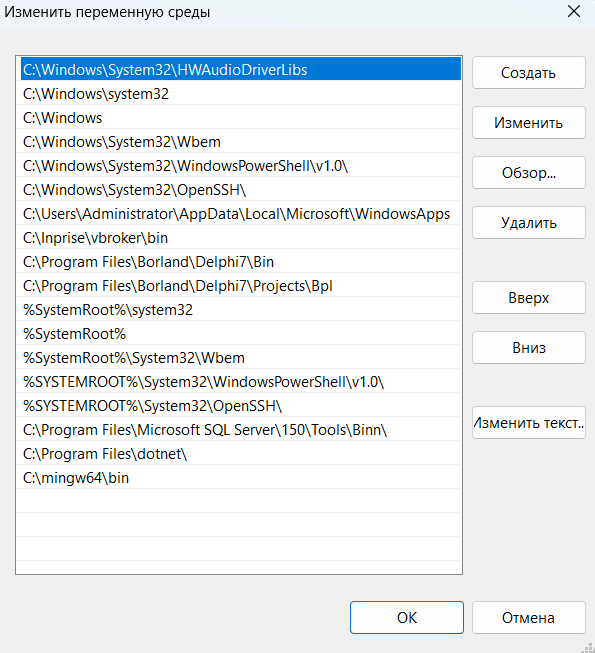
**Выполнил: Богушевич Данил ВМК-22**

**Тема: Работа с командной строкой в программах на языке C++**

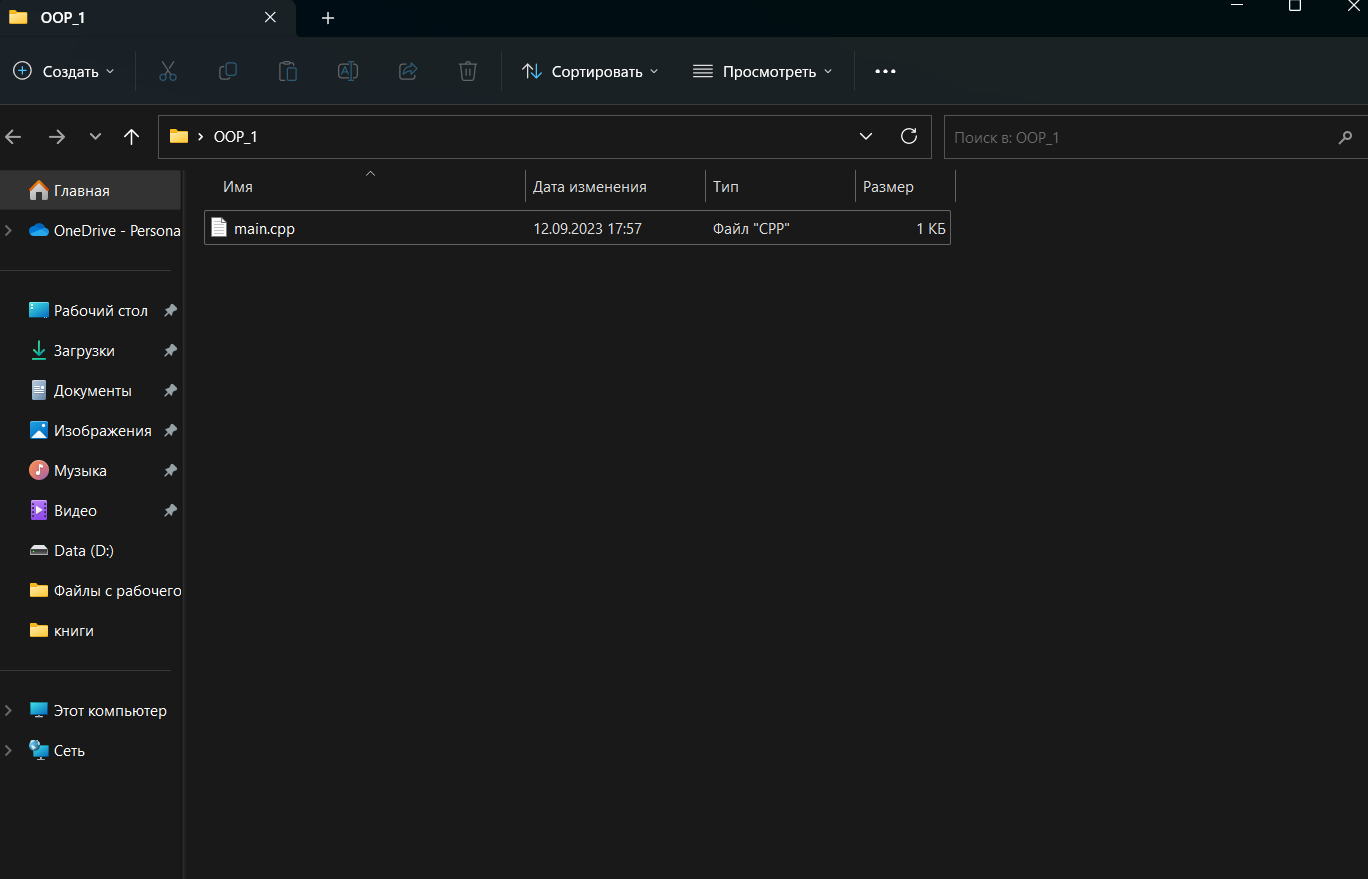
1.Для работы с командной строкой мы должны установить каталог с компиляторами

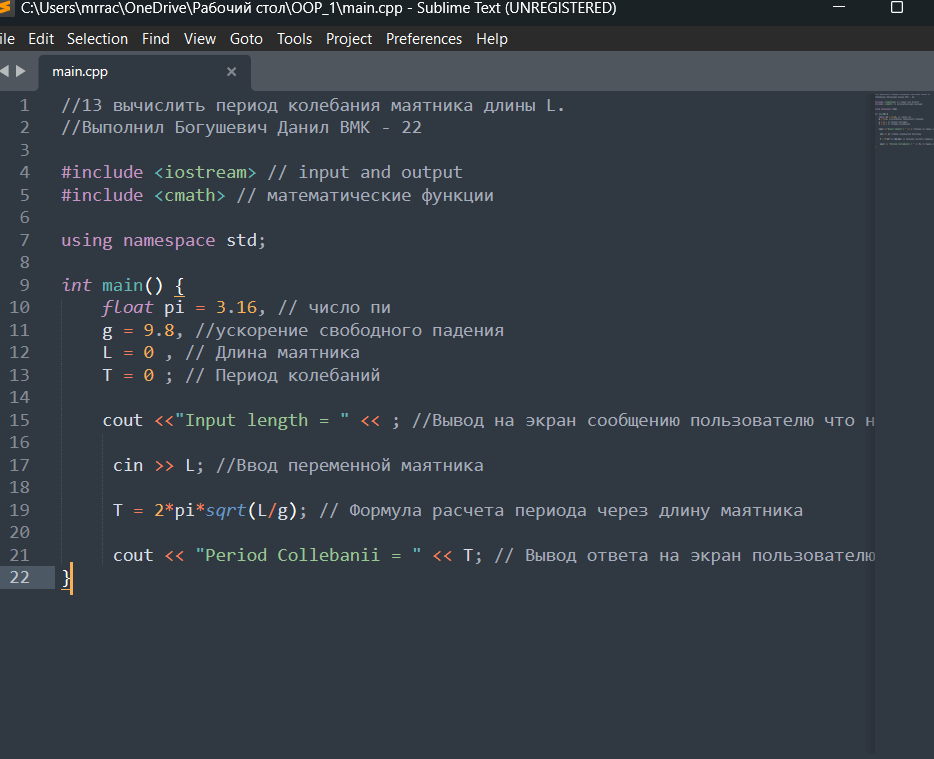
2.После установки мы создаем переменную в настройках для быстрого доступа к компилятору через консоль без написания полного пути. Заходим в изменение переменных среды системы, выбираем в системных переменных path и создаем путь к компилятору c



