Ксения Мальцева, студентка ВМК-21-1

Отчеты по ООП

Оглавление

[Отчет№1. Компилирование программ в консоли PowerShell 2](#_Toc118582486)

[Отчет№2.Устройство памяти программы. Где хранятся: глобальные переменные, локальные переменные, аргументы функций, динамические переменные? 4](#_Toc118582487)

[Отчет№3. Рекомендации по оформлению кода 6](#_Toc118582488)

[Отчет№4. Как компилировать в командной строке несколько файлов? 10](#_Toc118582489)

[Отчет№5. Значения аргументов функции по умолчанию 12](#_Toc118582490)

[Отчет№6.Уменьшение размера исполняемого файла, создаваемого компилятором MinGW 14](#_Toc118582491)

[Отчет№7. Работа с аргументами командной строки в программах на языке C++ 16](#_Toc118582492)

[Отчет№8. Обработка исключительных ситуации в языке C++ 20](#_Toc118582493)

[Отчет№9. Понимание lvalue и rvalue в C и С++ 25](#_Toc118582494)

[Отчет№10. Ключевое слово typedef 28](#_Toc118582495)

[Отчет№11.Static 32](#_Toc118582496)

[Отчет№12. Работа с системой управления версиями – git 36](#_Toc118582497)

[Отчет№13. Защита от повторного подключения заголовочных файлов. 45](#_Toc118582498)

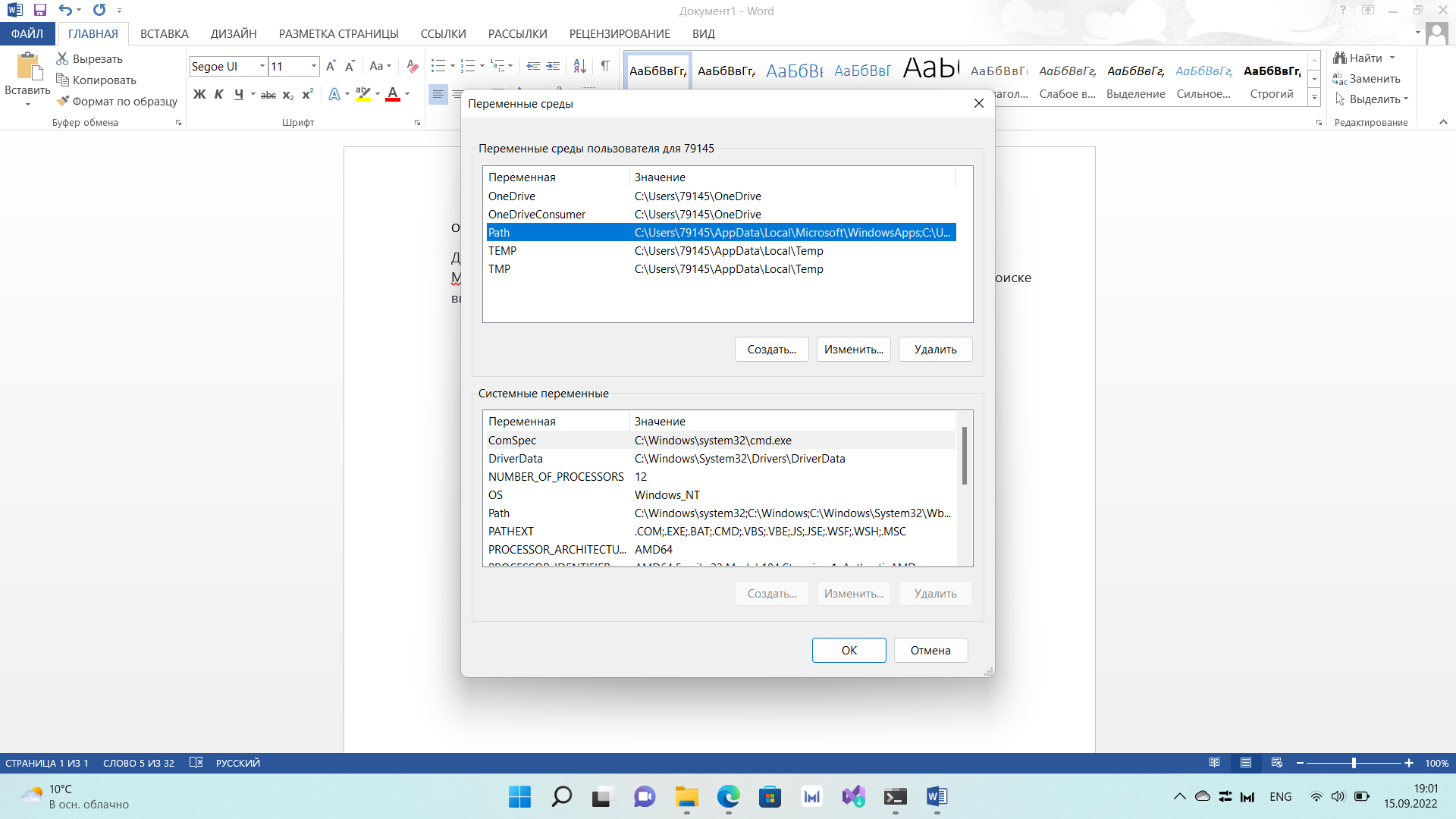
[Отчет№14. Рефакторинг. Средства рефакторинга в C++ 47](#_Toc118582499)

[Отчет№15. Этапы компиляции 51](#_Toc118582500)

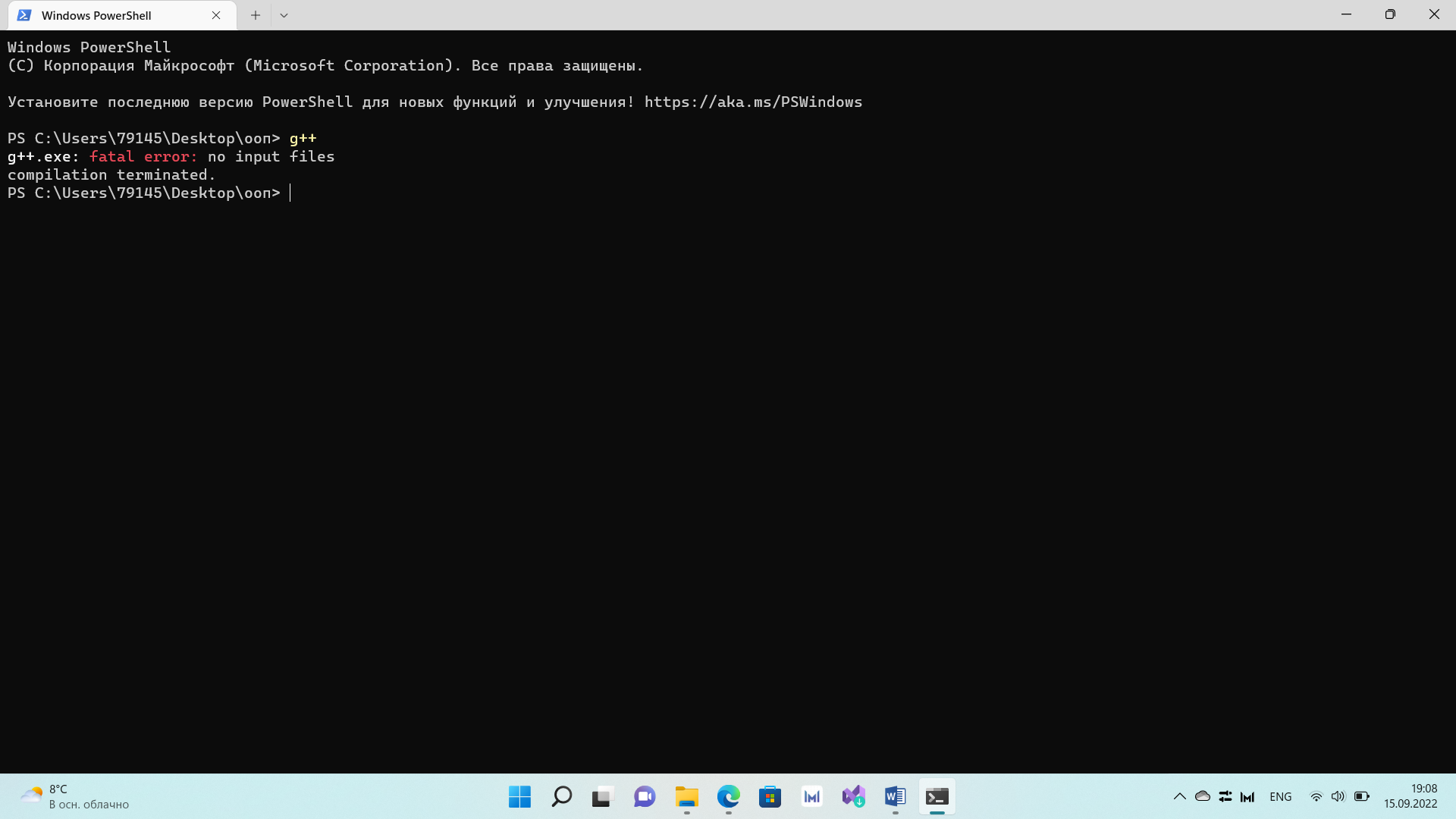
# Отчет№1. Компилирование программ в консоли PowerShell

Для компилирования программ через консоль необходимо установить компилятор MinGW.

Далее устанавливаем данный компилятор в переменную Path. Для этого в поиске вводим “Изменение переменных среды текущего пользователя”, выбираем пункт “Path” и в следующем окне создаем путь к папке, куда установлен компилятор, обычно это C:\MinGW\bin.

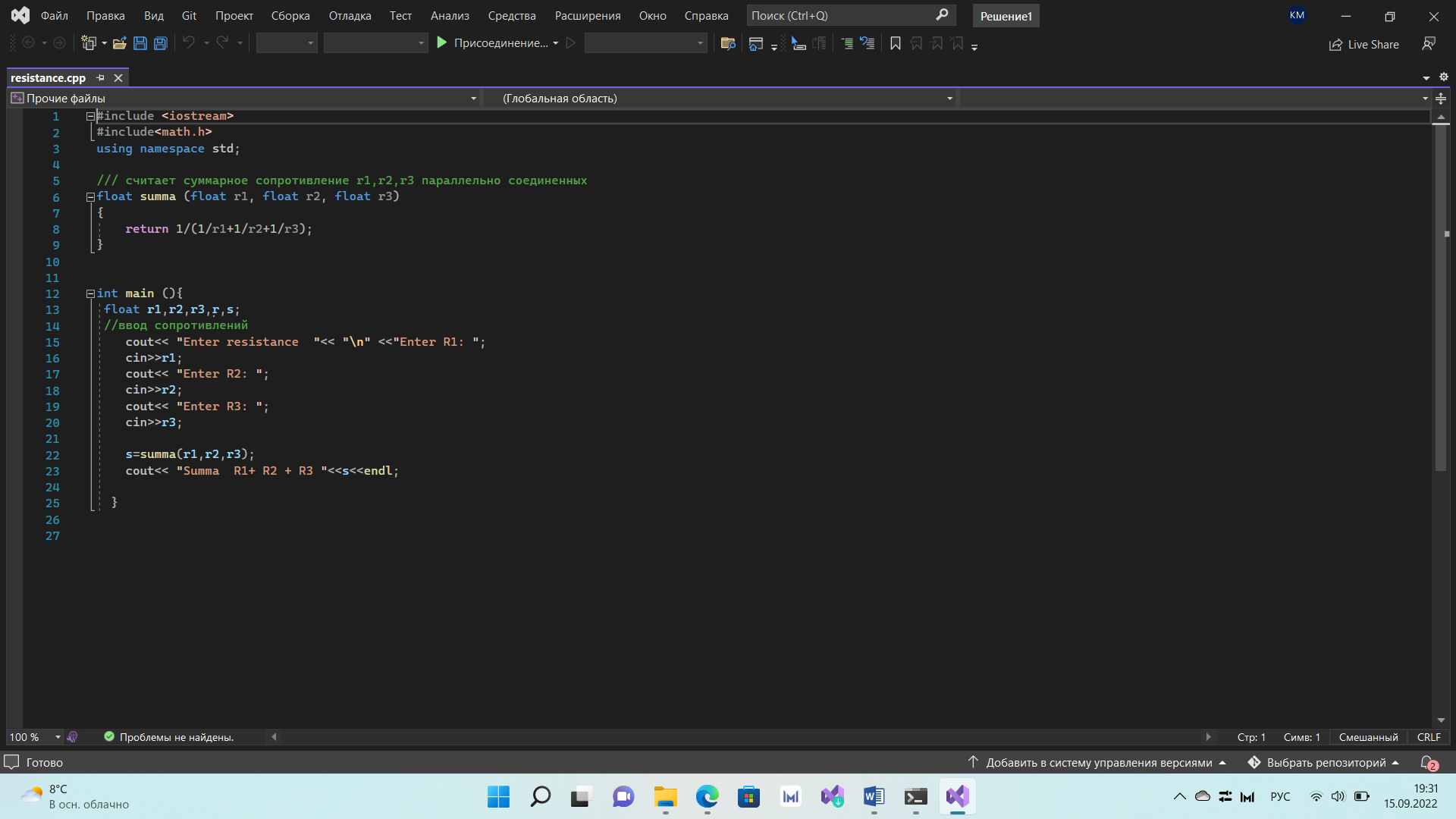


Включаем консоль Windows PowerShell и проверяем, что компилятор g++ работает.



