

Оглавление

Значения аргументов функции по умолчанию	2
Устройство памяти программы	3
Правила оформления кода	6
Пробелы и отступы	6
Названия и переменные	6
Базовые выражения C++	7
Чрезмерность	8
Комментарии	8
Эффективность	9
Функции и процедурное проектирование	9
Параметры функции main (argc, argv).....	11
.....	11
Уменьшение размера исполняемого файла	13
Обработка исключительных ситуаций	14
Typedef.....	18
Static	21
Auto	24
Include guard.....	26
Ivalue и rvalue в C и C++.....	28
Этапы компиляции	32
Рефакторинг. Средства рефакторинга в C++.....	36
Файлы и папки проекта VisualStudio.	40
Отладка в VS.....	41
Источники	48

Значения аргументов функции по умолчанию

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

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

Общая форма объявления функции, которая содержит аргументы по умолчанию

```
int my_func(int a, int b, int c=12);
```

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

```
result = my_func(1, 2, 3);
```

```
result = my_func(1, 2);
```

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

Зачем они нужны?

1. Улучшение читаемости кода: Использование значений по умолчанию позволяет избегать избыточного кода и упрощает понимание того, как функция работает. Когда разработчик видит функцию с аргументами по

умолчанию, он может сразу понять, какие параметры можно не передавать, если это необходимо.

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

Пример объявления и использования функций, которые используют аргументы по умолчанию

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

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

```
#include "stdafx.h"
#include <iostream>
using namespace std;

// функция, содержащая аргумент по умолчанию
int Inc(int value, int step = 1)
{
    return value + step;
}

int _tmain(int argc, _TCHAR* argv[])
{
    //
    int v1, v2;

    v1 = 5;

    // использование аргумента по умолчанию
    v2 = Inc(v1); // v2 = 5 + 1 = 6
    v1 = 5;
    v2 = Inc(v1, 3); // v2 = 5 + 3 = 8

    cout << v2 << endl;
    return 0;
}
```

Устройство памяти программы

Память — одна из самых сложных тем в информатике, но понимание устройства памяти компьютера позволяет разрабатывать более

эффективные программы, а для более низкоуровневого программирования, например, при разработке ОС, это понимание и вовсе является обязательным.

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

Виды памяти

Существует 3 типа памяти: статический, автоматический и динамический.

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

```
int id = 150; // определение статической глобальной переменной
```

```
int main()
{
    std::cout << id + 8; // её использование
}
```

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

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

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

Проще всего это понять из примера на C++:

```
int main()
{
    int a = 3;
    int result = factorial(a);
    std::cout << result;
}

int factorial(int n)
{
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Динамическая — выделение памяти из ОС по требованию приложения.

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

В C++:

```
int *ptr_i;
double *ptr_d;
...
ptr_i = new int;
ptr_d = new double[10];
...
delete ptr_i;
delete[] ptr_d;
```

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

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

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

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

Правила оформления кода

Пробелы и отступы

Отделяйте пробелами фигурные скобки:

```
// Плохая практика
int x = 3, y = 7; double z = 4.25; x++;
if (a == b) { foo(); }
```

```
// Хорошая практика
int x = 3;
int y = 7;
double z = 4.25;
```

```
x++;
if (a == b) {
    foo();
}
```

Ставьте пробелы между операторами и операндами:

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

Названия и переменные

Давайте переменным описательные имена, такие

как `firstName` или `homeworkScore` . Избегайте однобуквенных названий вроде `x` или `c`, за исключением итераторов вроде `i`.

Называйте переменные и функции, используя верблюжий Регистр .
Называйте классы Паскальным Регистром, а константы —
в ВЕРХНЕМ_РЕГИСТРЕ.

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

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

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

```
// Плохая практика: текстовая строка в стиле Си
char* str = "Hello there";
```

```
// Хорошая практика: текстовая строка в стиле C++
string str = "Hello there";
```

Базовые выражения C++

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

```
// Плохая практика
printf("Hello, world!\n");
// Хорошая практика
cout << "Hello, world!" << endl;
```

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

```
// Повторяет 'size' раз
for (int i = 0; i < size; i++) {
    ...
}
```

```
// Повторяет, пока больше не будет строк
string str;
while (input >> str) {
    ...
}
```

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

Чрезмерность

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

```
// Плохая практика
foo();
x = 10;
y++;
...

foo();
x = 15;
y++;

// Хорошая практика
helper(10);
helper(15);
...

void helper(int newX) {
    foo();
    x = newX;
    y++;
}
```

Комментарии

Заглавный комментарий. Размещайте заглавный комментарий, который описывает назначение файла, вверху каждого файла. Предположите, что читатель вашего комментария является

продвинутым программистом, но не кем-то, кто уже видел ваш код ранее.

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

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

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

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

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

Эффективность

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

```
// Плохая практика
if (reallySlowSearchForIndex("abc") >= 0) {
    remove(reallySlowSearchForIndex("abc"));
}

// Хорошая практика
int index = reallySlowSearchForIndex("abc");
if (index >= 0) {
    remove(index);
}
```

Функции и процедурное проектирование

Хорошо спроектированная функция имеет следующие характеристики:

1. Полностью выполняет четко поставленную задачу;
2. Не берет на себя слишком много работы;
3. Не связана с другими функциями бесцельно;
4. Хранит данные максимально сжато;

5.Помогает распознать и разделить структуру программы;

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

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

```
/*
 * Решает квадратное уравнение  $ax^2 + bx + c = 0$ ,
 * внося результаты в root1 и root2.
 * Предполагается, что данные уравнения имеют два
 * корня.
 */
void quadratic(double a, double b, double c,
               double& root1, double& root2) {
    double d = sqrt(b * b - 4 * a * c);
    root1 = (-b + d) / (2 * a);
    root2 = (-b - d) / (2 * a);
}
```

Когда требуется вернуть значение из функции, используйте значение return:

```
// Плохая практика
void max(int a, int b, int& result) {
    if (a > b) {
        result = a;
    } else {
        result = b;
    }
}
```

```
// Хорошая практика
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

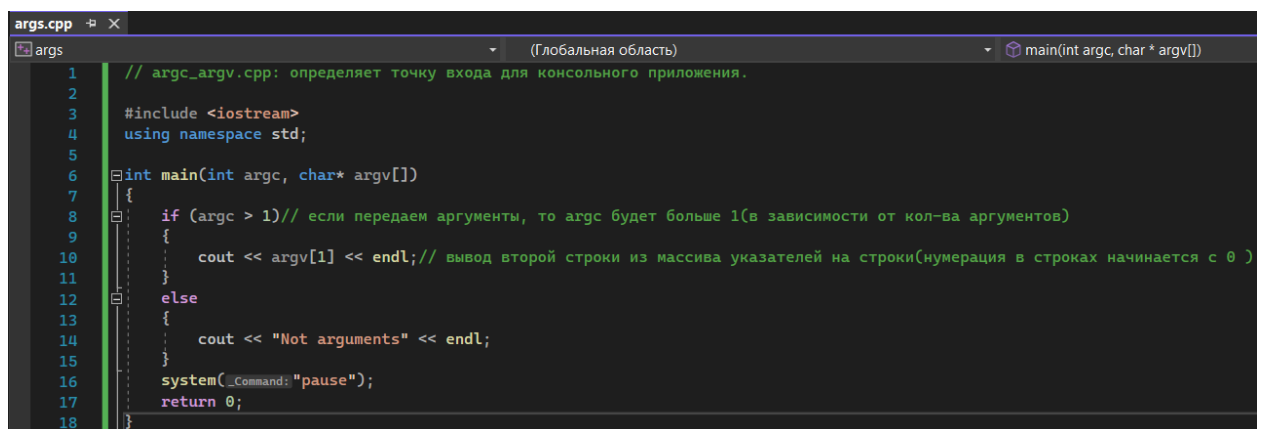
Параметры функции main (argc, argv)

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

```
int main(int argc, char* argv[]) // параметры функции main()
```

Эта строка — заголовок главной функции `main()`, в скобках объявлены параметры `argc` и `argv`. Так вот, если программу запускать через командную строку, то существует возможность передать какую-либо информацию этой программе, для этого и существуют параметры `argc` и `argv[]`. Параметр `argc` имеет тип данных `int`, и содержит количество параметров, передаваемых в функцию `main`.

