

Введение в разработку программного обеспечения (ИС и ЦД)

Структура системы программирования. Программные конструкции. Назначение отладчика. Понятие и назначение дизассемблера

План лекции:

- программные конструкции (блоки, функции, процедуры);
- объявление функции, определение функции, передача параметров в функцию;
- способы передачи параметров;
- область видимости переменных;
- программные библиотеки;
- модель памяти C/C++ (классы памяти);
- среда разработки: понятие, назначение, основные возможности дизассемблера.

1. Программные конструкции (блоки, функции, процедуры, модули)

Инструкция (или оператор (англ. statement) – это одна команда языка программирования; наименьшая законченная часть.

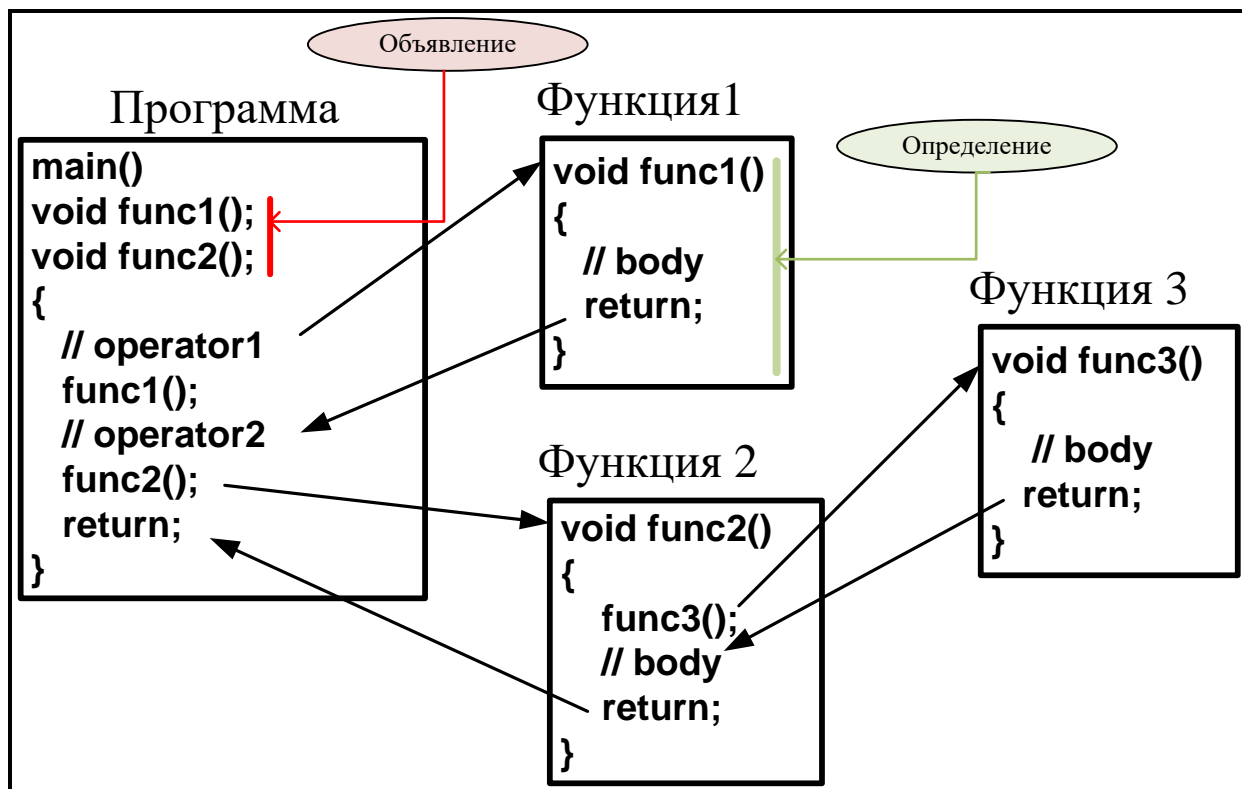
Блок (или составной оператор) – это группа логически связанных между собой инструкций языка программирования, рассматриваемых как единое целое.

В языках программирования инструкции могут быть сгруппированы в специальные логически связанные вычислительные блоки следующего вида:

Псевдокод	C/C++, C#, Java	Pascal/Delphi	Python
Начало <оператор> <оператор> ... Конец	{ <оператор> <оператор> ... }	begin <оператор> <оператор> ... end	<оператор> <оператор> ...

Вычислительный блок называют *составным оператором*.

Функция – фрагмент программного кода, к которому можно обратиться из другого места программы.



Имя функции	С функцией связывается <i>идентификатор</i> (имя функции). Некоторые языки допускают безымянные функции
Адрес функции	С <i>именем функции</i> неразрывно связан <i>адрес ее первой инструкции</i> , которой передается управление при обращении к функции
Адрес возврата	После выполнения функции управление возвращается в точку программы, где данная функция была вызвана (<i>адрес возврата</i>)

- ✓ функция может *принимать параметры*;
- ✓ функция может *возвращать некоторое значение*;
- ✓ функции, которые возвращают *пустое* значение, называют *процедурами*;
- ✓ функция должна быть объявлена и определена.

Объявление функции содержит:

имя функции;
 список имен и типов передаваемых параметров (или аргументов);
 тип возвращаемого функцией значения.

Определение функции содержит исполняемый код функции.

Вызов функции:

Для вызова функции необходимо в требуемом месте программного кода указать имя функции и перечислить передаваемые в функцию параметры.

Способы передачи параметров в функцию:

по значению;

по ссылке.

2.Способы передачи параметров в функцию

по значению	создается локальная копия переменной; любые изменения переменной в теле функции происходят с локальной копией; значение самой переменной не изменяется.
по ссылке	изменения происходят с самой переменной по адресу ее размещения в памяти.

Для переменной, переданной по ссылке функция определяет собственную (локальную) область видимости. В нее входят входные параметры и переменные, которые объявляются непосредственно в теле самой функции.

Функция – подпрограмма, выполняющая какие-либо операции и возвращающая значение.

Процедура – подпрограмма, которая выполняет операции, и не возвращает значения.

Метод – это функция или процедура, которая принадлежит классу или экземпляру класса.

Примеры функций на различных языках программирования:

C/C++	Visual Basic	Python
<pre>int getMul(int x, int y) { return x * y; };</pre>	<pre>Sub Name(text) Console.WriteLine(text) End Sub</pre>	<pre>def func(text): print(text)</pre>

3. Область видимости переменных:

Область видимости переменных – доступность переменных по их идентификатору в разных частях (блоках программы).

Область видимости переменных в C++:

- переменная должна быть объявлена до ее использования;
- переменная объявленная во внутреннем блоке (локальная переменная) не доступна во внешнем;
- переменная объявленная во внешнем блоке доступна во внутреннем;
- во внутреннем блоке переменная может быть переобъявлена.

Область видимости переменной (идентификатора) зависит от места ее объявления в тексте программы.

Область действия идентификатора – это часть программы, в которой его можно использовать для доступа к связанной с ним области памяти.

В зависимости от области действия переменная может быть *локальной* или *глобальной*.

```
#include "stdafx.h"
#include <locale>
#include <iostream>
```

```
int v1 = 1;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    std::cout<<"v1 ="<< v1 <<std::endl;

    system("pause");
    return 0;
```

C:\Users\User Pc\documents\visual studio 2...

v1 =1

Для продолжения нажмите любую клавишу . . .

Оператор разрешения области видимости «: :» (два двоеточия)

```
#include "stdafx.h"
#include <iostream>
```

```
int v1 = 1;
```

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

```
{
```

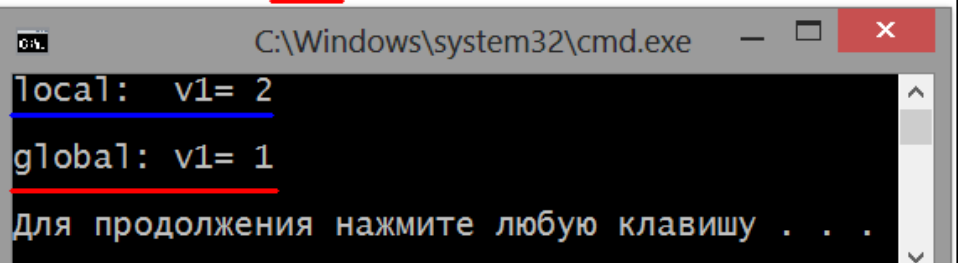
```
    int v1 = 2;
```

```
    std::cout << "local: v1= " << v1 << std::endl << std::endl;
```

```
    std::cout << "global: v1= " << ::v1 << std::endl << std::endl;
```

```
    return 0;
```

```
}
```



```
C:\Windows\system32\cmd.exe
local: v1= 2
global: v1= 1
Для продолжения нажмите любую клавишу . . .
```

```
#include <iostream>
```

```
int var = 1;
```

```
int main()
```

```
{
```

```
    std::cout << "var= " << var << std::endl;
```

```
    int var = 2;
```

```
    std::cout << "var= " << var << std::endl;
```

```
    while (var > 0) {
```

```
        int var = 0;
```

```
        --var;
```

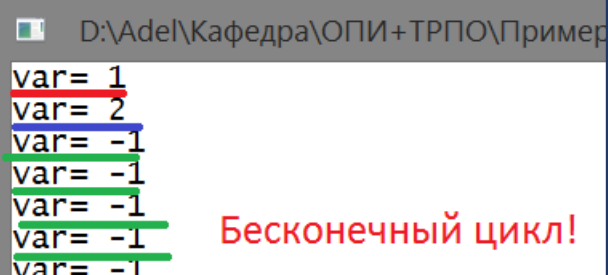
```
        std::cout << "var= " << var << std::endl;
```

```
    }
```

```
    system("pause");
```

```
    return 0;
```

```
}
```



```
D:\Adel\Кафедра\ОПИ+ТРПО\Пример
var= 1
var= 2
var= -1
var= -1
var= -1
var= -1
var= -1
var= -1
Бесконечный цикл!
```

Проблемы не найдены.

4. Программные библиотеки

Назначение библиотек – предоставить программисту стандартный простой и надёжный механизм повторного использования кода.

При выполнении определенных *соглашений* библиотеки можно использовать в программных проектах, реализуемых на нескольких языках программирования.

Классификация библиотек:

- библиотеки на языках программирования (библиотеки классов, шаблонов, функций и т. п.). Компилируются *вместе с исходными файлами проекта*;
- библиотеки объектных модулей (статические библиотеки). Компилируются *вместе с объектными файлами проекта*;
- библиотеки исполняемых модулей (динамические библиотеки). Загружаются в память в момент запуска программы или во время ее исполнения, по мере надобности.

