

Основы программной инженерии (ПИ)

Назначение отладчика. Понятие и назначение дизассемблера

План лекции:

- *среда разработки*: назначение и основные возможности отладчика;
- *среда разработки*: понятие и назначение дизассемблера.

1. На прошлых лекциях:

Среда разработки

Интегрированная среда разработки
(integrated development environment – IDE)

набор инструментов для разработки и отладки программ, имеющий общую графическую оболочку, поддерживающую выполнение всех основных функций жизненного цикла разработки программы

Примеры IDE:

Eclipse, Microsoft Visual Studio, NetBeans, Qt Creator, ...

Microsoft Visual Studio – линейка продуктов компании Microsoft, включающих интегрированную среду и другие инструменты для разработки консольных приложений, игр, приложений с графическим интерфейсом, веб-сайтов, веб-приложений, веб-служб как в нативном, так и в управляемом кодах для всех платформ, поддерживаемых Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone .NET Compact Framework и Silverlight. Построена на архитектуре, поддерживающей возможность использования встраиваемых дополнений (англ. Add-Ins) – плагинов от сторонних разработчиков, что позволяет расширять возможности среды разработки.

 Visual Studio 2019	Первый выпуск 1997
разработчик	Microsoft
написана на	C++ и C#
ОС	Microsoft Windows, macOS
последняя версия	Visual Studio 2022 последнее обновление 17.11.5 (08 октября 2024)
сайт	visualstudio.microsoft.com

2. Пример многофайлового проекта

```
lec09.cpp ④ x
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 } file_mul.cpp ④ x lec09.cpp
22 #include <iostream>
23
24 int getSum(int x, int y) {
25     std::cout << "getSum(" << x << "," << y << ") = " ;
26     return x + y;
27 }
```

```
D:\Ade\Кафедра\ОПИ+ТРПО\Примеры к лаборатории
i = 0 getSum(5,5) = 10
i = 1 getSum(5,6) = 11
i = 2 getSum(5,7) = 12
i = 3 getSum(5,8) = 13
i = 4 getSum(5,9) = 14
getMul(2,3) = 6
Для продолжения нажмите любую клавишу . . .
file_sum.cpp ④ x
1 int getSum(int x, int y) {
2     return x + y;
3 };
Обозреватель решений — поиск (C)
Файлы ресурсов
Lec08-2
Ссылки
Внешние зависимости
Исходные файлы
file_lec08-2.cpp
Файлы заголовков
Файлы ресурсов
Lec09
Ссылки
Внешние зависимости
Исходные файлы
file_mul.cpp
file_sum.cpp
lec09.cpp
Файлы заголовков
Файлы ресурсов
```

3. 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 исходном коде программы (программа запускается, отрабатывает, но не дает желаемого результата).

Отладка – процесс запуска и выполнения программы в режиме отладки.

a. 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.

b. Прекращение отладки

Способы остановки отладчика:

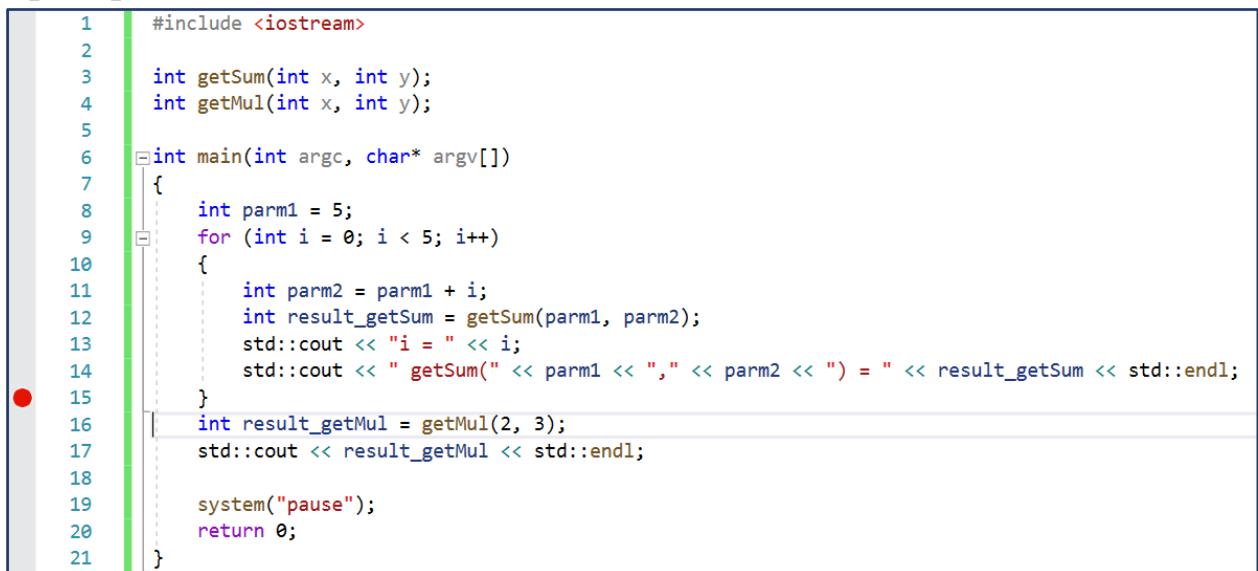
- пункт главного меню Отладка → Остановить отладку;
- комбинация клавиш SHIFT + F5;
- иконка остановки  на панели инструментов.

