.



**20.8. Закрепление материала**

1. Что такое указатель?
2. Как объявить указатель?
3. Что такое оператор взятия адреса &?
4. Что такое оператор разыменования \*?
5. Почему важно, чтобы тип указателя соответствовал типу данных переменной, на которую он указывает?
6. Почему не следует использовать неинициализированные указатели?
7. Как использовать указатели в качестве параметров функций?

**Практические задания:**

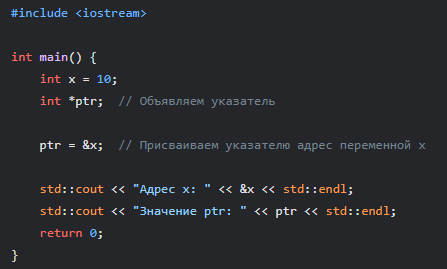
1. Объявите целую переменную и указатель на нее.
2. Присвойте указателю адрес переменной.
3. Измените значение переменной через указатель.
4. Выведите значение переменной до и после изменения.
5. Напишите функцию, которая принимает указатель на целое число и удваивает значение по этому адресу.
6. Напишите программу, которая обменивает значения двух переменных, используя указатели.
7. Создайте несколько переменных разных типов и указателей на них. Выведите значения и адреса этих переменных.

**Глава 21: Операции с указателями в C++**

В этой главе мы рассмотрим различные операции, которые можно выполнять с указателями, включая присваивание адреса, разыменование, присвоение указателю другого указателя, нулевые указатели, ссылки на указатели, получение адреса указателя, операции сравнения и приведение типов указателей.

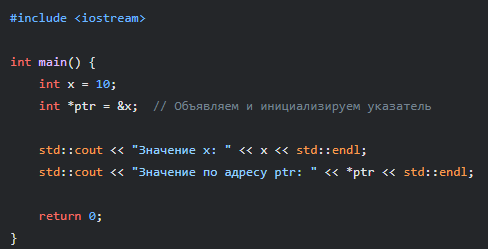
**21.1. Присваивание адреса**

Присваивание адреса указателю — это процесс, при котором указателю присваивается адрес переменной. Это делается с использованием оператора & (оператора взятия адреса).



**21.2. Разыменование указателя**

Разыменование указателя — это процесс получения значения, хранящегося по адресу, который хранится в указателе. Это делается с помощью оператора \* (оператора разыменования).

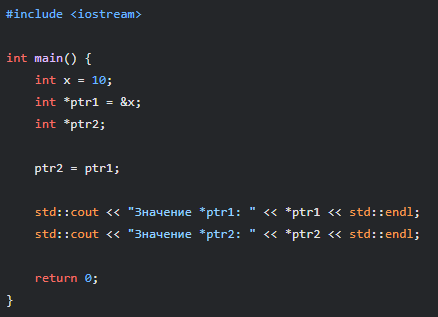


**Важно:**

* Разыменование неинициализированного или нулевого указателя приведет к неопределенному поведению.

**21.3. Присвоение указателю другого указателя**

Указателю можно присвоить значение другого указателя, если они указывают на данные одного и того же типа:



* Теперь ptr1 и ptr2 указывают на одну и ту же область памяти, где хранится переменная x.

**21.4. Нулевые указатели**

Нулевой указатель (null pointer) — это указатель, который не указывает ни на какую область памяти. Для обозначения нулевого указателя в C++ используется nullptr (начиная с C++11), который является предпочтительным вариантом. До C++11 для этих целей использовали 0 или NULL.

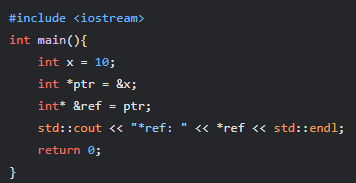


**Важно:**

* Разыменование нулевого указателя приводит к ошибке во время выполнения программы (обычно к сегментационной ошибке).
* Нулевые указатели часто используются для обозначения, что указатель не указывает ни на какой валидный объект, или для инициализации указателя перед его использованием.
* Использование nullptr более безопасно чем NULL.

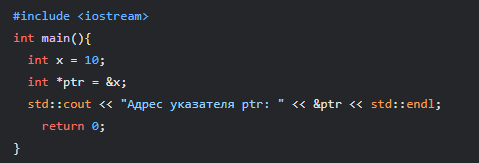
**21.5. Ссылки на указатели**

Можно создавать ссылки на указатели.



* int\* &ref = ptr: объявляет ссылку ref на указатель ptr.
* Изменение ref повлияет на ptr.

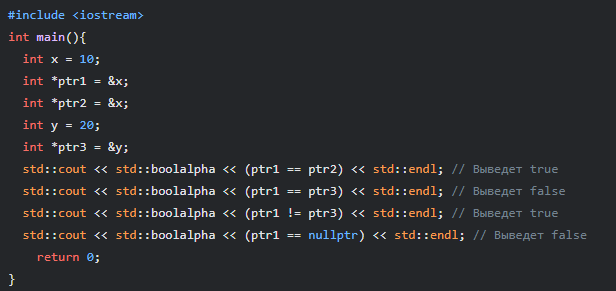
**21.6. Адрес указателя**  
Указатель, как и любая другая переменная, имеет адрес в памяти. Для получения адреса указателя используется оператор &.



* &ptr возвращает адрес переменной ptr, которая хранит адрес другой переменной.

**21.7. Операции сравнения указателей**

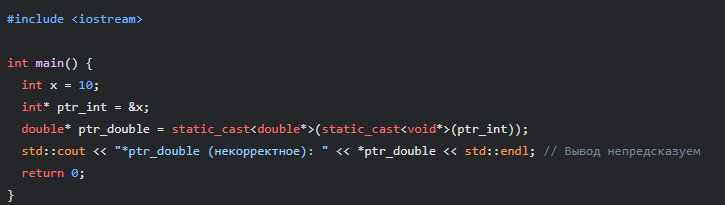
Указатели можно сравнивать с помощью операторов == (равно) и != (не равно). Сравнивать можно только указатели одного типа.



**Важно:**

* Сравнивать указатели имеет смысл только, если они указывают на один и тот же блок памяти (на одну переменную или на элементы одного массива) или с nullptr.
* Сравнивать указатели на разные переменные как правило не имеет смысла, т.к. результат сравнения - это адрес в памяти, который обычно не используется.

**21.8. Приведение типов указателей (Type Casting)**  
Можно выполнять приведение типов указателей, но это следует делать с осторожностью, поскольку некорректное приведение может привести к ошибкам.



* В данном примере мы преобразуем int\* в void\*, а затем в double\*, но этот код может привести к ошибкам, т.к. данные лежат в памяти по типу int.
* Использование static\_cast, reinterpret\_cast при работе с указателями должно быть хорошо обдумано.

**21.9. Закрепление материала**

1. Как присвоить адрес переменной указателю?
2. Что такое разыменование указателя и как его использовать?
3. Как присвоить одному указателю значение другого указателя?
4. Что такое нулевой указатель и как его объявить?
5. Можно ли создавать ссылки на указатели?
6. Как получить адрес самого указателя?
7. Как сравнивать указатели?
8. Когда нужно использовать приведение типов указателей?

**Практические задания:**

1. Объявите две целые переменные и два указателя на них.
2. Присвойте указателям адреса переменных.
3. Измените значения переменных, используя разыменование указателей.
4. Создайте указатель и присвойте ему значение другого указателя. Проверьте, что оба указателя указывают на одну и ту же область памяти.
5. Объявите указатель и инициализируйте его нулевым значением. Проверьте, что попытка разыменования этого указателя приведет к ошибке.
6. Создайте ссылку на указатель и попробуйте изменить значение исходного указателя через ссылку.
7. Сравните между собой два указателя, если они указывают на разные переменные и на одну и ту же переменную.

**Глава 22: Арифметика указателей в C++**

Арифметика указателей — это набор операций, которые позволяют выполнять арифметические действия над указателями, то есть перемещать указатель по памяти.

**22.1. Базовые принципы арифметики указателей**

* **Тип указателя важен:** Арифметика указателей зависит от типа данных, на который указывает указатель. Если указатель типа int\* указывает на ячейку памяти, то увеличение указателя на 1 переместит указатель на размер целого числа (обычно 4 байта), а не на 1 байт.
* **Перемещение по памяти:** Арифметические операции перемещают указатель на заданное количество *элементов* данного типа, а не на заданное количество байтов.

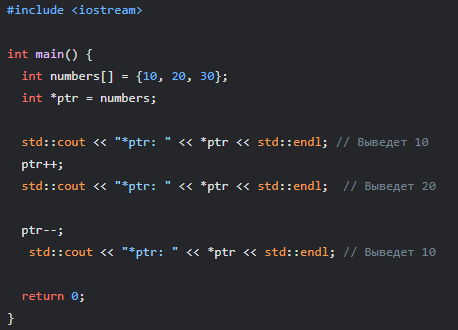
**22.2. Операции с указателями**

1. **Сложение и вычитание целых чисел:**



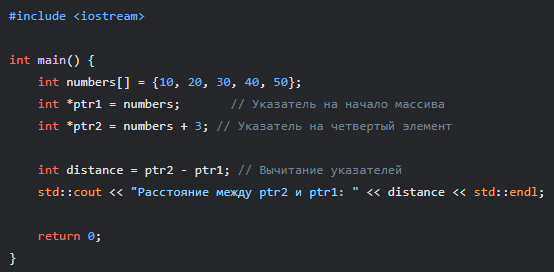
* ptr + 1 перемещает указатель на следующий элемент в массиве (не на один байт).

1. **Инкремент и декремент:**



* + ptr++: перемещает указатель на следующий элемент в массиве.
  + ptr--: перемещает указатель на предыдущий элемент в массиве.
  + Префиксный и постфиксный инкремент/декремент работают аналогично обычным целочисленным переменным.

1. **Вычитание указателей:**



* + ptr2 - ptr1: возвращает расстояние между указателями в *элементах*, а не байтах (в примере выше вернет 3).

**Важно:**

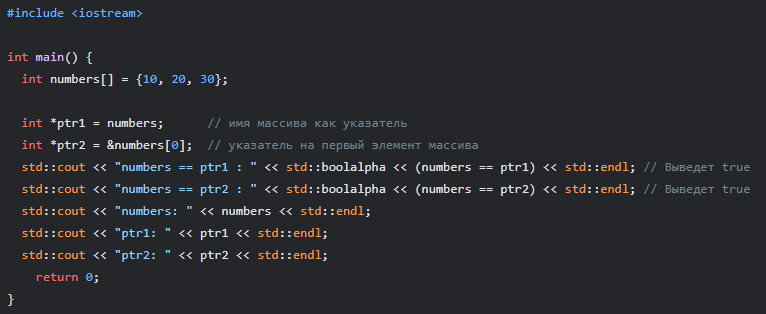
* При вычитании указателей они должны указывать на элементы одного и того же массива.

**22.3. Некоторые особенности арифметики указателей**

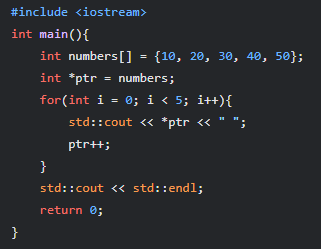
* **Выход за границы массива:** Арифметика указателей позволяет выйти за границы массива, но это может привести к непредсказуемым результатам и ошибкам. Выход за границы массива — это очень распространенная ошибка в C++.
* **Неопределенное поведение:** Результаты арифметики указателей с указателями, которые не указывают на элементы массива или на область памяти, выделенную динамически, приводят к неопределенному поведению.
* **Указатели на void**: Вы не можете применять арифметические операции к указателям типа void\*, т.к. компилятору неизвестен размер объекта, на который он указывает.

**22.4. Указатели и массивы**

Имя массива в C++ — это указатель на его первый элемент.



**22.5. Использование арифметики указателей для перебора массива**  
Арифметика указателей используется для доступа к элементам массива.



**22.6. Закрепление материала**

1. Что такое арифметика указателей?
2. Как влияет тип указателя на операции с ним?
3. Какие арифметические операции можно выполнять с указателями?
4. Что произойдет при выходе указателя за границы массива?
5. Что вернет вычитание двух указателей?
6. Почему имя массива - это указатель?

**Практические задания:**

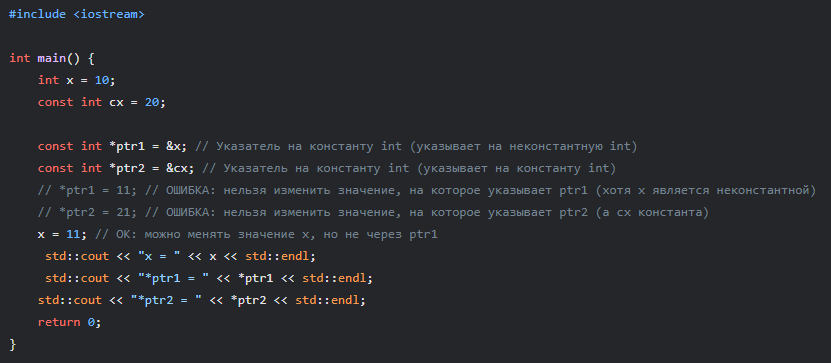
1. Объявите массив целых чисел и указатель на его начало.
2. Пройдитесь по массиву, используя арифметику указателей, и выведите его элементы на экран.
3. Напишите функцию, которая принимает массив целых чисел и его размер, и вычисляет сумму элементов массива, используя арифметику указателей.
4. Объявите два указателя на один массив и вычислите расстояние между ними.
5. Продемонстрируйте на примере выход указателя за границы массива.
6. Напишите программу, которая запрашивает у пользователя элементы массива и сохраняет их с использованием указателей.

**Глава 23: Константы и указатели в C++**

Константы и указатели могут взаимодействовать разными способами, и понимание этих взаимодействий важно для написания корректного кода.

**23.1. Указатель на константу (pointer to const)**

Указатель на константу — это указатель, который указывает на константное значение. Значение, на которое указывает такой указатель, нельзя изменить через этот указатель.

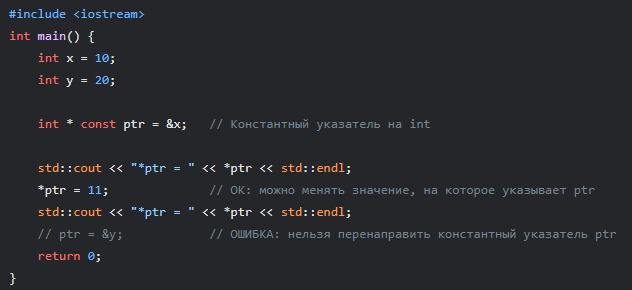


**Основные моменты:**

* const int \*ptr1: объявляет указатель ptr1, который указывает на константное целое число.
* Через указатель на константу нельзя менять значение, на которое он указывает.
* Указатель на константу можно использовать для указания как на константные, так и не константные переменные.

**23.2. Константный указатель (const pointer)**

Константный указатель — это указатель, который не может быть перенаправлен на другую область памяти. Но можно менять значение, на которое указывает константный указатель.

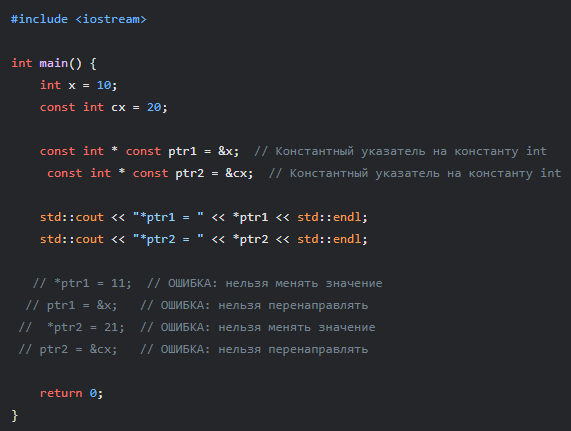


**Основные моменты:**

* int \* const ptr: объявляет ptr как константный указатель на целое число.
* Константный указатель всегда должен быть инициализирован при объявлении.
* Указатель ptr всегда будет указывать на одну и ту же область памяти (переменную x).

**23.3. Константный указатель на константу (const pointer to const)**

Константный указатель на константу — это указатель, который не может быть перенаправлен на другую область памяти, и через него нельзя изменить значение, на которое он указывает.

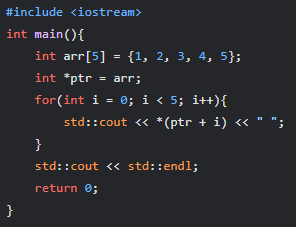


**Основные моменты:**

* const int \* const ptr: объявляет ptr как константный указатель на константное целое число.
* Константный указатель на константу должен быть проинициализирован при объявлении.
* Через такой указатель нельзя менять значение, на которое он указывает и его нельзя перенаправить на другую область памяти.

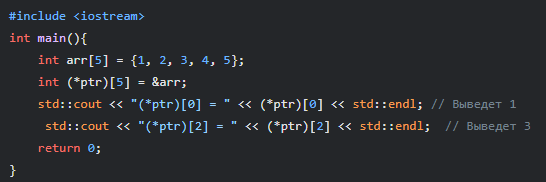
**23.4. Указатели и массивы**

Имя массива можно использовать как указатель на его первый элемент, поэтому с ним можно использовать операции с указателями:



* int \*ptr = arr; объявляет указатель ptr и указывает его на первый элемент массива.
* \*(ptr+i) получает доступ к элементу массива по индексу i.

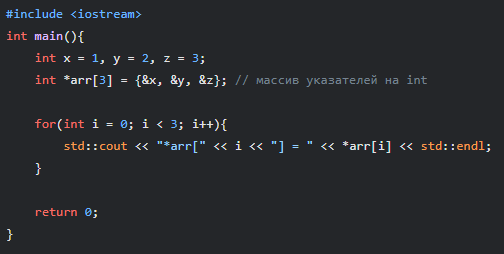
**23.5. Указатели на массивы**  
Можно объявить указатель на массив определенного размера.



* int (\*ptr)[5] = &arr; - ptr указатель на массив из 5 элементов типа int.

**23.6. Массивы указателей**

Можно объявлять массивы, элементами которых являются указатели:



* int \*arr[3]: объявляет массив arr, который может хранить 3 указателя на целые числа.

**23.7. Закрепление материала**

1. Что такое указатель на константу?
2. Что такое константный указатель?
3. Что такое константный указатель на константу?
4. Как использовать имя массива как указатель?
5. Можно ли создать указатель на массив?
6. Можно ли создать массив указателей?

**Практические задания:**

1. Объявите указатель на константное целое число и попробуйте изменить значение, на которое он указывает.
2. Объявите константный указатель и попробуйте перенаправить его на другую переменную.
3. Объявите константный указатель на константу и попробуйте изменить его значение и перенаправить его на другую переменную.
4. Создайте массив и переберите его элементы, используя указатель.
5. Создайте массив указателей и проинициализируйте его адресами целых переменных.
6. Напишите функцию, принимающую константный указатель на массив и выводящую его значения.

**Глава 24: Функции в C++**

Функция — это блок кода, выполняющий определенную задачу. Функции позволяют структурировать код, делать его более читаемым и переиспользуемым.

**24.1. Определение и объявление функций**

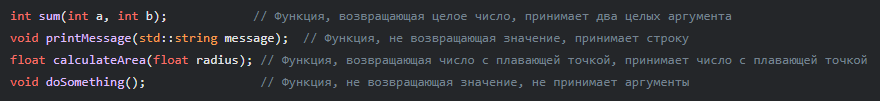
В C++ функция состоит из двух частей: объявления и определения.

1. **Объявление функции (Function Declaration):**  
   Объявление функции сообщает компилятору имя функции, типы ее параметров и тип возвращаемого значения. Объявление функции также называют прототипом функции.

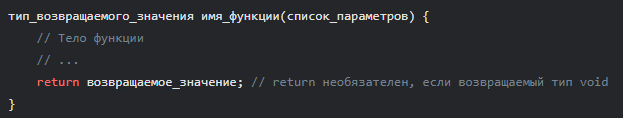


* тип\_возвращаемого\_значения: Тип данных, который функция возвращает (например, int, float, void). void означает, что функция не возвращает никакого значения.
* имя\_функции: Имя функции.
* список\_параметров: Список параметров функции, заключенный в круглые скобки, каждый параметр состоит из типа и имени (тип имя\_параметра). Параметры могут отсутствовать, если функция не принимает никаких аргументов.

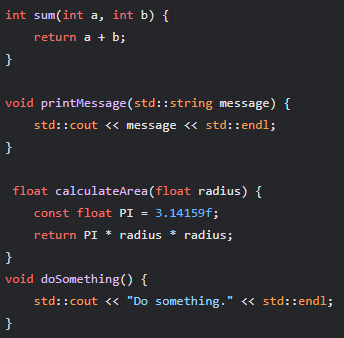
**Примеры:**



1. **Определение функции (Function Definition):**  
   Определение функции содержит код, который выполняется при вызове функции. Оно включает заголовок функции (объявление) и тело функции, заключенное в фигурные скобки.



**Примеры:**



**24.2. Вызов функции (Function Call)**  
Для выполнения кода функции необходимо ее вызвать. Вызов функции осуществляется по ее имени с указанием списка аргументов.

