

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Урок № 5

Вложенные циклы

Содержание

1. Вложенная конструкция.....	3
2. Практические примеры	7
Пример 1	7
Пример 2	9
Пример 3	11
3. Использование интегрированного отладчика Microsoft Visual Studio.....	16
Понятие отладки.	
Необходимость использования отладчика	16
Выполнение программы по шагам	17
Точка останова.....	20
«Умная» точка останова	22
4. Домашнее задание	26

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе [Adobe Acrobat Reader](#).

1. Вложенная конструкция

В прошлых уроках вы познакомились с конструкцией под названием цикл и вариантами реализации цикла в языке C++. Как вы уже успели заметить, цикл является одной из основополагающих конструкций программирования. С его помощью решается огромное количество задач. Также, вы уже столкнулись с тем, что в цикл можно вкладывать конструкции логического выбора, такие, как **if** и **switch**. Однако, не будем останавливаться на достигнутом и, попробуем вложить в цикл подобный ему оператор, т.е. — другой цикл. Такие конструкции называются вложенными циклами. По своей структуре такие циклы напоминают коробки разного размера. Их можно вкладывать друг в друга, то есть в коробку большего размера можно поместить коробку поменьше. В конце концов, чтобы посмотреть внутреннюю коробку придется сначала открыть все внешние.

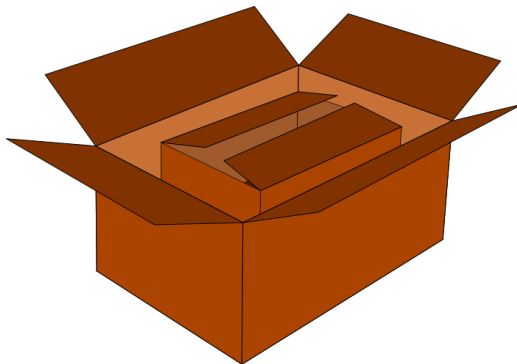


Рисунок 1

Именно по такому принципу и работают вложенные циклы. На первой итерации внешнего цикла запускается внутренний и он выполняется до своего полного завершения. Только после этого управление снова передается на внешний цикл. На второй же итерации опять запускается внутренний цикл. И так до тех пор, пока полностью не завершится внешний цикл. Давайте теперь попробуем разобраться в этом на примере. Необходимо создать программу, которая будет выводить дом на экран. Так что для этого нужно?

Во-первых, дом состоит из кирпичей. Для вывода одного кирпича мы будем использовать оператор `cout << "|###|";` Правда, такой дом будет стоять не на одном кирпиче, а на нескольких. Поэтому необходим цикл, который выведет на экран первый ряд из десяти кирпичей. Для этого мы можем использовать уже привычную конструкцию цикла `for`:

```
for (int j = 0; j < 10; j++)
{
    cout << "|###|";
}
cout << endl;
```

Первый ряд уже готов.



Рисунок 2

Во-вторых, как видим, даже одного ряда кирпичей недостаточно. Допустим их нужно семь. Поэтому вывод

такого одного ряда необходимо повторить семь раз. Для этого написанную нами конструкцию цикла нужно вложить еще в одну.

```
for (int i = 0; i < 7; i++)
{
    for (int j = 0; j < 10; j++)
    {
        cout << "#### ";
    }
    cout << endl;
}
cout << endl;
```

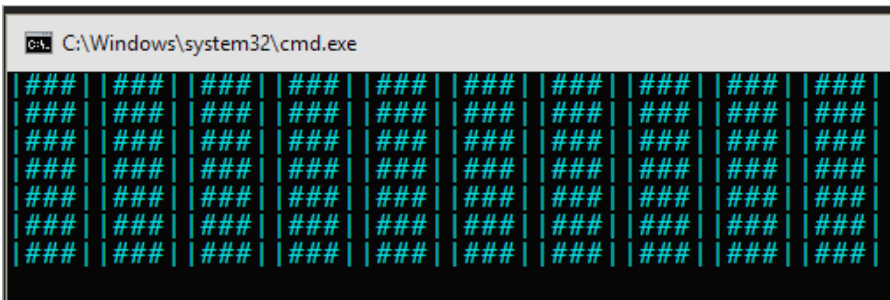


Рисунок 3

PS. Немного далее, мы рассмотрим пример построения крыши для этого дома.

Принцип работы программы, реализующей вложенный цикл, основан на том, что внутренний цикл срабатывает каждый раз на всех итерациях внешнего цикла от начала до конца. Другими словами, пока программа не выйдет из вложенного цикла (пока не закрыта внутренняя коробка) — выполнение внешнего не продолжится.

Ниже изображена схема работы вложенных циклов.