Причем `argc` всегда не меньше 1, даже когда мы не передаем никакой информации, так как первым параметром считается имя функции. Параметр `argv[]` это массив указателей на строки. Через командную строку можно передать только данные строкового типа. Указатели и строки — это две большие темы, под которые созданы отдельные разделы. Так вот именно через параметр `argv[]` и передается какая-либо информация. Разработаем программу, которую будем запускать через командную строку Windows, и передавать ей некоторую информацию.



```
args.cpp
1 // argc_argv.cpp: определяет точку входа для консольного приложения.
2
3 #include <iostream>
4 using namespace std;
5
6 int main(int argc, char* argv[])
7 {
8     if (argc > 1) // если передаем аргументы, то argc будет больше 1(в зависимости от кол-ва аргументов)
9     {
10         cout << argv[1] << endl; // вывод второй строки из массива указателей на строки(нумерация в строках начинается с 0 )
11     }
12     else
13     {
14         cout << "Not arguments" << endl;
15     }
16     system(_Command: "pause");
17     return 0;
18 }
```

Рисунок 1 - Код

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

```
C:\WINDOWS\system32\cmd.exe - C:\Users\wolfd\Desktop\Учеба\ООП\Конспекты\args\x64\Debug\args.exe
Microsoft Windows [Version 10.0.22000.1219]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\wolfd>C:\Users\wolfd\Desktop\Учеба\ООП\Конспекты\args\x64\Debug\args.exe
Not arguments
Для продолжения нажмите любую клавишу . . .
```

Рисунок 2

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

```
C:\Users\wolfd>C:\Users\wolfd\Desktop\Учеба\ООП\Конспекты\args\x64\Debug\args.exe open
open
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3

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

```
C:\Users\wolfd>C:\Users\wolfd\Desktop\Учеба\ООП\Конспекты\args\x64\Debug\args.exe "It work"
It work
Для продолжения нажмите любую клавишу . . .
```

Рисунок 4



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

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

Уменьшение размера исполняемого файла

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

При обычной компиляции размер файла будет



> Этот компьютер > Рабочий стол > Учеба > ООП > Конспекты > Тест размера			
Имя	Дата изменения	Тип	Размер
 hello_world.cpp	28.11.2022 22:20	C++ Source	1 КБ
 hello_world.exe	28.11.2022 22:29	Приложение	60 КБ

```
PS C:\Users\wolfd\Desktop\Учеба\ООП\Конспекты\Тест размера> g++ hello_world.cpp -o hello_world.exe
```

Чтобы уменьшить размер файла, необходимо дописать `-s`

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

```
PS C:\Users\wolfd\Desktop\Учеба\ООП\Конспекты\Тест размера> g++ hello_world.cpp -o hello_world.exe -s
```

> Этот компьютер > Рабочий стол > Учеба > ООП > Конспекты > Тест размера			
Имя	Дата изменения	Тип	Размер
 hello_world.cpp	28.11.2022 22:20	C++ Source	1 КБ
 hello_world.exe	28.11.2022 22:30	Приложение	3 КБ

Размер исполняемого файла успешно уменьшен.

Есть еще различные способы оптимизации

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

"-flto" межмодульные оптимизации

"-m32" 32 битный режим

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

"-funroll-loops" включает разворачивание циклов

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

"-Os" оптимизирует производительность и размер

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

///Добавить информацию про -s

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

Обработка исключительных ситуаций

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

- в системе закончилась оперативная память;
- соединение с сервером внезапно прервалось;
- пользователь выдернул флешку во время чтения или записи файла;

- понадобилось получить первый элемент списка, который оказался пустым;
- формат файла не такой, как ожидалось.

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

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

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

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

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

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

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

Обработчик ловит не все исключения, а только некоторые — те, что возникли в конкретной части определённой функции. Эту часть нужно явно обозначить, для чего используют конструкцию **try** (попробовать).

Также обработчик не поймает исключение, которое ранее попало в другой обработчик. После обработки исключения программа продолжает выполнение как ни в чём не бывало.

Начнем с примера:

```
1 void SomeFunction() {  
2     DoSomething0();  
3  
4     try {  
5         SomeClass var;  
6  
7         DoSomething1();  
8         DoSomething2();  
9  
10        // ещё код  
11  
12        cout << "Если возникло исключение, то этот текст не будет напечатан" << std::endl;  
13    }  
14    catch(ExceptionType e) {  
15        std::cout << "Поймано исключение: " << e.what() << std::endl;  
16        // ещё код  
17    }  
18  
19    std::cout << "Это сообщение не будет выведено, если возникло исключение в DoSomething0 или "  
20    "непойманное исключение внутри блока try." << std::endl;  
21 }
```

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

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

Во всех случаях объект `var` будет корректно удалён.

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

Ловим исключения нескольких типов:

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


```

1 void SomeFunction() {
2     DoSomething0();
3
4     try {
5         DoSomething1();
6         DoSomething2();
7         // ещё код
8     }
9     catch(ExceptionType1 e) {
10        std::cout << "Some exception of type ExceptionType1: " << e.what() << std::endl;
11        // ещё код
12    }
13    catch(ExceptionType2 e) {
14        std::cout << "Some exception of type ExceptionType2: " << e.what() << std::endl;
15        // ещё код
16    }
17    // ещё код
18 }

```

Ловим все исключения:

```

1 void SomeFunction() {
2     DoSomething0();
3
4     try {
5         DoSomething1();
6         DoSomething2();
7         // ещё код
8     }
9     catch(...) {
10        std::cout << "An exception any type" << std::endl;
11        // ещё код
12    }
13    // ещё код
14 }

```

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

//// Добавить свой пример, а также разобраться в вопросе использования try...

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      setlocale(LC_ALL, "RUS");
7      int numerator, denominator;
8      double result = 0.0;
9
10     cout << "Введите числитель: ";
11     cin >> numerator;
12
13     cout << "Введите знаменатель: ";
14     cin >> denominator;
15
16     try {
17         if (denominator == 0) {
18             throw runtime_error("Попытка деления на ноль!");
19         }
20
21         result = static_cast<double>(numerator) / denominator;
22         cout << "Результат деления: " << result << endl;
23     }
24     catch (const exception& e) {
25         cerr << "Произошла ошибка: " << e.what() << endl;
26     }
27
28     return 0;
29 }
30

```

Источник: <https://tproger.ru/articles/iskljuchenija-v-cpp-tipy-sintaksis-i-obrabotka/#part1>

Typedef

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

данных для псевдонима, за которым следует имя для псевдонима. Например:

```
typedef double distance_t; // определяем distance_t как псевдоним для типа 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.

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

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

```
distance_t milesToDestination{ 3.4 }; // определяет переменную типа double
```

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