Типы библиотек:	статические; динамические.
------------------------	-------------------------------

Статическая библиотека (англ. library) — набор подпрограмм или объектов, которые подключаются к исходной программе в виде объектных файлов во время *компоновки*.

Использование библиотек:

- отлаженные функции, разработанные в больших проектах, помещают в библиотеку (время трансляции уменьшается);
- функции из библиотеки можно вызывать в разных программах.

Статическая библиотека:

- это файл (в Microsoft с расширением *lib*, в Linux – *a*), содержащий объектные модули;
- является *входным файлом* для компоновщика (*Linker*).

Преимущества:

- просто использовать;
- не требуется наличие самой библиотеки;
- исполняемый файл один (расширение .exe).

Недостатки:

- платформенно зависима;
- загружается в память с каждым экземпляром запущенного приложения;
- при изменении кода библиотеки необходима компоновка всех приложений, которые используют библиотеку.

Динамическая библиотека (или «общая библиотека»):

файл, содержащий машинный код (в Microsoft с расширением *dll*, в Linux – *so*, в Mac OS – *dylib*);

Загружается в память процесса загрузчиком программ операционной системы либо при создании процесса, либо по запросу уже работающего процесса, то есть динамически.

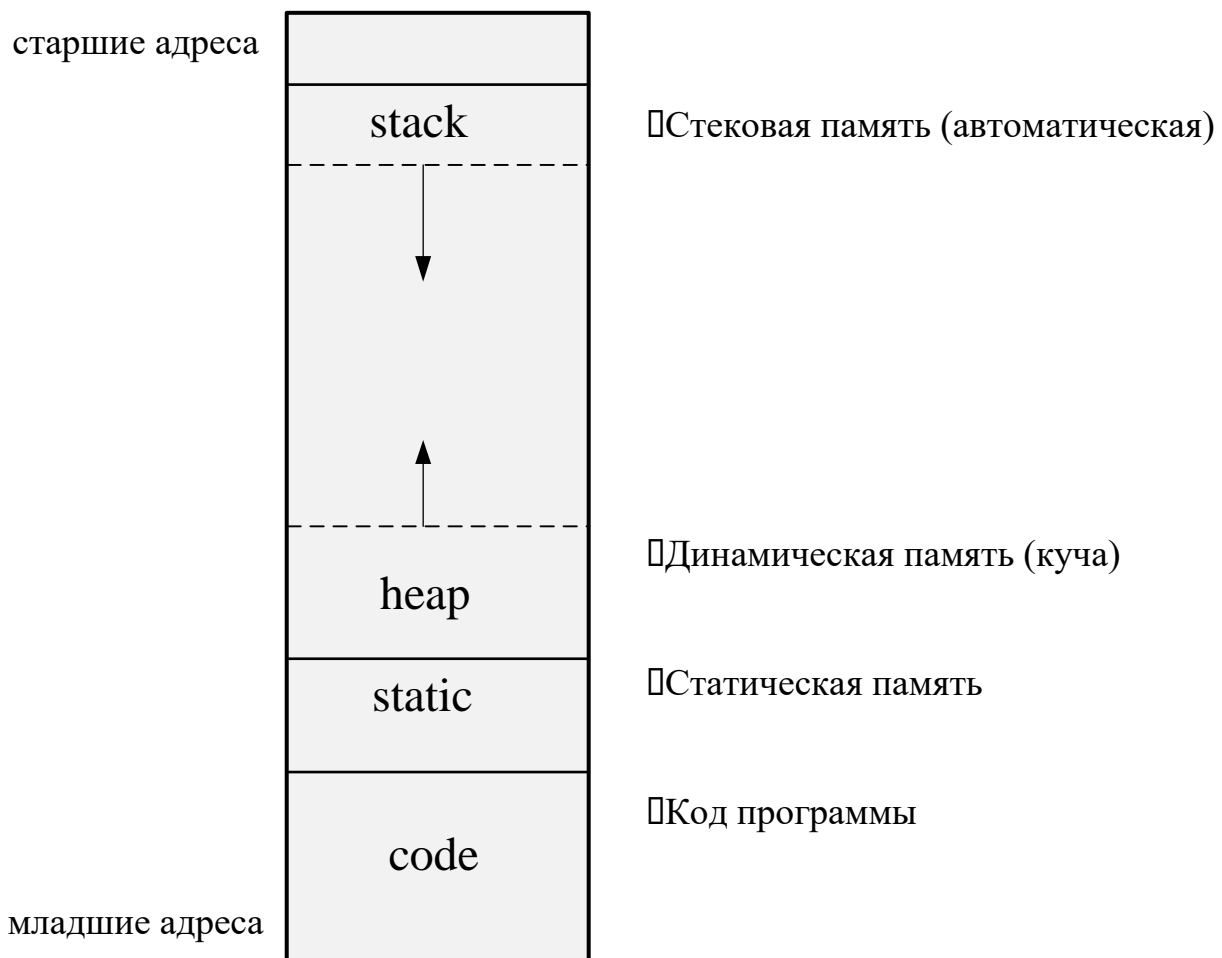
Преимущества:

- совместное использование одной копии динамической библиотеки несколькими программами (экономит адресного пространства);
- возможность обновления динамической библиотеки до более новой версии без необходимости перекомпиляции всех исполняемых файлов, которые ее используют.

5. Модель памяти C/C++:

Модель памяти языка C++ предоставляет области для хранения кода и данных.

Структура памяти C/C++-программ:



Область кода – память, в которой размещается код программы.

Статическая память

Static или **статическая память** выделяется до начала работы программы, на стадии компиляции и служит для хранения статических переменных.

Типы статических переменных: **глобальные переменные** и **статические переменные**.

Глобальные переменные – это переменные, определенные вне функций. Память для глобальных переменных выделяется на этапе компиляции. Глобальные переменные доступны в любой точке программы во всех ее файлах.

Статические переменные – это переменные, в описании которых присутствует ключевое слово **static**. Компилятор выделяет для таких переменных постоянное место хранения в статической области памяти.

При объявлении переменной в функции ключевое слово `static` указывает, что переменная *удерживает свое состояние между вызовами этой функции*.

Стековая память

Stack или *стековая (автоматическая) память* предназначена для хранения локальных переменных.

Локальные переменные хранятся в стеке.

Стек — это непрерывная область оперативной памяти, организованная по принципу LIFO (последний вошел, первый вышел).

Динамическая память

Heap или динамическая память, или куча — это область памяти, выделение которой в языке программирования C++ производится с помощью оператора `new`, освобождение — оператором `delete`.

Пример. Объявление глобальных статических переменных (компонуются редактором связей (linker)).

The screenshot shows a C++ IDE with a code editor and a build output window. The code editor contains the following code:

```
1 #include <iostream>
2
3 int getSum(int x, int y);
4 int getMul(int x, int y);
5
6 int parm1 = 2;
7 int parm2 = 3;
8
9 int main(int argc, char* argv[])
10 {
11     std::cout << "getSum(" << parm1 << ", " << parm2 << ") = " << getSum(2, 3) << std::endl;
12
13     parm1 = 5;
14     parm2 = 5;
15     std::cout << "getMul(" << parm1 << ", " << parm2 << ") = " << getMul(2, 3) << std::endl;
16
17     system("pause");
18     return 0;
19 }
```

Two callouts highlight function definitions:

- Blue box: `int getSum(int x, int y) { return x + y; };`
- Red box: `int getMul(int x, int y) { return x * y; };`

The right sidebar shows a solution explorer for "Решение 'Lec08'" (project: 1 и 2). It lists files: `file_mul.cpp`, `file_sum.cpp`, and `lec_08.cpp`.

The bottom window shows the build output:

```
Вывод
Показать выходные данные из: Сборка
1>Целевой объект InitializeBuildStatus:
1> Создание "Debug\Lec08.tlog\unsuccessfulbuild", так как было задано "AlwaysCreate".
1>Целевой объект ClCompile:
1> file_mul.cpp
1> file_sum.cpp
1> lec_08.cpp
1> Создание кода...
1>Целевой объект Link:
1> Lec08.vcxproj -> D:\Adel\Кафедра\ОПИ+ТРПО\Примеры к лабораторным работам\Lec08\Debug\Lec08.exe
1>Целевой объект FinalizeBuildStatus:
1> Файл "Debug\Lec08.tlog\unsuccessfulbuild" удаляется.
```

```
D:\Adel\Кафедра\ОПИ+ТРПО\Примеры к лабораторн
getSum(2,3) = 5
getMul(5,5) = 6
Для продолжения нажмите любую клавишу . . .
```

Локальная статическая память.

```
#include <iostream>

int getInc(int x) {
    static int k = 1;
    return x += k;
};

int main(int argc, char* argv[])
{
    for (int i = 0; i <= 5; i++) {
        int k = 2; <- локальная
        std::cout << i << ": " << getInc(i) << std::endl;
    }

    system("pause");
    return 0;
}
```

```
D:\Adel\Кафедра\ОПИ+ТР
0: 1
1: 2
2: 3
3: 4
4: 5
5: 6
Для продолжения нажми
<
```

Переменная с ключевым словом **static** — это **статическая** переменная. Время ее жизни — постоянное.

Область видимости статической переменной ограничена одним файлом, внутри которого она определена, ее можно использовать только после ее объявления.

Ключевое слово **static** в языке C/C++ используется для двух различных целей:

- как указание типа памяти: переменная располагается в статической памяти;
- как способ ограничить область видимости переменной рамками одного файла (в случае описания переменной вне функции).

