



# ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ЛЕКЦИЯ № 7

ПРЕПОДАВАТЕЛЬ: ХУСТОЧКА А.В.



# ОТЛАДКА

Отладка – этап разработки программного обеспечения, на котором обнаруживают, локализуют и исправляют ошибки. Чтобы понять, где возникла ошибка, приходится:

- узнавать текущие значения переменных;
- выяснять, по какому пути выполнялась программа.

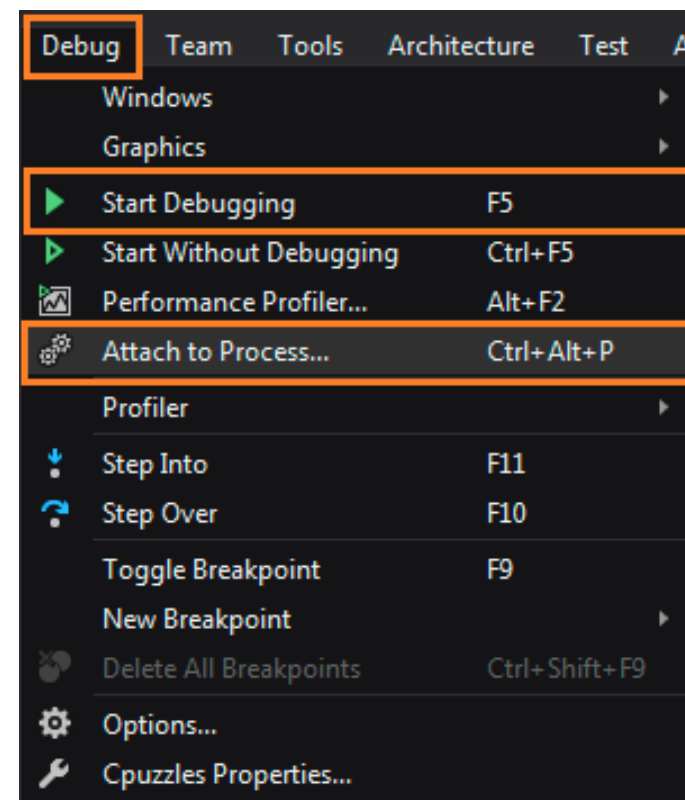
# ТЕХНОЛОГИИ ОТЛАДКИ

Существуют две технологии отладки:

- Использование отладчиков — программ, которые включают в себя интерфейс (пользовательский или консольный) для пошагового выполнения программы: оператор за оператором, функция за функцией, с остановками на некоторых строках исходного кода или при достижении определённого условия.
- Вывод текущего состояния программы с помощью расположенных в критических точках программы операторов вывода — на экран или в файл (создание логов).

