Сафошин А.С. ИВТ- 23.

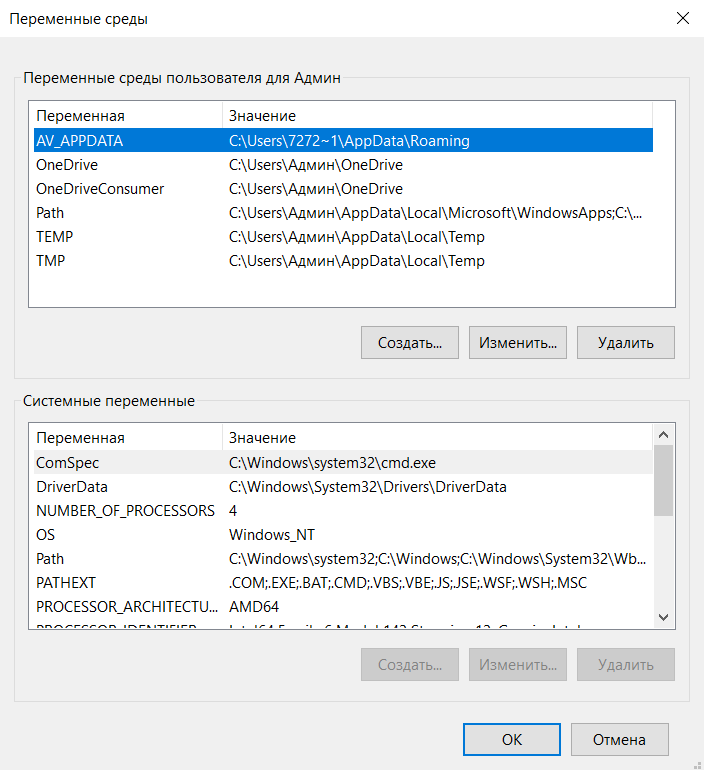
**Компилирование программ в консоли (PowerShell)**

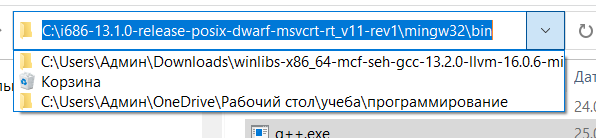
## Порядок установки компилятора MinGW64:

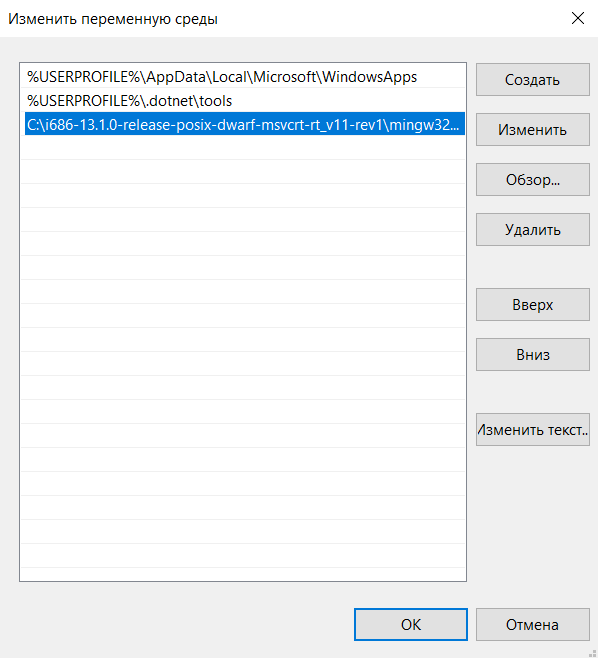
1. Открываем <https://www.mingw-w64.org/downloads/>
2. Выбираем MingW-W64-builds
3. Находим Installation: GitHub и переходим по ссылке
4. Скачиваем [i686-13.1.0-release-posix-dwarf-msvcrt-rt\_v11-rev1.7z](https://github.com/niXman/mingw-builds-binaries/releases/download/13.1.0-rt_v11-rev1/i686-13.1.0-release-posix-dwarf-msvcrt-rt_v11-rev1.7z)
5. Извлекаем файлы из архива

## Подключение

Для того, чтобы подключить компилятор, нужно добавить путь к нему в переменную среды окружения Path:

1. Открываем «Изменение переменных среды текущего пользователя»
2. Выбираем переменную Path и нажимаем «Изменить»
3. Копируем путь к файлу и добавляем в пустую ячейку переменной Path

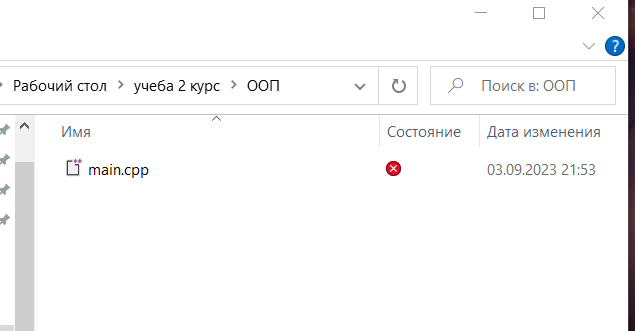


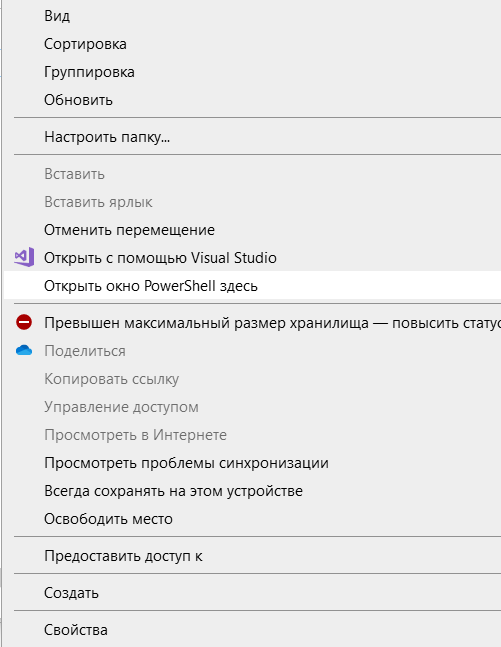
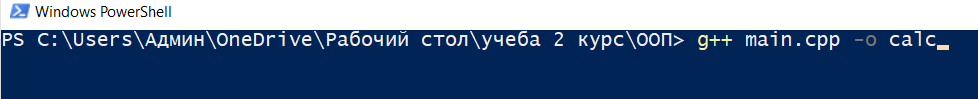


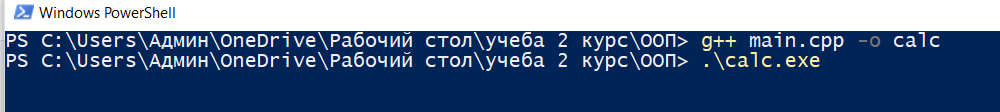
## Компиляция

Теперь мы можем компилировать программы, для этого:

1. Создаём в Sublime Text (или другом подобном приложении) файл с необходимым код и сохраняем в отдельной папке с расширением cpp

