Выполнил студент группы ИВТ(ВМК)-21

Рычков Родион

Оглавление

[Значения аргументов функции по умолчанию 2](#_Toc148555289)

[Устройство памяти программы 4](#_Toc148555290)

[Правила оформления кода 6](#_Toc148555291)

[Пробелы и отступы 6](#_Toc148555292)

[Названия и переменные 7](#_Toc148555293)

[Базовые выражения C++ 7](#_Toc148555294)

[Чрезмерность 8](#_Toc148555295)

[Комментарии 9](#_Toc148555296)

[Эффективность 9](#_Toc148555297)

[Функции и процедурное проектирование 10](#_Toc148555298)

[Параметры функции main (argc, argv) 11](#_Toc148555299)

[Уменьшение размера исполняемого файла 13](#_Toc148555301)

[Обработка исключительных ситуаций 15](#_Toc148555302)

[Typedef 19](#_Toc148555303)

[Static 22](#_Toc148555304)

[Auto 25](#_Toc148555305)

[Источники 27](#_Toc148555306)

# Значения аргументов функции по умолчанию

Аргумент по умолчанию – это такой аргумент функции, который программист может не указывать при вызове функции. Аргумент по умолчанию добавляется компилятором автоматически.

Чтобы использовать аргументы по умолчанию в функции, эта функция должна быть соответствующим образом объявлена. Аргументы по умолчанию объявляются в прототипе функции.

**Общая форма** объявления функции, которая содержит аргументы по умолчанию

int my\_func(int a, int b, int c=12);

Эта функция принимает три аргумента, последний из которых имеет значение по умолчанию 12. Программист может вызвать эту функцию двумя способами:

result = my\_func(1, 2, 3);

result = my\_func(1, 2);

В первом случае значение аргумента **c** определяется в вызове функции и равно трём. Во втором случае последний аргумент опущен и **c** примет значение двенадцать*.*

Зачем они нужны?

1. Улучшение читаемости кода: Использование значений по умолчанию позволяет избегать избыточного кода и упрощает понимание того, как функция работает. Когда разработчик видит функцию с аргументами по умолчанию, он может сразу понять, какие параметры можно не передавать, если это необходимо.

2. Поддержание обратной совместимости: Значения по умолчанию могут быть полезны при изменении интерфейса функции. Если в будущем нужно добавить новый аргумент, старый код, не передающий этот аргумент, продолжит работать с значениями по умолчанию.

##### Пример объявления и использования функций, которые используют аргументы по умолчанию

**Пример.** Объявляется функция Inc(),получающая два параметра. Функция возвращает сумму значений первого и второго параметров. Второй параметр функции есть параметром (аргументом) по умолчанию, которому присваивается значение 1.

Текст программы типа Win32 Console Application, демонстрирующий использование функции Inc(), имеет вид

#include "stdafx.h"

#include <iostream>

using namespace std;

// функция, содержащая аргумент по умолчанию

int Inc(int value, int step = 1)

{

return value + step;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

//

int v1, v2;

v1 = 5;

// использование аргумента по умолчанию

v2 = Inc(v1); // v2 = 5 + 1 = 6

v1 = 5;

v2 = Inc(v1, 3); // v2 = 5 + 3 = 8

cout << v2 << endl;

return 0;

}

# Устройство памяти программы

Память — одна из самых сложных тем в информатике, но понимание устройства памяти компьютера позволяет разрабатывать более эффективные программы, а для более низкоуровневого программирования, например, при разработке ОС, это понимание и вовсе является обязательным.

В этой статье будет рассмотрена модель памяти с высокоуровневой точки зрения — виды памяти, аллокаторы, сборщик мусора.

Виды памяти

Существует 3 типа памяти: статический, автоматический и динамический.

**Статический**— выделение памяти до начала исполнения программы. Такая память доступна на протяжении всего времени выполнения программы. Во многих языках для размещения объекта в статической памяти достаточно задекларировать его в глобальной области видимости.

int id = 150; // определение статической глобальной переменной

int main()

{

std::cout << id + 8; // её использование

}

**Автоматический**, также известный как «размещение на стеке», — самый основной, автоматически выделяет аргументы и локальные переменные функции, а также прочую метаинформацию при вызове функции и освобождает память при выходе из неё.

Стек, как структура данных, работает по принципу LIFO («последним пришёл — первым ушёл»). Другими словами, добавлять и удалять значения в стеке можно только с одной и той же стороны.

Автоматическая память работает именно на основе стека, чтобы вызванная из любой части программы функция не затёрла уже используемую автоматическую память, а добавила свои данные в конец стека, увеличивая его размер. При завершении этой функции её данные будут удалены с конца стека, уменьшая его размер. Длина стека останется той же, что и до вызова функции, а у вызывающей функции указатель на конец стека будет указывать на тот же адрес.

Проще всего это понять из примера на С++:

int main()

{

int a = 3;

int result = factorial(a);

std::cout << result;

}

int factorial(int n)

{

if (n <= 1) return 1;

return n \* factorial(n - 1);

}

**Динамическая** — выделение памяти из ОС по требованию приложения.

При работе с данными мы не всегда можем знать, сколько еще памяти нам понадобится. Для этого можно использовать остальную память – она называется **куча (heap)**. В ходе работы программы мы можем динамически выделять дополнительную (динамическую) память и работать через ее адреса.

В C++:

int \*ptr\_i;  
double \*ptr\_d;  
…  
ptr\_i = new int;  
ptr\_d = new double[10];

…

delete ptr\_i;  
delete[] ptr\_d;

После выделения памяти в распоряжение программы поступает указатель на начало выделенной памяти, который, в свою очередь, тоже должен где-то храниться: в статической, автоматической или также в динамической памяти.