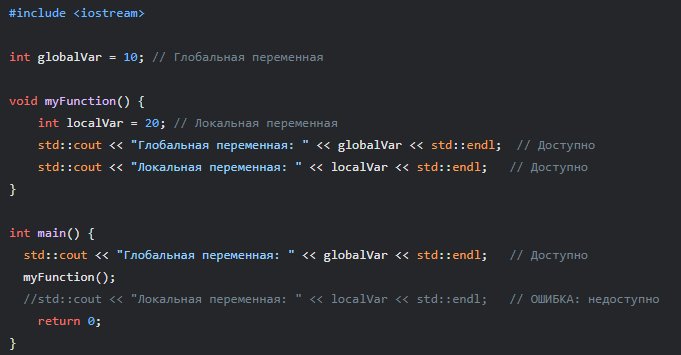


**24.3. Область видимости объектов (Scope)**

Область видимости объекта — это часть программы, в которой этот объект может быть доступен. В C++ есть следующие типы областей видимости:

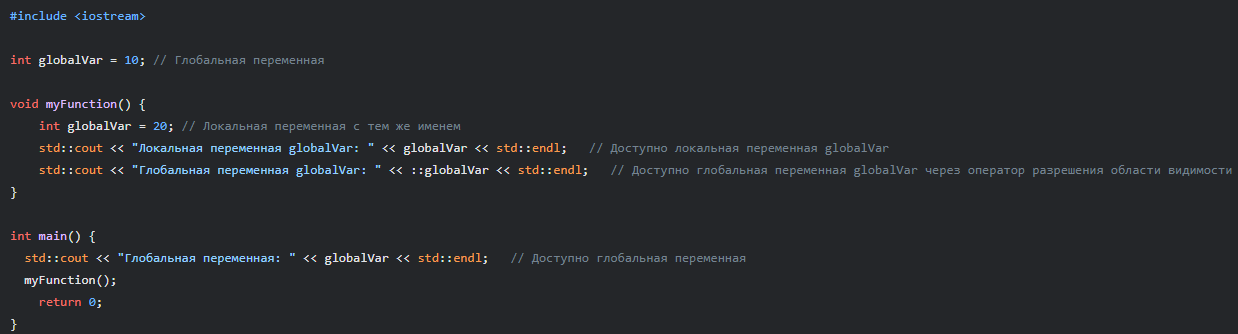
1. **Глобальная область видимости:** Объекты, объявленные вне всех функций, имеют глобальную область видимости и доступны во всей программе.
2. **Локальная область видимости:** Объекты, объявленные внутри блока кода (например, внутри функции), имеют локальную область видимости и доступны только внутри этого блока.

**Пример:**

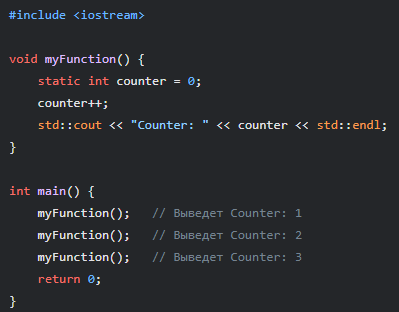


**24.4. Автоматические объекты**  
Объекты, созданные внутри функций (локальные переменные), являются автоматическими объектами (automatic object). Они автоматически создаются при входе в блок кода и уничтожаются при выходе из него.

**24.5. Скрытие объектов**  
Если в локальной области видимости объявляется объект с таким же именем, как и в глобальной, то глобальный объект будет скрыт внутри этой области видимости.



**24.6. Статические объекты**  
Локальные переменные могут быть объявлены как статические (static). Статические объекты инициализируются только один раз и сохраняют свое значение между вызовами функции.



* static int counter: counter будет инициализирована 0 только один раз, а затем будет сохранять свое значение между вызовами.

**24.7. Закрепление материала**

1. Что такое функция?
2. В чем разница между объявлением и определением функции?
3. Как вызвать функцию?
4. Что такое область видимости объекта?
5. Чем отличается глобальная область видимости от локальной?
6. Что такое автоматические объекты?
7. Как работает сокрытие объектов?
8. Что такое статические объекты?

**Практические задания:**

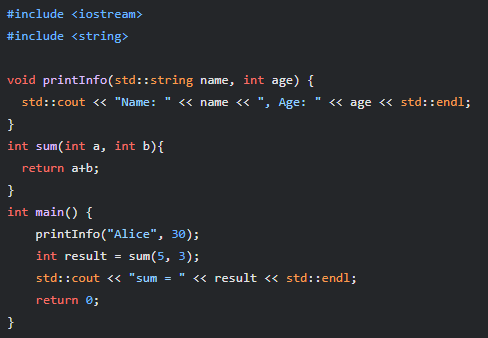
1. Напишите функцию, которая принимает два целых числа и возвращает их сумму.
2. Напишите функцию, которая принимает строку и выводит ее в обратном порядке.
3. Напишите функцию, которая принимает массив целых чисел и возвращает среднее арифметическое элементов массива.
4. Создайте глобальную переменную и используйте ее внутри и вне функции.
5. Создайте локальную переменную внутри функции и попробуйте получить к ней доступ вне функции.
6. Создайте статическую переменную внутри функции и проверьте, что она сохраняет свое значение между вызовами функции.

**Глава 25: Параметры функций и передача аргументов в C++**

Параметры функции — это переменные, которые используются для передачи данных в функцию. Аргументы — это фактические значения, которые передаются в функцию при ее вызове.

**25.1. Параметры функции**

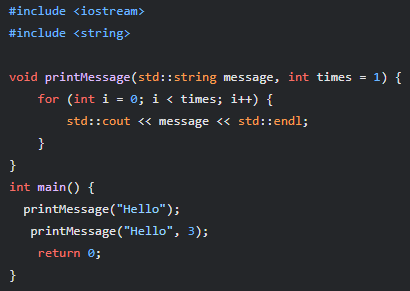
При объявлении функции можно указать список параметров, которые будут использоваться внутри тела функции. Каждый параметр состоит из типа и имени.



* std::string name, int age: это параметры функции printInfo.
* 5 и 3 это аргументы при вызове функции sum.

**25.2. Аргументы по умолчанию**

В C++ можно задавать аргументы по умолчанию для параметров функции. Если при вызове функции соответствующий аргумент не указан, будет использовано значение по умолчанию.



* int times = 1: задает значение по умолчанию для параметра times равное 1.

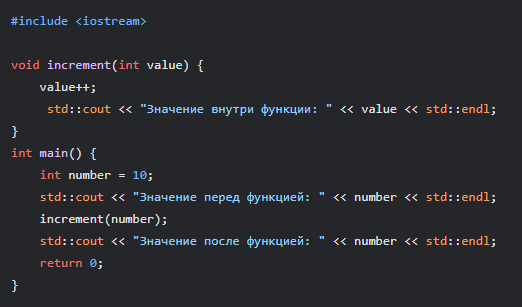
**Основные моменты:**

* Аргументы по умолчанию должны идти после параметров без значений по умолчанию.
* Нельзя задавать значения по умолчанию при объявлении, если объявление функции и определение находятся в разных файлах (об этом будет позже).

**25.3. Автоматическое выведение типа параметров (auto)**

Начиная с C++11, можно использовать auto для автоматического выведения типа параметра в лямбда-функциях (мы их будем изучать позже). В случае обычных функций автоматическое выведение типа параметров невозможно.

**25.4. Передача аргументов по значению**  
При передаче аргументов по значению (pass-by-value) в функцию передается *копия* значения. Изменение копии внутри функции не влияет на оригинал.

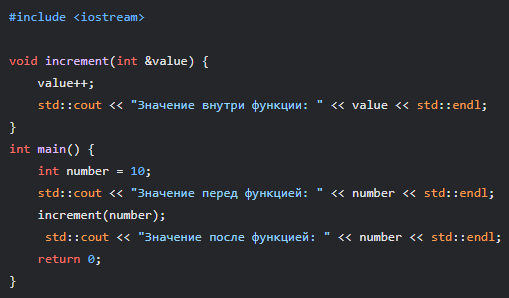


**Основные моменты:**

* Когда мы вызываем функцию increment(number) создается *копия* переменной number в качестве параметра value
* Изменение value не влияет на переменную number.

**25.5. Передача аргументов по ссылке (pass-by-reference)**

При передаче аргументов по ссылке (pass-by-reference) в функцию передается *ссылка* на исходную переменную. Изменение значения внутри функции *влияет* на значение исходной переменной.

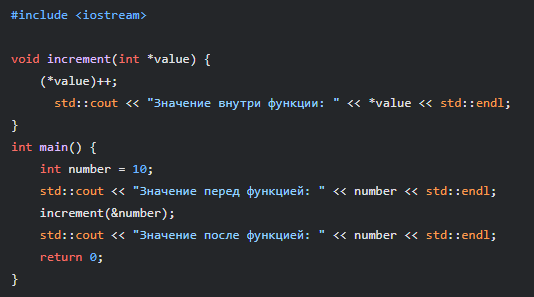


**Основные моменты:**

* int &value: объявляет параметр value как *ссылку* на целое число.
* Изменения переменной value внутри функции повлияют на исходную переменную number.
* При передаче по ссылке не происходит копирования, функция работает непосредственно с переменной, переданной в качестве аргумента.

**25.6. Передача аргументов по указателю (pass-by-pointer)**

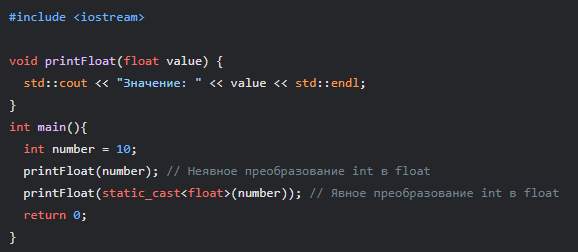
Передача по указателю работает аналогично передаче по ссылке, но вместо ссылки используется указатель.



* int \*value: объявляет параметр value как указатель на целое число.
* В функцию передается адрес переменной number, через который происходит ее изменение.
* Следует избегать использования указателей, если можно использовать ссылки.

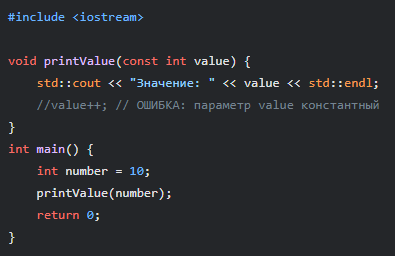
**25.7. Преобразования типов**

При передаче аргументов в функцию, компилятор может выполнить неявные преобразования типов. Однако, явное преобразование типов с использованием static\_cast может быть более безопасным.

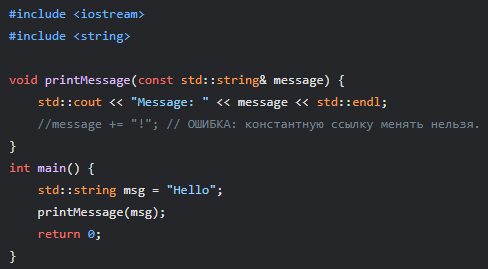


**25.8. Константные параметры**

Можно объявить параметры функции как константные, если функция не должна изменять их значение.



**25.9. Константные ссылки**  
Можно объявить параметры как константные ссылки (const &), если функция не должна изменять переданный по ссылке параметр.



**Основные моменты:**

* Константная ссылка позволяет избежать копирования и не дает менять переменную.
* Константные ссылки можно использовать как для константных, так и для неконстантных параметров.
* Передача аргументов по константной ссылке является предпочтительным способом передачи объектов, которые не должны быть изменены функцией.

**25.10. Закрепление материала**

1. Что такое параметры функции?
2. Что такое аргументы по умолчанию и как их использовать?
3. Как работает передача аргументов по значению?
4. Как работает передача аргументов по ссылке?
5. В чем отличие передачи по ссылке от передачи по указателю?
6. Что происходит при передаче аргументов с преобразованием типов?
7. Что такое константные параметры и когда их нужно использовать?
8. Что такое константные ссылки и когда их нужно использовать?

**Практические задания:**

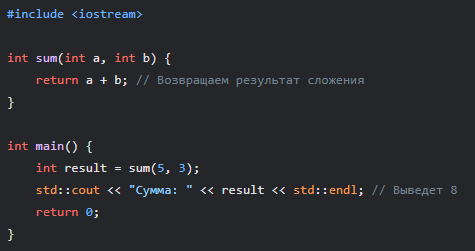
1. Напишите функцию, которая принимает два числа и возвращает их сумму.
2. Напишите функцию, которая принимает строку и количество раз, которое ее нужно вывести на экран (используйте параметр по умолчанию).
3. Напишите функцию, которая меняет значения двух целых чисел, переданных по ссылке.
4. Напишите функцию, которая принимает массив целых чисел и удваивает каждый элемент, используя передачу по ссылке.
5. Напишите функцию, которая принимает строку и возвращает ее длину, используя константную ссылку.
6. Напишите функцию, которая принимает число с плавающей точкой и выводит его на экран, используя константный параметр.

**Глава 26: Оператор return и возвращение результата в C++**

Оператор return используется для завершения выполнения функции и возвращения значения вызывающей стороне.

**26.1. Оператор return и возвращение значения**

Оператор return используется для возвращения значения из функции.

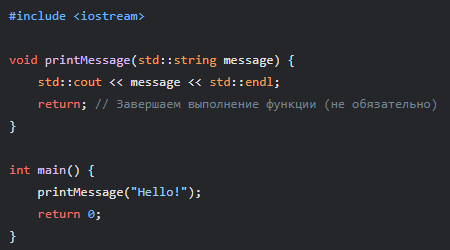


**Основные моменты:**

* Тип возвращаемого значения функции должен соответствовать типу выражения в операторе return.
* При выполнении return выполнение функции завершается и передается управление в место вызова функции.

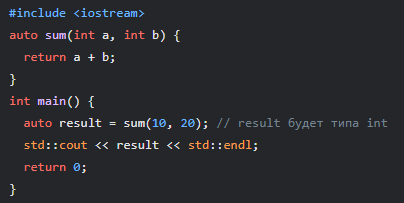
**26.2. return без возвращения значения**

Если функция имеет тип возвращаемого значения void, то оператор return можно использовать без указания возвращаемого значения. В этом случае он просто завершает выполнение функции.



**26.3. Выведение типа результата (auto)**

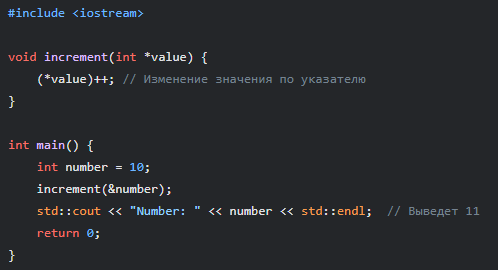
Начиная с C++11, можно использовать ключевое слово auto для автоматического выведения типа возвращаемого значения функции. Для этого тип возвращаемого значения нужно заменить на auto.



* Тип результата будет выведен по типу возвращаемого значения в операторе return.
* При этом функция должна иметь один и тот же тип результата во всех return операторах.
* Не рекомендуется использовать auto для возвращаемого типа, кроме случаев, когда тип возвращаемого значения сложный и его не удобно записывать явно.

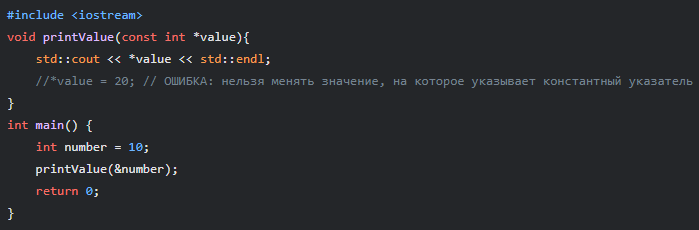
**26.4. Указатели в параметрах функции**

Указатели могут использоваться в качестве параметров функции, чтобы функция могла изменять значения переменных, переданных в качестве аргументов.



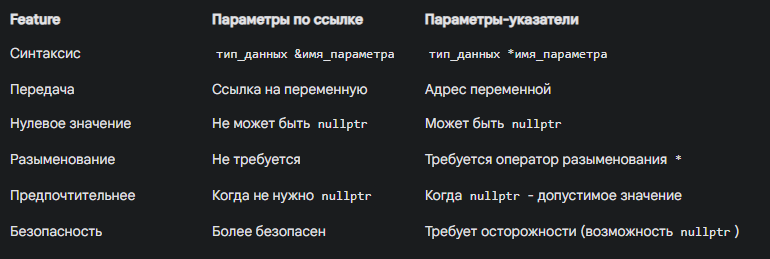
**26.5. Константные параметры-указатели**

Параметры-указатели могут быть объявлены как константные, что гарантирует, что функция не будет менять значение, на которое указывает указатель.



**26.6. Параметры по ссылке или параметры-указатели**

Как передавать аргументы в функцию — по ссылке или по указателю — это выбор программиста, который зависит от конкретной ситуации.



**Пример:**



**Основные моменты:**

* Используйте передачу по ссылке, если не нужно использовать нулевой указатель и вы хотите, чтобы функция изменяла значение переменной.
* Используйте передачу по указателю, если функция должна работать с массивами, динамической памятью или nullptr является допустимым значением.

**26.7. Закрепление материала**

1. Для чего используется оператор return?
2. Как вернуть значение из функции?
3. Что произойдет, если функция имеет тип возвращаемого значения void, и как использовать return в этом случае?
4. Можно ли вывести тип результата, используя auto?
5. Как использовать указатели в качестве параметров функций?
6. Что такое константные параметры-указатели и когда их нужно использовать?
7. В чем разница между передачей параметров по ссылке и по указателю?

**Практические задания:**

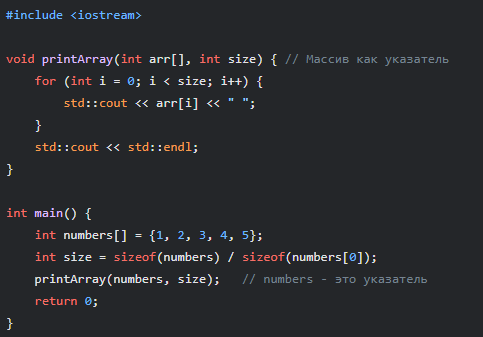
1. Напишите функцию, которая принимает два числа с плавающей точкой и возвращает их среднее арифметическое.
2. Напишите функцию, которая принимает строку и возвращает ее длину.
3. Напишите функцию, которая принимает целое число и возвращает его факториал (рекурсивно).
4. Напишите функцию, которая принимает указатель на целое число и устанавливает его значение равным 100.
5. Создайте константный указатель на число с плавающей точкой и передайте его в функцию.
6. Напишите функцию, которая меняет местами значения двух переменных, переданных по ссылке.
7. Напишите функцию, которая принимает указатель на массив и размер массива и выводит на экран все его элементы.

**Глава 27: Массивы в параметрах функции в C++**

В этой главе мы рассмотрим, как передавать массивы в функции, какие есть ограничения при передаче массивов, как передавать маркер конца массива, как использовать константные массивы в параметрах, как передавать массивы по ссылке и как передавать многомерные массивы.

**27.1. Массивы в параметрах функции (ограничения)**

В C++, когда массив передается в функцию как параметр, он автоматически преобразуется в *указатель* на свой первый элемент. Это означает, что функция не получает копию массива, а работает с оригинальным массивом по адресу.

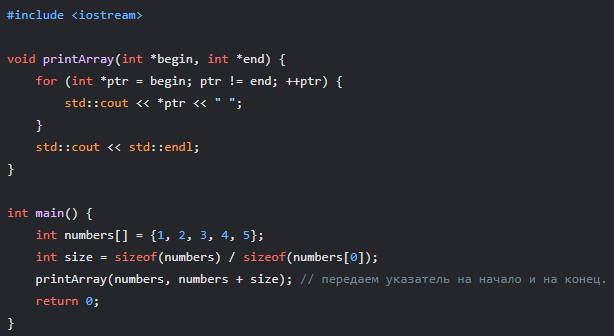


**Основные моменты:**

* int arr[]: на самом деле означает int \*arr.
* Передается указатель на первый элемент массива, а *не копия* массива.
* Функция не знает размер массива (поэтому его передают вторым аргументом), если не было передано дополнительно.
* Таким образом, внутри функции printArray мы работаем с указателем на int, а не массивом.

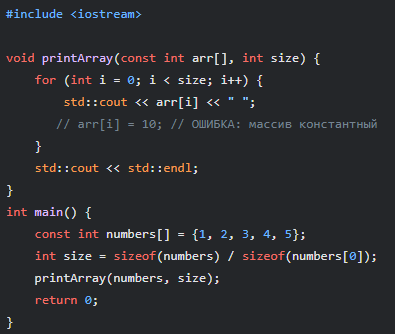
**27.2. Передача маркера конца массива**

Так как массив автоматически преобразуется в указатель на первый элемент, и функция не знает размер массива, нам нужно передать размер массива явно вторым аргументом.  
Однако можно передавать указатель на конец массива, что иногда удобнее.



**27.3. Константные массивы**

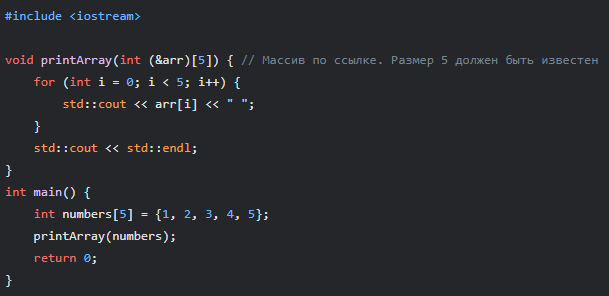
Если нужно защитить массив от изменения внутри функции, то можно использовать const.



**Основные моменты:**

* const int arr[]: объявляет параметр arr как указатель на константное целое число.
* Невозможно изменить элементы массива внутри функции.

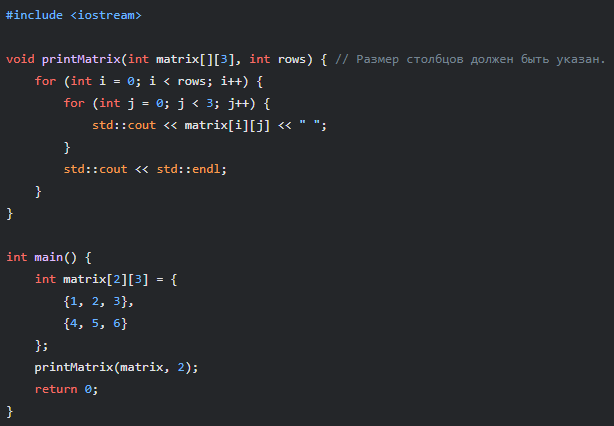
**27.4. Передача массива по ссылке**  
Можно передать массив по ссылке, но при этом размер массива должен быть известен на этапе компиляции.