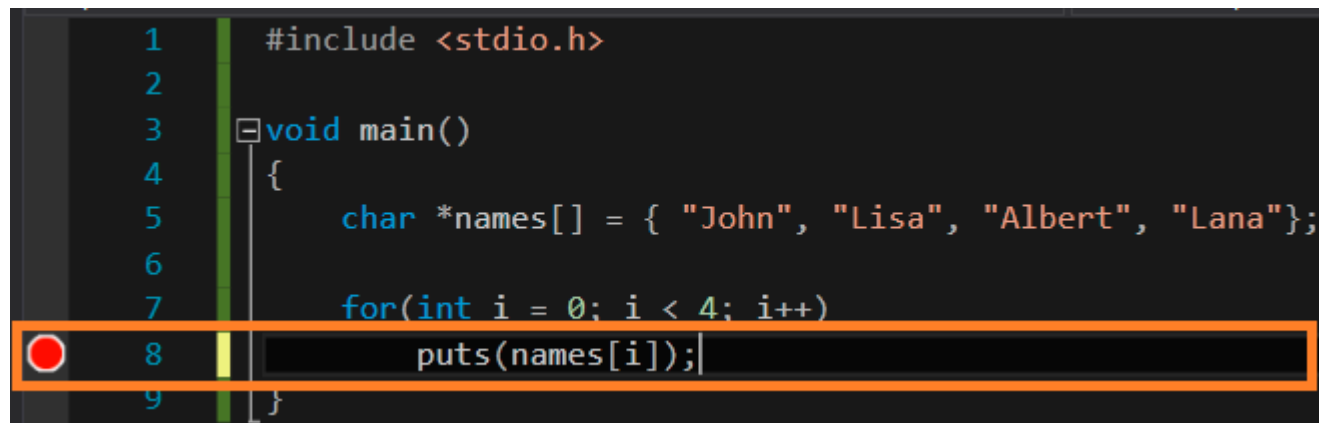
# НАЧАЛО ОТЛАДКИ

- Нажать **F5**.
- Отладка начнется если стоят точки останова (**breakpoints**)



# ТОЧКА ОСТАНОВА

- Точки останова используются чтобы показать , где отладчику необходимо остановиться.
- Точка ставится кликом на сайдбар слева от исходного кода, либо нажатием на **F9**.
- Точки останова обычно ставятся там, где есть сомнения в корректности кода.



```
1  #include <stdio.h>
2
3  void main()
4  {
5      char *names[] = { "John", "Lisa", "Albert", "Lana"};
6
7      for(int i = 0; i < 4; i++)
8          puts(names[i]);
9  }
```

The image shows a code editor with a dark background. A red circle, representing a breakpoint, is set on the left margin of line 8, which contains the code `puts(names[i]);`. The entire line 8 is highlighted with an orange rectangular border. The code is a C program that includes `<stdio.h>`, defines a `main` function, and prints the names of four people: John, Lisa, Albert, and Lana.

# ОТЛАДКА С ИСПОЛЬЗОВАНИЕМ ТОЧЕК ОСТАНОВА (DEBUGGING WITH BREAKPOINTS)

- **Перешагнуть (Step Over) F10** – автоматически выполняет блок кода под курсором.
- **Зайти (Step Into) F11** – заходит в блок кода под курсором.
- **Выйти (Step Out) Shift + F11** - выходит из текущего блока.
- **Продолжить (Continue) F5** - переходит к следующей точки останова.

```
1  #include <stdio.h>
2  void function()
3  {
4      puts("Break Point in function()");
5  }
6
7  void main()
8  {
9      int num = 0;
10     function();
11     puts("We are in main()");
12 }
```

Нажать F10

```
1  #include <stdio.h>
2  void function()
3  {
4      puts("Break Point in function()");
5  }
6
7  void main()
8  {
9      int num = 0;
10     function();
11     puts("We are in main()");
12 }
```

Нажать F11

# УСЛОВНЫЕ ОСТАНОВКИ (CONDITIONAL BREAKPOINT)

- В циклах может обрабатываться большое количество данных.
- Условная остановка нужна чтобы остановить выполнение кода в нужном месте



Location: Source1.cpp, Line: 8, Must match source

☒ Conditions

Conditional Expression    ▾    Is true    ▾    **i==2** ×    Saved  
[Add condition](#)

Location: Source1.cpp, Line: 8, Must match source

☒ Conditions

Conditional Expression    ▾    Is true    ▾    **strcmp(names[i], "Albert") == 0** ×    Saved  
[Add condition](#)

```
{  
    char *names[] = { "John",  
    for(int i = 0; i < 4; i++)  
        puts(names[i]);  
}
```

```
{  
    char *names[] = { "John", "Lisa", "Albert",  
    for(int i = 0; i < 4; i++)  
        puts(names[i]);  
}
```

names[i] 0x00f76b58 "Albert"



# КОЛИЧЕСТВО ОСТАНОВОК (BREAKPOINT HIT COUNT)

- Отслеживание сколько остановок отладчик сделает на конкретной точке останова

The screenshot shows the Visual Studio IDE with a C++ program and a breakpoint set on line 8. The program code is as follows:

```
4 {  
5     char *names[] = { "John", "Lisa", "Albert", "Lana"};  
6  
7     for(int i = 0; i < 4; i++)  
8         puts(names[i]);  
}
```

The breakpoint settings window is open, showing the location as `Source1.cpp, Line: 8, Must match source`. The **Conditions** checkbox is checked, and the **Hit Count** is set to `3`. The current hit count is `3`, and the `Reset` button is visible.