В C++ компилятор не отслеживает выделение памяти пользователем и указатели/ссылки на нее, поэтому если указатель будет утерян (удален), то область в куче так и останется выделенной до перезапуска программы.

Данный механизм потери памяти из-за потери указателя называется утечкой (**leaks)**.

Если утечек будет много — память закончится, и код не сможет выполняться.

# Правила оформления кода

# Пробелы и отступы

Отделяйте пробелами фигурные скобки:

// Плохая практика

int x = 3, y = 7; double z = 4.25; x++;

if (a == b) { foo(); }

// Хорошая практика

int x = 3;

int y = 7;

double z = 4.25;

x++;

if (a == b) {

foo();

}

Ставьте пробелы между операторами и операндами:

int x = (a + b) \* c / d + foo();

# Названия и переменные

Давайте переменным описательные имена, такие как firstName или homeworkScore. Избегайте однобуквенных названий вроде x или c, за исключением итераторов вроде i.

Называйте переменные и функции, используя верблюжийРегистр. Называйте классы ПаскальнымРегистром, а константы — в ВЕРХНЕМ\_РЕГИСТРЕ.

Если переменная используется лишь внутри определенного if, то делайте её локальной, объявляя в том же блоке кода, а не глобальной.

Выбирайте подходящий тип данных для ваших переменных. Если переменная содержит лишь целые числа, то определяйте её как int, а не double.

Используйте текстовую строку, стандартную для C++, а не С. С++ путает тем, что имеет два вида текстовых строк: класс string из С++ и старый char\* (массив символов) из С:

// Плохая практика: текстовая строка в стиле Cи

char\* str = "Hello there";

// Хорошая практика: текстовая строка в стиле C++

string str = "Hello there";

# Базовые выражения C++

С++ основан на С, поэтому всегда есть вариант решить задачу «путем С++» и «путем С». Например, когда вы желаете вывести что-либо на системную консоль, вы можете сделать это «путем С++» , использовав оператор вывода cout, в то время как «путем С» вы бы использовали глобальную функцию вроде printf:

// Плохая практика

printf("Hello, world!\n");

// Хорошая практика

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

Частенько затрудняетесь с выбором между for и while? Используйте цикл for, когда вы знаете количество повторений, а цикл while, когда количество повторений неизвестно:

// Повторяет 'size' раз

for (int i = 0; i < size; i++) {

...

}

// Повторяет, пока больше не будет строк

string str;

while (input >> str) {

...

}

Старайтесь избегать использования выражений break или continue. Используйте их только в том случае, если это абсолютно необходимо.

# Чрезмерность

Если вы используете один и тот же код дважды или более, то найдите способ удалить излишний код, чтобы он не повторялся. К примеру, его можно поместить во вспомогательную функцию. Если повторяемый код похож, но не совсем, то постарайтесь сделать вспомогательную функцию, которая принимает параметры и представляет разнящуюся часть:

// Плохая практика

foo();

x = 10;

y++;

...

foo();

x = 15;

y++;

// Хорошая практика

helper(10);

helper(15);

...

void helper(int newX) {

foo();

x = newX;

y++;

}

# Комментарии

**Заглавный комментарий**. Размещайте заглавный комментарий, который описывает назначение файла, вверху каждого файла. Предположите, что читатель вашего комментария является продвинутым программистом, но не кем-то, кто уже видел ваш код ранее.

**Заголовок функции / конструктора**. Разместите заголовочный комментарий на каждом конструкторе и функции вашего файла. Заголовок должен описывать поведение и / или цель функции.

**Параметры / возврат**. Если ваша функцию принимает параметры, то кратко опишите их цель и смысл. Если ваша функция возвращает значение — кратко опишите, что она возвращает.

**Исключения**. Если ваша функция намеренно выдает какие-то исключения для определенных ошибочных случаев, то это требует упоминания.

**Комментарии на одной строке**. Если внутри функции имеется секция кода, которая длинна, сложна или непонятна, то кратко опишите её назначение.

**TODO**. Следует удалить все // TODO комментарии перед тем, как заканчивать и сдавать программу.

# Эффективность

Вызывая большую функцию и используя результат несколько раз, сохраните результат в переменной вместо того, чтобы постоянно вызывать данную функцию:

// Плохая практика

if (reallySlowSearchForIndex("abc") >= 0) {

remove(reallySlowSearchForIndex("abc"));

}

// Хорошая практика

int index = reallySlowSearchForIndex("abc");

if (index >= 0) {

remove(index);

}

# Функции и процедурное проектирование

Хорошо спроектированная функция имеет следующие характеристики:

1.Полностью выполняет четко поставленную задачу;

2.Не берет на себя слишком много работы;

3.Не связана с другими функциями бесцельно;

4.Хранит данные максимально сжато;

5.Помогает распознать и разделить структуру программы;

6.Помогает избавиться от излишков, которые иначе присутствовали бы в программе.

Используйте параметры, чтобы отправлять информацию из функции или когда функции нужно возвратить несколько значений. Не используйте параметры без необходимости. Заметьте, что a,b, и с не являются параметрами в нижеприведенной функции, так как это не нужно:

/\*

\* Решает квадратное уравнение ax^2 + bx + c = 0,

\* внося результаты в root1 и root2.

\* Предполагается, что данные уравнения имеют два корня.