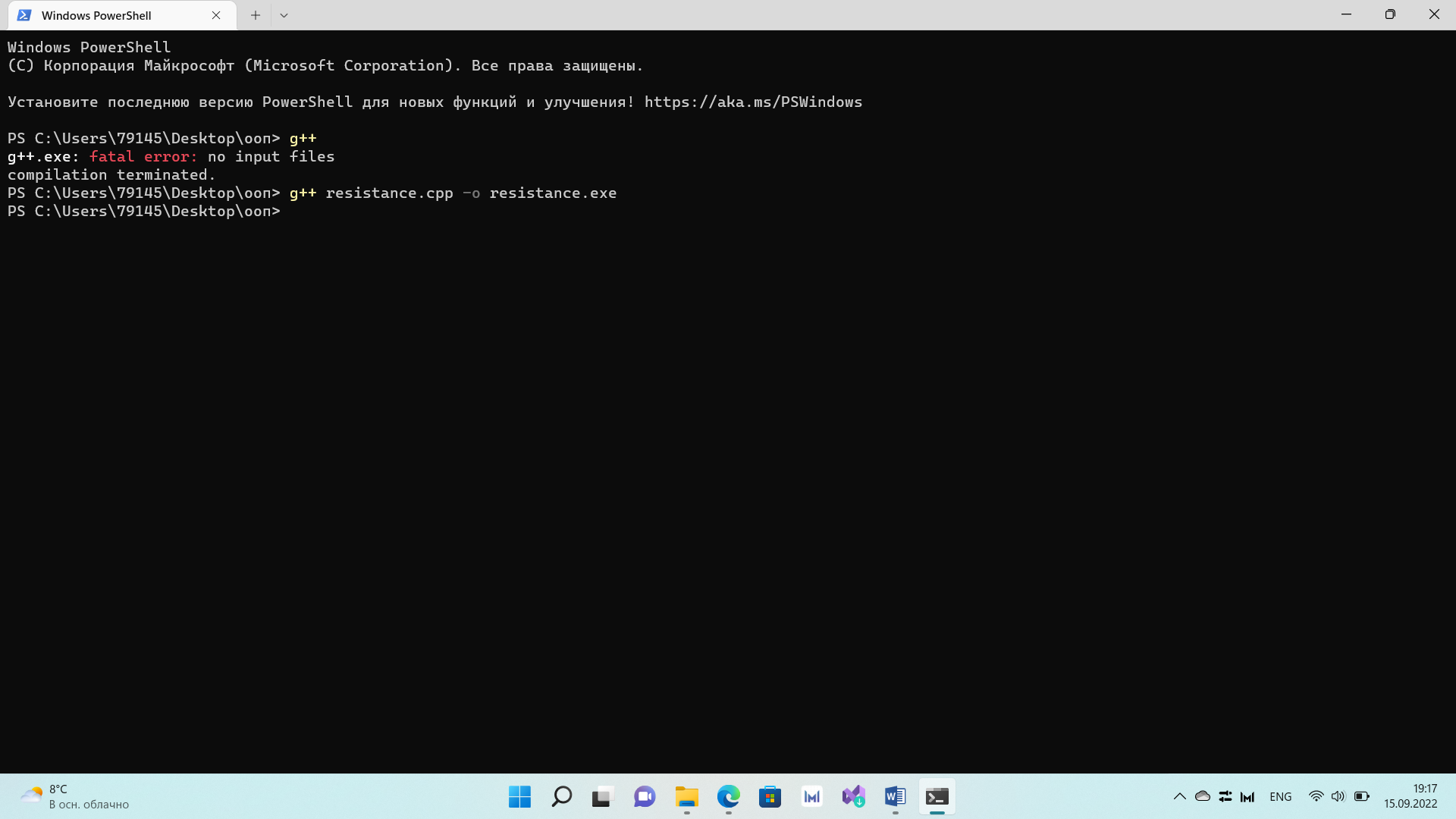
Если появилась данная ошибка, то компилятор успешно установлен.

Далее создаем новую папку, а в ней файл с расширением cpp. ; пишем код в данном файле.

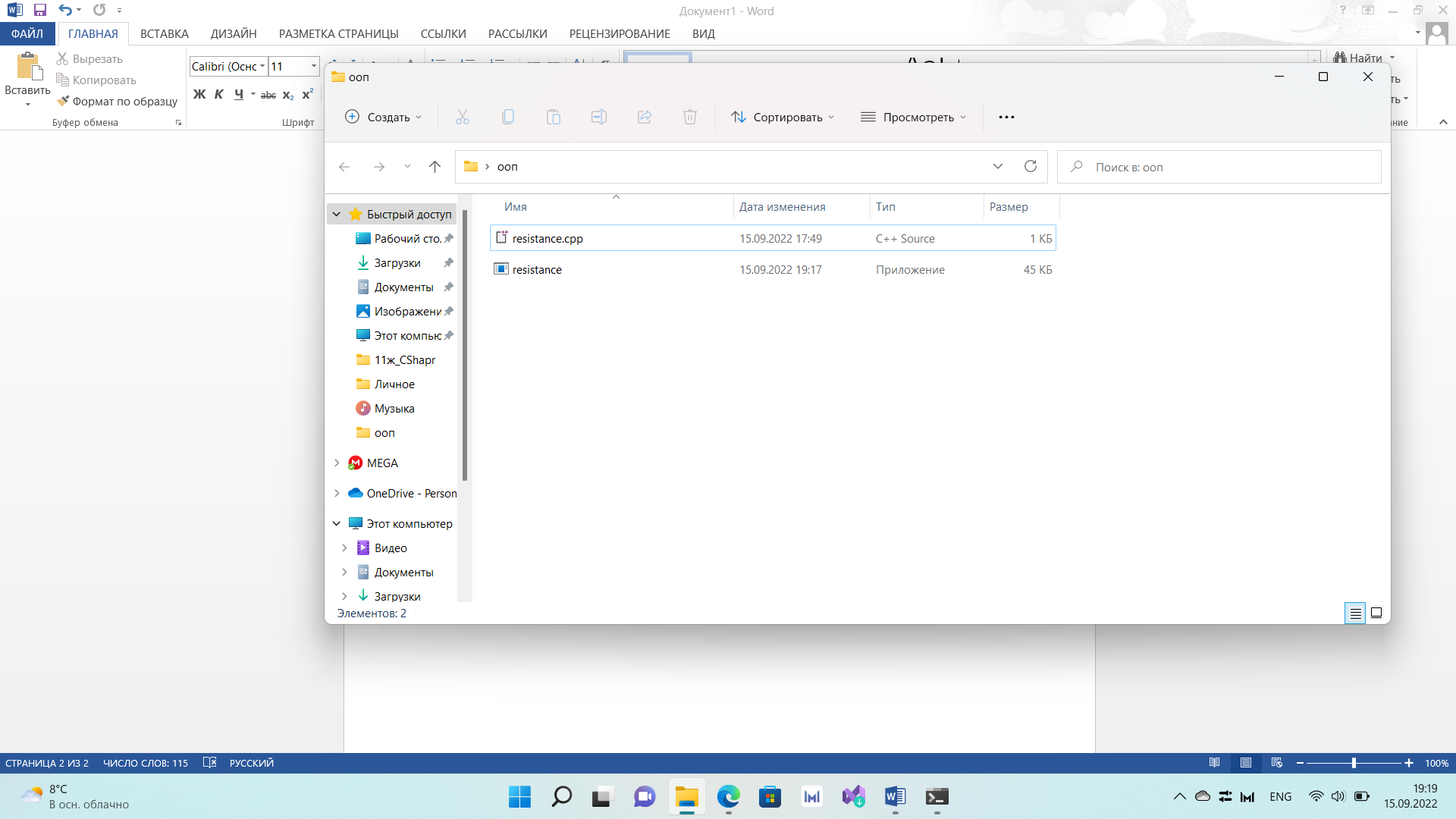


В данной папке зажимаем “Shift”+ правая кнопка мыши, чтобы запустить Windows PowerShell.

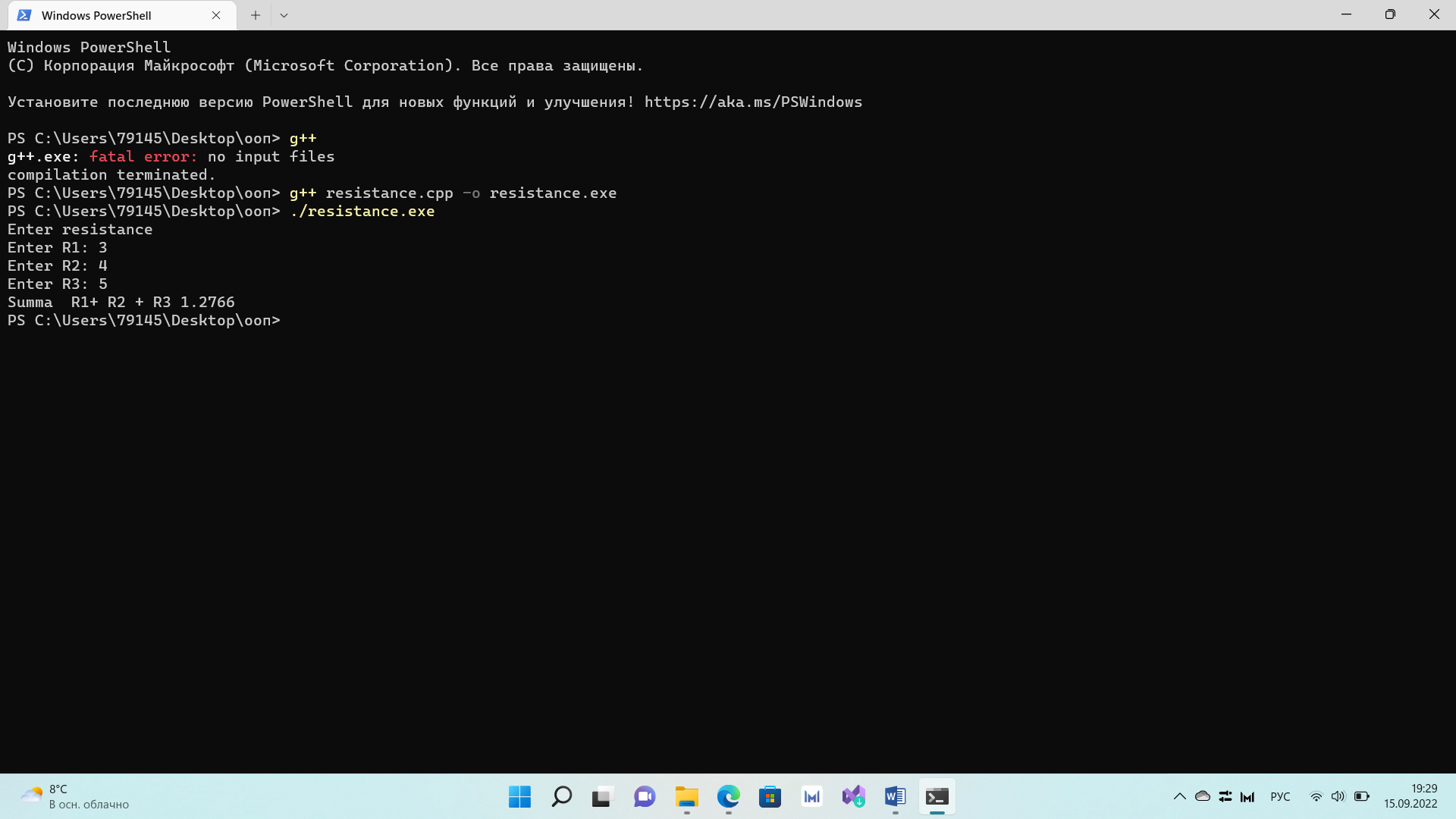
В консоли пишем *: g++ <название>.cpp –o <название>.exe*



В папке должен появиться новый файл с расширением .exe



Теперь можно запустить созданную программу через консоль, для этого в консоли набираем команду *./<название>.exe* . В результате программа будет работать так:



# Отчет№2.Устройство памяти программы. Где хранятся: глобальные переменные, локальные переменные, аргументы функций, динамические переменные?

**Виды памяти**

**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.(сегмент данных)

**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.

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

Приложение при необходимости может запросить у ОС дополнительную память через аллокатор или напрямую через системный вызов.

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.

Языки сверхвысокого уровня используют динамическую память как основную: создают все или почти все объекты в динамической памяти, а на стеке или в статической памяти держат указатели на эти объекты.

Максимальный размер динамической памяти зависит от многих факторов: среди них ОС, процессор, аппаратная архитектура в целом, не говоря уже о самом очевидном — максимальном размере ОЗУ у конкретного устройства.

Автоматические переменные вроде

Void foo(){

int a;

double b;}

хранятся на стеке. Как следует из названия, стек работает с переменными по схеме FILO (first in last out). Управление стеком происходит автоматически. При выходе переменной из области видимости, соответствующая ей в стеке память освобождается. Этот механизм позволяет разработчику не следить за удалением автоматических переменных. Стек работает очень быстро, но имеет ограниченный размер, который обычно не превосходит нескольких мегабайт.

Второй тип памяти - куча - устроен иначе. В куче объекты можно хранить в произвольном месте, создавать и удалять их в произвольном порядке, а размер кучи обычно значительно превосходит размер стека. Платить за эти преимущества приходится скоростью: работа с кучей происходит значительно медленнее, чем со стеком. Кроме того, объекты из кучи не удаляются автоматически.

1. Глобальные переменные – хранятся в фиксированной области памяти, устанавливаемой компилятором. Глобальные переменные чрезвычайно полезны, когда одни и те же данные используются в нескольких функциях программы. Следует избегать ненужного использования глобальных переменных по трем причинам:

a. Они используют память в течение всего времени работы программы, а не тогда, когда они необходимы.