! Также необходимо закрыть окно консоли.

c. 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 }
```

D:\Адел\Кафедра\ОПИ+ТРПО\Примеры к лабораторным
i = 0 getSum(5,5) = 10

d. Пошаговая отладка

Некоторые возможности управления режимом отладки:

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

e. Проход по коду в отладчике с помощью пошаговых команд

4. Выполнить команду «Продолжить».

Результат:

The screenshot shows the Microsoft Visual Studio debugger interface. The code editor window is titled "Lec09" and contains C++ code. The code defines two functions, `getSum` and `getMul`, and a main function that prints the sum of 5 and each integer from 0 to 5. The debugger's status bar at the bottom right shows the current line of code and its output: "i = 0 getsum(5,5) = 10" and "i = 1 getsum(5,6) = 11". A red arrow icon on the left indicates the current instruction point.

```
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    } ≤ 6 мс прошло
16    int result_getMul = getMul(2, 3);
17    std::cout << result_getMul << std::endl;
18
19    system("pause");
```

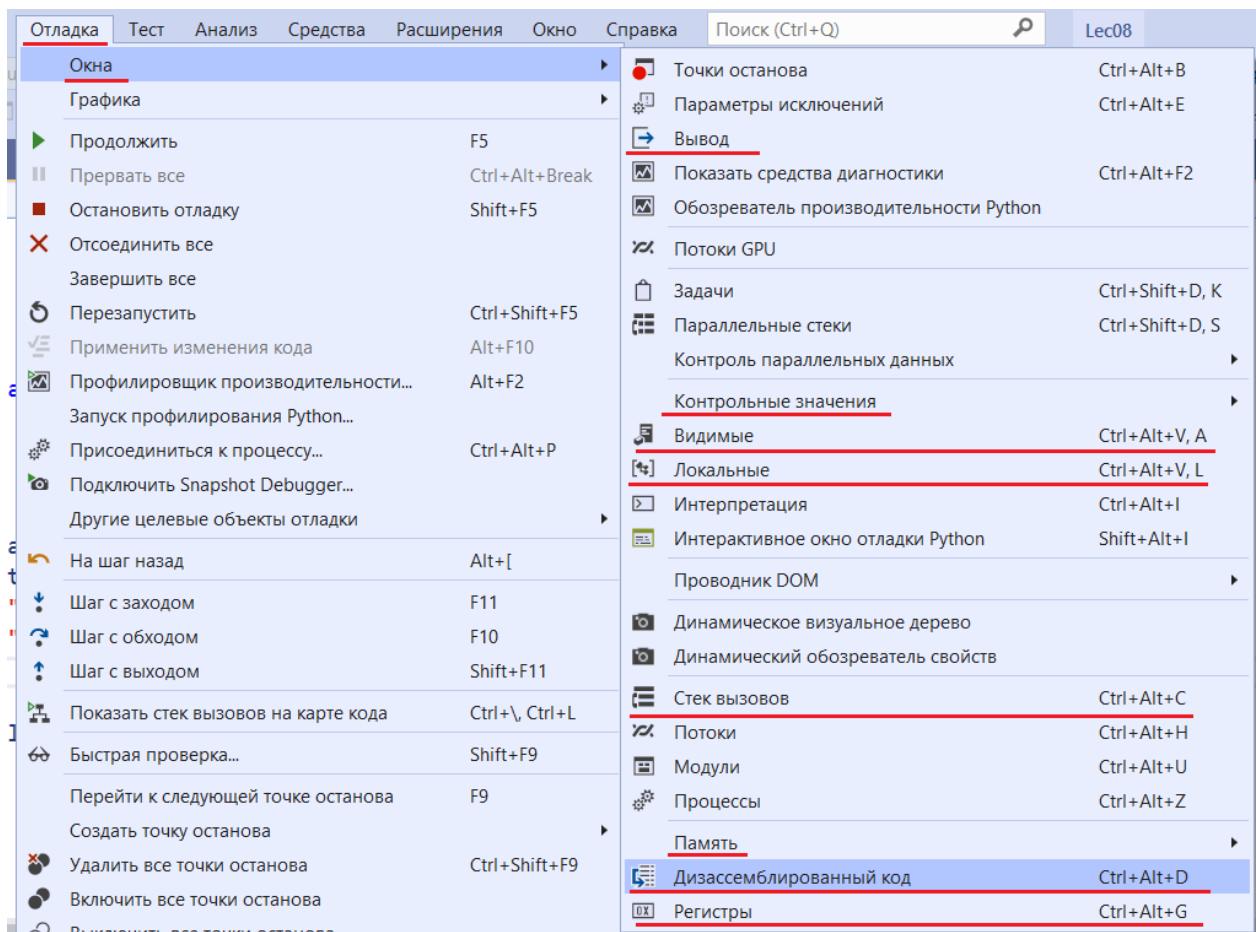
Нажать клавишу **F10** (или выбрать пункт меню Отладка → Шаг с обходом).
Отладчик выполняет инструкции без захода в функции (или методы) в коде приложения.

f. Быстрый перезапуск приложения

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

4. Окна отладчика

Показать и скрыть отладочные окна: меню Отладка → Окна:



Окно «Локальные»

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

The screenshot shows the PyCharm IDE during runtime. The code in main() is as follows:

