# .Как уменьшить размер exe файла.

По умолчанию оптимизация отключена. Это приводит к самому быстрому времени компиляции, но компилятор не делает абсолютно никаких попыток оптимизировать, а сгенерированные программы значительно больше и медленнее, чем при включенной оптимизации. Существуют различные переключатели -O (разрешенные формы -O0, -O1 -O2, -O3 и -Os) в gcc для управления уровнем оптимизации:

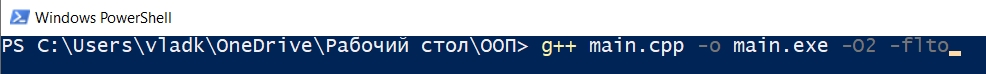
* -O0 Нет оптимизации; генерирует неоптимизированный код, но имеет самое быстрое время компиляции. Это значение по умолчанию.
* -O1 Оптимизация. Оптимизация компиляции занимает несколько больше времени, и много больше памяти для большой функции.
* -O2 GCC выполняет почти все поддерживаемые оптимизации, которые не связаны с компромиссом между пространством и скоростью. Компилятор не выполняет развёртывание цикла или встраивание функций при указании

Это самый высокий по-настоящему безопасный способ оптимизации. -O3 делает реорганизации, которые иногда могут быть хлопотными.

* -O3 Полная оптимизация как в -O2; также использует более агрессивную автоматическую накладку подпрограмм внутри объекта и пытается векторизировать циклы. Он также включает параметры -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload и -ftree-vectorize.
* Чаще всего, когда речь заходит об оптимизации размера, используется опция “-Os”, считается самой эффективной оптимизацией.   
  -Os Оптимизировать под размер. -Os включает все оптимизации -O2, которые обычно не увеличивают размер кода.
* “-flto” делает возможным удаление неиспользуемых функций.

Пример использования оптимизации с помощью –O2 –flto:







[GCC x86, как уменьшить размер кода](https://habr.com/ru/company/intel/blog/167417/)

[Значения –O](https://stackoverflow.com/questions/655382/meaning-of-gcc-o2)

# Static и extern.

В дополнение к области видимости и продолжительности жизни, переменные имеют еще одно свойство — связь. **Связь** **переменной** определяет, относятся ли несколько упоминаний одного идентификатора к одной и той же переменной или нет.

Переменная без связей — это переменная с локальной областью видимости, которая относится только к блоку, в котором она определена. Это обычные локальные переменные. Две переменные с одинаковыми именами, но определенные в разных функциях, не имеют никакой связи — каждая из них считается независимой единицей.

Переменная, имеющая внутренние связи, называется **внутренней переменной** (или ***«статической переменной»***). Она может использоваться в любом месте файла, в котором определена, но не относится к чему-либо вне этого файла.

Переменная, имеющая внешние связи, называется **внешней переменной**. Она может использоваться как в файле, в котором определена, так и в других файлах.

Если вы хотите сделать глобальную переменную внутренней (которую можно использовать только внутри одного файла) — используйте **ключевое слово static**:

#include <iostream>

static int g\_x; // g\_x - это статическая глобальная переменная, которую можно использовать только внутри этого файла

int main(){

    return 0;

}

Аналогично, если вы хотите сделать глобальную переменную внешней (которую можно использовать в любом файле программы) — используйте **ключевое слово extern**:

#include <iostream>

extern double g\_y(9.8); // g\_y - это внешняя глобальная переменная и её можно использовать и в других файлах программы

int main(){

    return 0;

}

По умолчанию, неконстантные переменные, объявленные вне блока, считаются внешними. Однако константные переменные, объявленные вне блока, считаются внутренними.

## Статические переменные внутри функции

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.