Стек

```
#include "stdafx.h"
#include <locale>
#include <iostream>

int func2 (int y, int z)
{
    int g = 8;
    int h = 9;
    int r = 10;
    return y + z + g + h + r;
};

int func1 (int x, int v)
{
    int k = 5;
    int l = 6;
    int m = 7;
    return k+= func2(k, 7);
};

int main(int argc, _TCHAR* argv[])
{
    std::cout << func1(3, 4)<< std::endl;
    return 0;
}
```

argc

argv

```
#include "stdafx.h"
#include <locale>
#include <iostream>

int func2 (int y, int z)
{
    int g = 8;
    int h = 9;
    int r = 10;
    return y + z + g + h + r;
};

int func1 (int x, int v)
{
    int k = 5;
    int l = 6;
    int m = 7;
    return k+= func2(k, 7);
};

int main(int argc, _TCHAR* argv[])
{
    std::cout << func1(3, 4)<< std::endl;
    return 0;
}
```

m

l

k

x

v

argc

argv

```

#include "stdafx.h"
#include <locale>
#include <iostream>

int func2 (int y, int z)
{
    int g = 8;
    int h = 9;
    int r = 10;
    return y + z + g + h + r;
};

int func1 (int x, int v)
{
    int k = 5;
    int l = 6;
    int m = 7;
    return k+= func2(k, 7);
};

int main(int argc, _TCHAR* argv[])
{
    std::cout << func1(3, 4)<< std::endl;
    return 0;
}

```

r
h
g
y
z
m
l
k
x
v
argc
argv

```

#include "stdafx.h"
#include <locale>
#include <iostream>

int func2 (int y, int z)
{
    int g = 8;
    int h = 9;
    int r = 10;
    return y + z + g + h + r;
};

int func1 (int x, int v)
{
    int k = 5;
    int l = 6;
    int m = 7;
    return k+= func2(k, 7);
};

int main(int argc, _TCHAR* argv[])
{
    std::cout << func1(3, 4)<< std::endl;
    return 0;
}

```

m
l
k
x
v
argc
argv

```

#include "stdafx.h"
#include <locale>
#include <iostream>

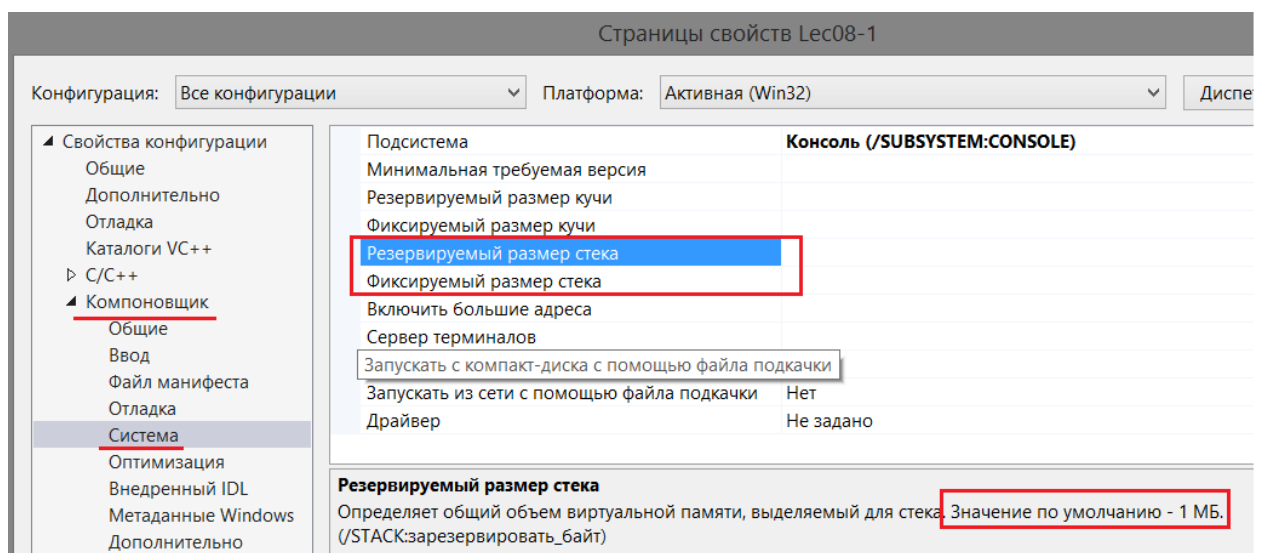
int func2 (int y, int z)
{
    int g = 8;
    int h = 9;
    int r = 10;
    return y + z + g + h + r;
};
int func1 (int x, int v)
{
    int k = 5;
    int l = 6;
    int m = 7;
    return k+= func2(k, 7);
};

int main(int argc, _TCHAR* argv[])
{
    std::cout << func1(3, 4)<< std::endl;
    return 0;
}

```

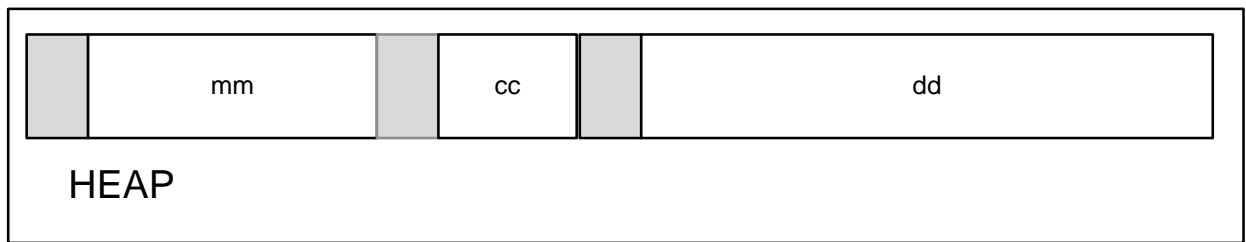
argc
argv

Заданный по умолчанию резервируемый размер стека можно изменить в свойствах проекта раздела Компоновщик □ система.

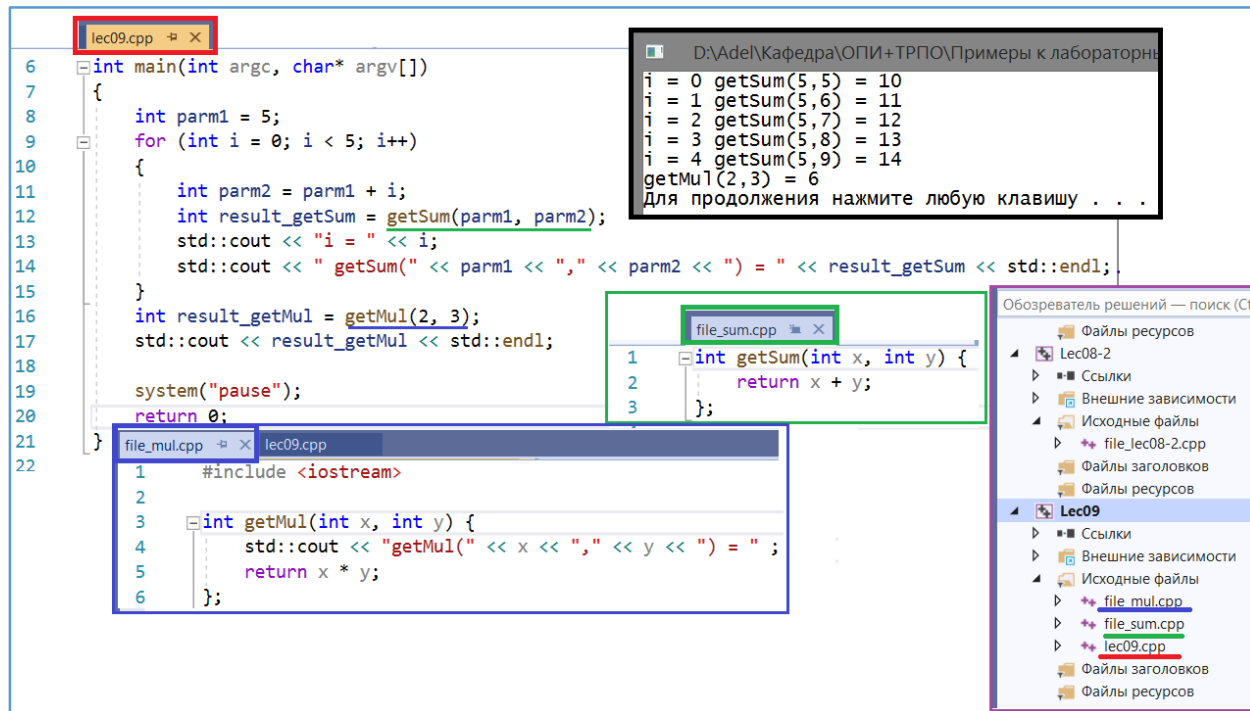


Heap (куча).

Выделение памяти в C++ производится с помощью оператора new, освобождение — оператором delete. Память, выделяемая функциями динамического распределения памяти, находится в куче (heap).



6. Пример многофайлового проекта



7. Основные понятия: отладчик и отладка

Отладчик — инструментальное средство разработки программ, которое присоединяется к работающему приложению и позволяет проверять код, наблюдать за выполнением исследуемой программы, останавливать и перезапускать её, изменять значения в памяти, просматривать стек вызовов и т.д.


Назначение отладчика — устранение ошибок в коде программы.

Отладка — процесс запуска и выполнения программы в режиме отладки.

а. Запуск отладчика



Способы запуска отладчика в Visual Studio для C++:

- пункт главного меню Отладка ☐ Начать отладку;

- горячая клавиша F5;
- горячая клавиша F10 (запуск в пошаговом режиме);
- иконка  на панели инструментов.

в. Прекращение отладки

Способы остановки отладчика:

- пункт главного меню Отладка  Остановить отладку;
- комбинация клавиш SHIFT + F5;
- иконка остановки  на панели инструментов.

Также необходимо закрыть окно консоли.

с. Установка точки останова и запуск отладчика

Точка останова (breakpoint) – это точка, в которой процесс выполнения программы приостанавливается и отладчик получает управление.

Пример.

```

1  #include <iostream>
2
3  int getSum(int x, int y);
4  int getMul(int x, int y);
5
6  int main(int argc, char* argv[])
7  {
8      int parm1 = 5;
9      for (int i = 0; i < 5; i++)
10     {
11         int parm2 = parm1 + i;
12         int result_getSum = getSum(parm1, parm2);
13         std::cout << "i = " << i;
14         std::cout << " getSum(" << parm1 << ", " << parm2 << ") = " << result_getSum << std::endl;
15     }
16     int result_getMul = getMul(2, 3);
17     std::cout << result_getMul << std::endl;
18
19     system("pause");
20     return 0;
21 }

```

Установить точку останова можно, щелкнув слева от строки с номером 15 по серому полю.

Пример 1. Выполнить следующую последовательность действий.

1. Начать отладку.
2. Установить точку останова на 15-й строке кода.

В этом месте появится красный круг, отмечающий точку останова.