```
1  #include <iostream>
2
3  int main()
4  {
5      typedef double distance_t; //определяем distance_t как псевдоним для типа double
6
7      distance_t milesToDestination{ 3.4 }; // определяет переменную типа double
8
9      std::cout << milesToDestination << '\n'; // выводит значение double
10
11     return 0;
12 }
```

Этот код печатает:

```
3.4
```

В приведенной выше программе мы сначала определяем **typedef distance_t** как псевдоним для типа **double**.

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

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

///Добавить про using

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

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

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

Пример использования

```
1 //У нас есть структура, представляющая точку в 3-х мерном пространстве
2 struct Point3D {
3     double x, y, z;
4 };
5
6 //С помощью typedef или using мы можем создать псевдоним для этой структуры :
7 using ThreeDPoint = Point3D;
8
9 //Теперь мы можем использовать ThreeDPoint вместо Point3D :
10 ThreeDPoint myPoint;
11 myPoint.x = 1.0;
12 myPoint.y = 2.0;
13 myPoint.z = 3.0;
14
```

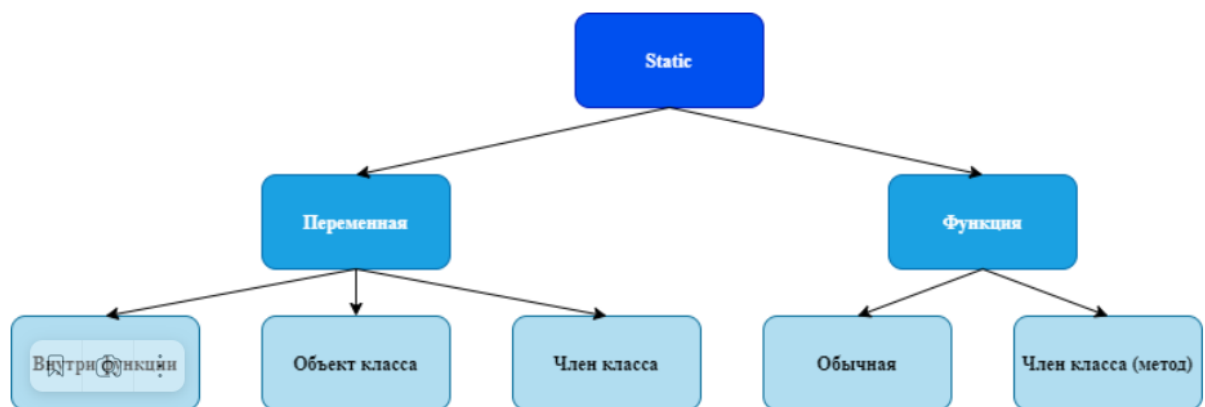
`typedef` и **`using`** - это мощные инструменты, которые позволяют улучшить читаемость и модульность кода в С++. Выбор между ними зависит от ваших предпочтений и стиля программирования. Однако в большинстве современных проектах рекомендуется использовать `using` из-за его более читаемого синтаксиса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

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

Static

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

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



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.

Вторая 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

```
#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()`.

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

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

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

Может быть использовано в генерации айдишников с помощью `static`

```
// 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.

Статическая локальная переменная – это глобальная переменная, доступная только в пределах функции. Время жизни – от первого вызова функции до конца программы.

Auto

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

Основные аспекты использования `auto`:

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

```
auto x = 42;           // x имеет тип int
auto name = "John";    // name имеет тип const char*
```

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

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

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

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (auto num : numbers) {
    // num имеет тип int
}
```


3. Авто и шаблоны:

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

```
1  template <typename T, typename U>
2  auto add(T a, U b) -> decltype(a + b) {
3      return a + b;
4  }
5
6  |
```

Ограничения auto

Хотя auto приносит множество преимуществ, есть и некоторые ограничения:

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

Заключение

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

Компилятор автоматически назначает определенный тип переменной на основе выражения. Этот тип становится фактическим типом переменной во время компиляции.

Include guard

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

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

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

2. Принцип работы Include Guard:

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

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

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

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

```
1  #ifndef MYHEADER_H
2  #define MYHEADER_H
3
4  // Содержимое заголовочного файла
5
6  #endif // MYHEADER_H
7
8  |
```

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

4. Подсказки по использованию Include Guard:

- Следуйте соглашениям по именованию макросов для Include Guard, чтобы избежать конфликтов.
- Убедитесь, что макрос Include Guard находится в самом начале заголовочного файла, перед всеми другими директивами `#include`.
- Помните, что Include Guard не является частью стандарта C++, но является действенным соглашением в сообществе разработчиков.

5. Заключение:

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

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

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

lvalue и rvalue в C и C++

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 сообщении 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.

Переменные:

```
int b = 5; int temp; temp = b;
```

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

```
int a = 5; (double&)a = 23.23;
```

Раскрытый указатель / адрес:

```
int *p = new int(23); *p = 5;
```

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

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;
}
```

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.

Результат применения оператора &.

```

int a = 23;

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

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

```

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;

}

```

Результат вызова функции, тип возвращаемого значения которой не является ссылкой.

```

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;

}

```

Этапы компиляции

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

```
#include <iostream>
using namespace std;
#define RETURN return 0

int main() {
    cout << "Hello, world!" << endl;
    RETURN;
}
```

1) Преппроцессинг

Самая первая стадия компиляции программы.

Преппроцессор — это *макро процессор*, который преобразовывает вашу программу для дальнейшего компилирования. На данной стадии происходит работа с преппроцессорными директивами. Например, преппроцессор 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 преппроцессинга и включаются в выпускаемый файл. Однако, каждый хэдер может быть открыт во время преппроцессинга несколько раз, поэтому, обычно, используются специальные преппроцессорные директивы, предохраняющие от циклической зависимости.

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

```
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;  
}
```

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
```

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.

3) 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!
```

Рефакторинг. Средства рефакторинга в C++

Рефакторинг вышел в релизной версии Visual Studio 2015 Preview, в котором были представлены новые возможности увеличения продуктивности разработки кода на C++.

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

Переименование (Rename)

Извлечение функции (Extract Function)

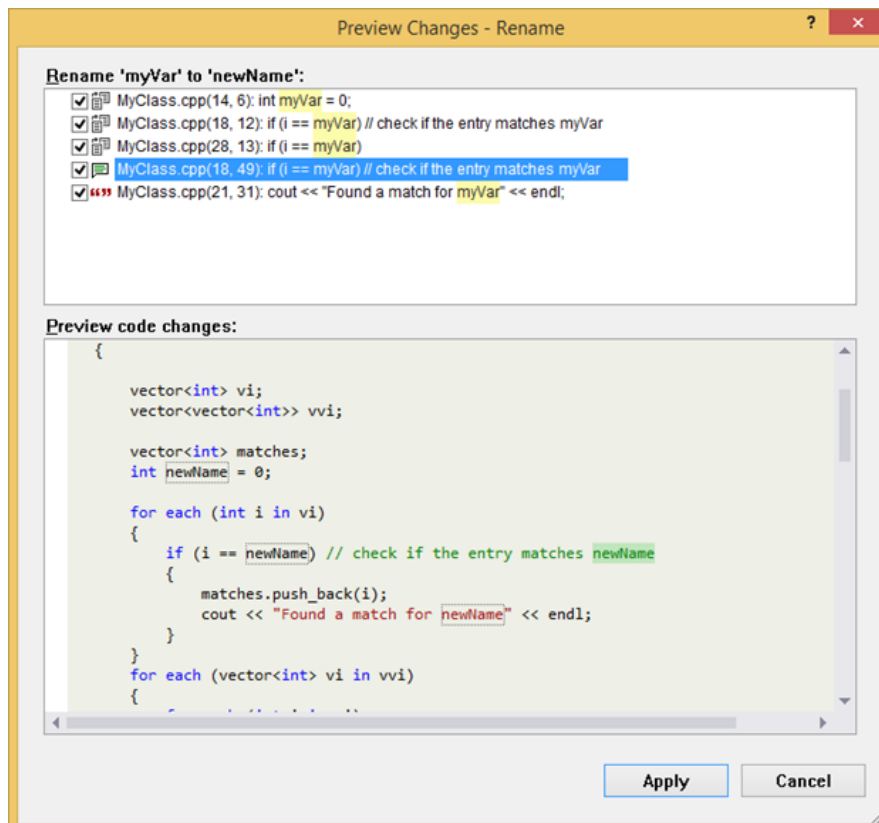
Генерация заглушек чисто виртуальных методов (Implement Pure Virtuals)