б. Использование глобальных переменных вместо локальных приводит к тому, что функции становятся более частными, поскольку они зависят от переменных, определяемых снаружи.

в. Использование большого числа глобальных переменных может вызвать ошибки в программе из-за неизвестных и нежелательных эффектов.

2. Локальные переменные – адрес локальной переменной нигде не хранится, если у нет указателя, указывающего на локальную переменную. Сами локальные переменные обычно хранятся в стеке в большинстве современных систем. И компилятор на самом деле не помещает и не выталкивает отдельные переменные в / из стека, он просто резервирует достаточно места и использует смещения.

3. Аргументы функций – они будут храниться в текстовой или кодовой секции двоичного файла только один раз независимо от количества экземпляров созданного класса.

Функции не хранятся отдельно нигде для каждого экземпляра класса. Они обрабатываются так же, как и любая другая функция, не являющаяся членом. Единственное отличие заключается в том, что компилятор фактически добавляет дополнительный параметр функции, который является указателем типа класса.

4. Динамические переменные – в куче объекты можно хранить в произвольном месте, создавать и удалять их в произвольном порядке, а размер кучи обычно значительно превосходит размер стека. Платить за эти преимущества приходится скоростью: работа с кучей происходит значительно медленнее, чем со стеком. Кроме того, объекты из кучи не удаляются автоматически.

Источники:

[Переменные (cpp.com.ru)](https://cpp.com.ru/shildt_spr_po_c/02/0204.html)

[Где в памяти хранятся мои переменные на C? - Переполнение стека (turbopages.org)](https://translated.turbopages.org/proxy_u/en-ru.ru.58b1b283-632302d0-b565a87d-74722d776562/https/stackoverflow.com/questions/14588767/where-in-memory-are-my-variables-stored-in-c-)

# Отчет№3. Рекомендации по оформлению кода

Имеющиеся среды разработки могут улучшить читаемость кода с помощью отображения модификаторов доступа, подсветки кода, автоматического форматирования и прочего, но программисту не следует полагаться на эти инструменты. Исходный код должен рассматриваться не только в рамках используемой среды разработки и должен быть написан так, чтобы максимально улучшить читаемость независимо от среды. Существует множество рекомендации по оформлению кода, ниже приведены некоторые из них.

1. **Все имена следует записывать по-английски.**

fileName; // НЕ РЕКОМЕНДУЕТСЯ: imyaFayla

1. **Переменные, имеющие большую область видимости, следует называть длинными именами, имеющие небольшую область видимости — короткими.**

Имена временных переменных, использующихся для хранения временных значений или индексов, лучше всего делать короткими. Программист, читающий такие переменные, должен иметь возможность предположить, что их значения не используются за пределами нескольких строк кода. Обычно это переменные i, j, k, l, m, n (для целых), а также c и d (для символов).

1. **Следует избегать сокращений в именах.**

computeAverage(); // НЕЛЬЗЯ: compAvg();

1. **Заголовочным файлам C++ следует давать расширение .h (предпочтительно) либо .hpp. Файлы исходных кодов могут иметь расширения .c++ (рекомендуется), .C, .cc либо .cpp.**

MyClass.c++, MyClass.h

Это расширения, одобряемые стандартом C++.

1. **Директивы включения следует сортировать (по месту в иерархии системы, ниже уровень — выше позиция) и группировать. Оставляйте пустую строку между группами.**

#include <fstream>

#include <iomanip>

#include <qt/qbutton.h>

#include <qt/qtextfield.h>

1. **Следует инициализировать переменные в месте их объявления.**

Это даёт гарантию, что переменные пригодны для использования в любой момент времени. Но иногда нет возможности осуществить это:

int x, y, z;

getCenter(&x, &y, &z);

В этих случаях лучше оставить переменные неинициализированными, чем присваивать им какие-либо значения.

1. **Переменные никогда не должны иметь двойной смысл.**

Улучшайте читаемость, убеждаясь, что все представленные концепции не предполагают разночтений. Сокращайте возможность ошибки из-за побочных эффектов.

1. **Следует избегать использования глобальных переменных.**

Не существует причины использовать глобальные переменные в C++. То же касается глобальных функций и (статических) переменных, область видимости которых — весь файл.

1. **Основной отступ следует делать в два пробела.**

for (i = 0; i < nElements; i++)

a[i] = 0;

Отступ в один пробел достаточно мал, чтобы отражать логическую структуру кода. Отступ более 4 пробелов делает глубоко вложенный код нечитаемым и увеличивает вероятность того, что строки придётся разбивать. Широко распространены варианты в 2, 3 или 4 пробела; причём 2 и 4 — более широко.

1. **Пробелы.**  
   — Операторы следует отбивать пробелами.  
   — После зарезервированных ключевых слов языка C++ следует ставить пробел.  
   — После запятых следует ставить пробелы.  
   — Двоеточия следует отбивать пробелами.  
   — После точек с запятой в цикле for следует ставить пробелы.
2. **Все комментарии следует писать на английском.**

В интернациональной среде английский — предпочтительный язык.

1. **Используйте // для всех комментариев, включая многострочные.**

// Комментарий, расположенный

// на нескольких строках.

1. **Именование**

Основные правила стиля кодирования приходятся на именование. Вид имени сразу же (без поиска объявления) говорит нам что это: тип, переменная, функция, константа и т.д.

**Общие принципы именования**

* Используйте имена, который будут понятны даже людям из другой команды.
* Имя должно говорить о цели или применимости объекта.
* Не экономьте на длине имени, лучше более длинное и более понятное (даже новичкам) имя.
* Поменьше аббревиатур, особенно если они незнакомы вне проекта.
* Не сокращайте слова.

**Имена файлов**

Имена файлов должны быть записаны только строчными буквами, для разделения можно использовать подчёркивание (\_) или дефис (-).   
  
Примеры подходящих имён:

* ***my\_useful\_class.cc***
* ***my-useful-class.cc***
* ***myusefulclass.cc***
* ***myusefulclass\_test.cc // \_unittest and \_regtest are deprecated.***

Когда файлы используются парами, лучше давать им одинаковые имена. Например, ***foo\_bar.h***и***foo\_bar.cc*** 

**Имена типов**

Имена типов начинаются с прописной буквы, каждое новое слово также начинается с прописной буквы. Подчёркивания не используются: *MyExcitingClass, MyExcitingEnum*.  
  
Имена всех типов — классов, структур, псевдонимов, перечислений, параметров шаблонов — именуются в одинаковом стиле. Имена типов начинаются с прописной буквы, каждое новое слово также начинается с прописной буквы. Подчёркивания не используются. Например:

// classes and structs

**class** **UrlTable** { ...

**class** **UrlTableTester** { ...

**struct** **UrlTableProperties** { ...

// typedefs

**typedef** hash\_map<UrlTableProperties \*, std::string> PropertiesMap;

**Имена переменных**

Имена переменных (включая параметры функций) и членов данных пишутся строчными буквами с подчёркиванием между словами. Члены данных классов (не структур) дополняются подчёркиванием в конце имени.

Например:*a\_local\_variable, a\_struct\_data\_member, a\_class\_data\_member\_*.

**Имена обычных переменных**  
Например:

std::string table\_name; // OK - строчные буквы с подчёркиванием

std::string tableName; // Плохо - смешанный стиль

**Имена констант**Объекты объявляются как constexpr или const, чтобы значение не менялось в процессе выполнения. Имена констант начинаются с символа «k», далее идёт имя в смешанном стиле (прописные и строчные буквы). Подчёркивание может быть использовано в редких случаях когда прописные буквы не могут использоваться для разделения.

Например:

**const** **int** kDaysInAWeek = 7;

**const** **int** kAndroid8\_0\_0 = 24; // Android 8.0.0

**Имена функций**

Обычные функции именуются в смешанном стиле (прописные и строчные буквы). Имя функции начинается с прописной буквы и каждое слово в имени пишется с прописной буквы.

**void** **AddTableEntry**();

**void** **DeleteUrl**();

**void** **OpenFileOrDie**();

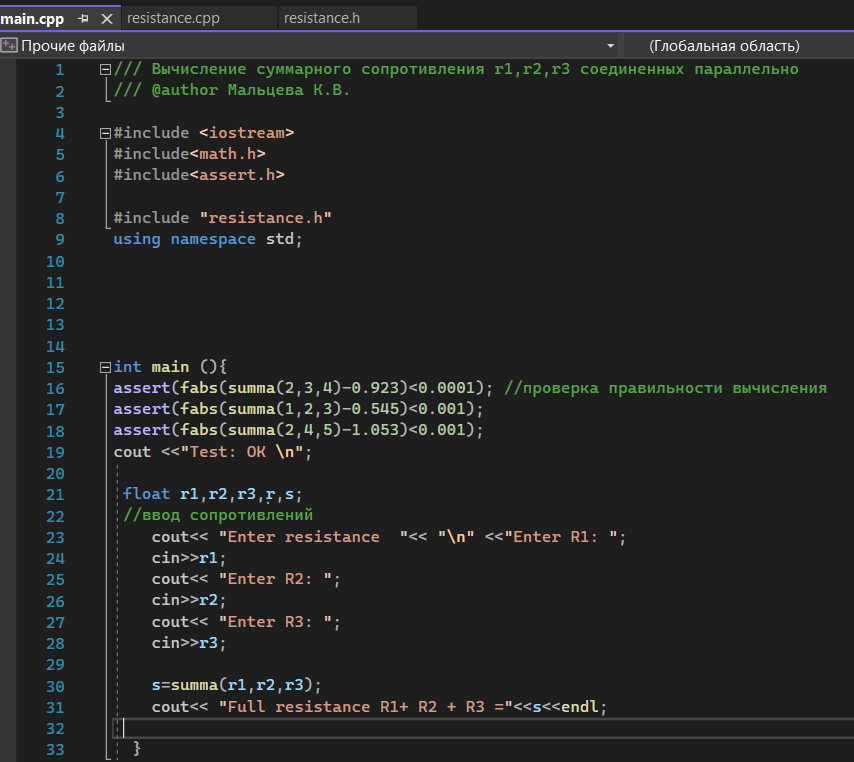
***Чтобы ознакомиться с остальными рекомендациями по оформлению кода перейдите по ссылке:***

[*90 рекомендаций по стилю написания программ на C++ / Хабр (habr.com)*](https://habr.com/ru/post/172091/)

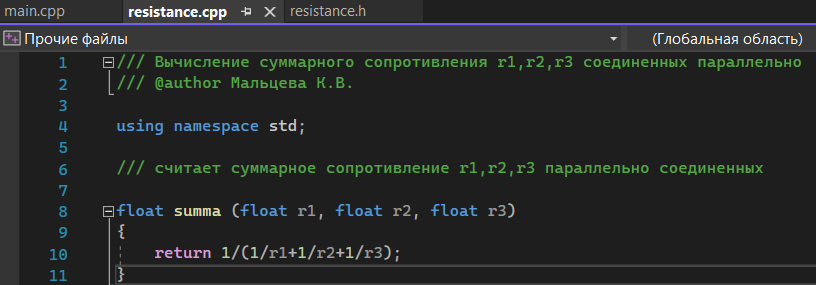
[*Руководство Google по стилю в C++. Часть 8 / Хабр (habr.com)*](https://habr.com/ru/post/477722/?ysclid=l89unwt66a847127724)

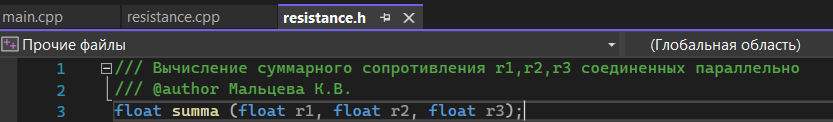
# Отчет№4. Как компилировать в командной строке несколько файлов?

Для компиляции нескольких файлов в командной строке разбиваем ранее написанную программу на несколько модулей: в файл с расширением «.cpp» переносим функции, аналог implementation в Паскале; в файл с расширением «.h» добавляются заголовки, аналог interface в Паскале.



Чтобы подключить модули к главной программе, необходимо написать *команду #include “название.h”*

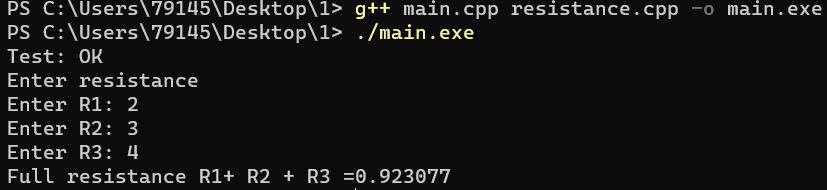




После пробуем запустить программу через консоль, для этого пишем:

*g++ <название главной программы.h> <название модуля.cpp> -o <название главной программы.exe>*

Результат работы данной программы будет выглядеть так:



# Отчет№5. 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.

Во многих случаях функции имеют аргументы, которые используются настолько редко, что достаточно значения по умолчанию. В таких случаях возможность задания аргументов по умолчанию позволяет указывать только те аргументы функции, которые важны в конкретном вызове.

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

***returned\_type*** FunName(***type1*** *v1* = *val1*, ***type2*** *v2* = *val2*, ..., ***typeN*** *vN* = *valN*)

{

// ...

}

где

* *returned*\_*type* – тип возвращаемый функцией;
* *FunName* – имя функции;
* *type1*, *type2*, …, *typeN* – типы переменных *v1*, *v2*, …, *vN*;
* *val2*, …, *valN* – значения, которое присваиваются по умолчанию переменным *v2*, …, *vN*.