The **Autos** window shows the current state of variables:

Name	Value	Type
i	2	int
names	0x002df8ec {0x00f76b30 "John", 0x00f76b50 "Lisa", 0x00f76b58 "Albert", 0x00f76b5f "Lana"}	char *[4]
names[i]	0x00f76b58 "Albert"	char *

The **Breakpoints** window shows the following breakpoint:

Name	Labels	Condition	Hit Count
Source1.cpp, line 8	NAMES	(no condition)	when hit count is equal to 3 (currently 3)

# ПОДСКАЗКИ (DATA TIP)

- Можно через подсказки менять значения

```
#include <stdio.h>

void main()
{
    char *names[] = { "John", "Lisa", "Albert", "Lana"};

    for(int i = 0; i < 10; i++)
        puts(names[i]);
}
```

≤ 2ms elapsed

**i | 4**

(names)[0]	↗ 0x00366b30	"John"
(names)[1]	↗ 0x00366b50	"Lisa"
(names)[2]	↗ 0x00366b58	"Albert"
(names)[3]	↗ 0x00366b60	"Lana"

**names[i] | 0xffffffff <Error reading characters of string.>**

# ОКНО ПРОСМОТРА ДАННЫХ (WATCH WINDOWS)

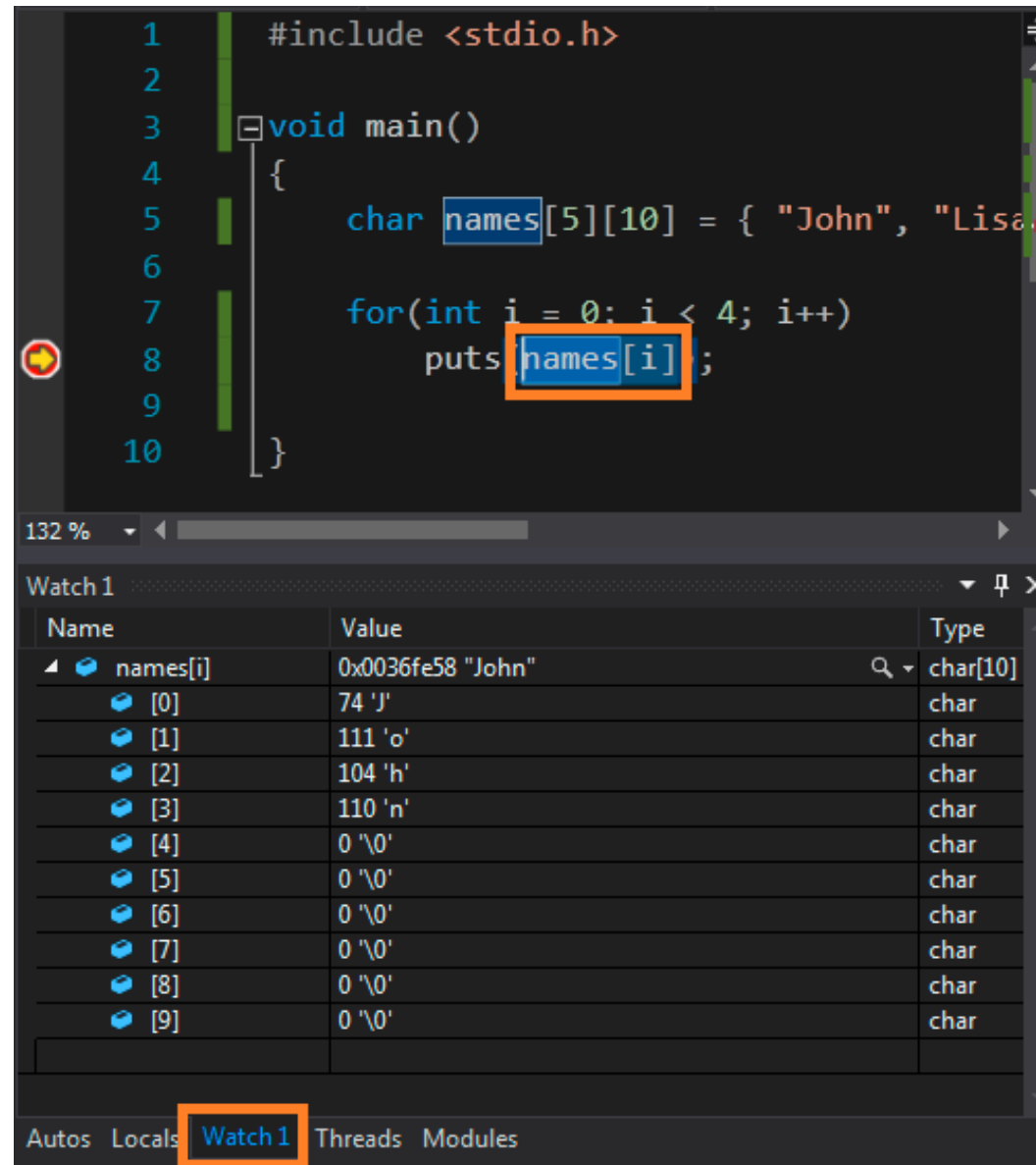
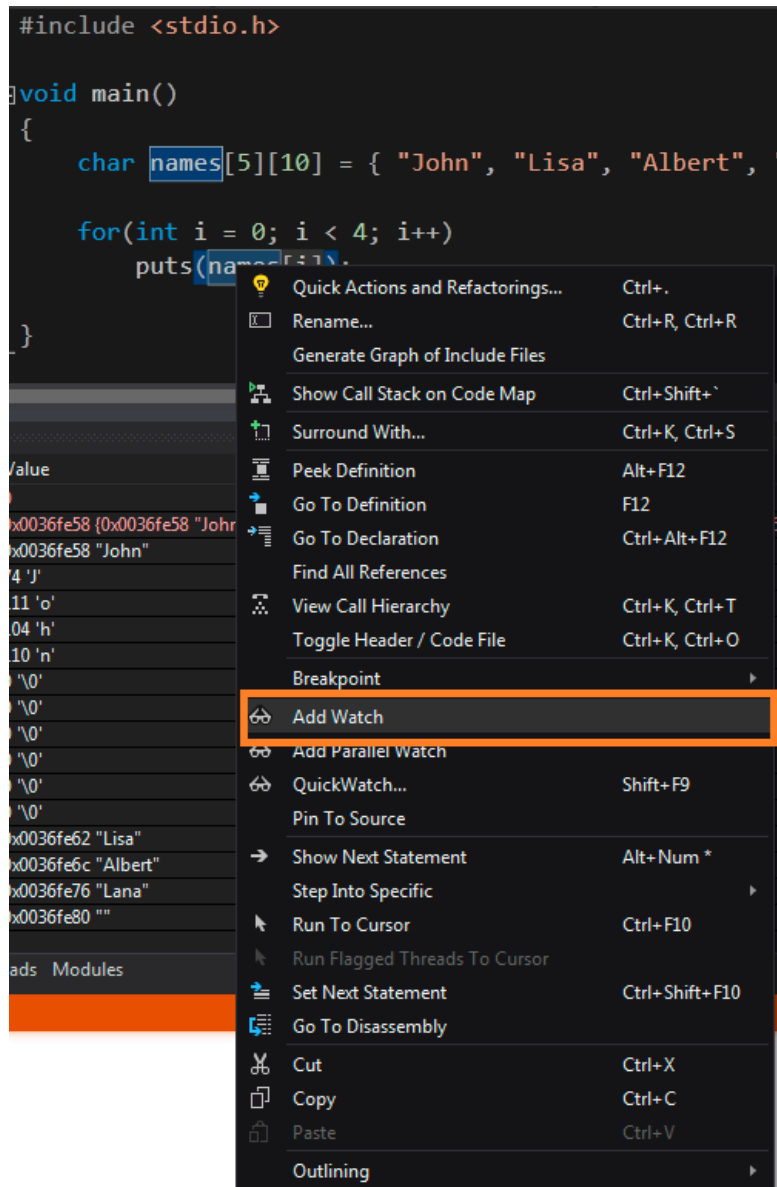
```
1  #include <stdio.h>
2
3  void main()
4  {
5      char names[5][10] = { "John", "Lisa", "Albert", "Lana"};
6
7      for(int i = 0; i < 4; i++)
8          puts(names[i]);
9  }
```