\*/

void quadratic(double a, double b, double c,

double& root1, double& root2) {

double d = sqrt(b \* b - 4 \* a \* c);

root1 = (-b + d) / (2 \* a);

root2 = (-b - d) / (2 \* a);

}

Когда требуется вернуть значение из функции, используйте значение return:

// Плохая практика

void max(int a, int b, int& result) {

if (a > b) {

result = a;

} else {

result = b;

}

}

// Хорошая практика

int max(int a, int b) {

if (a > b) {

return a;

} else {

return b;

}

}

# Параметры функции main (argc, argv)

При создании консольного приложения в языке программирования С++, автоматически создается строка очень похожая на эту:

|  |  |
| --- | --- |
|  | int  main(int argc, char\* argv[]) // параметры функции main() |

Эта строка — заголовок главной функции main(), в скобочках объявлены параметры argс и argv. Так вот, если программу запускать через командную строку, то существует возможность передать какую-либо информацию этой программе, для этого и существуют параметры argcи argv[]. Параметр argc имеет тип данных int, и содержит количество параметров, передаваемых в функцию main. Причем argc всегда не меньше 1, даже когда мы не передаем никакой информации, так как первым параметром считается имя функции. Параметр argv[] это  массив указателей на строки. Через командную строку  можно передать только данные строкового типа. Указатели и строки — это две большие темы, под которые созданы отдельные разделы. Так вот именно через параметр argv[] и передается какая-либо информация. Разработаем программу, которую будем запускать через командную строку Windows, и передавать ей некоторую информацию.

# 

Рисунок - Код

После того как отладили программу, открываем командную строку Windows и перетаскиваем в окно командной строки exe файл нашей программы, в командной строке отобразится полный путь к программе(но можно прописать путь к программе в ручную), после этого можно нажимать **ENTER** и программа  запустится (см. Рисунок 2).

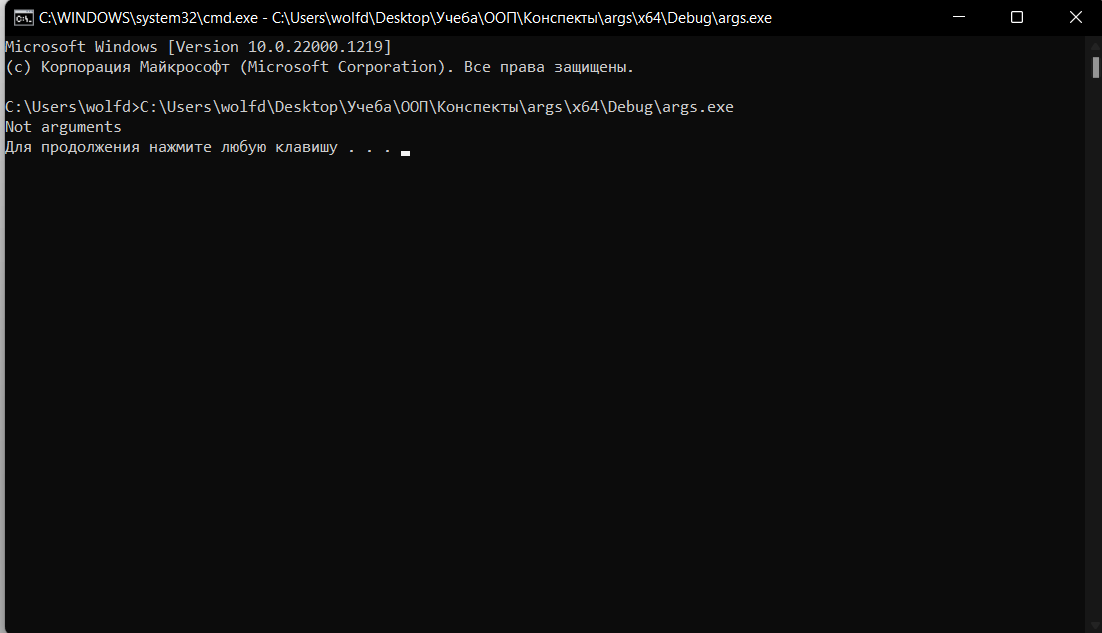


Рисунок 2

Так как мы просто запустили программу и не передавали ей никаких аргументов, появилось сообщение Not arguments. На рисунке 3 изображён запуск этой же программы через командную строку, но уже с передачей ей аргумента Open.

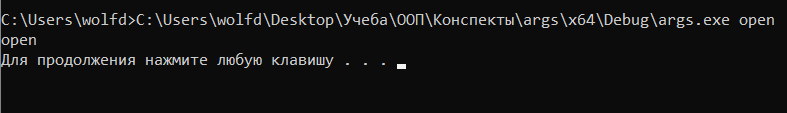


Рисунок 3

Аргументом является слово Open, как видно из рисунка, это слово появилось на экране. Передавать можно несколько параметров сразу, отделяя их между собой запятой. Если необходимо передать параметр состоящий из нескольких слов, то их необходимо взять в двойные кавычки, и тогда эти слова будут считаться как один параметр. Например, на рисунке изображен запуск программы, с передачей ей аргумента, состоящего из двух слов — It work.