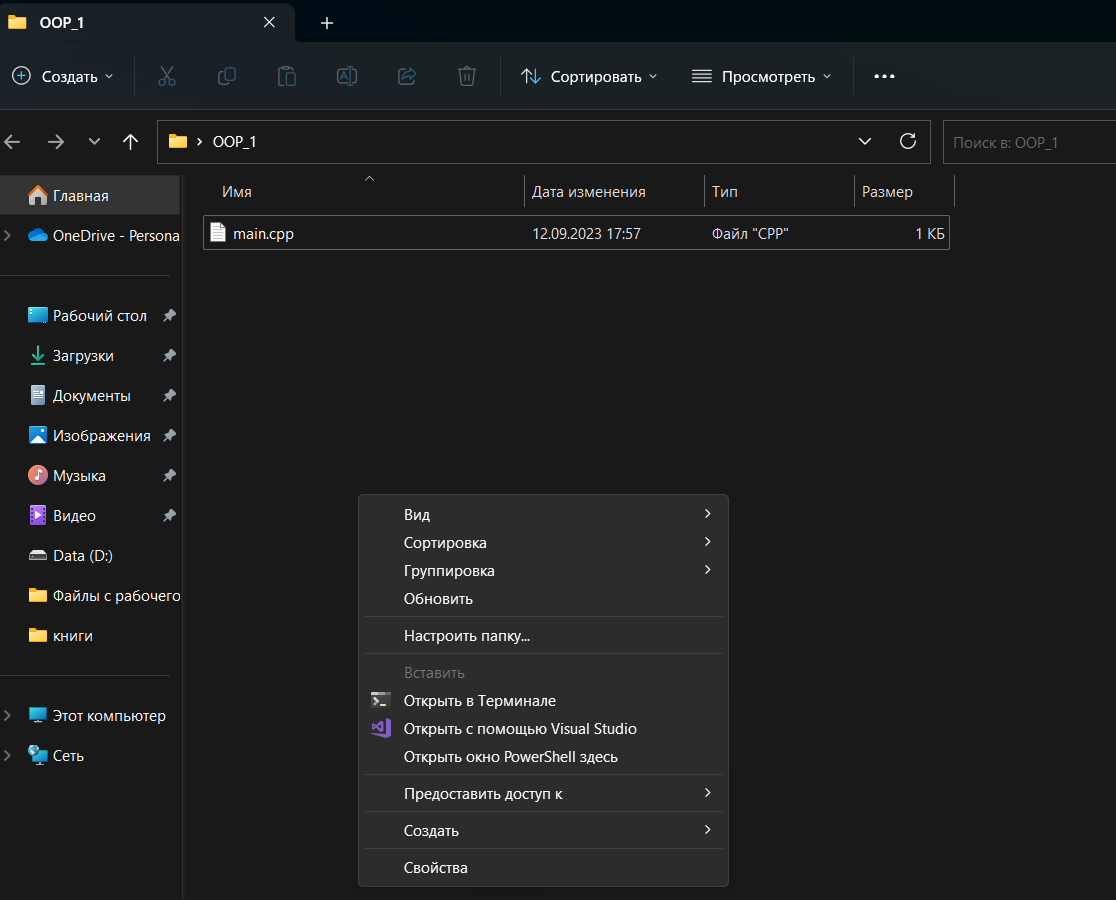
3.После этого создаем папку проекта и создаем файл в текстовом редакторе расширения .cpp

И в созданном файле пишем код программы.

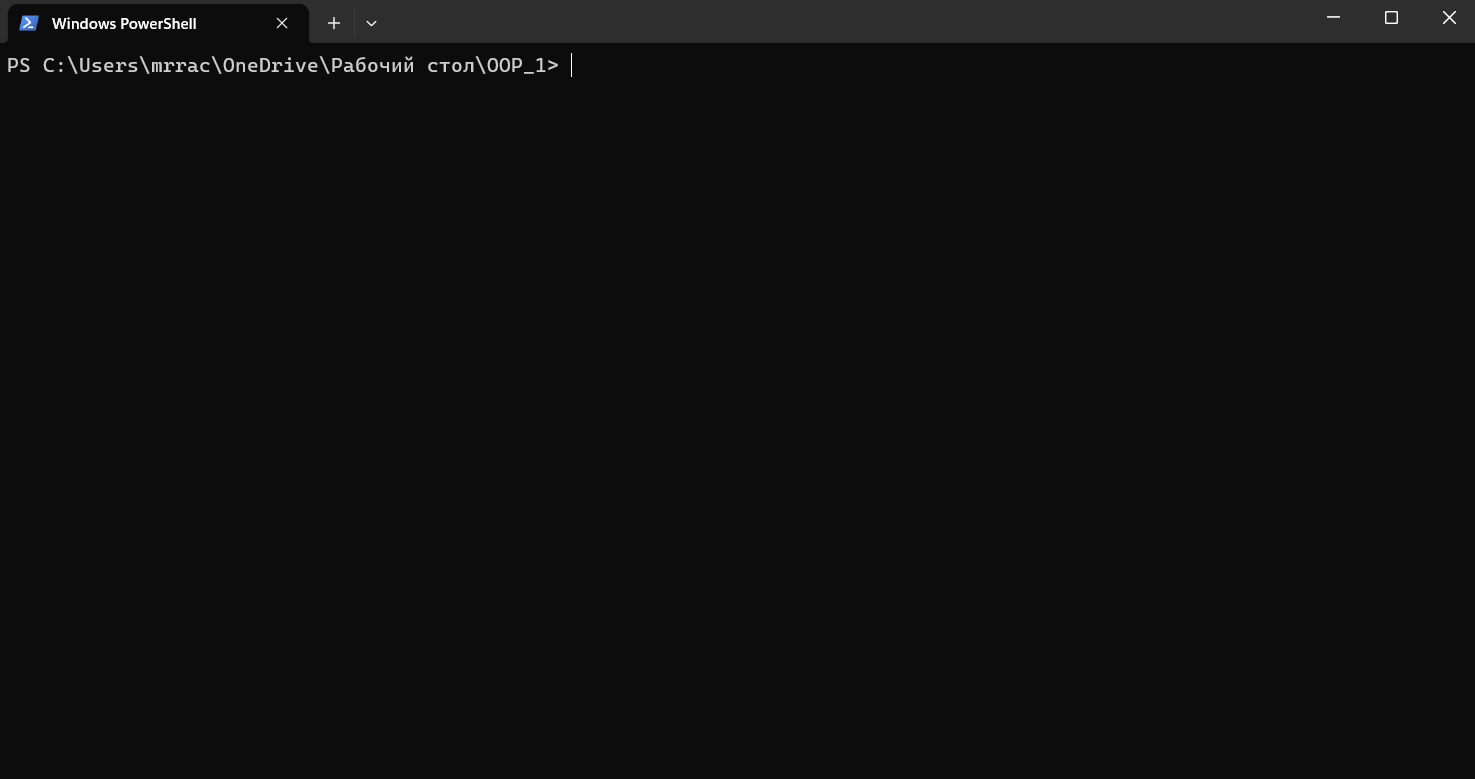




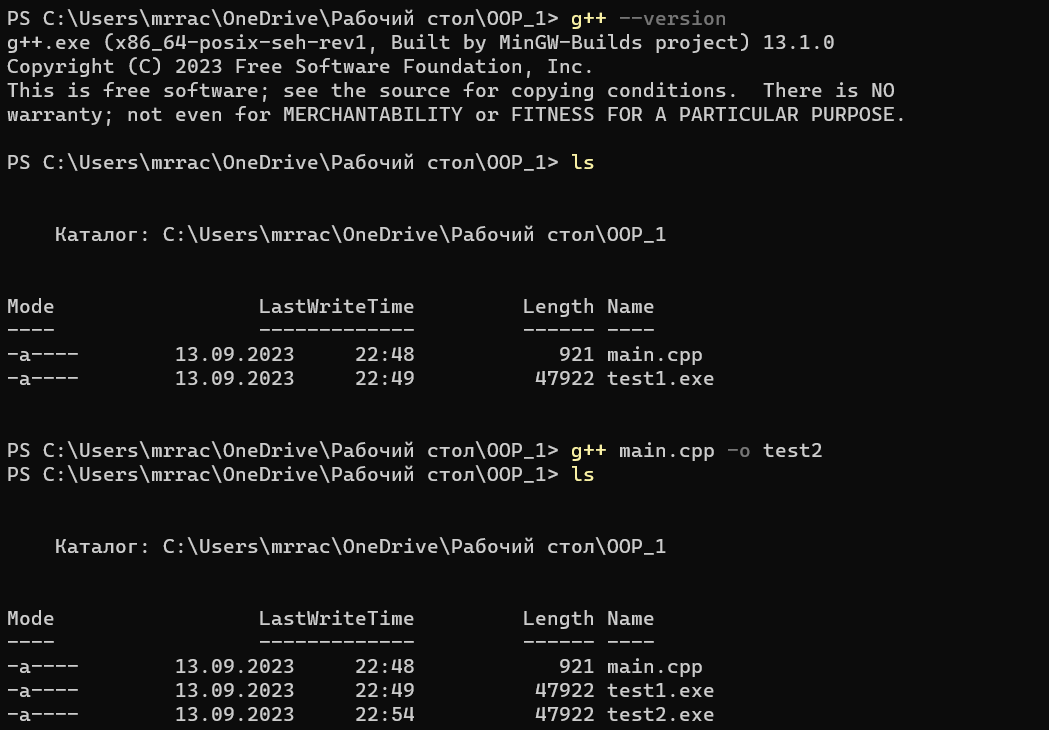
4.В папке с кодом программы нажимаем в пустом месте шифт и правую кнопку мыши для того чтобы открыть консоль именно в этой папке



5.Запускаем консоль



6.Проверяем версию компилятора командой (g++ --version), проверяем список файлов в папке (ls), создаем программу запуска и запускаем exe (g++ .\main.cpp –o test1), выполняем программу.