132 %

Locals

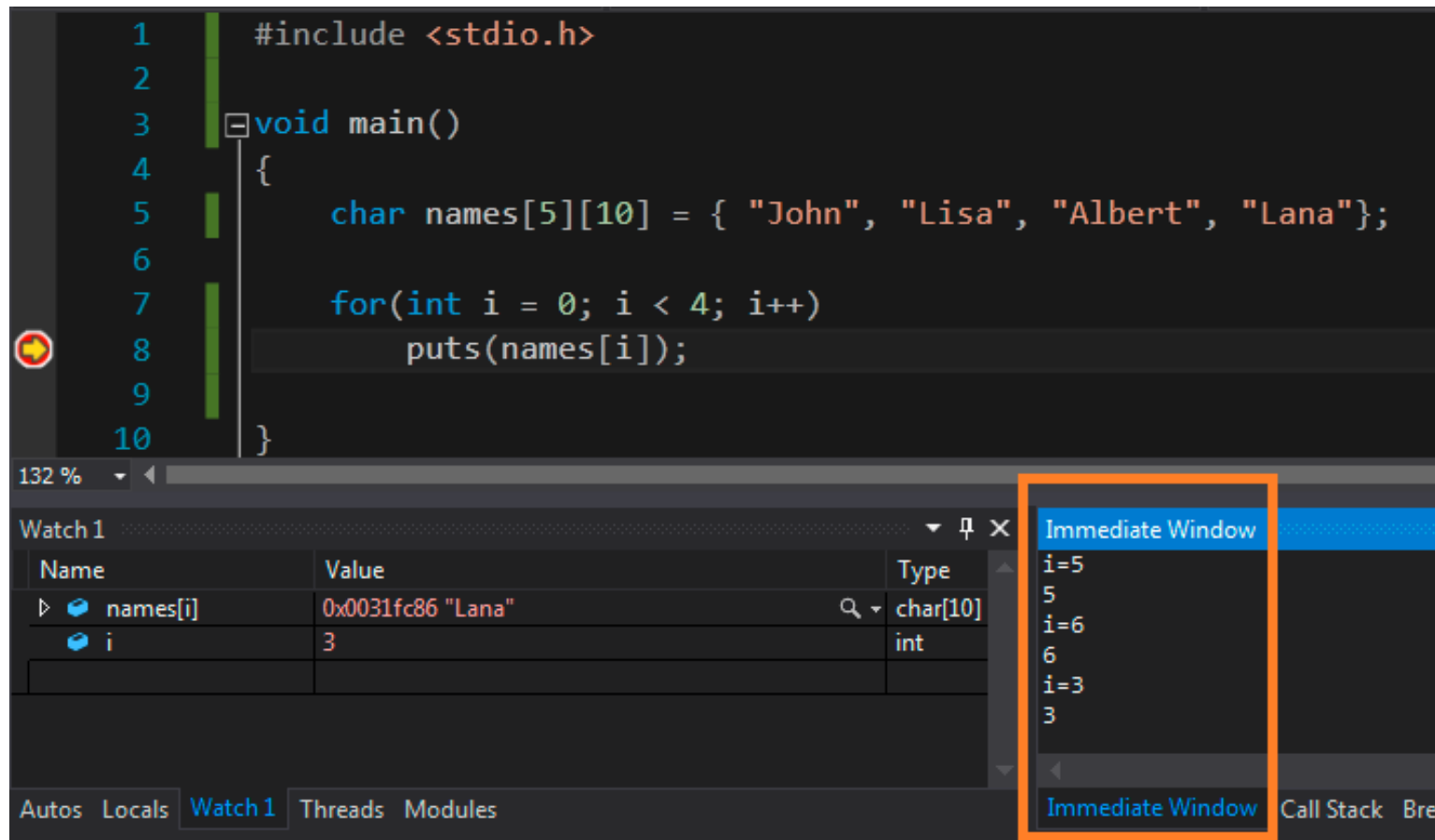
Name	Value	Type
i	0	int
names	0x0036fe58 {0x0036fe58 "John", 0x0036fe62 "Lisa", 0x0036fe6c "Albert", 0x0036fe76 "Lana", 0x0036fe80 ""}	char[5][10]
[0]	0x0036fe58 "John"	char[10]
[0]	74 'J'	char
[1]	111 'o'	char
[2]	104 'h'	char
[3]	110 'n'	char
[4]	0 '\0'	char
[5]	0 '\0'	char
[6]	0 '\0'	char
[7]	0 '\0'	char
[8]	0 '\0'	char
[9]	0 '\0'	char
[1]	0x0036fe62 "Lisa"	char[10]
[2]	0x0036fe6c "Albert"	char[10]
[3]	0x0036fe76 "Lana"	char[10]
[4]	0x0036fe80 ""	char[10]