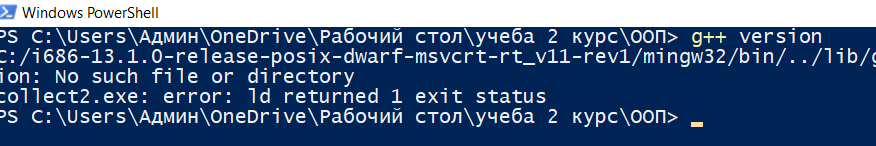


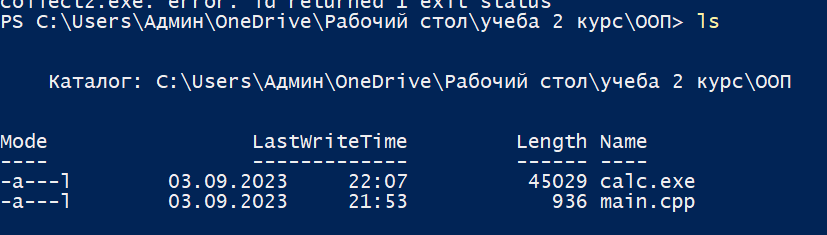
1. Открываем PowerShell нажатием Shift и правой кнопки мыши в любом свободном месте папки
2. Создаём exe файл программы с помощью аргумента –o (название файла)
3. Вызываем и запускаем программу, можно ввести лишь название файла и нажать клавишу Tab, можно ввести вручную

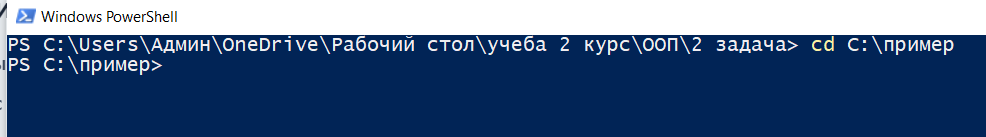


Для того чтобы переключаться между последними командами нужно использовать стрелочки вверх и вниз

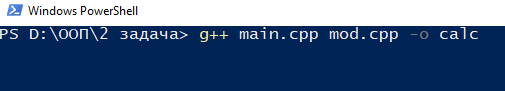
Аргумент version выдаёт нам версию компилятора



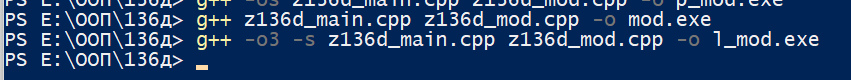
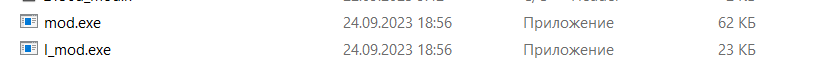
Команда ls используется для составления списка файлов и каталогов

Команда cd изменяет папку, с которой мы работаем

Для компиляции нескольких файлов нужно указать имя дополнительного файла после имени основного



С помощью опций–o3 и –s мы можем уменьшить размер компилируемого файла в несколько раз

Так мы смогли изменить размер файла с 62 до 23 КБ

## Статическая библиотека

**Библиотека** – это пакет кода, который предназначен для повторного использования многими программами. Обычно библиотека C++ состоит из двух частей:

1. заголовочный файл, который определяет функциональность, которую библиотека предоставляет (предлагает) программам, использующим ее;
2. предварительно скомпилированный двоичный файл, который содержит реализацию этой функциональности, предварительно скомпилированную в машинный код.

Некоторые библиотеки могут быть разделены на несколько файлов и/или иметь несколько файлов заголовков.

**Статическая библиотека** (иногда называемая archive, "архив") состоит из подпрограмм, которые скомпилированы и связываются непосредственно с вашей программой. Когда вы компилируете программу, использующую статическую библиотеку, все функции статической библиотеки, которые использует ваша программа, становятся частью вашего исполняемого файла. В Windows статические библиотеки обычно имеют расширение .lib, а в Linux – расширение .a (archive, архив).

Одним из преимуществ статических библиотек является то, что вам нужно распространять только исполняемый файл, чтобы пользователи могли запускать вашу программу. Поскольку библиотека становится частью вашей программы, это гарантирует, что с вашей программой всегда будет использоваться правильная версия библиотеки. Кроме того, поскольку статические библиотеки становятся частью вашей программы, вы можете использовать их так же, как функции, которые вы написали для своей программы. С другой стороны, поскольку копия библиотеки становится частью каждого исполняемого файла, который ее использует, это может привести к потере большого количества места. Статические библиотеки также не могут быть легко обновлены – для обновления библиотеки необходимо заменить весь исполняемый файл.

Для создания статической библиотеки нужно заранее подготовить папку lib

Создание статический библиотеки:

g ++ -c random\_names.cpp -o библиотека /random\_names.o

ar rcs lib/random\_names.библиотека / random\_names.o

В итоге имеем два файла библиотеки:

* random\_names.h
* random\_names.a -- скомпилированный cpp файл.

[Компиляция программы с включением статической библиотеки](https://github.com/VetrovSV/OOP/tree/master/examples/example_libs/simple_lib#%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B-%D1%81-%D0%B2%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B5%D0%BC-%D1%81%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B9-%D0%)

Статическая библиотека при компиляции указывается в ряду остальных файлов исходного кода

g ++ main.cpp библиотека/random\_names.a -o main.exe

Источники:

[Что такое PowerShell? - PowerShell | Microsoft Learn](https://learn.microsoft.com/ru-ru/powershell/scripting/overview?view=powershell-7.3)

[Вопросы производительности сценариев PowerShell - PowerShell | Microsoft Learn](https://learn.microsoft.com/ru-ru/powershell/scripting/dev-cross-plat/performance/script-authoring-considerations?view=powershell-7.3)

<https://github.com/VetrovSV/OOP/tree/master/examples/example_libs/simple_lib#компиляция-статический-библиотеки>

# **Работа с аргументами командной строки в программах на языке C++**

Аргументы командной строки очень полезны, если вы хотите передать любые входные строки в свою основную программу из командной строки.

Эти аргументы передаются в качестве параметров функции main(). Рассмотрим, как их можно эффективно использовать.

Часто нам очень удобно напрямую вводить данные в нашу программу. Одним из распространенных способов является использование scanf() или getchar() и т.п. для ожидания ввода данных пользователем.

Но эти вызовы тратят много времени на ожидание и требуют, чтобы пользователь вводил данные вручную.

Мы можем сэкономить много времени, просто передав эти данные в нашу основную программу.

Формат будет чем-то вроде:

./executable input1 input2

Программа автоматически сохранит эти аргументы командной строки в специальных переменных, из которых мы можем получить к ним прямой доступ.

Для этого потребуется только один раз ввести их при запуске программы. Давайте посмотрим, как мы можем использовать их в таком случае.

**Специальные переменные**

Программа передаст аргументы командной строки в функцию main().

В С/С++ функция main() принимает два дополнительных параметра для этих аргументов.

* argc -> Количество аргументов (Argument count). Дает количество аргументов, которые мы передаем (включая имя программы).
* argv -> Argument vector. Это массив строк char\*. Это сами значения аргументов.

Таким образом, argv[0] - это название самой программы, а argv[1] … argv[argc-1] - это все наши аргументы командной строки.

int main(int argc, char\* argv[]);

Чтобы увидеть это в действии, давайте рассмотрим пример.

**Пример использования**

Давайте рассмотрим программу, которая объединяет две строки, заданные в качестве входных данных.

Мы передадим в нашу программу два аргумента командной строки, поэтому общее argc должно быть 3 (включая имя программы).

Мы можем написать нашу программу так:

#include <iostream>

#include <string>

using namespace std;

string concat\_strings(string s1, string s2) {

return s1 + s2;

}

int main(int argc, char\* argv[])

{

cout << "You have entered " << argc

<< " arguments:" << "\n";

if (argc != 3) {

cerr << "Program is of the form: " << argv[0] << " <inp1> <inp2>\n";

return 1;

}

string result = concat\_strings(argv[1], argv[2]);

cout << "Result: " << result << endl;

return 0;

}

Если исполняемое имя **test.out**, на компьютере с Linux запускаем исполняемый файл с помощью этой команды:

./test.out Hello \_ArduinoPlus

Обратите внимание, что аргументы разделены пробелом. Таким образом у нас аргументы командной строки: «Hello» и «\_ArduinoPlus».

