# 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:

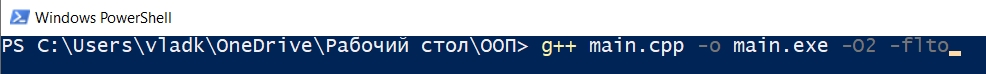
* -O0 Нет оптимизации; генерирует неоптимизированный код, но имеет самое быстрое время компиляции. Это значение по умолчанию.
* -O1 Оптимизация. Оптимизация компиляции занимает несколько больше времени, и много больше памяти для большой функции.
* -O2 GCC выполняет почти все поддерживаемые оптимизации, 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.

[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;

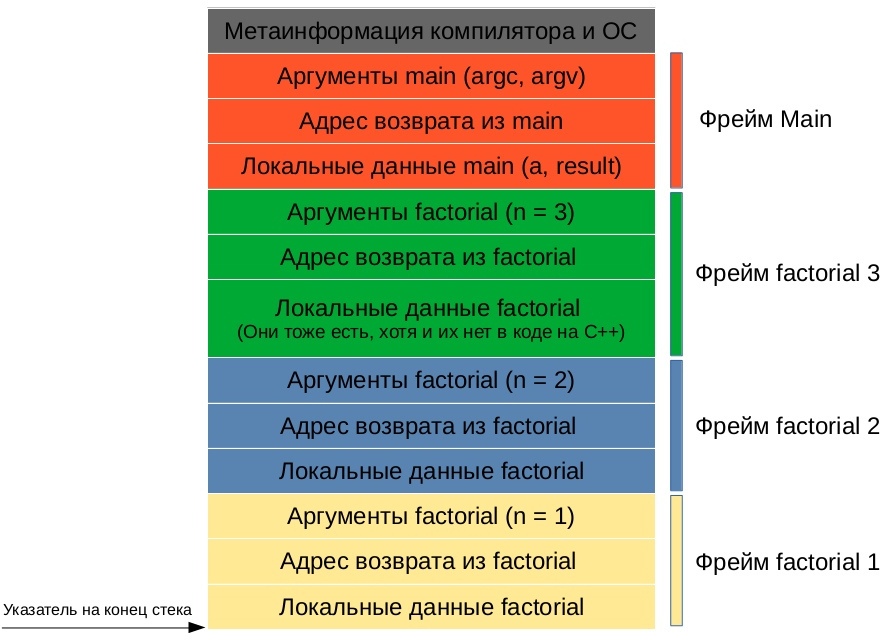
}

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.

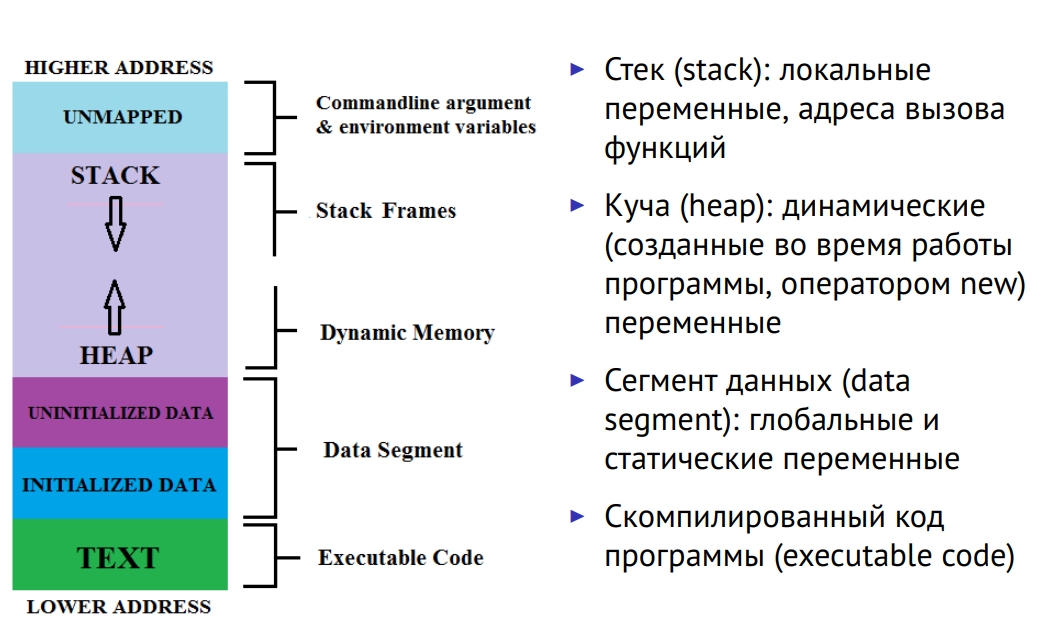
Аллокатор — это часть программы, которая запрашивает память большими кусками напрямую у ОС через системные вызовы (в Си это могут быть функции malloc()/free()), затем по частям отдаёт эту память приложению. Такой подход увеличивает производительность, но может вызвать фрагментацию памяти при длительной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

