

对VS2022调试工具基本 使用方法的学习

班级：10071706

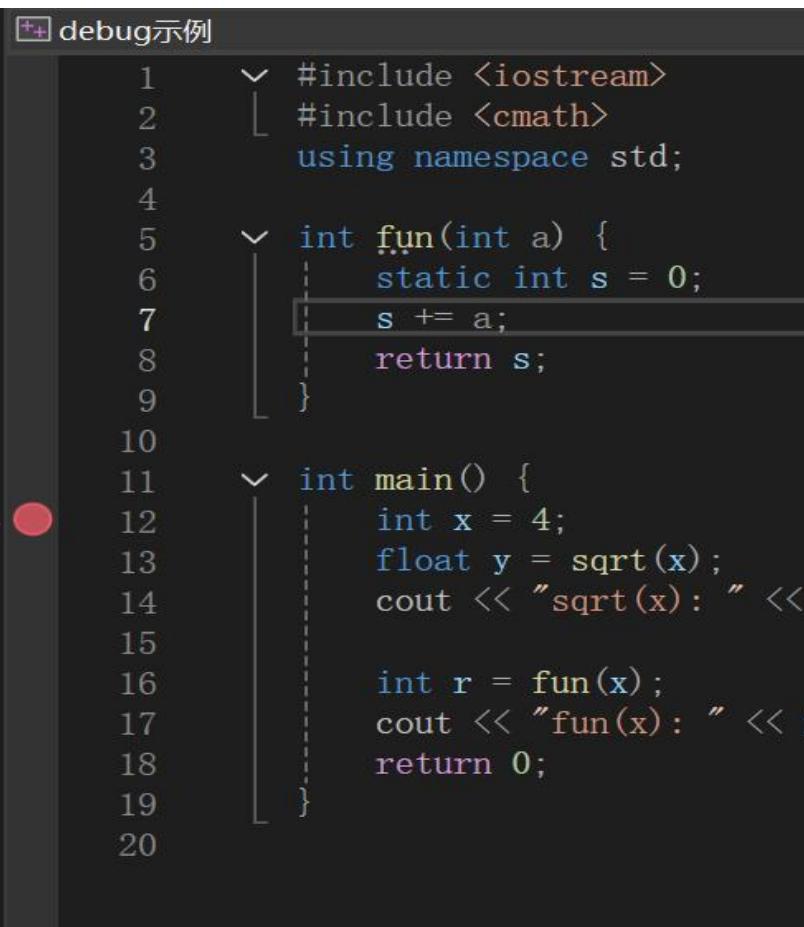
学号：2452654

姓名：郭炫君

日期：2025. 6. 1

1.1 开始调试（知识点1.1）

先在想要调试的代码位置设置断点，开始调试时就会运行到断点处。



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

int fun(int a) {
    static int s = 0;
    s += a;
    return s;
}

int main() {
    int x = 4;
    float y = sqrt(x);
    cout << "sqrt(x): " << y << endl;