На выходе получаем:

You have entered 3 arguments:

Result: Hello\_ArduinoPlus

Отлично! Похоже, что всё работает, как и ожидалось, так как первый аргумент - название самой программы.

Попробуем теперь запустить программу с 4 аргументами.

./test.out Hello from ArduinoPlus

На выходе получаем:

You have entered 4 arguments:

Program is of the form: ./test.out <inp1> <inp2>

Источники:

https://arduinoplus.ru/argumenty-komandnoj-stroki-v-c-c/

**Этапы компиляции на С++**

1) Препроцессинг

Препроцессор — это *макро процессор*, который преобразовывает вашу программу для дальнейшего компилирования. На данной стадии происходит происходит работа с препроцессорными директивами. Например, препроцессор добавляет хэдеры в код (#include), убирает комментирования, заменяет макросы (#define) их значениями, выбирает нужные куски кода в соответствии с условиями #if, #ifdef и #ifndef.

Хэдеры, включенные в программу с помощью директивы #include, рекурсивно проходят стадию препроцессинга и включаются в выпускаемый файл. Однако, каждый хэдер может быть открыт во время препроцессинга несколько раз, поэтому, обычно, используются специальные препроцессорные директивы, предохраняющие от циклической зависимости.

Получим препроцессированный код в выходной файл driver.ii (прошедшие через стадию препроцессинга C++ файлы имеют расширение .ii), используя флаг -E, который сообщает компилятору, что компилировать (об этом далее) файл не нужно, а только провести его препроцессинг:

g++ -E driver.cpp -o driver.ii

Взглянув на тело функции *main* в новом сгенерированном файле, можно заметить, что макрос RETURN был заменен:

**int** main() {

cout << "Hello, world!" << endl;

**return** 0;

}

В новом сгенерированном файле также можно увидеть огромное количество новых строк, это различные библиотеки и хэдер iostream.

### 2) Трансляция

На данном шаге g++ выполняет свою главную задачу — компилирует, то есть преобразует полученный на прошлом шаге код без директив в *ассемблерный код*. Это промежуточный шаг между высокоуровневым языком и машинным (бинарным) кодом.

Ассемблерный код — это доступное для понимания человеком представление машинного кода.

Используя флаг -S, который сообщает компилятору остановиться после стадии компиляции, получим ассемблерный код в выходном файле driver.s:

$ g++ -S driver.ii -o driver.s

Мы можем все также посмотреть и прочесть полученный результат. Но для того, чтобы машина поняла наш код, требуется преобразовать его в машинный код, который мы и получим на следующем шаге.

### 3) Ассемблирование

Так как x86 процессоры исполняют команды на бинарном коде, необходимо перевести ассемблерный код в машинный с помощью ассемблера.

Ассемблер преобразовывает ассемблерный код в машинный код, сохраняя его в *объектном файле*.

Объектный файл — это созданный ассемблером промежуточный файл, хранящий кусок машинного кода. Этот кусок машинного кода, который еще не был связан вместе с другими кусками машинного кода в конечную выполняемую программу, называется *объектным кодом*.

Далее возможно сохранение данного объектного кода в *статические библиотеки* для того, чтобы не компилировать данный код снова.

Получим машинный код с помощью ассемблера (as) в выходной объектный файл driver.o:

$ as driver.s -o driver.o

Но на данном шаге еще ничего не закончено, ведь объектных файлов может быть много и нужно их всех соединить в единый исполняемый файл с помощью компоновщика (линкера). Поэтому мы переходим к следующей стадии.

### 4) Компоновка

Компоновщик (линкер) связывает все объектные файлы и статические библиотеки в единый исполняемый файл, который мы и сможем запустить в дальнейшем. Для того, чтобы понять как происходит связка, следует рассказать о *таблице символов*.

Таблица символов — это структура данных, создаваемая самим компилятором и хранящаяся в самих объектных файлах. Таблица символов хранит имена переменных, функций, классов, объектов и т.д., где каждому идентификатору (символу) соотносится его тип, область видимости. Также таблица символов хранит адреса ссылок на данные и процедуры в других объектных файлах.  
Именно с помощью таблицы символов и хранящихся в них ссылок линкер будет способен в дальнейшем построить связи между данными среди множества других объектных файлов и создать единый исполняемый файл из них.

Получим исполняемый файл driver:

$ g++ driver.o -o driver // также тут можно добавить и другие объектные файлы и библиотеки

### 5) Загрузка

Последний этап, который предстоит пройти нашей программе — вызвать загрузчик для загрузки нашей программы в память. На данной стадии также возможна подгрузка *динамических библиотек*.

Запустим нашу программу:

$ ./driver

// Hello, world!

Источники:

[Процесс компиляции программ на C++ / Хабр](https://habr.com/ru/articles/478124/)

[О GCC, компиляции и библиотеках](https://uzverss.livejournal.com/57719.html)

**Стандарты оформления кода Yandex**

### Политика исправления стиля

* Общее правило исправления стиля в старом коде – "один файл - один стиль". Другими словами, если вы исправляете ошибку, которая включает изменения всего в нескольких строках, следуйте оригинальному стилю в файле. Если исправленный код написан в новом стиле, то вам необходимо изменить весь код во всем файле для обеспечения согласованности, а в некоторых случаях даже весь код для класса (например, если переименованы поля или методы).
* Если автор кода не согласен со стилем исправлений, он может переписать его, но авторские изменения должны соответствовать стилю, описанному в этом руководстве (включая предыдущий пункт).
* Не меняйте стиль в файлах, которые вы не изменяете, и не меняйте форматирование строк, если оно соответствует рекомендациям этого руководства. Не забывайте, что у каждой строки есть автор, и это отслеживается в журнале VCS (например, svn, git, hg и т.д.)

## **Названия**

Название должно отражать суть данных, тип или действие, которые оно обозначает. В названиях допускаются только общеупотребительные сокращения. Обычные однобуквенные имена (i, j, k) разрешены только для счетчиков и итераторов. Структуры - это тоже классы, и все, что связано с классами, применимо и к структурам (если явно не указано иное).

### **Переменные**

#### **Заглавные буквы, префиксы и подчеркивания**

* Локальные и глобальные переменные начинаются со строчной буквы.
* Названия функций начинаются с заглавной буквы.
* Указатели на функции, как и обычные переменные, начинаются со строчной буквы: auto localFunction = [&]() { ... }
* Аргументы функции начинаются со строчной буквы.
* Участники занятий начинаются с заглавной буквы.
* Методы класса начинаются с заглавной буквы.
* Именам классов и определениям типов (typedefs) предшествует префикс T, за которым следует название класса, начинающееся с заглавной буквы. Имена виртуальных интерфейсов начинаются с "I".
* Все глобальные константы и определения полностью пишутся с заглавной буквы.
* Маркеры в сложных именах переменных и функций различаются заглавной первой буквой маркера (без подчеркивания). Маркеры в полностью заглавных именах констант разделяются символами подчеркивания.
* Использование подчеркивания в качестве первого символа имени запрещено.
* Использование венгерских обозначений запрещено.

*Исключение:* имена функций, классов и так далее, которые имитируют или расширяют функции стандартных библиотек (libc, stl и т.д.), должны соответствовать соглашению об именовании библиотек. Примерами являются yvector, fget, autoarray, sprintf, эквиваленты функции main . Эти классы и функции обычно находятся в /util.

#### 

#### **Семантика и прагматика именования**

Постарайтесь убедиться, что программа понятна на английском языке, то есть она напоминает связный и осмысленный английский текст:

* Первый маркер в имени функции должен отражать действие, выполняемое функцией.
* Имена переменных контейнера - множественное число.