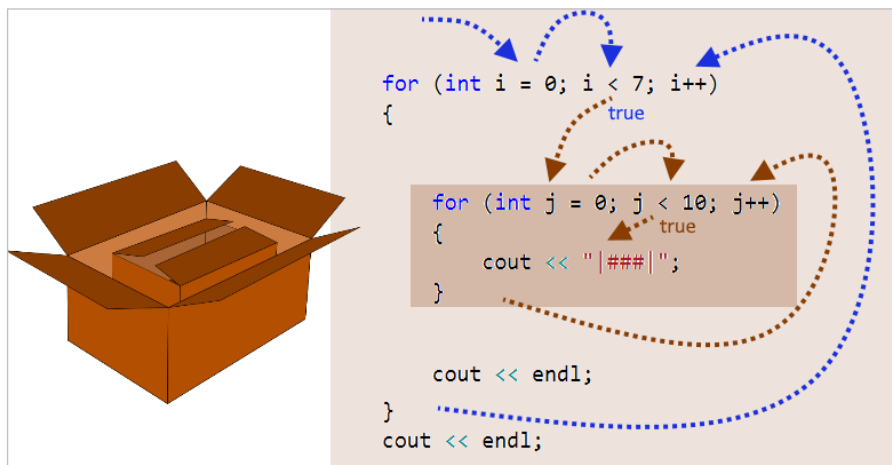


Рисунок 4

Как видите, все просто. Несмотря на это, вложенные конструкции значительно упрощают реализацию большинства сложных алгоритмов. Убедитесь в этом, рассматривая следующий раздел урока, в котором мы подготовили для вас несколько примеров.

2. Практические примеры

Пример 1

Постановка задачи

Написать программу, которая выводит на экран таблицу умножения.

Код реализации:

```
#include <iostream>
using namespace std;

int main()
{
    for (int i = 1; i<10; i++)
    {
        for (int j = 0; j<10; j++)
        {
            cout << i*j << "\t";
        }

        cout << "\n\n";
    }
    return 0;
}
```

Комментарий к коду

1. Управляющие переменные внешнего и внутреннего циклов осуществляют функции множителей.
2. Управляющая переменная **i** создается и инициализируется значением **1**.

3. Программа проверяет условие $i < 10$, так как 1 меньше 10 условие является истинным и программа входит во внешний цикл.
4. Управляющая переменная j создается и инициализируется значением 1 .
5. Программа проверяет условие $j < 10$, так как 1 меньше 10 условие является истинным и программа входит во внутренний цикл.
6. Осуществляется показ на экран произведения i на $j - 1$.
7. Осуществляется изменение управляющей переменной j .
8. Снова проверяется условие $j < 10$, так как 2 меньше 10 условие является истинным и программа снова входит во внутренний цикл.
9. Осуществляется показ на экран произведения i на $j - 2$
10. Осуществляется изменение управляющей переменной j .

Действия с 5 по 7 повторяются до тех пор, пока j не становится равно 10 , при этом текущее значение $i(1)$ умножается на каждое значение j (от 1 до 9 включительно), результат показывается на экран. Получается строка таблицы умножения на 1 .

Затем программа выходит из внутреннего цикла и переводит экранный курсор на две строки вниз. После этого, осуществляется увеличение переменной i на единицу и снова вход во внутренний цикл. Теперь уже для вывода цепочки умножения на 2 .

Таким образом, в конце концов на экране появляется вся таблица умножения.

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Для продолжения нажмите любую клавишу . . .

Рисунок 5

Пример 2

Постановка задачи

Вывести на экран прямоугольник из символов **20** на **20**.

Код реализации

```
#include <iostream>
using namespace std;
int main()
{
    int str;
    int star_count;
    int length = 20;
    str = 1;
    while (str <= length)
    {
        star_count = 1;
        while (star_count <= length)
        {
            cout << " * ";
            star_count++;
        }
    }
}
```

```
        cout << "\n";
        str++;
    }
    cout << "\n";
    return 0;
}
```

Комментарий к коду

1. Управляющая переменная внешнего цикла — **str** контролирует количество строк в прямоугольнике.
2. Управляющая переменная внутреннего цикла — **star_count** контролирует количество символов в каждой строке.
3. **length** — длина стороны прямоугольника
4. После отрисовки каждой строки, внешний цикл осуществляет переход на следующую строчку прямоугольника.
5. Результат таков:

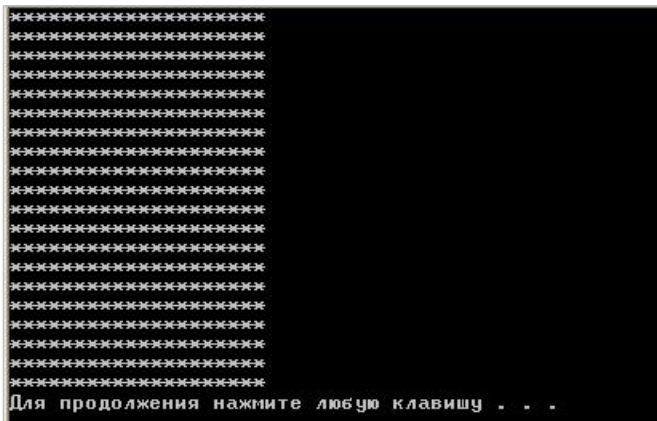


Рисунок 6

Примечание. Обратите внимания, что несмотря на то, что количество строк соответствует количеству символов в строке — на экране не квадрат! Это связано с тем, что высота и ширина символа разные.