Генерация объявлений/заглушек методов (Create Declaration/Definition)

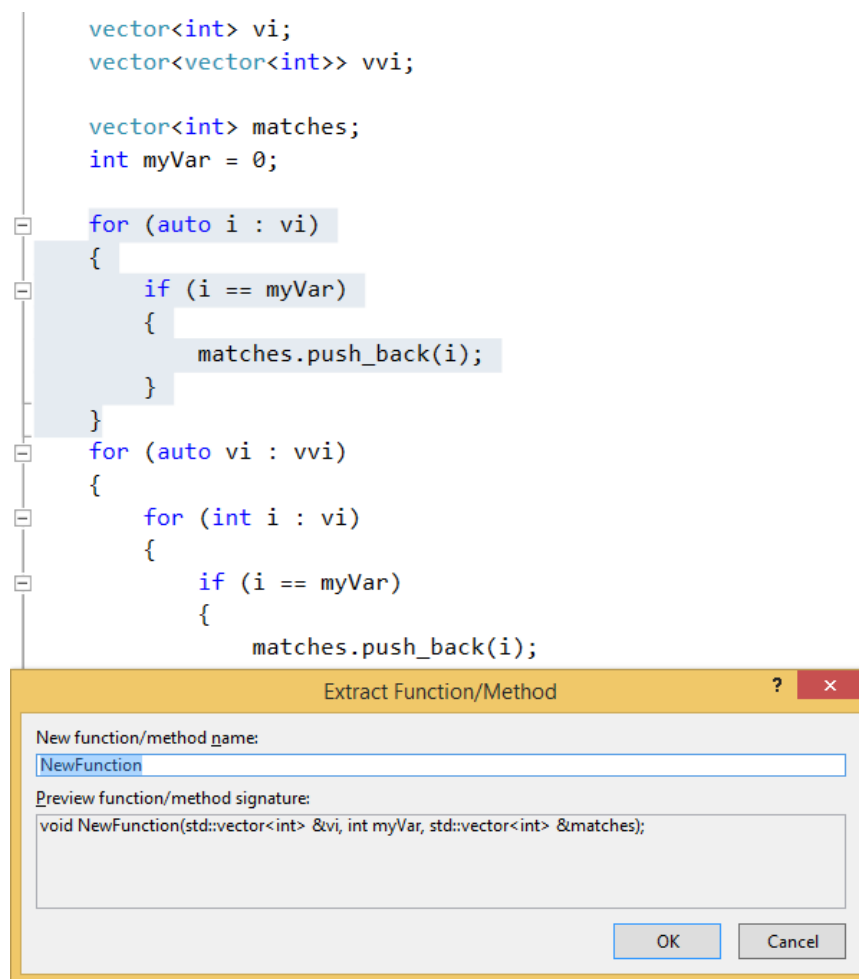
Перемещение объявлений функций (Move Function Definition)

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.



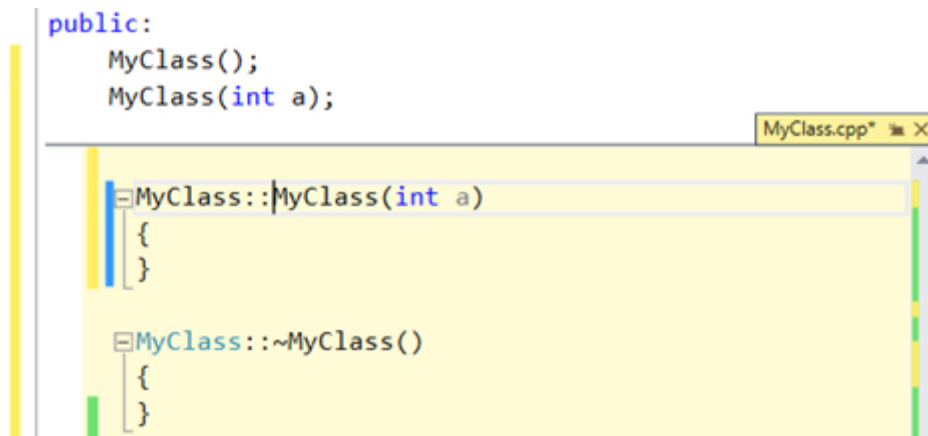
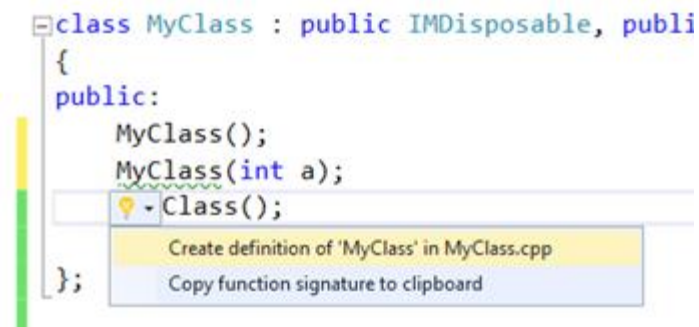
Извлечение функции После его установки выделите блок кода, который хотите выделить в отдельную функцию, затем правый клик и в меню «Refactor...» выберите «Extract Function/Method».



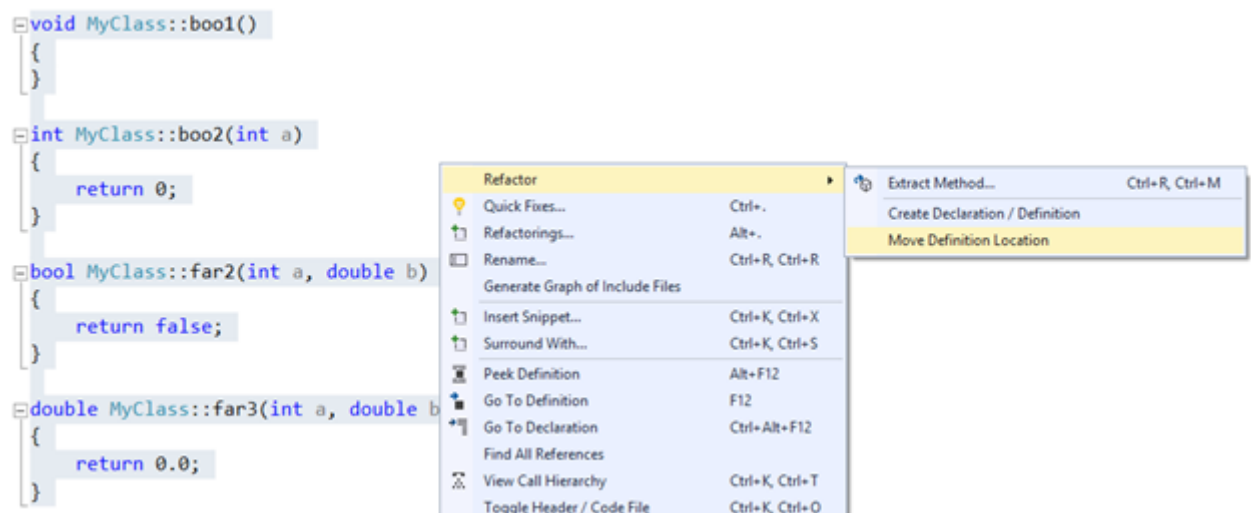
Генерация заглушек виртуальных методов позволяет создать тела всех чисто виртуальных методов в наследуемом классе. Поддерживается множественное наследование. Инструмент вызывается из контекстного меню объявления класса.