Для переменных счетчика не используйте имена DocNum, NumDoc, DocsCount, DocsNum и CountDoc, поскольку они неграмотны и неоднозначны. Для количества элементов (например, документов) можно использовать NumDocs или DocCount. Для функции, которая явно подсчитывает это число в течение длительного времени, CountDocs() приемлемо.

## **Форматирование**

### **Вкладки**

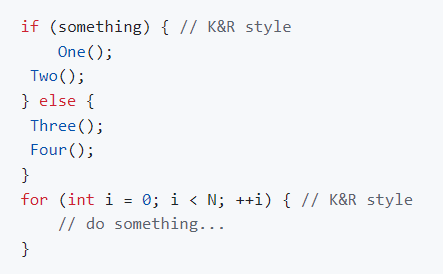
Не используйте вкладки в текстовом редакторе. Причина в том, что это единственный способ гарантировать, что ваша программа читаема на любом устройстве. Убедитесь, что в вашем текстовом редакторе есть опция замены символа табуляции пробелами. Например, в редакторе текстовой панели выберите опцию "Преобразовать новые табуляции в пробелы".

### **Отступы**

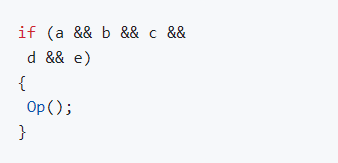
Наш стандартный отступ состоит из 4 пробелов. Отступ должен быть заполнен пробелами, даже если вы используете кнопку Tab.

### **Стиль блока**

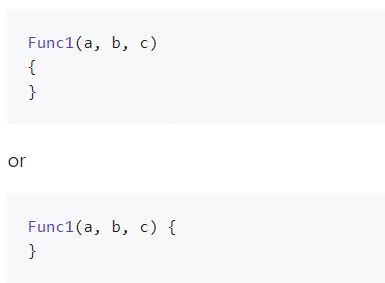
Для операторов блоков используйте стиль 1TBS:



Многострочные условия являются исключением (если условие не умещается в одной строке, разделите его на несколько), и они записываются следующим образом:



Для функций и методов можно использовать любой из двух стилей:



**Операторы**

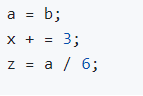
Не используйте более одного оператора в строке.

**Пустые строки**

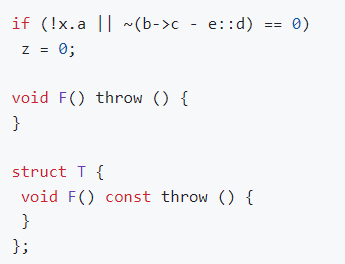
Мы рекомендовали оставлять пустые строки между отдельными логическими блоками кода. Это значительно улучшает читаемость.

#### **Символы оператора**

Все символы операторов, за исключением унарных операторов и оператора доступа к элементам для структур, должны иметь пробелы с обеих сторон:



Сюда входит оператор присваивания. Другими словами, напишите:

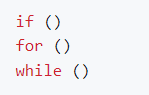


#### **Скобки**

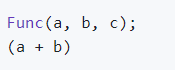
Не ставьте пробел после имени функции, после открывающей круглой скобки или перед закрывающей скобкой:



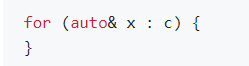
Обязательно ставьте пробел между оператором и скобкой:



Пробелы внутри скобок должны выглядеть следующим образом:



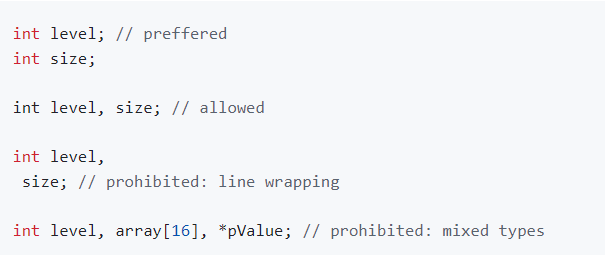
Внутри диапазона, основанного на:



## **Переменные и классы**

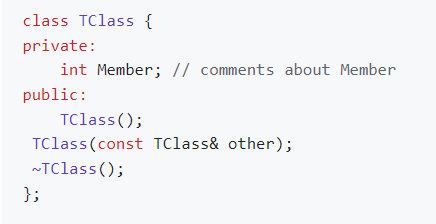
### **Объявления переменных**

Предпочтительный формат - "одно объявление в строке". Допускается объявлять несколько переменных одного типа в одной строке. Запрещается смешивать массивы, указатели, ссылки и простые типы. Не используйте перенос строк в объявлении.



### **Объявления классов и структур**

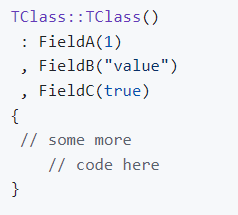
* Структура может содержать только открытые элементы. Вам не нужно указывать public для этого. Если структура содержит что-либо, кроме элементов, конструктора и деструктора, мы рекомендуем вам переименовать ее в класс.
* Метки областей начинаются с того же столбца, с которого начинается объявление класса. Указание областей является обязательным, включая первую закрытую область.
* Элементы и методы не могут находиться в одном разделе областей. Их следует разделить, повторно указав область. Должно быть минимальное количество меток области видимости, сокращенное до минимально возможного путем изменения порядка частей объявления класса.
* В рамках одной области:
  + Конструкторы должны предшествовать деструктору.
  + Деструктор должен предшествовать переопределенным операторам.
  + Переопределенные операторы должны предшествовать остальным методам.
* Общедоступная область с методами должна предшествовать protected и private областям с методами.
* Элементы данных класса следует размещать в начале или в конце описания класса. Описания типов классов могут предшествовать описаниям данных.



### 

### **Конструкторы**

Конструкторы должны быть отформатированы следующим образом:



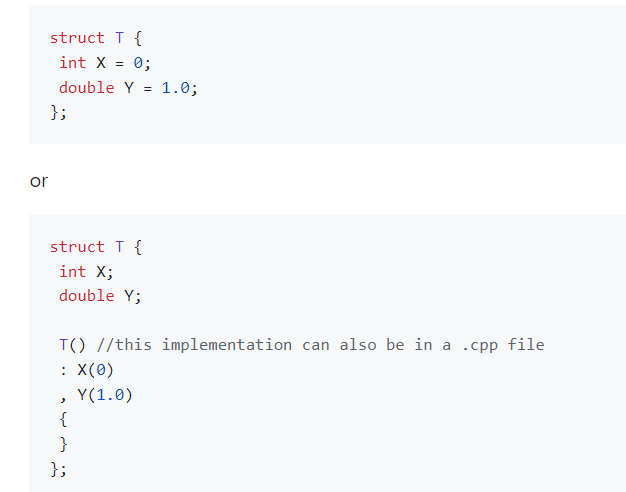
### 

### 

### 

### **Инициализация данных класса**

Допускается один из следующих вариантов:



## **Комментарии**

Комментарии предназначены для пояснения кода, в котором они расположены. Не используйте комментарии для удаления ненужной функции или блока, особенно если это старая версия функции, которую вы исправили. Просто удалите все ненужные части кода – вы всегда можете перейти к VCS (например, svn, git, hg и т.д.), Чтобы восстановить удаленный раздел, если вдруг поймете, насколько он был полезен. Основной вред от комментирования предыдущих версий кода вместо их удаления заключается в том, что VCS diff не будет работать корректно.

Комментарии должны быть написаны на английском языке с правильной орфографией и грамматикой.