Autos **Locals** Threads Modules



# ОПЕРАТИВНЫЕ ИЗМЕНЕНИЯ (IMMEDIATE WINDOW)

- **Debug > Window > Immediate Window**
- Позволяет задавать значения выражений/переменных во время отладки



# ПРОСМОТР ПАМЯТИ

Память 1

Адрес: 0x0000004F3D18D860

Столбцы: Авто

0x0000004F3D18D860	cd cd cd cd cd cd cd cd cd fd fd fd fd 00 00 00 00 00 00 00 00 00 00 cd 3a b7 86 00 13 00 80 00 00 00 00 00	ННННННННННээээ.....
0x0000004F3D18D885	00 00	.....
0x0000004F3D18D8A0	00 00	.....

Память 1 | Потоки

example.cpp

example (Глобальная область) main(int argc, char \* argv[])

```
366 int main(int argc, char* argv[])
367 {
368     char* str = (char*)malloc(sizeof(char) * 10);
369     strcpy(str, "meow");
370     return 0;
```

str 0x0000004F3d18d860 "ННННННННННээээ"

Память 1

Адрес: 0x0000004F3D18D860

Столбцы: Авто

0x0000004F3D18D860	6d 65 6f 77 00 cd cd cd cd cd fd fd fd fd 00 00 00 00 00 00 00 00 00 00 cd 3a b7 86 00 13 00 80 00 00 00 00 00	meow.ННННННННННээээ.....
0x0000004F3D18D885	00 00	.....
0x0000004F3D18D8A0	00 00	.....