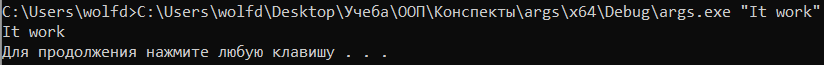


Рисунок 4

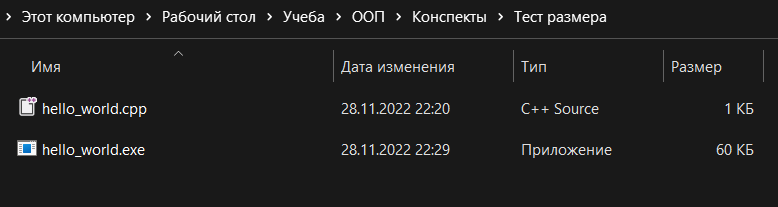
А если убрать кавычки. То увидим только слово It. Если не планируется передавать какую-либо информацию при запуске программы, то можно удалить аргументы в функции main(), также можно менять имена данных аргументов. Иногда встречается модификации параметров argc и argv[], но это все зависит от типа создаваемого приложения или от среды разработки.

Источник: <http://cppstudio.com/post/421/>

# Уменьшение размера исполняемого файла

При создании какой-либо программы, даже самой простой, которая выводит знаменитую во всем мире программирования первую фразу: “Hello world!”, можно заметить, что размер файла на удивление большой.

При обычной компиляции размер файла будет

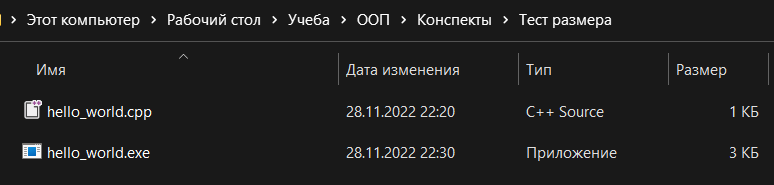




Чтобы уменьшить размер файла, необходимо дописать -s

Параметр -s (или --strip-all) позволяет максимально уменьшить размер исполняемого файла, удалив из него всю информацию о символах и релокации.





Размер исполняемого файла успешно уменьшен.

Есть еще различные способы оптимизации

"-Ofast" аналогично "-O3 -ffast-math" включает более высокий уровень оптимизаций и более агрессивные оптимизации для арифметических вычислений (например, вещественную реассоциацию)

"-flto" межмодульные оптимизации

"-m32" 32 битный режим

"-mfpmath=sse" включает использование XMM регистров в вещественной арифметике (вместо вещественного стека в x87 режиме)

"-funroll-loops" включает развертывание циклов

"-ffunction-sections" размещает каждую функцию в отдельной секции

"-Os" оптимизирует производительность и размер

"-fno-asynchronous-unwind-tables" гарантирует точность таблиц раскрутки только в пределах функции

///Добавить информацию про -s

Источник: <https://stackoverflow.com/questions/1042773/gcc-c-hello-world-program-exe-is-500kb-big-when-compiled-on-windows-how>

# Обработка исключительных ситуаций

В жизни любой программы бывают моменты, когда всё идёт не совсем так, как задумывал разработчик. Например:

* в системе закончилась оперативная память;
* соединение с сервером внезапно прервалось;
* пользователь выдернул флешку во время чтения или записи файла;
* понадобилось получить первый элемент списка, который оказался пустым;
* формат файла не такой, как ожидалось.

Примеры объединяет одно: возникшая ситуация достаточно редка, и при нормальной работе программы, всех устройств, сети и адекватном поведении пользователя она не возникает.

Хороший программист старается предусмотреть подобные ситуации. Однако это бывает сложно: перечисленные проблемы обладают неприятным свойством — они могут возникнуть практически в любой момент.

**Try-блок (Блок попытки)**: В этом блоке вы размещаете код, который может вызвать исключение (ошибку). Ваша цель - защитить этот код, чтобы предотвратить аварийное завершение программы в случае ошибки.

**Catch-блоки (Блоки перехвата)**: После блока try следуют один или несколько блоков catch. Каждый catch-блок предназначен для обработки конкретного типа исключения. Если исключение сгенерировано в блоке try, система обработки исключений будет искать соответствующий catch-блок для обработки этого исключения.

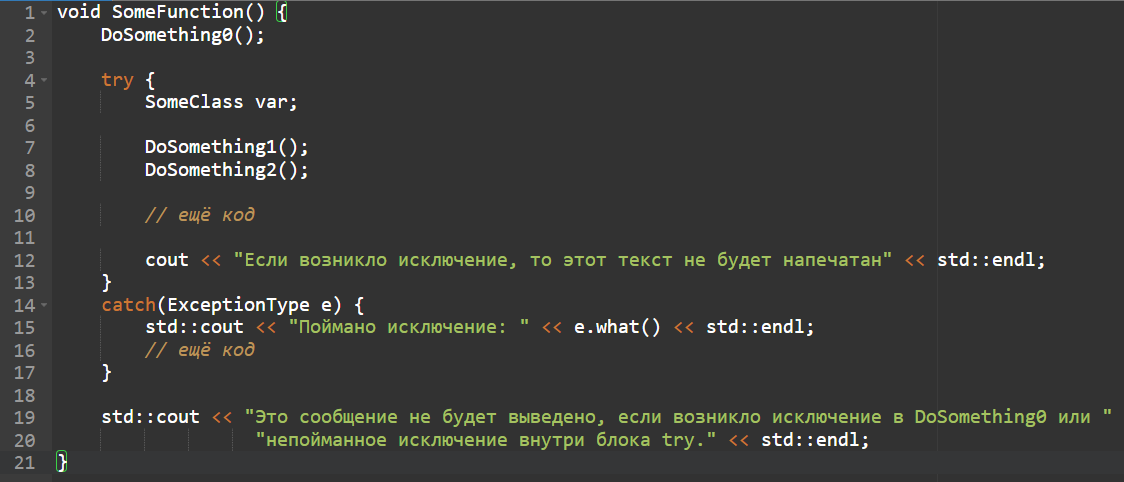
На помощь программисту приходят исключения (**exception**). Так называют объекты, которые хранят данные о возникшей проблеме. Механизмы исключений в разных языках программирования очень похожи. В зависимости от терминологии языка исключения либо выбрасывают (**throw**), либо генерируют (**raise**). Это происходит в тот момент, когда программа не может продолжать выполнять запрошенную операцию.