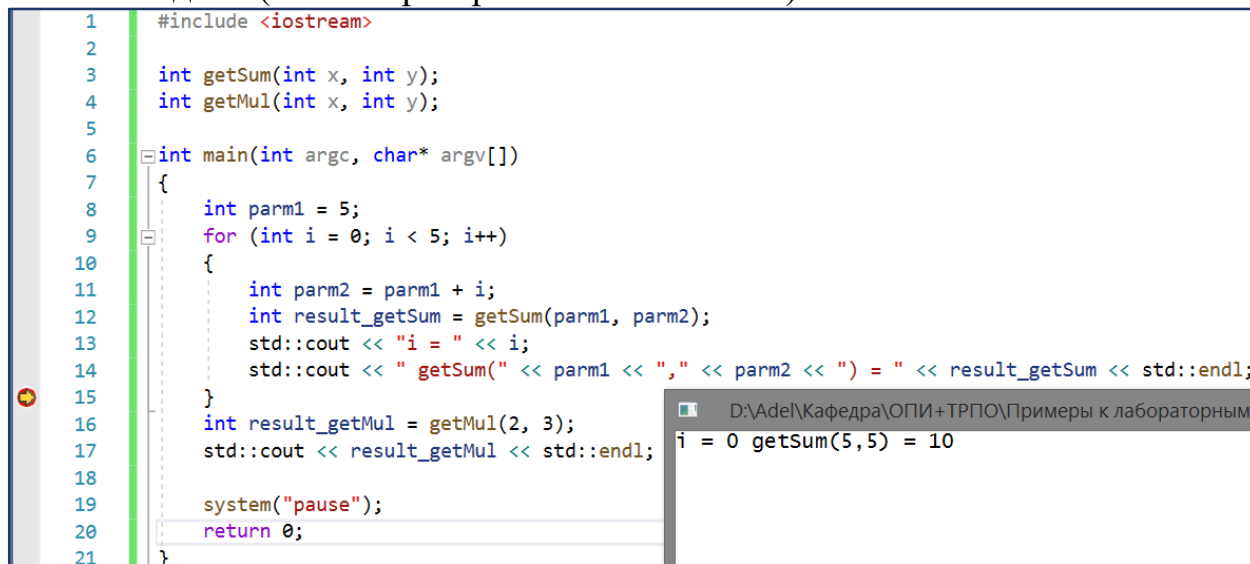
Точка останова указывает, где Visual Studio приостановит выполнение кода и обеспечит возможность для выполнения необходимых действий в режиме отладки.

Если точка останова не установлена, то отладчик запускается и выполняет приложение целиком.

Иначе отладчик запускается и останавливается в первой точке останова.

3. Нажать  для запуска процесса отладки.

Желтая стрелка отмечает оператор в коде, на котором приостановлен отладчик (этот оператор пока не выполнен).



```




1  #include <iostream>
2
3  int getSum(int x, int y);
4  int getMul(int x, int y);
5
6  int main(int argc, char* argv[])
7  {
8      int parm1 = 5;
9      for (int i = 0; i < 5; i++)
10     {
11         int parm2 = parm1 + i;
12         int result_getSum = getSum(parm1, parm2);
13         std::cout << "i = " << i;
14         std::cout << " getSum(" << parm1 << ", " << parm2 << ") = " << result_getSum << std::endl;
15     }
16     int result_getMul = getMul(2, 3);
17     std::cout << result_getMul << std::endl;
18
19     system("pause");
20     return 0;
21 }
  
```

Output: i = 0 getSum(5,5) = 10

d. Пошаговая отладка

Некоторые возможности управления режимом отладки:

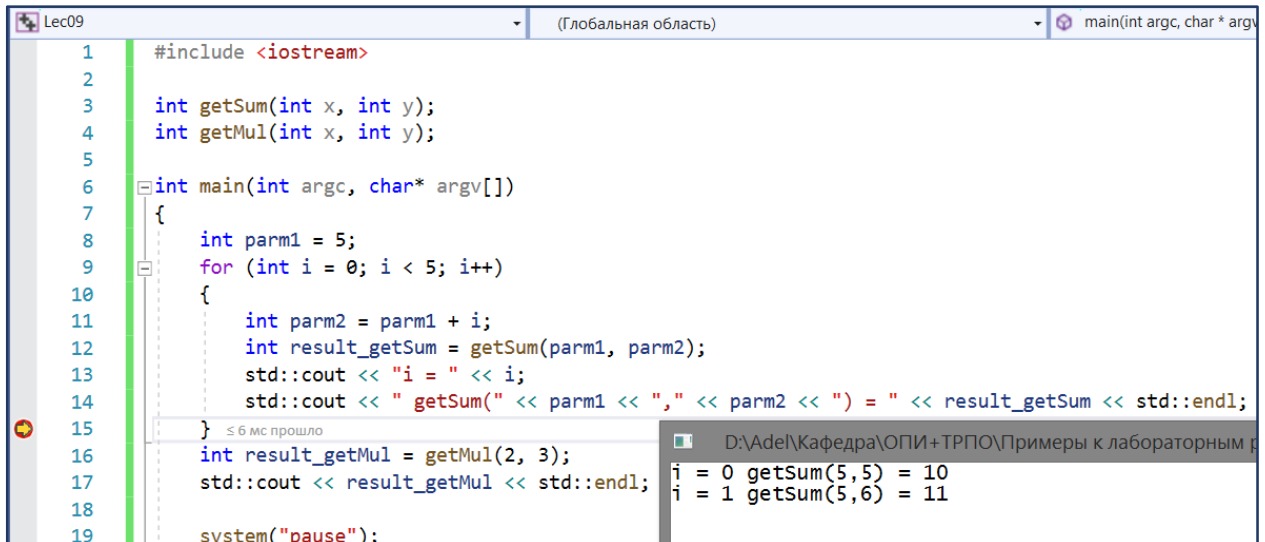
Иконка на панели инструментов	Пункт меню «Отладка»	Горячие клавиши	Описание
 Go	Продолжить	F5	продолжить выполнение программы до следующей точки останова
	Остановить отладку	Shift+F5	
	Перезапустить	Ctrl+Shift+F5	
 Step Into	Шаг с заходом	F11	выполнить одну инструкцию с «заходом» в функцию. Если это вызов функции, то точка выполнения перемещается на первую инструкцию этой функции
 Step Over	Шаг с обходом	F10	выполнить одну инструкцию. Если это вызов функции, то она выполняется целиком
 Step Out	Шаг с выходом	Shift+F11	прервать выполнение текущей функции и вернуться в вызывающую функцию

	Перейти к следующей точке останова	F9	
	На шаг назад	Alt+[
	Остановить отладку	Shift+F5	

е. Проход по коду в отладчике с помощью пошаговых команд

4. Выполнить команду «Продолжить».

Результат:



```

1  #include <iostream>
2
3  int getSum(int x, int y);
4  int getMul(int x, int y);
5
6  int main(int argc, char* argv[])
7  {
8      int parm1 = 5;
9      for (int i = 0; i < 5; i++)
10     {
11         int parm2 = parm1 + i;
12         int result_getSum = getSum(parm1, parm2);
13         std::cout << "i = " << i;
14         std::cout << " getSum(" << parm1 << ", " << parm2 << ") = " << result_getSum << std::endl;
15     }
16     int result_getMul = getMul(2, 3);
17     std::cout << result_getMul << std::endl;
18
19     system("pause");

```

≤ 6 мс прошло

D:\Ade\Кафедра\ОПИ+ТРПО\Примеры к лабораторным р

i = 0 getSum(5,5) = 10
i = 1 getSum(5,6) = 11

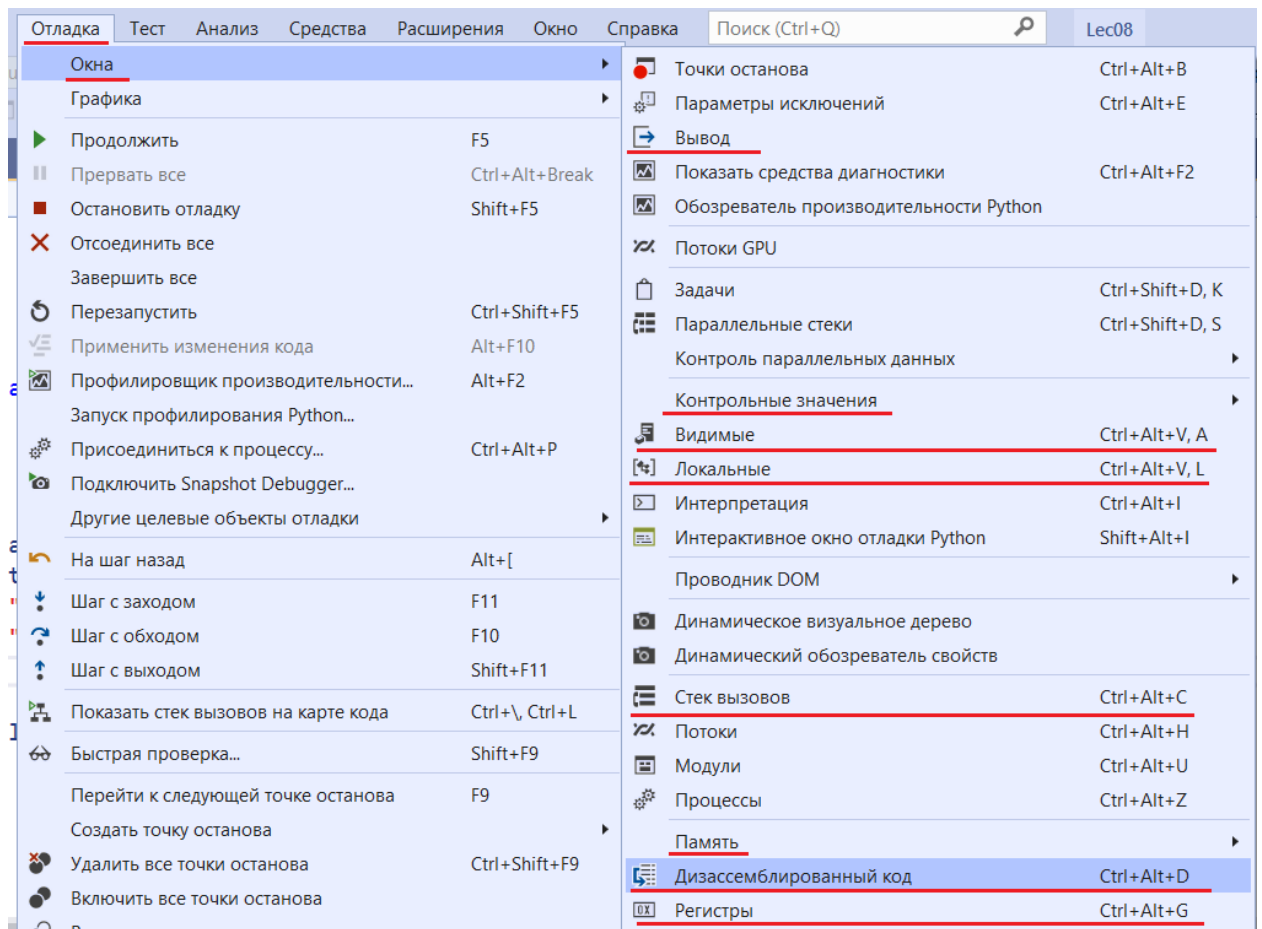
Нажать клавишу **F10** (или выбрать пункт меню Отладка □ Шаг с обходом).
Отладчик выполняет инструкции без захода в функции (или методы) в коде приложения.