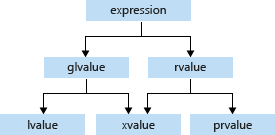


[Память в языках программирования](https://tproger.ru/articles/memory-model/)  
[Динамическое выделение памяти](https://ravesli.com/urok-85-dinamicheskoe-vydelenie-pamyati-operatory-new-i-delete/)  
[Память в C++](https://github.com/mtrempoltsev/msu_cpp_lectures/blob/master/02.memory.md)

# 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;

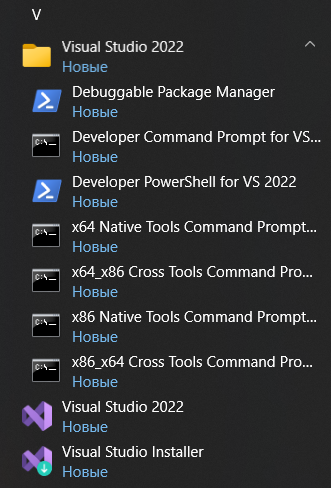
}

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

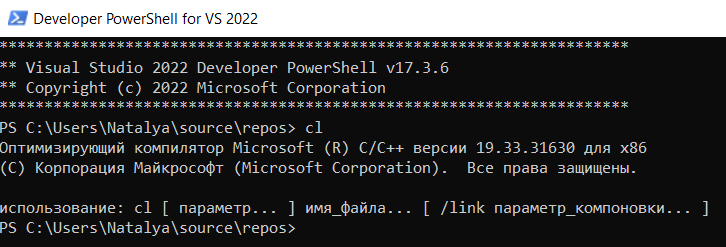
[Понимание lvalue и rvalue в C и С++](https://habr.com/ru/post/348198/)  
[Значения Lvalue и Rvalue (C++)](https://learn.microsoft.com/ru-ru/cpp/cpp/lvalues-and-rvalues-visual-cpp?view=msvc-170)  
[Ссылки r-value](https://ravesli.com/urok-190-ssylki-r-value/)  
[Перегрузка функций](https://ravesli.com/urok-102-peregruzka-funktsij/)

# Как использовать компилятор MSVC?

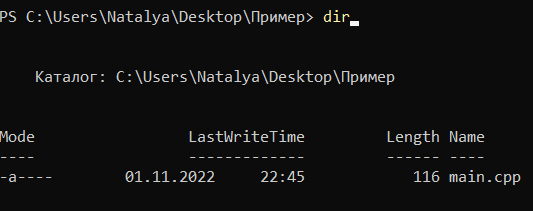
1. Установить IDE Visual Studio 2022.
2. Найти командную строку для разработки в меню пуск:



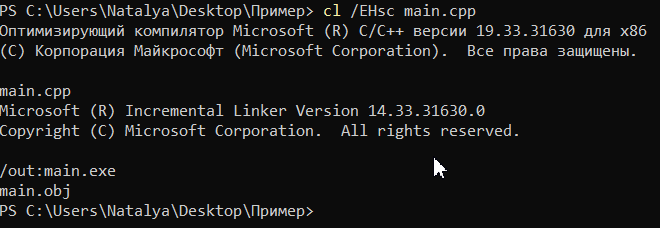
1. Проверить, что командная строка разработчика Visual C++ настроена правильно. В данном примере используется PowerShell for VS 2022.