В этом случае типы и присваиваемые значения должны быть совместимыми. Если в объявлении функции в списке параметров встречается запись вида:

*type v = val ,* то аргумент *v* есть аргументом по умолчанию (здесь *type*-некоторый тип, *v* — переменная типа *type, val* — значение что присваивается по умолчанию переменной *v*).

**Пример.** Задается функция AreaTriangle(),вычисляющая площадь треугольника, заданного длинами сторон *a*, *b*, *c*. Функция получает параметры, которые имеют единичные значения по умолчанию.

Листинг объявления и использования функции следующий:

#include "stdafx.h"

#include <iostream>

using namespace std;

// Площадь треугольника

// функция получает параметры по умолчанию

double AreaTriangle(double a = 1, double b = 1, double c = 1)

{

double p, s;

if ((a+b)<c || (b+c)<a || (a+c)<b)

return 0;

p = (a + b + c)/2.0; // полупериметр

s = ::sqrt(p \* (p-a) \* (p-b) \* (p-c)); // формула Герона

return s;

}

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

{

double area;

area = AreaTriangle(); // area = 0.433013

area = AreaTriangle(1.5); // area = 0.496078

area = AreaTriangle(1.5, 1.5); // area = 0.707107

area = AreaTriangle(1, 1, 1); // area = 0.433013

area = AreaTriangle(2, 2, 2); // area = 1.7325

area = AreaTriangle(2, 2, 8); // area = 0

cout << area << endl;

return 0;

}

Использование аргументов по умолчанию дает следующие преимущества:

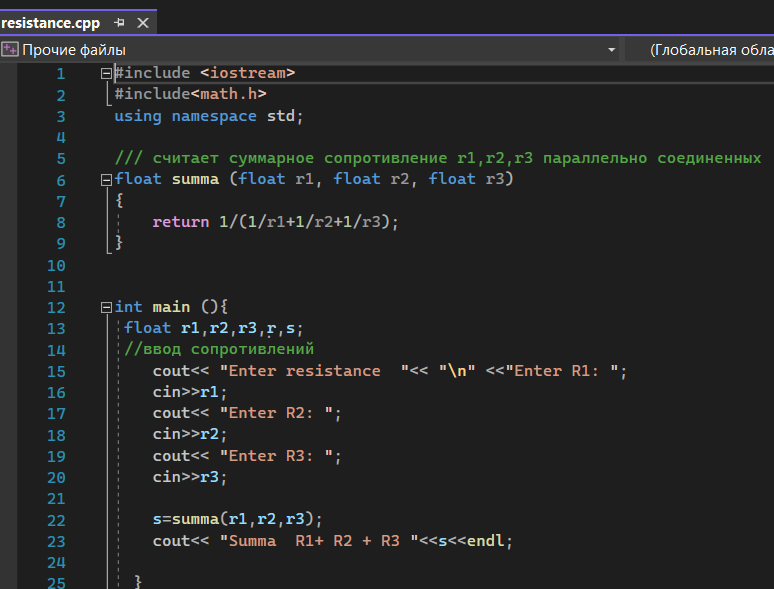
* сокращается листинг программного кода за счет избежания написания лишних функций, которые выполняют ту же работу только с другими значениями аргументов;
* обеспечивается простой, естественный и эффективный стиль программирования;
* в некоторых случаях аргументы по умолчанию есть сокращенной формой перегрузки функции. Это, в свою очередь улучшает читабельность программного кода и упрощает вызов функции.

Источники:

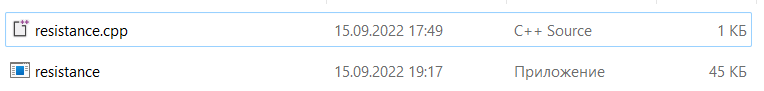
[C++. Функции. Аргументы по умолчанию в функциях | BestProg](https://www.bestprog.net/ru/2018/07/30/functions-arguments-by-default-in-functions_ru/#q01)

[Аргументы по умолчанию | Microsoft Docs](https://docs.microsoft.com/ru-ru/cpp/cpp/default-arguments?view=msvc-160)

# Отчет№6.Уменьшение размера исполняемого файла, создаваемого компилятором MinGW

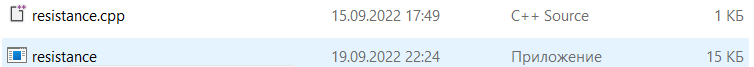
У нас есть простая программа, которая вычисляет суммарное сопротивление параллельно соединенных проводников

Программа скомпилирована командой *g++ <название>.cpp –o <название>.exe* в исполняемый файл размером 45 КБ компилятором MinGW g ++ под Win 11.



Попробуем уменьшить размер файла при помощи различных флагов g++ :

Снова скомпилируем программу, но командой *g++ <название>.cpp –o <название>.exe –s*



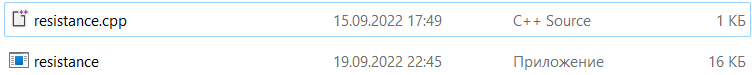
Компилятор помещает в исполняемый файл не только то, что нужно для запуска (то есть код и данные), но ещё и дополнительную (и в общем-то ненужную) информацию (к примеру, имена всех переменных и функций, включая те, что были объявлены со спецификатором static).

Эта дополнительная информация нужна только на стадии компоновки, но компилятор оставляет её и в конечном исполняемом файле, возможно для удобства дальнейшей отладки. За её вырезание и отвечает флаг «-s». В данном случае вес исполняемого файла уменьшился с 45 до 15КБ.

Попробуем перекомпилировать программу с помощью флага «strip»



Вес программы уменьшился с 45 до 16КБ



Данная команда также удаляет дополнительную информацию и никак не влияет на работу программы.

Существуют и другие различные команды для компилирования и оптимизации программ с помощью MinGW:

[Флаги и параметры компилятора CPP/C++ (caiorss.github.io)](https://caiorss.github.io/C-Cpp-Notes/compiler-flags-options.html)

# Отчет№7. Работа с аргументами командной строки в программах на языке C++

*Аргументы командной строки – это необязательные строковые аргументы, которые операционная система передает программе при ее запуске. Затем программа может использовать их в качестве входных данных (или игнорировать).*

**Передача аргументов командной строки**

Исполняемые программы можно запускать из командной строки, вызывая их по имени. Например, чтобы запустить исполняемый файл "WordCount", который находится в корневом каталоге диска C: на компьютере с Windows, вы можете ввести:

C:\>WordCount

Чтобы передать аргументы командной строки в WordCount, мы просто перечисляем их после имени исполняемого файла:

C:\>WordCount Myfile.txt

Теперь при выполнении WordCount Myfile.txt будет предоставлен в качестве аргумента командной строки. Программа может иметь несколько аргументов командной строки, разделенных пробелами:

C:\>WordCount Myfile.txt Myotherfile.txt

Это также работает и в других операционных системах командной строки, таких как Linux.

**Использование аргументов командной строки**

Теперь, когда вы знаете, как предоставить программе аргументы командной строки, следующим шагом будет доступ к ним из нашей программы на C++. Для этого мы используем другую форму main(). Эта новая форма main() принимает два аргумента (по соглашению названные argc и argv) следующим образом:

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

Иногда вы также можете увидеть и такое:

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

Несмотря на то, что они обрабатываются одинаково, мы предпочитаем первое представление, потому что его интуитивно легче понять.

argc – это целочисленный параметр, содержащий счетчик числа аргументов, переданных программе (argc = argument count, число аргументов). argc всегда будет не меньше 1 потому, что первым аргументом всегда является имя самой программы.

argv – это место, где хранятся фактические значения аргументов (argv = argument values, значения аргументов, хотя на самом деле расшифровывается как «argument vectors»). Хотя объявление argv выглядит устрашающе, на самом деле argv представляет собой просто массив строк в стиле C.

Давайте напишем короткую программу с именем MyArgs для вывода значений всех параметров командной строки:

// Программа: MyArgs

#include <iostream>

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

{

std::cout << "There are " << argc << " arguments:\n";

// Перебираем все аргументы и

// выводим номер и значение каждого из них

for (int count{ 0 }; count < argc; ++count)

{

std::cout << count << ' ' << argv[count] << '\n';

}

return 0;

}

Теперь, когда мы вызываем эту программу (MyArgs) с аргументами командной строки "Myfile.txt" и "100", вывод будет следующим:

There are 3 arguments:

0 C:\MyArgs

1 Myfile.txt

2 100

Аргумент 0 – это путь и имя текущей запущенной программы. Аргументы 1 и 2 в данном случае – это два параметра командной строки, которые мы передали.

**Работа с числовыми аргументами**

Аргументы командной строки всегда передаются в виде строк, даже если предоставленное значение является числовым. Чтобы использовать аргумент командной строки в качестве числа, вы должны преобразовать его из строки в число. В C++ это можно сделать следующим образом:

#include <iostream>

#include <sstream> // для std::stringstream

#include <string>

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

{

if (argc <= 1)

{

// В некоторых операционных системах argv[0] может оказаться пустой

// строкой вместо имени программы.

// Мы добавим зависимость для нашего ответа от того,

// является ли argv[0] пустым или нет.

if (argv[0])

std::cout << "Usage: " << argv[0] << " <number>" << '\n';

else

std::cout << "Usage: <program name> <number>" << '\n';

return 1;

}

// устанавливаем переменную строкового потока с именем convert,

// инициализированную входными данными из argv[1]

std::stringstream convert{ argv[1] };

int myint{};

if (!(convert >> myint)) // выполняем преобразование

myint = 0; // если преобразование не удалось,

//установим для myint значение по умолчанию

std::cout << "Got integer: " << myint << '\n';

return 0;

}

При запуске с вводом "567" эта программа печатает:

Got integer: 567

**Обработка специальных символов**

Как правило, в операционных системах действуют особые правила обработки специальных символов, таких как двойные кавычки и обратный слеш.

Например:

MyArgs Hello world!

печатает:

There are 3 arguments:

0 C:\MyArgs

1 Hello

2 world!

Обычно строки, переданные в двойных кавычках, считаются частью одной строки:

MyArgs "Hello world!"

печатает:

There are 2 arguments:

0 C:\MyArgs

1 Hello world!

Если вы хотите включить двойную кавычку в состав строки, вам нужно поставить перед двойной кавычкой обратный слеш:

MyArgs \"Hello world!\"

печатает:

There are 3 arguments:

0 C:\MyArgs

1 "Hello

2 world!"

**Заключение**

Аргументы командной строки предоставляют пользователям и другим программам отличный способ передать входные данные в программу при запуске. Рассмотрите возможность использования любых входных данных, необходимых для работы программы при запуске, в качестве параметров командной строки. Если они не будут переданы через командную строку, вы всегда можете обнаружить это и попросить пользователя ввести данные. Таким образом, ваша программа сможет работать в любом случае.

Источник: [11.11 – Аргументы командной строки (radioprog.ru)](https://radioprog.ru/post/1201?ysclid=l7wvrge81791105343)

# Отчет№8. Обработка исключительных ситуации в языке C++

Исключительная ситуация – это событие, которое привело к сбою в работе программы. В результате возникновения исключительной ситуации программа не может корректно продолжить свое выполнение.

Примеры действий в программе, которые могут привести к возникновению исключительных ситуаций:

1. деление на ноль;
2. нехватка оперативной памяти при использовании оператора new для ее выделения (или другой функции);
3. доступ к элементу массива за его пределами (ошибочный индекс);
4. переполнение значения для некоторого типа;
5. взятие корня из отрицательного числа и т.п;

Язык программирования C++ дает возможность перехватывать исключительные ситуации и соответствующим образом их обрабатывать.

Механизм перехвата исключений C++ позволяет генерировать исключения в том месте, в котором оно возникает – это очень удобно. Не нужно «выдумывать» собственные способы обработки исключений, которые возникают в функциях нижних уровней, для того чтобы передать их в функции высших уровней.

Для перехвата и обработки исключительных ситуаций в языке C++ введена конструкция try…catch, которая имеет следующую общую форму:

try {

// тело блока try

// ...

// генерирование исключения оператором throw

}

catch(*type1 argument1*)

{

// тело блока catch

}

catch(*type2 argument2*)

{

// тело блока catch

}

...

catch(*typeN* argumentN)

{

// тело блока catch

}

где

*type1*, *type2*, …, *typeN* – соответственно тип аргументов *argument1*, *argument2*, …, *argumentN*.

Код, который нужно проконтролировать, должен выполняться в середине блока try. Исключительные ситуации перехватываются оператором catch, который следует непосредственно за блоком try в котором они возникли.

В блоке try могут быть размещены операторы и функции. Если в блоке try генерируется соответствующая исключительная ситуация, то она перехватывается соответствующим блоком catch. Выбор того или иного блока catch осуществляется в зависимости от типа исключительной ситуации. После возникновения исключительной ситуации определенного типа, вызывается блок catch с таким самым типом аргумента. Аргумент принимает некоторое значение, которое соответствующим образом обрабатывается (выводится на экран сообщение об ошибке и т.п.).

Если в блоке try возникнет исключительная ситуация, которая не предусмотрена блоком catch, то вызывается стандартная функция terminate(), которая по умолчанию вызовет функцию abort(). Эта стандартная функция останавливает выполнение программы.

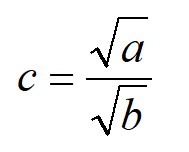
Чтобы в блоке try сгенерировать исключительную ситуацию, нужно использовать оператор throw. Оператор throw может быть вызван внутри блока try или внутри функции, которая вызывается из блока try.