ф. Быстрый перезапуск приложения

Нажать иконку  на панели инструментов отладки для перезапуска приложения (или сочетание клавиш CTRL + SHIFT + F5).

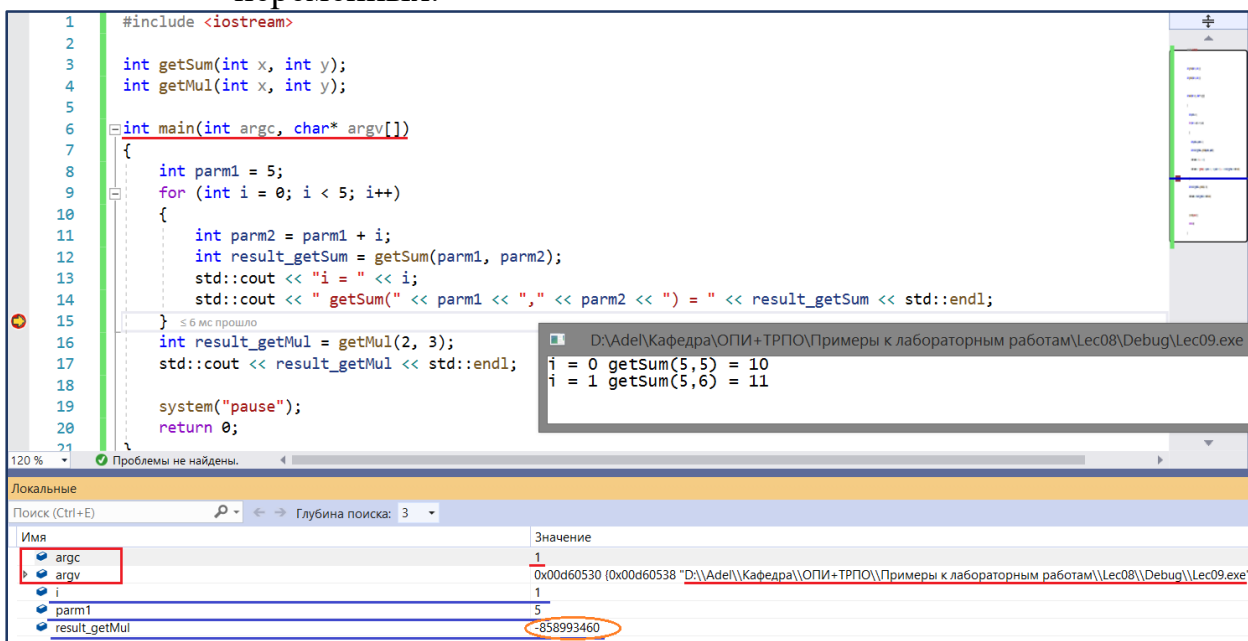
Окна отладчика

Показать и скрыть отладочные окна: меню Отладка □ Окна:



Окно «Локальные»

5. В окне «Локальные» автоматически отображаются значения локальных переменных:



g. Проверка переменных и изменение их значений с помощью подсказок по данным

6. При наведении указателя мыши на переменную **i** можно посмотреть ее текущее значение – это целочисленное значение **1**.

```
9   for (int i = 0; i < 5; i++)
10  {
11      int parm2 = parm1 + i;
12      int result_getSum = getSum(parm1, parm2);
13      std::cout << "i = " << i;
```

7. Навести указатель мыши на переменную **i**, чтобы изменить ее текущее значение на новое значение – **3**.

Для этого в правой части прямоугольника набрать нужное значение:

```
9   for (int i = 0; i < 5; i++)
10  {
11      int parm2 = parm1 + i;
12      int result_getSum = getSum(i 3 parm1, parm2);
13      std::cout << "i = " << i;
14      std::cout << " getSum(" << parm1 << ", " << pa
```

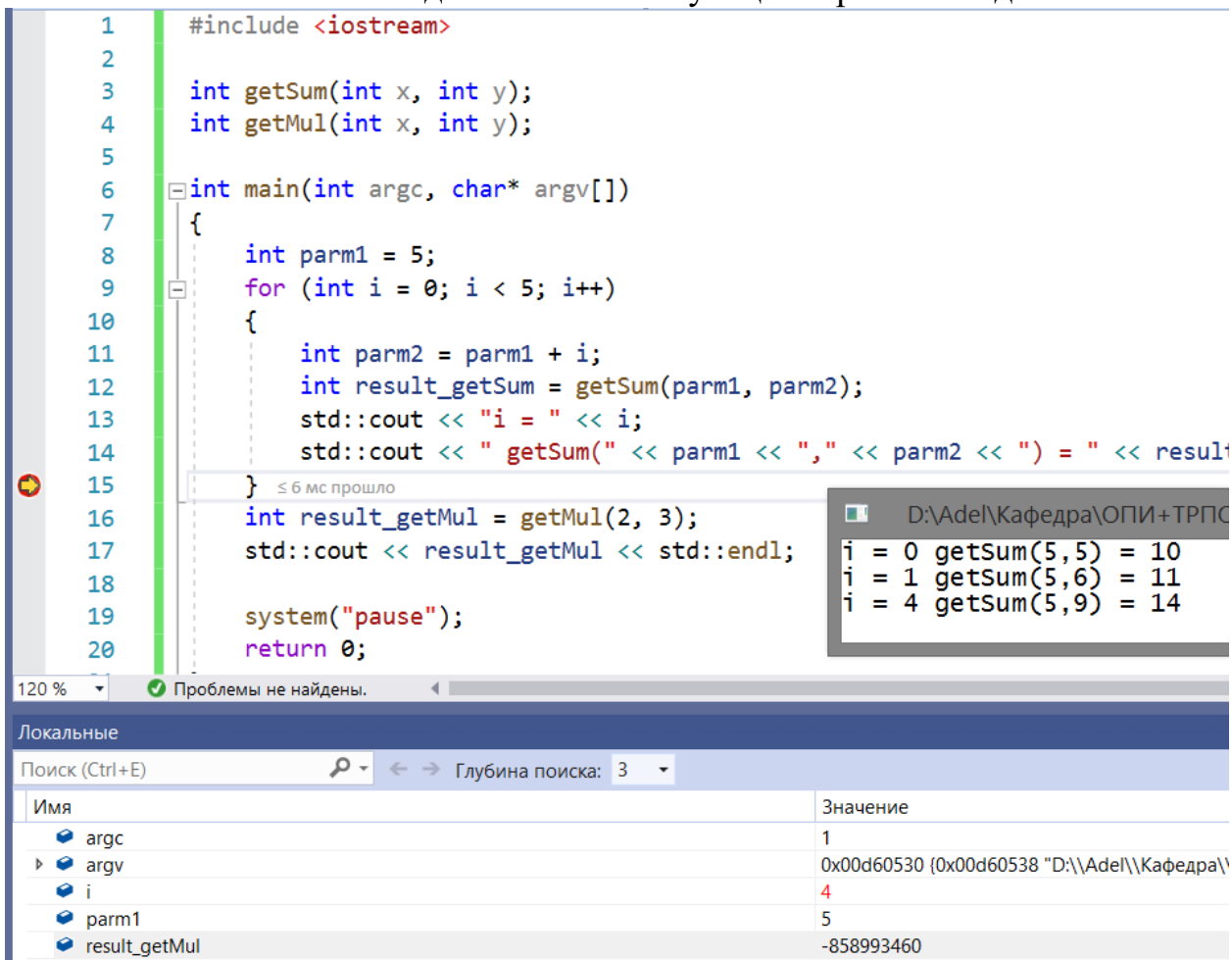
Результат:

Локальные	
Поиск (Ctrl+E) 🔍 ← → Глубина поиска: 3	
Имя	Значение
argc	1
argv	0x00d60530 {0x00d60538 "D:\\
i	3
parm1	5
result_getMul	-858993460

8. Продолжить выполнение отладки, нажав клавишу F5 (или выберите Отладка □ Продолжить).

Выполнена еще одна итерация цикла `for` и, при наведении указателя мыши в точке останова на переменную `i`, отображается ее новое вычисленное значение.

В окне «Локальные» отображаются значения локальных переменных и в окно консоли выводится соответствующая строка вывода:



```
1  #include <iostream>
2
3  int getSum(int x, int y);
4  int getMul(int x, int y);
5
6  int main(int argc, char* argv[])
7  {
8      int parm1 = 5;
9      for (int i = 0; i < 5; i++)
10     {
11         int parm2 = parm1 + i;
12         int result_getSum = getSum(parm1, parm2);
13         std::cout << "i = " << i;
14         std::cout << " getSum(" << parm1 << "," << parm2 << ") = " << result_getSum;
15     }
16     int result_getMul = getMul(2, 3);
17     std::cout << result_getMul << std::endl;
18
19     system("pause");
20     return 0;
}
```

≤ 6 мс прошло

D:\Adel\Кафедра\ОПИ+ТРПС

```
i = 0 getSum(5,5) = 10
i = 1 getSum(5,6) = 11
i = 4 getSum(5,9) = 14
```

120 % Проблемы не найдены.

Локальные

Поиск (Ctrl+E) Глубина поиска: 3

Имя	Значение
argc	1
argv	0x00d60530 {0x00d60538 "D:\Adel\Кафедра\ОПИ+ТРПС"
i	4
parm1	5
result_getMul	-858993460

Окно «Видимые»

В окне «Видимые» отображаются все переменные и их текущие значения. Окно «Видимые» позволяет просматривать/изменять значения переменных и выражений.

9. Продолжить выполнение отладки в пошаговом режиме (F10).

Результат после выполнения 12-й строки кода:

The screenshot shows a C++ code editor with the following code:

```
8   int parm1 = 5;
9   for (int i = 0; i < 5; i++)
10  {
11      int parm2 = parm1 + i;
12      int result_getSum = getSum(parm1, parm2);
13      std::cout << "i = " << i;
14      std::cout << " getSum(" << parm1 << ", " << parm2 << "
15  }
16  int result_getMul = getMul(2, 3);
17  std::cout << result_getMul << std::endl;
18
19  system("pause");
20  return 0;
21
22
```

The 'Видимые' (Visible) window is open, showing the following variables and their values:

Имя	Значение
Функция "getSum" вернула	14
i	4
parm1	5
parm2	9
result_getSum	14

10. Остановить отладку.

Окно «Контрольные значения»

Окно «Контрольные значения» позволяет просматривать/изменять значения переменных, выполнять операторы и вычислять выражения.