    int r = fun(x);
    cout << "fun(x): " << r << endl;
    return 0;
}
```

第一部分构造的源代码

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

int fun(int a) {
    static int s = 0;
    s += a;
    return s;
}

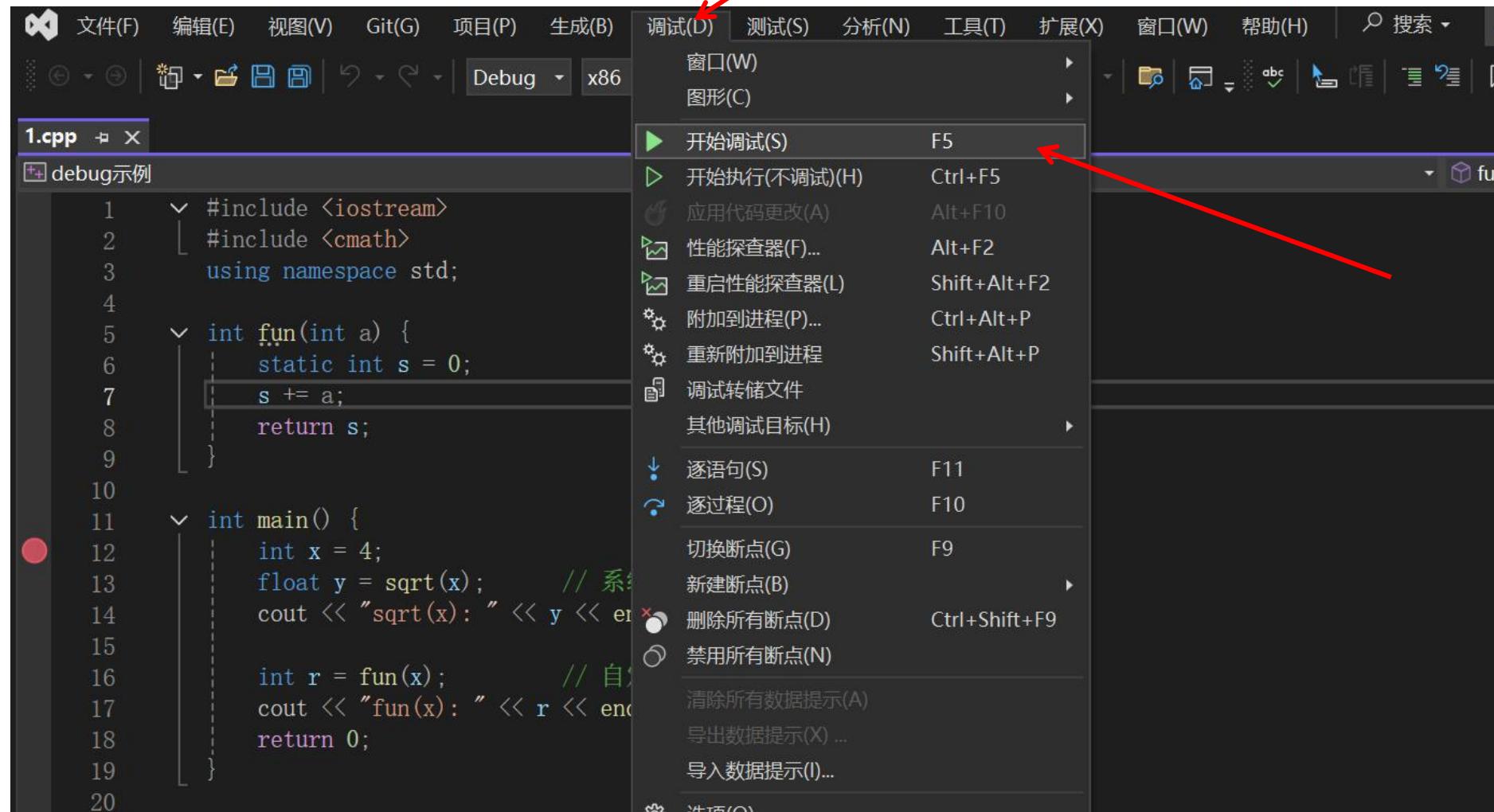
int main() {
    int x = 4;
    float y = sqrt(x); // 系统函数
    cout << "sqrt(x): " << y << endl;

    int r = fun(x); // 自定义函数
    cout << "fun(x): " << r << endl;
    return 0;
}
```

1.1 开始调试（知识点1.1）

方法1：在菜单栏选择“调试”，在出现的菜单中选择“开始调试”

方法2：按F5



1.1 开始调试（知识点1.1）

进入调试模式

The screenshot shows a debugger interface with a dark theme. At the top, there's a code editor window titled "1.cpp" with the file content:

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 int fun(int a) {
6     static int s = 0;
7     s += a;
8     return s;
9 }
10
11 int main() {
12     int x = 4;
13     float y = sqrt(x); // 系统函数
14     cout << "sqrt(x): " << y << endl;
15
16     int r = fun(x); // 自定义函数
17     cout << "fun(x): " << r << endl;
18
19 }
20
```

The code editor has a "debug示例" tab open. Below the code editor is a toolbar with various icons. The bottom part of the interface contains a search bar labeled "搜索(Ctrl+E)" and a table showing variable values:

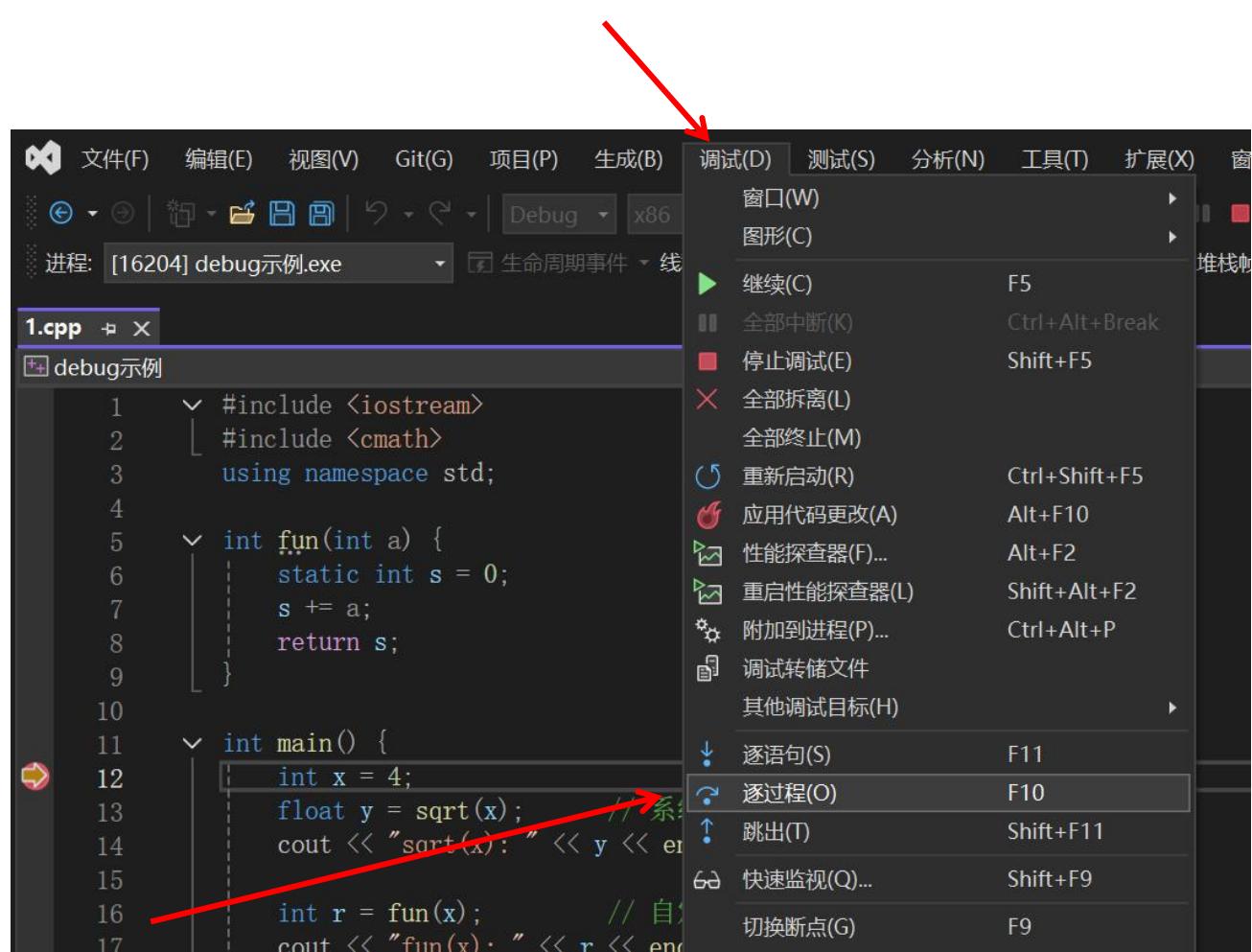
名称	值
x	-858993460

At the very bottom, there are tabs for "自动窗口", "局部变量", and "监视 1", along with a red button labeled "就绪".

1.2 在函数中单步执行语句（知识点1.2，1.3）

1.2.1 每个语句单步执行，并不进入函数内部

在菜单栏选择“调试”，在出现的菜单中选择“逐过程”，或者直接按F10，在遇到系统函数时，会直接一步完成，不进入内部。



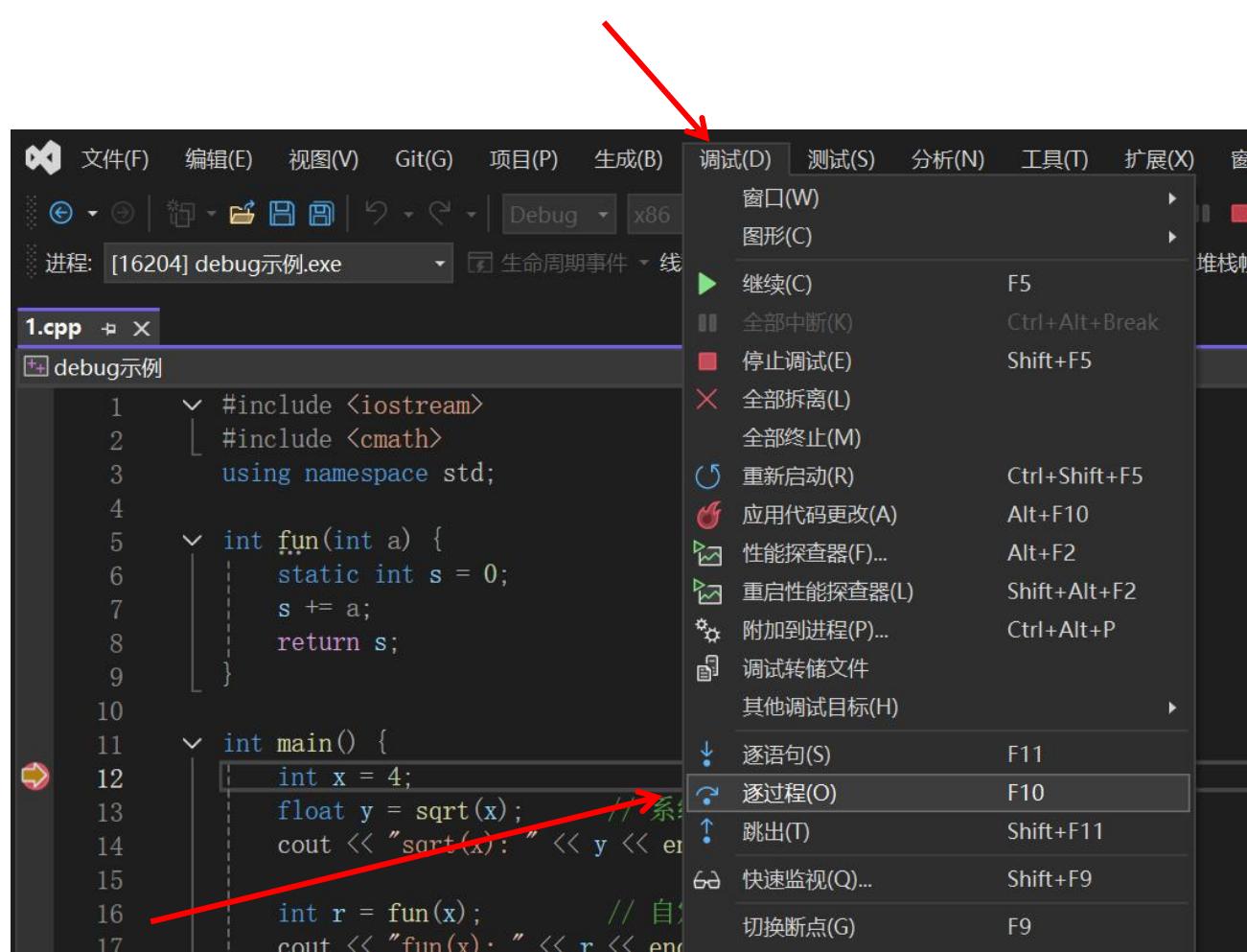
```
10
11     int x = 4;
12     float y = sqrt(x); // 系统函数 已用时间 <= 1ms
13     cout << "sqrt(x): " << y << endl;
14
15     int r = fun(x); // 自定义函数
16     cout << "fun(x): " << r << endl;
17 }
```