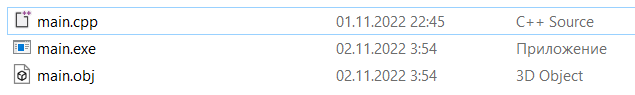
1. С помощью команды cd <путь до папки> перейти в папку, в которой будет программа.
2. Создать cpp файл с кодом и разместить в данную папку.
3. С помощью команды dir можно проверить содержимое папки. В ней cpp файл с кодом: main.cpp

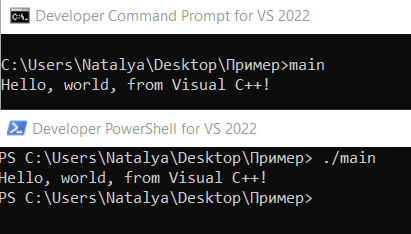


1. При помощи команды cl /EHsc <имя программы>.cpp компилируется программа.



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

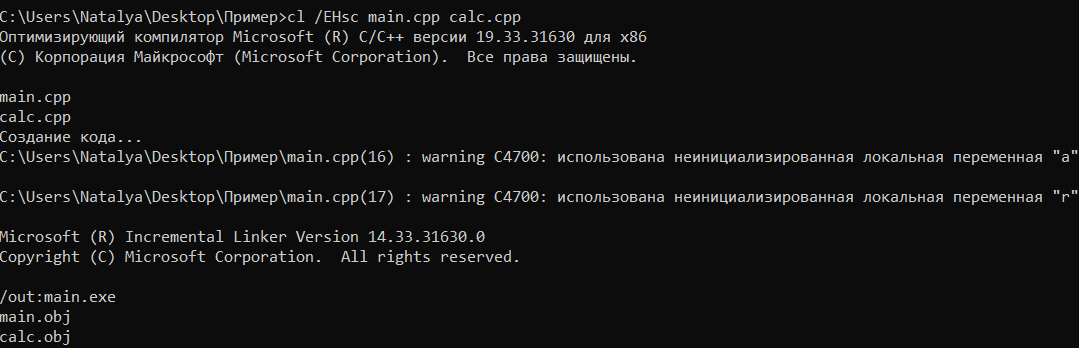


1. Для того чтобы запустить программу в PowerShell for VS нужно написать ./имя программы, а если командная строка Command Promt for VS, то нужно просто написать имя программы.  
   

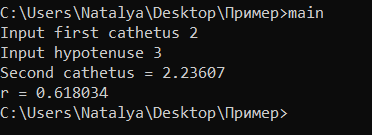
## Компиляция с модулем

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.

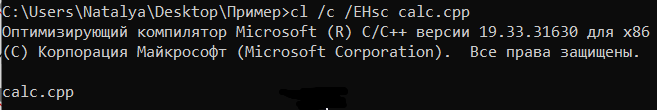
Исходные коды:

main.cpp - основной файл программы

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

1. Создание объектного файла > calc.obj

cl /c /EHsc calc.cpp

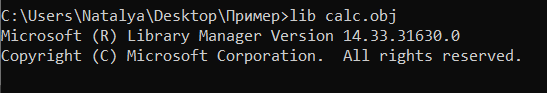


/c - компиляция без линковки

/EHsc - настройки обработки исключений

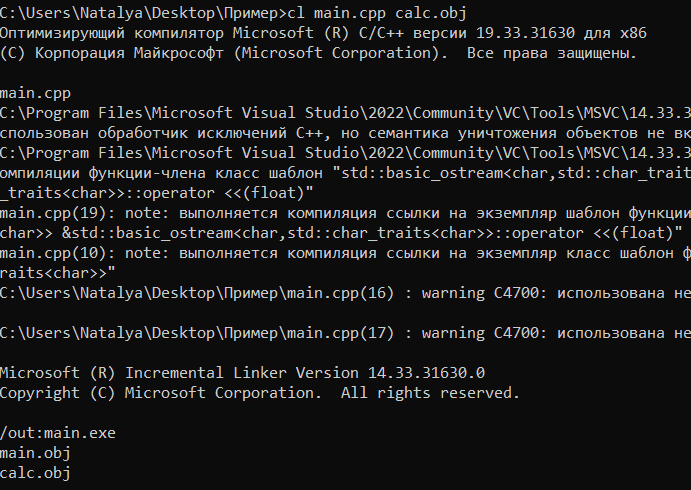
1. Создание статический библиотеки > calc.lib

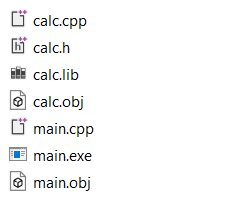
lib calc.obj



1. Компиляция программы со статической библиотекой > main.exe

cl main.cpp calc.lib ???





[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 и анонимные функции. Статическая библиотека