```
#include <iostream>
int getSum(int x, int y);
int getMul(int x, int y);
int main(int argc, char* argv[])
{
    int parm1 = 5;
    for (int i = 0; i < 5; i++)
    {
        int parm2 = parm1 + i;
        int result_getSum = getSum(parm1, parm2);
        std::cout << "i = " << i;
        std::cout << " getSum(" << parm1 << "," << parm2 << ") = " << result_getSum << std::endl;
    }
    int result_getMul = getMul(2, 3);
    std::cout << result_getMul << std::endl;
    system("pause");
    return 0;
}
```

The output window shows the following results:

```
D:\Adel\Кафедра\ОПИ+ТРПО\Примеры к лабораторным работам\Lec08\Debug\Lec09.exe
i = 0 getSum(5,5) = 10
i = 1 getSum(5,6) = 11
```

The Local Variables window displays the current values of local variables:

Имя	Значение
argc	1
argv	0x00d60530 (0x00d60538 "D:\Adel\Кафедра\ОПИ+ТРПО\Примеры к лабораторным работам\Lec08\Debug\Lec09.exe")
i	1
parm1	5
result_getMul	-858993460

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

6. При наведении указателя мыши на переменную **i** можно просмотреть 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(i + 1, 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 << "," << par
```

Результат:

Локальные	
Поиск (Ctrl+E)	🔍
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    } ≤ 6 мс прошло
16    int result_getMul = getMul(2, 3);
17    std::cout << result_getMul << std::endl;
18
19    system("pause");
20    return 0;
```

120 % Проверка не найдены.

Локальные

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

D:\Adel\Кафедра\ОПИ+ТРПС
i = 0 getSum(5,5) = 10
i = 1 getSum(5,6) = 11
i = 4 getSum(5,9) = 14

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 the code editor with a red breakpoint at line 15. Below it is the 'Visible' window. The code is as follows:

```
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;  ≤ 4 мс прошло
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
21 }
22 }
```

The 'Visible' window contains the following table:

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.

Пример 2. Выполнить следующую последовательность действий.

1. Начать отладку.
2. Установить точку останова на 15-й строке кода.
3. Продолжить отладку до точки останова.
4. Открыть окно «Контрольные значения».
5. Выделить имя переменной `parm1` и перетащить его в окно «Контрольные значения».
6. Изменить значение переменной `parm1` на 10.
7. Продолжить выполнение отладки в пошаговом режиме.

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