**Пример:**

#**include** <iostream>

**void** **counter**() {

**static** **int** count = 0; // строка 4

std::cout << count++;

}

**int** **main**() {

**for** (**int** i = 0; i < 10; ++i) {

counter();

}

**return** 0;

}

**Вывод программы:** 0123456789

**Вывод без слова static:** 0000000000

[Все о static](https://habr.com/ru/post/527044/?ysclid=l8z9i9zmor851601172)  
[О глобальных переменных, extern, static](https://ravesli.com/urok-49-globalnye-peremennye/" \l "toc-6)

# Устройство памяти программы.

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 x = 42; // определение статической глобальной переменной

int main(){

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

}

**Автоматическое выделение памяти** выполняется для [параметров функции](https://ravesli.com/urok-13-parametry-i-argumenty-funktsij/) и [локальных переменных](https://ravesli.com/urok-48-lokalnye-peremennye-oblast-vidimosti-i-vremya-zhizni/). Память выделяется при входе в блок, в котором находятся эти переменные, и удаляется при выходе из него.

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.

int main(){

int a = 3;

int result = factorial(a);

std::cout << result;

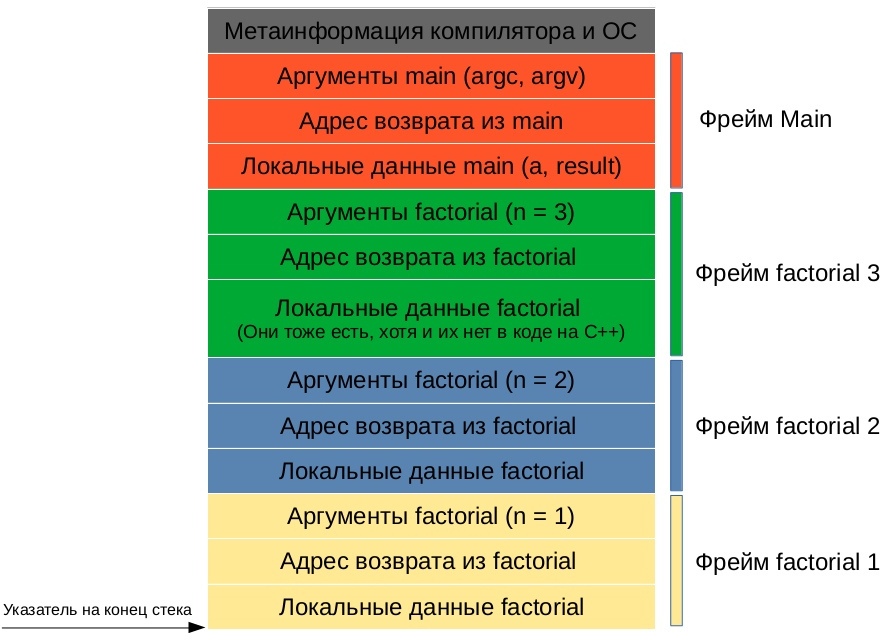
}

int factorial(int n){

if (n <= 1) return 1;

return n \* factorial(n - 1); // Стек заполняется значениями пока не наступит конец функции  
}

Стек при вызове последней рекурсивной функции будет выглядеть следующим образом:



**Динамическая, куча (heap)**    
Приложение при необходимости может запросить у ОС дополнительную память через аллокатор или напрямую через системный вызов. Пример использования динамической памяти с помощью аллокатора:

char \*i = malloc(sizeof(char)); // просим у аллокатора память для char

if (i != NULL) { // аллокатор может вернуть NULL (0)

\*i = 120; // делаем что-то с памятью, на которую указывает указатель i

printf("Чтение символа из выделенной памяти: %c\n", \*i);

free(i); // возвращаем память обратно аллокатору

}

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. Для возвращения памяти обратно в аллокатор необходим только сам указатель. Попытка использования уже очищенной памяти может привести к завершению программы с сигналом SIGSEGV.