Полезно объяснить назначение каждого члена класса в описании класса. Редактор MSVC отображает эту строку во всплывающей подсказке в режиме "интеллектуального редактирования".

Комментарии в стиле Doxygen приветствуются.

Чтобы упростить поиск ваших комментариев к TODO в коде, используйте один из двух форматов:

// Предположительно временный комментарий с пометками для себя:

// ЗАДАЧА (имя пользователя): исправьте меня позже

// Комментарий с тикетом:

// Задача (ticket\_number): исправьте меня позже

Источники:

<https://github.com/yandex/CMICOT/blob/master/CPP_STYLE_GUIDE.md>

# **Операторы try, throw и catch**

Для реализации обработки исключений в C++ используются tryвыражения, throw и catch.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <iostream>    double divide(int a, int b)  {  return a / b;  }    int main()  {  int x{500};  int y{};  double z {divide(x, y)};    std::cout << z << std::endl;  std::cout << "The End..." << std::endl;  } |

Во-первых, используйте try блок для заключения одной или нескольких инструкций, которые могут вызвать исключение.

Выражение throw сигнализирует о том, что в try блоке произошло исключительное условие ( часто ошибка). В качестве операнда throw выражения можно использовать объект любого типа. Обычно этот объект используется для передачи информации об ошибке. В большинстве случаев рекомендуется использовать [std::exception](https://learn.microsoft.com/ru-ru/cpp/standard-library/exception-class?view=msvc-170) класс или один из производных классов, определенных в стандартной библиотеке. Если один из них не подходит, рекомендуется создать собственный класс исключений из std::exception.

Для обработки исключений, которые могут возникнуть, реализуйте один или несколько catch блоков сразу после try блока. Каждый catch блок указывает тип исключения, которое он может обработать.

Например, в следующей программе происходит деление чисел:

Эта программа успешно скомпилируется, но при ее выполнении возникнет ошибка, поскольку в коде производится деление на ноль, после чего программа аварийно завершится.

С одной стороны, мы можем в функции divide определить проверку и выполнять деление, если параметр b не равен 0. Однако нам в любом случае надо возвращать из функции divide некоторый результат - некоторое число. То есть мы не можем просто написать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | double divide(int a, int b)  {  if (b)  return a / b;  else  std::cout << "Error! b must not be equal to 0" << std::endl;  } |

И в этом случае нам надо известить систему о возникшей ошибке. Для этого используется оператор throw.

Оператор throw генерирует исключение. Через оператор throw можно передать информацию об ошибке. Например, функция divide могла бы выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | double divide(int a, int b)  {  if (b)  return a / b;  throw "Division by zero!";  } |

То есть если параметр b равен 0, то генерируем исключение.

Но это исключение еще надо обработать в коде, где будет вызываться функция divide. Для обработки исключений применяется конструкция try...catch. Она имеет следующую форму:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | try  {  инструкции, которые могут вызвать исключение  }  catch(объявление\_исключения)  {  обработка исключения  } |

В блок кода после ключевого слова try помещается код, который потенциально может сгенерировать исключение.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #include <iostream>    double divide(int a, int b)  {  if (b)  return a / b;  throw "Division by zero!";  }    int main()  {  int x{500};  int y{};    try  {  double z {divide(x, y)};  std::cout << z << std::endl;  }  catch (...)  {  std::cout << "Error!" << std::endl;  }  std::cout << "The End..." << std::endl;  } |

После ключевого слова catch в скобках идет параметр, который передает информацию об исключении. Затем в блоке производится собственно обработка исключения.

Так изменим весь код следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include >iostream<  double divide(int a, int b)  {  if (b)  return a / b;  throw "Division by zero!";  }  int main()  {  int x{500};  int y{};  try  {  double z {divide(x, y)};  std::cout << z << std::endl;  }  catch (const char\* error\_message)  {  std::cout << error\_message << std::endl;  }  std::cout << "The End..." << std::endl; |

Код, который потенциально может сгенерировать исключение - вызов функции divide помещается в блок try.

В блоке catch идет обработка исключения. Причем многоточие в скобках после оператора catch (catch(...)) позволяет обработать любое исключение.

В итоге когда выполнение программы дойдет до строки

double z {divide(x, y)};

При выполнении этой строки будет сгенерировано исключение, поэтому последующие инструкции из блока try выполняться не будут, а управление перейдет в блок catch, в котором на консоль просто выводится сообщение об ошибке. После выполнения блока catch программа аварийно не завершится, а продолжит свою работу, выполняя операторы после блока catch:

Error!

The End...

Однако в данном случае мы только знаем, что произошла какая-то ошибка, а какая именно, неизвестно. Поэтому через параметр в блоке catch мы можем получить то сообщение, которое передается оператору throw:

С помощью параметра const char\* error\_message получаем сообщение, которое предано оператору throw, и выводим это сообщение на консоль. Почему здесь мы получаем сообщение об ошибке в виде типа const char\*? Потому что после оператора throw идет строковый литерал, который представляет как раз тип const char\*. И в этом случае консольный вывод будет выглядеть следующим образом:

Division by zero!

The End...

Несколько catch

#include <iostream.h>  
 // можно перехватывать различные типы исключений  
 void Xhandler(int test)  
 {  
 try {  
 if (test) throw test;  
 else throw "Value is zero";  
 }  
 catch (int i) {  
 cout << "Caught Exception #: " << i << '\n';  
 }  
 catch(char \*str) {  
 cout << "Caught a string: ";  
 cout << str << '\n';  
 }  
 }  
 int main()  
 {  
 cout << "Start\n";  
 Xhandler(1);  
 Xhandler(2);  
 Xhandler(0);  
 Xhandler(3);  
 cout << "End";  
 return 0;  
 }

Эта программа выдаст следующий результат на экран:  
  
 Start  
 Caught Exception #: 1  
 Caught Exception #: 2  
 Caught a strung: Value is zero  
 Caught Exception #: 3  
 End

Как можно видеть, каждая инструкция catch отвечает только на свой тип исключений.

В общем случае инструкции catch проверяются на соответствие типа в том порядке, в котором они расположены в программе. Выполняется только та инструкция catch, чей тип соответствует типу исключения. Все остальные блоки catch игнорируются.

## **Источники**

<https://metanit.com/cpp/tutorial/6.1.php>

<https://www.c-cpp.ru/books/ispolzovanie-neskolkih-instrukciy-catch>

**Класс вектор. Эффективность операций.**

## Основные сведения[​](https://iu5edu.ru/wiki/cpp2/docs/labs/lab8/ReferenceMaterial/Vector/#%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5-%D1%81%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D1%8F)

Шаблонный класс vector входит в библиотеку STL. Вскоре вы убедитесь, что можно очень много сделать, если знать основные принципы работы класса vector, не разбираясь в его реализации.

В классе vector поддерживаются динамические массивы. Динамическим массивом называется массив, размеры которого могут увеличиваться по мере необходимости. Класс vector оформлен в виде шаблона, что позволяет эффективно использовать его с разными типами. Другими словами, можно создать вектор объектов double, вектор объектов int, вектор объектов string и т.д. При помощи шаблона можно создать «класс чего угодно». Чтобы сообщить компилятору, с каким типом данных будет работать класс (в данном случае — какие элементы будут храниться в векторе), укажите имя нужного типа в угловых скобках <...>. Так, вектор объектов string обозначается vector<string>. Такая запись определяет специализированный вектор, в котором могут храниться только объекты string. Если попытаться занести в него объект другого типа, компилятор выдаст сообщение об ошибке.