После выбрасывания в дело вступает системный код, который ищет подходящий обработчик. Особенность в том, что тот, кто выбрасывает исключение, не знает, кто будет его обрабатывать. Может быть, что и вовсе никто — такое исключение останется сиротой и приведёт к падению программы.

Если обработчик всё же найден, то он ловит (**catch**) исключение и программа продолжает работать как обычно. В некоторых языках вместо catch используется глагол **except** (исключить).

Обработчик ловит не все исключения, а только некоторые — те, что возникли в конкретной части определённой функции. Эту часть нужно явно обозначить, для чего используют конструкцию **try** (попробовать). Также обработчик не поймает исключение, которое ранее попало в другой обработчик. После обработки исключения программа продолжает выполнение как ни в чём не бывало.

Начнем с примера:

В примере есть один try-блок и один catch-блок. Если в блоке try возникает исключение типа ExceptionType, то выполнение блока заканчивается. При этом корректно удаляются созданные объекты — в данном случае переменная var. Затем управление переходит в конструкцию catch. Сам объект исключения передаётся в переменную e. Выводя e.what(), мы предполагаем, что у типа ExceptionType есть метод what.

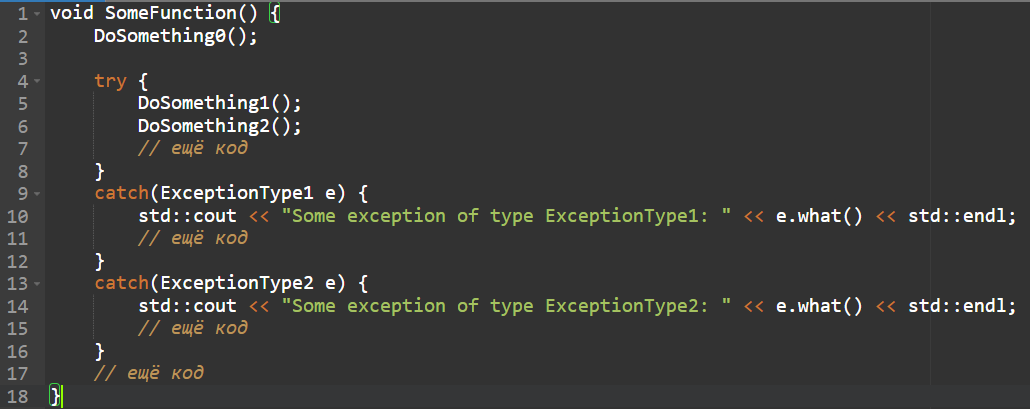
Если в блоке try возникло исключение другого типа, то управление также прервётся, но поиск обработчика будет выполняться за пределами функции SomeFunction — выше по стеку вызовов. Это также касается любых исключений, возникших вне try-блока.

Во всех случаях объект var будет корректно удалён.

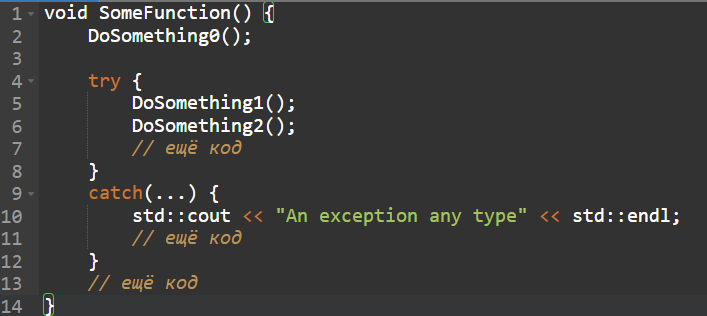
Исключение не обязано возникнуть непосредственно внутри DoSomething\*(). Будут обработаны исключения, возникшие в функциях, вызванных из DoSomething\*, или в функциях, вызванных из тех функций, да и вообще на любом уровне вложенности. Главное, чтобы исключение не было обработано ранее.

Ловим исключения нескольких типов:

Можно указать несколько блоков catch, чтобы обработать исключения разных типов:

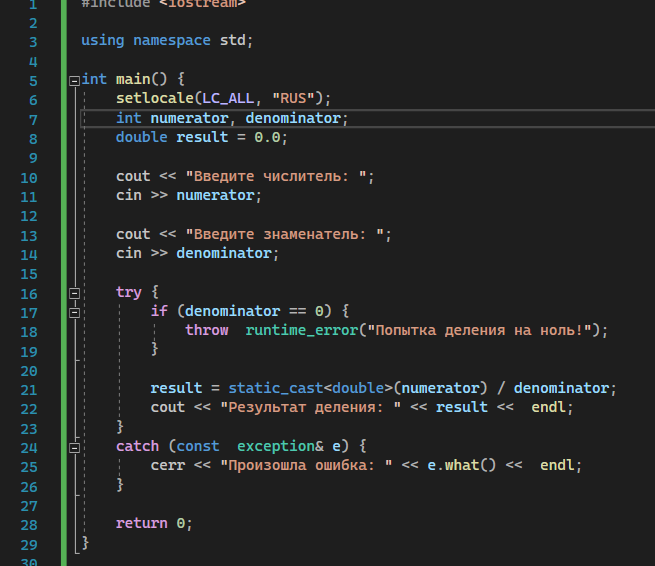


Ловим все исключения:



Если перед catch(...) есть другие блоки, то он означает «поймать все остальные исключения». Ставить другие catch-блоки после catch(...) не имеет смысла.