Память 1 | Потоки

example.cpp




example (Глобальная область) main(int argc, char \* argv[])

```
366 int main(int argc, char* argv[])
367 {
368     char* str = (char*)malloc(sizeof(char) * 10);
369     strcpy(str, "meow");
370     return 0;
```

str 0x0000004F3d18d860 "meow"


# ПЕРЕКЛЮЧЕНИЕ МЕЖДУ ПОТОКАМИ

Потоки

Поиск:   Поиск в стеке вызовов   Группировать по: Идентификатор процес

	Идентификатор	Идентификатор управления	Категория	Имя	Размещение
^ Идентификатор процесса: 0x000013AC (2 потока)					
▼	0x00001E70	0x00	Основной поток	Основной поток	▼ example.exe!main
▼	0x000017D4	0x00	Рабочий поток	ntdll.dll!DbgUiRemoteBreakin()	▼ ntdll.dll!RtlUserThreadStart


Память 1 Потоки

example.cpp 

example (Глобальная область)

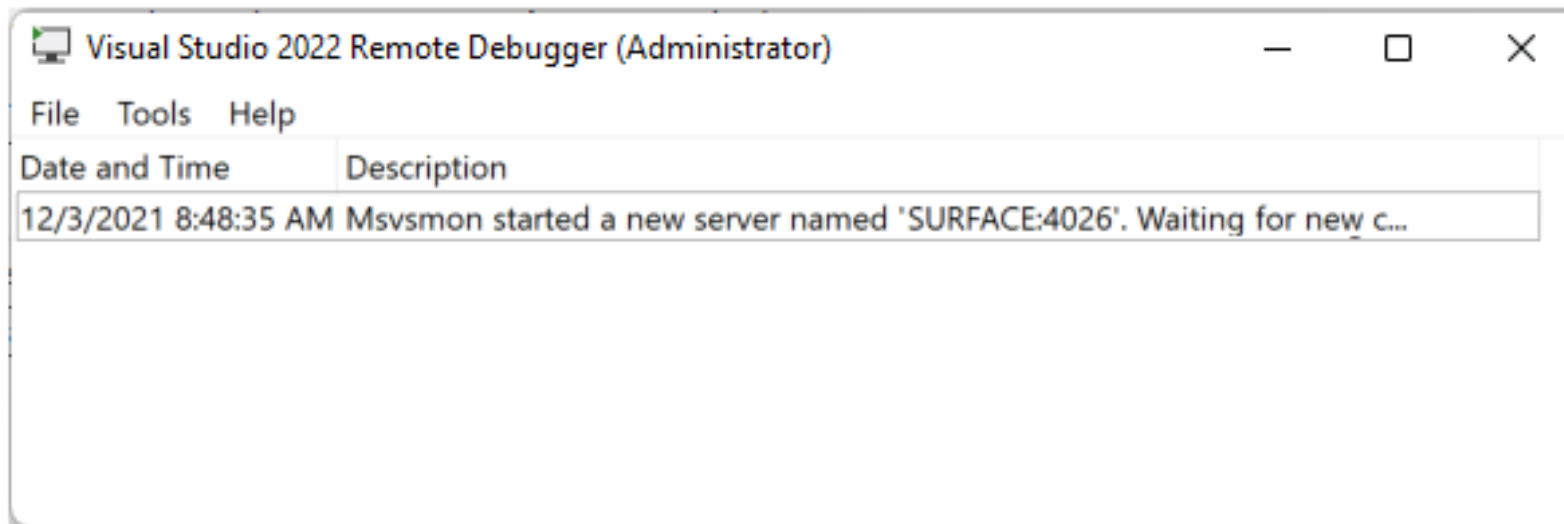
```
366 int main(int argc, char* argv[])
367 {
368     char* str = (char*)malloc(sizeof(char) * 10);
369     strcpy(str, "meow");
370     return 0;
```

1 мс прошло

 str - 0x0000004f3d18d860 "meow"

# УДАЛЁННАЯ ОТЛАДКА

- Msvsmon.exe (скачать можно тут: <https://learn.microsoft.com/ru-ru/visualstudio/debugger/remote-debugging?view=vs-2022> )