```
4
5     int fun(int a) {
6         static int s = 0;
7         s += a;
8         return s;
9     }
10
11     int main() {
12         int x = 4;
13         float y = sqrt(x); // 系统函数
14         cout << "sqrt(x): " << y << endl; // 已用时间 <= 1ms
15
16         int r = fun(x); // 自定义函数
17         cout << "fun(x): " << r << endl;
18     }
19
20 }
```

1.2 在函数中单步执行语句（知识点1.2，1.5）

1.2.1 每个语句单步执行，并不进入函数内部

在菜单栏选择“调试”，在出现的菜单中选择“逐过程”，或者直接按F10，在遇到自定义函数时，会直接一步完成，不进入内部。



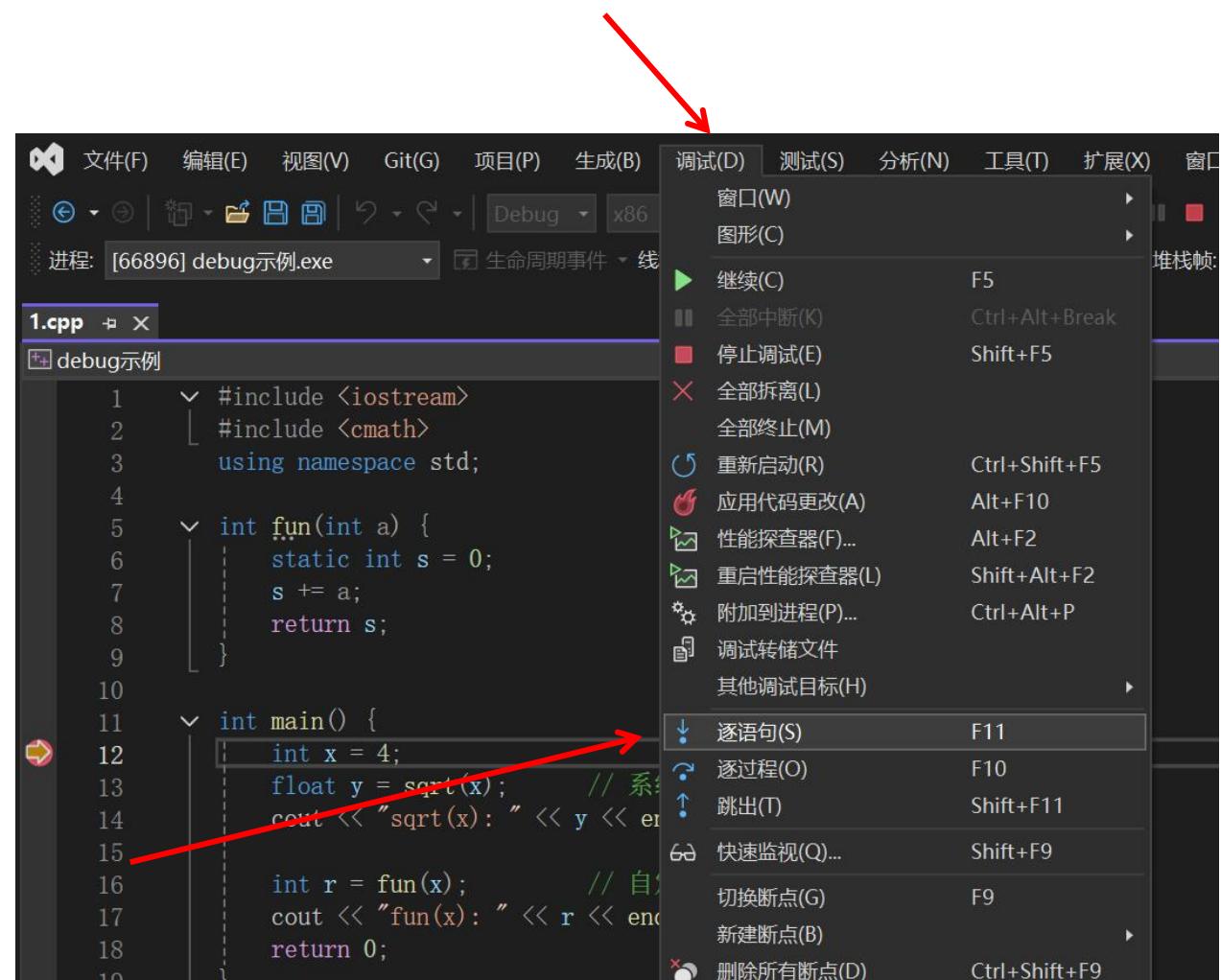
```
10
11 int main() {
12     int x = 4;
13     float y = sqrt(x); // 系统函数
14     cout << "sqrt(x): " << y << endl;
15
16     int r = fun(x); // 自定义函数
17     cout << "fun(x): " << r << endl; // 此行被单步跳过
18 }
```