В консоли чтобы посмотреть историю команд используй стрелоки.

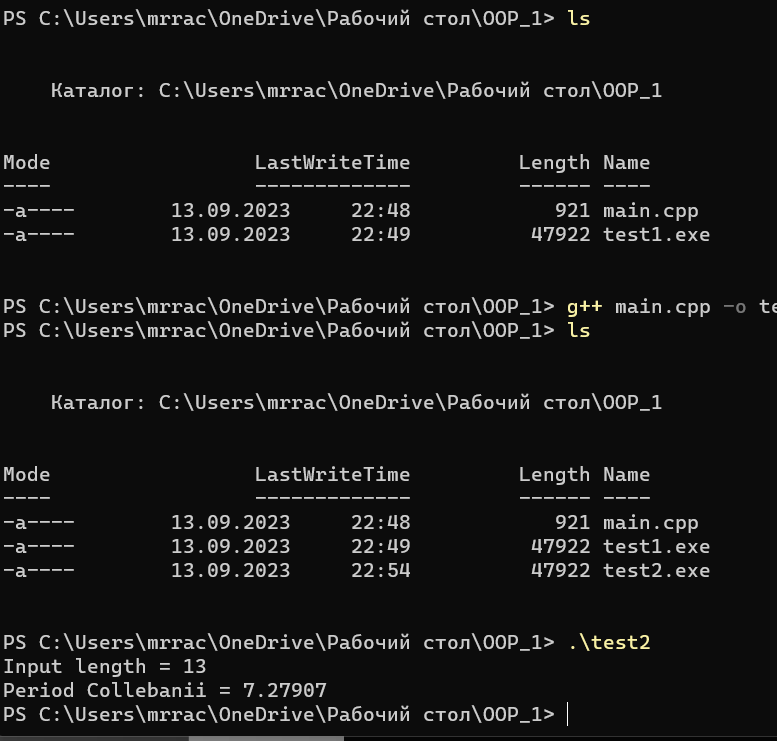
Для копирования выдели текст и нажми enter.

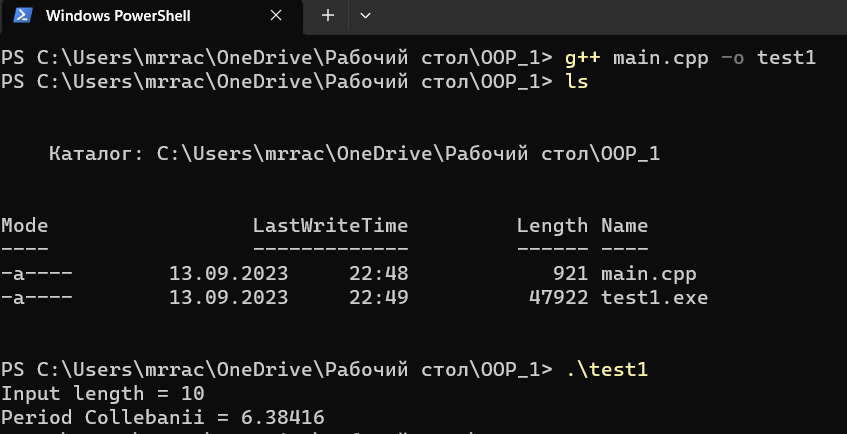
Для помощи в написании названия используй tab(автодополнение).

Для эскренного прекращения команды используй ctr + c.

ls показывает какие файлы есть в директории в которой мы находимся.

cd помогает изменить директорию в которой мы находимся пример (cd C:\......).





Источники: пара по ООП.

Выполнил Богушевич Данил ВМК-22

**Тернарный оператор**

*Важное примечание для использования ключевых слов, операторов языка программирования не нужно подключать библиотеку.*

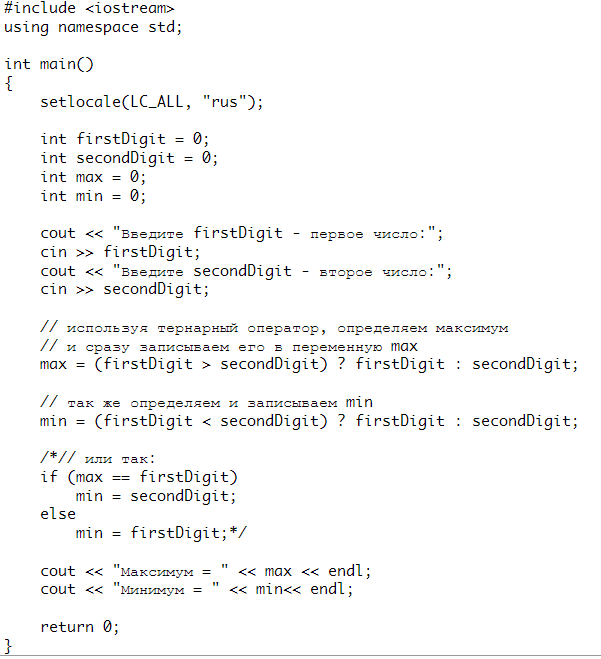
*Для использования функции нужно подключить стандартную библиотеку <iostream>.*

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

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

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

Пример \\ В нем, с помощью тернарного оператора, определяется минимальное и максимальное число из двух значений, которые введет пользователь.



Выполнил Богушевич Данил ВМК – 22

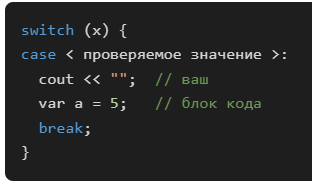
Оператор switch case

## Что такое switch

Это множественный условный оператор. Он работает также как несколько условий if, но может проверять лишь одну переменную в своих условиях. Если одно условие является правильным происходит выход.

Для его использования не нужно подключать какие-то дополнительные библиотеки, кроме стандартной <iostream>.

Для проверки условия нужно использовать конструкцию **case**:

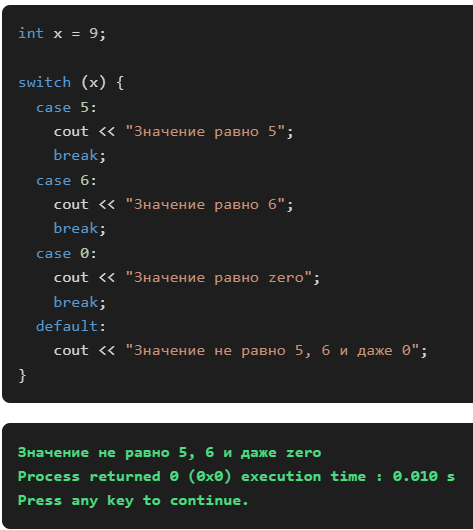


После слова case используются : (двоеточие), а в самом ее конце должен присутствовать оператор break.

### Что такое default

Он работает лишь когда все case являются не верными. В конструкции использовать не обязательно.