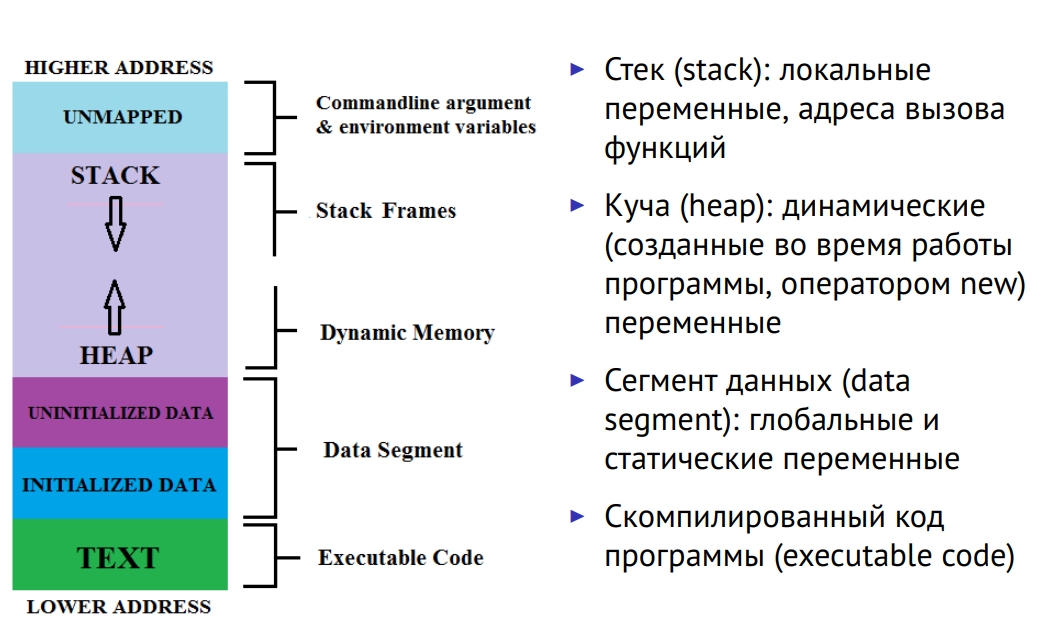
Аллокатор — это часть программы, которая запрашивает память большими кусками напрямую у ОС через системные вызовы (в Си это могут быть функции malloc()/free()), затем по частям отдаёт эту память приложению. Такой подход увеличивает производительность, но может вызвать фрагментацию памяти при длительной работе программы.

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)**. Поэтому нужно вручную освобождать память.

int \*ptr\_i; // Указатель на переменную  
double \*ptr\_d; // Указатель на голову массива  
…  
ptr\_i = new int;   
ptr\_d = new double[10];

…

delete ptr\_i; // Освобождение памяти  
delete[] ptr\_d;



[Память в языках программирования](https://tproger.ru/articles/memory-model/)  
[Динамическое выделение памяти](https://ravesli.com/urok-85-dinamicheskoe-vydelenie-pamyati-operatory-new-i-delete/)  
[Память в C++](https://github.com/mtrempoltsev/msu_cpp_lectures/blob/master/02.memory.md)

# Rvalue и lvalue.

lvalue = rvalue;

Выражения, которым можно присваивать, называются lvalue (left value, т. е. слева от знака равенства). Остальные выражения называются rvalue.

**glvalue** (“generalized” lvalue) - Выражение, чьё вычисление определяет сущность объекта, битового поля или функции.

**prvalue** (“pure” rvalue) - Выражение, чьё вычисление инициализирует объект, битовое поле или вычисляет значение операнда оператора, с соответствии с контекстом использования. Например, литералы 42, true, nullptr за исключением строковых литералов, которые являются *lvalue* выражениями.

**xvalue** (“eXpiring” lvalue) - Это *glvalue*, которое обозначает объект или битовое поле, чьи ресурсы могут быть повторно использованы (обычно потому, что они находятся около конца своего времени жизни). Например, результат вызова std::move, a[n],a.m, a.\*mp даёт выражение *xvalue*.

**Lvalue** - Это *glvalue*, которое не является *xvalue*. Например, имя переменной, функции или члена-данных, независимо от их типа, даже переменная, имеющая тип rvalue-ссылки, образует выражение *lvalue*.