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

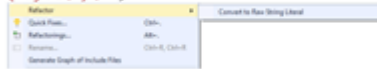


Перемещение объявлений методов позволяет быстро переместить тело метода из заголовочного файла в сpp-файл или наоборот.



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.

```
std::string s = "\nSome reasons this string is hard
to read:\n\t1. It would go off the screen in your
editor\n\t2. It has sooooo many escape
sequences.\n\nThey should make a type for this called
\"long\" string, har har har.\n\nHave fun parsing
this! (maybe?)\n";
```



```
std::string s = R"(
Some reasons this string is hard to read:
1. It would go off the screen in your editor
2. It has sooooo many escape sequences.
```

They should make a type for this called "long" string, har har har.

```
Have fun parsing this! (maybe?)
);
```

Файлы и папки проекта VisualStudio.

Папки:

.vs - Обычно, **.vs** папка требуется Visual Studio для хранения открытых документов, брейкпоинтов, и другой информации о состоянии решения. А значит содержит типичные файлы вроде,

- Временные кэши, используемые Roslyn для IntelliSense.
- Файл IIS Express applicationHost.config.

Файлы:

.sln - Файл решения. Используется для организации всех элементов проекта или нескольких проектов в единое решение.

.suo - Файл параметров решения . Сохраняет настройки решения, чтобы при любом открытии проекта или файла в решении оно выглядело и вело себя необходимым образом.

.vcxproj - Файл проекта . Хранит информацию, относящуюся к каждому проекту. (В более ранних версиях этот файл был назван Projname.vcproj или Projname.dsp.)

.vcxitems - Файл проекта общих элементов. Этот проект не создается. Вместо этого на него может сослаться другой проект C++, и его файлы станут частью процесса сборки ссылающегося проекта. Это можно использовать для совместного использования общего кода в кроссплатформенных проектах C++.

.sdf - Файл базы данных просмотра. Поддерживает возможности просмотра и навигации, такие как перейти к определению, найти все ссылки и Представление классов. Создается путем анализа файлов заголовков.

.vcxproj.filters - Файл фильтров. Указывает, куда поместить файл, который добавляется в решение. Например, H-файл помещается в узел Файлы заголовков.

.vcxproj.user - Файл пользователя миграции. После миграции проекта из Visual Studio 2008 в этом файле появляются данные, преобразованные из любых VSPROPS-файлов.

.idl - (Для конкретных проектов) Содержит исходный код на языке описания интерфейсов (IDL) для библиотеки типов элементов управления. Используется Visual C++ для создания библиотеки типов. Созданная библиотека предоставляет доступ к интерфейсу элемента управления другим клиентам автоматизации. Дополнительные сведения см. в разделе Файл определения интерфейса (IDL-файл) для пакета Windows SDK.

.txt - Создается мастером приложений и описывает файлы в проекте.

Установка точки останова и запуск отладчика

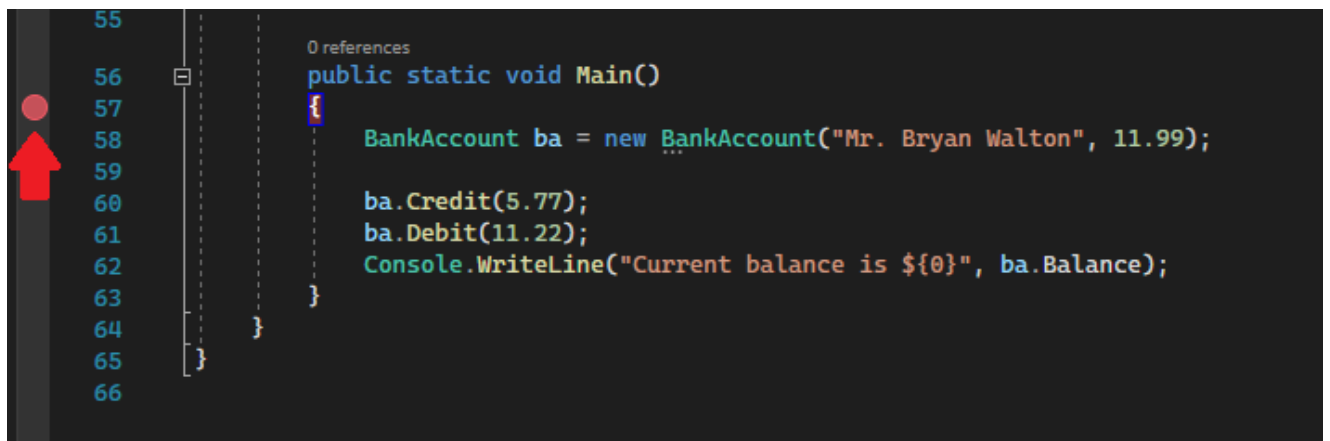
Точки останова полезны, если вам известны строка или раздел кода, которые вы хотите подробно изучить в среде выполнения.

Для отладки нужно запустить приложение с отладчиком, подключенным к процессу приложения. Для этого выполните следующие действия.

- Нажмите клавишу **F5** (**Отладка > Начать отладку**), которая является наиболее распространенным методом.

Точки останова — это один из самых простых и важных компонентов надежной отладки. Точка останова указывает, где Visual Studio следует приостановить выполнение кода, чтобы вы могли проверить значения переменных или поведение памяти либо выполнение ветви кода.

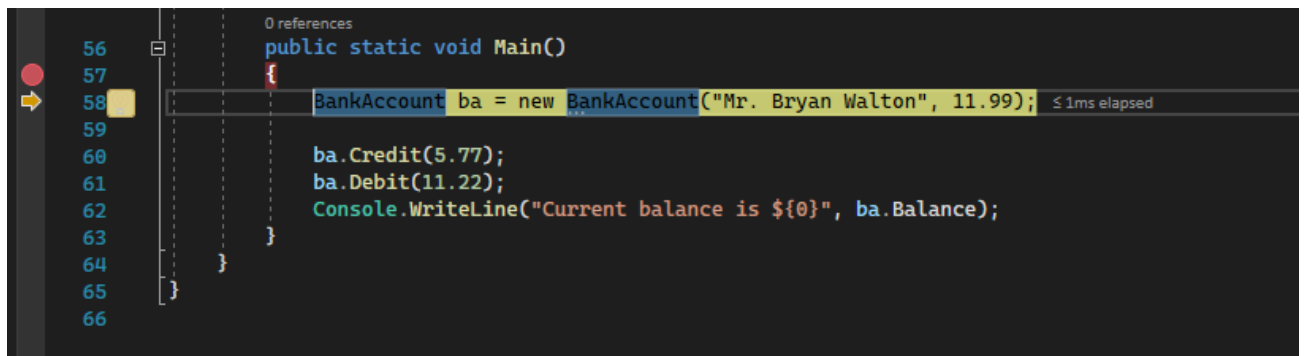
Если вы открыли файл в редакторе кода, точку останова можно задать, щелкнув в поле слева от строки кода.