```
7 int parm1 = 5;
8
9 for (int i = 0; i < 5; i++)
10 {
11     int parm2 = parm1 + i;
12     int result_getSum(parm1 | 10 ↳parm1, parm2);
13     std::cout << "i = " << i;
14     std::cout << " getSum(" << parm1 << "," <<
15 } ≤ 6 мс прошло
16 int result_getMul = getMul(2, 3);
17 std::cout << result_getMul << std::endl;
18
19 system("pause");
20
21 return 0;
```

A red oval highlights the assignment statement `int result_getSum(parm1 | 10 ↳parm1, parm2);`. The debugger status bar at the bottom left shows "120 %".

The "Control values" window is open at the bottom, showing the following table:

Имя	Значение	Тип
parm1	10	int

A red underline is under the entire row of the table.

Результат в строке 15 (следующая итерация цикла):

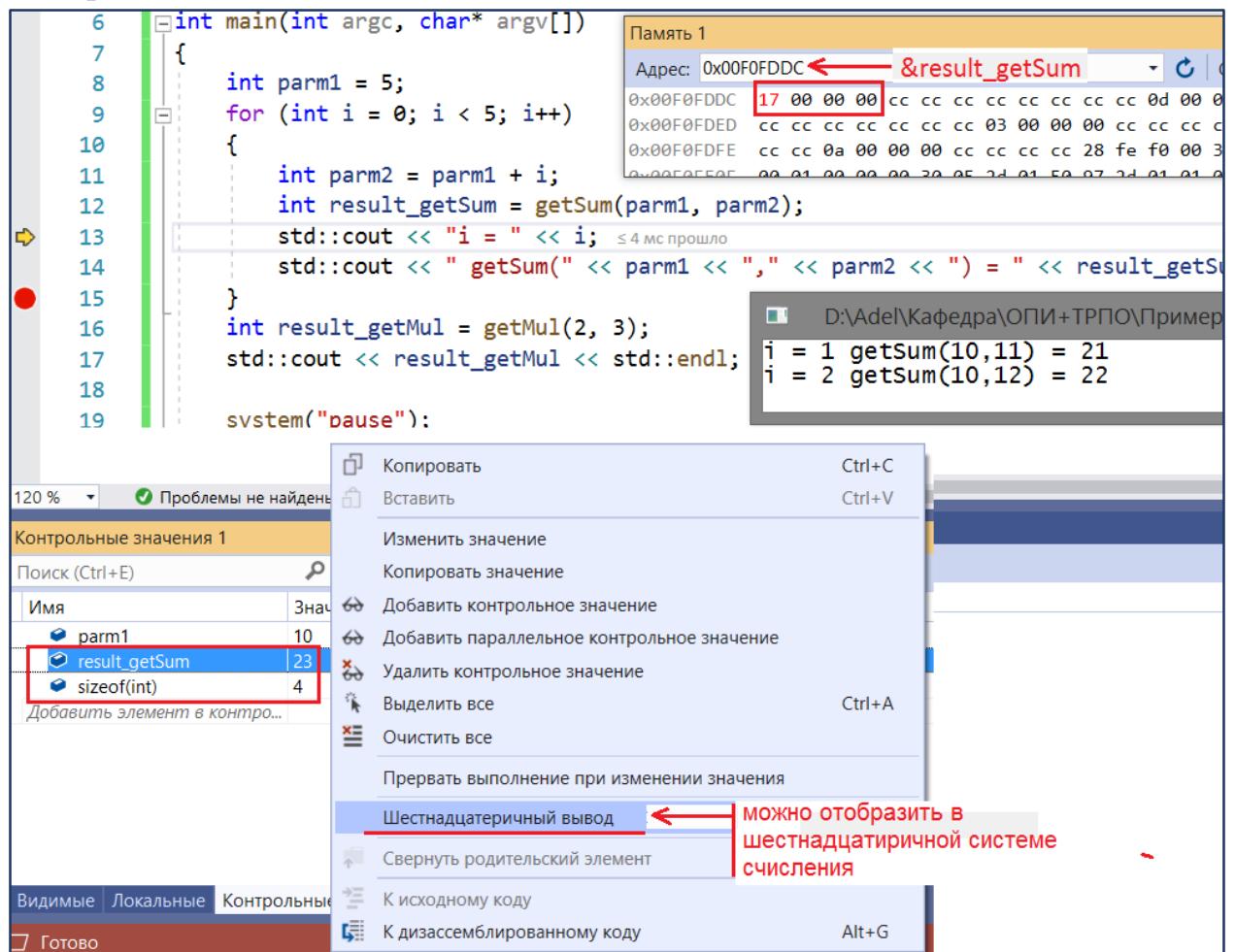
The screenshot shows a C++ IDE interface with the following details:

- Code Editor:** Displays the main function with a cursor at the end of the line "int result_getSum = getSum(parm1, parm2);". A tooltip indicates "≤ 20 мс прошло".
- Output Window:** Shows the console output:

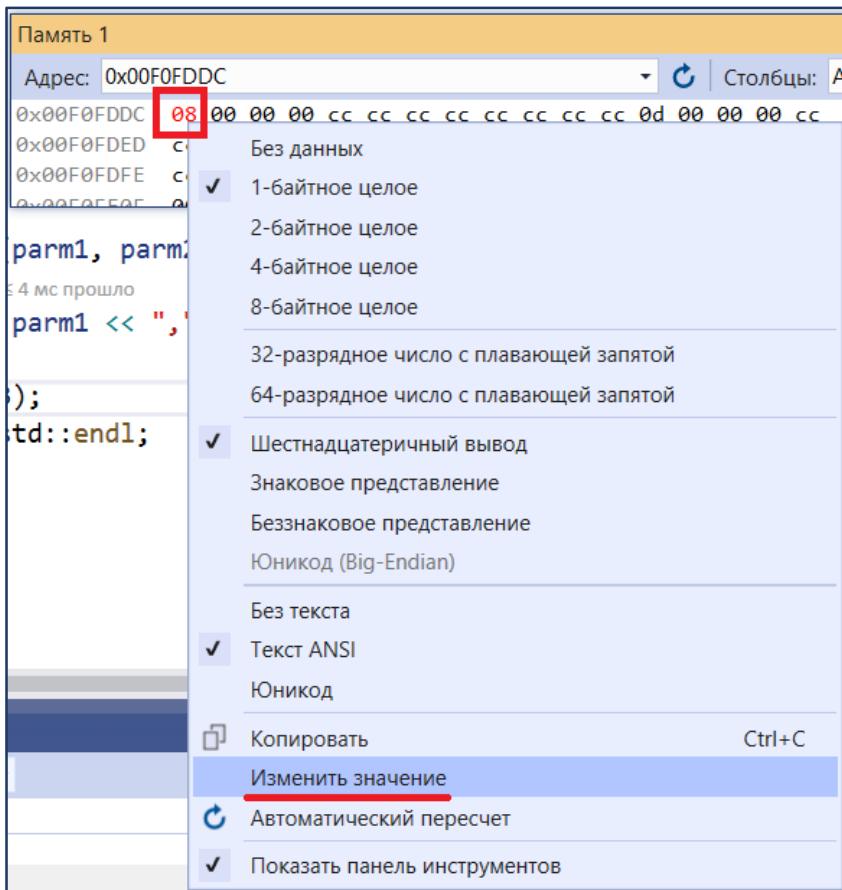
```
i = 0 getSum(5,5) = 10