* int (&arr)[5]: объявляет параметр arr как *ссылку* на массив из 5 элементов типа int.
* При передаче по ссылке, функция работает непосредственно с массивом, переданным в качестве аргумента.
* Но данный способ имеет ограничение — размер массива должен быть известен на этапе компиляции.

**27.5. Передача многомерного массива**

При передаче многомерного массива в функцию нужно указать размер всех измерений, кроме первого:

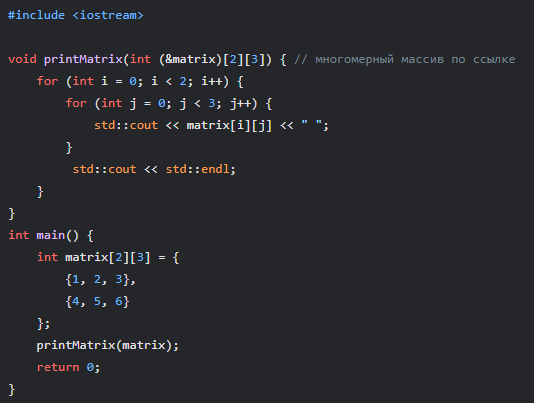


**Основные моменты:**

* Функция printMatrix принимает двумерный массив matrix и количество строк rows.
* Размер столбцов (3) должен быть указан в параметре функции.
* Передается указатель на массив int[3].
* Если мы не укажем количество столбцов, то компилятор не сможет посчитать размер.

**27.6. Передача многомерного массива по ссылке**

Также можно передавать многомерный массив по ссылке, но при этом должны быть известны все размеры.



* int (&matrix)[2][3]: объявляет matrix как ссылку на двумерный массив.
* Размерности массива должны быть известны при компиляции.
* Функция работает непосредственно с массивом переданным в качестве аргумента.

**27.7. Закрепление материала**

1. Что происходит, когда массив передается в функцию как параметр?
2. Почему функция не знает размер массива, если размер не передан вторым аргументом?
3. Как передать маркер конца массива в функцию?
4. Как сделать массив константным в параметрах функции?
5. Как передать массив по ссылке?
6. Какие ограничения существуют при передаче многомерных массивов?
7. Как передать многомерный массив по ссылке?

**Практические задания:**

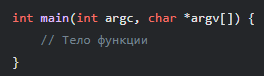
1. Напишите функцию, которая принимает массив целых чисел и его размер и выводит на экран все его элементы.
2. Напишите функцию, которая принимает массив целых чисел и его размер, и удваивает каждый элемент массива (передача по указателю).
3. Напишите функцию, которая принимает массив символов и выводит его в обратном порядке.
4. Напишите функцию, которая принимает двумерный массив и выводит его на экран в виде таблицы.
5. Напишите функцию, которая принимает двумерный массив и его размеры, и вычисляет сумму всех элементов массива.
6. Напишите функцию, которая транспонирует двумерный массив, переданный по ссылке.

**Глава 28: Параметры функции main в C++**

Функция main — это точка входа в программу, и она также может принимать параметры, которые позволяют передавать информацию в программу при ее запуске.

**28.1. Параметры функции main**

Функция main может быть объявлена с двумя параметрами:



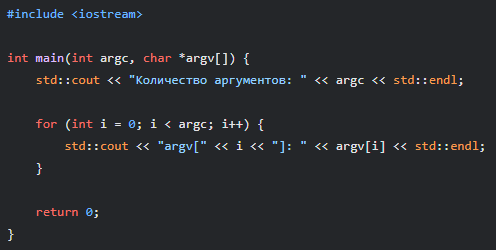
* int argc (argument count): Целое число, представляющее количество аргументов командной строки, включая имя самой программы.
* char \*argv[] (argument vector): Массив указателей на C-строки (массивы символов), представляющие аргументы командной строки.

**28.2. Как это работает**

Когда программа запускается из командной строки, операционная система передает ей аргументы в виде строки. C++ обрабатывает эту строку, разделяя ее на отдельные аргументы и передавая их в функцию main:

* argc содержит общее количество аргументов, включая имя программы.
* argv — это массив строк, где:
  + argv[0] — это имя самой программы (путь к исполняемому файлу).
  + argv[1] — это первый аргумент командной строки.
  + argv[2] — это второй аргумент командной строки.
  + ... и так далее.
  + argv[argc] всегда будет nullptr (нулевой указатель) (не входит в количество argc).

**28.3. Пример использования**



**Компиляция и запуск:**

1. Сохраните этот код в файле с именем main.cpp.
2. Скомпилируйте программу с помощью компилятора C++ (например, g++):



1. Запустите программу из командной строки с различными аргументами:

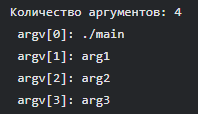


Вывод:

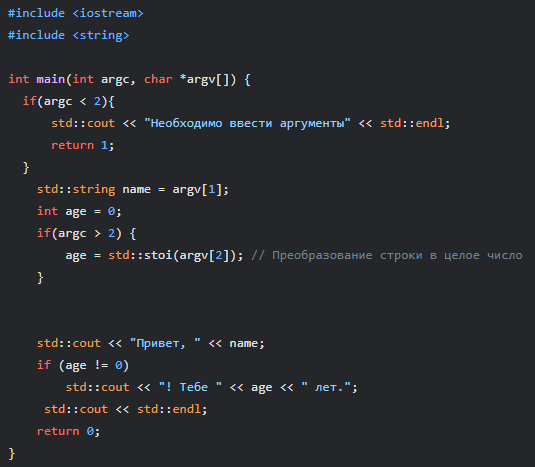




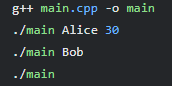
Вывод:



**28.4. Пример обработки аргументов**



**Компиляция и запуск:**



**Основные моменты:**

* argc всегда будет не меньше 1, так как как минимум имя программы будет передано в качестве первого аргумента.
* В примере выше используется std::stoi для преобразования строки в целое число.

**28.5. Закрепление материала**

1. Какие параметры может принимать функция main?
2. Что такое argc и argv?
3. Какой тип данных имеют argc и argv?
4. Как получить доступ к аргументам, переданным в программу из командной строки?
5. Какое значение будет содержать argv[0]?
6. Как проверить количество переданных аргументов?

**Практические задания:**

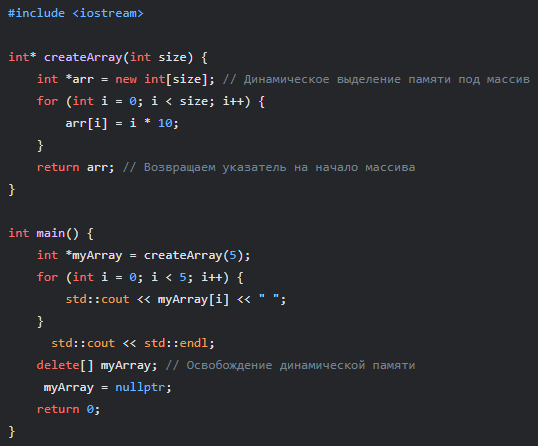
1. Напишите программу, которая выводит на экран все аргументы командной строки, переданные ей при запуске.
2. Напишите программу, которая принимает имя пользователя в качестве аргумента командной строки и выводит приветствие с этим именем.
3. Напишите программу, которая принимает два числа в качестве аргументов командной строки и выводит их сумму.

**Глава 29: Возвращение указателей и ссылок, перегрузка функций в C++**

В этой главе мы разберем, как функции могут возвращать указатели и ссылки, а также изучим, что такое перегрузка функций, и как она работает с параметрами по ссылке и константами.

**29.1. Возвращение указателя**

Функция может возвращать указатель, который указывает на область памяти.



**Основные моменты:**

* Функция createArray возвращает указатель int\*, который указывает на динамически выделенную память под массив.
* Необходимо помнить, что память, выделенную оператором new, необходимо освободить через delete[].
* Необходимо устанавливать указатель в nullptr после освобождения памяти.
* Использование nullptr позволяет убедиться, что вы не пытаетесь работать с уже освобожденной памятью.

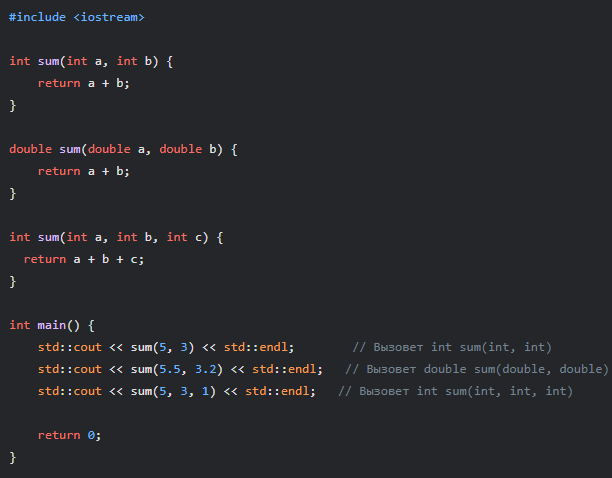
**29.2. Возвращение ссылки**  
Функция может возвращать ссылку на переменную, как на локальную, так и на глобальную.



* int& getNumber(int& x): возвращает ссылку на переменную типа int.
* Изменение возвращаемой ссылки, приводит к изменению исходной переменной.
* Следует избегать возвращения ссылок на локальные переменные, созданные внутри функции.

**29.3. Перегрузка функций**

Перегрузка функций — это возможность объявить несколько функций с одним и тем же именем, но с разными параметрами. При вызове функции компилятор выбирает наиболее подходящую версию на основе типа и количества аргументов.

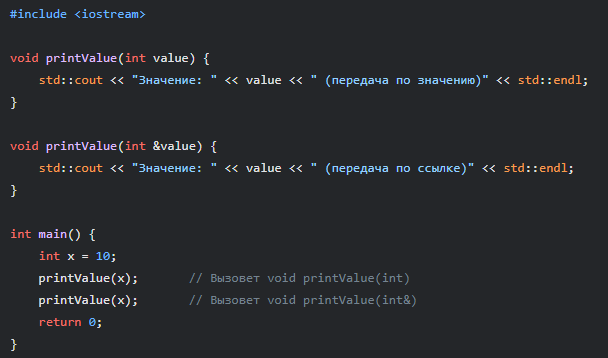


**Основные моменты:**

* Функции с одинаковым именем должны иметь разный набор параметров (разное количество, разные типы).
* Тип возвращаемого значения не является частью сигнатуры функции и не влияет на перегрузку.

**29.4. Перегрузка функций и параметры-ссылки**

Перегрузку функций можно использовать с параметрами по ссылке:

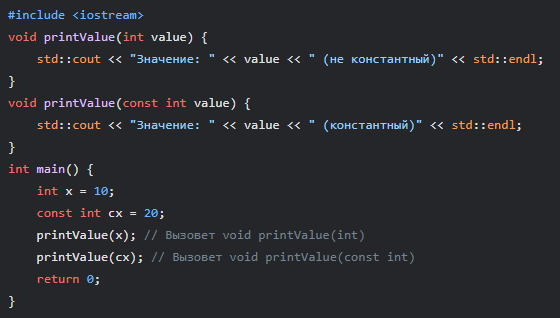


**Основные моменты:**

* Передача по значению и по ссылке это разные сигнатуры.
* При вызове printValue(x) вызывается void printValue(int) т.к. передается переменная, а не ссылка.
* При вызове printValue(x) вызывается void printValue(int&), т.к. передается ссылка на переменную.

**29.5. Перегрузка и параметры-константы**

Перегрузку функций можно использовать с константными параметрами.



* void printValue(int value) и void printValue(const int value) - это разные функции (разные сигнатуры).
* Компилятор выбирает более подходящую версию на основе константности аргумента.

**29.6. Закрепление материала**

1. Как функция может возвращать указатель?
2. Что нужно помнить при возвращении указателя из функции?
3. Как функция может возвращать ссылку?
4. Когда не следует возвращать ссылку из функции?
5. Что такое перегрузка функций?
6. Как перегрузка функций работает с параметрами по ссылке?
7. Как перегрузка функций работает с константными параметрами?

**Практические задания:**

1. Напишите функцию, которая принимает размер массива и возвращает указатель на динамически созданный массив целых чисел.
2. Напишите функцию, которая принимает массив целых чисел и индекс, и возвращает ссылку на элемент массива по этому индексу.
3. Напишите перегруженные функции sum для суммирования целых и чисел с плавающей точкой (два и три параметра).
4. Напишите функцию, принимающую параметр по значению, и перегрузите ее, чтобы принимать параметр по ссылке.
5. Напишите функцию, принимающую параметр и перегрузите ее, чтобы принимать константный параметр.

**Глава 30: Рекурсивные функции в C++**

Рекурсивная функция — это функция, которая вызывает саму себя в своем теле. Рекурсия позволяет элегантно решать задачи, которые можно разбить на более мелкие подзадачи того же типа.

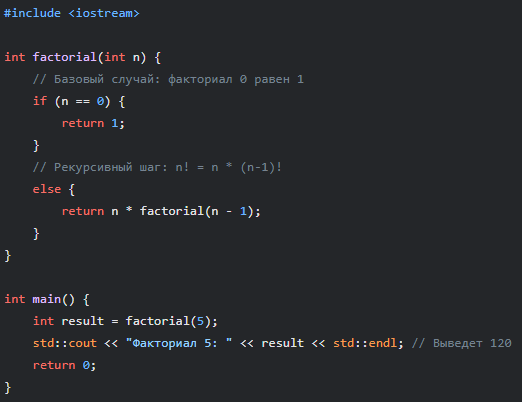
**30.1. Основы рекурсии**

Для того, чтобы рекурсия работала корректно, нужно соблюдать следующие правила:

1. **Базовый случай (Base Case):**  
   Рекурсивная функция должна иметь хотя бы один базовый случай, когда она не вызывает себя, а возвращает какое-либо значение. Это условие, при котором рекурсия заканчивается, чтобы избежать бесконечного вызова функции.
2. **Рекурсивный шаг (Recursive Step):**  
   Рекурсивный шаг — это вызов функции самой себя с более простыми аргументами, которые приближают решение к базовому случаю.

**30.2. Пример рекурсивной функции: факториал**

Вот пример рекурсивной функции для вычисления факториала числа:

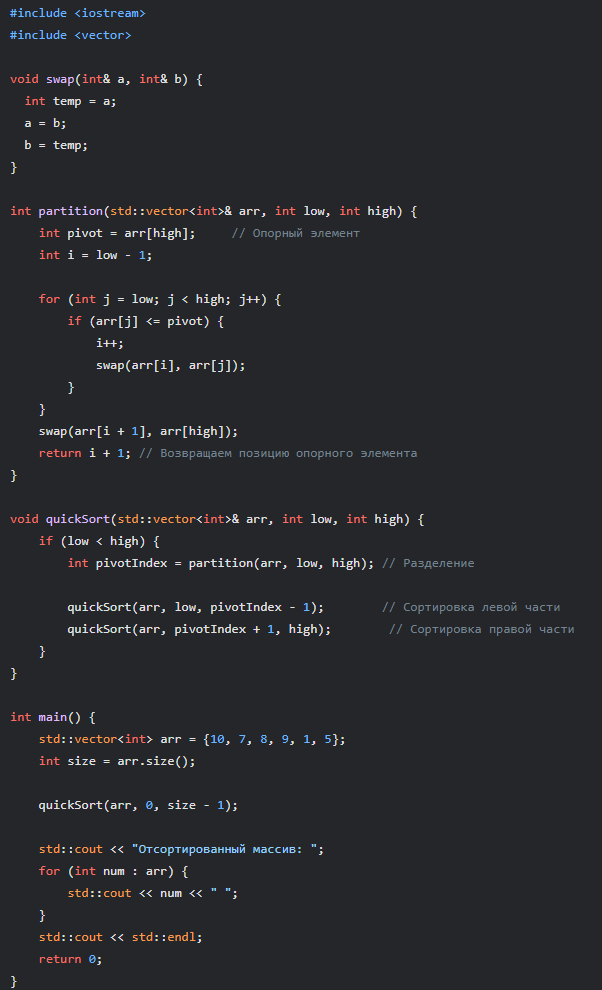


**Разбор примера:**

* **Базовый случай:** если n равно 0, то функция возвращает 1.
* **Рекурсивный шаг:** если n не равно 0, то функция вызывает сама себя, передавая в качестве аргумента n-1, и умножает результат на n.
* При вызове factorial(5) происходит следующее:
  + factorial(5) = 5 \* factorial(4)
  + factorial(4) = 4 \* factorial(3)
  + factorial(3) = 3 \* factorial(2)
  + factorial(2) = 2 \* factorial(1)
  + factorial(1) = 1 \* factorial(0)
  + factorial(0) = 1 (базовый случай)
  + Результаты "возвращаются" обратно, вычисляя итоговый результат: 5 \* 4 \* 3 \* 2 \* 1 \* 1 = 120.

**30.3. Рекурсия на примере быстрой сортировки (Quicksort)**

Быстрая сортировка — это эффективный алгоритм сортировки, который часто реализуется с использованием рекурсии.



**Разбор примера:**

* **partition:** Функция выбирает последний элемент массива как опорный, переставляет элементы в массиве, чтобы все элементы меньше опорного оказались слева от него, а большие справа, и возвращает индекс опорного элемента.
* **quickSort:** Функция:
  + **Базовый случай:** если low (начальный индекс) больше или равен high (конечный индекс) (массив из 0 или 1 элемента), то выходим из рекурсии.
  + **Рекурсивный шаг:**
    - Вызывает partition, чтобы разделить массив на две части, где элементы слева от опорного, меньше, чем опорный, а справа - больше.
    - Вызывает quickSort для левой части массива.
    - Вызывает quickSort для правой части массива.

**Основные моменты:**

* Быстрая сортировка рекурсивно разделяет массив на подмассивы, пока размер подмассива не станет равен 0 или 1.
* Рекурсия позволяет элегантно выразить алгоритм сортировки.

**30.4. Особенности рекурсивных функций**

* **Глубина рекурсии:** Каждое рекурсивное обращение функции создает новый кадр стека. Если глубина рекурсии слишком большая (например, отсутствие базового случая), то программа может переполнить стек и завершиться с ошибкой.
* **Производительность:** Рекурсивные функции могут быть менее эффективными по производительности, чем итеративные (на основе циклов), из-за накладных расходов на вызовы функций.
* **Читаемость:** Рекурсивный код часто более компактный и элегантный, чем итеративный, особенно для задач, которые имеют рекурсивную структуру.

**Практические задания:**

1. Напишите рекурсивную функцию для вычисления суммы чисел от 1 до n.
2. Напишите рекурсивную функцию для вычисления n-го числа Фибоначчи.
3. Напишите рекурсивную функцию для вывода элементов массива в обратном порядке.
4. Реализуйте алгоритм бинарного поиска с использованием рекурсии.
5. Разберитесь как работает алгоритм быстрой сортировки, и попробуйте переписать его с циклами, вместо рекурсии.

**Глава 31: Указатели на функции в C++**

Указатель на функцию — это переменная, которая хранит адрес памяти, где находится исполняемый код функции. Указатели на функции позволяют нам обрабатывать функции как данные, что открывает новые возможности для гибкого и динамического программирования.

**31.1. Определение и инициализация указателя на функцию**

Для объявления указателя на функцию нужно указать тип возвращаемого значения и типы параметров, которые принимает функция, на которую будет указывать этот указатель:



* тип\_возвращаемого\_значения: Тип возвращаемого значения функции.
* \*имя\_указателя: Имя указателя (символ \* обязательно заключить в скобки).
* список\_типов\_параметров: Типы параметров функции.

**Пример:**

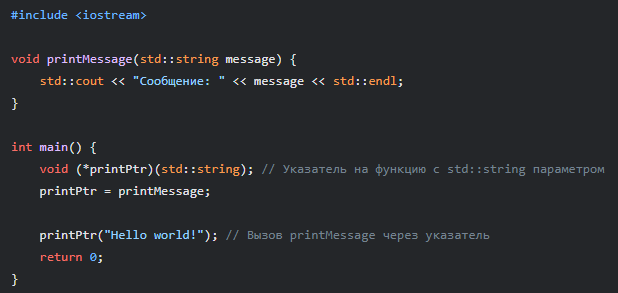


**Основные моменты:**

* int (\*operation)(int, int): объявляет указатель operation, который может указывать на любую функцию, возвращающую int и принимающую два аргумента типа int.
* operation = add: присваивает указателю operation адрес функции add.
* operation(10, 5): вызывает функцию, на которую указывает operation.

**31.2. Указатель на функцию с параметрами**

Указатель на функцию может указывать на функции, принимающие любые типы параметров, главное, чтобы сигнатура функции (типы параметров и тип возвращаемого значения) совпадала с сигнатурой указателя.



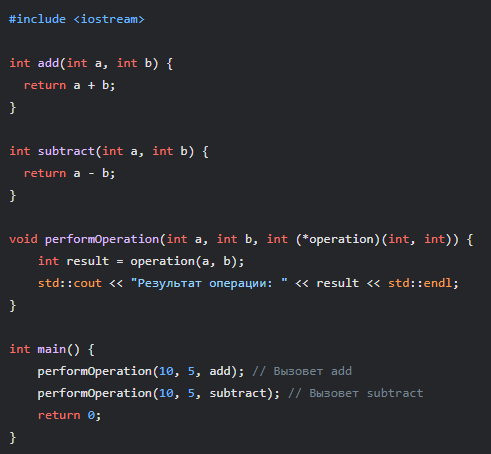
**31.3. Массивы указателей на функции**

Можно создавать массивы, элементами которых являются указатели на функции.