Нажмите клавишу **F5** (**Отладка > Начать отладку**) или кнопку **Начать отладку** на панели инструментов отладки. При этом отладчик выполняется до первой встреченной точки останова. Если приложение еще не запущено, при нажатии клавиши **F5** запускается отладчик и выполняется остановка в первой точке останова.

Переход по коду в отладчике с помощью пошаговых команд

Для запуска приложения с подключенным отладчиком нажмите клавишу **F11** (**Отладка > Шаг с заходом**). **F11** — это команда **Шаг с заходом**, которая выполняет приложение с переходом к следующему оператору. При запуске приложения с помощью клавиши **F11** отладчик останавливается на первом выполняемом операторе.



Желтая стрелка представляет оператор, на котором приостановлен отладчик. В этой же точке приостанавливается выполнение приложения (этот оператор пока не выполнен).

Клавишу F11 удобно использовать для более детальной проверки потока выполнения. По умолчанию отладчик пропускает непользовательский код

Примечание

В управляемом коде вы увидите диалоговое окно с запросом о том, хотите ли вы получать уведомления при автоматическом обходе свойств и операторов (поведение по умолчанию). Если вы хотите изменить этот параметр позже, отключите параметр **Шаг с обходом свойств и операторов** в меню **Инструменты > Параметры** в разделе **Отладка**.


Шаг с обходом по коду для пропуска функций

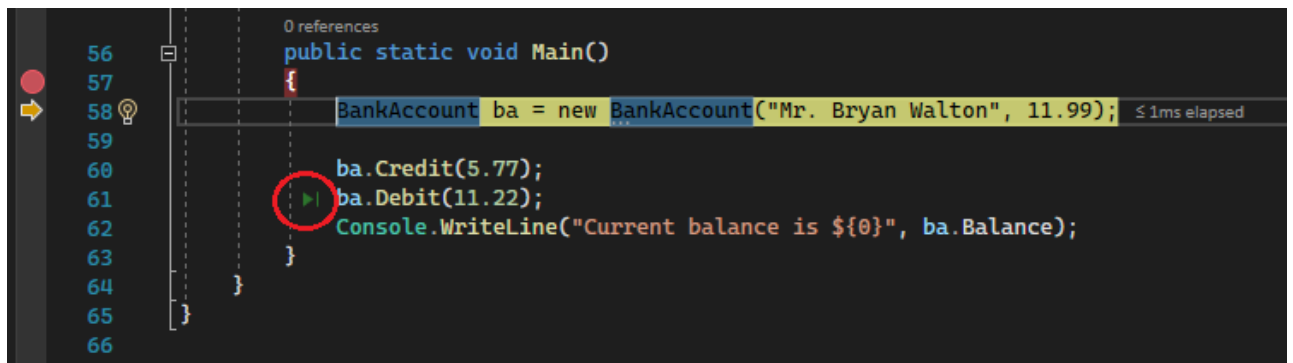
Когда вы находитесь в строке кода, представляющей собой вызов функции или метода, можно нажать клавишу **F10 (Отладка > Шаг с обходом)** вместо F11.

Клавиша F10 продолжает выполнение отладчика без захода в функции или методы в коде приложения (код продолжает выполняться). Нажав клавишу F10, вы можете обойти код, который вас не интересует. Так можно быстро перейти к важному для вас коду.

Быстрое выполнение до точки в коде с помощью мыши

Использование кнопки **Выполнение до щелкнутого** аналогично установке временной точки останова. Кроме того, эта команда удобна для быстрой работы в видимой области кода приложения. **Выполнение до щелкнутого** можно использовать в любом открытом файле.

Находясь в отладчике, наведите курсор на строку кода, пока слева не появится кнопка **выполнения до щелкнутого** (Выполнить до этого места) 



```
0 references
public static void Main()
{
    BankAccount ba = new BankAccount("Mr. Bryan Walton", 11.99);
    ba.Credit(5.77);
    ba.Debit(11.22);
    Console.WriteLine("Current balance is ${0}", ba.Balance);
}
```

Примечание

Кнопка **Выполнить о щелчка** (Выполнить до этого места) доступна начиная с Visual Studio 2017.

Нажмите кнопку **выполнения до щелкнутого** (Выполнить до этого места). Отладчик продолжает выполнение до строки кода, которую вы щелкнули.

Вывод отладчика из текущей функции

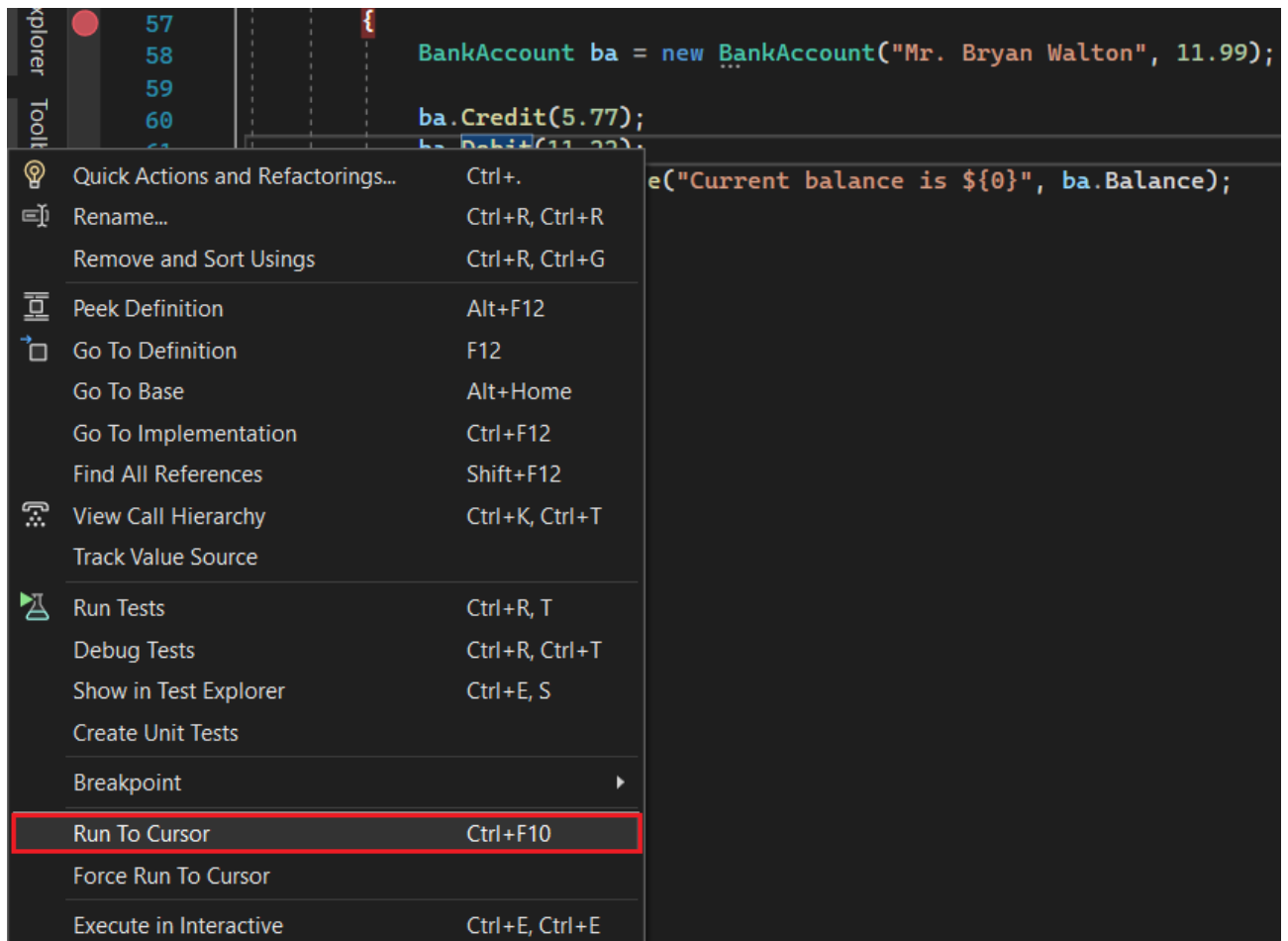
В некоторых случаях может потребоваться продолжить сеанс отладки, однако полностью проведя отладчик сквозь текущую функцию.