Вот и всё! Теперь у вас имеется полная информация о циклах, их разновидностях и принципах работы. Но, прежде чем выполнять домашнее задание, следует ознакомиться с еще одним разделом урока. Этот раздел поможет вам не только писать программы, но и анализировать их работу.

Пример 3

Одним из часто используемых примеров вложенных циклов являются таблицы. Когда необходимо проанализировать все строки и в каждой строке просмотреть ее ячейки. Внешний цикл будет перебирать строки от первой до последней. А внутренний — все ячейки каждой такой строки. Очень часто мы сталкиваемся с таблицами по работе, учебе. Их удобно использовать, когда необходимо систематизировать большой объем данных. Это могут быть турнирные таблицы, таблицы расстояний между городами и т.д.

В программировании такие таблицы разделяют на прямоугольные или квадратные.

	0	1	2	3
0	+	+	+	+
1	+	+	+	+
2	+	+	+	+

Прямоугольная таблица
(разное количество строк
и столбцов)

	0	1	2
0	№	№	№
1	№	№	№
2	№	№	№

Квадратная таблица
(одинаковое количество
строк и столбцов)

Исходный код:

```
for (int i = 0; i < 3; i++)
{
    for (int j = 0;
          j < 4; j++)
    {
        cout << " + ";
    }
    cout << endl;
}
cout << endl;
```

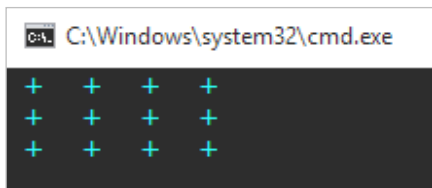


Рисунок 7

Исходный код:

```
for (int i = 0; i < 3; i++)
{
    for (int j = 0;
          j < 3; j++)
    {
        cout << " # ";
    }
    cout << endl;
}
cout << endl;
```

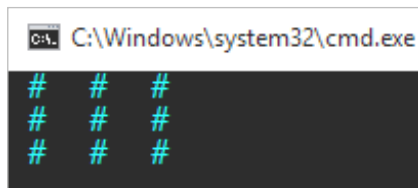


Рисунок 8

Более того, в квадратных таблицах выделяют еще два понятия: *главная и побочная диагональ*.

	0	1	2	3
0	+	+	+	+
1	+	+	+	+
2	+	+	+	+

$i == j$ — главная диагональ;
 $i > j$ — область под главной диагональю;
 $i < j$ — область над главной диагональю;

	0	1	2
0	№	№	№
1	№	№	№
2	№	№	№

N — количество строк или столбцов
 $i + j == N - 1$ — побочная диагональ;
 $i + j > N - 1$ — область под побочной диагональю;
 $i + j < N - 1$ — область над побочной диагональю;

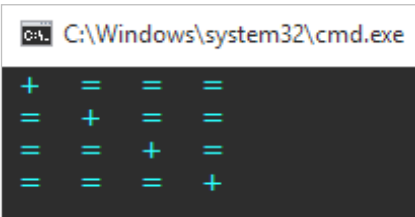
Исходный код:

```

int N = 4;
// N - размерность
// таблицы
for (int i = 0;
      i < N; i++)
{
    for (int j = 0;
          j < N; j++)

        //главная диагональ
        if (i == j)
            cout << " + ";
        else
            cout << " = ";
    }
    cout << endl;
}
cout << endl;

```



```

+  =  =  =
=  +  =  =
=  =  +  =
=  =  =  +

```

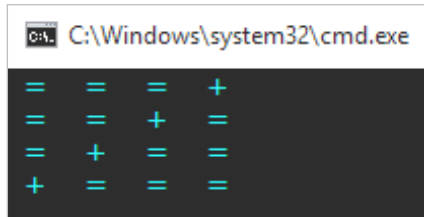
Рисунок 9*Исходный код:*

```

int N = 4;
// N - размерность
// таблицы
for (int i = 0;
      i < N; i++)
{
    for (int j = 0;
          j < N; j++)

        //побочная диагональ
        if (i + j == N-1)
            cout << " + ";
        else
            cout << " = ";
    }
    cout << endl;
}
cout << endl;

```



```

=  =  =  +
=  =  +  =
=  +  =  =
+  =  =  =

```

Рисунок 10

Используя эти знания, в будущем нам нетрудно будет составить, к примеру, таблицу расстояний между городами. На пересечении города с самим собой нам нужно будет выделить такую ячейку цветом, а мы уже знаем, что она лежит на главной диагонали.