Общая форма оператора throw следующая

throw исключение;

В результате выполнения оператора throw генерируется исключение некоторого типа. Это исключение должно быть обработано в блоке catch.

**Пример.** Демонстрируется использование блока try…catch для обработки выражения:

[](https://www.bestprog.net/wp-content/uploads/2019/09/05_02_02_16_formula01.jpg)

В данном выражении в трех случаях может возникнуть исключительная ситуация:

1. корень из отрицательного числа **a**, если **a**<0;
2. корень из отрицательного числа **b**, если **b**<0;
3. деление на 0, если **b**=0.

Поэтому, в блоке try…catch нужно обработать эти три случая.

#include <iostream>

#include<cmath>

using namespace std;

main()

{

// обработка выражения sqrt(a)/sqrt(b)

double a, b;

cout << "a = ";

cin >> a;

cout << "b = ";

cin >> b;

double c;

try { // начало блока try

if (b == 0)

throw 1;

if (b < 0)

throw 2;

if (a < 0)

throw 2;

c = sqrt(a) / sqrt(b);

cout << "c = " << c << endl;

}

catch (int e) // перехват ошибки

{

if (e == 1)

cout << "Division by 0." << endl;

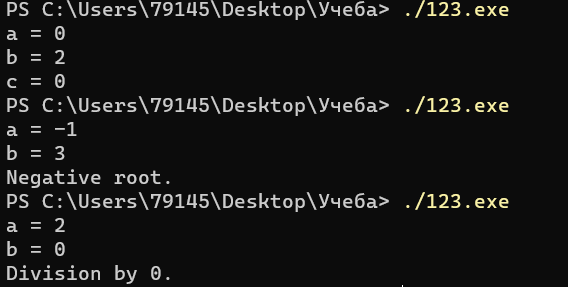
if (e == 2)

cout << "Negative root." << endl;

}

}

Попробуем ввести разные значения в откомпилированной программе, результат будет следующий:



Как мы видим после применения блока try..catch работа программы не завершается и в результате ее выполнения мы можем узнать произошла ли исключительная ситуация или программа успешно обработала данные.

Однако, оператор throw совместно с операторами try...catch стоит использовать только если исключительная ситуация (и соответственно вызов throw) и код её обработки должны находится на разных уровнях вложенности вызова функций. Т.е. исключительная ситуация может возникнуть внутри функции (или внутри функции, которая вызвана в другой функции и т.д.), а обработка по логике алгоритма возможна только вне этой функции. В остальных случаях для обработки исключительных ситуаций стоит использовать условный оператор.(поэтому в примере выше является только показательным, а на практике там следует пользоваться условным оператором)

Так же можно "поймать" и все исключения:  
catch(...) { }  
Троеточие в этом случае показывает, что будут пойманы все исключения. При таком подходе нельзя указать имя переменной. В случае, если "кидаются" данные нестандартного типа (экземпляры определенных вами классов, структур и т.д.), лучше "ловить" их по ссылке, иначе вся "кидаемая" переменная будет скопирована в стек вместо того, чтобы просто передать указатель на нее. Если кидаются данные нескольких типов и вы хотите поймать конкретную переменную (вернее, переменную конкретного типа), то можно использовать несколько блоков catch, ловящих "свой" тип данных:  
  
try {  
throw 1;  
// throw 'a';  
}  
catch (long b) {  
cout « "пойман тип long: " « b « endl;  
}  
catch (char b) {  
cout « "пойман тип char: " « b « endl;  
}"

Пример:

[#include](https://vk.com/im?sel=98115250&st=%23include) <iostream>  
double divide(int, int);  
int main()  
{  
int x = 500;  
int y = 0;  
try  
{  
double z = divide(x, y);  
std::cout « z « std::endl;  
}  
catch (...)  
{  
std::cout « "Error!" « std::endl;  
}  
std::cout « "The End..." « std::endl;  
return 0;  
}  
  
double divide(int a, int b)  
{  
if (b == 0)  
throw "Division by zero!";  
return a / b;  
}

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

int main()  
{  
int x = 500;  
int y = 0;  
try  
{  
double z = divide(x, y);  
std::cout « z « std::endl;  
}  
catch (const char\* msg)  
{  
std::cout « msg « std::endl;  
}  
std::cout « "The End..." « std::endl;  
return 0;  
}  
С помощью параметра const char\* msg получаем сообщение, которое передано оператору throw, и выводит это сообщение на консоль. И в этом случае консольный вывод будет выглядеть следующим образом:  
Division by zero!  
The End...  
Таким образом, мы можем узнать суть возникшего исключения.

Источники:

[C++. Понятие исключительной ситуации. Блок try…catch. Оператор throw. Примеры использования | BestProg](https://www.bestprog.net/ru/2019/09/14/c-the-concept-of-an-exceptional-situation-instruction-try-catch-operator-throw-examples-of-using-ru/)

[C++ try catch: обработка исключений (exception). Операторы throw, try и catch, примеры (highload.today)](https://highload.today/throw-try-i-catch-v-c/)

# Отчет№9. Понимание lvalue и rvalue в C и С++

Термины *lvalue* и *rvalue* не являются чем-то таким, с чем часто приходится сталкиваться при программировании на C/C++, а при встрече не сразу становится ясным, что именно они означают. Наиболее вероятное место столкнуться с ними — это сообщения компилятора.

Пример нагляден при компиляции следующего кода при помощи g++:

int& foo()

{

return 2;

}

Вы увидите следующую ошибку:

testcpp.cpp: In **function** 'int& foo()':

testcpp.cpp:5:12: error: invalid initialization of non-const reference

of type 'int&' from an rvalue of type 'int'

В сообщение об ошибке упоминается  rvalue. Что же в C и C++ понимается под lvalue и rvalue?

**Простое определение**

lvalue (locator value) представляет собой объект, который занимает идентифицируемое место в памяти (например, имеет адрес).

rvalue определено путём исключения, говоря, что любое выражение является либо lvalue, либо rvalue. Таким образом из определения lvalue следует, что rvalue — это выражение, которое не представляет собой объект, который занимает идентифицируемое место в памяти.  
  
**lvalue**

lvalue назван так исторически, так как он ​​может находиться с левой стороны в операциях присвоения.  lvalue (левосторонние данные) – данные, которым можно присвоить какое-либо значение и адрес расположения этих данных в памяти можно получить используя оператор «&».

К левосторонним данным относятся.

1. Переменные:

  int b = 5; int temp; temp = b;

1. Ссылка (в момент явного приведения с созданием временной ссылки ((тип&)lvalue)):

  int a = 5; (double&)a = 23.23;

1. Раскрытый указатель / адрес:

 int \*p = new int(23);    \*p = 5;

int a = 5; \*(double\*)&a = 23.23; // Извлекаем адрес, по которому расположена переменная-а, приводим этот адрес к типу double\*, раскрываем этот адрес, после чего по этому же адресу неявно создается временная переменная типа double, которой присваивается константна.

4.     Нераскрытый указатель, если ему присваивается адрес:

int main()

{

   int a = 23;  int \*p = new int(); int \*q = new int();

   p = q;                        // Присваиваем адрес, присвоенный указателю q.

   p = &a;                       // Присваиваем адрес переменной.

   p = new int(5);               // Присваиваем адрес выделенной памяти.

   return 0;

}

5.     Возвращаемое значение функции по ссылке, указателю; в этом случае, на месте инициатора вызова функции, будет подставлено левостороннее значение.

int& f1(int& a)          { return a;  }                    // Возвращает ссылочную переменную.

int\* f2(int& a)          { return &a; }                    // Возвращает адрес, на который ссылается ссылка.

int\*& f3(int\*& a)        { return a;  }                    // Возвращает ссылочную переменную-указатель.

int main()

{

   int a = 5, b = 10, temp;

   int \*p = new int(23);

   cout << "a == " << a << " " << ", b == " << b << ", \*p == " << \*p << endl;

   temp  = a;

   f1(a) = b;                     // f1(a) - левостороннее значение.

   \*(f2(b)) = temp;               // f2(b) - правостороннее значение. \*(f2(b)) - левостороннее значение.

   p = &temp;                     // Имя указателя - левостороннее значение.

   f3(p) = &temp;                 // f3(p) - левостороннее значение.

   cout << "a == " << a << " " << ", b == " << b << ", \*p == " << \*p << endl;

   return 0;

}

**rvalue**

1.     r-value назван так исторически, так как он ​​может находиться с правой стороны в операциях присвоения.

2.     r-value (правосторонние данные) – это данные, которые можно присвоить левосторонним данным и адрес расположения этих данных в памяти нельзя получить используя оператор «&».

3.     Результатом всех выражений всегда есть rvalue.

К правосторонним данным относятся.

1.     Результат применение оператора &.

int a = 23;

&a;              // Результат взятия адреса - правостороннее значение.

//&a = 5;        // Недопустимо: попытка выполнить присвоение правосторонним данным.

2.     Приведение значения дает r-value.

int main()

{

 int i = 5;

 //(double)i = 23.23; // Ошибка. (double)i - приведение значения дает r-value.

 return 0;

}

Исключение (VS2012): приведение значения к своему же типу дает l-value.

int main()

{

   int x = 23;

   const int &s = x;

   (int)s = 32;

   //(unsigned)s = 32; // Ошибка. Результатом приведения к другому типу является r-value.

   cout << x << endl;

   return 0;

}

3.     Результат вызова функции, тип возвращаемого значения которой не является ссылка.

int r()                                   // Возвращает rvalue

{ return 0; }

int f() { int i = 23; return i; }

int main()

{

    int a = f();     // Ок. f() возвращает правостороннее значение, может быть присвоено другим данным.

    f();             // Эквивалентно: 23; - чистое правостороннее значение.

    //f() = 5;       // Недопустимо: попытка выполнить присвоение чистым правосторонним данным.

    return 0;

}

Источники:

[Enot++: Категории значений С++ (enotcpp.blogspot.com)](https://enotcpp.blogspot.com/2012/06/l-valuer-value.html)

[Понимание lvalue и rvalue в C и С++ / Хабр (habr.com)](https://habr.com/ru/post/348198/?ysclid=l8mqupksws838818269)

# Отчет№10. Ключевое слово typedef

Typedef является одним из ключевых слов, которые он позволяет разработчикам использовать для создания новых дополнительных имен для типов данных по умолчанию, таких как int, float, long, short и т. Д. Он создает только типы данных, но не создает никаких дополнительных типов значений. Когда мы используем ключевое слово typedef, оно возвращает значения, которые пользователь также должен знать, если мы хотим изменить что-либо в коде, легко перетащить определения и дать ясность о старых кодах, а также о новых кодах изменений, но мы выделим коды, используя базовые параметры для типов 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:

typedef double distance\_t; // определяем distance\_t как псевдоним для типа double

Есть возможность определить псевдоним, используя using:

using distance\_t = double;

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.

Cинтаксис typedef становится уродливым при использовании более сложных типов, особенно указателей на функции

typedef int (\*fcn\_t)(double, char); // fcn\_t - идентификатор typedef

Псевдонимы типов (using) функционально эквивалентны определениям typedef, но имеют преимущество в более удобном синтаксисе определения.

typedef int (\*fcn\_t)(double, char); // fcn\_t трудно найти

using fcn\_t = int(\*)(double, char); // fcn\_t легче найти

**Синтаксис:**

В C++ каждый объект, переменные и ключевые слова имеют свой синтаксис и атрибуты для объявления в программных кодах. Исходя из требования, мы будем использовать специальные ключевые слова или зарезервированные ключевые слова из библиотеки программирования.

#include <iostream>

datatype main()

{

typedef old datatype name new datatype name;

typedef struct structure name;

{

datatype initialization;

}

---coding logics---

}

Приведенные выше коды являются базовым синтаксисом для использования ключевых слов typedef в логике программирования. Мы можем использовать typedef в качестве структуры, используя тип ключевого слова struct; используя этот тип, мы можем создать объявление n типов данных, а также изменить имя в одном процессе.

Как работает typedef в C++?

* Как правило, typedef является одним из зарезервированных ключевых слов; он обеспечивает тот же уровень абстракций от фактического типа данных и изменяет типы данных, которые используются программистами, чтобы позволить ему уделять больше внимания концепциям кодирования. Это также облегчает написание программных кодов и их очистку с помощью некоторых методов destroy() или любых других методов по умолчанию, потому что сборщики мусора являются основной областью для уничтожения нежелательных кодов и очистки ее для областей пространства памяти. В зависимости от типов данных должны быть рассчитаны размеры, и он выделяет пространство памяти для переменных типа данных большого хранилища и переменных хранилища малого объема.
* Обычно объявления typedef используются с двумя различными типами, такими как typedef type-declaration и typedef type-definition identifier; эти два стандартных объявления typedef покрыты новым псевдонимом типов имен с некоторыми синтаксическими объявлениями для идентификаторов языка, а другой тип покрыт некоторыми стандартными библиотеками, а в других спецификациях POSIX включены в определения typedef, и он чаще встречается как в префиксе, так и в суффиксе, и он включает общие размеры с большим количеством вычислений времени, включенных в коды. Более того, ключевое слово typedef часто используется с некоторыми документами при указании номера конкретной переменной через включенные в программирование контексты, которые могут включать в выражения с переменной количество единиц измерения с подсчетами.
* Ключевое слово также использовалось для упрощения объявлений переменных для некоторых составных типов, таких как struct, union и т. Д., Или, хотя мы включили некоторые типы указателей в синтаксис, в то время как мы использовали указатель в ключевом слове typedef, оно создает конкретное хранилище памяти в компиляторе, а также адрес переменной также изменился как в старом, так и в новом типах данных. Мы также можем использовать указатели структуры в ключевом слове typedef, которое содержит несколько переменных с одним и тем же типом, и объявлять с помощью одноуровневых операторов, даже если указатели включены или нет в тип структуры. Функциональные указатели, также указанные с помощью typedef предыдущего количества строк кодирования, будут переписаны с помощью ключевого слова typedef, что уменьшит длину и сложность программ. Функциональные объявления будут более загадочными, и это ясно показывает, что с принятием аргументов или типов он будет возвращать переменные.