Нажмите сочетание клавиш **SHIFT + F11** (или выберите **Отладка > Шаг с выходом**).

Эта команда возобновляет выполнение приложения (и перемещает отладчик) до возврата текущей функции.

Выполнить до текущей позиции

Если вы находитесь в режиме редактирования кода (то есть работа отладчика не приостановлена), щелкните правой кнопкой мыши строку кода в приложении и выберите команду **Выполнить до текущей позиции** (или нажмите клавиши **CTRL+F10**). Эта команда запускает отладку и задает временную точку останова на текущей строке кода.




Если имеются заданные точки останова, отладчик приостанавливается в первой достигнутой точке останова.

Нажимайте клавишу **F5**, пока не достигнете строки кода, для которой выбрали **Выполнить до текущей позиции**.


Эта команда удобна, когда вы редактируете код и хотите быстро задать временную точку останова и одновременно запустить отладчик.

Вы можете использовать функцию **Выполнить до текущей позиции** в окне **Стек вызовов** во время отладки.

Быстрый перезапуск приложения

Нажмите кнопку **Перезапустить**  на панели инструментов отладки (или нажмите сочетание клавиш **CTRL+SHIFT+F5**).

Кнопка **Перезапустить** позволяет сэкономить время, затрачиваемое на остановку приложения и перезапуск отладчика. Отладчик приостанавливается в первой точке останова, достигнутой при выполнении кода.

Если вы хотите остановить отладчик и вернуться в редактор кода, можно нажать красную кнопку останова  вместо кнопки **Перезапустить**.

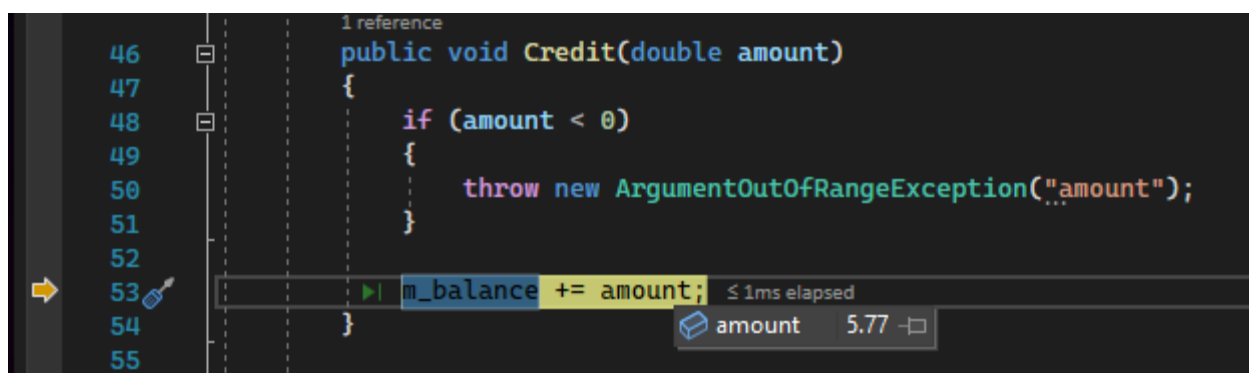
Редактирование кода в реальном времени

Visual Studio 2022 поддерживает динамическое редактирование кода в процессе отладки.

Проверка переменных с помощью подсказок по данным

Теперь, когда вы немного освоились, у вас есть хорошая возможность проверить состояние приложения (переменные) с помощью отладчика. Функции, позволяющие проверять переменные, являются одними из самых полезных в отладчике. Реализовывать эту задачу можно разными способами. Часто при попытке выполнить отладку проблемы пользователь старается выяснить, хранятся ли в переменных значения, которые требуются в определенном состоянии приложения.

В режиме приостановки в отладчике наведите указатель мыши на объект, чтобы увидеть его текущее значение или значение по умолчанию.



Если переменная имеет свойства, объект можно развернуть, чтобы увидеть все его свойства.

Часто при отладке бывает необходимо быстро проверить значения свойств для объектов. Лучше всего для этого подходят подсказки по данным.

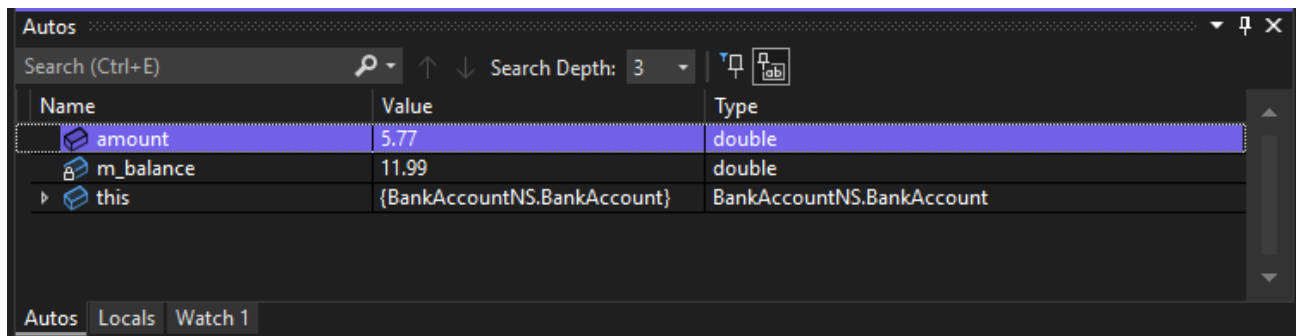
Совет

В большинстве поддерживаемых языков можно изменять код во время сеанса отладки.