Более этого теперь мы можем наконец-то достроить крышу дому, который строили в начале урока. Для начала вспомним, что 1 ряд дома состоял из 10 кирпичей:

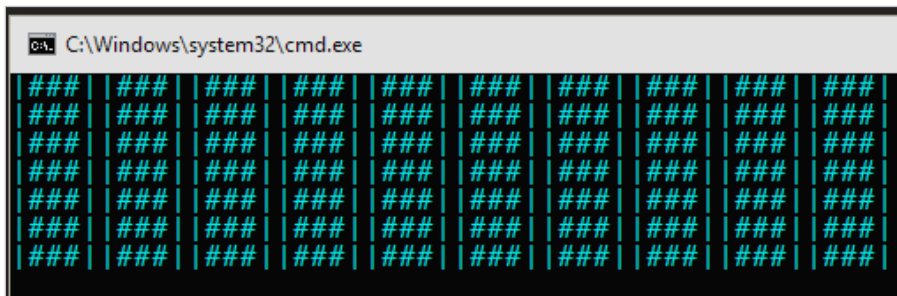


Рисунок 11

Поэтому ширина крыши должна состоять из того же количества. Теперь прорисуем таблицу 10x10 и закрасим ячейки с крышей.

				==	==				
			==	==	==	==			
		==	==	==	==	==	==		
	==	==	==	==	==	==	==	==	
==	==	==	==	==	==	==	==	==	==

Как видим, крыша находится под главной и побочной диагональю.

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
        //область под главной и
        // под побочной диагоналями
        if (i + j >= 9 && i >= j)
            cout << "|==|" ;
        else
            cout << "    ";
    }
    cout << endl;
}
```

В результате соединения этой части кода и предыдущей можем увидеть целиком готовый дом.

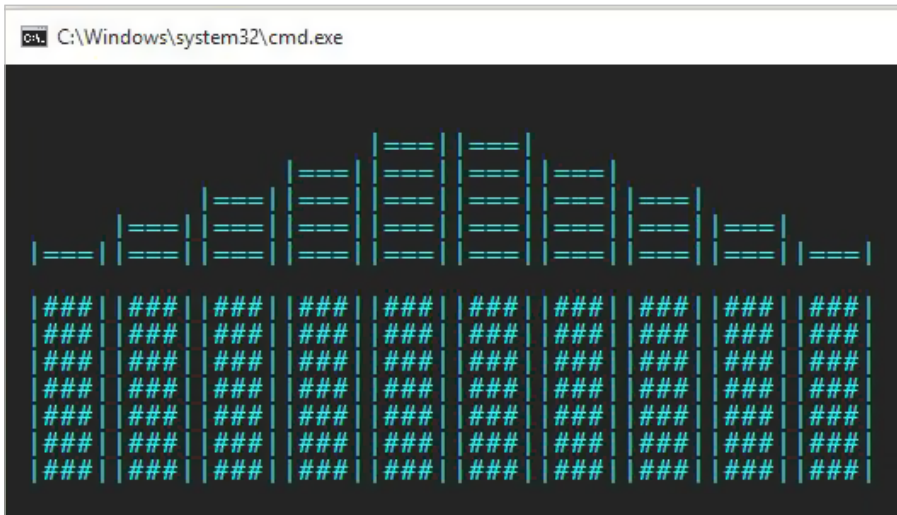


Рисунок 12

3. Использование интегрированного отладчика Microsoft Visual Studio

Понятие отладки.

Необходимость использования отладчика

Как вы уже знаете — существует два вида ошибок программы.

Ошибка на этапе компиляции — ошибка синтаксиса языка программирования. Такие ошибки или опечатки контролируются компилятором. Программа, содержащая такую ошибку просто не запустится на выполнение и компилятор укажет в какой строке кода произошла ошибка.

Ошибка на этапе выполнения — ошибка, приводящая к некорректной работе программы, либо к полной остановке последней. При этом следует учитывать, что такая ошибка компилятором не контролируется. Лишь в редких случаях компилятор может выдать предупреждение о какой-то некорректной инструкции, и конечно, в общем и целом в таких ситуациях программисту приходится выпутываться самому.

Именно об ошибках на этапе выполнения и пойдет речь. Зачастую, чтобы обнаружить подобную ошибку необходимо пройти некий фрагмент программы по шагам, так как если бы программа выполнялась. Безусловно, при этом желательно четко просчитать какое значение в определенный момент времени находится в конкретных

переменных. Можно конечно же произвести такой подсчет на листе бумаги построчно анализируя программу, однако в среде разработки Visual Studio есть специальное средство для организации анализа программы — отладчик. Данный раздел урока посвящен основам работы с отладчиком.

Выполнение программы по шагам

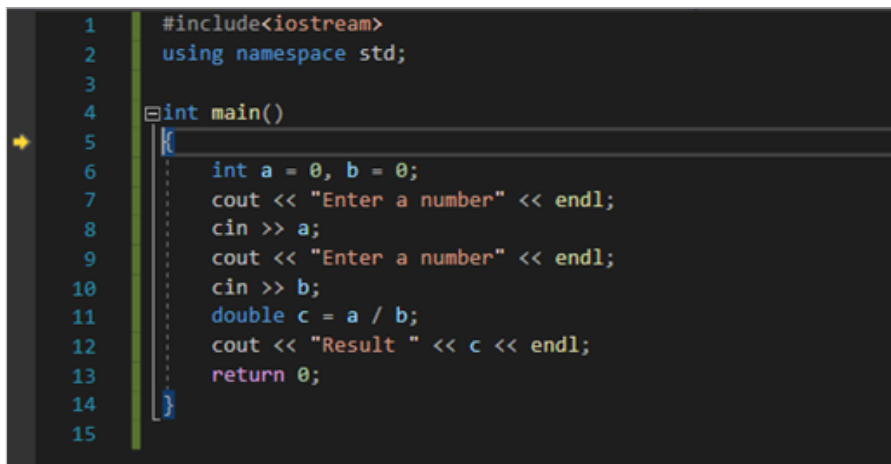
Предположим, что мы собираемся проанализировать следующий код:

```
#include<iostream>
using namespace std;

int main()
{
    int a = 0, b = 0;
    cout << "Enter a number" << endl;
    cin >> a;
    cout << "Enter a number" << endl;
    cin >> b;
    double c = a / b;
    cout << "Result " << c << endl;

    return 0;
}
```

Для начала создайте проект и наберите этот код. Скомпилируйте его и убедитесь, что нет синтаксических ошибок. Теперь приступим. Нажмите на клавиатуре функциональную клавишу **F10**. Рядом с первой выполняемой строкой кода у вас на экране появится желтая стрелка.



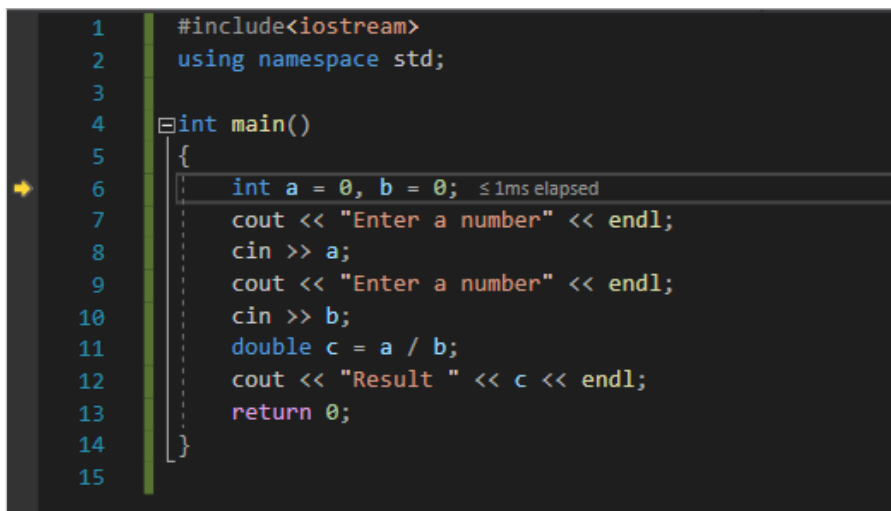
```

1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a = 0, b = 0;
7      cout << "Enter a number" << endl;
8      cin >> a;
9      cout << "Enter a number" << endl;
10     cin >> b;
11     double c = a / b;
12     cout << "Result " << c << endl;
13     return 0;
14 }
15

```

Рисунок 13

Именно эта стрелка указывает, какая строка кода сейчас «выполняется». Для передвижения на следующий шаг программы снова нажмите **F10**. И вы попадете в следующую строку:



```

1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a = 0, b = 0; ≤ 1ms elapsed
7      cout << "Enter a number" << endl;
8      cin >> a;
9      cout << "Enter a number" << endl;
10     cin >> b;
11     double c = a / b;
12     cout << "Result " << c << endl;
13     return 0;
14 }
15

```

Рисунок 14

Обратите внимание на то, что внизу экрана у вас располагается набор вкладок для анализа переменных:

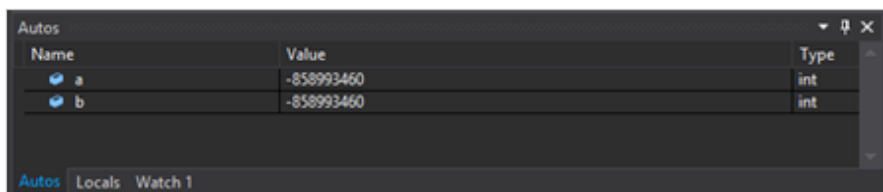


Рисунок 15

Autos — эта вкладка предназначена для просмотра значений переменных, которые существуют в момент выполнения текущей строки кода. Вписать на данной вкладке что-то от себя нельзя — это автоматическая функция. Если вкладка **Autos** скрыта, для её отображения нужно использовать комбинацию клавиш **Ctrl+D+A**.

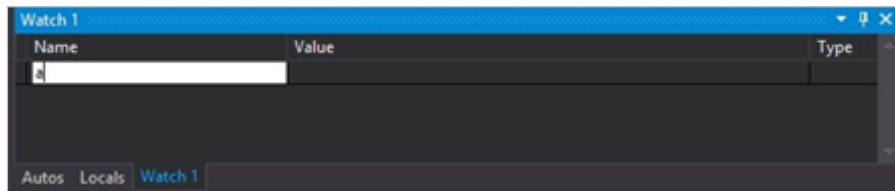


Рисунок 16

Watch — предназначена, как раз для тех случаев, когда необходимо самому выбрать переменную для просмотра. Вы просто вписываете в поле **Name** название переменной и она отображается независимо от выполняемого кода. Теперь просто нажимая **F10**, «пройдитесь» по коду и посмотрите, как будут изменяться данные во вкладках.

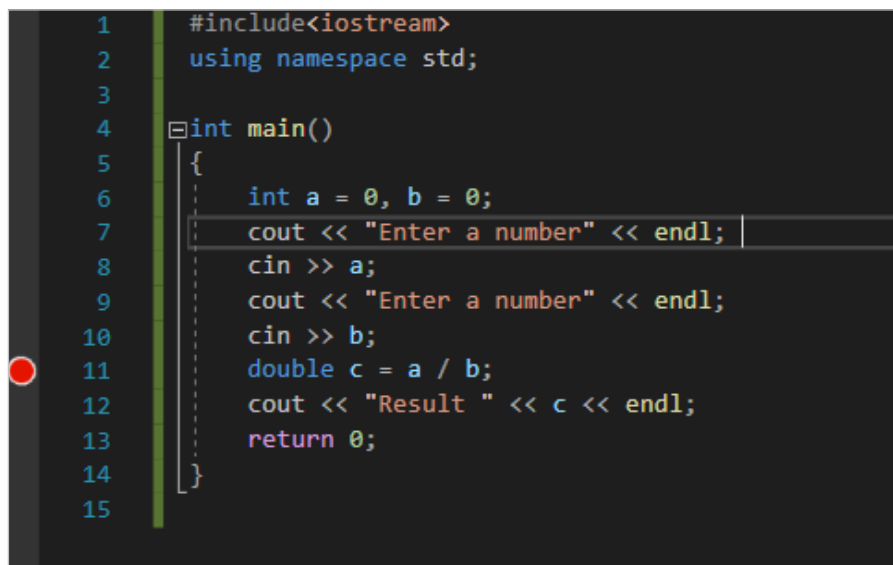
Примечание. Если вы хотите остановить отладчик раньше, чем завершиться анализ кода нажмите сочетание клавиш **Shift+F5**.

Примечание. Вы, наверное, заметили, что отладчик начинает анализ с первой строки программы. Если хотите запустить отладчик с определенной строки программы — установите курсор в необходимую строку и нажмите сочетание клавиш **Ctrl+F10**.

Точка останова

Рассмотрим ситуацию, когда нам необходимо вы полнить отрезок кода и остановившись в определенном месте, запустить отладчик. Для этого используется так называемая — точка останова.

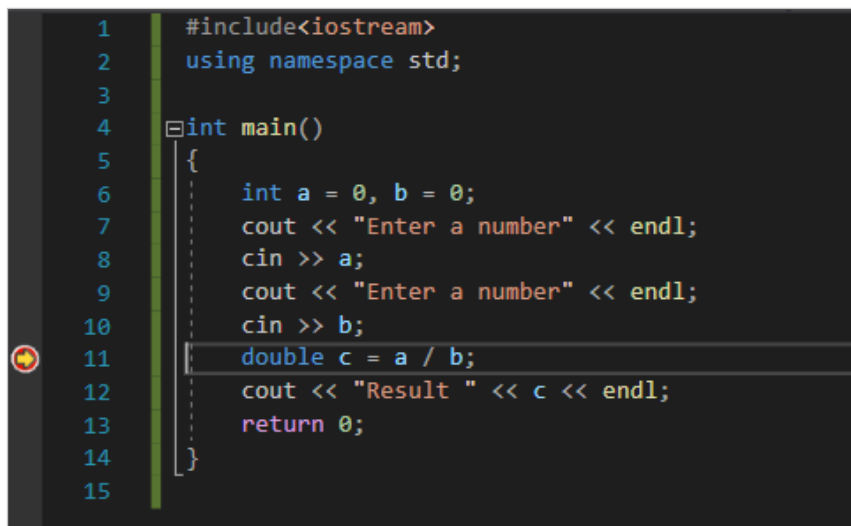
Установите курсор в строку **double c = a / b;** и нажмите клавишу **F9**. Рядом со строкой появилась красная точка это и есть точка останова.

A screenshot of a code editor with a dark background. The code is C++ and includes headers, namespace declarations, and a main function. A red circle, representing a breakpoint, is positioned to the left of line 11. The code is as follows:

```
1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a = 0, b = 0;
7      cout << "Enter a number" << endl;
8      cin >> a;
9      cout << "Enter a number" << endl;
10     cin >> b;
11     double c = a / b;
12     cout << "Result " << c << endl;
13     return 0;
14 }
15
```

Рисунок 17

Теперь нажмите **F5**, программа запустится, выполнится до того момента, где установлена точка останова и перейдет в режим отладчика.

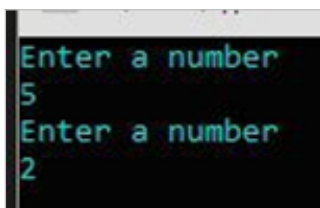