Поскольку класс vector представляет собой "контейнер", то есть предназначается для хранения однотипных элементов, в нем должны быть предусмотрены средства для сохранения и извлечения элементов. Новые элементы добавляются в конец вектора функцией push\_back() (помните, что эта функция принадлежит классу, поэтому, чтобы вызвать ее для конкретного объекта, нужно отделить ее имя от имени объекта символом точки). Извлечь элементы из вектора можно, используя индексацию: в классе выполнена перегрузка индексации. Благодаря перегрузке операторов программист работает с вектором как с массивом.

Учитывая все сказанное, рассмотрим пример использования векторов. Для этого следует включить в программу заголовочный файл <vector>:

*// Копирование всего содержимого файла в вектор строк*  
#*include* <fstream>  
#*include* <iostream>  
#*include* <string>  
#*include* <vector>  
  
*int* main() {  
#*ifdef* WIN32  
 system("chcp 65001");  
#*else*  
 setlocale(LC\_ALL, "Russian");  
#*endif*  
 std::vector<std::string> v;  
 std::ifstream in("text.txt");  
 std::string line;  
 *while* (getline(in, line))  
 v.push\_back(line); *// Занесение строки в конец вектора*  
 *// Нумерация строк:*  
 *for* (*int* i = 0; i < v.size(); i++) {  
 std::cout << i << ": " << v[i] << std::endl;  
 }  
}

Результат выполнения программы:

0: Aaaaddad@lll  
1: weqwe231321  
2: 123adsad  
3: 12312sadasd  
4: 3dffdsf  
5: asdasdsadasd

Мы открываем файл и последовательно читаем его строки в объекты string. Объекты заносятся в конец вектора v. После завершения цикла while все содержимое файла будет находиться в памяти внутри объекта v.

Условие проверки цикла for означает, что для продолжения работы цикла счетчик i должен быть меньше количества элементов в векторе v (количество элементов определяется функцией size() класса vector).

Откомпилируйте и запустите программу, и вы увидите результат — строки выходного файла окажутся пронумерованными.

Подход оператора >> к работе с потоками позволяет легко изменить программу так, чтобы файл разбивался не по строкам, а по словам, разделенным пропусками:

*// Разбиение файла по словам, разделенный пропусками*   
#*include* <fstream>  
#*include* <iostream>  
#*include* <string>  
#*include* <vector>  
  
*int* main() {  
#*ifdef* WIN32  
 system("chcp 65001");  
#*else*  
 setlocale(LC\_ALL, "Russian");  
#*endif*  
 std::vector<std::string> words;  
 std::ifstream in("text.txt");  
 std::string word;  
 *while* (in >> word) words.push\_back(word);  
 *for* (*int* i = 0; i < words.size(); i++) {  
 std::cout << words[i] << std::endl;  
 }  
}

Результат выполнения программы:

Aaaaddad@lll  
weqwe231321  
123adsad  
12312sadasd  
3dffdsf  
asdasdsadasd

Следующее выражение обеспечивает ввод очередного "слова": while (in >> word). Когда условие цикла становится ложным, это означает, что был достигнут конец файла. Чтобы убедиться в том, как просто работать с классом vector, рассмотрим следующий пример, в котором создается вектор с элементами типа int:

*// Создание вектора для хранения целых чисел*   
#*include* <iostream>  
#*include* <vector>  
  
*int* main() {  
 std::vector<*int*> v;  
 *for* (*int* i = 0; i < 10; i++) {  
 v.push\_back(i);  
 }  
 *for* (*int* i = 0; i < v.size(); i++) {  
 std::cout << v[i] << ", ";  
 }  
 std::cout << std::endl;  
 *for* (*int* i = 0; i < v.size(); i++) {  
 v[i] = v[i] \* 10; *// Присваивание*  
 }  
 *for* (*int* i = 0; i < v.size(); i++) {  
 std::cout << v[i] << ", ";  
 }  
 std::cout << std::endl;  
}

Результат выполнения программы:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,   
0, 10, 20, 30, 40, 50, 60, 70, 80, 90,

Для класса vector определяются следующие операторы сравнения: =, <, <=, !=, >, >=.

Наиболее важными функциями-членами класса vector являются функции size(), begin(), end(), push\_back(), insert() и erase(). Функция size() возвращает текущий размер вектора. Эта функция особенно полезна, поскольку позволяет узнать размер вектора во время выполнения программы. Помните, вектор может расти по мере необходимости, поэтому размер вектора необходимо определять не в процессе компиляции, а в процессе выполнения программы. Функция begin() возвращает итератор начала вектора. Функция end() возвращает итератор конца вектора. Как уже говорилось, итераторы очень похожи на указатели и с помощью функций begin() и end() можно получить итераторы (читай: указатели) начала и конца вектора. Функция push\_back() помещает значение в конец вектора. Если это необходимо для размещения нового элемента, вектор удлиняется. В середину вектора элемент можно добавить с помощью функции insert(). Вектор можно инициализировать. В любом случае, если в векторе хранятся элементы, то с помощью оператора индекса массива к этим элементам можно получить доступ и их изменить. Удалить элементы из вектора можно с помощью функции erase().

## Примеры работы с векторами[​](https://iu5edu.ru/wiki/cpp2/docs/labs/lab8/ReferenceMaterial/Vector/#%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80%D1%8B-%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%8B-%D1%81-%D0%B2%D0%B5%D0%BA%D1%82%D0%BE%D1%80%D0%B0%D0%BC%D0%B8)

### Пример 1[​](https://iu5edu.ru/wiki/cpp2/docs/labs/lab8/ReferenceMaterial/Vector/#%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80-1)

Как вы знаете, в C++ массивы и указатели очень тесно связаны. Доступ к массиву можно получить либо через оператор индексирования, либо через указатель. По аналогии с этим в библиотеке стандартных шаблонов имеется тесная связь между векторами и итераторами. Доступ к членам вектора можно получить либо через оператор индексирования, либо через итератор. В следующем примере показаны оба этих подхода.

#*include* <iostream>  
#*include* <vector>  
  
*int* main() {  
 std::vector<*int*> v; *// создание вектора нулевой длины*  
 *int* i;  
 *// помещение значений в вектор*  
 *for* (i = 0; i < 10; i++) {  
 v.push\_back(i);  
 }  
 *// доступ к содержимому вектора*  
 *// с использованием оператора индекса*  
 *for* (i = 0; i < 10; i++) {  
 std::cout << v[i] << " ";  
 }  
 std::cout << std::endl;  
 *// доступ к вектору через итератор*  
 std::vector<*int*>::iterator p = v.begin();  
 *while* (p != v.end()) {  
 std::cout << \*p << " ";  
 p++;  
 }  
 std::cout << std::endl;  
}

Результат выполнения программы:

0 1 2 3 4 5 6 7 8 9   
0 1 2 3 4 5 6 7 8 9

В этой программе сначала создается вектор v нулевой длины. Далее с помощью функции-члена push\_back() к концу вектора v добавляются некоторые значения и размер вектора v увеличивается. Обратите внимание на объявление итератора р. Тип iterator определяется с помощью класса-контейнера. То есть, чтобы получить итератор для выбранного контейнера, объявить его нужно именно так, как показано в примере: просто укажите перед типом iterator имя контейнера. С помощью функции-члена begin() итератор инициализируется, указывая на начало вектора. Возвращаемым значением этой функции как раз и является итератор начала вектора. Теперь, применяя к итератору оператор инкремента, можно получить доступ к любому выбранному элементу вектора. Этот процесс совершенно аналогичен использованию указателя для доступа к элементам массива. С помощью функции-члена end() определяется факт достижения конца вектора. Возвращаемым значением этой функции является итератор того места, которое находится сразу за последним элементом вектора, Таким образом, если итератор р равен возвращаемому значению функции v.end(), значит, конец вектора был достигнут.