i = 1 getSum(10,11) = 21
```
- File Path:** The current file is located at D:\Adel\Кафедра\ОПИ+ТРПО\Примеры к лаборат.
- Search Bar:** Contains "120 %", "Проверка не найдены.", and a search icon.
- Control Values Table:** Titled "Контрольные значения 1", it lists the variable "parm1" with value 10 and type int.

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:

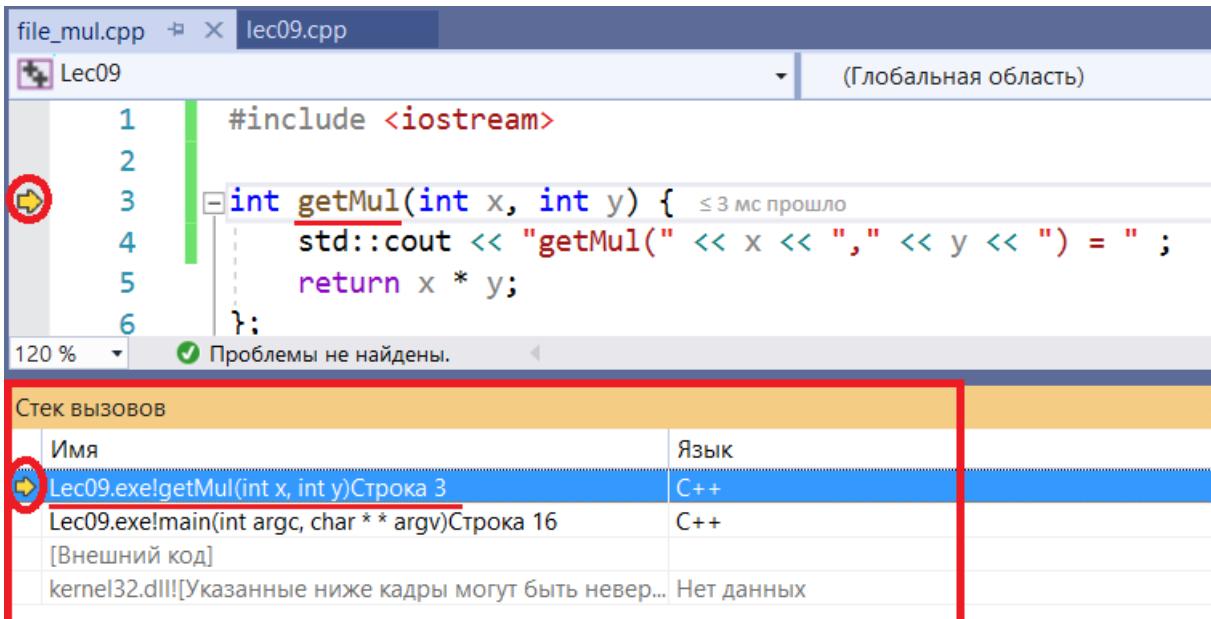


5. Просмотр стека вызовов

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.



Стек вызовов	
Имя	Язык
Lec09.exe!getMul(int x, int y)Строка 3	C++
Lec09.exe!main(int argc, char ** argv)Строка 16	C++
[Внешний код]	
kernel32.dll!GetModuleHandleA()	Нет данных

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).

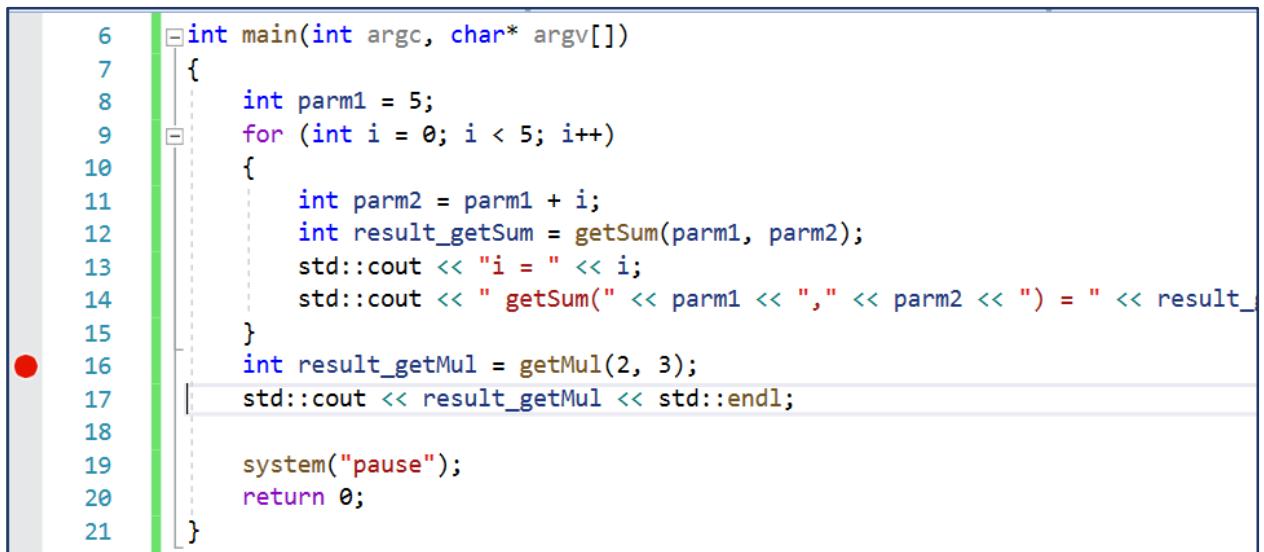
Стек вызовов используется для изучения и анализа потока выполнения приложения.

Окно «Регистры»

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

6. Просмотр дизассемблированного кода в отладчике

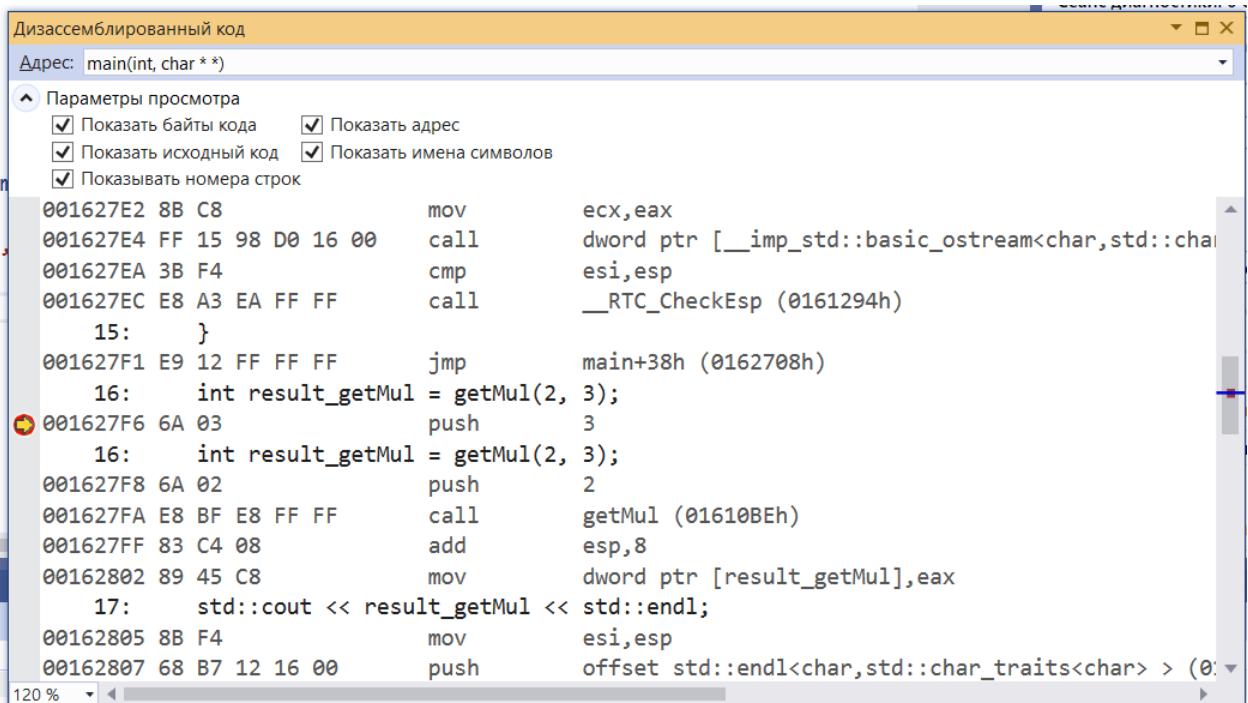
В окне «Дизассемблированный код» отображается код сборки, соответствующий инструкциям, созданным **компилятором**.