```
1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a = 0, b = 0;
7      cout << "Enter a number" << endl;
8      cin >> a;
9      cout << "Enter a number" << endl;
10     cin >> b;
11     double c = a / b;
12     cout << "Result " << c << endl;
13     return 0;
14 }
15
```

The image shows a Visual Studio code editor with a C++ program. A red circle with a yellow arrow (breakpoint) is placed on line 11, which is `double c = a / b;`. The code is as follows:

Рисунок 18

Обратите внимание на состояние консоли (окна программы). Здесь отображается все, что успело произойти:



```
Enter a number
5
Enter a number
2
```

The image shows the Visual Studio console window. It displays the output of the program: "Enter a number", followed by the input "5", then "Enter a number", followed by the input "2".

Рисунок 19

Далее следует обычная работа отладчика. Перемещайте желтую стрелку с помощью **F10** и следите, что происходит с переменными. Кроме того, заглядывайте в окно

консоли, все изменения происходящие в коде будут отображаться и там.

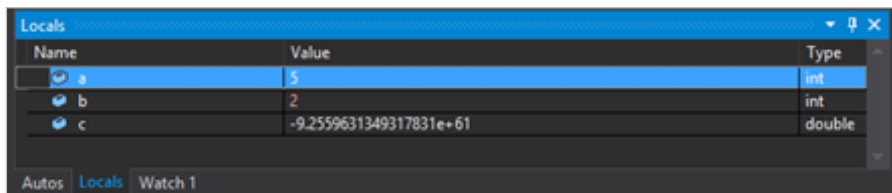


Рисунок 20

После выполнения следующего оператора можем убедиться в том, как изменяются переменные и проследить за промежуточным этапом выполнения программы с целью обнаружения ошибок.

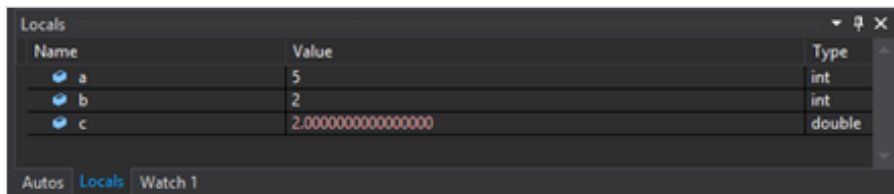


Рисунок 21

Как видим в этом примере были допущены логические ошибки с типами данных, т.к. при выполнении операции с однотипными данных результатом вычисления будет тот же тип данных (в нашем случае это **int** и только после присвоения станет **double**).

«Умная» точка останова

Только что мы запустили анализ программы с определенного места. Давайте рассмотрим теперь пример с циклом:

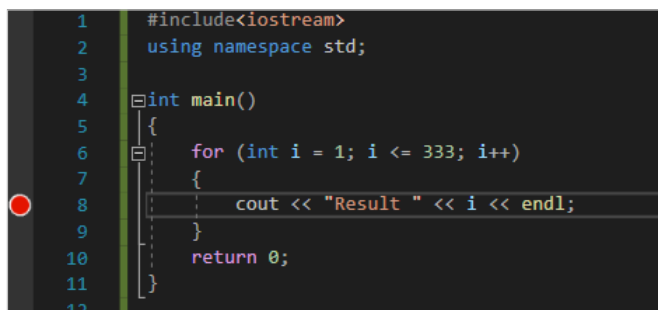


Рисунок 22

Заметим, что отладчик сработал сразу, как только началось выполнение тела цикла, т.е. на первой итерации. Это неудобно, в случае, если итераций большое количество, и несколько из них вам надо пропустить. Другими словами — вы хотите начать анализ, например, с 5-й итерации цикла. Решить проблему просто — сделать точку останова «умной».

Для этого на самой точке щелкаете правой кнопкой мыши и в открывшемся меню выбираете **Conditions...**

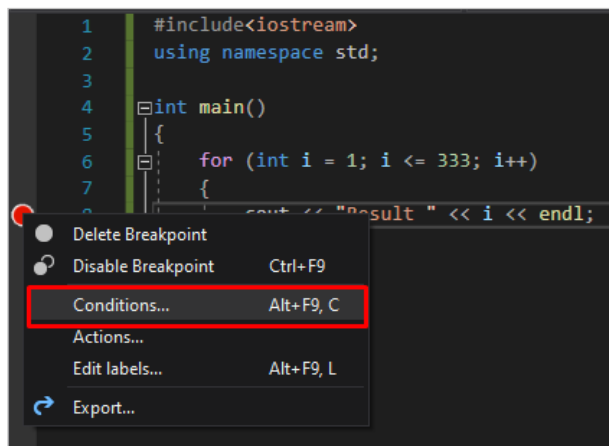


Рисунок 23

Перед вами появилось окошко **Breakpoint Settings...**
Нас интересует кнопка **Conditional Expression**.

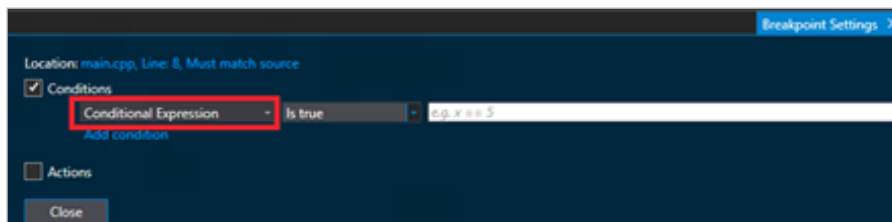


Рисунок 24

Также нам необходимо вписать условие при котором запустится отладчик. Наше условие **i==5**. Выбираем **is true** — то есть остановиться, когда условие станет истинным.

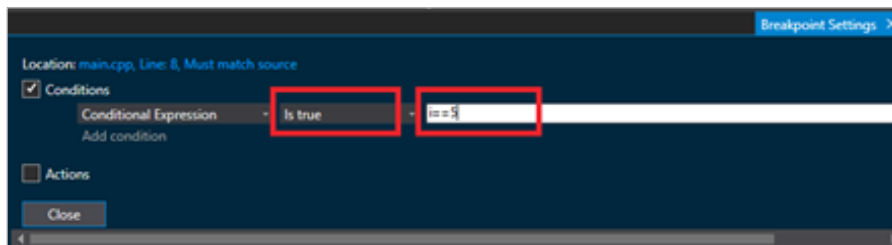


Рисунок 25

Нажимаем кнопку **Close**. Кроме этого, при наведении курсором мыши на **breakpoint** можем убедиться в его настройке:

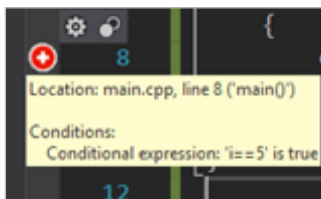
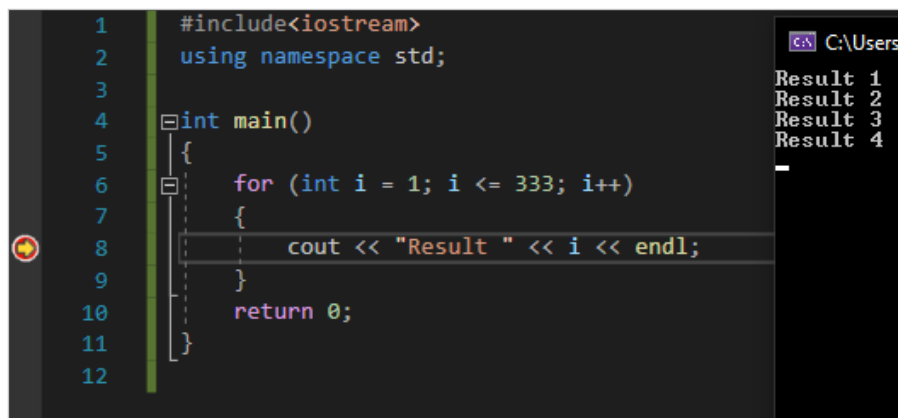


Рисунок 26

Когда все готово, нажимаем **F5** и смотрим, что произойдет. Как видите, все успешно — отладчик запустился в нужный нам момент.