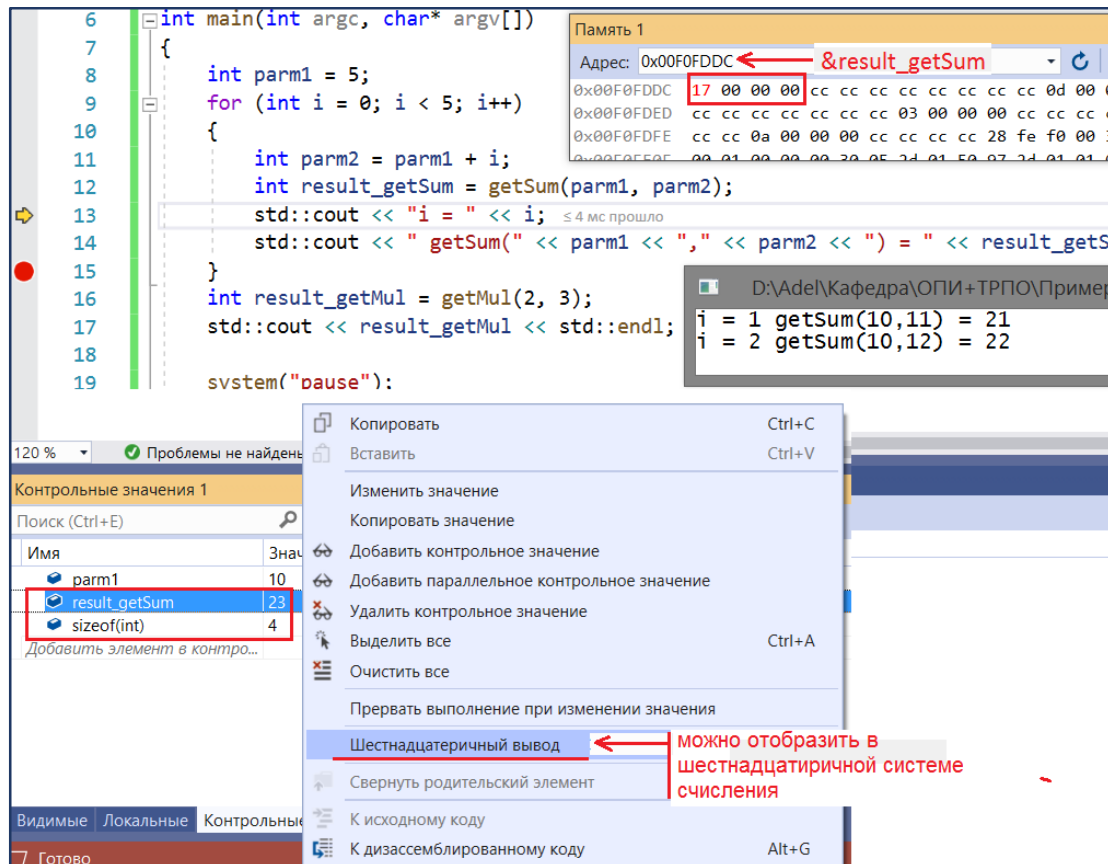
Добавить переменную или выражение в окно «Контрольные значения» можно одним из следующих способов:

- ввести имя переменной с клавиатуры;
- перетащить из окна редактора исходного кода (для этого нужно предварительно выделить нужную переменную или выражение);
- вызвать контекстное меню на имени переменной и выбрать команду «Добавить контрольное значение».

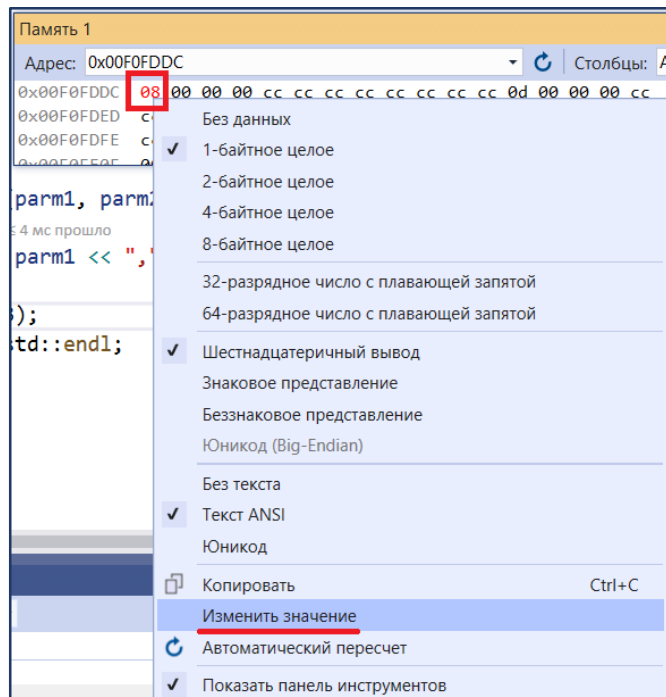
Чтобы изменить значение переменной, достаточно сделать двойной щелчок на старом значении и ввести новое.

Окно «Памяти»

Окно «Памяти» позволяет просматривать содержимое ячеек памяти. Содержимое памяти может отображаться в различных форматах, которые выбираются из контекстного меню.



Значение любой ячейки памяти можно изменить. Для этого следует переместить курсор ввода в нужное место и используя пункт контекстного меню «Изменить значение» ввести новое значение поверх старого:

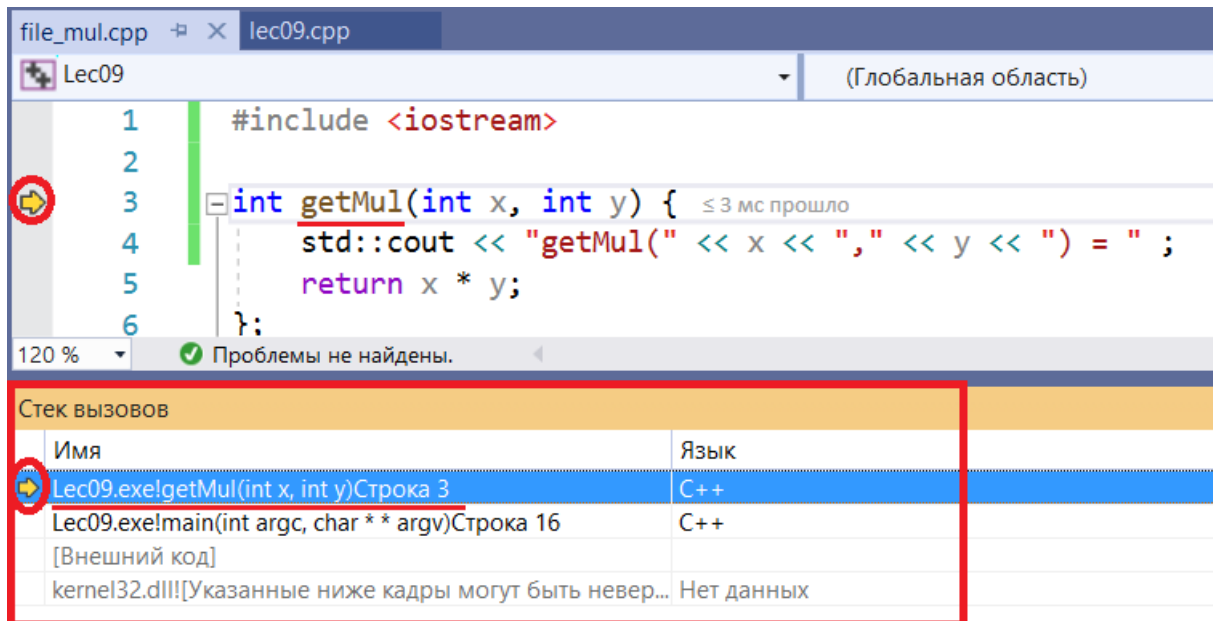


Просмотр стека вызовов

Стек вызовов (call stack) – это список всех активных функций, которые вызывались, до текущей точки выполнения исходного кода.

Открыть окно «Стека вызовов» можно в режиме отладки, выбрав пункт меню:

Отладка ☐ Окна ☐ Стек вызовов.



Когда происходит вызов функции, эта функция добавляется в вершину стека вызовов. Когда выполнение этой функции прекращается, она удаляется с вершины стека и управление передается к вызывающей функции (ее имя теперь лежит в вершине стека вызовов).

Стек вызовов используется для изучения и анализа потока выполнения приложения.

Окно «Регистры»

Открыть окно отладчика «Регистры». В контекстном меню окна выбирать ЦП для отображения содержимое регистров.

Просмотр дизассемблированного кода в отладчике

В окне «Дизассемблированный код» отображается код сборки, соответствующий инструкциям, созданным *компилятором*.

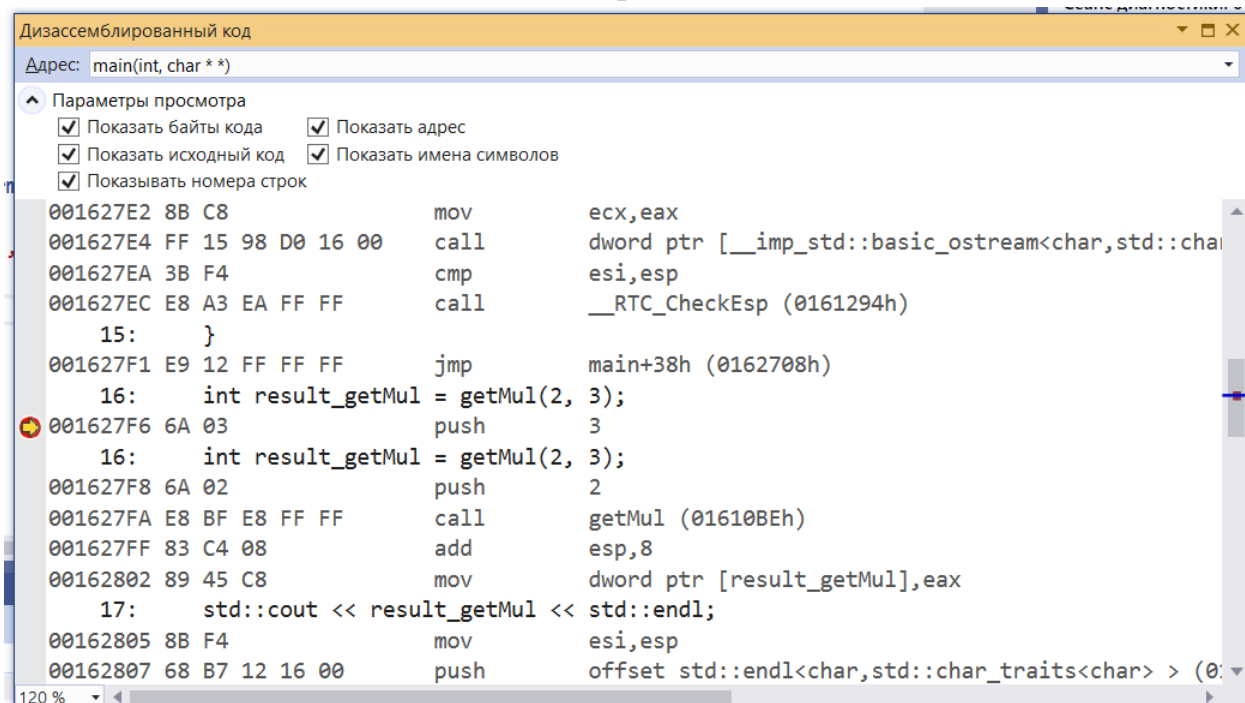
```

6  int main(int argc, char* argv[])
7  {
8      int parm1 = 5;
9      for (int i = 0; i < 5; i++)
10     {
11         int parm2 = parm1 + i;
12         int result_getSum = getSum(parm1, parm2);
13         std::cout << "i = " << i;
14         std::cout << " getSum(" << parm1 << "," << parm2 << ") = " << result_
15     }
16     int result_getMul = getMul(2, 3);
17     std::cout << result_getMul << std::endl;
18
19     system("pause");
20     return 0;
21 }