[std::getline](http://en.cppreference.com/w/cpp/string/basic_string/getline)([std::cin](http://en.cppreference.com/w/cpp/io/cin), str), [std::cout](http://en.cppreference.com/w/cpp/io/cout) << 1, str1 = str2, ++it;

a = b, a += b, a %= b

**rvalue** - Это *prvalue* или *xvalue*.

Несколько правильных и неправильных способов использования значений lvalue и rvalues:

int main(){

int i, j, \*p;

// Правильное использование: переменная i является значением lvalue, а литерал 7 - значением prvalue.

i = 7;

// Неправильное использование: Левый операнд должен быть значением lvalue (C2106).`j \* 4` - это значение prvalue.

7 = i; // C2106

j \* 4 = 7; // C2106

// Правильное использование: разыменованный указатель является значением lvalue.

\*p = i;

// Неправильное использование: константа ci является неизменяемым значением lvalue (C3892).

const int ci = 7;

ci = 9; // C3892

}

// Неправильное присваивание функции: значение lvalue требуется в качестве левого операнда присваивания

int foo() { return 2; }

int main(){

foo() = 2;

return 0;

}

Не все присваивания результату вызова функции ошибочны. Например, использование ссылок в C++ делает это возможным:

int globalvar = 20;

int& foo(){

return globalvar;

}

int main(){

foo() = 10;

return 0;

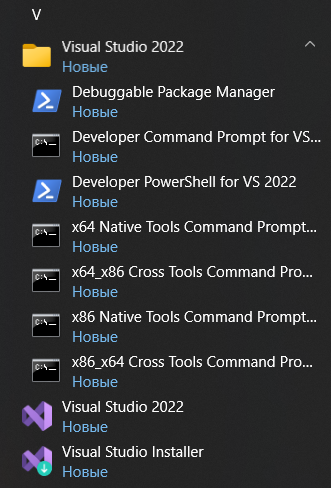
}

Здесь foo возвращает ссылку, *которая является lvalue*, то есть ей можно придать значение.

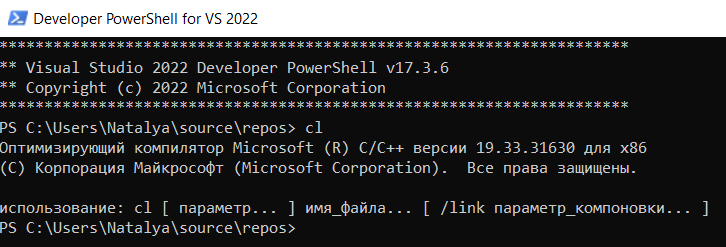
[Понимание lvalue и rvalue в C и С++](https://habr.com/ru/post/348198/)  
[Значения Lvalue и Rvalue (C++)](https://learn.microsoft.com/ru-ru/cpp/cpp/lvalues-and-rvalues-visual-cpp?view=msvc-170)  
[Ссылки r-value](https://ravesli.com/urok-190-ssylki-r-value/)  
[Перегрузка функций](https://ravesli.com/urok-102-peregruzka-funktsij/)

# Как использовать компилятор MSVC?

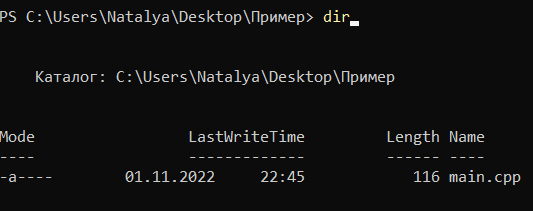
1. Установить IDE Visual Studio 2022.
2. Найти командную строку для разработки в меню пуск:



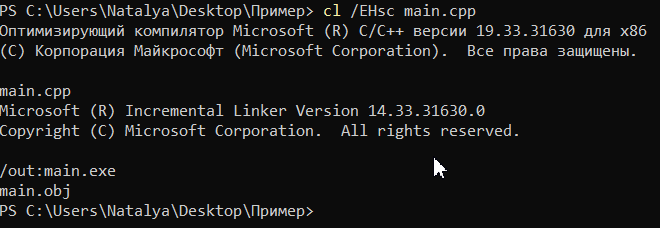
1. Проверить, что командная строка разработчика Visual C++ настроена правильно. В данном примере используется PowerShell for VS 2022.