```
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
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».



Дизассемблированный код

Адрес: main(int, char **)

Параметры просмотра

Показать байты кода Показать адрес
 Показать исходный код Показать имена символов
 Показывать номера строк

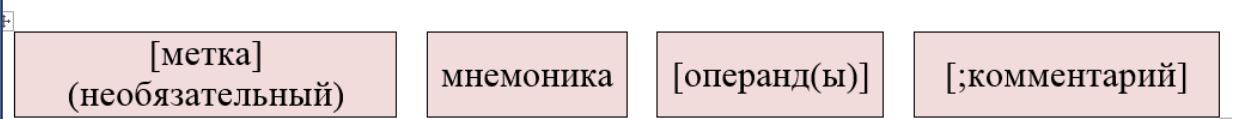
Адрес	Инструкция	Описание
001627E2 8B C8	mov	ecx,eax
001627E4 FF 15 98 D0 16 00	call	dword ptr [_imp_std::basic_ostream<char, std::char_traits<char> >::__imp_ostreambuf_iterator@161294h]
001627EA 3B F4	cmp	esi,esp
001627EC E8 A3 EA FF FF	call	__RTC_CheckEsp (0161294h)
15: }		
001627F1 E9 12 FF FF FF	jmp	main+38h (0162708h)
16: int result_getMul = getMul(2, 3);		
001627F6 6A 03	push	3
16: int result_getMul = getMul(2, 3);		
001627F8 6A 02	push	2
001627FA E8 BF E8 FF FF	call	getMul (01610BEh)
001627FF 83 C4 08	add	esp,8
00162802 89 45 C8	mov	dword ptr [result_getMul],eax
17: std::cout << result_getMul << std::endl;		
00162805 8B F4	mov	esi,esp
00162807 68 B7 12 16 00	push	offset std::endl<char, std::char_traits<char> > (01610BCh)

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 – 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 operandов несколько, то они отделяются друг от друга запятыми.

а. Функции, взгляд на уровне дизассемблера

```
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);
13        std::cout << "i = " << i;
14        std::cout << " getSum(" << parm1
15    }
16    int result_getMul = getMul(2, 3);
17    std::cout << result_getMul << std::endl;
18    system("pause");
19    return 0;
20 }
21 }
```

Регистры
EAX = 00000005 EBX = 00D5F940 ECX = D786C9AF EDX = 0FB54A60 ESI = 00D5F94C EDI = 00D5F948
EIP = 001627F6 ESP = 00D5F94C EBP = 00D5FA58 EFL = 00000246

Дизассемблированный код
Адрес: main(int, char **)

Параметры просмотра

- Показать байты кода
- Показать адрес
- Показать исходный код
- Показать имена символов
- Показывать номера строк