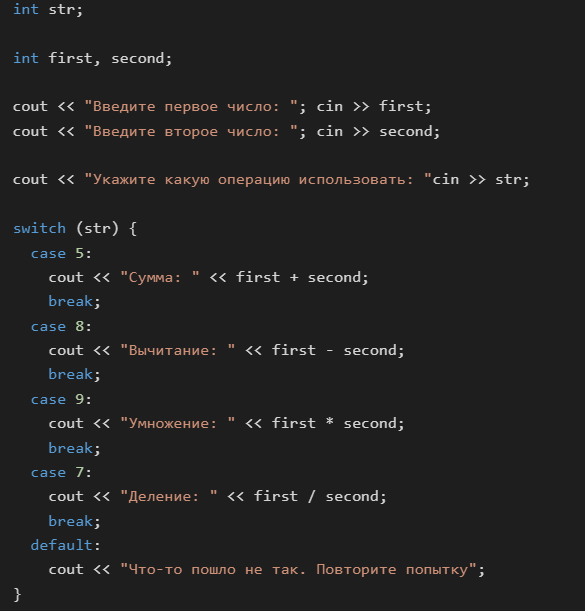
Если ни одно case и default не правильны, то программа выйдет из switch и продолжит работать дальше.



## Пример использования switch

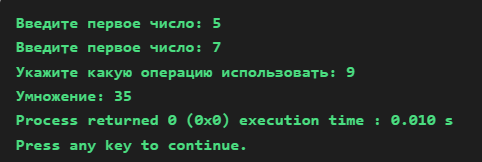
Давайте сделаем программу с использованием switch. Смысл программы будет заключаеться в нахождении суммы, разности, умножения, деления двух чисел, которые, как и операции будет выбирать пользователь.

* 5 - сумма
* 8 - разность
* 9 - умножение
* 7 – деление



* **В строках 10, 13, 16, 19:** применяется проверка значений str.
* **В строке 22:** если пользователь ошибся с числом, то программа его оповестит с помощью оператора default.

Вот пример успешного выполнения программы:

****

**Источник: https://metanit.com/cpp/tutorial/2.12.php**

## **Выполнил Богушевич Данил ВМК-22**

# Пространства имен (C++)

Пространство имен — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек. Все идентификаторы в пределах пространства имен доступны друг другу без уточнения. Идентификаторы за пределами пространства имен могут обращаться к членам, используя полное имя для каждого идентификатора, например std::vector<std::string> vec;, или с помощью [объявления using](https://learn.microsoft.com/ru-ru/cpp/cpp/using-declaration?view=msvc-170) для одного идентификатора (using std::string) или [директивы using](https://learn.microsoft.com/ru-ru/cpp/cpp/namespaces-cpp?view=msvc-170#using_directives) для всех идентификаторов в пространстве имен (using namespace std;). Код в файлах заголовков всегда должен содержать полное имя в пространстве имен.

В следующем примере показано объявление пространства имен и продемонстрированы три способа доступа к членам пространства имен из кода за его пределами.

C++

namespace ContosoData

{

class ObjectManager

{

public:

void DoSomething() {}

};

void Func(ObjectManager) {}

}

Использование полного имени:

C++

ContosoData::ObjectManager mgr;

mgr.DoSomething();

ContosoData::Func(mgr);

Чтобы добавить в область видимости один идентификатор, используйте объявление using:

C++

using ContosoData::ObjectManager;

ObjectManager mgr;

mgr.DoSomething();

Чтобы добавить в область видимости все идентификаторы пространства имен, используйте директиву using:

C++

using namespace ContosoData;

ObjectManager mgr;

mgr.DoSomething();

Func(mgr);

## Директивы using

Директива **using** позволяет использовать все имена в без имени пространства имен в **namespace** качестве явного квалификатора. Используйте директиву using в файле реализации (т. е. \*.cpp), если вы используете несколько разных идентификаторов в пространстве имен; Если вы используете только один или два идентификатора, рассмотрите возможность объявления using, чтобы перенести эти идентификаторы только в область, а не все идентификаторы в пространстве имен. Если локальная переменная имеет такое же имя, как и переменная пространства имен, то переменная пространства имен будет скрытой. Создавать переменную пространства имен с те же именем, что и у глобальной переменной, является ошибкой.

Примечание

Директиву using можно поместить в верхнюю часть CPP-файла (в области видимости файла) или внутрь определения класса или функции.

Без особой необходимости не размещайте директивы using в файлах заголовков (\*.h), так как любой файл, содержащий этот заголовок, добавит все идентификаторы пространства имен в область видимости, что может вызвать скрытие или конфликты имен, которые очень трудно отлаживать. В файлах заголовков всегда используйте полные имена. Если эти имена получаются слишком длинными, используйте псевдоним пространства имен для их сокращения. (См. ниже.)

## Объявление пространств имен и их членов

Как правило, пространство имен объявляется в файле заголовка. Если реализации функций находятся в отдельном файле, определяйте имена функций полностью, как показано в следующем примере.

C++

//contosoData.h

#pragma once

namespace ContosoDataServer

{

void Foo();

int Bar();

}

Реализации функций в contosodata.cpp должны использовать полное имя, даже если вы размещаете директиву **using** в верхней части файла:

C++

#include "contosodata.h"

using namespace ContosoDataServer;

void ContosoDataServer::Foo() // use fully-qualified name here

{

// no qualification needed for Bar()

Bar();

}

int ContosoDataServer::Bar(){return 0;}

Пространство имен может быть объявлено в нескольких блоках в одном файле и в нескольких файлах. Компилятор соединит вместе все части во время предварительной обработки и полученное в результате пространство имен будет содержать все члены, объявленные во всех частях. Примером этого является пространство имен std, которое объявляется в каждом из файлов заголовка в стандартной библиотеке.

Члены именованного пространства имен могут определяться за его границами, если они объявлены путем явной квалификации определяемого пространства имен. Однако определение должно располагаться после точки объявления в пространстве имен, окружающем то пространство имен, где находится объявление. Пример:

C++

// defining\_namespace\_members.cpp

// C2039 expected

namespace V {

void f();

}

void V::f() { } // ok

void V::g() { } // C2039, g() is not yet a member of V

namespace V {

void g();

}

Эта ошибка может возникнуть, когда члены пространства имен объявляются в нескольких файлах заголовка и эти заголовки не включены в правильном порядке.

## Глобальное пространство имен

Если идентификатор не объявлен явно в пространстве имен, он неявно считается входящим в глобальное пространство имен. Как правило, старайтесь по возможности избегать объявлений в глобальных область, за исключением точки входа [main функции](https://learn.microsoft.com/ru-ru/cpp/c-language/main-function-and-program-execution?view=msvc-170), которая должна находиться в глобальном пространстве имен. Чтобы явно указать глобальный идентификатор, используйте оператор разрешения области видимости без имени, как сделано в ::SomeFunction(x);. Это позволит отличать данный идентификатор от любого другого элемента с таким же именем, находящегося в другом пространстве имен. Кроме того, это облегчит понимание кода.

## Пространство имен std

Все типы и функции стандартной библиотеки C++ объявляются в std пространстве имен или пространствах имен, вложенных в std.

## Вложенные пространства имен

Пространства имен могут быть вложенными. Обычное вложенное пространство имен имеет неквалифицированный доступ к родительским членам, но родительские члены не имеют неквалифицированного доступа к вложенному пространству имен (если оно не объявлено как встроенное), как показано в следующем примере:

C++

namespace ContosoDataServer

{

void Foo();

namespace Details

{

int CountImpl;

void Ban() { return Foo(); }

}

int Bar(){...};

int Baz(int i) { return Details::CountImpl; }

}

Обычные вложенные пространства имен можно использовать для инкапсуляции данных о внутренней реализации, которые не являются частью открытого интерфейса родительского пространства имен.

## Встроенные пространства имен (C++ 11)

В отличие от обычных вложенных пространств имен члены встроенного пространства имен обрабатываются как члены родительского пространства имен. Эта особенность позволяет выполнять поиск перегруженных функций с зависимостью от аргументов среди функции, которые имеют перегрузки в родительском и вложенном встроенном пространстве имен. Это также позволяет объявлять специализации в родительском пространстве имен для шаблонов, объявленных во встроенном пространстве имен. В следующем примере показано, как внешний код привязывается к встроенному пространству имен по умолчанию.

C++

//Header.h

#include <string>

namespace Test

{

namespace old\_ns

{

std::string Func() { return std::string("Hello from old"); }

}

inline namespace new\_ns

{

std::string Func() { return std::string("Hello from new"); }

}

}

#include "header.h"

#include <string>

#include <iostream>

int main()

{

using namespace Test;

using namespace std;

string s = Func();

std::cout << s << std::endl; // "Hello from new"

return 0;

}

В следующем примере показано, как можно объявить специализацию в родительском пространстве имен шаблона, объявленного во встроенном пространстве имен.