Пример 1:

#include <iostream>

int main(){

typedef unsigned int a;

a p = 3, q = 5;

std::cout << "p = " << p << std::endl;

std::cout << "q = " << q << std::endl;

return 1;

}



Пример 2:

#include<iostream>

using namespace std;

struct example

{

char \*domain;

int id;

};

typedef int \*pr;

int main()

{

struct example eg;

eg.domain = "Welcome To My Domain";

eg.id = 7879;

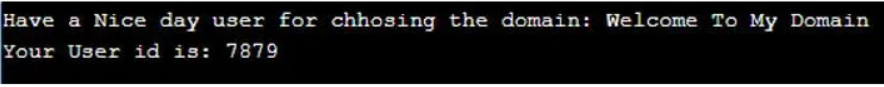
cout << "Have a Nice day user for chhosing the domain: " << eg.domain << '\n';

cout << "Your User id is: " << eg.id;

pr pvar;

return 0;

}



В приведенных выше примерах мы использовали typedef в различных категориях, используя понятия указателя и переменные, которые обращались к старым и новым.

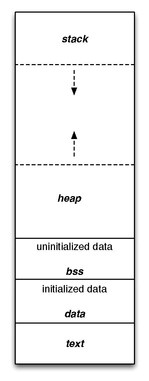
Заключение

В typedef находится конкретная зарезервированная область ключевых слов C++. Он имеет много предопределенных ключевых слов, используемых для реализации логики программирования в приложении. Здесь мы использовали это ключевое слово, чтобы в основном уменьшить строки кодов и области памяти; мы можем использовать и редактировать коды очень четко, поэтому, лежащие в основе кодов, являются обязательными для этого ключевого слова.

[| C++ typedef Как typedef работает в C++ с примерами (educba.com)](https://www.educba.com/c-plus-plus-typedef/)

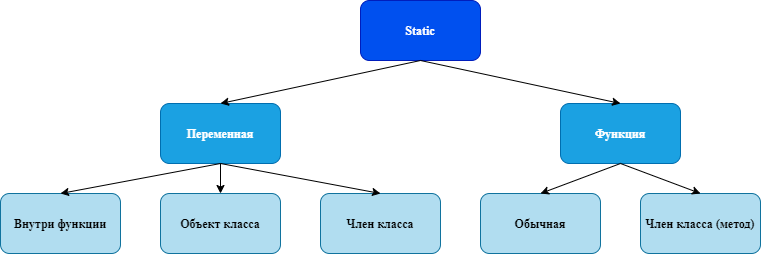
# Отчет№11.Static

**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.



**Где используется?**

Ниже приведена схема, как и где используется **static** в программе.



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. Давайте рассмотрим две почти одинаковые программы и их поведение. Отличие в них только в том, что одна использует статическую переменную, а вторая нет.  
  
**Первая программа:**

#**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

**Вторая программа:**

#**include** <iostream>

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

**int** count = 0; // строка 4

std::cout << count++;

}

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

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

counter();

}

**return** 0;

}

**Вывод программы:**

000000000

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()*.

**Статические функции**

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*). Ниже приведен листинг трех файлов одной программы.

// extend\_math.cpp

**int** **sum**(**int** a, **int** b) {

**int** some\_coefficient = 1;

**return** a + b + some\_coefficient;

}

// math.cpp

**int** **sum**(**int** a, **int** b) {

**return** a + b;

}

// main.cpp

**int** **sum**(**int**, **int**); // declaration

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

**int** result = sum(1, 2);

**return** 0;

}

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

// extend\_math.cpp

**static** **int** **sum**(**int** a, **int** b) {

**int** some\_coefficient = 1;

**return** a + b + some\_coefficient;

}

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.

**Итоги:**

Статические переменные медленнее, чем нестатические переменные. Для того, чтобы обратиться к статической переменной, нам нужно сделать несколько дополнительных действий, таких как переход в другой сегмент памяти и проверка инициализации переменной. Чаще всего, быстрее выделить локальную переменную на стеке, чем делать дополнительные действия по использованию статической переменной.

* Если вы используете многопоточность, то здесь вы должны быть крайне осторожными, т.к. возможна ситуация, когда два и более потока захотят писать в одну статическую переменную. Если вы будете использовать нестатические переменные в функциях, то избежите подобного, т.к. для каждого потока будет создана собственная нестатическая переменная.
* Ключевое слово static является неотъемлемой частью порождающего шаблона проектирования Singleton, который гарантирует, что будет создан только один экземпляр этого класса. В реализации этого паттерна используется и статический объект, и статическая функция-член. На практике вы можете использовать Singleton для создания объекта трейсера, логгера или любого другого объекта, который должен быть один на всё ваше приложение.
* Иногда для того, чтобы функция отработала только один раз без хранения предыдущего состояния где-то в объекте, используют статические переменные. Пример вы можете посмотреть в разделе «Статические переменные внутри функции». Но это не очень хороший подход, и может привести к долгим часам поиска ошибки, если вы используете многопоточность.
* На практике, программисты C++ часто используют статические функции-члены как альтернативу обычным функциям, которые не требуют создания объекта для выполнения ее.

Источник:

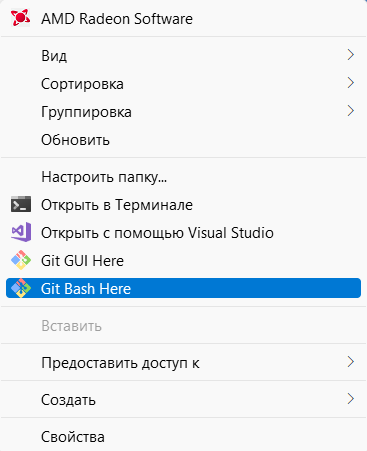
[Пока смерть не разлучит нас или всё о static в C++ / Хабр (habr.com)](https://habr.com/ru/post/527044/?ysclid=l8vj3nq0vr616083574)

# Отчет№12. Работа с системой управления версиями – git

**Git**  - это программное обеспечение для отслеживания изменений в любом наборе файлов, обычно используемое для координации работы программистов, совместно разрабатывающих исходный код во время разработки программного обеспечения.

До начала работы c git необходимо скачать приложение на ваш компьютер с сайта *git-scm.com*

Чтобы запустить эту оболочку нужно в проводнике Windows в контекстном меню выбрать "git Bash here". Запустить эту программу лучше прямо из той папки, где у вас хранится исходный код программы.

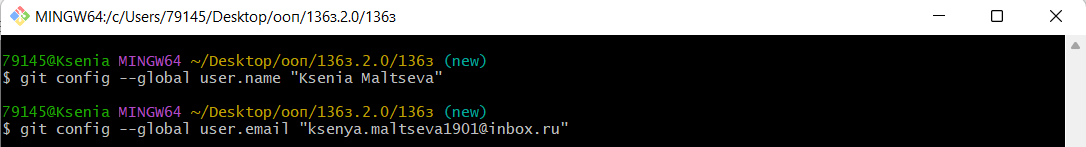


Далее нужно сделать несколько подготовительных действий:

* Настроить свое имя и эл.почту в git, данная информация необходима для создания коммитов (о них узнаем позднее)

Для этого используются команды:

Git config—global user.name “ Ваше имя”

Git config—global user.email “ Ваша почта” 

Эти две команды добавят строчки в особый файл в вашей домашней директории. Этот файл будет использован для всех ваших проектов. По умолчанию этот файл ~/.gitconfig, и его содержимое будет выглядеть следующим образом:

[user]

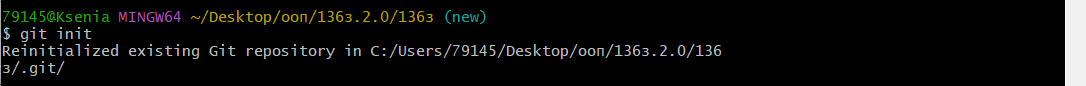
name = *Имя*

email = *Почта*

Если вы желаете переопределить эти значения для какого-либо определенного проекта (например чтобы использовать ваш рабочую не персональную эл.почту), то вы можете выполнить команду git config без параметра --global когда находитесь внутри этого проекта (Git также определяет возможность вручную указать файл конфигурации в параметрах запуска команды git). Это добавит секцию [user], такую же как и было показано выше, в файл .git/config в вашей корневой директории проекта.

* Дать понять git’у, что мы хотим считать эту попку репозитарием – создать репозитарий (хранилище данных)

Создать репозиторий *git init* .Git создаст скрытую папку, где будет хранить служебную информацию.



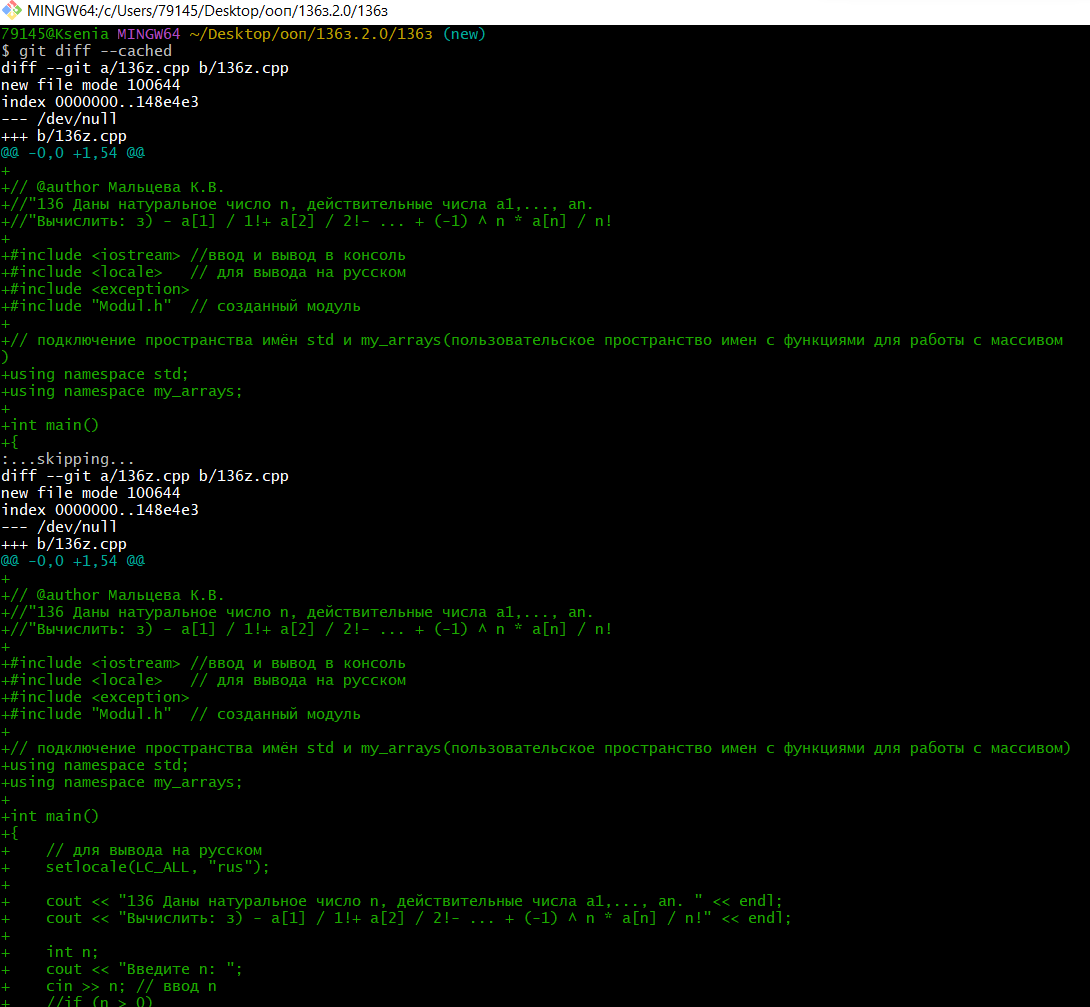
* Указать, за изменением каких файлов нужно следить – добавить файлы к отслеживанию



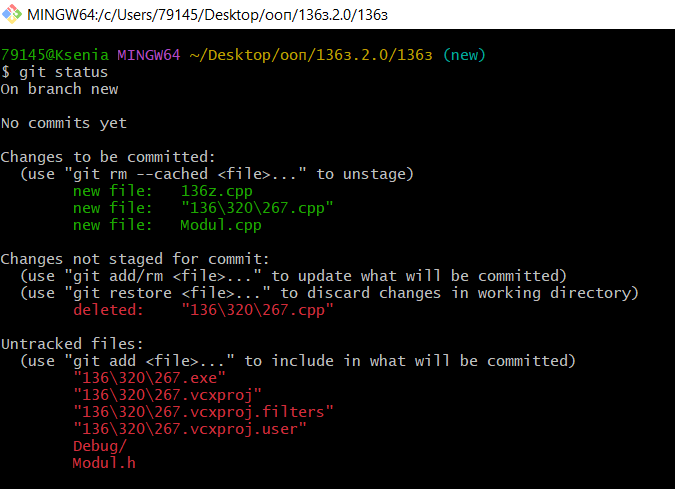
Добавить файлы к отслеживанию *git add <названия файлов>.* Как правило в отслеживании должны быть файлы исходных кодов и другие файлы, необходимые для компиляции и запуска программы. Исполняемые файлы не отслеживаются. Потому, что их всегда можно получить после компиляции и чтобы не засорять ими репозитарий.