**31.4. Указатели на функции как параметры**

Указатели на функции можно передавать в качестве параметров другим функциям, что позволяет создавать гибкие и настраиваемые функции.



**Основные моменты:**

* void performOperation(int a, int b, int (\*operation)(int, int)): функция performOperation принимает в качестве параметра указатель на функцию.

**Практические задания:**

1. Объявите указатель на функцию, которая принимает два целых числа и возвращает их сумму.
2. Создайте массив указателей на функции, каждая из которых выполняет одну из простых арифметических операций (сложение, вычитание, умножение, деление).
3. Напишите функцию, которая принимает указатель на другую функцию, и выполняет ее.
4. Напишите программу, которая в зависимости от выбора пользователя вызывает соответствующую арифметическую функцию через указатель.
5. Создайте структуру, содержащую указатель на функцию, и используйте этот указатель для вызова функции.

**Глава 32: Тип функции, возвращение указателей на функции в C++**

В этой главе мы разберем, что такое тип функции, как функции могут возвращать указатели на другие функции, и какие особенности при этом следует учитывать.

**32.1. Тип функции**

Тип функции определяется типом ее возвращаемого значения и типами ее параметров. Например:

* int(int, int) — это тип функции, принимающей два параметра типа int и возвращающей значение типа int.
* void(std::string) — это тип функции, принимающей параметр типа std::string и не возвращающей никакого значения (void).
* double(double) — это тип функции, принимающей параметр типа double и возвращающей значение типа double.

Тип функции используется при объявлении указателей на функцию, чтобы гарантировать правильность вызова и совместимость.

**32.2. Указатель на функцию как возвращаемое значение**

Функция может возвращать указатель на другую функцию. Для этого нужно указать тип возвращаемого значения как указатель на функцию:



* тип\_возвращаемого\_значения - тип возвращаемого значения функции, на которую будет указывать указатель.
* \*имя\_функции(параметры) - имя функции и ее параметры.
* список\_параметров - список типов параметров функции, на которую будет указывать возвращаемый указатель.

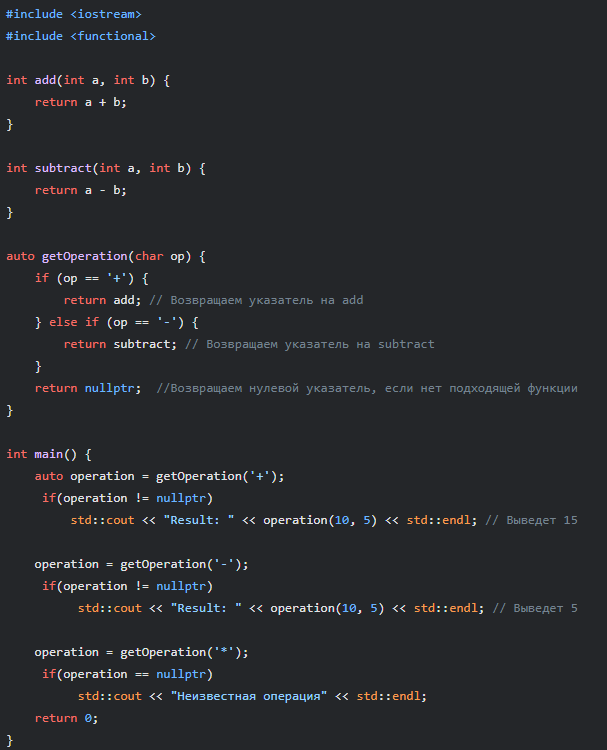


**Основные моменты:**

* std::function<int(int, int)>: позволяет объявлять указатель на функцию с любым типом, соответствующим int(int, int). (Для использования нужно подключить #include <functional>).
* getOperation возвращает указатель на функцию в зависимости от значения аргумента.
* Указатель на функцию может быть использован для вызова, если он не нулевой.
* Функция должна возвращать nullptr, если нет подходящей функции для возврата.

**29.2. Возвращение указателя на функцию с использованием auto**

Начиная с C++11, можно использовать ключевое слово auto для автоматического выведения типа возвращаемого значения, что сделает код более читаемым.



* auto getOperation(char op): компилятор сам выведет тип возвращаемого значения.
* Это делает код более кратким, но нужно следить за корректностью возвращаемого типа.

**29.3. Особенности возвращения указателей на функции:**

* **Время жизни:** Важно помнить о времени жизни объектов, на которые указывают возвращаемые указатели. Если функция возвращает указатель на локальную переменную, то после завершения функции этот указатель станет недействительным.
* **Динамическая память:** Возвращаемый указатель, часто, указывает на динамически выделенную память, и эту память необходимо освободить при помощи delete или delete[] (когда возвращается указатель на массив) после использования.
* **Использование std::function:** std::function является более безопасным и гибким способом работы с указателями на функции в C++.
* **Опциональное возвращение:** Если функция не всегда может вернуть указатель на функцию, то нужно возвращать nullptr и обрабатывать этот случай в вызывающей стороне.

**Практические задания:**

1. Напишите функцию, которая возвращает указатель на одну из простых арифметических функций (сложение, вычитание), в зависимости от переданного символа.
2. Напишите функцию, которая принимает массив целых чисел и возвращает указатель на функцию, которая ищет максимальное значение в массиве.
3. Напишите функцию, которая возвращает указатель на функцию, которая печатает переданную строку с определенным форматом.
4. Используйте auto для определения типа возвращаемого значения функции, возвращающей указатель на функцию.
5. Продемонстрируйте на примере использование std::function.

**Глава 33: Разделение программы на файлы в C++**

Разделение программы на несколько файлов — это хорошая практика, которая позволяет нам организовать код, сделать его более модульным и переиспользуемым.

**33.1. Зачем нужно разделять программу на файлы?**

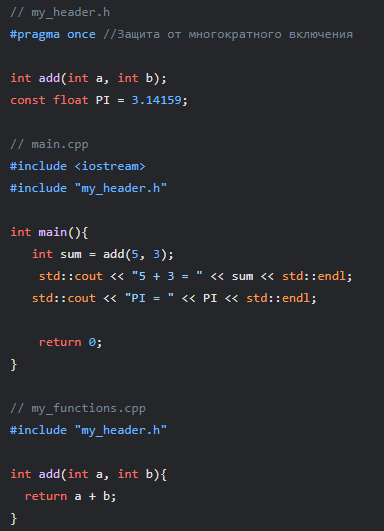
* **Улучшение организации кода:** Разбиение кода на файлы по функциональности делает код более структурированным и легким для понимания.
* **Переиспользование кода:** Код, выделенный в отдельные файлы, можно повторно использовать в других проектах.
* **Упрощение совместной работы:** Когда над проектом работает несколько человек, разделение кода на файлы позволяет им работать параллельно над разными частями проекта, не мешая друг другу.
* **Ускорение компиляции:** При изменении кода в одном файле компилятору нужно перекомпилировать только этот файл, а не весь проект.

**33.2. Создание нескольких файлов**

Обычно проект на C++ разделяется на следующие виды файлов:

1. **Заголовочные файлы (.h или .hpp):**
   * Содержат объявления функций, классов, структур, констант и других объектов.
   * Не содержат реализацию (тела) функций (за исключением шаблонных функций).
   * Используются для того, чтобы код в других файлах мог узнать о существующих функциях и структурах.
2. **Файлы с реализацией (.cpp):**
   * Содержат реализацию функций, которые были объявлены в заголовочных файлах.
   * Содержат исполняемый код программы.

**33.3. Использование #include для подключения заголовочных файлов**  
Для того чтобы использовать объекты, объявленные в заголовочных файлах, в исходных файлах необходимо использовать директиву препроцессора #include.



**Основные моменты:**

* #include <iostream>: подключает системные заголовочные файлы из стандартной библиотеки C++. Угловые скобки указывают компилятору искать файл в стандартных директориях.
* #include "my\_header.h": подключает заголовочный файл, расположенный в текущей директории. Кавычки указывают компилятору искать файл в текущей директории.
* #pragma once: директива, которая гарантирует, что заголовочный файл будет включен в исходный файл только один раз, даже если он подключается несколько раз. Это позволяет избежать ошибок.

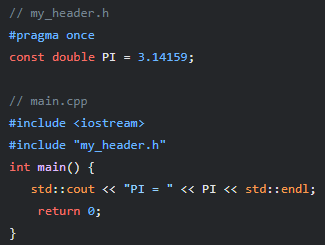
**33.4. Внешние объекты**  
Если необходимо использовать глобальную переменную в нескольких файлах, необходимо объявить ее с ключевым словом extern в заголовочном файле. А в одном из исходных файлов определить эту переменную.



**Основные моменты:**

* extern int globalCounter; : объявляет глобальную переменную globalCounter в заголовочном файле.
* int globalCounter = 0;: определяет глобальную переменную в исходном файле.

**33.5. Подключение констант**  
Константы следует выносить в заголовочный файл, чтобы их можно было использовать в нескольких исходных файлах.



* Константы должны быть определены в заголовочном файле, чтобы их можно было использовать в нескольких исходных файлах.

**33.6. Вынесение объявлений в заголовочные файлы**

При разделении программы на файлы рекомендуется придерживаться следующих правил:

* Объявления функций, классов, структур, перечислений и других типов данных нужно размещать в заголовочных файлах.
* Реализации (тела) функций должны находиться в файлах с расширением .cpp.
* Глобальные переменные должны быть объявлены с ключевым словом extern в заголовочных файлах и определены в одном из .cpp файлов.
* Константы и макросы должны быть объявлены в заголовочных файлах.

**Практические задания:**

1. Разбейте программу, которую вы писали в предыдущей главе, на несколько файлов: заголовочный файл с объявлениями функций, и файл с реализацией функций.
2. Создайте заголовочный файл с объявлением глобальной переменной и используйте ее в нескольких исходных файлах.
3. Создайте заголовочный файл с объявлением нескольких констант и используйте их в нескольких исходных файлах.
4. Напишите небольшую библиотеку (набор функций) для работы с массивами и разделите ее на заголовочный файл и файл с реализацией.

**Глава 34: Динамическая память и смарт-указатели в C++**

Динамическое выделение памяти — это механизм, который позволяет программе выделять и освобождать память во время выполнения. В C++ для этого используются операторы new и delete.

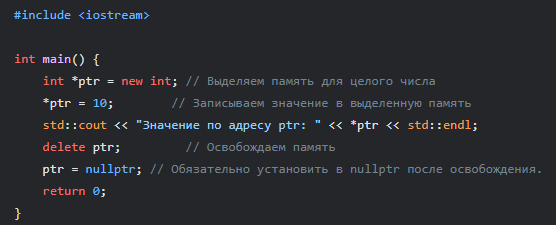
**34.1. Динамические объекты**

Когда мы объявляем переменную, компилятор выделяет память для нее в стеке (stack). Размер памяти для переменных в стеке известен во время компиляции. Но часто нам нужно выделять память во время выполнения программы, когда размер данных неизвестен заранее. Для этого используется динамическая память (heap).

**34.2. Выделение памяти (new)**

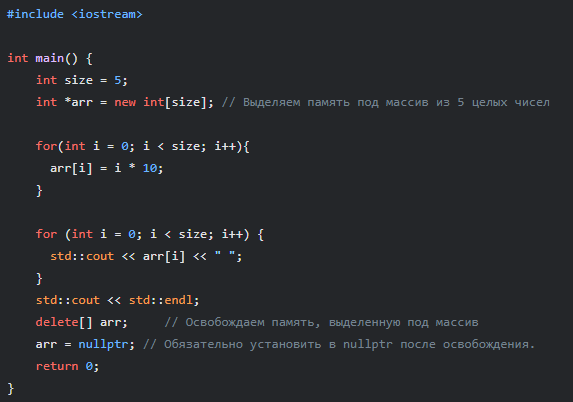
Оператор new используется для выделения памяти из кучи (heap) и возвращает указатель на начало выделенной области.

1. **Выделение памяти для одного объекта:**



* + new int: выделяет память для целого числа и возвращает указатель int\*.
  + \*ptr = 10;: записывает значение 10 по адресу, на который указывает ptr.

1. **Выделение памяти для массива (динамические массивы):**



* + new int[size]: выделяет память под массив из size целых чисел и возвращает указатель int\*.
  + delete[] arr; : освобождает память выделенную под массив.

**34.3. Освобождение памяти (delete)**

Когда выделенная динамически память больше не нужна, ее необходимо освободить с помощью оператора delete.

1. **Освобождение памяти для одного объекта:**



1. **Освобождение памяти для массива:**

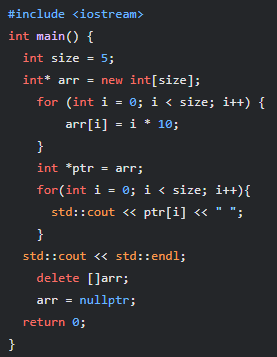


**Важно:**

* Освобождение памяти с помощью delete (или delete[]) нужно выполнять только с указателями, которые получили адрес памяти с помощью new (или new[]).
* Если не освободить динамически выделенную память, происходит *утечка памяти* (memory leak), что может привести к нестабильной работе программы или ее краху.
* После освобождения памяти, рекомендуется присваивать указателю значение nullptr, чтобы предотвратить попытку повторного освобождения.

**34.4. Указатель на массив**

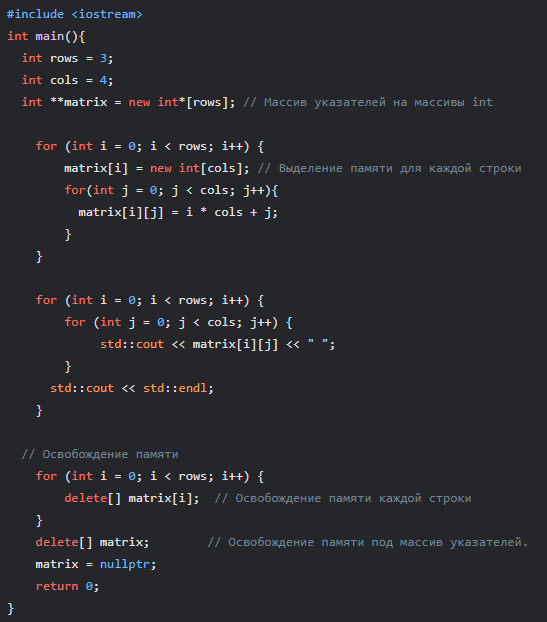
Когда вы выделяете память под массив, new возвращает указатель на первый элемент массива.



* int\* arr: переменная arr является указателем на int, и хранит адрес начала массива.
* int \*ptr = arr: указатель ptr указывает на начало массива.
* С указателем ptr можно обращаться как с обычным массивом.

**34.5. Многомерные массивы (динамическое выделение)**

Для создания динамических многомерных массивов нужно выделять память для каждого измерения массива.



**Основные моменты:**

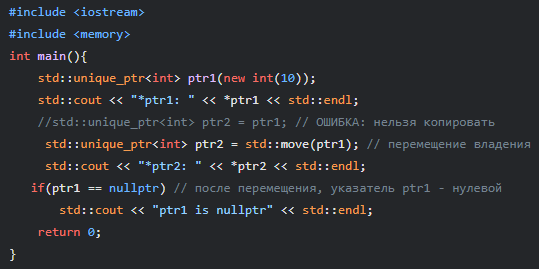
* int \*\*matrix = new int\*[rows];: Выделяет память под массив указателей, каждый из которых будет указывать на начало строки.
* В цикле выделяется память для каждого массива matrix[i] = new int[cols];, и заполняется значениями.
* Для освобождения памяти необходимо сначала освободить память каждой строки в цикле delete[] matrix[i];, а затем освободить память под массив указателей delete[] matrix;.
* Обязательно нужно установить указатели в nullptr после освобождения памяти.

**34.6. Смарт-указатели**

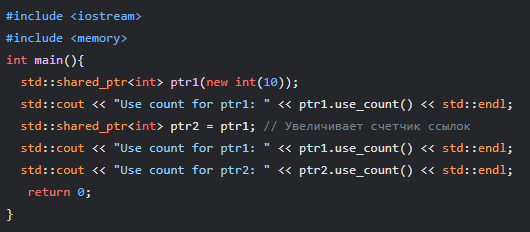
Для избежания утечек памяти и для более удобного управления динамической памятью, в C++ есть *смарт-указатели*. Смарт-указатели являются объектами, которые управляют временем жизни динамически выделенной памяти. Они автоматически освобождают память при выходе из области видимости.

C++ предоставляет три основных вида смарт-указателей:

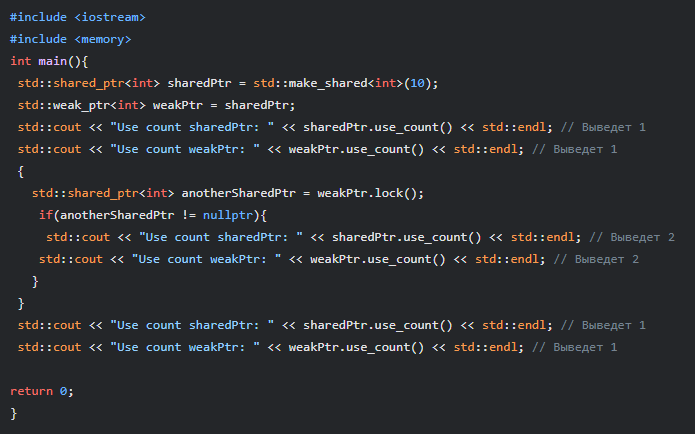
1. std::unique\_ptr:  
   Предназначен для владения единственным объектом в памяти. При выходе unique\_ptr из области видимости, объект автоматически уничтожается. Нельзя копировать.



1. std::shared\_ptr:  
   Позволяет нескольким смарт-указателям владеть одним и тем же объектом. Объект уничтожается, когда ни один shared\_ptr на него не ссылается.



1. std::weak\_ptr:  
   Похож на shared\_ptr, но не увеличивает счетчик ссылок. Используется для избежания циклической зависимости между shared\_ptr.



**Основные моменты:**

* Смарт-указатели автоматически освобождают память, когда она больше не нужна, что позволяет избежать утечек памяти.
* Используйте unique\_ptr, когда объект имеет одного владельца.
* Используйте shared\_ptr, когда объект может иметь нескольких владельцев.
* Используйте weak\_ptr для хранения ссылки на объект, который может быть удален.

**Практические задания:**

1. Выделите динамическую память для массива целых чисел и заполните его значениями.
2. Выделите динамическую память для двумерного массива и выведите его на экран.
3. Используйте смарт-указатель (unique\_ptr или shared\_ptr) для управления динамически выделенным массивом.
4. Напишите функцию, которая динамически выделяет память под строку и возвращает указатель на эту строку.
5. Напишите функцию, которая принимает двумерный массив (динамически выделенный) и его размеры, и выводит все его элементы.
6. Попробуйте использовать weak\_ptr и убедитесь, как он работает.

**Глава 35: Объектно-ориентированное программирование в C++**

Объектно-ориентированное программирование (ООП) — это подход, который организует код вокруг *объектов*, которые сочетают данные (атрибуты) и поведение (методы). В C++ для создания объектов используются *классы*.

**35.1. Определение классов**

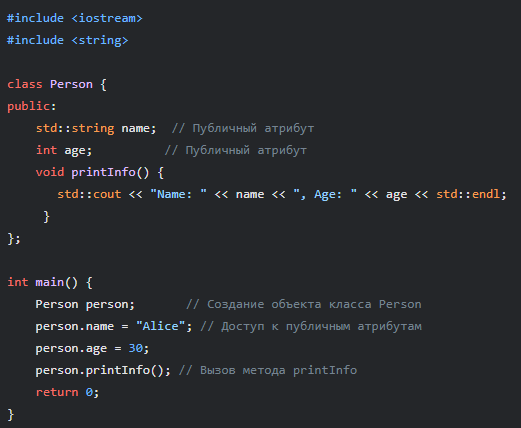
Класс — это пользовательский тип данных, который определяет структуру объектов, которые могут иметь собственные данные и методы.

Для определения класса используется ключевое слово class:



* class ИмяКласса: объявляет класс с указанным именем.
* **Модификаторы доступа:**
  + public: члены класса (атрибуты и методы) доступны из любого места программы.
  + private: члены класса доступны только внутри самого класса.
  + protected: члены класса доступны внутри самого класса и в его подклассах (наследование).
* **Атрибуты:** Переменные, которые представляют данные объекта.
* **Методы:** Функции, которые определяют поведение объекта.

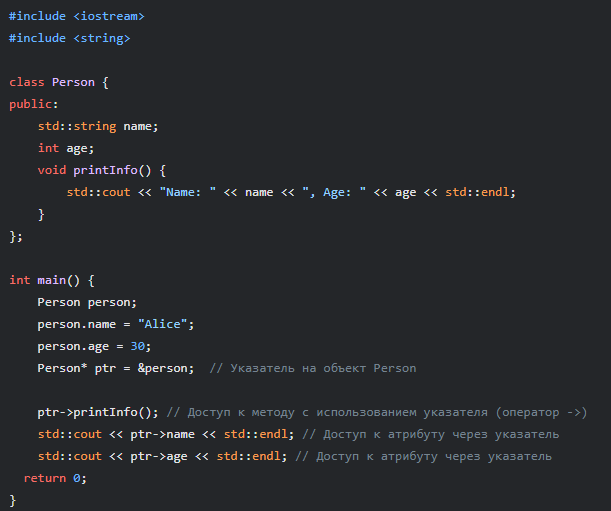
**Пример:**



* В этом примере, мы объявляем класс Person, который содержит публичные атрибуты name и age и публичный метод printInfo.