C++

namespace Parent

{

inline namespace new\_ns

{

template <typename T>

struct C

{

T member;

};

}

template<>

class C<int> {};

}

Встроенные пространства имен можно использовать как механизм управления версиями для управления изменениями в открытом интерфейсе библиотеки. Например, можно создать одно родительское пространство имен и инкапсулировать каждую версию интерфейса в своем собственном пространстве имен, вложенном в родительское. Пространство имен, которое содержит самую последнюю или основную версию, квалифицируется как встроенное и поэтому представляется так, будто оно является непосредственным членом родительского пространства имен. Клиентский код, вызывающий Parent::Class, автоматически привязывается к новому коду. Клиенты, которые предпочитают использовать старую версию, могут по-прежнему получить доступ к ней, используя полный путь к вложенному пространству имен, содержащему данный код.

Ключевое слово inline должно применяться к первому объявлению пространства имен в единице компиляции.

В следующем примере показано две версии интерфейса: каждое — во вложенном пространстве имен. Пространство имен v\_20 содержит некоторые изменения из интерфейса v\_10 и помечается как встроенное. Клиентский код, который использует новую библиотеку и вызывает Contoso::Funcs::Add, вызовет версию v\_20. Код, который пытается вызвать Contoso::Funcs::Divide, теперь будет вызывать ошибку времени компиляции. Если действительно требуется эта функция, доступ к версии v\_10 можно получить путем явного вызова Contoso::v\_10::Funcs::Divide.

C++

namespace Contoso

{

namespace v\_10

{

template <typename T>

class Funcs

{

public:

Funcs(void);

T Add(T a, T b);

T Subtract(T a, T b);

T Multiply(T a, T b);

T Divide(T a, T b);

};

}

inline namespace v\_20

{

template <typename T>

class Funcs

{

public:

Funcs(void);

T Add(T a, T b);

T Subtract(T a, T b);

T Multiply(T a, T b);

std::vector<double> Log(double);

T Accumulate(std::vector<T> nums);

};

}

}

## Псевдонимы пространств имен

Имена пространств имен должны быть уникальными, из-за чего зачастую они получаются не слишком короткими. Если длина имени затрудняет чтение кода или утомительно для ввода в файл заголовка, в котором невозможно использовать директивы using, можно создать псевдоним пространства имен, который служит сокращением для фактического имени. Пример:

C++

namespace a\_very\_long\_namespace\_name { class Foo {}; }

namespace AVLNN = a\_very\_long\_namespace\_name;

void Bar(AVLNN::Foo foo){ }

## анонимные или безымянные пространства имен

Вы можете создать явное пространство имен, но не присвоить ему имя.

C++

namespace

{

int MyFunc(){}

}

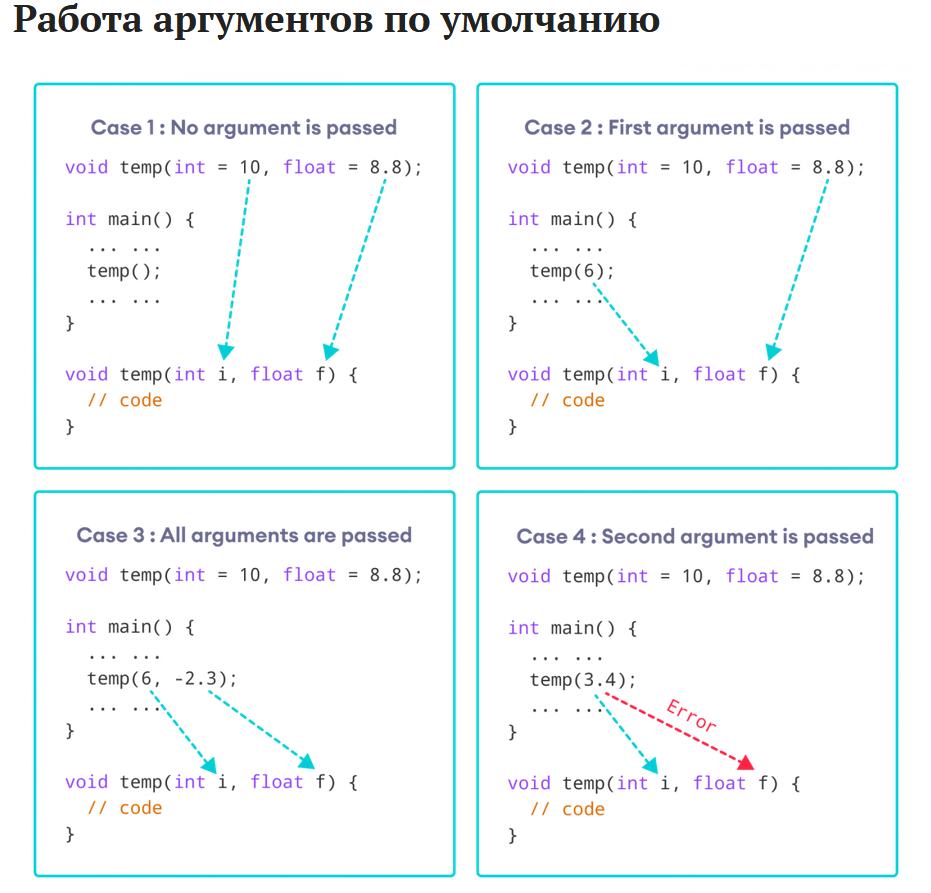
Это называется неименованным или анонимным пространством имен и полезно, когда требуется сделать объявления переменных невидимыми для кода в других файлах (т. е. предоставить им внутреннюю компоновку) без создания именованного пространства имен. Весь код, находящийся в том же файле, может видеть идентификаторы в безымянном пространстве имен, но эти идентификаторы, а также само пространство имен, будет невидимым за пределами этого файла или, точнее, вне блока перевода.

**Источник:** [**https://learn.microsoft.com/ru-ru/cpp/cpp/namespaces-cpp?view=msvc-170**](https://learn.microsoft.com/ru-ru/cpp/cpp/namespaces-cpp?view=msvc-170)

**Выполнил** *Богушевич Данил ВМК - 22*

**Значение аргументов функции по умолчанию**

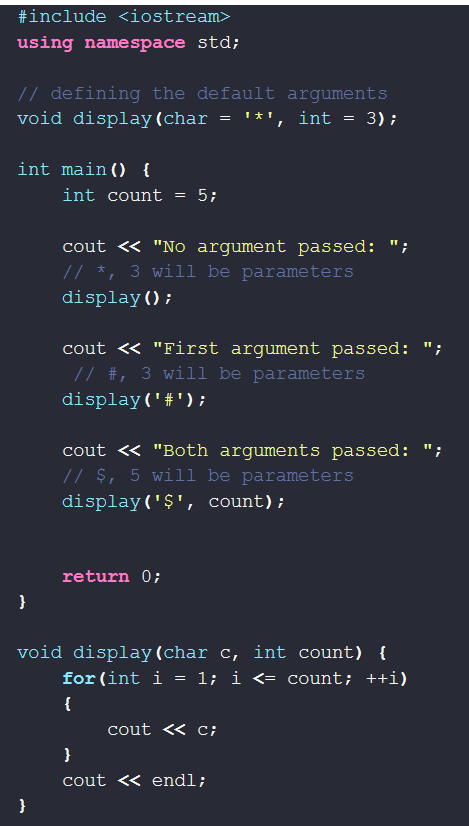
В программировании на С++ мы можем предоставить значения по умолчанию для параметров функции. Если функция с аргументами по умолчанию вызывается без передачи аргументов, то используются параметры по умолчанию. Однако, если аргументы передаются при вызове функции, аргументы по умолчанию игнорируются.



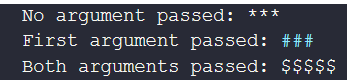
Мы можем понять работу аргументов по умолчанию из изображения выше:

* Когда вызывается temp(), функция использует оба параметра по умолчанию.
* Когда вызывается temp(6), первый аргумент становится 6, в то время как значение по умолчанию используется для второго параметра.
* Когда вызывается temp(6, -2,3), оба параметра по умолчанию переопределяются, в результате чего i = 6 и f = -2,3.
* Когда передается temp(3.4), функция ведет себя нежелательным образом, потому что второй аргумент не может быть передан без передачи первого аргумента. Таким образом, 3.4 передается как первый аргумент. Поскольку первый был определен как int, фактически передается значение 3.