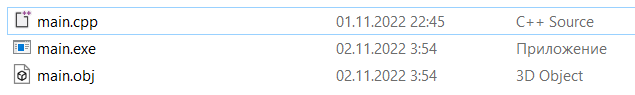
1. С помощью команды cd <путь до папки> перейти в папку, в которой будет программа.
2. Создать cpp файл с кодом и разместить в данную папку.
3. С помощью команды dir можно проверить содержимое папки. В ней cpp файл с кодом: main.cpp

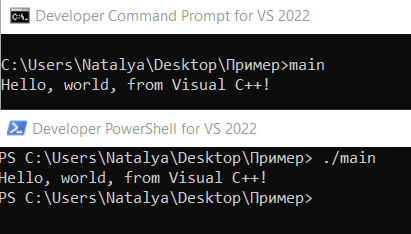


1. При помощи команды cl /EHsc <имя программы>.cpp компилируется программа.



Компилятор cl.exe создаст OBJ-файл, содержащий скомпилированный код, а затем запустит компоновщик для создания исполняемой программы с именем main.exe.

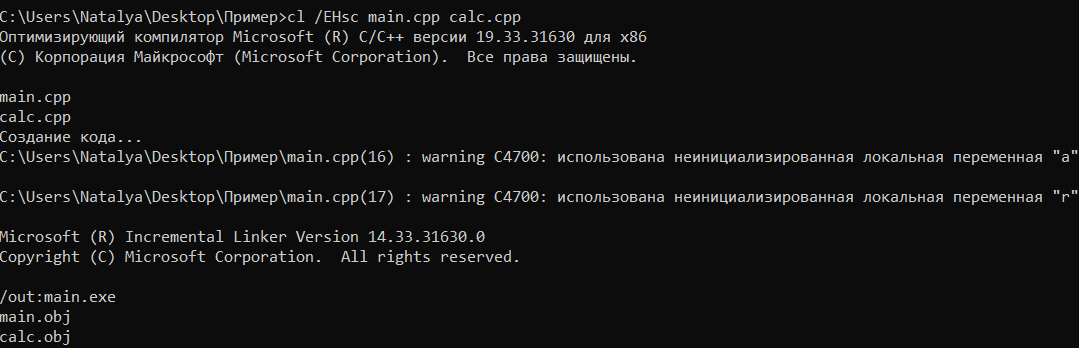


1. Для того чтобы запустить программу в PowerShell for VS нужно написать ./имя программы, а если командная строка Command Promt for VS, то нужно просто написать имя программы.  
   

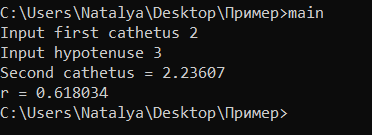
## Компиляция с модулем

1. Чтобы скомпилировать программу с дополнительными файлами исходного кода, нужно ввести их все в командной строке, например:

cl /EHsc file1.cpp file2.cpp file3.cpp

**

1. Запуск программы:



1. При указании дополнительных исходных файлов компилятор использует первый входной файл для создания имени программы. В этом случае выводится программа с именем file1.exe. Чтобы изменить имя на program1.exe, добавьте параметр компоновщика [/out](https://learn.microsoft.com/ru-ru/cpp/build/reference/out-output-file-name?view=msvc-160):

cl /EHsc file1.cpp file2.cpp file3.cpp /link /out:program1.exe

1. Чтобы автоматически перехватывать другие ошибки программирования, рекомендуется выполнить компиляцию с помощью порога предупреждений [/W3](https://learn.microsoft.com/ru-ru/cpp/build/reference/compiler-option-warning-level?view=msvc-160) или [/W4](https://learn.microsoft.com/ru-ru/cpp/build/reference/compiler-option-warning-level?view=msvc-160):

cl /W4 /EHsc file1.cpp file2.cpp file3.cpp /link /out:program1.exe

Параметр командной строки /EHsc указывает компилятору на необходимость стандартной обработки исключений C++. В противном случае созданные исключения могут привести к неуничтоженным объектам и утечкам ресурсов.