```

Пример 3. Выполнить следующую последовательность действий.

1. В отладчике установить точку останова на 16-й строке кода.
2. Начать отладку.
3. Выполнение программы остановится на 16-ой строке кода.
4. Открыть окно отладчика «Регистры», отображающее содержимое регистров (в контекстном меню окна выбираем ЦП).
5. Открыть окно отладчика «Память».
6. Установить курсор на строку 16 и вызвать с помощью контекстного меню «Дизассемблированный код».



В окне дизассемблированного кода установить параметры просмотра. Для этого отметить следующие чекбоксы:

- ✓ Показать байты кода
- ✓ Показать исходный код

- ✓ Показать адрес
- ✓ Показать имена символов
- ✓ Показывать номера строк

Команда – оператор программы, который непосредственно выполняется процессором.

Команды языка ассемблера – это символьная форма записи машинных команд. Команды имеют следующий синтаксис:

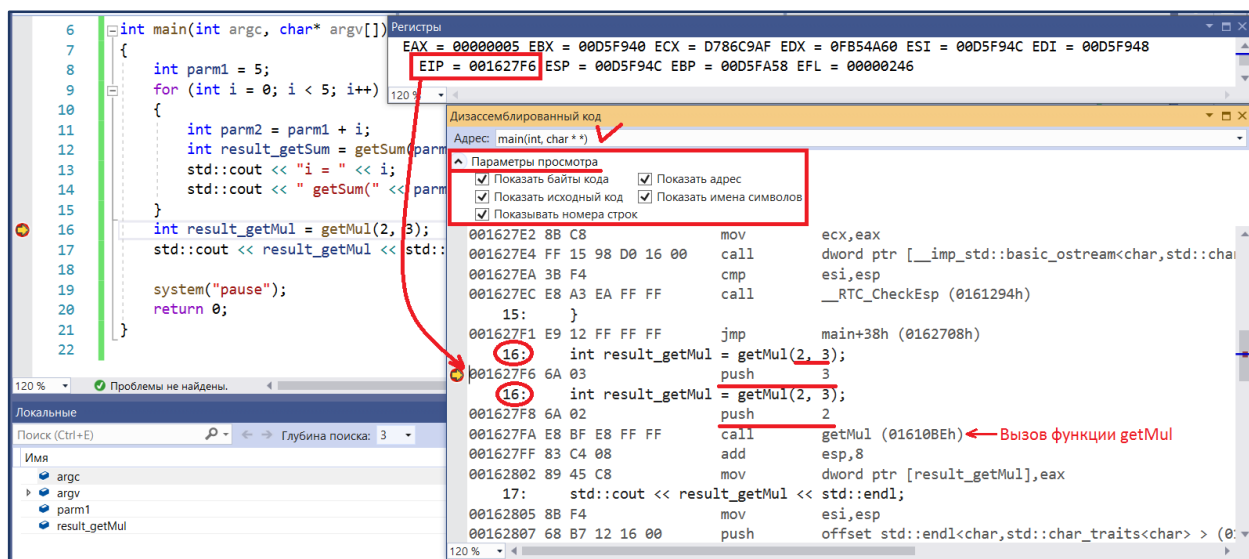
[метка] (необязательный)	мнемоника	[операнд(ы)]	[:комментарий]
-----------------------------	-----------	--------------	----------------

Метка – идентификатор, с помощью которого, можно пометить участок кода или данных. Метка кода должна отделяться двоеточием.

Мнемоника команды – короткое имя, определяющее тип выполняемой процессором операции.

Операнд определяет данные (регистр, ссылка на участок памяти, константное выражение), над которыми выполняется действие по команде, если операндов несколько, то они отделяются друг от друга запятыми.

г. Функции, взгляд на уровне дизассемблера

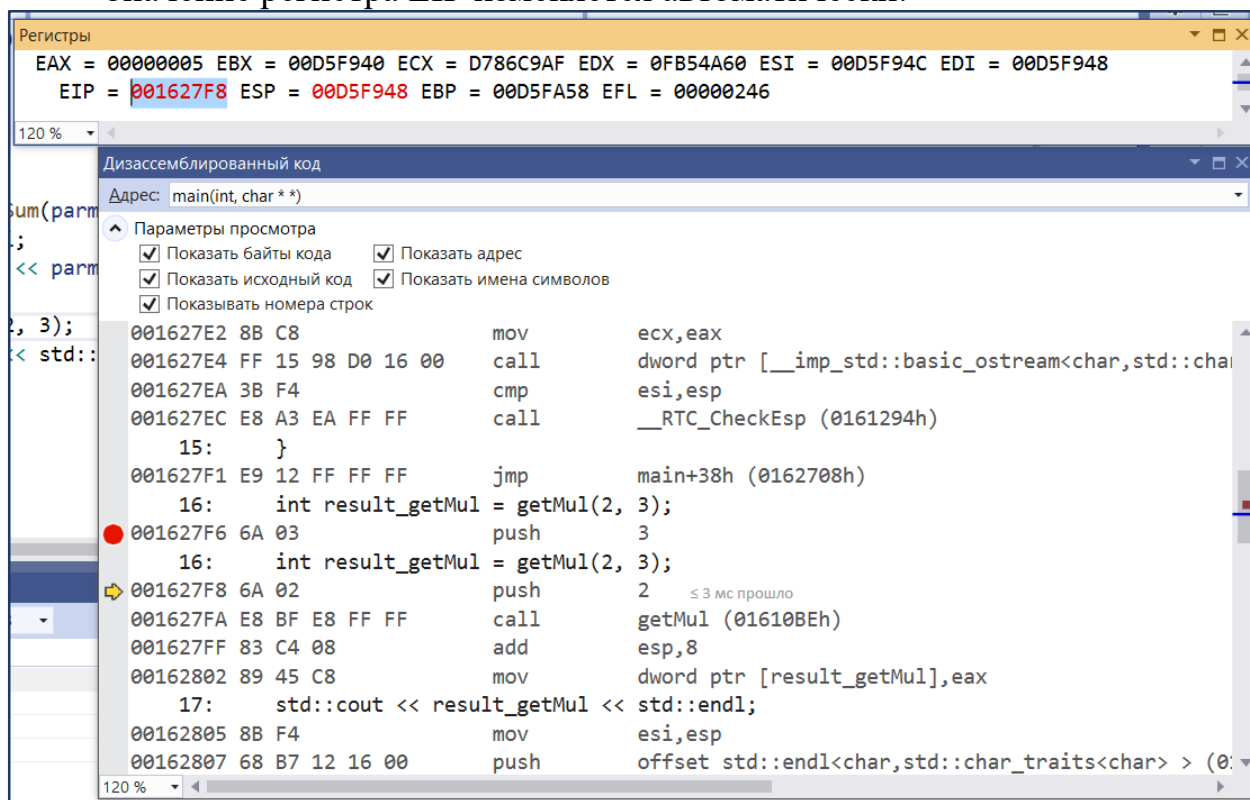


Регистр EIP - указатель на инструкцию, которая должна быть выполнена процессором. Содержимое регистра **EIP** нельзя изменять явно. Он **обновляется** автоматически в следующих случаях:

1. **Процессор закончил выполнение инструкции.** Инструкция имеет определенную длину – определенное количество байт выполняемого кода. Процессор знает, сколько байт занимает инструкция и, соответственно, сдвигает указатель на нужное количество байт после каждой инструкции.
2. **Выполнена инструкция ret (return) - возврат.**
3. **Выполнена инструкция call - вызов.**

7. Выполняем шаг отладки в окне дизассемблированного кода (F10) для выполнения данной инструкции и перехода к следующей. Значение регистра EIP автоматически увеличилось на 2 и стало равным **0x001627F8**, так как инструкция использовала ровно 2 байта машинного кода (байты **6A 03** по адресу **0x001627F6**).

Значение регистра EIP изменяется автоматически:

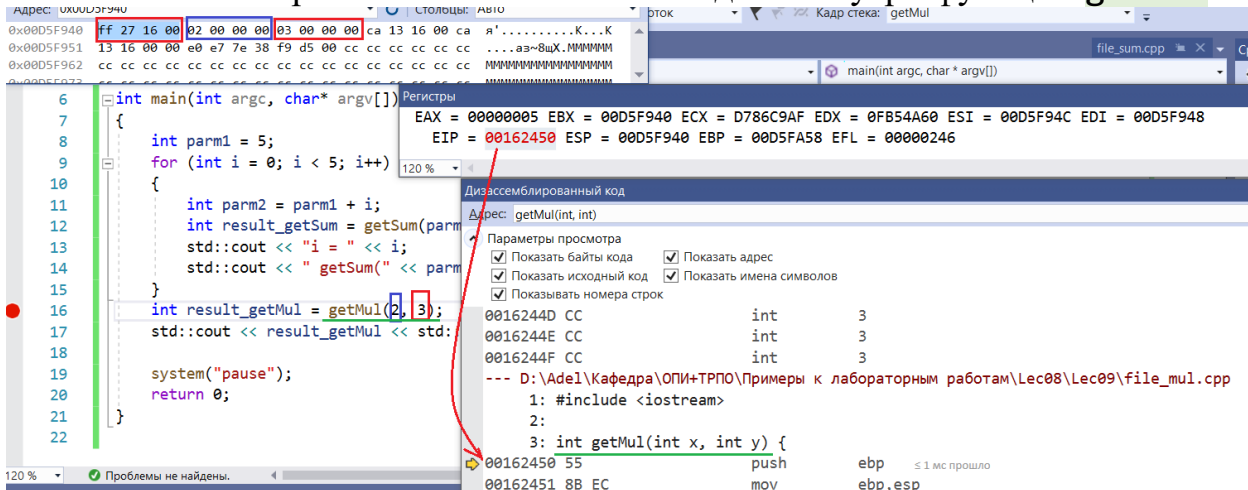


8. Выполняем шаг отладки (F10) и проверяем значение регистра EIP, оно опять увеличилось на 2 и стало равным **0x001627FA**. Следующая строка кода (инструкция **call**) – это вызов функции **getMul**. Эта инструкция переносит поток выполнения по указанному адресу. В коде, приведенном на рисунке выше, это адрес **0x001627FA**.