**Пример:**



Вывод на экран:



Вот как работает эта программа:

* display() вызывается без передачи аргументов. В этом случае display() использует оба параметра по умолчанию c = ‘\*’ и n = 1.
* display(‘#’) вызывается только с одним аргументом. В этом случае первая цифра становится “#”. Второй параметр по умолчанию n = 1 сохраняется.
* display(‘#’, count) вызывается с обоими аргументами. В этом случае аргументы по умолчанию не используются.

Источник: https://calmsen.ru/argumenty-po-umolchaniyu-c-plusplus/

**Выполнил** *Богушевич Данил ВМК – 22*

# Конспект: Эллипсис

Язык C (и, по наследству, C++) допускает определение функций, принимающих переменное количество параметров. Это довольно низкоуровневый механизм, и не существует стандартного способа определить, сколько параметров было помещено в стек вызовов вызывающей функцией или какого они типа, поэтому обычно эта информация в некоторой форме передаётся первым параметром. Примерами таких функций являются стандартные функции printf и scanf. Общий синтаксис заголовка функции с переменным числом параметров имеет следующий вид:

Тип-результатаИмя-функции*(*Обязательные-параметры*, ...)*

Обязательные параметры объявляются так же, как в обычных функциях. Формально C++ (но не C) позволяет объявлять функции с переменным числом параметров, которые не принимают обязательных параметров, но определить такую функцию в рамках стандарта языка не получится.

При вызове функции с переменным числом параметров необходимо указать значения всех обязательных параметров и далее можно указать произвольное (от нуля до некоторого максимума, зафиксированного компилятором) количество дополнительных параметров, которые могут иметь разный тип. При этом при вызове значения типа float автоматически приводятся к double, значения целочисленных типов меньшей ширины, чем int (например, bool, char, short), приводятся к int.

Для извлечения дополнительных параметров из стека вызовов стандартом предусмотрен следующий набор определений, размещённый в заголовочном файле [cstdarg](http://en.cppreference.com/w/cpp/header/cstdarg):

* **va\_list** — тип данных, описывающий состояние стека;
* **va\_start**(args, last) — макрос, инициализирующий переменную args типа va\_list, last — имя последнего из обязательных параметров. Данный макрос можно применить лишь однажды за вызов функции (иначе [UB](https://teccxx.neocities.org/mx1/ub.html)).
* **va\_copy**(args1, args2) [C99, C++11] — макрос, выполняющий копирование args2 в args1 (оба параметра — переменные типа типа va\_list). Копирование позволяет повторно пройти по параметрам.
* **va\_arg**(args, T) — макрос, извлекающий следующий (в соответствии с состоянием, хранящимся в переменной args) параметр типа T из стека вызовов.
* **va\_end**(args) — макрос, корректно завершающий извлечение параметров из стека. Этот макрос должен быть применён к каждой переменной типа va\_list, которая до того была инициализирована с помощью va\_start или va\_copy (иначе UB).

Пример реализации функции с переменным числом параметров типа double:

*///* Длина вектора с количеством компонент comps.

*///* Компоненты вектора передаются параметрами функции.

double vec\_len(int comps, ...)

{

std::va\_list args;

va\_start(args, comps);

double s = 0.0;

**while** (comps-- > 0)

{

const double arg = va\_arg(args, double);

s += arg \* arg;

}

va\_end(args);

**return** std::sqrt(s);

}

Источники

1. <https://ru.stackoverflow.com/questions/1174618/%D0%A7%D1%82%D0%BE-%D0%BF%D1%80%D0%B5%D0%B4%D0%BF%D0%BE%D1%87%D0%B5%D1%81%D1%82%D1%8C-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8E-%D1%81-%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D1%8B%D0%BC-%D0%BA%D0%BE%D0%BB%D0%B8%D1%87%D0%B5%D1%81%D1%82%D0%B2%D0%BE%D0%BC-%D0%B0%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%BE%D0%B2-%D0%B8%D0%BB%D0%B8-%D0%BF%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D1%83-%D0%BC%D0%B0%D1%81%D1%81%D0%B8>
2. <https://www.cyberforum.ru/cpp-beginners/thread158423.html>
3. <https://learn.microsoft.com/ru-ru/cpp/cpp/ellipses-and-variadic-templates?view=msvc-170>
4. <https://teccxx.neocities.org/mx1/memory>

# Выполнил Богушевич Данил ВМК -22

# Конспект: Устройство памяти программы. Где хранятся: глобальные переменные, локальные переменные, аргументы функций, динамические переменные

Работа вычислителя.

Внутри вычислителя есть память, в которой находится программа. Каждая следующая вызванная функция/программа обычно расположена в памяти с большим адресом, и является первой на освобождение (очищение).  
Память под программу рассчитывается на этапе КОМПИЛИРОВАНИЯ и ЛИНКОВКИ и выделяется в процессе выполнения программы.

**Следовательно**

* внутри оперативной памяти идет разбиение (фрагментация) на зоны для каждой программы.
* при вызове дополнительных программ или функций также выделяется место в памяти
* при выходе из функции/ программы место очищается.

Скомпилированная программа имеет тот объем памяти, что ему задали  
Данная память, рассчитанная на этапе компиляции/линковки и выделенная в ходе запуска программы называется **стеком (stack)**.  
Следует еще раз акцентировать внимание, что программы запускаются по принципу LIFO (Last In, First Out),  то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека.  
Так как мы не знаем сколько нам ещё понадобится памяти, ведь она уже выделена в стеке  
Для этого можно использовать остальную память – она называется **куча (heap)**. В ходе работы программы мы можем **динамически** (по своему хотению, в любой момент времени, при наличии свободного места) выделять дополнительную память и работать через ее адреса.  
Эта дополнительная память, выделенная в куче и с которой работают из стека называется **динамической памятью.**   
**Примечание:** На самом деле **не всегда** стек и куча расположены в одной зоне памяти, физически для более быстрого выполнения стек может быть перенесен в область **кэш (cash)** процессора (эта память обычно существенно быстрее оперативной), но за счет адресации и работы ide данные.  
   
Важно отметить, что разработчик сам выделяет память в куче, чем может ограничить работу других программ, 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/ перезапуска вычислителя.  
Данный механизм потери памяти из-за потери указателя называется утечками leaks.  
Если утечек будет много — память закончится и код не сможет выполняться.

Память, которую использует программа делится на три вида:

1. Статическая память (static memory)
   * хранит глобальные переменные и константы;
   * размер определяется при компиляции.
2. Стек (stack)
   * хранит локальные переменные, аргументы функций и промежуточные значения вычислений;
   * размер определяется при запуске программы (обычно выделяется 4 Мб).
3. Куча (heap)
   * динамически распределяемая память;
   * ОС выделяет память по частям (по мере необходимости).

Динамически распределяемую память следует использовать в случае если мы заранее (на момент написания программы) не знаем сколько памяти нам понадобится (например, размер массива зависит от того, что введет пользователь во время работы программы) и при работе с большими объемами данных (например, массив из 1 000 000 int`ов не поместится на стеке).

**Глобальная переменная** — переменная, определённая вне функций и доступная из разных функций.

**Локальная переменная** — переменная, определённая внутри некоторой функции и доступная только из неё.

**Аргументы функций** — значения, передаваемые параметрам функции при ее вызове, называются аргументами.

**Динамическая переменная** — переменная в программе, место в оперативной памяти под которую выделяется во время выполнения программы.

Источники

1. <https://teccxx.neocities.org/mx1/memory>
2. <https://cpp.com.ru/shildt_spr_po_c/02/0204.html>
3. <https://logic.pdmi.ras.ru/~smal/aptu/cpp10/2010_09_17.html>

# Выполнил Богушевич Данил ВМК - 22

# Конспект: Обработка исключительных ситуаций

В процессе работы программы могут возникать различные ошибки. Например, при передаче файла по сети оборвется сетевое подключение или будут введены некорректные и недопустимые данные, которые вызовут падение программы. Такие ошибки еще называются исключениями. Исключение представлякт временный объект любого типа, который используется для сигнализации об ошибке. Цель объекта-исключения состоит в том, чтобы передать информацию из точки, в которой произошла ошибка, в код, который должен ее обработать. Если исключение не обработано, то при его возникновении программа прекращает свою работу.