Проверка переменных с помощью окон "Видимые" и "Локальные"

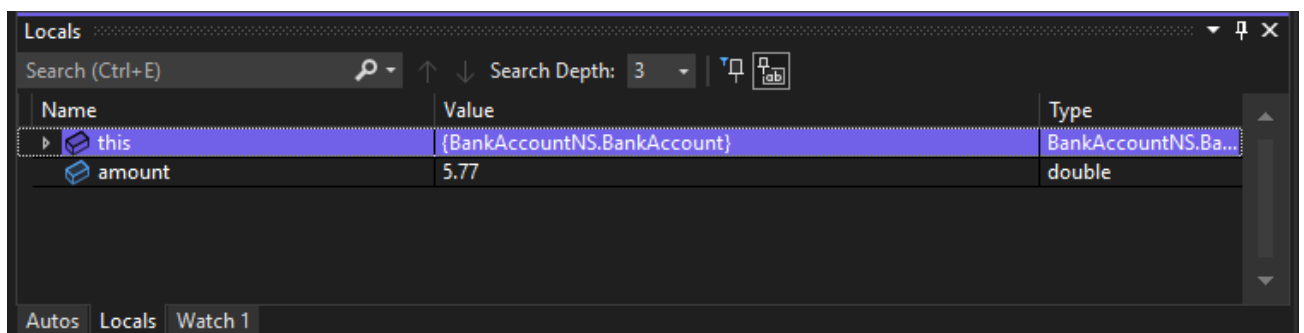
В окне **Видимые** отображаются переменные вместе с текущим значением и типом. Окно **Видимые** показывает все переменные, используемые в текущей или предыдущей строке (в C++ это окно показывает переменные в трех предыдущих строках кода; сведения о поведении для конкретного языка см. в документации).

Во время отладки взгляните на окно **Видимые** в нижней части редактора кода.



Примечание

Взгляните в окно **Локальные**. В окне **Локальные** показаны переменные, которые находятся в текущей области.

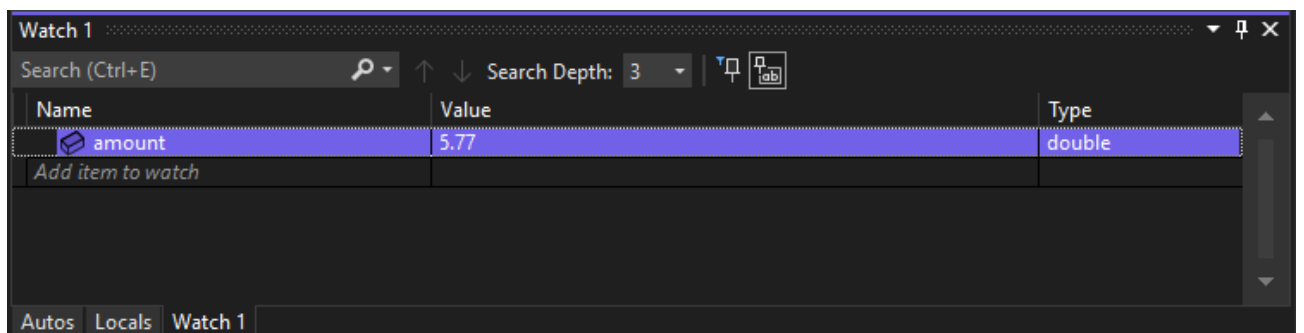


В этом примере объекты `this` и `f` находятся в области действия.

Установка контрольного значения

В окне **Контрольное значение** можно указать переменную (или выражение), которую необходимо отслеживать.

Во время отладки щелкните правой кнопкой мыши и выберите пункт **Добавить контрольное значение**.



В этом примере у вас есть контрольное значение, заданное для объекта, и по мере перемещения по отладчику вы можете наблюдать за изменением его значения. В отличие от других окон переменных, в окне **Контрольное значение** всегда отображаются просматриваемые вами переменные (они выделяются серым цветом, когда находятся вне области действия).

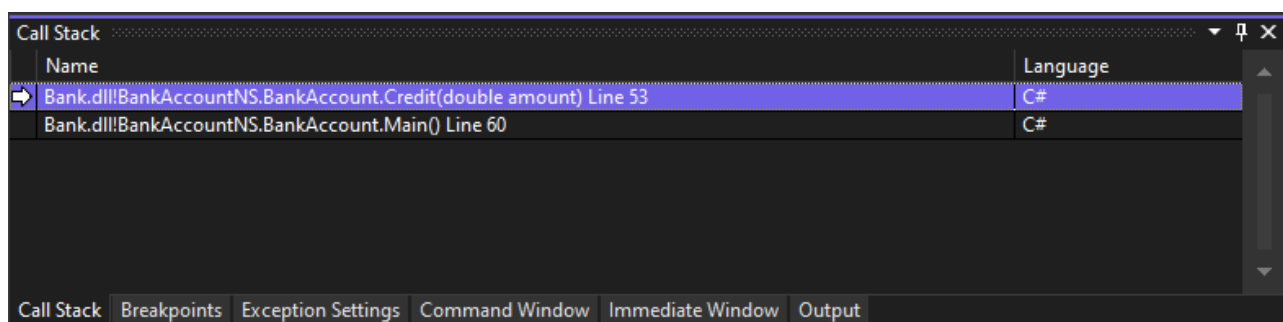
Просмотр стека вызовов

В окне **Стек вызовов** показан порядок вызова методов и функций. В верхней строке показана текущая функция. Во второй строке показана функция или свойство, из которого она вызывалась, и т. д. Стек вызовов хорошо подходит для изучения и анализа потока выполнения приложения.

Примечание

Окно **Стек вызовов** аналогично перспективе "Отладка" в некоторых интегрированных средах разработки, например Eclipse.

Во время отладки щелкните окно **Стек вызовов**, которое по умолчанию открыто в нижней правой области.

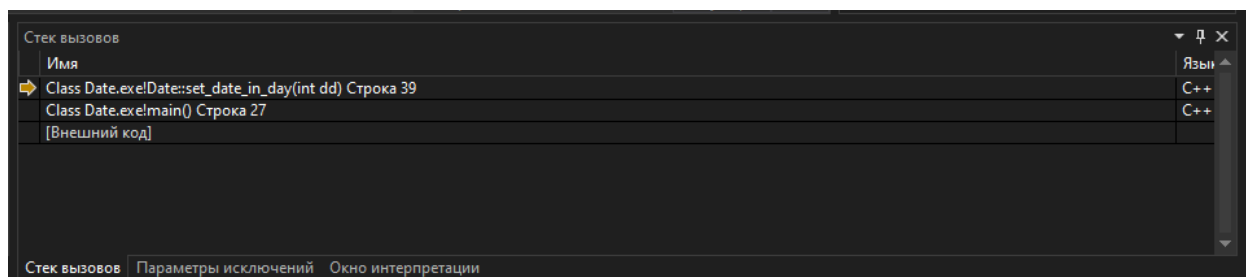


Дважды щелкните строку кода, чтобы просмотреть исходный код. При этом также изменится текущая область, проверяемая отладчиком. Это не перемещает отладчик.

Для выполнения других задач можно воспользоваться контекстными меню из окна **Стек вызовов**. Например, можно вставлять точки останова в указанные функции, перезапускать приложение с помощью функции **Выполнение до текущей позиции** и изучать исходный код.

Стек вызовов (call stack) — стек, хранящий информацию для возврата управления из подпрограмм (функций) в программу или подпрограмму (при вложенных или рекурсивных вызовах).

При вызове подпрограммы в стек заносится адрес возврата — адрес в памяти следующей инструкции приостанавливаемой программы, а управление передается подпрограмме. При последующем вложенном или рекурсивном вызове в стек заносится очередной адрес возврата и так далее.



Источники

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

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

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

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

<https://prog-cpp.ru/cpp-newdelete/>

<https://habr.com/ru/articles/527044/>

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

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

<https://learn.microsoft.com/ru-ru/cpp/build/reference/project-and-solution-files?view=msvc-160>

<https://learn.microsoft.com/ru-ru/cpp/build/reference/file-types-created-for-visual-cpp-projects?view=msvc-160&viewFallbackFrom=vs-2019>

<https://learn.microsoft.com/ru-ru/visualstudio/debugger/debugger-feature-tour?view=vs-2022>