**35.2. Указатели на объекты классов**

Можно объявлять указатели на объекты классов.



**Основные моменты:**

* Person \*ptr = &person; объявляет ptr как указатель на объект типа Person.
* Для доступа к членам объекта класса через указатель используется оператор ->, например, ptr->printInfo().

**35.3. Конструкторы и инициализация объектов**

Конструктор — это специальный метод класса, который автоматически вызывается при создании нового объекта этого класса. Конструкторы используются для инициализации начальных значений атрибутов объекта.



**Основные моменты:**

* Конструктор имеет имя, совпадающее с именем класса.
* Конструктор не имеет типа возвращаемого значения (даже void).
* Конструктор автоматически вызывается при создании объекта.
* В примере выше, конструктор Person(std::string nameValue, int ageValue) инициализирует name и age.
* : name(nameValue), age(ageValue) - это список инициализации, который является более эффективным способом инициализации атрибутов, чем присваивание в теле конструктора.

**35.4. Определение нескольких конструкторов (перегрузка конструкторов)**

В классе можно объявить несколько конструкторов с разными параметрами (перегрузка конструкторов):



* Person(std::string nameValue, int ageValue) - конструктор с параметрами
* Person() - конструктор по умолчанию (без параметров)

**35.5. Параметры по умолчанию**

Можно использовать параметры по умолчанию в конструкторах, как и в обычных функциях:



**35.6. Инициализация констант и списки инициализации**

Константные атрибуты класса необходимо инициализировать с использованием списка инициализации в конструкторе:



**Основные моменты:**

* Circle(double radiusValue) : radius(radiusValue), PI(3.14159): список инициализации.
* Константные атрибуты могут быть инициализированы только один раз в списке инициализации конструктора.

**Практические задания:**

1. Создайте класс Rectangle с атрибутами width и height.
2. Добавьте в класс Rectangle конструктор для инициализации атрибутов.
3. Добавьте в класс Rectangle метод для вычисления площади прямоугольника.
4. Создайте несколько объектов класса Rectangle с разными значениями атрибутов.
5. Создайте класс Student с атрибутами name и id.
6. Создайте несколько конструкторов для класса Student.
7. Создайте класс с константным атрибутом и правильно инициализируйте его.

**Глава 36: Управление доступом, инкапсуляция, опосредование доступа в C++**

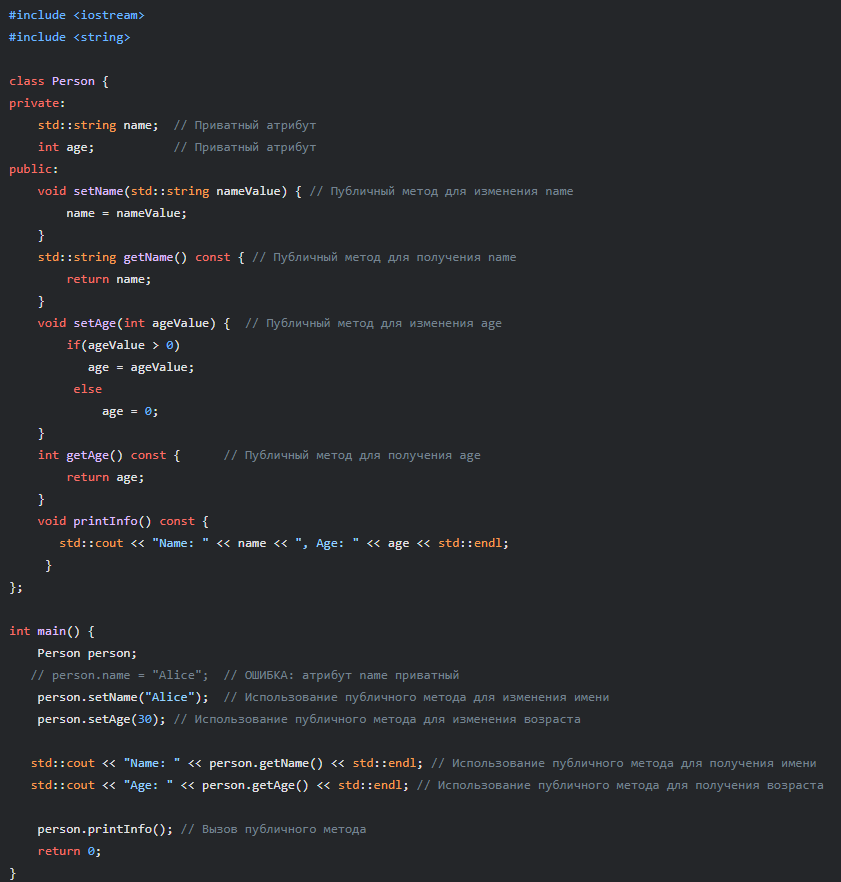
В этой главе мы подробно разберем, как управлять доступом к членам класса с помощью модификаторов доступа (public, private, protected), что такое инкапсуляция, как использовать методы класса для опосредования доступа к данным, и как правильно объявлять и определять функции класса.

**36.1. Модификаторы доступа**

В C++ модификаторы доступа используются для управления видимостью и доступностью членов класса (атрибутов и методов).

* **public:** Члены класса, объявленные с модификатором доступа public, доступны из любой части программы (как внутри, так и вне класса).
* **private:** Члены класса, объявленные с модификатором доступа private, доступны только внутри самого класса. Это позволяет скрыть внутренние детали реализации и контролировать доступ к данным.
* **protected:** Члены класса, объявленные с модификатором доступа protected, доступны внутри самого класса, а также в классах-наследниках.

**Пример:**



**Основные моменты:**

* Атрибуты name и age объявлены как private, поэтому доступ к ним напрямую из main невозможен.
* Публичные методы setName, getName, setAge и getAge обеспечивают контролируемый доступ к этим атрибутам.
* Метод printInfo может получить доступ к приватным атрибутам.
* Методы, не изменяющие атрибуты объекта, можно объявлять константными с помощью const (например getName, getAge, printInfo).

**36.2. Инкапсуляция**

Инкапсуляция — это один из ключевых принципов ООП, который заключается в объединении данных (атрибутов) и методов, которые работают с этими данными, в единый объект (класс). При этом детали реализации скрываются от внешнего мира.  
В примере выше, класс Person инкапсулирует данные (имя и возраст) и методы, работающие с этими данными. Атрибуты объявлены как private, а доступ к ним осуществляется через публичные методы (геттеры и сеттеры).

**36.3. Опосредование доступа (геттеры и сеттеры)**

Опосредование доступа — это техника, которая позволяет контролировать доступ к атрибутам класса через специальные методы:

* **Геттеры (getters):** методы, которые возвращают значение атрибута (например, getName, getAge).
* **Сеттеры (setters):** методы, которые устанавливают значение атрибута (например, setName, setAge).

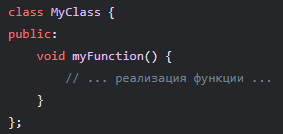
Геттеры и сеттеры позволяют:

* Контролировать доступ к атрибутам (например, проверять корректность введенных данных при присваивании).
* Изменять логику доступа к данным в одном месте (внутри метода), не затрагивая остальной код программы.
* Скрыть внутреннее представление данных.

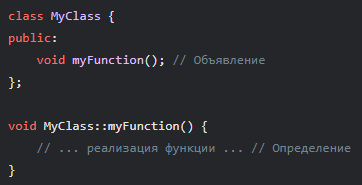
**36.4. Объявление и определение функций класса**

Функции (методы) класса можно объявлять и определять внутри класса или вне класса.

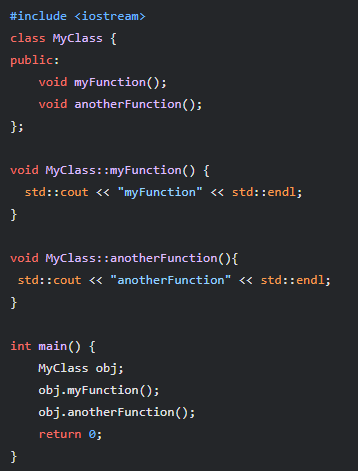
1. **Объявление и определение внутри класса:**



1. **Объявление внутри класса и определение вне класса:**



* void MyClass::myFunction(): MyClass:: указывает, что метод myFunction принадлежит классу MyClass.



**Основные моменты:**

* Методы можно объявлять внутри класса, а определять (реализовывать) их можно как внутри класса, так и вне.
* Если определение метода находится вне класса, то нужно указывать имя класса с помощью оператора разрешения области видимости ::, например, MyClass::myFunction().

**Практические задания:**

1. Создайте класс Rectangle с приватными атрибутами width и height.
2. Добавьте в класс Rectangle публичные геттеры и сеттеры для доступа к атрибутам.
3. Добавьте в класс Rectangle метод для вычисления площади прямоугольника.
4. Создайте класс BankAccout с приватным атрибутом balance.
5. Добавьте в класс BankAccout публичные методы для внесения депозита, снятия средств и получения текущего баланса.
6. Создайте класс Car с приватными атрибутами model и speed.
7. Добавьте публичные методы для управления скоростью автомобиля.
8. Создайте класс Counter с приватным атрибутом count.
9. Добавьте публичные методы для увеличения, уменьшения и получения текущего значения счетчика.

**Глава 37: Конструктор копирования, константные объекты и функции, mutable и this в C++**

В этой главе мы подробно разберем конструкторы копирования, рассмотрим работу с константными объектами и функциями, а также изучим ключевые слова mutable и this.

**37.1. Конструктор копирования**

Конструктор копирования — это специальный конструктор, который вызывается, когда создается новый объект на основе существующего объекта того же класса. Конструктор копирования используется для корректного копирования значений атрибутов объекта.

Если в классе не определен конструктор копирования, то компилятор сгенерирует его автоматически (поверхностное копирование). В некоторых случаях это может быть нежелательно (например при использовании динамической памяти).



**Основные моменты:**

* Конструктор копирования принимает константную ссылку на объект того же класса (const Person& other).
* В примере выше, конструктор копирования копирует значения атрибутов name и age из объекта other.
* Конструктор копирования необходим, если у класса есть указатели на динамически выделенную память. В этом случае нужно не просто копировать указатели, а выделить новую память и скопировать данные.

**37.2. Удаление конструктора копирования**

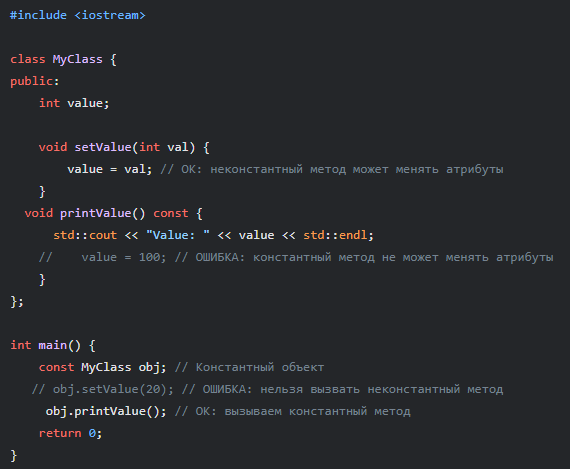
Если мы хотим запретить компилятору генерировать конструктор копирования (и запретить копирование объекта), можно удалить его, объявив со спецификатором = delete;



* Person(const Person& other) = delete; - удаляет конструктор копирования и запрещает создание копий объекта.

**37.3. Константные объекты**

Объект можно объявить константным, используя ключевое слово const. Для константных объектов можно вызывать только константные методы.

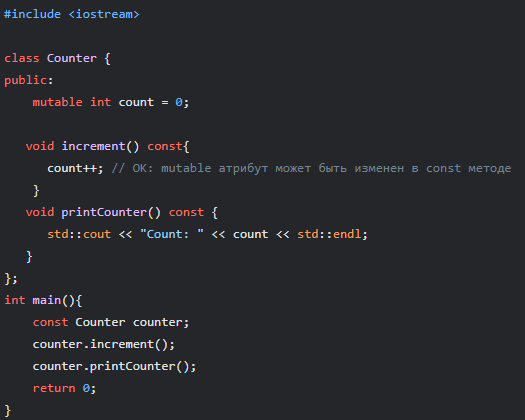


**Основные моменты:**

* const MyClass obj;: объявляет obj как константный объект.
* Константные объекты могут вызывать только константные методы.
* Константные методы не могут изменять атрибуты объекта.

**37.4. Ключевое слово mutable**

Ключевое слово mutable позволяет сделать атрибут объекта изменяемым даже из константного метода:



* mutable int count: объявляет count как изменяемый атрибут, даже если объект константный.
* increment() объявлен как константный метод, но при этом он может менять атрибут count.
* mutable следует использовать с осторожностью, когда есть очень веская причина, изменять атрибуты объекта внутри константных методов.

**37.5. Ключевое слово this**

Ключевое слово this представляет собой указатель на текущий объект класса внутри методов класса:



**Основные моменты:**

* this — это указатель на текущий объект, например this->name - доступ к атрибуту name текущего объекта.
* getThis метод возвращает ссылку на текущий объект.

**Практические задания:**

1. Создайте класс MyString с указателем на динамически выделенную память под строку.
2. Реализуйте конструктор копирования для класса MyString.
3. Протестируйте работу конструктора копирования с и без удаления.
4. Создайте константный объект класса MyClass и вызовите константный и не константный метод.
5. Создайте класс с mutable атрибутом и модифицируйте его в константном методе.
6. Создайте класс, который выводит адрес текущего объекта с использованием ключевого слова this.
7. Попробуйте использовать ссылку на текущий объект.

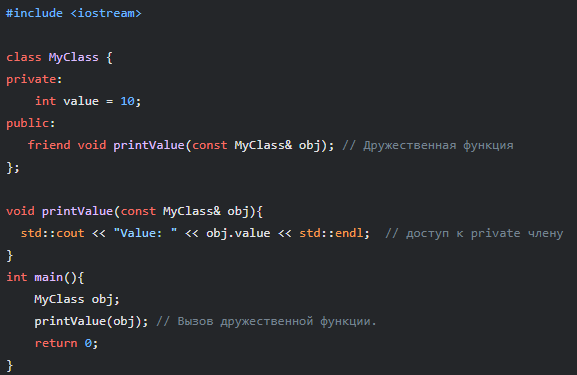
**Глава 38: Дружественные функции и классы, статические члены, деструкторы в C++**

В этой главе мы подробно разберем, что такое дружественные функции и классы, как они работают, что такое статические члены класса, как их использовать, и зачем нужны деструкторы.

**38.1. Дружественные функции**

Дружественная функция (friend function) — это функция, которая не является членом класса, но имеет доступ к private и protected членам этого класса. Дружественные функции используются в тех случаях, когда нужно предоставить доступ к внутренним данным класса извне, но не через интерфейс публичных методов.

Для объявления дружественной функции используется ключевое слово friend:



**Основные моменты:**

* friend void printValue(const MyClass& obj): объявляет функцию printValue дружественной классу MyClass.
* Дружественная функция printValue может напрямую обращаться к приватному атрибуту value класса MyClass.
* Дружественная функция *не является* методом класса, поэтому не имеет указателя this.

**38.2. Дружественные классы**

Дружественный класс (friend class) — это класс, который имеет доступ к private и protected членам другого класса. Дружественные классы применяются, когда необходимо дать доступ к внутренним данным одного класса другому.



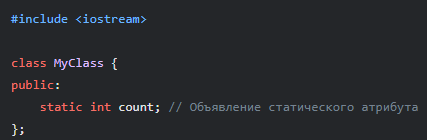
**Основные моменты:**

* friend class FriendClass;: объявляет класс FriendClass дружественным классу MyClass.
* Методы FriendClass имеют доступ к приватным атрибутам MyClass.
* Дружественность не является взаимной. Если FriendClass является другом MyClass, то это не значит, что MyClass является другом FriendClass.

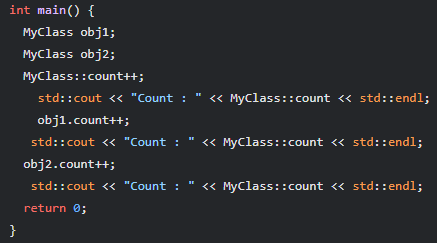
**38.3. Статические члены класса**

Статические члены класса (static members) принадлежат всему классу, а не конкретному объекту.

1. **Статические атрибуты:**
   * Статические атрибуты существуют в единственном экземпляре для всех объектов класса.
   * Статические атрибуты обычно используются для хранения глобальной информации, связанной с классом.
   * Статический атрибут объявляется с помощью ключевого слова static внутри класса и определяется вне класса.

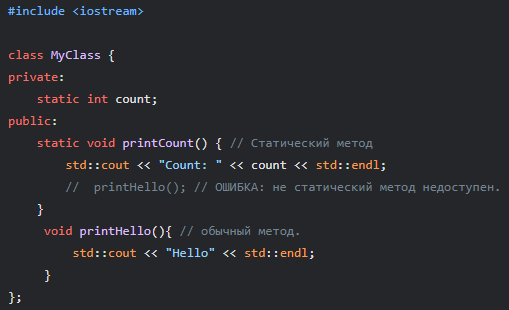


int MyClass::count = 0; // Определение статического атрибута

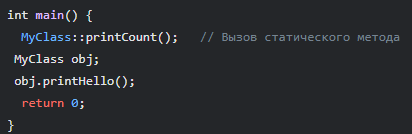


* static int count;: объявляет статический атрибут count.
* int MyClass::count = 0;: определяет (и инициализирует) статический атрибут count вне класса.
* MyClass::count++, obj1.count++, obj2.count++ - все увеличивают одну и ту же статическую переменную.

1. **Статические методы:**
   * Статические методы могут вызываться без создания объекта класса.
   * Статические методы имеют доступ только к статическим членам класса.



int MyClass::count = 10;

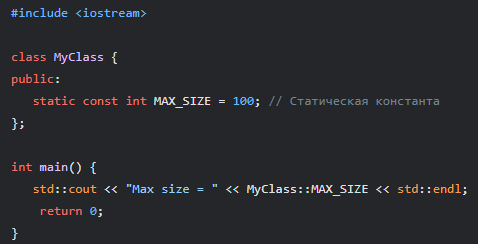


**Основные моменты:**

* static void printCount(): объявляет статический метод printCount().
* Статический метод может вызываться через имя класса (MyClass::printCount()) без создания объекта класса.
* Статические методы имеют доступ только к статическим членам класса.

**38.4. Статические константы**

Статические константные атрибуты можно объявлять и инициализировать непосредственно в классе.

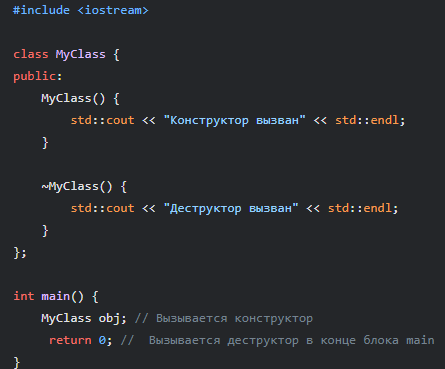


* static const int MAX\_SIZE = 100: объявляет статическую константу.
* Инициализация происходит непосредственно в классе.

**38.5. Деструктор**

Деструктор — это специальный метод класса, который автоматически вызывается при уничтожении объекта. Деструкторы используются для освобождения ресурсов, которые были выделены объектом (например, динамически выделенной памяти).

Деструктор имеет имя, совпадающее с именем класса, но перед именем ставится тильда ~:



**Основные моменты:**

* Деструктор не имеет параметров и не возвращает значения.
* Деструктор вызывается автоматически, когда объект выходит из области видимости или удаляется с помощью delete.
* Деструктор необходим, когда класс работает с динамической памятью, чтобы освободить выделенную память.

**Практические задания:**

1. Создайте класс Counter с приватным атрибутом count и дружественной функцией, которая может увеличивать значение счетчика.
2. Создайте класс Logger, который пишет сообщения в файл. Сделайте этот класс другом другого класса DataProcessor, который использует Logger для записи логов.
3. Создайте класс IdGenerator, который генерирует уникальные ID для создаваемых объектов. Используйте статический атрибут для хранения текущего ID и статический метод для его получения.
4. Создайте класс, который динамически выделяет память в конструкторе, и освобождает её в деструкторе.
5. Создайте класс, содержащий статический константный атрибут.
6. Создайте класс, где один метод статический, а другой обычный.

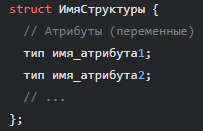
**Глава 39: Структуры и перечисления в C++**

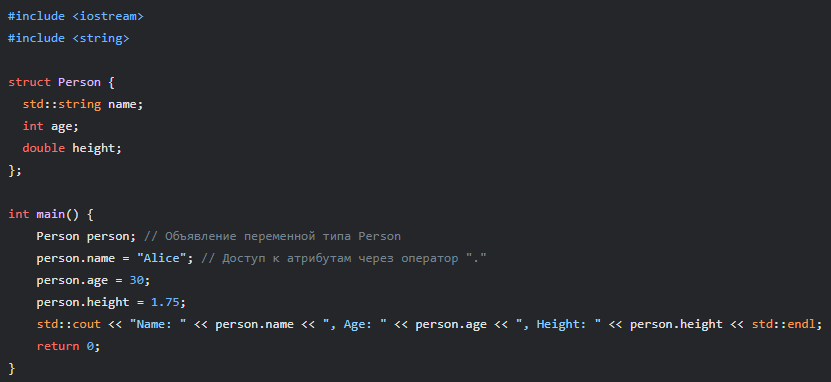
В этой главе мы рассмотрим структуры (struct), перечисления (enum) и их применение в C++. Структуры позволяют объединять данные разных типов в одну сущность, а перечисления позволяют создавать именованные константы, что улучшает читаемость кода.