Адрес	Инструкция	Описание
001627E2 8B C8	mov	ecx, eax
001627E4 FF 15 98 D0 16 00	call	dword ptr [_imp_std::basic_ostream<char, std::char_traits<char> > (0161294h)]
001627EA 3B F4	cmp	esi, esp
001627EC E8 A3 EA FF FF	call	__RTC_CheckEsp (0161294h)
15:		
001627F1 E9 12 FF FF FF	jmp	main+38h (0162708h)
16: 001627F6 6A 03	int result_getMul = getMul(2, 3); push 3	
16: 001627F8 6A 02	int result_getMul = getMul(2, 3); push 2	
001627FA E8 BF E8 FF FF	call	getMul (01610BEh) ← Вызов функции getMul
001627FF 83 C4 08	add	esp, 8
00162802 89 45 C8	mov	dword ptr [result_getMul], eax
17: std::cout << result_getMul << std::endl;		
00162805 8B F4	mov	esi, esp
00162807 68 B7 12 16 00	push	offset std::endl<char, std::char_traits<char> > (0161294h)

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:

Регистры

EAX = 00000005 EBX = 00D5F940 ECX = D786C9AF EDX = 0FB54A60 ESI = 00D5F94C EDI = 00D5F948
EIP = 001627F8 ESP = 00D5F948 EBP = 00D5FA58 EFL = 00000246

Дизассемблированный код

Адрес: main(int, char **)

Параметры просмотра

Показать байты кода Показать адрес
Показать исходный код Показать имена символов
Показывать номера строк

```
001627E2 8B C8      mov     ecx,eax
001627E4 FF 15 98 D0 16 00    call    dword ptr [_imp_std::basic_ostream<char, std::char_traits<char> > (01610BEh)
001627EA 3B F4      cmp     esi,esp
001627EC E8 A3 EA FF FF      call    __RTC_CheckEsp (0161294h)
15:   }
001627F1 E9 12 FF FF FF      jmp     main+38h (0162708h)
16:   int result_getMul = getMul(2, 3);
001627F6 6A 03      push    3
16:   int result_getMul = getMul(2, 3);
001627F8 6A 02      push    2      ≤ 3 мс прошло
001627FA E8 BF E8 FF FF      call    getMul (01610BEh)
001627FF 83 C4 08      add    esp,8
00162802 89 45 C8      mov     dword ptr [result_getMul],eax
17:   std::cout << result_getMul << std::endl;
00162805 8B F4      mov     esi,esp
00162807 68 B7 12 16 00      push    offset std::endl<char, std::char_traits<char> > (01610BEh)
```

- 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.

- 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`:

The screenshot shows the WinDbg debugger interface. The assembly window displays the code for the `getMul` function. The instruction at address `00162450` is highlighted with a red box. The assembly code for this instruction is `push`. The registers window shows the current state of the CPU registers. The stack window shows the current state of the stack. The call stack window shows the history of function calls. The Registers window shows the values of the CPU registers. The Dump window shows memory dump information.

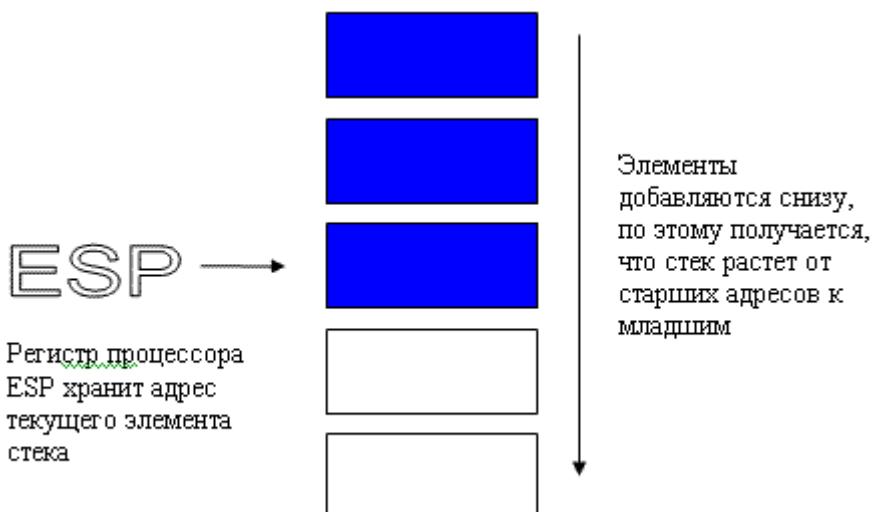
```

    int main(int argc, char* argv[])
    {
        int parm1 = 5;
        for (int i = 0; i < 5; i++)
        {
            int parm2 = parm1 + i;
            int result_getSum = getSum(parm1);
            std::cout << "i = " << i;
            std::cout << " getSum(" << parm1
        }
        int result_getMul = getMul(2, 3);
        std::cout << result_getMul << std::endl;
        system("pause");
        return 0;
    }

```

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 кладь тарелку только поверх стопки. Тарелки из середины стопки недоступны.