//// Добавить свой пример, а также разобратсья в вопросе использования try…



Источник: [https://tproger.ru/articles/iskljuchenija-v-cpp-tipy-sintaksis-i-obrabotka/#part1](https://tproger.ru/articles/iskljuchenija-v-cpp-tipy-sintaksis-i-obrabotka/%23part1)

# Typedef

В C++ typedef (сокращенно от «type definition», «определение типа») – это ключевое слово, которое создает псевдоним для существующего типа данных. Чтобы создать такой псевдоним, мы используем ключевое слово typedef, за которым следует существующий тип данных для псевдонима, за которым следует имя для псевдонима. Например:



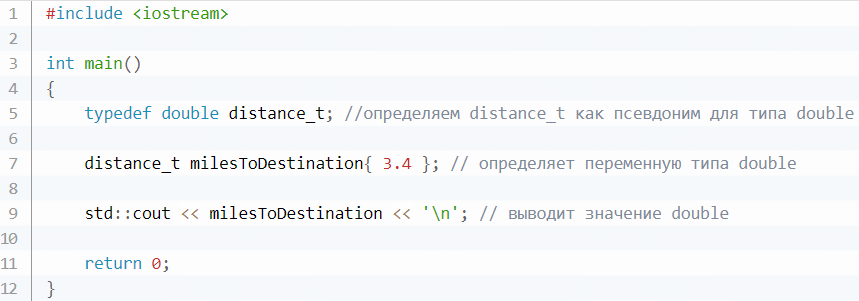
По соглашению имена typedef объявляются с использованием суффикса "\_t". Это помогает указать, что идентификатор представляет собой тип, а не переменную или функцию, а также помогает предотвратить конфликты имен с другими типами идентификаторов.

Называйте свои псевдонимы typedef с суффиксом \_t, чтобы указать, что это имя является псевдонимом типа, и чтобы помочь предотвратить конфликты имен с другими типами идентификаторов.

После определения имя typedef можно использовать везде, где требуется тип. Например, мы можем создать переменную с именем typedef в качестве типа:



Когда компилятор встречает имя typedef, он подставляет тип, на который указывает typedef. Например:



Этот код печатает:



В приведенной выше программе мы сначала определяем typedef distance\_t как псевдоним для типа double.

Затем мы определяем переменную с именем milesToDestination типа distance\_t. Поскольку компилятор знает, что distance\_t – это typedef, он будет использовать тип, на который указывает псевдоним, то есть double. Таким образом, переменная milesToDestination фактически компилируется как переменная типа double, и во всех отношениях она будет вести себя как double.

Наконец, мы печатаем значение milesToDestination, которое печатается как значение double.

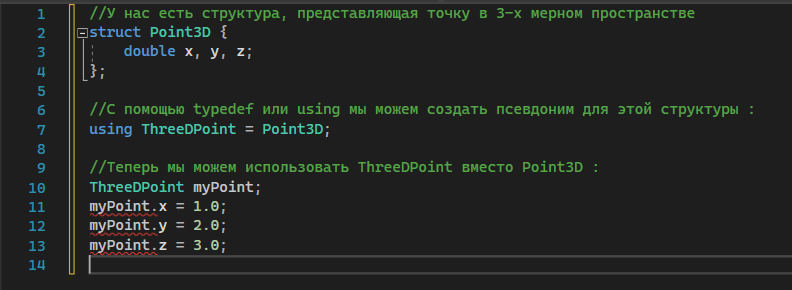
///Добавить про using

С C++11 появилась более современная и удобная альтернатива typedef - ключевое слово using. Оно позволяет определять псевдонимы типов более явным образом:

Синтаксис с using гораздо более читаем и интуитивно понятен, чем с **typedef**. При этом using не ограничивается только созданием псевдонимов для существующих типов данных. Он также позволяет создавать алиасы для шаблонов и указателей на функции:

Это делает код более модульным и улучшает его читаемость, что особенно важно при работе с шаблонами и сложными типами данных.

Пример использования

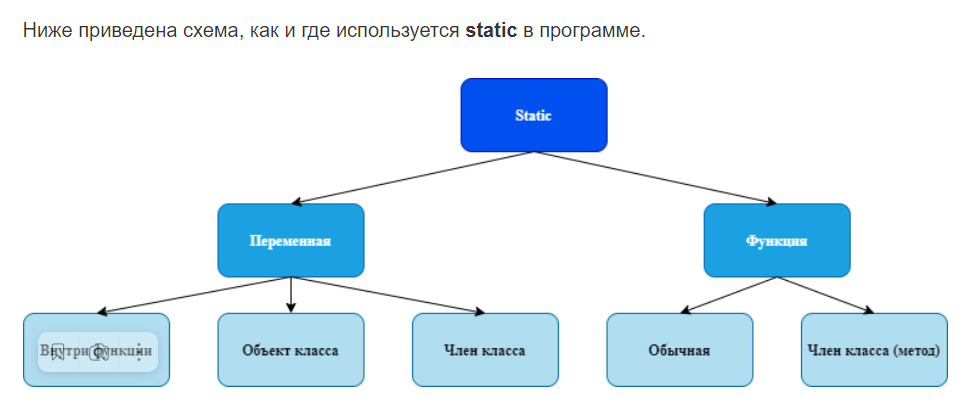


**typedef** и **using** - это мощные инструменты, которые позволяют улучшить читаемость и модульность кода в C++. Выбор между ними зависит от ваших предпочтений и стиля программирования. Однако в большинстве современных проектах рекомендуется использовать using из-за его более читаемого синтаксиса и более широких возможностей.