**39.1. Структуры (struct)**

Структура — это составной тип данных, который позволяет объединять переменные разных типов в одну сущность. Структуры похожи на классы, но имеют некоторые отличия (по умолчанию все члены public).

Для объявления структуры используется ключевое слово struct:



**Пример:** ****

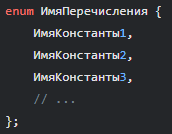
**Основные моменты:**

* struct Person: объявляет структуру с именем Person.
* Внутри структуры объявляются атрибуты, которые могут иметь разные типы.
* Доступ к атрибутам структуры осуществляется с помощью оператора ..

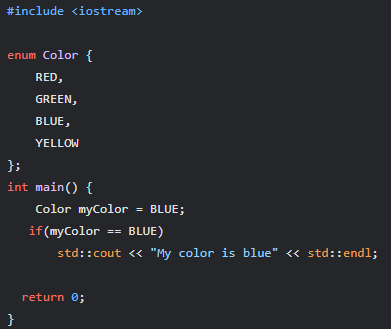
**39.2. Перечисления (enum)**

Перечисление (enum) — это тип данных, который позволяет объявить набор именованных целочисленных констант. Перечисления улучшают читаемость кода, когда требуется использовать набор именованных значений.

Для объявления перечисления используется ключевое слово enum:

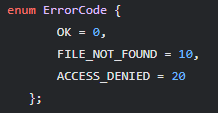


**Пример:**



**Основные моменты:**

* enum Color: объявляет перечисление с именем Color.
* RED, GREEN, BLUE, YELLOW: это константы в перечислении. По умолчанию им присваиваются значения 0, 1, 2, 3 и так далее, по порядку.
* Можно задавать конкретное значение константам:



**39.3. Применение перечислений**

Перечисления часто используются для:

* Обозначения состояний (например, enum Status { ON, OFF, PENDING };).
* Обозначения выбора из нескольких вариантов (например, enum Direction { UP, DOWN, LEFT, RIGHT };).
* Обозначения кодов ошибок (например, enum ErrorCode { OK, FILE\_NOT\_FOUND, ACCESS\_DENIED };).
* Создания набора именованных констант, которые могут быть использованы в качестве параметров функций



* Перечисления улучшают читаемость кода, так как вместо числовых значений используются понятные имена констант.
* Также, enum может быть использован в switch, когда нужно обрабатывать несколько вариантов.

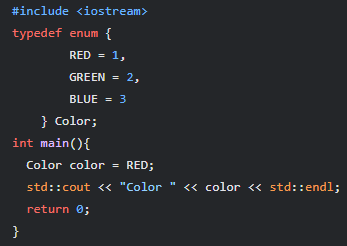
**39.4. Подключение констант перечисления**

Можно подключать константы перечисления как обычные константы, если объявить их как статические константы.



* static const TrafficLight RED\_LIGHT = RED; - статическая константа типа TrafficLight.
* Объявление константы в class TrafficLightSystem, позволяет использовать ее как TrafficLightSystem::GREEN\_LIGHT

**39.5. Перечисления в C-стиле**  
C-стиль для объявления перечислений, это с ключевым словом typedef



**Практические задания:**

1. Создайте структуру Point с атрибутами x и y (числа с плавающей точкой).
2. Создайте структуру Book с атрибутами title (строка), author (строка) и year (целое число).
3. Объявите перечисление LogLevel с константами DEBUG, INFO, WARNING, ERROR.
4. Напишите функцию, которая принимает параметр типа LogLevel и выводит на экран сообщение, соответствующее этому уровню логирования.
5. Объявите перечисление с именованными константами для дней недели.
6. Создайте структуру, в которой один из атрибутов будет иметь тип объявленного перечисления.
7. Создайте структуру которая содержит в себе другую структуру.

**Глава 40: Наследование в C++**

Наследование — это механизм, который позволяет создать новый класс (производный, дочерний или подкласс), основываясь на существующем классе (базовом, родительском или суперклассе). Производный класс наследует атрибуты и методы базового класса, что позволяет повторно использовать код.

**40.1. Наследование**  
Для объявления наследования используется символ : и ключевое слово, которое определяет тип наследования (public, private, protected) после имени производного класса:



**Основные моменты:**

* class Dog : public Animal: объявляет класс Dog как производный от класса Animal. public - это тип наследования.
* Класс Dog наследует публичные методы и атрибуты класса Animal.
* При создании объекта класса Dog, сначала вызывается конструктор базового класса, а затем конструктор производного класса.

**40.2. Конструкторы и наследование**

Конструкторы базового класса *не наследуются* производными классами. При создании объекта производного класса, конструктор базового класса должен быть вызван явно через список инициализации:

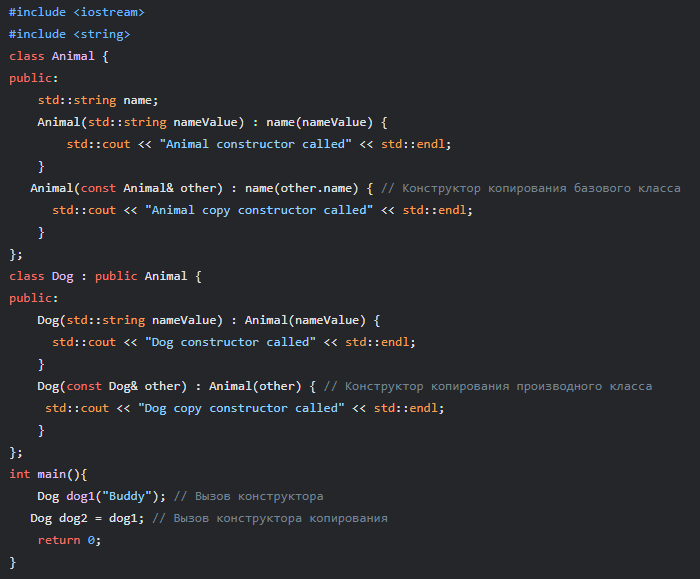


**Основные моменты:**

* Dog(std::string nameValue) : Animal(nameValue) - вызывает конструктор базового класса Animal с именем nameValue
* При создании объекта Dog, сначала вызывается конструктор базового класса Animal, а затем конструктор производного класса Dog.
* Если конструктор базового класса не будет вызван явно, то будет вызван конструктор по умолчанию, если он существует.

**40.3. Определение конструкторов копирования**

Конструкторы копирования в производных классах нужно определять, когда требуется копировать не только собственные, но и унаследованные атрибуты базового класса. Конструктор копирования базового класса, также должен быть вызван явно через список инициализации.



**Основные моменты:**

* Dog(const Dog& other) : Animal(other): Конструктор копирования производного класса явно вызывает конструктор копирования базового класса.

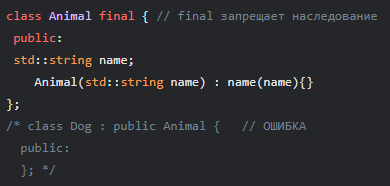
**40.4. Наследование деструкторов**

В отличии от конструкторов, деструкторы наследуются. При уничтожении объекта производного класса, сначала вызывается деструктор производного класса, а затем деструктор базового класса.



**40.5. Запрет наследования**

Можно запретить наследование класса, используя ключевое слово final.



**40.6. Управление доступом в базовых и производных классах**

При наследовании используется модификатор доступа (public, private или protected) для контроля доступа к членам базового класса в производном классе.

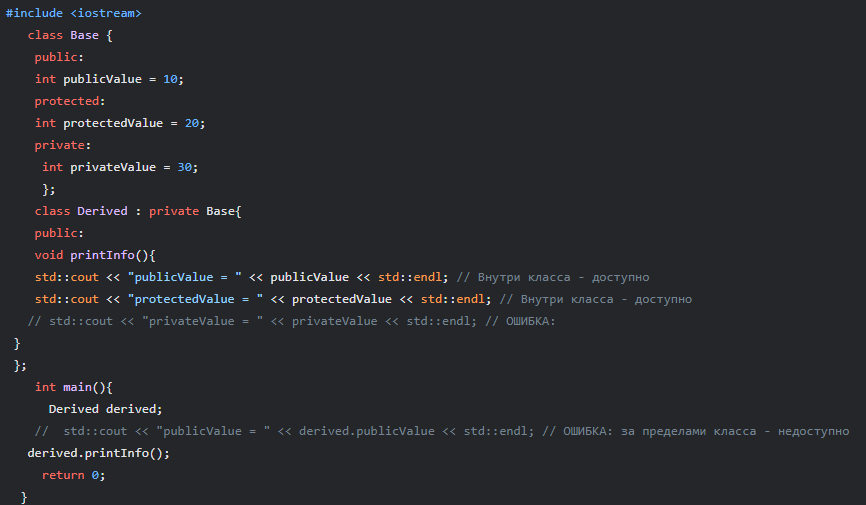
1. **public наследование:**
   * public члены базового класса остаются public в производном классе.
   * protected члены базового класса остаются protected в производном классе.
   * private члены базового класса недоступны в производном классе.

#include <iostream>

class Base {  
public:  
int publicValue = 10;  
protected:  
int protectedValue = 20;  
private:  
int privateValue = 30;  
};  
class Derived : public Base{  
public:  
void printInfo(){  
std::cout << "publicValue = " << publicValue << std::endl;  
std::cout << "protectedValue = " << protectedValue << std::endl;  
//std::cout << "privateValue = " << privateValue << std::endl; // ОШИБКА  
}  
};  
int main(){  
Derived derived;  
std::cout << "publicValue = " << derived.publicValue << std::endl;  
derived.printInfo();  
return 0;  
}

2. **private наследование:**

* public члены базового класса становятся private в производном классе.  
  \* protected члены базового класса становятся private в производном классе.  
  \* private члены базового класса недоступны в производном классе.



* 1. **`protected` наследование:**
* `public` члены базового класса становятся `protected` в производном классе.
* `protected` члены базового класса остаются `protected` в производном классе.
* `private` члены базового класса недоступны в производном классе.

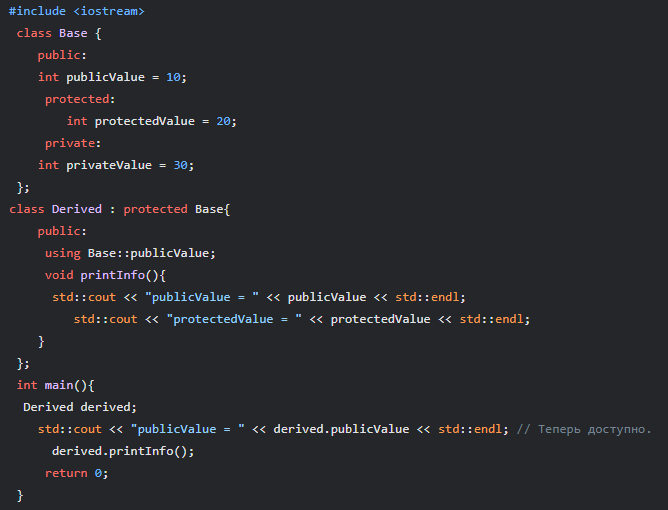


**40.7. Уровень доступа protected**

protected члены класса доступны:

* Внутри самого класса.
* В производных классах (независимо от типа наследования).
* Но недоступны извне.

**40.8. Установка публичного доступа (использование using)**  
Можно установить публичный доступ к членам базового класса, если был использован private или protected тип наследования, при помощи using



**Основные моменты:**

* using Base::publicValue; устанавливает публичный доступ к publicValue в производном классе Derived, даже если был использован protected тип наследования.

**Практические задания:**

1. Создайте класс Shape с атрибутом color.
2. Создайте производный класс Rectangle от класса Shape с атрибутами width и height.
3. Добавьте конструкторы в базовый и производный классы.
4. Создайте деструкторы в базовом и производном классе.
5. Попробуйте запретить наследование класса.
6. Создайте базовый класс с модификаторами доступа public, private и protected и проверьте доступ к этим членам в производном классе с разными типами наследования.
7. Создайте базовый класс и производный, с разными модификаторами доступа и установите публичный доступ, через using
8. Создайте класс Animal с методом eat(). Создайте производные классы Dog и Cat и переопределите метод eat() в каждом из них.

**Глава 41: Скрытие функционала, множественное наследование и виртуальные базовые классы в C++**

В этой главе мы разберем, как происходит скрытие функционала, что такое множественное наследование, какие проблемы оно может вызывать, и как использовать виртуальные базовые классы для их решения.

**41.1. Скрытие функционала базового класса**

При наследовании, в производном классе можно определить метод с таким же именем, как и метод в базовом классе. В этом случае метод базового класса будет *скрыт* в производном классе.



**Основные моменты:**

* void printMessage() в Derived *скрывает* метод printMessage() в Base.
* При вызове derived.printMessage() вызывается метод производного класса Derived.
* Для явного вызова скрытого метода базового класса, необходимо использовать оператор разрешения области видимости :: (например, derived.Base::printMessage()).

**41.2. Скрытие функций**

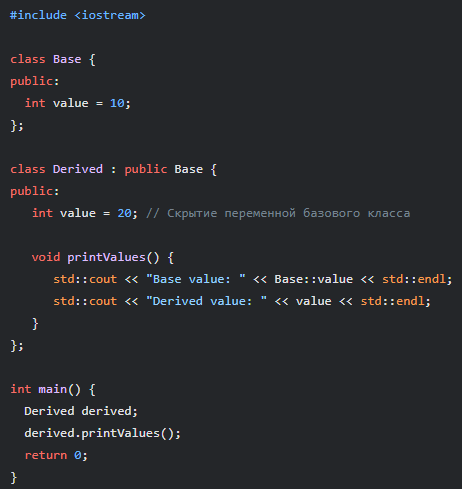
Скрытие функций может возникать не только при полном совпадении имени и сигнатуры метода, но и при совпадении имени и разных параметрах.



* Метод void printValue() в классе Derived скрывает метод void printValue(int) в классе Base.
* Скрытые функции могут быть доступны с использованием оператора разрешения области видимости ::.

**41.3. Скрытие переменных**

Аналогично методам, можно скрыть атрибуты базового класса.



**Основные моменты:**

* Атрибут int value в классе Derived скрывает атрибут int value в классе Base.
* Для явного доступа к скрытому атрибуту используется оператор разрешения области видимости ::, например Base::value.

**41.4. Множественное наследование**  
Множественное наследование — это возможность наследовать от нескольких базовых классов.



* class Derived : public Base1, public Base2 - класс Derived наследует от Base1 и Base2.

**41.5. Проблема множественного наследования**  
Множественное наследование может вызвать проблему "ромбовидного наследования", когда несколько классов наследуются от общего базового класса.



**Основные моменты:**

* DerivedDerived наследуется от Derived1 и Derived2, которые в свою очередь наследуются от Base.
* Возникает двойственность при доступе к атрибуту value, который есть в базовых классах Base1 и Base2, т.к. оба наследуются от Base.
* Для доступа к атрибутам базового класса нужно явно указывать имя базового класса Derived1::value.

**41.6. Виртуальные базовые классы**

Для решения проблемы "ромбовидного наследования" используется виртуальное наследование базового класса, это гарантирует, что будет создан только один экземпляр базового класса.



* class Derived1 : virtual public Base: объявляет Derived1 как виртуальный наследник класса Base.
* Теперь в DerivedDerived будет один экземпляр Base, и нет двойственности при обращении к value.
* При создании объекта DerivedDerived конструктор Base будет вызван только один раз.

**41.7. Конструкторы и деструкторы при множественном наследовании**

При множественном наследовании конструкторы базовых классов вызываются в порядке их объявления в списке наследования, а деструкторы вызываются в обратном порядке.



**Основные моменты:**

* При создании объекта Derived вызываются конструкторы в порядке объявления в списке наследования Base1, затем Base2 и после - Derived
* При уничтожении объекта Derived вызываются деструкторы в обратном порядке: Derived, затем Base2 и после - Base1.

**Практические задания:**

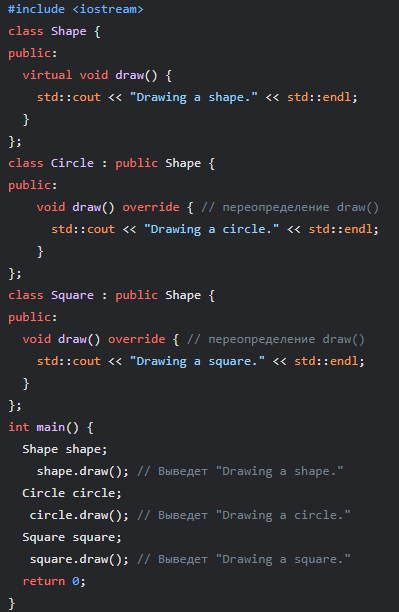
1. Создайте базовый класс Vehicle с методом start(). Создайте производный класс Car, который скроет метод start() базового класса.
2. Создайте базовый класс Base с атрибутом value. Создайте производный класс Derived, который скроет этот атрибут.
3. Создайте классы A, B и C, где C наследуется от A и B. Продемонстрируйте проблему "ромбовидного наследования".
4. Измените код, чтобы использовать виртуальное наследование и решить проблему ромбовидного наследования.
5. Создайте несколько классов с конструкторами и деструкторами и проверьте порядок их вызова при множественном наследовании.
6. Продемонстрируйте на примере, как можно переопределить метод базового класса и как вызвать этот метод из производного класса.

**Глава 42: Виртуальные функции и полиморфизм в C++**

Полиморфизм (многообразие форм) — это возможность обрабатывать объекты разных классов с помощью общего интерфейса. В C++ полиморфизм достигается с помощью виртуальных функций.

**42.1. Виртуальные функции и их переопределение**

Виртуальная функция — это метод класса, объявленный с ключевым словом virtual. Виртуальные функции позволяют производным классам *переопределять* реализацию метода базового класса.

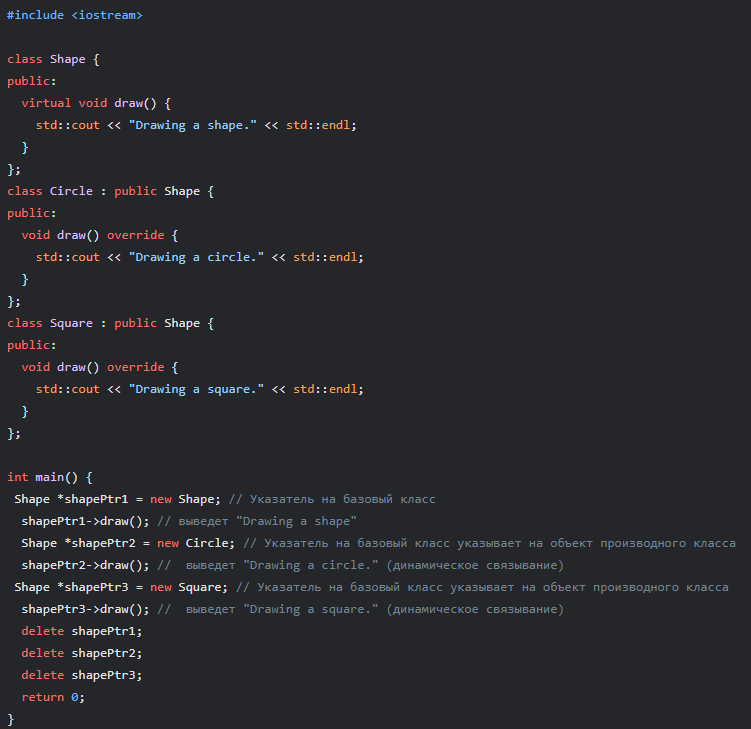


**Основные моменты:**

* virtual void draw(): объявляет функцию draw как виртуальную в классе Shape.
* void draw() override: переопределяет метод draw() в классах Circle и Square. override - помогает избежать ошибок при переопределении и проверяет, что метод из базового класса действительно переопределен.
* Объекты производных классов Circle и Square вызывают свою реализацию draw(), переопределяя поведение базового класса.

**42.2. Динамическое связывание и виртуальные функции**

Виртуальные функции реализуют *динамическое связывание* (или позднее связывание). Это означает, что вызываемая функция определяется во время выполнения программы (runtime), основываясь на фактическом типе объекта.



**Основные моменты:**

* Shape \*shapePtr2 = new Circle;: указатель на базовый класс Shape указывает на объект производного класса Circle.
* При вызове shapePtr2->draw(), вызывается метод draw() именно объекта Circle (полиморфизм), а не базового класса Shape.
* Компилятор решает какой метод нужно вызвать во время выполнения программы, основываясь на *фактическом* типе объекта, на который указывает указатель.

**42.3. Ключевое слово override**

Ключевое слово override используется для явного указания, что метод в производном классе должен переопределить метод базового класса. Если такого метода в базовом классе нет, то будет ошибка компиляции.  
Использование override делает код более читаемым и безопасным.



**Основные моменты:**

* void myFunction() override: гарантирует, что метод myFunction переопределяет метод базового класса.
* Если в базовом классе нет virtual метода с таким же именем и сигнатурой, то будет ошибка компиляции.

**42.4. Принцип выполнения виртуальных функций**

Виртуальные функции работают за счет использования *таблицы виртуальных функций* (vtable). Каждый класс, содержащий виртуальные функции, имеет свою vtable, которая содержит адреса виртуальных методов. Каждый объект класса, содержащего виртуальные методы имеет скрытый указатель на vtable своего класса.

При вызове виртуальной функции:

1. Компилятор определяет vtable объекта на основе его фактического типа.
2. По vtable определяется адрес нужной функции.
3. Вызывается функция по полученному адресу.

Это механизм динамического связывания.

**42.5. Запрет переопределения**  
Можно запретить дальнейшее переопределение виртуальной функции в производных классах, используя ключевое слово final.



**Основные моменты:**

* void myFunction() override final: переопределяет метод и запрещает дальнейшее переопределение в наследниках.