# КОНСОЛЬНЫЙ ОТЛАДЧИК

- **GDB** – GNU Debugger — переносимый отладчик проекта GNU, который работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования, включая Си, С++, Free Pascal, FreeBASIC, Ada, Фортран и Rust. GDB — свободное программное обеспечение, распространяемое по лицензии GPL.
- Известные команды: <http://server.179.ru/tasks/gdb/>

```
hello.c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello, world!\n");
6
7      return 0;
8  }
9
10
11
12
13
child process 9054 In: main          Line: 5    PC: 0x8048395
This GDB was configured as "i486-slackware-linux"...
(gdb) b main
Breakpoint 1 at 0x8048395: file hello.c, line 5.
(gdb) r
Starting program: /home/beej/hello

Breakpoint 1, main () at hello.c:5
(gdb) █
```

```
(gdb) r
Starting program: /home/sree/debugging/test

Breakpoint 1, main () at test.c:8
8          int a = x;
(gdb) p x
$1 = -7904
(gdb) p a
$2 = 32767
(gdb) n
9          int b = x;
(gdb) p x
$3 = -7904
(gdb) p a
$4 = -7904
(gdb) p b
$5 = 0
(gdb) n
10         int c = a + b;
(gdb) p x
$6 = -7904
(gdb) p a
$7 = -7904
(gdb) p b
$8 = -7904
(gdb) p c
$9 = 0
(gdb) n
11         printf("%d\n", c);
(gdb) p x
$10 = -7904
(gdb) p a
$11 = -7904
(gdb) p b
$12 = -7904
(gdb) p c
$13 = -15808
(gdb) n
-15808
```

KD 'net:port=50005,key=\*\*\*\*', Default Connection - WinDbg (1.0.2001.02001)

File Home View Breakpoints Time Travel Model Scripting Notes Command Memory Source

Command Watch Locals Registers Memory Stack Disassembly Threads Breakpoints Logs Notes Timelines Layouts Reset Windows Window Layout Workspace

PlugAndPlayDeviceTree.js OK

```
1 // PlugAndPlayDeviceTree.js
2 // An ES6 generator function
3 // Copyright (C) Microsoft A
4
5 function *filterDevices(deviceNode)
6 {
7     // If the device instance
8     if (deviceNode.InstancePa
9     {
10         yield deviceNode;
11     }
12
13     // Recursively invoke the
14     for (var childNode of deviceNode.Children)
15     {
16         yield* filterDevices(childNode);
17     }
18 }
```

Symbol search path is: srv\*;E:\Data1\Vibrani  
Executable search path is:  
Windows 10 Kernel Version 19041 MP (4 procs)  
Product: WinNt, suite: TerminalServer Single  
19041.1.amd64fre.vb\_release.191206-1406  
Machine Name:  
Kernel base = 0xfffff804`51e00000 PsLoadedMo  
Debug session time: Wed Jan 15 14:44:08.995  
System Uptime: 0 days 0:24:49.831  
Break instruction exception - code 80000003  
\*\*\*\*\*  
\* You are seeing this message because you  
\* CTRL+C (if you run console kernel de

0: kd>

Address: @\$scopeip

Notes

The new debugger!

Registers

Name	Value
User	
rax	0x0000000000000000
rbx	0xffffd387f0e130
rcx	0x0000000000000000
rdx	0x0000003ac0000000
rsi	0x0000000000000000
edi	0xfffff901507111

Disassembly

Address: @\$scopeip

☒ Follow current instruction

fffff804`521a56d5 cc	int	3
fffff804`521a56d6 cc	int	3
fffff804`521a56d7 cc	int	3
fffff804`521a56d8 0f1f840000000000	nop	dword ptr [rax+rax]
nt!DbgBreakPointWithStatus:		
fffff804`521a56e0 cc	int	3

Stack

Frame Index	Name
[0x0]	nt!DbgBreakPointWithStatus
[0x1]	kdnic!TXTransmitQueuedSends + 0x125
[0x2]	kdnic!TXSendCompleteDpc + 0x149
[0x3]	nt!KiProcessExpiredTimerList + 0x146
[0x4]	nt!KiExpireTimerTable + 0x1a7