Например, в следующей программе происходит деление чисел:

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

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

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

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

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

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

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

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

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

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

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

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

Так изменим весь код следующим образом:

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

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

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

В итоге когда выполнение программы дойдет до строки

**double z {divide(x, y)};**

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

Error!

The End...

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

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

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

Division by zero!

The End...

Таким образом, мы можем узнать суть возникшего исключения. Подобным образом мы можем передавать информацию об исключении через любые типы, например, std::string:

|  |  |
| --- | --- |
| 1 | throw std::string{"Division by zero!!"}; |

Тогда в блоке **catch** мы можем получить эту информацию в виде объекта std::string:

|  |  |
| --- | --- |
| 1  2  3  4 | catch (std::string error\_message)  {      std::cout << error\_message << std::endl;  } |

Если же исключение не обработано, то вызывается функция **std::terminate()** (из модуля <exception> стандартной библиотеки C++), которая, в свою очередь, по умолчанию вызывает другую функцию - **std::abort()** (из <cstdlib>), которая собственно и завершает программу.

Существует очень много функций и в стандартной библиотеке С++, и в каких-то сторонних библиотеках. И может возникнуть вопрос, какие из них вызывать в конструкции try-catch, чтобы не столкнуться с необработанным исключением и аварийным завершением программы. В этом случае может помочь прежде всего документация по функции (при ее наличии). Другой сигнал - ключевое слово **noexcept**, которое при использовании в заголовке функции указывает, что эта функция никогда не будет генерировать исключения. Например:

|  |  |
| --- | --- |
| 1 | void print(int argument) noexcept; |

Здесь указываем, что функция print() никогда не вызовет исключение. Таким образом, встретив функцию с подобным ключевым словом, можно ожидать, что она не вызовет исключения. И соответственно нет необходимости помещать ее вызов в конструкцию try-catch.

Мы можем генерировать и обрабатывать несколько разных исключительных ситуаций. Допустим, нам надо, чтобы при делении делитель был меньше делимого:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | #include <iostream>    double divide(int a, int b)  {      if(!b)  // если b == 0      {          throw 0;      }      if(b > a)      {          throw "The first number is greater than the second one";      }      return a / b;  }    void test(int a, int b)  {      try      {          double result {divide(a, b)};          std::cout << result << std::endl;      }      catch (int code)      {          std::cout << "Error code: " << code << std::endl;      }      catch (const char\* error\_message)      {          std::cout << error\_message << std::endl;      }  }    int main()  {      test(100, 20);      // 5      test(100, 0);       // Error code: 0      test(100, 1000);    // The first number is greater than the second one  } |

В функции divide в зависимости от значения числа b оператору throw передаем либо число:

|  |
| --- |
| 1 throw 0; |

либо строковый литерал:

|  |  |
| --- | --- |
| 1 | throw "The first number is greater than the second one"; |

ля обработки каждого типа исключений определены два разных блока catch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | catch (int code)  {      std::cout << "Error code: " << code << std::endl;  }  catch (const char\* error\_message)  {      std::cout << error\_message << std::endl;  }  Таким образом, в данном случае мы получим следующий консольный вывод:  5  Error code: 0  The first number is greater than the second one  Если, нет блока catch для обработки исключения указанного типа, то при генерации исключения  программа не найдет нужный блок catch для обработки исключения, и программа аварийно  завершит свое выполнение. |

Источники

1. <https://metanit.com/cpp/tutorial/6.1.php>
2. <https://tproger.ru/articles/iskljuchenija-v-cpp-tipy-sintaksis-i-obrabotka>

# Богушевич Данил ВМК-22

# Конспект: Этапы компиляции программы на C++

**Исходный C++ файл** — это всего лишь код, но его невозможно запустить как программу или использовать как библиотеку. Поэтому каждый исходный файл требуется скомпилировать в исполняемый файл, динамическую или статическую библиотеки.

Перед тем, как приступать, давайте создадим исходный .cpp файл, с которым и будем работать в дальнейшем.

**driver.cpp(Код)**:

#**include** <iostream

**using** **namespace** std;

#**define** RETURN return 0

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

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

RETURN;

}

## Этапы компиляции:

### Препроцессинг

Самая первая стадия компиляции программы.

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

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

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

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

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

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

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

**return** 0;

}

[driver.ii](https://pastebin.com/7EDJTPKK)

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

### 2) 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.

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

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

driver.s

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

### Ассемблирование

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

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

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

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

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

$ as driver.s -o driver.o

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

### 4) Компоновка

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

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

Получим исполняемый файл **driver**:

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

### 5) Загрузка

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

Запустим нашу программу:

$ ./driver

// Hello, world!

## Источники:

## <https://habr.com/ru/articles/478124/>

## <https://cpp-kt.github.io/cpp-notes/05_compilation.html>

# Выполнил Богушевич Данил ВМК-22

# Конспект: Перечислимый тип

Перечисления (enum) представляют еще один способ определения своих типов. Их отличительной особенностью является то, что они содержат набор числовых констант. Перечисление имеет следующую форму:

|  |  |
| --- | --- |
| 1 | enum class имя\_перечисления { константа\_1, константа\_2, ... константа\_N}; |

После ключевых **enum class** идет название перечисления, и затем в фигруных скобках перечисляются через запятую константы перечисления.

Определим простейшее перечисление:

|  |  |
| --- | --- |
| 1 | enum class Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} |

В данном случае перечисление называется **Day** и представляет дни недели. В фигурных скобках заключены все дни недели. Фактически они представляют числовые константы.

Каждой константе сопоставляется некоторое числовое значение. По умолчанию первая константа получает в качестве значения 0, а остальные увеличиваются на единицу. Так, в примере выше Monday будет иметь значение 0, Tuesday - 1 и так далее. Таким образом, последняя константа - Sunday будет равна 6.

После создания перечисления мы можем определить его переменную и присвоить ей одну из констант:

|  |  |
| --- | --- |
| 1  2  3 | Day today {Day::Thursday};  // или так  //Day today = Day::Thursday; |

В данном случае определяется переменная today, которая равна Day::Thursday, то есть четвертой константе перечисления Day.

Чтобы вывести значение переменной на консоль, можно использовать преобразование к типу целочисленному типу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #include <iostream>    enum class Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};  int main()  {      Day today {Day::Thursday};      std::cout << "Today: " << static\_cast<int>(today) << std::endl;  } |

То есть в данном случае на консоль будет выведено Today: 3, так как константа Thursday имеет значение 3.

Мы также можем управлять установкой значений в перечислении. Так, мы можем задать начальное значение для одной константы, тогда у последующих констант значение увеличивается на единицу:

|  |  |
| --- | --- |
| 1 | enum class Day {Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}; |

В данном случае Tuesday будет равно 2, а Sunday - 7.

Можно назначить каждой константе индивидуальное значение или сочетать этот подход с автоустановкой:

|  |  |
| --- | --- |
| 1 | enum class Day {Monday = 2, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday=1}; |

В данном случае Saturday будет равно 7, а Sunday - 1.

Можно даже назначать двум константам одно и то же значение:

|  |  |
| --- | --- |
| 1 | enum class Day {Monday = 1, Mon = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }; |

Здесь константы Monday и Mon имеют одно и то же значение.

Можно присвоить константам значение уже имеющихся констант:

|  |  |
| --- | --- |
| 1 | enum class Day {Monday = 1, Mon = Monday, Tuesday = Monday + 1, Wednesday, Thursday, Friday, Saturday, Sunday }; |

Стоит учитывать, что константы перечисления должны представлять целочисленные константы. Однако мы можем выбрать другой целочисленный тип, например, char:

|  |  |
| --- | --- |
| 1 | enum class Operation: char {Add = '+', Subtract='-', Multiply='\*'}; |

Если мы захотим вывести значения этих констант на консоль в виде символов, то необходимо преобразовать их к типу char:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <iostream>    enum class Operation: char {Add = '+', Subtract='-', Multiply='\*'};  int main()  {      std::cout << "add: " << static\_cast<char>(Operation::Add) << std::endl;      std::cout << "subtracte: " << static\_cast<char>(Operation::Subtract) << std::endl;      std::cout << "multiply: " << static\_cast<char>(Operation::Multiply) << std::endl;  } |

### Применение перечислений

Перечисления удобны, когда необходимо хранить ограниченный набор состояний и в зависимости от текущего состояния выполнять некоторые действия. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #include <iostream>    enum class Operation {Add, Subtract, Multiply};    void calculate(int n1, int n2, Operation op)  {      switch (op)      {          case Operation::Add:              std::cout << n1 + n2 << std::endl;              break;          case Operation::Subtract:              std::cout << n1 - n2 << std::endl;              break;          case Operation::Multiply:              std::cout << n1 \* n2 << std::endl;              break;      }  }  int main()  {      calculate(10, 6, Operation::Add);           // 16      calculate(10, 6, Operation::Subtract);      // 4      calculate(10, 6, Operation::Multiply);      // 60  } |

В данном случае все арифметические операции хранятся в перечислении Operation. В функции calculate зависимости от значения третьего параметра - применяемой операции выполняются определенные действия с двумя первыми параметрами.