**Практические задания:**

1. Создайте класс Animal с виртуальным методом makeSound().
2. Создайте производные классы Dog, Cat и Cow, которые переопределяют метод makeSound().
3. Напишите функцию, которая принимает указатель на Animal и вызывает метод makeSound(). Проверьте как работает полиморфизм.
4. Создайте базовый класс Shape с виртуальным методом area().
5. Создайте производные классы Circle и Rectangle, которые переопределяют метод area().
6. Создайте базовый класс с виртуальным методом и переопределите его в производном классе.
7. Попробуйте запретить дальнейшее переопределение виртуальной функции в производных классах с помощью ключевого слова final.
8. Продемонстрируйте работу override когда функция не переопределяет метод базового класса.

**Глава 43: Преобразование типов, динамическое связывание и виртуальные деструкторы в C++**

В этой главе мы рассмотрим различные способы преобразования типов, особенности динамического связывания, а также виртуальные деструкторы и управление доступом при наследовании.

**43.1. Преобразование типов указателей (Type Casting)**

Преобразование типов указателей — это процесс изменения типа указателя. В C++ для преобразования типов указателей используются следующие операторы:

1. **static\_cast**:
   * Используется для преобразования указателей между базовыми и производными классами в пределах иерархии наследования.
   * Преобразование вниз по иерархии (Base\* в Derived\*) требует явного приведения, и оно *небезопасно* (при неправильном приведении может вызвать неопределенное поведение). Преобразование вверх по иерархии (Derived\* в Base\*) выполняется неявно.



1. **dynamic\_cast**:

* Используется для преобразования указателей между базовыми и производными классами в пределах иерархии наследования, но с проверкой типа во время выполнения.
* Преобразование вниз по иерархии (Base\* в Derived\*) выполняется только в случае, если объект, на который указывает базовый указатель, является объектом производного класса. Иначе, возвращается nullptr
* dynamic\_cast работает только с полиморфными классами (классами с виртуальными методами).



1. **reinterpret\_cast:**

* Используется для низкоуровневого преобразования указателей и может привести к непредсказуемым результатам, если используется неправильно.
* Как правило его нужно избегать, если можно использовать другие более безопасные методы.

**43.2. Преобразование смарт-указателей**

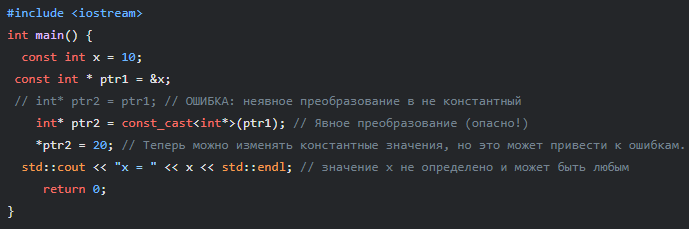
Смарт-указатели тоже можно преобразовывать между собой, но здесь также надо учитывать иерархию классов.



**Основные моменты:**

* Для преобразования shared\_ptr в производный тип необходимо использовать std::dynamic\_pointer\_cast, чтобы была выполнена проверка типов.

**43.3. Константность**  
При преобразовании указателей нужно помнить о константности. Указатель на константу можно преобразовать в указатель на неконстантное значение, но это может быть опасно.

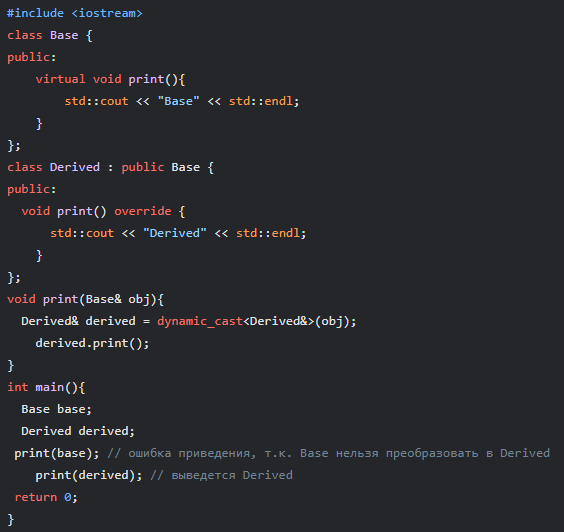


**Основные моменты:**

* const\_cast удаляет константность объекта, но нужно быть крайне осторожным при его использовании.

**43.4. Преобразование ссылок**

Преобразование ссылок аналогично преобразованию указателей, но оно более безопасное, так как при неверном приведении типов возникнет исключение.



**Основные моменты:**

* При неверном преобразовании (преобразование Base в Derived), будет сгенерировано исключение.

**43.5. Особенности динамического связывания**  
Динамическое связывание работает только если:

* Указатель или ссылка имеет тип базового класса.
* Метод объявлен как virtual в базовом классе.
* Метод должен быть переопределен в производном классе, чтобы работал полиморфизм.

**43.6. Динамическое связывание при передаче параметров**

Динамическое связывание также работает, когда указатели или ссылки на базовый класс передаются в качестве параметров функции.



* animalSound(Animal& animal): принимает ссылку на базовый класс.
* При вызове animalSound вызывается переопределенный метод производного класса.

**43.7. Динамическое связывание и коллекции**

Динамическое связывание работает при работе с коллекциями объектов базового класса.

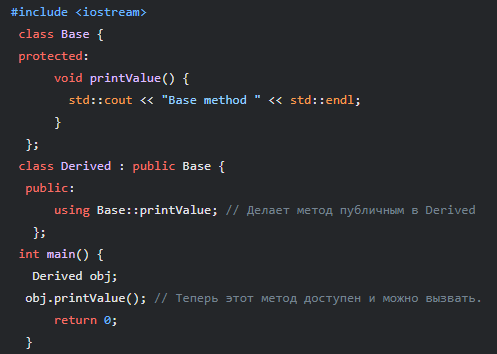


**43.8. Виртуальные деструкторы**  
В иерархии наследования с виртуальными методами деструктор базового класса должен быть объявлен как virtual. Иначе будет утечка памяти, когда через указатель на базовый класс будет удаляться производный объект.



* virtual ~Base(): объявляет деструктор Base как виртуальный.
* ~Derived() override: переопределяет деструктор базового класса.
* Теперь через указатель на базовый класс будет вызван и деструктор производного класса.

**43.9. Переопределение спецификатора доступа**  
В производном классе можно изменить спецификатор доступа к унаследованному методу с помощью using:



* using Base::printValue: делает protected метод printValue - public в производном классе Derived.

**Практические задания:**

1. Создайте иерархию классов с виртуальными методами и попробуйте выполнить динамическое преобразование типов указателей.
2. Создайте иерархию классов с виртуальным деструктором и проверьте, что он вызывается при удалении производных объектов через указатель на базовый класс.
3. Создайте классы с разным уровнем доступа к методам и попробуйте изменить доступ в производных классах.
4. Напишите программу, которая создает объекты разных классов и сохраняет их в векторе через указатель на базовый класс, и затем вызывает их методы.
5. Напишите функцию, которая принимает ссылку на базовый класс и пытается преобразовать ее в ссылку на производный класс.
6. Создайте классы, которые демонстрируют порядок вызова деструкторов при наследовании.

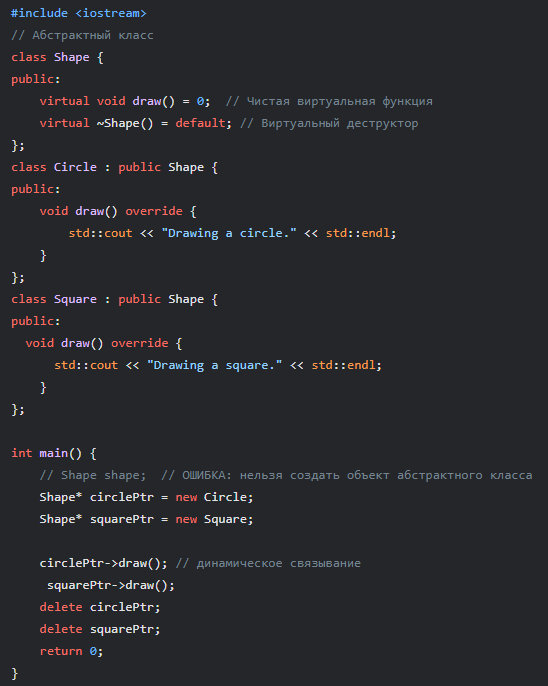
**Глава 44: Абстрактные классы, перегрузка операторов в C++**

В этой главе мы подробно разберем, что такое абстрактные классы и чистые виртуальные функции, как перегружать операторы, включая операторы сравнения, присваивания, унарные операции, инкремента и декремента, а также как выражать одни операторы через другие.

**44.1. Чистые виртуальные функции и абстрактные классы**

Чистая виртуальная функция — это виртуальная функция, которая объявлена в базовом классе, но не имеет реализации (тела). Чистая виртуальная функция объявляется с = 0.

Абстрактный класс — это класс, который содержит хотя бы одну чистую виртуальную функцию. Абстрактный класс не может иметь объектов, он может использоваться только как базовый класс для других классов. Абстрактные классы используются для определения общего интерфейса для классов-наследников.



**Основные моменты:**

* virtual void draw() = 0: объявляет draw() как чистую виртуальную функцию.
* Класс Shape является абстрактным, так как содержит чистую виртуальную функцию.
* Нельзя создать объект абстрактного класса, но можно использовать указатели на абстрактный класс для полиморфного поведения.
* Производные классы от абстрактного класса, должны реализовать чистые виртуальные функции, иначе они тоже будут абстрактными.

**44.2. Перегрузка операторов**

Перегрузка операторов позволяет нам определять, как операторы (например, +, -, ==, <, []) будут вести себя при применении к объектам наших классов. Это позволяет сделать код более выразительным и интуитивно понятным.

Для перегрузки оператора используется ключевое слово operator и символ оператора:



**44.3. Перегрузка операторов сравнения**

Для сравнения объектов можно перегрузить операторы ==, !=, <, >, <=, >=:



**Основные моменты:**

* bool operator ==(const Point& other) const: перегрузка оператора ==.
* Операторы сравнения возвращают логическое значение (true или false).
* Операторы сравнения, как правило, объявляются как const, так как они не должны менять объект.

**44.4. Перегрузка оператора присваивания**

Оператор присваивания = по умолчанию выполняет *побитовое* копирование, но для классов с динамической памятью нужно определить перегрузку оператора присваивания, чтобы обеспечить правильное копирование данных:



**Основные моменты:**

* MyString& operator = (const MyString& other) - перегрузка оператора присваивания.
* Оператор присваивания должен возвращать ссылку на текущий объект return \*this;.
* Необходимо предусмотреть проверку на самоприсваивание if(this != &other).
* Необходимо скопировать выделенную память, а не только указатель на нее.
* Необходимо освободить предыдущую память, если она была выделена.

**44.5. Перегрузка унарных операторов**

Унарные операторы работают с одним операндом. Например, - (унарный минус), ! (логическое отрицание) и другие.



* Number operator-() const; - перегрузка унарного минуса.
* bool operator !() const; - перегрузка логического отрицания.

**44.6. Перегрузка операций инкремента и декремента**

Перегрузка префиксного (++a) и постфиксного (a++) операторов инкремента/декремента требует немного больше кода. Для различения префиксных и постфиксных операций используются фиктивные параметры int.

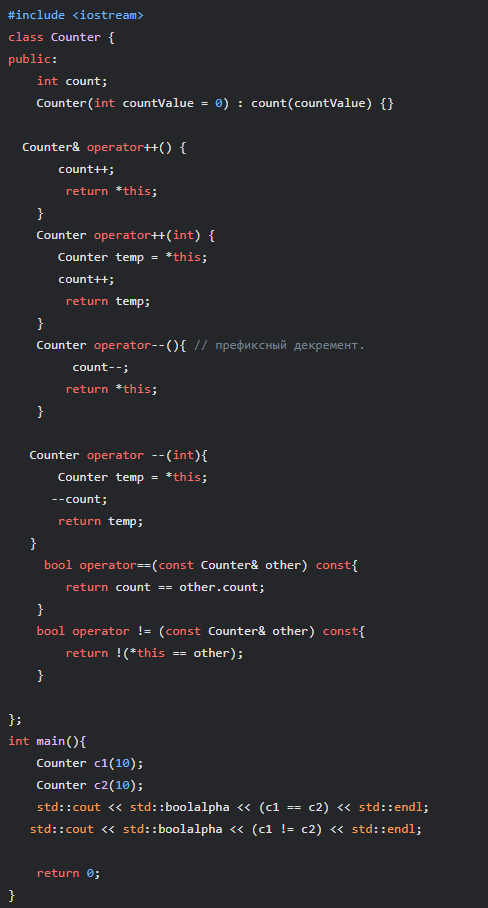


**Основные моменты:**

* Counter& operator++(): перегрузка префиксного инкремента (++a).
* Counter operator++(int): перегрузка постфиксного инкремента (a++), int - фиктивный параметр, который помогает компилятору различать префиксную и постфиксную версии.

**44.7. Выражение одних операторов через другие**

Обычно перегрузку одних операторов можно выразить через другие. Например, оператор != можно выразить через ==, а оператор -- через ++.



**Практические задания:**

1. Создайте абстрактный класс Shape с чистой виртуальной функцией area().
2. Создайте производные классы Circle и Rectangle от Shape и реализуйте в них метод area().
3. Создайте класс Vector с атрибутами x и y (целые числа).
4. Перегрузите операторы + и - для сложения и вычитания векторов.
5. Перегрузите операторы сравнения ==, !=, <, >, <=, >= для сравнения векторов.
6. Создайте класс с динамически выделенной памятью, а также перегрузите оператор присваивания для этого класса.
7. Перегрузите префиксный и постфиксный операторы ++ и -- для некоторого класса.
8. Создайте класс, в котором перегрузите оператор != через оператор ==.
9. Создайте класс со строковым атрибутом и перегрузите оператор !, чтобы проверить, пустая ли строка или нет.

**Глава 45: Операторы преобразования типов, индексирования и присваивания в C++**

В этой главе мы подробно разберем, как определять операторы преобразования типов, как перегружать оператор индексирования ([]), и как правильно переопределять оператор присваивания (в том числе как его удалять).

**45.1. Операторы преобразования типов (Type Conversion Operators)**

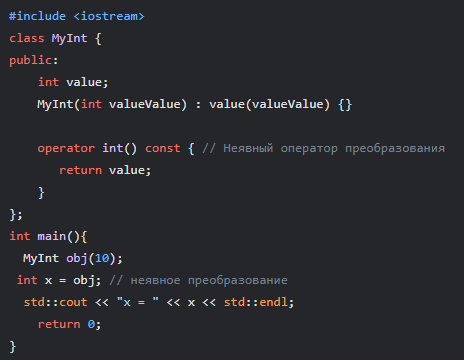
Операторы преобразования типов позволяют нам определять, как объект одного класса может быть преобразован в объект другого класса или в другой тип. Для этого используется ключевое слово operator и тип, к которому мы хотим преобразовать.

1. **Явные преобразования (explicit conversion)**  
   Для создания явного преобразования используется ключевое слово explicit. Оператор преобразования будет вызван только при явном преобразовании типов.

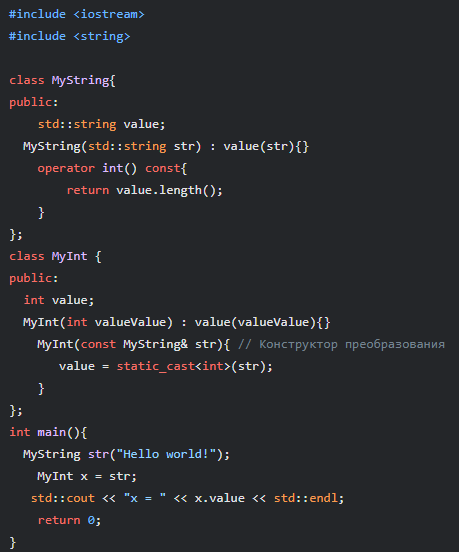


* + explicit operator int() const: объявляет оператор преобразования в int.
  + explicit: говорит компилятору, что не нужно использовать это преобразование неявно.
  + Оператор преобразования вызывается с помощью static\_cast<int>(obj).

1. **Неявные преобразования**  
   Если не использовать explicit то компилятор сможет использовать оператор преобразования неявно.



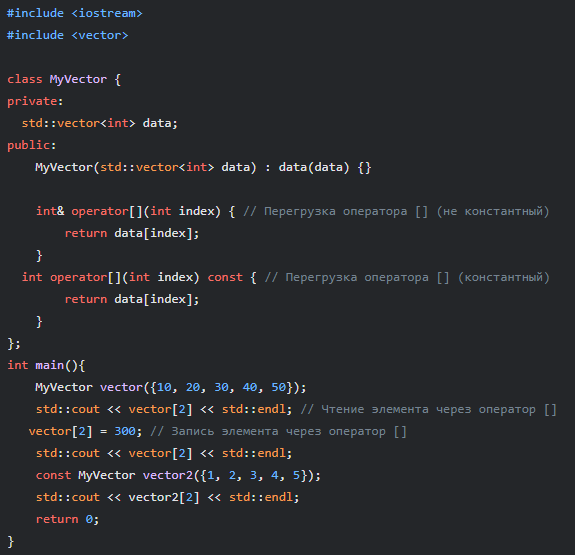
**45.2. Преобразование между классами**  
Можно преобразовать объект одного класса в объект другого класса.



* operator int() const - определяет неявный оператор преобразования класса MyString в int.
* MyInt(const MyString& str) - конструктор преобразования, который преобразует MyString в MyInt.

**45.3. Оператор индексирования ([])**

Оператор индексирования [] можно перегрузить, чтобы обеспечить доступ к элементам объекта как к элементам массива.

e

**Основные моменты:**

* int& operator[](int index) - перегрузка неконстантного оператора [], возвращает ссылку, что позволяет изменять элемент.
* int operator[](int index) const - перегрузка константного оператора [], возвращает константный элемент, что позволяет читать элемент для константного объекта.

**45.4. Переопределение оператора присваивания (=)**

Как мы уже видели в предыдущих главах, оператор присваивания можно переопределить, чтобы правильно обрабатывать динамическую память.

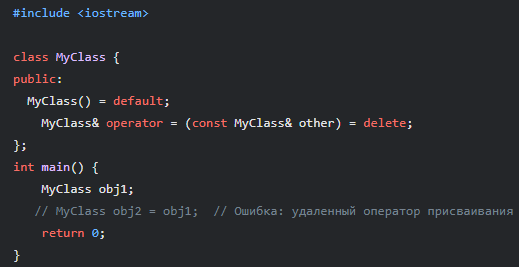


**Основные моменты:**

* MyString& operator = (const MyString& other) - перегрузка оператора присваивания.
* Оператор присваивания должен возвращать ссылку на текущий объект return \*this;.
* Необходимо предусмотреть проверку на самоприсваивание if(this != &other).
* Необходимо скопировать выделенную память, а не только указатель на нее.
* Необходимо освободить предыдущую память, если она была выделена.

**45.5. Удаление оператора присваивания**

Можно запретить компилятору генерировать оператор присваивания по умолчанию, объявив его со спецификатором = delete.



**Практические задания:**

1. Создайте класс Temperature, содержащий температуру в градусах Цельсия.
2. Перегрузите оператор преобразования типа для преобразования объекта Temperature в float, чтобы получить значение температуры в Фаренгейтах.
3. Создайте класс Array с динамически выделенной памятью.
4. Перегрузите оператор индексирования ([]) для доступа к элементам массива.
5. Реализуйте перегруженный оператор присваивания для корректного копирования массивов.
6. Создайте класс, у которого будет удален оператор присваивания и попробуйте присвоить объект этого класса.

**Глава 47: Вложенные классы в C++**

Вложенный класс — это класс, объявленный внутри другого класса. Вложенные классы являются средством структурирования кода и могут использоваться для реализации взаимосвязанных сущностей.

**47.1. Определение вложенного класса**

Для объявления вложенного класса, необходимо объявить класс внутри другого класса.



**Основные моменты:**

* class Inner объявлен внутри класса Outer.
* Inner является вложенным классом.
* Объект Inner может быть создан с использованием оператора разрешения области видимости :: (Outer::Inner inner).

**47.2. Область видимости вложенного класса**

* Вложенный класс Inner находится в области видимости класса Outer.
* Вложенный класс Inner может использовать имена и типы, которые доступны в классе Outer.
* Члены Inner не имеют автоматического доступа к членам Outer, необходимо использовать объекты или указатели.
* И наоборот, Outer может использовать Inner.



* void innerMethod(Outer& outer): Inner имеет доступ к Outer через переданный объект типа Outer.
* Метод outerMethod класса Outer создает объект класса Inner, и вызывает его метод передавая в качестве параметра \*this (текущий объект типа Outer).

**47.3. Модификаторы доступа и вложенные классы**

Вложенный класс может быть объявлен с любым модификатором доступа (public, private, protected) в классе, в котором он объявлен.



* private class Inner: делает класс Inner приватным для класса Outer, т.е. к этому классу можно получить доступ только изнутри класса Outer.

**47.4. Зачем нужны вложенные классы?**

* **Организация кода:** Вложенные классы помогают организовывать код, когда один класс тесно связан с другим.
* **Инкапсуляция:** Вложенные классы могут предоставлять более детальную инкапсуляцию, скрывая реализации и детали от остального кода.
* **Уменьшение количества глобальных имен:** Вложенные классы позволяют избежать загрязнения глобального пространства имен.
* **Доступ к внутренним данным:** Вложенные классы могут иметь доступ к private членам класса, в котором они вложены, и это может быть полезно для реализации сложных алгоритмов.