## Компиляция статической библиотеки в MSVC

**Статическая библиотека** — файл со скомпилированным исходным кодом, из которого выборочно или полностью код вставляется в программу (исполняемый файл) на этапе компоновки.

В Windows файлы статической библиотеки имеют расширение .a, в UNIX-подобных - .lib.

Статические библиотеки созданные на языке C++ как правило распространяются в паре с заголовочным файлом. Файл самой статической библиотеки -- это скомпилированный cpp файл.

Другой вид библиотек -- *динамические* (.dll в Windows, .so в UNIX-подобных ). Эти библиотеки не становятся частью исполняемого файла программы, а подключаются во время запуска или во время выполнения программы.

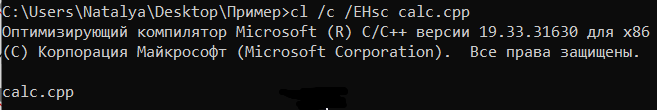
Исходные коды:

main.cpp - основной файл программы

calc.h, calc.cpp - исходные коды статический библиотеки

1. Создание объектного файла > calc.obj

cl /c /EHsc calc.cpp

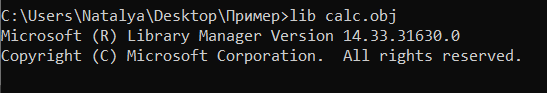


/c - компиляция без линковки

/EHsc - настройки обработки исключений

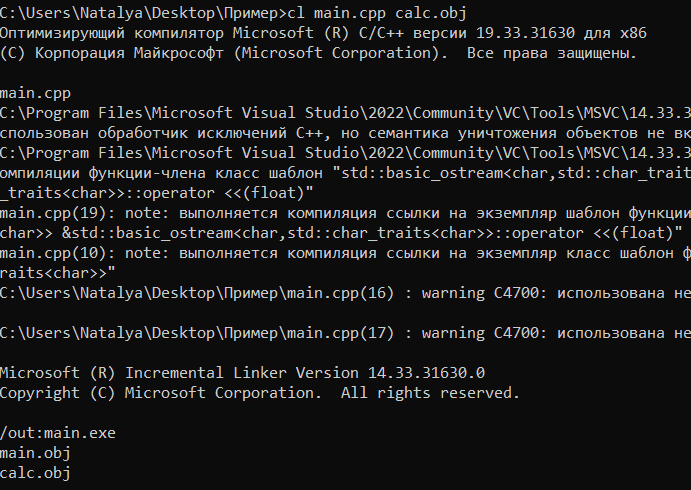
1. Создание статический библиотеки > calc.lib

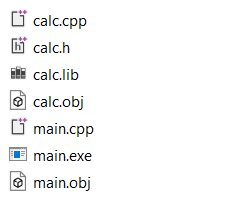
lib calc.obj



1. Компиляция программы со статической библиотекой > main.exe

cl main.cpp calc.lib или cl main.cpp calc.obj ???





[Компиляция собственной программы на языке C++ из командной строки](https://learn.microsoft.com/ru-ru/cpp/build/walkthrough-compiling-a-native-cpp-program-on-the-command-line?view=msvc-160)  
[Компиляция статической библиотеки](https://github.com/VetrovSV/OOP/tree/master/examples/example_libs/simple_lib)

# Auto.

Начиная с C++11 **ключевое слово auto** при инициализации переменной может использоваться вместо типа переменной, чтобы сообщить компилятору, что он должен присвоить тип переменной исходя из инициализируемого значения. Это называется **выводом типа** (или ***«автоматическим определением типа данных компилятором»***). Например:

auto x = 20; // int   
auto y = 3.14159; // float   
auto z = "gues type"; // char\*   
auto x = 4.0; // 4.0 - это литерал типа double   
auto y = 3 + 4; // выражение 3 + 4 обрабатывается как целочисленное, поэтому и переменная y должна быть типа int   
auto a; // ошибка! Не задано значение!