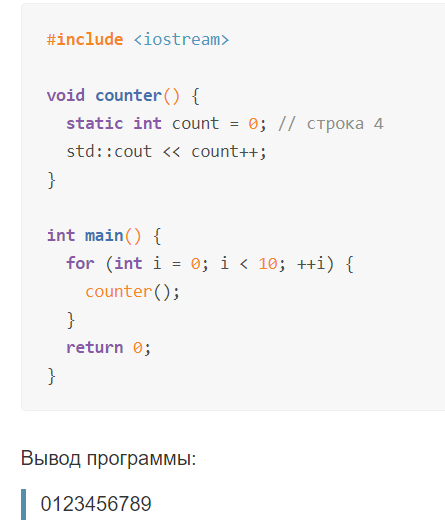
Источник: <https://radioprog.ru/post/1131>

# Static

**Static -** это ключевое слово в C++, используемое для придания элементу особых характеристик. Для статических элементов выделение памяти происходит только один раз и существуют эти элементы до завершения программы. Хранятся все эти элементы не в heap и не на stack, а в специальных сегментах памяти, которые называются *.data* и *.bss* (зависит от того инициализированы статические данные или нет). На картинке ниже показан типичный макет программной памяти.



Статические переменные при использовании внутри функции инициализируются только один раз, а затем они сохраняют свое значение. Эти статические переменные хранятся в статической области памяти (*.data или .bss*), а не в стеке, что позволяет хранить и использовать значение переменной на протяжении всей жизни программы.



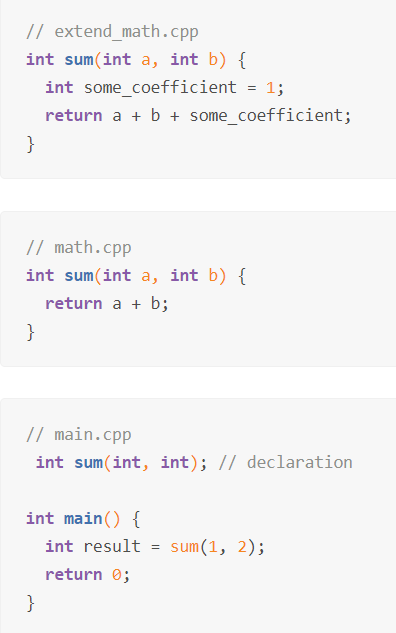
Если не использовать **static**в *строке 4*, выделение памяти и инициализация переменной count происходит при каждом вызове функции *counter()*, и уничтожается каждый раз, когда функция завершается. Но если мы сделаем переменную статической, после инициализации (при первом вызове функции *counter()*) область видимости *count*будет до конца функции *main()*, и переменная будет хранить свое значение между вызовами функции *counter()*.

Статический объект класса имеет такие же свойства как и обычная статическая переменная, описанная выше, т.е. хранится в *.data или .bss* сегменте памяти, создается на старте и уничтожается при завершении программы, и инициализируется только один раз. Инициализация объекта происходит, как и обычно — через конструктор класса.

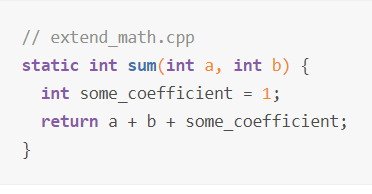
Статические функции пришли в С++ из С. По умолчанию все функции в С глобальные и, если вы захотите создать две функции с одинаковым именем в двух разных .c(.cpp) файлах одного проекта, то получите ошибку о том, что данная функция уже определена (*fatal error LNK1169: one or more multiply defined symbols found*).

Например, в классе яблоко есть поля вес и цвет, то у каждого объекта будут свои поля, если же мы объявим перменную static вес, то у всех объектов класса будет общий вес.

Может быть использовано в генерации айдишников с помощью static



Для того чтобы исправить данную проблему, одну из функций мы объявим статической. Например эту:

 В этом случае вы говорите компилятору, что доступ к статическим функциям ограничен файлом, в котором они объявлены. И он имеет доступ только к функции *sum()* из *math.cpp* файла. Таким образом, используя **static** для функции, мы можем ограничить область видимости этой функции, и данная функция не будет видна в других файлах, если, конечно, это не заголовочный файл (.h).

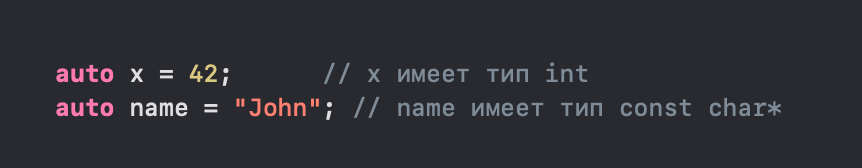
Как известно, мы не можем определить функцию в заголовочном файле не сделав ее **inline**или **static**, потому что при повторном включении этого заголовочного файла мы получим такую же ошибку, как и при использовании двух функций с одинаковым именем. При определении статической функции в заголовочном файле мы даем возможность каждому файлу (.cpp), который сделает #include нашего заголовочного файла, иметь свое собственное определение этой функции. Это решает проблему, но влечет за собой увеличение размера выполняемого файла, т.к. директива *include*просто копирует содержимое заголовочного файла в .cpp файл.