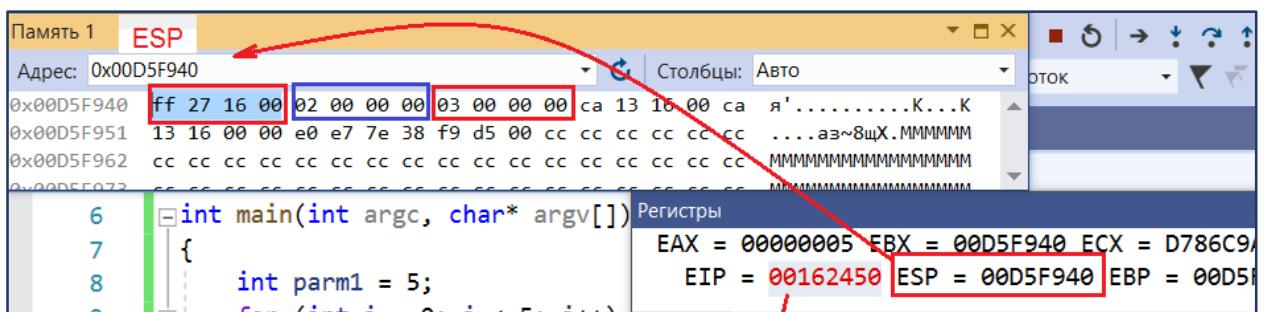
Специальные команды ассемблера для работы со стеком:

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

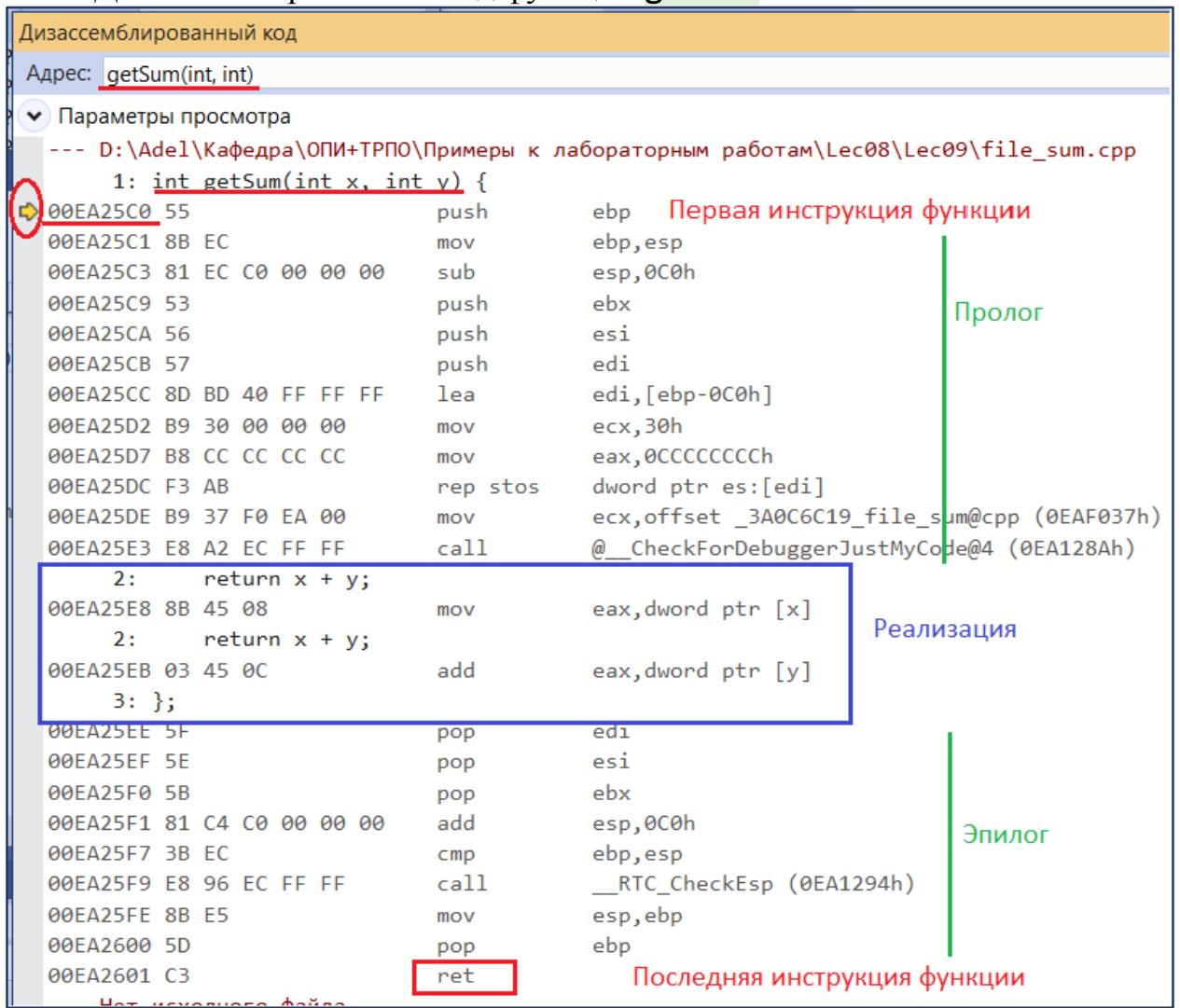
10. В окне «Память» отладчика в поле для ввода «Адрес» вводим имя регистра: **ESP**.

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:

- приложение состоит из одного или нескольких процессов, процесс всегда имеет по крайней мере один поток выполнения, известный как основной поток; поток – это единица выполнения внутри процесса, которая разделяет память и ресурсы с другими потоками того же процесса.
- 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 выполнимая 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**.