В функиях:

int subtract(int a, int b){  
 return a - b;

}

int main(){

    auto result = subtract(4, 3); // функция subtract() возвращает значение типа int и, следовательно, переменная result также должна быть типа int

    return 0;

}

В C++14 функционал ключевого слова auto был расширен до автоматического определения типа возвращаемого значения функции. Например:

auto subtract(int a, int b){  
 return a - b;

}

Так как выражение a − b является типа int, то компилятор делает вывод, что и функция должна быть типа int.

[Вывод типов: ключевое слово auto](https://ravesli.com/urok-62-klyuchevoe-slovo-auto-vyvod-tipov/)

# Совместный цикл. For-each.

Существует более простой и безопасный тип цикла, называемый циклом **for-each** (также называемый **циклом for на основе диапазона**) для случаев, когда мы хотим перебрать все элементы в массиве (или другой структуре типа списка).

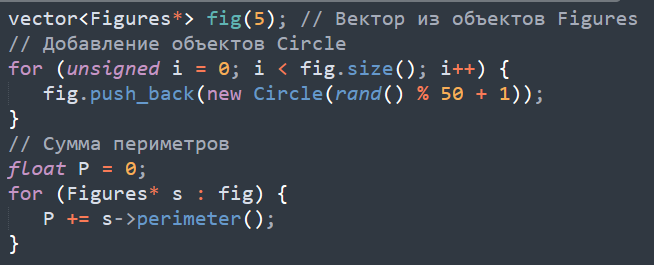
Синтаксис оператора for-each выглядит следующим образом:

for (объявление\_элемента : массив)

инструкция;

Когда встречается этот оператор, цикл будет перебирать все элементы в массиве, присваивая значение текущего элемента массива переменной, объявленной в объявление\_элемента. Для достижения наилучших результатов объявление\_элемента должно иметь тот же тип, что и элементы массива, в противном случае будет выполнено преобразование типа.

**Пример для вектора:**



В объявлениях элементов циклов for-each, если ваши элементы не принадлежат базовым типам, используйте ссылки или константные ссылки для повышения производительности.

**Например:**

std::string array[]{ "peter", "likes", "frozen", "yogurt" };

// element является константной ссылкой на текущий итерируемый элемент массива

for (const auto& element: array){

std::cout << element << ' ';

}

[Циклы for-each (циклы на основе диапазона)](https://radioprog.ru/post/1180)

# Файлы и папки проекта VisualStudio.

В Visual Studio используются два типа файлов (SLN и SUO) для хранения параметров, связанных с решениями:

* .sln - организует проекты, элементы проектов и решений в решении.
* .suo - сохраняет настройки и параметры уровня пользователя, например точки останова.

Остальные файлы:

* .cpp, .с - основные файлы исходного кода приложения.
* .h - Файл заголовка.
* .vcxproj - файл проекта . Хранит информацию, относящуюся к каждому проекту. (В более ранних версиях этот файл был назван Projname.vcproj или Projname.dsp.)
* .vcxproj.filters - файл фильтров. Указывает, куда поместить файл, который добавляется в решение. Например, H-файл помещается в узел Файлы заголовков.
* .vcxproj.user - файл пользовательских настроек, связанных с проектом.
* Readme.txt - файл сведений. Создается мастером приложений и описывает файлы в проекте.
* .vcxitems - файл проекта общих элементов. Этот проект не создается. Вместо этого на него может сослаться другой проект C++, и его файлы станут частью процесса сборки ссылающегося проекта. Это можно использовать для совместного использования общего кода в кроссплатформенных проектах C++.
* .sdf - файл базы данных просмотра. Поддерживает возможности просмотра и навигации, такие как Перейти к определению, Найти все ссылки и Представление классов. Создается путем анализа файлов заголовков.
* .OBJ - объектные файлы — скомпилированные, но не скомпонованные.

Папка .vs: Visual Studio требуется папка для хранения открытых документов, точек останова и другой информации о состоянии вашего решения, это означает, что он содержит типичные файлы, такие как,

* Файл .suo
* Временные кэши, используемые Roslyn для IntelliSense.
* Многие другие возможные файлы.