### Подключение констант перечисления

При обращении к константам перечисления по умолчанию необходимо указывать название перечисления, например, Day::Monday. Но начиная со стандарта C++20 мы можем подключить константы перечисления в текущий контекст с помощью оператора **using**.

|  |  |
| --- | --- |
| 1 | using enum Day; |

И в дальнейшем использовать только имя констант:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <iostream>    enum class Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};  using enum Day;     // подключаем константы перечисления в текущую область видимости    int main()  {      Day today {Thursday};   // используем только имя константы      // или так      //Day today = Thursday;      std::cout << static\_cast<int>(today) << std::endl;      // 3      // выводим значение констаты Sunday      std::cout << static\_cast<int>(Sunday) << std::endl;     // 6  } |

Также мы можем подключить только одну константу:

|  |  |
| --- | --- |
| 1  2  3 | using Day::Monday;     // подключаем только Monday  ........................  Day today {Monday}; |

В данном случае подключаем только константу Day::Monday. Для обращения к другим константам по-прежнему необходимо использовать имя перечисления.

Поскольку такая возможность добавлена лишь начиная со стандарта С++20, то при компиляции с g++ или clang++ добавляется соответствующий флаг - -std=c++20

### Перечисления в С-стиле

Стоит отметить, что раньше в С++ использовалась другая форма перечислений, которые пришли из языка С и определяются без ключевого слова class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <iostream>    enum Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};    int main()  {      Day today = Tuesday;      std::cout << today << std::endl;    // 1  } |

Такие перечисления еще называют unscoped (то есть не ограниченные ни какой областью видимостью). Естественно такие перечисления можно встретить в старых программах. Однако в виду того, что они потенциально могут привести к большему количеству ошибок, то в настоящее время такая форма все меньше и меньше используется.

Источники:

1. <https://metanit.com/cpp/tutorial/5.9.php>
2. <https://radioprog.ru/post/1156>
3. <https://sysprog.ru/post/perechisleniya-v-yazyke-c>

# Выполнил Богушевич Данил ВМК-22

# Конспект: Статические переменные

Статические переменные являются долговременными переменными, существующими на протяжении функции или файла. Они отличаются от глобальных переменных, поскольку не известны за пределами функции или файла, но могут хранить свои значения между вызовами. Данная возможность оказывается очень полезной тогда, когда необходимо написать универсальные функции и библиотеки функций, которые могут использоваться программистами. Поскольку эффект использования static на локальных переменных отличается от эффекта на глобальных переменных, то мы рассмотрим их по отдельности.

**Статические локальные переменные**

Когда static применяется к локальной переменной, это приводит к тому, что компилятор создает долговременную область для хранения переменной почти таким же способом, как это делается для глобальной переменной. Ключевое различие между статической локальной и глобальной переменными заключается в том, что статическая локальная переменная остается известной только в том блоке, в котором она была объявлена. Проще говоря, статическая локальная переменная - это локальная переменная, сохраняющая свое значение между вызовами функций.

Наличие статических локальных переменных очень важно для создания самостоятельных функций поскольку имеется несколько типов подпрограмм, сохраняющих значение между вызовами. Если использование статических переменных недопустимо, то следует использовать глобальные переменные, но это может привести к побочным эффектам. Ниже приведен простой пример того, как статитеская локальная переменная может использоваться в функции count():

#include <stdio.h>  
#include <conio.h>  
  
int count (int i) ;  
  
int main(void)  
{  
  
do {  
count(0);  
}  
while(!kbhit());  
printf("count called %d times", count (1));  
return 0;  
}  
  
int count (int i)  
{  
static int c=0;  
  
if(i) return c;  
else c++;  
return 0;  
}

Иногда полезно знать, как часто функция вызывается во время работы программы. Это возможно сделать через использование глобальных переменных, но лучшим вариантом является использование функции, сохраняющей информацию внутри себя, как это сделано в функции count(). В данном примере, если count() вызывается со значением 0, то переменная с увеличивается. (Скорее всего в настоящих приложениях функция будет также выполнять некоторую другую полезную работу.) Если count() вызывается с любым другим значением, то она возвращает число сделанных вызовов. Подсчет числа вызовов функции может быть полезен при разработке программы, которая вызывает эти функции достаточно часто, и требуется привлечь к вызовам внимание.

Другим хорошим примером функции, требующей использования статических локальных переменных, является генератор последовательности чисел, создающий новое число, основываясь на старом. Это можно сделать, объявив глобальную переменную. Тем не менее, каждый раз, когда функция используется в программе, следует помнить об объявлении глобальной переменной и постоянно необходимо смотреть: не конфликтует ли переменная с другими, ранее объявленными, переменными. Также использование глобальной переменной приводит к тому, что функцию трудно поместить в библиотеку функций. Лучшим решением является объявление переменной, содержащей сгенерированное число как static, как в нижеприведенном фрагменте.

int series(void)  
{  
static int series\_num;  
series\_num = series\_num+23;  
return(series\_num);  
}

В данном примере переменная series\_num существует между вызовами функций вместо того, чтобы каждый раз создаваться и уничтожаться как обычная локальная переменная. Это означает, что каждый вызов series() может создать новый член серии, основываясь на последнем члене без глобального объявления переменной.

Можно было заметить нечто необычное в функции series(). Статическая переменная series\_num не инициализируется. Это означает, что при первом вызове функции series\_num имеет значение по умолчанию 0. Хотя это приемлемо для некоторых приложений, большинство генераторов последовательности требуют какую-либо другую стартовую точку. Чтобы сделать это, требуется инициализировать series\_num до первого вызова series(), что может быть легко сделано, если series\_num является глобальной переменной. Тем не менее, следует избегать использования series\_num как глобальной переменной и лучше объявить ее как static. Это приводит ко второму способу использования static.

# Статические глобальные переменные

Когда спецификатор static применяется к глобальной переменной, он сообщает компилятору о необходимости создания глобальной переменной, которая будет известна только в файле, где статическая глобальная переменная объявлена. Это означает, что, даже если переменная является глобальной, другие подпрограммы в других файлах не будут знать о ней. Таким образом, не возникает повода для побочных эффектов. В некоторых ситуациях, где локальные статические переменные неприменимы, можно создать раздельно компилируемый файл и использовать его без боязни возникновения побочных эффектов.

Для того, чтобы понять, как можно использовать статические глобальные переменные, пример с генератором последовательности из предыдущего раздела переделан таким образом, что стартовое значение может использоваться для инициализации серии путем вызова второй функции — series\_start(). Ниже показан файл, содержащий series(), series\_start() и series\_num:  
  
/\* все должно быть в одном файле \* /  
static int series\_num;  
  
int series(void) ;  
void series\_start(int seed);  
  
series(void)  
{  
series\_num = series\_num + 23;  
return(series\_num);  
}  
  
/\* инициализация series\_num \*/  
void series\_start (int seed)  
{  
series\_num = seed;  
}

Вызывая series\_start() с некоторым известным целым числом, мы инициализируем генератор последовательности. После этого вызов series() приводит к генерации следующего элемента последовательности.

Имена статических локальных переменных известны только функции или блоку кода, в которых они объявлены, а имена статических глобальных переменных известны только в файле, в котором они находятся. Это означает, что если поместить функции series() и series\_start() в отдельный файл, то можно использовать данные функции, но нельзя обращаться к переменной series\_num. Она спрятана от остального кода программы. Фактически можно даже объявлять и использовать другую переменную, называемую series\_num, в программе (в другом файле) и не бояться напутать. В сущности модификатор static разрешает использование функциями переменных, не беспокоя другие функции.

Статические переменные позволяют прятать части программы. Это может привести к большим преимуществам при разработке больших и сложных программ.

Источники:

1. <https://www.c-cpp.ru/books/staticheskie-globalnye-peremennye>
2. <https://habr.com/ru/articles/527044/>