Источники:

https://iu5edu.ru/wiki/cpp2/docs/labs/lab8/ReferenceMaterial/Vector/

**Умные указатели.**

Указатели в языках C и C++ — те еще штучки. Они чрезвычайно мощные, но в то же время такие опасные: достаточно небольшого недосмотра, чтобы сломать все ваше приложение. Проблема в том, что управление указателями полностью зависит от вас. За каждым динамическим выделением объекта (например, new T) должно следовать ручное удаление (например, delete T). Забудете это сделать, и в итоге получите хорошенькую утечку памяти.

## Суть умных указателей

Умные указатели были созданы для устранения вышеупомянутых неудобств. По сути, они обеспечивают автоматическое управление памятью: когда умный указатель больше не используется, то есть выходит из области видимости, память, на которую он указывает, автоматически высвобождается. Традиционные указатели теперь также называют «обычными» указателями.

## Понимание std::unique\_ptr: одиночный вариант

std::unique\_ptr владеет объектом, на который он указывает, и никакие другие умные указатели не могут на него указывать. Когда std::unique\_ptr выходит из области видимости, объект удаляется. Это полезно, когда вы работаете с временным, динамически выделенным ресурсом, который может быть уничтожен после выхода из области действия.

Как создать std::unique\_ptr

A std::unique\_ptr создается следующим образом:

std::unique\_ptr<Type> p(**new** Type);

Например:

std::unique\_ptr<**int**> p1(**new** **int**);

std::unique\_ptr<**int**[]> p2(**new** **int**[50]);

std::unique\_ptr<**Object**> p3(**new** **Object**("Lamp"));

Также можно создать std::unique\_ptrs с помощью специальной функции std::make\_unique, вот так:

std::unique\_ptr<Type> p = std::make\_unique<Type>(...размер или параметры...);

Например:

std::unique\_ptr<**int**> p1 = std::make\_unique<**int**>();

std::unique\_ptr<**int**[]> p2 = std::make\_unique<**int**[]>(50);

std::unique\_ptr<**Object**> p3 = std::make\_unique<**Object**>("Lamp");

Если есть возможность, всегда старайтесь выделять объекты с помощью std::make\_unique. Почему лучше поступать именно так, я покажу в последнем разделе этой статьи.

std::unique\_ptr в действии

Главная особенность этого умного указателя — исчезать, когда он больше не используется. Рассмотрим следующий код:

**void** compute()

{

std::unique\_ptr<**int**[]> data = std::make\_unique<**int**[]>(1024);

/\* выполнение некоторых значимых вычислений над вашими данными...\*/

} // `data` выходит из области действия здесь: она автоматически уничтожается

**int** main()

{

compute();

}

Умный указатель выходит из области видимости, когда функция compute() достигает конца тела. Вызывается деструктор указателя, и память очищается автоматически. Больше ни о чем волноваться не нужно.

Один ресурс, один std::unique\_ptr

Я могу сказать, что std::unique\_ptr очень ревниво относится к динамическому объекту, который он хранит: невозможно иметь несколько ссылок на его динамические данные. Например:

**void** compute(std::unique\_ptr<**int**[]> p) { ... }

**int** main()

{

std::unique\_ptr<**int**[]> ptr = std::make\_unique<**int**[]>(1024);

std::unique\_ptr<**int**[]> ptr\_copy = ptr; // ОШИБКА! Копирование запрещено

compute(ptr); // ОШИБКА! `ptr` передается копией, а копирование не разрешено

}

Это сделано специально, и это важная особенность std::unique\_ptr: на любой ресурс может указывать не более одного std::unique\_ptr. Это предотвращает ошибочное многократное удаление указателя.

Технически это происходит потому, что у std::unique\_ptr нет конструктора копирования: это может быть очевидно для вас, если вы знакомы с семантикой перемещения (я написал [об этом вводную статью](https://www.internalpointers.com/post/c-rvalue-references-and-move-semantics-beginners), если вы не знакомы). Во второй части этой статьи я покажу, как правильно передавать умные указатели.

## Понимание std::shared\_ptr: конвивиальный вариант

std::shared\_ptr владеет объектом, на который он указывает, но, в отличие от std::unique\_ptr, он допускает множественные ссылки. Специальный внутренний счетчик уменьшается каждый раз, когда std::shared\_ptr, указывающий на тот же ресурс, выходит из области видимости. Эта техника называется подсчетом ссылок. Когда последняя из них будет уничтожена, счетчик станет равным нулю, и данные будут высвобождены.

Умный указатель такого типа полезен, когда требуется обмениваться динамически распределенными данными, точно так же, как это делается с обычными указателями или ссылками.

Как создать std::shared\_ptr

std::shared\_ptr создается так:

std::shared\_ptr<Type> p(**new** Type);

Например:

std::shared\_ptr<**int**> p1(**new** **int**);

std::shared\_ptr<**Object**> p2(**new** **Object**("Lamp"));

Существует альтернативный способ создания std::shared\_ptr, использующий специальную функциюstd::make\_shared:

std::shared\_ptr<Type> p = std::make\_shared<Type>(...parameters...);

Например:

std::shared\_ptr<**int**> p1 = std::make\_shared<**int**>();

std::shared\_ptr<**Object**> p2 = std::make\_shared<**Object**>("Lamp");

Это должен быть наиболее предпочтительный способ построения такого рода умных указателей. Я покажу вам почему в последнем разделе этой статьи.

Проблемы с массивами

До C++17 не было простого способа соорудить std::shared\_ptr, хранящий массив. До C++17 этот умный указатель по умолчанию всегда вызывает delete (а не delete[]) на своем ресурсе: вы можете создать обходной путь, используя кастомное удаление. Один из многих конструкторов std::shared\_ptr принимает в качестве второго параметра лямбду, в которой вы вручную удаляете принадлежащий ему объект. Например:

std::shared\_ptr<**int**[]> p2(**new** **int**[16], [] (**int**\* i) {

delete[] i; // Кастомное удаление

});

К сожалению, нет возможности сделать это при использовании std::make\_shared.

std::shared\_ptr в действии

Одна из главных особенностей std::shared\_ptr — возможность отслеживать, сколько указателей ссылаются на один и тот же ресурс. Получить информацию о количестве ссылок можно с помощью метода use\_count(). Рассмотрим следующее:

**void** compute()

{

std::shared\_ptr<**int**> ptr = std::make\_shared<**int**>(100);

// ptr.use\_count() == 1

std::shared\_ptr<**int**> ptr\_copy = ptr; // Сделать копию: с shared\_ptr возможно!

// ptr.use\_count() == 2

// ptr\_copy.use\_count() == 2, в конце концов, это одни и те же базовые данные.

} // Здесь `ptr` и `ptr\_copy` выходят из области действия. Больше никаких ссылок

// исходные данные (т.е. use\_count() == 0), поэтому они автоматически убираются.

**int** main()

{

compute();

}

Обратите внимание, как ptr и ptr\_copy выходят из области видимости в конце функции, доводя счетчик ссылок до нуля. В этот момент деструктор последнего объекта обнаруживает, что ссылок больше нет, и запускает очистку памяти.

Один ресурс, много std::shared\_ptr. Не забывайте о циклических ссылках!

Сила множественных ссылок может привести к неприятным сюрпризам. Скажем, я пишу игру, в которой у игрока есть другой игрок в качестве компаньона, например, так:

struct Player

{

std::shared\_ptr<Player> companion;

~Player() { std::cout << "~Player\n"; }

};

**int** main()

{

std::shared\_ptr<Player> jasmine = std::make\_shared<Player>();

std::shared\_ptr<Player> albert = std::make\_shared<Player>();

jasmine->companion = albert; // (1)

albert->companion = jasmine; // (2)

Логично, не так ли? К сожалению, я только что создал так называемую круговую ссылку. В начале моей программы я создаю два умных указателя jasmine и albert, которые хранят динамически создаваемые объекты: назовем эти динамические данные jasmine-data и albert-data, чтобы было понятнее.