```
1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6      for (int i = 1; i <= 333; i++)
7      {
8          cout << "Result " << i << endl;
9      }
10     return 0;
11 }
12
```

Output window: C:\Users\...
Result 1
Result 2
Result 3
Result 4

Рисунок 27

Примечание. Снять точку останова можно простым нажатием F9, в той строке, где точка располагается.

Сегодня мы рассказали не обо всех возможностях отладчика. Мы еще затронем эту тему в будущем при изучении функций. А, сейчас, настоятельно рекомендуем вам — потренироваться работе с отладчиком, например, на всех примерах урока и на своем домашнем задании. Удачи!

4. Домашнее задание

1. Написать программу, которая для чисел в диапазоне от А до В определяла количество их делителей. К примеру, А= 10, В = 15.

Делители для числа 10 — 1 2 5 10;

Делители для числа 11 — 1 11;

Делители для числа 12 — 1 2 3 4 6 12;

Делители для числа 13 — 1 13;

Делители для числа 14 — 1 2 7 14;

Делители для числа 15 — 1 3 5 15.

2. Создать программу, которая выводит на экран простые числа в диапазоне от 2 до 1000. (Число называется простым, если оно делится только на 1 и на само себя без остатка; причем число 1 простым не считается).
3. Написать программу, которая выводит на экран — следующую фигуру:

*			*			*
	*		*		*	
		*	*	*		
*	*	*	*	*	*	*
		*	*	*		
	*		*		*	
*			*			*

Ширина и высота фигуры запрашивается у пользователя как положительное нечетное число.

4. На чемодане стоит трехзначный код. Он состоит из цифр, которые не повторяются. Напишите программу, которая выведет все возможные такие комбинации цифр. А также определите сколько времени понадобится для открытия чемодана в худшем случае, если на один такой набор уходит 3 секунды.
5. В конце мая фирма формирует отчет по заработной плате 12 сотрудников за весенний квартал. Написать программу, которая будет запрашивать сумму заработной платы каждого сотрудника за Март, Апрель и Май. Необходимо определить:
 - выплату по каждому сотруднику за квартал;
 - общую выплату по всем сотрудникам за квартал.