# Auto

Ключевое слово auto - это одна из ключевых особенностей C++11 и последующих стандартов, которая внесла существенные изменения в способ объявления переменных и повысила читаемость и гибкость кода. auto позволяет компилятору самостоятельно вывести тип переменной на основе значения, которое ей присваивается. Это упрощает код и снижает вероятность ошибок типизации.

**Основные аспекты использования** auto:

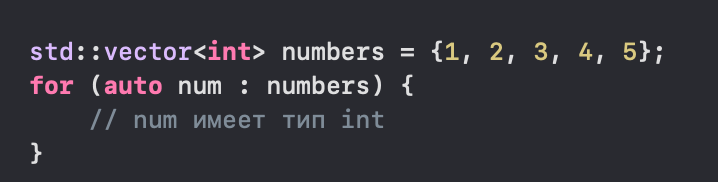
**1.Авто-вывод типа:** В простейшей форме, auto позволяет компилятору определить тип переменной на основе выражения, которое присваивается этой переменной. Например:



Это особенно полезно, когда тип данных сложно написать или изменяется часто.

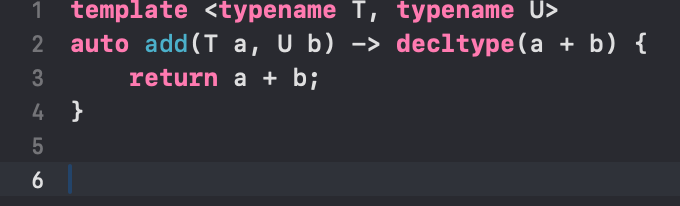
2. **Использование в циклах (диапазонный** for**):**

auto может быть использовано в циклах диапазона для более читаемого и гибкого кода:



3. **Авто и шаблоны:**

auto также полезно в сочетании с шаблонами, что позволяет создавать более универсальные и гибкие функции:



**Ограничения** auto

Хотя auto приносит множество преимуществ, есть и некоторые ограничения:

* Не всегда подходит для переменных, значения которых не могут быть однозначно определены (например, перегруженные функции).
* Нельзя использовать auto в объявлениях классов и структур.
* Иногда может усложнить чтение кода, особенно если не используются описательные имена переменных.

**Заключение**

auto предоставляет мощное средство для более читаемого и гибкого кода в C++. Правильное использование auto может значительно улучшить процесс разработки и облегчить поддержку кода, уменьшая вероятность ошибок типизации. Однако важно использовать его с умом и учитывать ограничения при проектировании своих программ.

# Include guard

1. В языке программирования C++, заголовочные файлы (header files) используются для определения интерфейсов классов, функций и переменных.

При использовании заголовочных файлов в нескольких местах вашего проекта, может возникнуть проблема множественного включения.

Include Guard (защита от повторного включения) - это механизм, который предотвращает множественное включение одного и того же заголовочного файла.

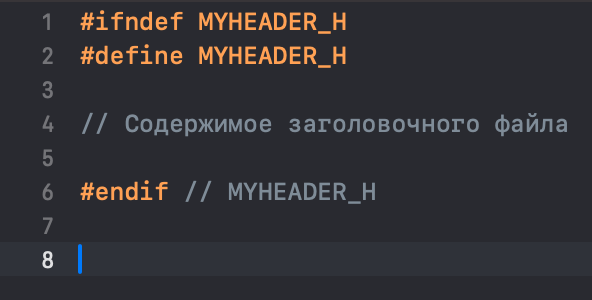
2. Принцип работы Include Guard:

- Include Guard реализуется с использованием директивы #ifndef, #define и #endif.

- Обычно, в начале каждого заголовочного файла создается макрос, который уникален для данного файла. Этот макрос обычно называется в стиле "NAMESPACED\_FILENAME\_H" (где "NAMESPACED" - это имя вашего проекта или пространства имен, "FILENAME" - имя самого файла, и "H" - расширение заголовочного файла).

-При первом включении файла макрос #ifndef установлен в истину, и файл включается в компиляцию.

-При последующих попытках включения файла, макрос уже определен (поскольку он был создан при первом включении), и код между #ifndef и #endif игнорируется компилятором.



В этом примере MYHEADER\_H - уникальный макрос, который предотвращает множественное включение заголовочного файла MyHeader.h.

4. Подсказки по использованию Include Guard:

- Следуйте соглашениям по именованию макросов для Include Guard, чтобы избежать конфликтов.

- Убедитесь, что макрос Include Guard находится в самом начале заголовочного файла, перед всеми другими директивами #include.

- Помните, что Include Guard не является частью стандарта C++, но является действенным соглашением в сообществе разработчиков.

5. Заключение:

Include Guard - это важный механизм в C++, который помогает избежать проблем с множественным включением заголовочных файлов.

Правильное использование Include Guard способствует поддерживаемости и чистоте вашего кода, делая его более надежным для использования в больших проектах.

Источники: <https://learn.microsoft.com/ru-ru/cpp/cpp/header-files-cpp?view=msvc-170>

# Источники

<https://www.bestprog.net/ru/2018/07/30/functions-arguments-by-default-in-functions_ru/>

<https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82_%D0%BF%D0%BE_%D1%83%D0%BC%D0%BE%D0%BB%D1%87%D0%B0%D0%BD%D0%B8%D1%8E>

<https://tproger.ru/articles/memory-model/>

<https://tproger.ru/translations/stanford-cpp-style-guide/>

<https://prog-cpp.ru/cpp-newdelete/>  
  
<https://habr.com/ru/articles/527044/>

<https://learn.microsoft.com/ru-ru/cpp/cpp/header-files-cpp?view=msvc-170>