Затем в (1) я передаю jasmine указатель на albert-data, а в (2) albert хранит указатель на jasmine-data. Это все равно что дать каждому игроку компаньона.

Когда jasmine выходит из области видимости в конце программы, ее деструктор не может очистить память: все еще есть один умный указатель, указывающий на jasmine-data, это albert->companion. Аналогично, когда albert выходит из области видимости в конце программы, его деструктор не может очистить память: ссылка на albert-data все еще живет через jasmine->companion. В этот момент программа просто завершается, не освободив память: утечка памяти во всем ее великолепии. Если вы запустите приведенный выше фрагмент, то заметите, что ~Player() никогда не будет вызван.

Это не такая большая проблема, так как операционная система позаботится об очистке памяти за вас. Однако вам не стоит иметь такие круговые зависимости (т.е. утечки памяти) в середине вашей программы. К счастью, на помощь придет последний тип умного указателя.

## Понимание std::weak\_ptr: поверхностный вариант

std::weak\_ptr — это, по сути, std::shared\_ptr, который не увеличивает счетчик ссылок. Он определяется как умный указатель, который содержит несобственную ссылку, или ослабленную ссылку, на объект, управляемый другим std::shared\_ptr.

Этот умный указатель полезен для решения некоторых раздражающих проблем, которые нельзя решить с помощью необработанных указателей. Вскоре мы увидим, как это сделать.

Как создать std::weak\_pt

Вы можете создать std::weak\_ptr только из std::shared\_ptr или другого std::weak\_ptr. Например:

std::shared\_ptr<**int**> p\_shared = std::make\_shared<**int**>(100);

std::weak\_ptr<**int**> p\_weak1(p\_shared);

std::weak\_ptr<**int**> p\_weak2(p\_weak1);

В приведенном выше примере p\_weak1 и p\_weak2 указывают на одни и те же динамические данные, принадлежащие p\_shared, но счетчик ссылок не растет.

std::weak\_ptr в действии

std::weak\_ptr является своего рода инспектором дляstd::shared\_ptr от которого он зависит. Вы должны сначала преобразовать его в std::shared\_ptr с помощью метода lock() если вы действительно хотите работать с реальным объектом:

std::shared\_ptr<**int**> p\_shared = std::make\_shared<**int**>(100);

std::weak\_ptr<**int**> p\_weak(p\_shared);

// ...

std::shared\_ptr<**int**> p\_shared\_orig = p\_weak.lock();

//

Конечно, p\_shared\_orig может быть нулевым в случае, если p\_shared был удален в другом месте.

std::weak\_ptr решает проблемы

С помощью std::weak\_ptr очень легко решить проблему висящих указателей — тех, которые указывают на уже удаленные данные. Он предоставляет метод expired(), который проверяет, был ли объект, на который ссылается ссылка, уже удален. Если expired() == true, исходный объект был где-то удален, и вы можете действовать соответствующим образом. Это то, что вы не можете сделать с необработанными указателями.

Как я уже говорил, std::weak\_ptr также используется для разрыва циклической ссылки. Давайте вернемся к примеру Player, приведенному выше, и изменим переменную-член с std::shared\_ptr companion на std::weak\_ptr companion. В данном случае мы использовали std::weak\_ptr для устранения запутанного владения. Фактически имкющиеся динамически выделяемые данные остаются в основном теле, в то время как каждый Player теперь имеет слабую ссылку на них. Запустите код с этим изменением, и вы увидите, что деструктор вызывается дважды, правильно.

### **Источники**

<https://habr.com/ru/companies/piter/articles/706866/>

**Значение по умолчанию для параметров функций.**

**Значения по умолчанию для аргументов функции** — это возможность гибкого использования перегрузки функций в сочетании с сокращением объёма кода. Функция со значениями по умолчаниями может быть вызвана с разным количеством параметров.

При этом значения по умолчанию могут быть только у некоторого количества *последних* аргументов.

**Пример**

**void f(int a = 1, int b = 2, int c = 3);**

Данную функцию можно вызвать как с одним, так и с двумя, так и с тремя параметрами.

**f(10); // будет вызвано f(10, 2, 3)**

**f(10, 20); // будет вызвано f(10, 20, 3)**

**f(10, 20, 30); // значения по умолчанию совсем не использованы**

Её можно вызвать вообще без параметров:

**f(); //для всех параметров использованы значения по умолчанию - f(1, 2, 3)**

Значением по умолчанию не может являться значение другого аргумента:

**void f(int a, int b = a); // Не допустимо!**

**Функции с аргументами по умолчанию и перегрузка**

Поскольку такая функция может вызвана с разным количеством параметров, она участвует в перегрузке одновременно конкурируя с одноимёнными функциями с разным количеством параметров.

Источники:

<https://foxford.ru/wiki/informatika/argumenty-funktsii-po-umolchaniyu-v-s?srsltid=AfmBOorX4Yi7pdnEXZsl5AzBP7ve6cC9UTZGR8DhMW-Q57z2-Sedrfo_&utm_referrer=https%3A%2F%2Fwww.google.com%2F>

### **Тернарный оператор**

Тернарный оператор в некотором роде похож на конструкцию if-else. Он принимает три операнда в следующем виде:

|  |  |
| --- | --- |
| 1 | операнд1? операнд2 : операнд3 |

Первый операнд представляет условие. Если это условие верно (равно true), тогда выбирается/выполняется второй операнд, который помещается после символа **?**. Если условие не верно, тогда выбирается/выполняется третий операнд, который помещается после двоеточия.

Например, возьмем следующую конструкцию if-else:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <iostream>    int main()  {      int a {5};      int b {8};      int c{};      if(a > b)      {          c = a - b;      }      else      {          c = a + b;      }      std::cout << "c = " << c << std::endl;  // c = 13  } |

Здесь если a больше b, то c=a-b, иначе c=a+b. Перепишем ее с помощью тернарного оператора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include <iostream>    int main()  {      int a {5};      int b {8};      int c = a > b ? a - b : a + b;        std::cout << "c = " << c << std::endl;  // c = 13  } |

Здесь первым операндом тернарного оператора является условие a > b. Если это условие верно, то возвращается второй операнд - результат выражения a - b. Если условие не верно, то возвращается третий операнд - a + b. И возвращенный операнд присваивается переменной c.

Тернарный оператор не обязательно должен возвращать некоторое значение, он может просто выполнять некоторые действия. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #include <iostream>    int main()  {      int a {5};      int b {8};      a > b ? std::cout << a-b : std::cout << a+b;  } |

Здесь тот же первый операнд-условие. Если оно верно, выполняется второй операнд - std::cout << a-b, если нет, то третий операнд - std::cout << a+b.

В рамках одного тернарного оператора можно комбинировать несколько других. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <iostream>    int main()  {      int a {5};      int b {8};      std::cout << (a < b ? "a is less than b" :          (a == b ? "a is equal to b" : "a is greater than b"));  } |

Здесь условие представляет выражение a < b. Если оно верно, то возвращается второй операнд - строка "a is less than b". Но если условие не верно, то возвращается третий операнд, который, в свою очередь, представляет другой тернарный оператор (a == b ? "a is equal to b" : "a is greater than b"). Здесь опять же оценивается условие-первый операнд a == b. Если оно верно, то возвращается строка "a is equal to b". Если нет, то строка - "a is greater than b".

Источники:

<https://metanit.com/cpp/tutorial/2.12.php>

<https://habr.com/ru/articles/205848/>