* Теперь можно выполнить коммит. Измените свой код, добавьте некоторые строки или функции. Чтобы узнать что было добавлено в коммит используется команда git diff с параметром --cached:

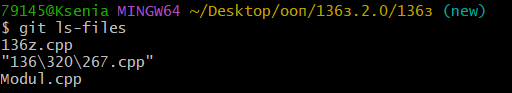
git diff --cached



(Без --cached, git diff покажет вам все изменения, которые вы сделали, но не добавили еще в индекс.) Вы также можете получить краткое описание сложившейся ситуации с помощью *git status*:



Просмотреть список отслеживаемых файлов git ls-files



**Начало работы**

Фиксация изменений Git не запоминает изменения в реальном времени. Это происходит потому, что каждое изменение должно быть логически завершённым. А это решает разработчик. Чтобы записать текущее состояние файлов используется команда *commit*. Такое действие называется фиксацией или коммитом (commit)

Зафиксировать изменения (сделать коммит) можно командой *git commit -am "кратко об изменениях"*

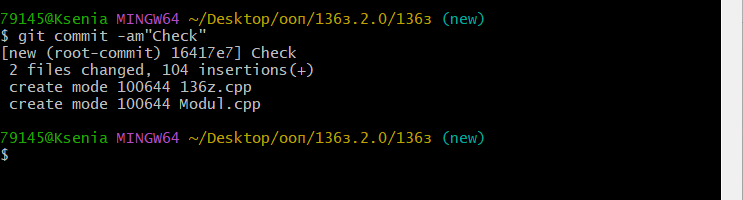
Ключи команды commit:

-a - добавить все отслеживаемые файлы в фиксацию

-m - комментарий к фиксации

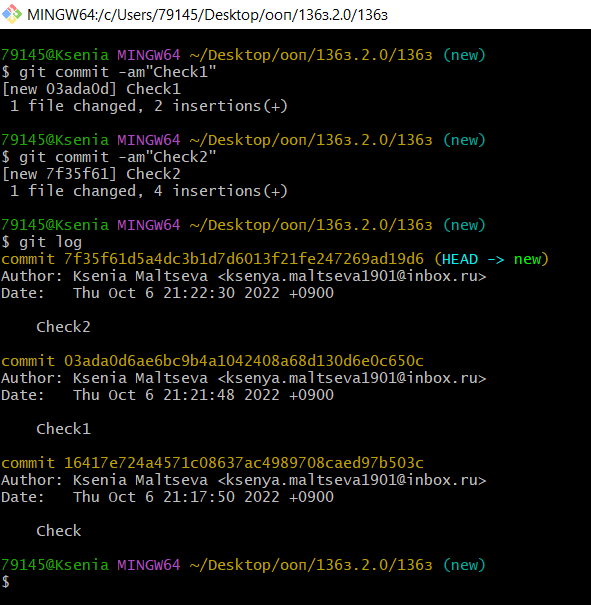
В комментариях следует кратко описывать сделанные изменения.

Добавим в нашу программу новую строку и сделаем коммит .



**Просмотр истории коммитов**

После того как вы создадите несколько коммитов, или же вы склонируете репозиторий с уже существующей историей коммитов, вы, вероятно, захотите оглянуться назад и узнать, что же происходило с этим репозиторием. Наиболее простой и в то же время мощный инструмент для этого — команда git log.



По умолчанию, без аргументов, git log выводит список коммитов созданных в данном репозитории в обратном хронологическом порядке. То есть самые последние коммиты показываются первыми.

**Ветви**

Один git репозиторий может заключать в себе множество ветвей разработки. В git отдельные версии программы хранятся в ветках (branches).

Одновременно с созданием репозитория создаётся основная ветка – master/main (в новых версиях можно ввести свое название)

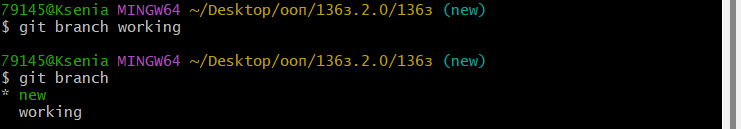
Чтобы создать новое ответвление под именем "working", выполните команду

$ git branch working

Теперь если вы выполните,

$ git branch

то получите список всех существующих ветвей:



Ветка "*working*" это та, которую вы только что создали, а ветка "*new*" это ветка по умолчанию которая создается автоматически. Звездочка указывает в какой ветке вы в данный момент находитесь; наберите

$ git checkout working

чтобы переключиться на ветку working . Теперь отредактируйте файл, выполните комит, и переключитесь обратно в главную ветку "new":

(edit file)

$ git commit -a

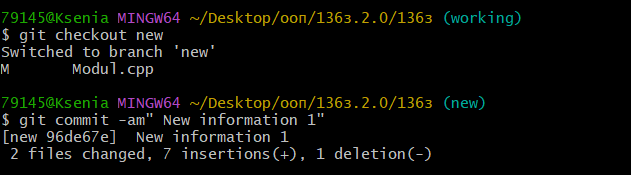
$ git checkout new

Убедитесь, что сделанные изменения невидимы, поскольку они были сделаны в ветке working, а вы сейчас в главной ветке "new".

Вы можете сделать другое изменение в ветке "new", затем выполнить коммит:

(edit file)

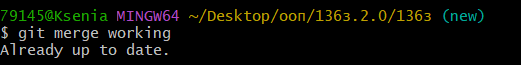
$ git commit –am



на этом этапе две ветки разошлись, поскольку в каждой из них различные изменения. Чтобы включить изменения в ветке working в new, выполните

$ git merge working

Если изменения не конфликтуют, то вы закончили. Если же существуют какие-либо конфликты, то в проблемных файлах останутся заметки которые можно увидеть.



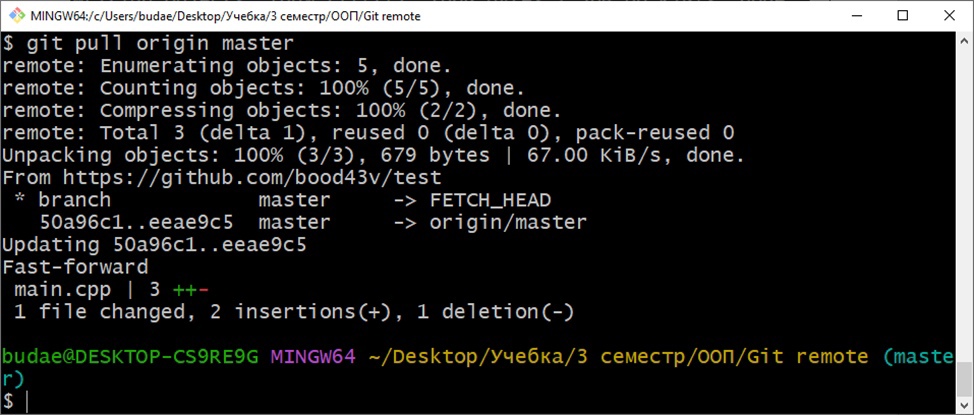
$ git pull

Отличие этой команды состоит в том, что она используется для извлечения и загрузки содержимого из удаленного репозитория и немедленного обновления локального репозитория этим содержимым. Слияние удаленных вышестоящих изменений в локальный репозиторий — это обычная задача рабочего процесса, возникающая при совместной работе на основе системы Git

Распространенные варианты использования и опции для git pull

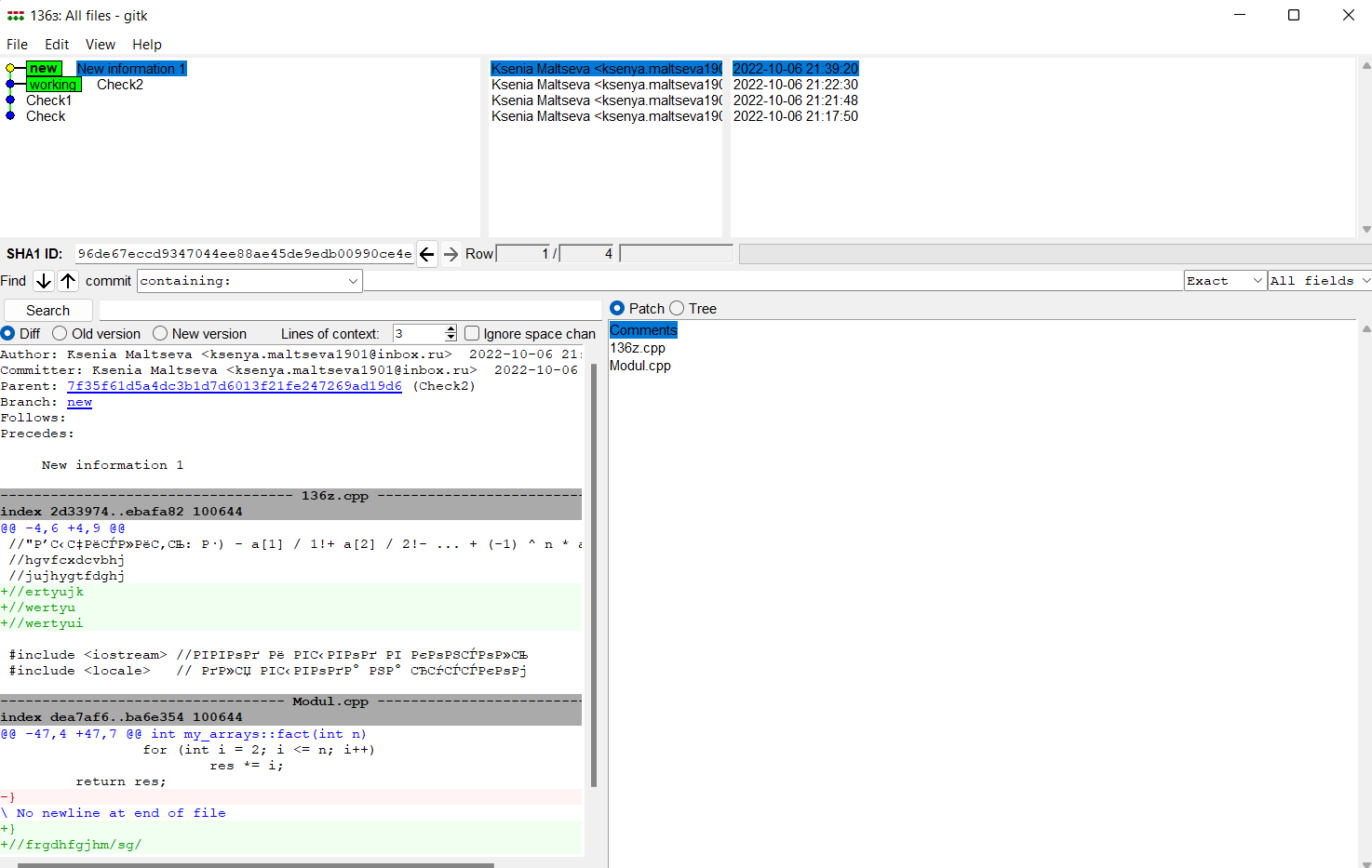
* git pull: Обновите локальную рабочую ветвь с помощью коммитов с удаленного доступа и обновите все ветви удаленного отслеживания.
* git pull --rebase: Обновите локальную рабочую ветвь с помощью коммитов из удаленного доступа, но перепишите историю, чтобы любые локальные фиксации происходили после всех новых фиксаций, поступающих с удаленного доступа, избегая фиксации слияния.
* git pull --force: Этот параметр позволяет принудительно получить определенную ветвь удаленного отслеживания при использовании параметра, который в противном случае не был бы извлечен из-за конфликтов. Чтобы заставить Git перезаписать текущую ветвь в соответствии с ветвью удаленного отслеживания, прочитайте ниже об использовании .<refspec>git reset
* git pull --all: Получить все пульты ДУ - это удобно, если вы работаете на вилке или в другом случае использования с несколькими пультами.

Во время выполнения команды pull выдает подробный вывод о загружаемом содержимом и информацию о слиянии.