Внимание! Адрес инструкции, следующей за **call** в нашем примере это адрес **0x001627FF**. Запомним его. Сюда поток должен вернуться сразу после выполнения кода вызываемой функции, на который указывала инструкция **call** – это адрес точки возврата.

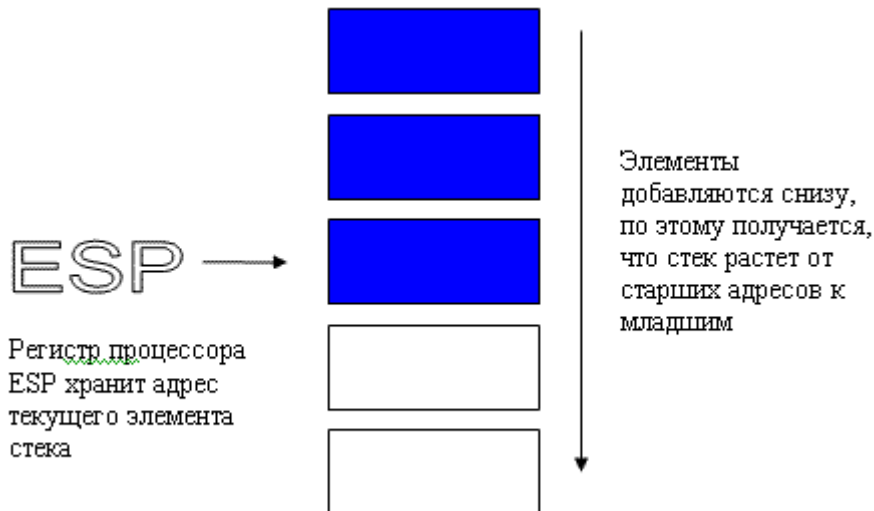
9. Выполнение инструкции **call** (F11 – шаг с заходом) передаст управление в функцию **getMul**. При этом значение EIP изменится на **0x00162450** – это адрес первой инструкции функции **getMul**.

Теперь поток выполнения находится внутри функции **getMul**:



Регистр ESP – указатель на стек – это область памяти, зарезервированная операционной системой, в которой создаются локальные переменные функции и помещаются параметры, передаваемые в функцию. Стек увеличивается или уменьшается по мере того, как функции вызываются или завершают свое выполнение.

Архитектура x86 поддерживает стек.



Стек – это непрерывная область оперативной памяти, организованная по принципу стопки тарелок (LIFO): тарелку можно только брать верхнюю и класть тарелку только поверх стопки. тарелки из середины стопки недоступны.

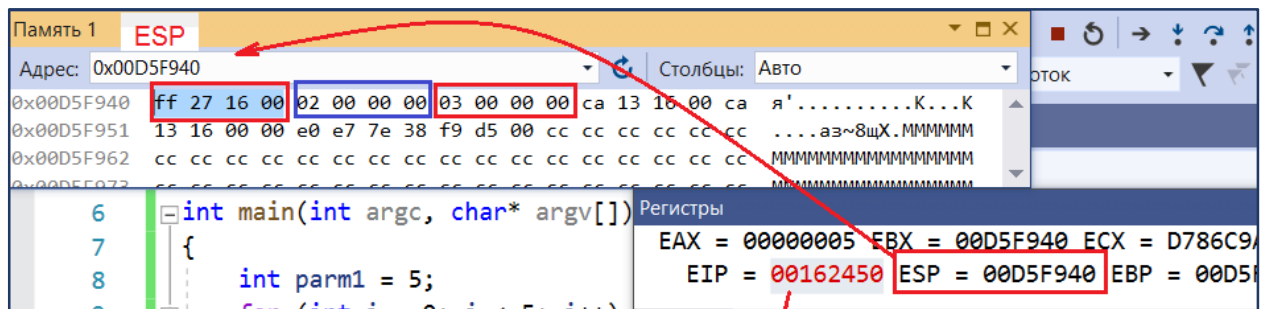
Специальные команды ассемблера для работы со стеком:

push <operand>	pop <operand>
помещает операнд в стек	снимает с вершины стека значение и помещает его в операнд

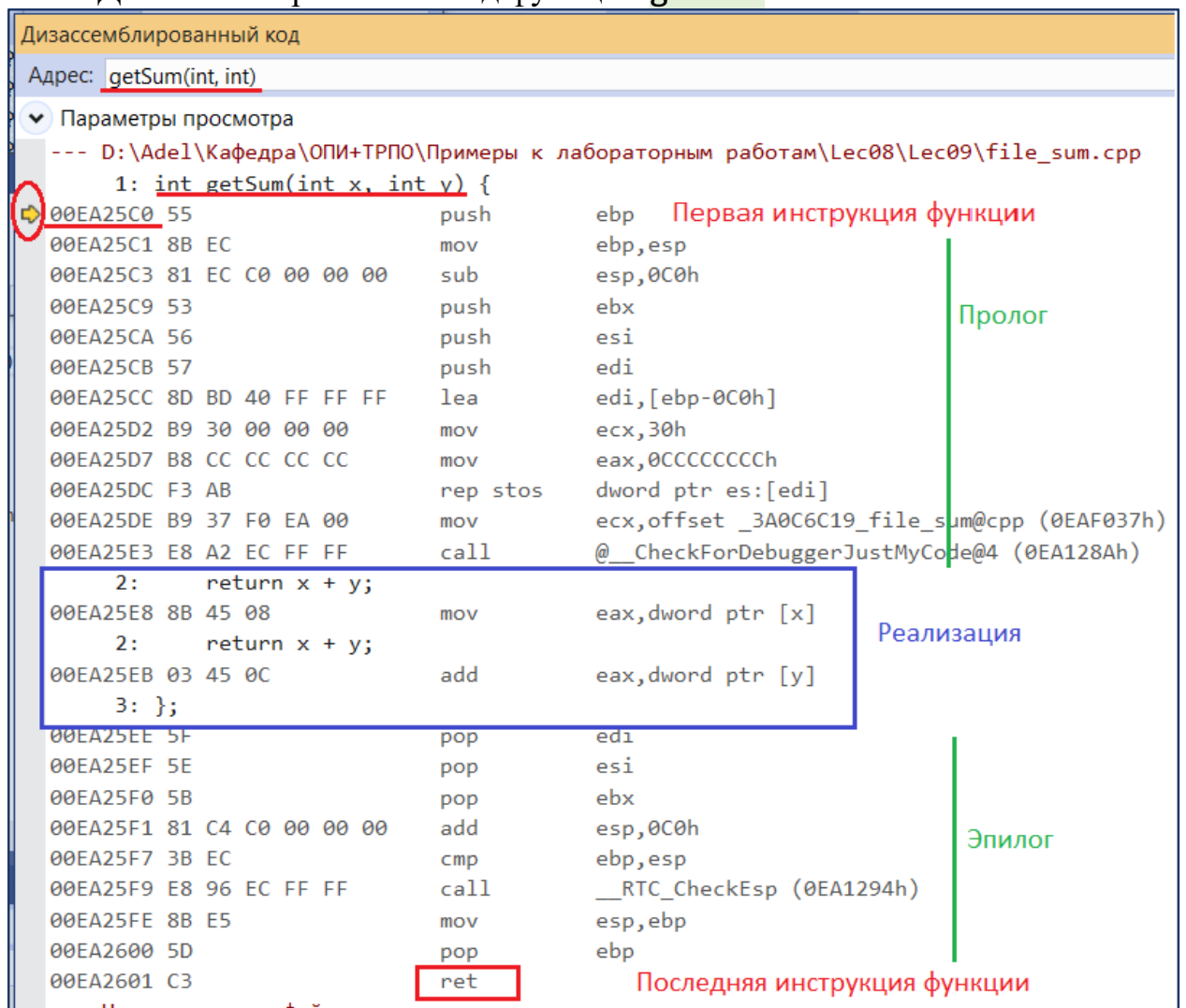
10. В окне «Память» отладчика в поле для ввода «Адрес» вводим имя регистра: **ESP**.

Содержимое памяти по адресу, хранящемуся в регистре ESP (в вершине стека) равно **001627FF** – это адрес точки возврата, т.е. адрес инструкции, следующей за инструкцией **call**.

Далее в стеке лежит целочисленное значение 2 (размером 4 байта – это левый фактический параметр) и в глубине стека лежит целочисленное значение 3 (размером 4 байта – это правый фактический параметр):



Дизассемблированный код функции **getSum**:



Выводы:

- каждый поток имеет свой собственный указатель на текущую инструкцию, и его значение меняется автоматически и всегда актуально. Этот указатель хранится в регистре **EIP**.
- каждый поток имеет свой собственный стек, где хранятся параметры функции, локальные переменные, адрес инструкции, которой будет передано управление после выхода из функции (адрес точки возврата). Адрес стека хранится в регистре **ESP**.
- вызов функций осуществляется с помощью инструкций **call**.
- возврат из функции происходит с помощью инструкции **ret** – последняя выполняемая инструкция в вызываемой функции.

Инструкция **call** помещает в вершину стека (по указателю ESP) адрес точки возврата в вызывающий код (адрес инструкции, следующей за **call**). Затем она обновляет регистр **EIP**, помещая в него адрес вызванного в данный момент кода, и выполнение потока продолжается с этого нового адреса, сохраненного в **EIP**.

Инструкция **ret** снимает с вершины стека, на которую указывает **ESP**, двойное слово (это **DWORD** (4 байта) в ассемблере соответствует типу **int** языка C/C++) и помещает его в регистр EIP. Затем выполнение потока продолжается с адреса, который теперь находится в EIP.