Папка x64 хранит конфигурации решения Release и Debug, в которых хранится .exe, .pdb ( файл, который содержит отладочную информацию) и так далее.

[Что такое решения и проекты VS?](https://learn.microsoft.com/ru-ru/visualstudio/ide/solutions-and-projects-in-visual-studio?view=vs-2022)

[Файлы и проекта решения](https://learn.microsoft.com/ru-ru/cpp/build/reference/project-and-solution-files?view=msvc-170)

[Типы файлов, создаваемые для проектов Visual Studio C++](https://learn.microsoft.com/ru-ru/cpp/build/reference/file-types-created-for-visual-cpp-projects?view=msvc-160&viewFallbackFrom=vs-2019)

[Debug/Release](https://translated.turbopages.org/proxy_u/en-ru.ru.4001e239-63808d15-0caf72a4-74722d776562/https/stackoverflow.com/questions/933739/what-is-the-difference-between-release-and-debug-modes-in-visual-studio)

[Для чего используется папка .vs в решениях Visual Studio?](https://translated.turbopages.org/proxy_u/en-ru.ru.abaab83d-63808b95-827bb5c8-74722d776562/https/stackoverflow.com/questions/48897191/what-is-the-vs-folder-used-for-in-visual-studio-solutions)

[Папка .vs](https://ru.stackoverflow.com/questions/743757/Папка-vs-весит-слишком-много-visual-studio-2017)

# Header guards и #pragma once

Довольно легко можно попасть в ситуацию, когда определения одних и тех же заголовочных файлов будут подключаться больше одного раза в файл .cpp. Очень часто это случается при подключении одного заголовочного файла другим.

## Header guards

**Header guards** — это директивы [условной компиляции](https://ravesli.com/urok-22-preprotsessor-direktivy/#toc-4), которые состоят из следующего:

#ifndef SOME\_UNIQUE\_NAME\_HERE

#define SOME\_UNIQUE\_NAME\_HERE

// Основная часть кода

#endif

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

Все ваши заголовочные файлы должны иметь header guards. SOME\_UNIQUE\_NAME\_HERE может быть любым идентификатором, но, как правило, в качестве идентификатора используется имя заголовочного файла с окончанием \_H. Например, в файле math.h идентификатор будет MATH\_H:

#ifndef MATH\_H

#define MATH\_H

int getSquareSides(){

    return 4;

}

#endif

## #pragma once

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

#pragma once

// основная часть кода

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

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

[Header guards и #pragma once](https://ravesli.com/urok-23-header-guards/#toc-0)

[Заголовочные файлы](https://ravesli.com/urok-21-zagolovochnye-fajly/)

[Защита от множественного подключения файлов](https://ru.stackoverflow.com/questions/302878/Защита-от-множественного-подключения-заголовочного-файла)

[#ifndef/#define VS #pragma once](https://ru.stackoverflow.com/questions/630573/ifndef-define-vs-pragma-once)

# Этапы компиляции программы на С++

Компилирование с++ файла с использованием компилятора g++.

**Этапы компиляции:**

1. Препроцессор — это макро процессор, который преобразовывает вашу программу для дальнейшего компилирования.

* Замена комментариев пустыми строками
* Текстовое включение файлов — #include
* Макроподстановки — #define
* Обработка директив условной компиляции — #if, #ifdef, #elif, #else, #endif

1. Компиляция

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.

1. 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.

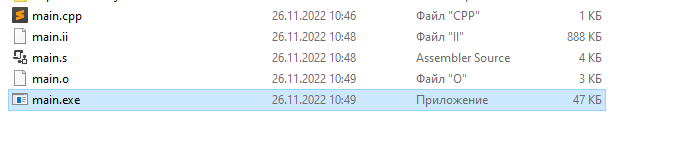
**Объектный файл** — это созданный ассемблером промежуточный файл, хранящий кусок машинного кода. Этот кусок машинного кода, который еще не был связан вместе с другими кусками машинного кода в конечную выполняемую программу, называется объектным кодом.

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

1. 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.



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