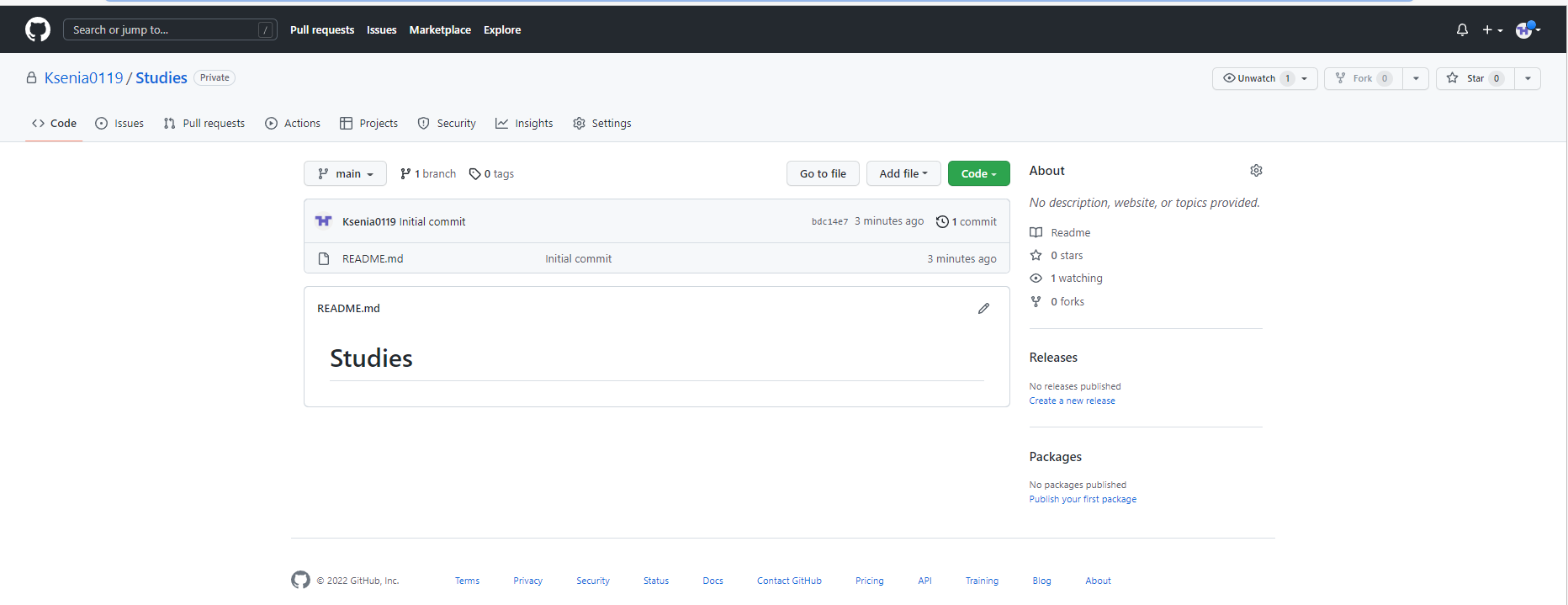
Следующая команда позволяет посмотреть на репозитории. Она покажет наглядное графическое представление истории.

$ gitk



**Удалённый репозиторий**

1. Создать удалённый репозиторий на сайте



1. Настройка локального репозитория.

git remote add origin <https://github.com/Usernsme/Reponame.git>



1. Отправка ветки master в удалённый репозиторий

git push [удал. сервер] [ветка]

git push -u origin master

origin - псевдоним для удалённого репозитория.

**Создание локальной копии удалённого репозитория.**

git clone git://github.com/Username/Reponame.git

Источники:

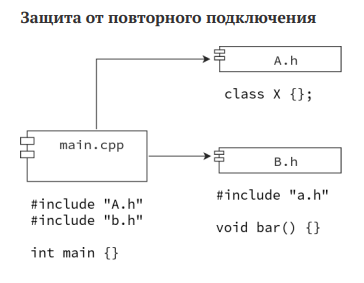
[Основы работы с Git (github.com)](https://gist.github.com/rdnvndr/cb21a06c5a71fd71213aed1619380b8e)

<https://raw.githubusercontent.com/VetrovSV/Programming/master/git_lec.pdf>

# Отчет№13. Защита от повторного подключения заголовочных файлов.

В случае если у вас есть заголовочный файл, который подключается несколькими другими файлами. Вы хотите убедиться, что препроцессор сканирует объявления в заголовочном файле не более одного раза.

Если вы не используете эту методику, которая называется *защитой заголовка*, то вы, вероятно, уже видели ошибки компиляции «symbol already defined» (символ уже определен), которые являются следствием отсутствия защитных мер против множественных определений. C++ не позволяет определять один и тот же символ несколько раз, и если вы это сделаете (целенаправленно или случайно), то получите ошибку компилятора. Включение защиты предотвращает такие ошибки, и она стала стандартной методикой.



В результате включения файлов по схеме класс X будет объявлен дважды

после обработки директив #include препроцессором:

// #include "a.h":

class X{};

// #include "b.h":

//// #include "a.h":

class X{

};

void bar(){};

// ...

int main(){

}

Для защиты от повторного подключения используются директивы препроцессора.

Вариант 1. Директива #pragma once.

#pragma once

class X{};

// ...

pragma – специальная директива для реализации не входящих в стандарт

языка С++ возможностей. Изначально #pragma once использовалась только в компиляторе MSVC, но потом её поддержка появилась и в других компиляторах.

Вариант 2 include guards.

#ifndef unit\_a\_h

#define

class X{};

// ...

#endif

В языках программирования Си и C++ #include guards (защита подключения), иногда также называемая macro guard (макрозащита) — это особая конструкция, применяемая для избегания проблем с «двойным подключением» при использовании директивы компилятора #include. Добавление #include guards в заголовочный файл является одним из способов сделать этот файл идемпотентным, то есть таким, что многократные его подключения эквивалентны однократному и не приводят к ошибкам.

Источники:

[GitHub - VetrovSV/OOP](https://github.com/VetrovSV/OOP) ([OOP\_StudBook.pdf](https://github.com/VetrovSV/OOP/blob/master/OOP_StudBook.pdf))

[Обеспечение единственности подключения заголовочного файла. C++. Сборник рецептов (wikireading.ru)](https://it.wikireading.ru/29953?ysclid=la3z7v306108636669)

# Отчет№14. Рефакторинг. Средства рефакторинга в C++

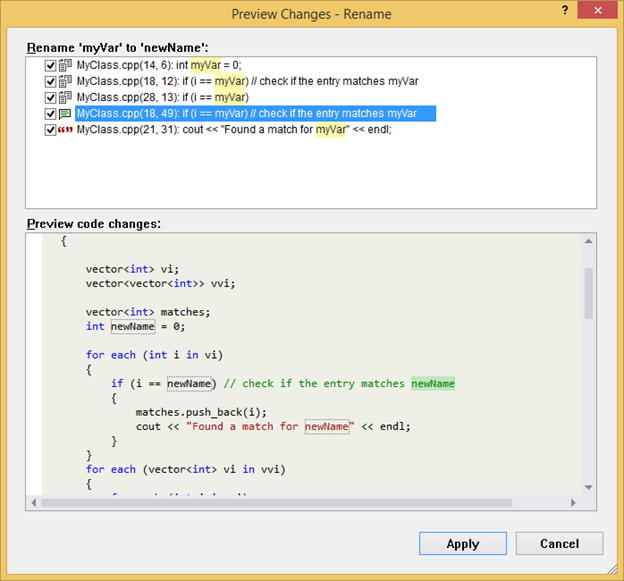
Рефакторинг относится к модификации существующего кода в улучшенную версию. Хотя рефакторинг часто выполняется при смене кода для добавления функций или исправления ошибок, термин, в частности, означает улучшение кода без необходимости добавления функций или исправления ошибок.

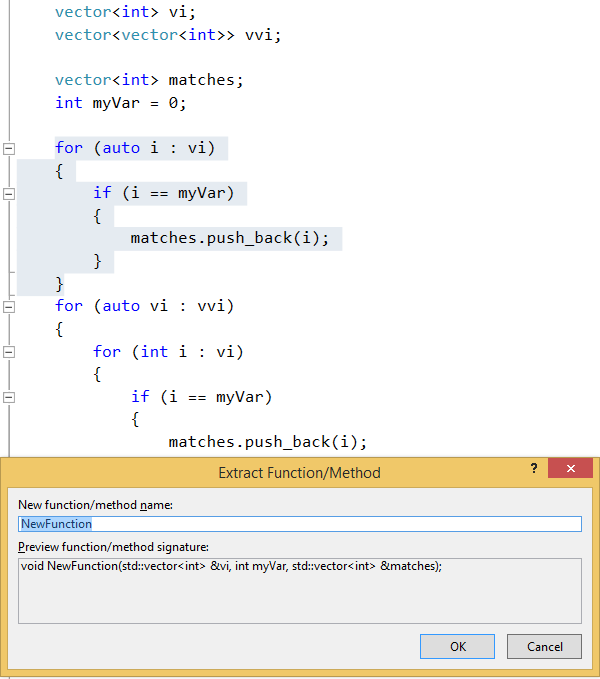
Рефакторинг доступен в контекстном меню быстрых действий. Или можно щелкнуть [лампочку](https://learn.microsoft.com/ru-ru/visualstudio/ide/perform-quick-actions-with-light-bulbs) в редакторе. Некоторые также находятся в меню **Правка > Рефакторинг**. Эти функции включают перечисленные ниже.

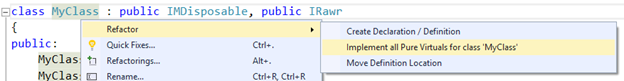
 Рассмотрим такие возможности Visual Studio 2015 Preview по работе над С++ кодом, как:

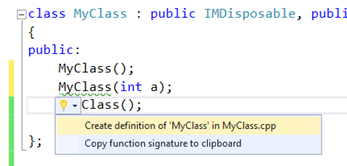
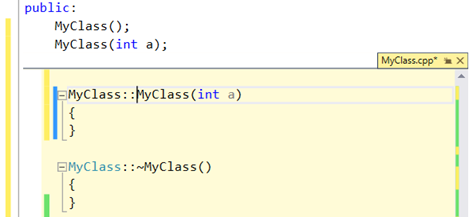
* Переименование (Rename)
* Извлечение функции (Extract Function)
* Генерация заглушек чисто виртуальных методов (Implement Pure Virtuals)
* Генерация объявлений/заглушек методов (Create Declaration/Definition)
* Перемещение объявлений функций (Move Function Definition)
* Преобразование в Raw-String (Convert to Raw-String Literal)

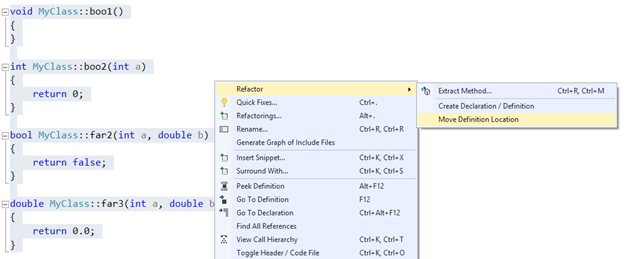
**Переименование,** безусловно, наиболее часто нужный инструмент. Его поместили в самом верху контекстного меню при правом клике на классе\функции\переменной. Кроме того, мы можете активировать его двойным нажатием комбинации Ctrl+R. Сам инструмент двухшаговый — в первом окне вы указываете настройки переименования, второе окно — превью.

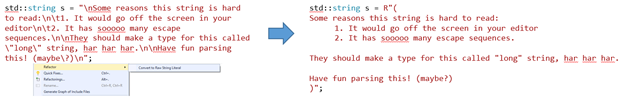


**Извлечение функции** . После его установки выделите блок кода, который хотите выделить в отдельную функцию, затем правый клик и в меню «Refactor…» выберите «Extract Function/Method».  
  


**Генерация заглушек виртуальных методов** позволяет создать тела всех чисто виртуальных методов в наследуемом классе. Поддерживается множественное наследование. Инструмент вызывается из контекстного меню объявления класса.  
  


**Генерация объявлений/заглушек методов** позволяет вам быстро сгенерировать недостающее объявление или заглушку тела метода.  
  
  
  


**Перемещение объявлений методов** позволяет быстро переместить тело метода из заголовочного файла в cpp-файл или наоборот.  
  


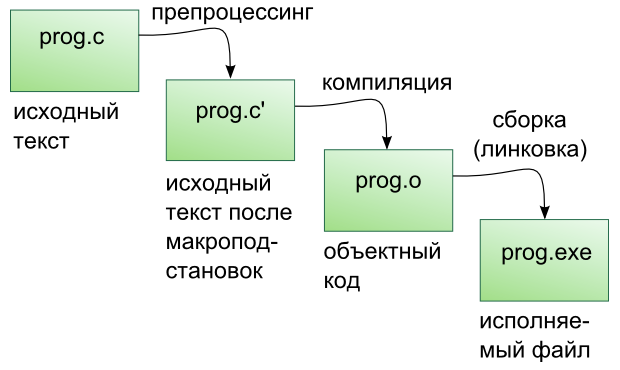
**Преобразование в Raw-String** позволяет сконвертировать любую строку в Raw-String, что значительно улучшает читабельность строк с escape-последовательностями. Функция вызывается из контекстного меню в любом месте строки.  
  


Источники:

[Редактирование и рефакторинг кода C++ в Visual Studio | Microsoft Learn](https://learn.microsoft.com/ru-ru/cpp/ide/writing-and-refactoring-code-cpp?view=msvc-160&viewFallbackFrom=vs-2017)

# Отчет№15. Этапы компиляции в С++

Компиляция исходных текстов на Си в исполняемый файл происходит в четыре этапа (преобразование в ассемблерный код).



* **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**.

* **Преобразование в Ассемблерный код.**
* **Преобразование в машинный код (компиляция).** В результате создаются объектные файлы из всех cpp файлов, переданных компилятору.

Процесс компиляции состоит из следующих этапов:

* 1. Лексический анализ. Последовательность символов исходного файла преобразуется в последовательность лексем.
  2. Синтаксический анализ. Последовательность лексем преобразуется в дерево разбора.
  3. Семантический анализ. Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д.
  4. Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла.
  5. Генерация кода. Из промежуточного представления порождается объектный код.

Результатом компиляции является объектный код.

Объектный код — это программа на языке машинных кодов с частичным сохранением символьной информации, необходимой в процессе сборки.

При отладочной сборке возможно сохранение большого количества символьной информации (идентификаторов переменных, функций, а также типов).

* **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.

Источники:

[GitHub - VetrovSV/OOP](https://github.com/VetrovSV/OOP) ([OOP\_StudBook.pdf](https://github.com/VetrovSV/OOP/blob/master/OOP_StudBook.pdf))

[Процесс компиляции программ на C++ / Хабр (habr.com)](https://habr.com/ru/post/478124/?ysclid=la3zyxskrm996901623)