**Практические задания:**

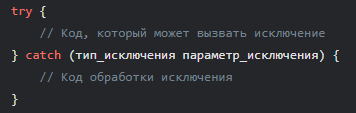
1. Создайте класс LinkedList, внутри которого определите класс Node для представления узла списка.
2. Создайте класс Graph, внутри которого определите класс Vertex для представления вершины графа.
3. Создайте класс Computer, внутри которого определите класс CPU и класс Memory для представления компонентов компьютера.
4. Создайте класс Outer с приватным вложенным классом Inner и создайте публичный метод для работы с приватным вложенным классом.
5. Попробуйте создать объект вложенного класса вне внешнего класса (проверьте модификаторы доступа).
6. Создайте вложенный класс, который может получить доступ к приватному атрибуту внешнего класса.

**Глава 48: Исключения в C++**

Исключения — это механизм в C++ для обработки ошибок и исключительных ситуаций, которые могут возникнуть во время выполнения программы. Когда происходит исключение, выполнение обычного потока программы прерывается, и управление передается блоку обработки исключений.

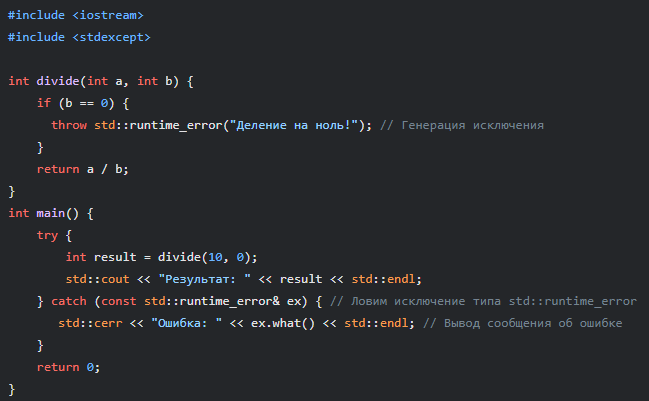
**48.1. Обработка исключений (try-catch)**

Для обработки исключений используются блоки try и catch:



* try: Блок кода, в котором может возникнуть исключение.
* catch: Блок кода, который обрабатывает исключение указанного типа.
* тип\_исключения: Тип исключения, которое должен обрабатывать блок catch.
* параметр\_исключения: Имя переменной, которая будет содержать информацию об исключении.

**Пример:**



**Основные моменты:**

* throw std::runtime\_error("Деление на ноль!");: генерирует исключение типа std::runtime\_error.
* catch (const std::runtime\_error& ex): обрабатывает исключение типа std::runtime\_error, используя ссылку на константу, чтобы не копировать объект исключения.
* ex.what(): метод объекта исключения, который возвращает описание ошибки.

**48.2. Создание объекта исключения**

Можно создавать свои собственные типы исключений путем наследования от базового класса исключений std::exception или от одного из его производных классов, например std::runtime\_error, std::logic\_error.



**Основные моменты:**

* class MyException : public std::runtime\_error: объявляет класс MyException производным от std::runtime\_error.
* MyException(std::string message) : std::runtime\_error(message): конструктор, вызывающий конструктор базового класса std::runtime\_error
* throw MyException("My custom exception - Деление на ноль!"): генерирует исключение типа MyException.

**48.3. Обработка и генерация разных типов исключений**  
Можно обрабатывать исключения разных типов в разных блоках catch.



**Основные моменты:**

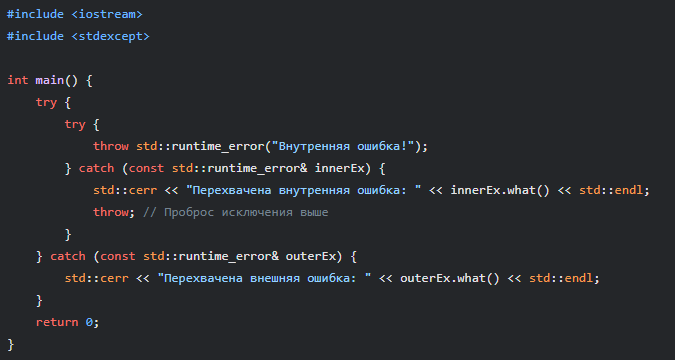
* Можно использовать несколько блоков catch, чтобы обрабатывать исключения разных типов.
* Блок catch будет вызван, только если исключение соответствует объявленному типу.
* Если ни один catch блок не обрабатывает исключение, то выполнение программы прервется.

**48.4. try-catch и деструкторы**

Если исключение генерируется в конструкторе, то деструктор объекта *не будет* вызван. Поэтому необходимо выделять ресурсы в блоке try, а освобождать в блоке catch, если деструктор не выполнится. Также можно использовать RAII (Resource Acquisition Is Initialization), где все ресурсы выделяются в конструкторе класса, а освобождаются в деструкторе, чтобы при возникновении исключения выделенные ресурсы были освобождены.

**48.5. Вложенные try-catch**

Можно вкладывать блоки try-catch друг в друга:



* Исключение сгенерированное во внутреннем блоке try перехватывается во внутреннем блоке catch.
* Оператор throw; перебрасывает пойманное исключение, в следующий блок catch который обрабатывает исключение этого типа.

**Практические задания:**

1. Напишите функцию, которая выбрасывает исключение, если передано некорректное значение аргумента.
2. Обработайте это исключение в main с помощью try-catch.
3. Создайте свой собственный тип исключения, унаследованный от std::runtime\_error.
4. Напишите функцию, которая может генерировать исключения разных типов в зависимости от разных условий.
5. Напишите функцию, которая динамически выделяет память и генерирует исключение, если выделение памяти не удалось. Используйте RAII для безопасной работы с выделенной памятью.
6. Продемонстрируйте работу вложенных блоков try-catch.
7. Создайте класс, в конструкторе которого может быть сгенерировано исключение, и проверьте, что деструктор не будет вызван.

**Глава 49: Создание своих типов исключений и типы исключений в C++**

В этой главе мы разберем, как создавать свои собственные типы исключений, и рассмотрим базовый класс std::exception и некоторые стандартные типы исключений.

**49.1. Создание своих типов исключений**

Для создания своего типа исключения обычно создается класс, который наследуется от std::exception или от одного из его производных классов (например, std::runtime\_error, std::logic\_error)



**Основные моменты:**

* class MyException : public std::runtime\_error: объявляет MyException как наследника от std::runtime\_error.
* class MyLogicError : public std::logic\_error: объявляет MyLogicError как наследника от std::logic\_error.
* MyException(std::string message) : std::runtime\_error(message): конструктор класса MyException вызывает конструктор базового класса std::runtime\_error и передает ему сообщение об ошибке.
* Теперь можно использовать MyException и MyLogicError как типы исключений и обрабатывать их отдельно.

**49.2. Базовый класс исключений std::exception**

std::exception — это базовый класс для всех исключений в C++. Он определяет базовый интерфейс для всех классов исключений и имеет метод what(), который возвращает описание исключения.

Большинство стандартных исключений являются производными от std::exception.

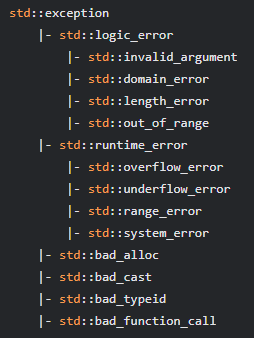
**49.3. Типы исключений**

Стандартная библиотека C++ предоставляет множество типов исключений, которые можно использовать в программах.

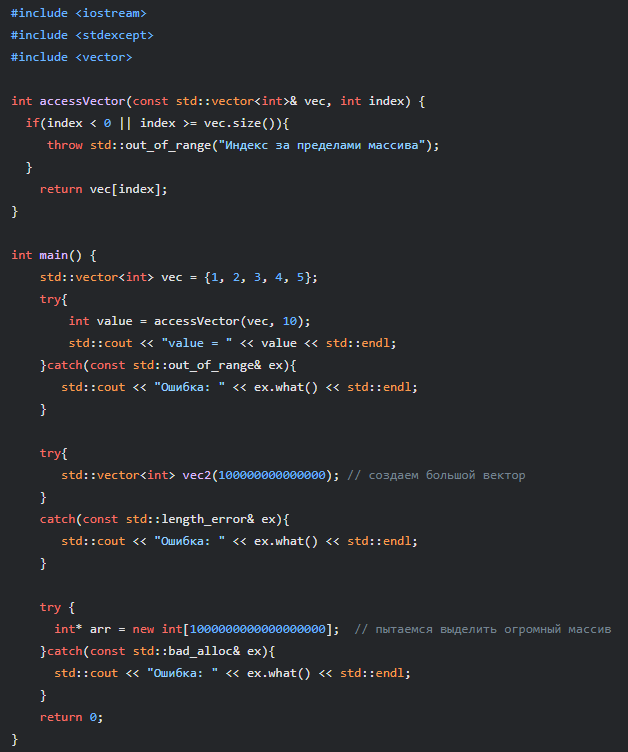
1. **Исключения времени выполнения (std::runtime\_error):**  
   Исключения, которые возникают во время выполнения программы.
   * std::runtime\_error: Общее исключение времени выполнения.
   * std::overflow\_error: Исключение, возникающее при переполнении.
   * std::underflow\_error: Исключение, возникающее при исчезновении разряда.
   * std::range\_error: Исключение, возникающее при выходе за границы диапазона.
   * std::system\_error: Исключение, возникающее при системной ошибке.
   * std::out\_of\_range: Исключение, возникающее при выходе за допустимый диапазон (например, при доступе к элементам массива).
2. **Логические ошибки (std::logic\_error):**  
   Исключения, которые указывают на логические ошибки в программе.
   * std::logic\_error: Общая логическая ошибка.
   * std::invalid\_argument: Исключение, возникающее при передаче недопустимого аргумента в функцию.
   * std::domain\_error: Исключение, возникающее при недопустимом значении в домене функции.
   * std::length\_error: Исключение, возникающее, когда создается объект, превышающий максимально допустимый размер.
   * std::out\_of\_range: Исключение, возникающее при доступе к элементу контейнера за пределами диапазона.
3. **Исключения для работы с памятью:**
   * std::bad\_alloc: Исключение, возникающее при неудачной попытке выделения динамической памяти.
4. **И другие:**
   * std::bad\_cast: Исключение, возникающее при неудачном dynamic\_cast преобразовании.
   * std::bad\_function\_call: Исключение, возникающее при попытке вызова функции, на которую не указывает указатель на функцию.
   * std::bad\_typeid: Исключение, возникающее при неправильном использовании typeid.

**49.4. Иерархия исключений:**

В C++ исключения образуют иерархию наследования:



**49.5. Пример использования стандартных исключений:**



* В примере показано использование стандартных исключений std::out\_of\_range, std::length\_error, и std::bad\_alloc
* При генерации исключения соответствующего типа, будет вызван соответствующий блок catch.

**Практические задания:**

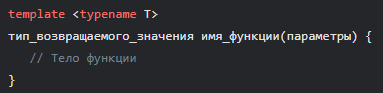
1. Создайте свой класс исключения для обработки ошибок ввода данных.
2. Создайте класс исключения, который имеет дополнительный атрибут для хранения кода ошибки.
3. Напишите функцию, которая генерирует несколько различных типов исключений.
4. Напишите программу, которая обрабатывает все стандартные исключения, которые мы рассматривали в этой главе.
5. Создайте иерархию исключений, где пользовательские исключения наследуются от std::runtime\_error или std::logic\_error.
6. Протестируйте как работает std::bad\_alloc.
7. Протестируйте как работает std::bad\_cast.
8. Напишите программу, которая обрабатывает исключение std::out\_of\_range, когда происходит попытка доступа к элементу вектора вне диапазона.

**Глава 50: Шаблоны функций в C++**

Шаблон функции (function template) — это описание функции, которая может работать с разными типами данных. Компилятор использует шаблон для создания конкретных версий функции для каждого используемого типа.

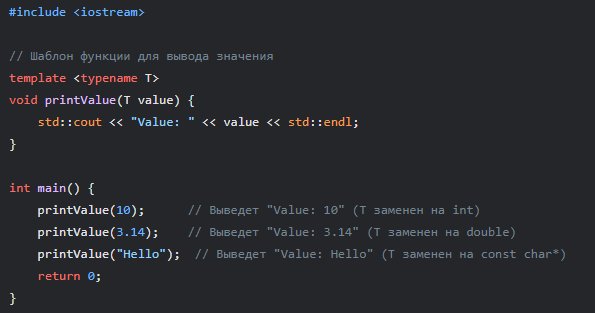
**50.1. Определение шаблона функции**

Для объявления шаблона функции используется ключевое слово template и список параметров шаблона, заключенный в угловые скобки <>.



* template <typename T>: объявляет шаблон с параметром типа T. Вместо typename можно использовать class, это одно и тоже.
* T: это имя параметра типа, который будет заменен на конкретный тип при использовании шаблона.
* тип\_возвращаемого\_значения: тип возвращаемого значения функции.
* имя\_функции: имя функции.
* параметры: параметры функции.

**Пример:**

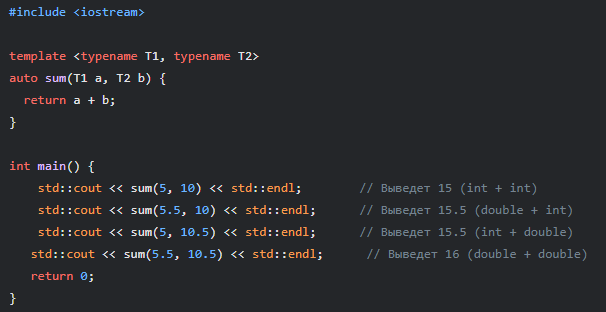


**Основные моменты:**

* template <typename T>: объявляет шаблон функции, в которой T — это параметр типа.
* При вызове функции printValue компилятор создает версии функции для типов int, double, const char\*.
* Шаблонная функция может работать с разными типами данных без необходимости дублирования кода.

**50.2. Шаблоны функций с несколькими параметрами**

Шаблон функции может иметь несколько параметров типа:

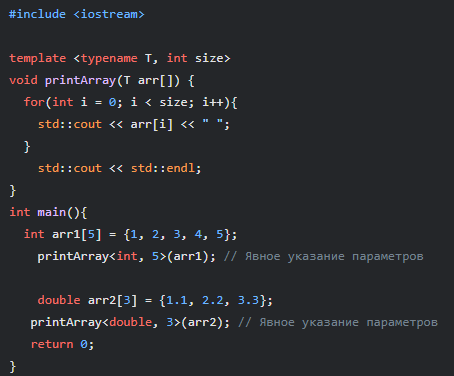


**Основные моменты:**

* template <typename T1, typename T2>: объявляет шаблон с двумя параметрами типа T1 и T2.
* auto sum(T1 a, T2 b) - автоматически выводит тип возвращаемого значения на основе типов параметров.

**50.3. Шаблоны функций с параметрами не типа**

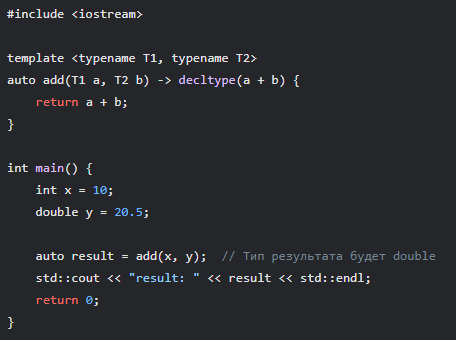
Можно объявлять параметры не типа, значения которых будут известны во время компиляции (например, int, char, указатели).



**Основные моменты:**

* template <typename T, int size>: объявляет шаблон с параметром типа T и нетиповым параметром size.
* printArray<int, 5>(arr1);: явное указание параметров шаблона, компилятор создаст функцию, работающую для массива типа int и размера 5.
* Значение нетипового параметра должно быть известно во время компиляции.

**50.4. Использование auto и decltype с шаблонами**  
auto может использоваться для автоматического выведения типа возвращаемого значения.  
decltype позволяет получить тип выражения во время компиляции.



**Основные моменты:**

* auto add(T1 a, T2 b) -> decltype(a + b): тип возвращаемого значения выводится компилятором, с использованием decltype(a + b), который определяет тип значения выражения a + b.

**50.5. Преимущества шаблонов функций**

* **Обобщенность:** Шаблоны функций позволяют писать код, который работает с разными типами данных, не дублируя код.
* **Безопасность:** Шаблоны функций обеспечивают типовую безопасность на этапе компиляции, так как компилятор проверяет, что типы используются правильно.
* **Производительность:** Шаблоны функций не создают накладных расходов во время выполнения, так как компилятор генерирует конкретные версии функций во время компиляции.
* **Удобство использования:** Шаблонная функция может быть вызвана как обычная, без явного указания типов, в большинстве случаев компилятор может самостоятельно вывести тип на основе переданных аргументов.

**Практические задания:**

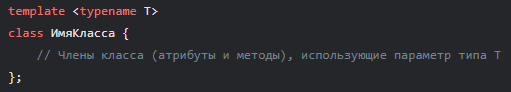
1. Создайте шаблон функции для поиска максимального значения в массиве.
2. Создайте шаблон функции для обмена значений двух переменных.
3. Создайте шаблон функции для вычисления среднего арифметического значения элементов массива.
4. Создайте шаблон функции, которая принимает два аргумента и возвращает их произведение.
5. Создайте шаблон функции для вывода массива на экран, с разным типом данных.
6. Используйте auto и decltype для автоматического выведения типа возвращаемого значения шаблонной функции.
7. Попробуйте создать шаблон, который принимает параметр типа и нетиповой параметр.

**Глава 51: Шаблоны классов в C++**

Шаблон класса (class template) — это описание класса, который может работать с разными типами данных. Компилятор использует шаблон класса для создания конкретных версий класса для каждого используемого типа.

**51.1. Определение шаблона класса**

Для объявления шаблона класса используется ключевое слово template и список параметров шаблона, заключенный в угловые скобки <>.



* template <typename T>: объявляет шаблон класса с параметром типа T. Вместо typename можно использовать ключевое слово class, это одно и тоже.
* T: параметр типа, который будет заменен на конкретный тип при создании объекта класса.
* ИмяКласса: имя класса.

**Пример:**

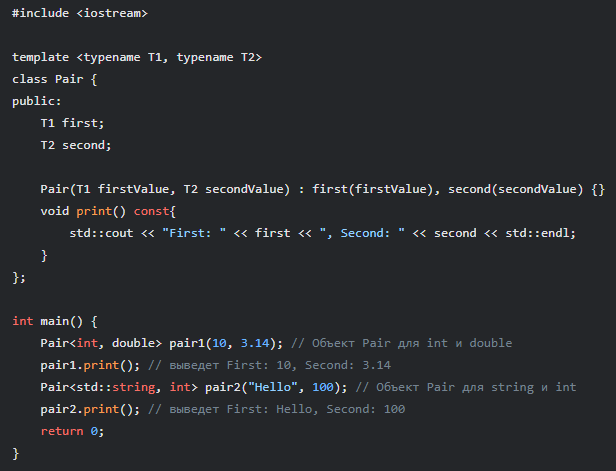


**Основные моменты:**

* template <typename T>: объявляет шаблон класса с параметром типа T.
* При создании объекта MyArray<int>, компилятор создает класс MyArray с типом int, в котором T заменяется на int.
* При создании объекта MyArray<std::string>, компилятор создает класс MyArray с типом std::string, в котором T заменяется на std::string.
* Шаблонный класс может работать с любым типом данных, без необходимости дублирования кода.

**51.2. Шаблон класса с несколькими параметрами**

Шаблон класса может иметь несколько параметров типа.



* template <typename T1, typename T2>: объявляет шаблон класса с двумя параметрами типа.

**51.3. Специализация шаблона класса**

Шаблоны классов могут быть специализированными для определенных типов данных. Это означает, что можно создать отдельную версию шаблонного класса, которая работает по-другому для конкретного типа данных.

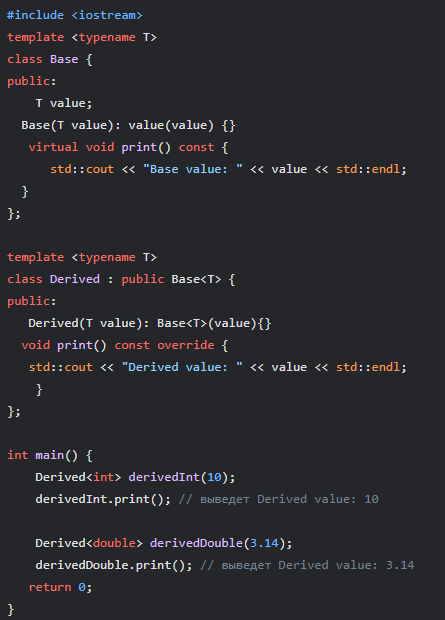


**Основные моменты:**

* template <> class MyClass<int> { ... };: объявляет специализацию шаблона для типа int.
* Когда компилятор встречает MyClass<int>, он использует специализированную версию класса.
* Специализация шаблона используется, когда нужно обеспечить специфичное поведение для конкретного типа.

**51.4. Наследование и шаблоны классов**

Можно создавать шаблоны классов, наследуемые от других шаблонов классов.



**Основные моменты:**

* class Derived : public Base<T>: шаблон класса Derived наследуется от шаблон класса Base, передавая тип параметра шаблона.
* При создании производного класса, ему также необходимо указать тип параметра шаблона.

**Практические задания:**

1. Создайте шаблон класса Stack для реализации стека с методами push, pop, peek и isEmpty.
2. Создайте шаблон класса Queue для реализации очереди с методами enqueue, dequeue, front и isEmpty.
3. Создайте шаблон класса Pair для хранения пары значений с разными типами данных.
4. Создайте специализированную версию шаблона MyClass для типа std::string.
5. Создайте шаблонный класс для представления матрицы, и используйте специализацию для int и float.
6. Создайте базовый шаблонный класс и производный от него шаблонный класс и попробуйте вызывать их методы.