```
10
11 int main() {
12     int x = 4;
13     float y = sqrt(x); // 系统函数
14     cout << "sqrt(x): " << y << endl;
15
16     int r = fun(x); // 自定义函数
17     cout << "fun(x): " << r << endl; // 此行被单步跳过
18 }
```

1.2 在函数中单步执行语句（知识点1.2，1.4）

1.2.2 每个语句单步执行，进入函数内部

在菜单栏选择“调试”，在出现的菜单中选择“逐语句”，或者直接按F11，在遇到系统函数的时候，会进入内部。按shift+F11可跳出。



```
10 int main() {  
11     int x = 4;  
12     float y = sqrt(x); // 系统函数  
13     cout << "sqrt(x): " << y << endl;  
14  
15     int r = fun(x); // 自定义函数  
16     cout << "fun(x): " << r << endl;  
17     return 0;  
18 }
```

```
10 int main() {  
11     int x = 4;  
12     float y = sqrt(x); // 系统函数  
13     cout << "sqrt(x): " << y << endl;  
14  
15     int r = fun(x); // 自定义函数  
16     cout << "fun(x): " << r << endl;  
17     return 0;  
18 }
```

↓ 进入sqrt函数内部

```
851 // No modf(), types must match  
852 // abs() has integer overloads  
853 // GENERIC_MATH1(log, int)  
854 // GENERIC_MATH1(log)  
855 // GENERIC_MATH1(log10)  
856 // GENERIC_MATH1(log1p)  
857 // GENERIC_MATH1(log2)  
858 // GENERIC_MATH1(logb)  
859 // GENERIC_MATH1(scalbn, int)  
860 // GENERIC_MATH1(scalbln, long)  
861 // GENERIC_MATH1(cbrt)  
862 // GENERIC_MATH1(fabs)  
863 // GENERIC_MATH1(hypot)  
864 // 3-arg hypot() is hand-crafted  
865 // GENERIC_MATH1(pow)  
866 // GENERIC_MATH1(sqrt) 已用时间 <= 1ms  
867 // GENERIC_MATH1(erf)  
868 // GENERIC_MATH1(erfc)  
869 // GENERIC_MATH1(lgamma)  
870 // GENERIC_MATH1(tgamma)  
871 // GENERIC_MATH1(ceil, __builtin_ceil, __ceil)  
872 // GENERIC_MATH1(floor, __builtin_floor, __floor)  
873 // GENERIC_MATH1(nearbyint)  
874 // GENERIC_MATH1(rint)
```

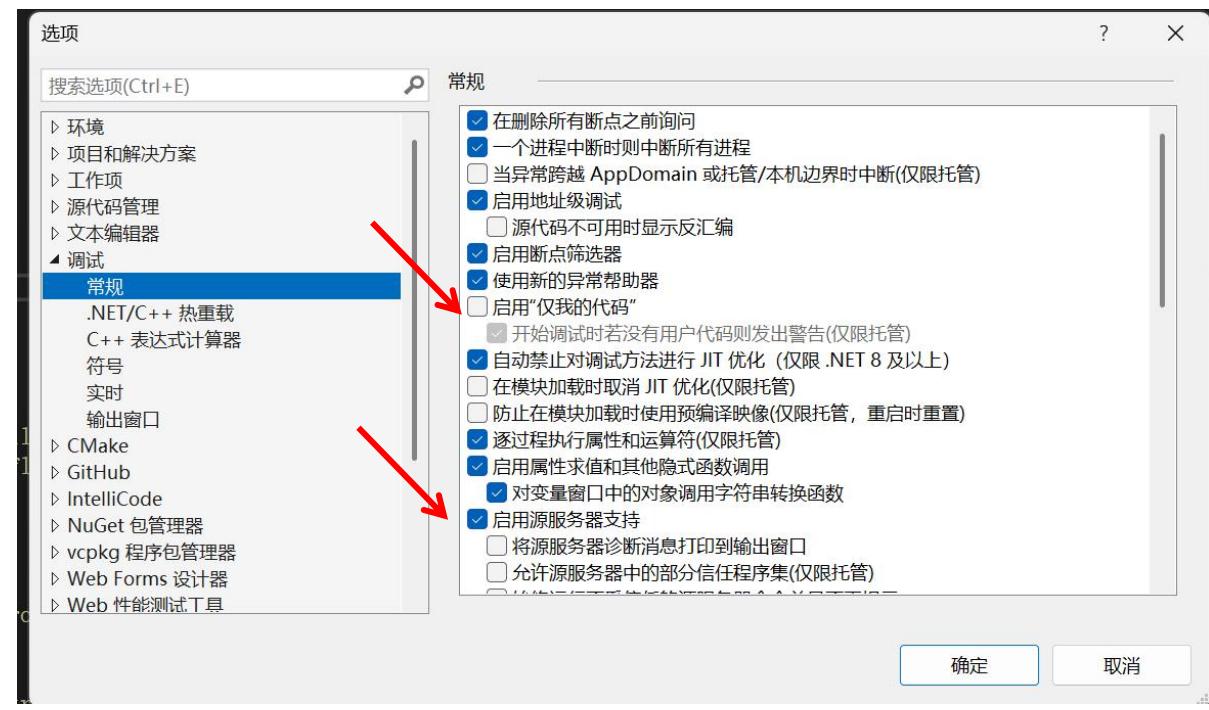
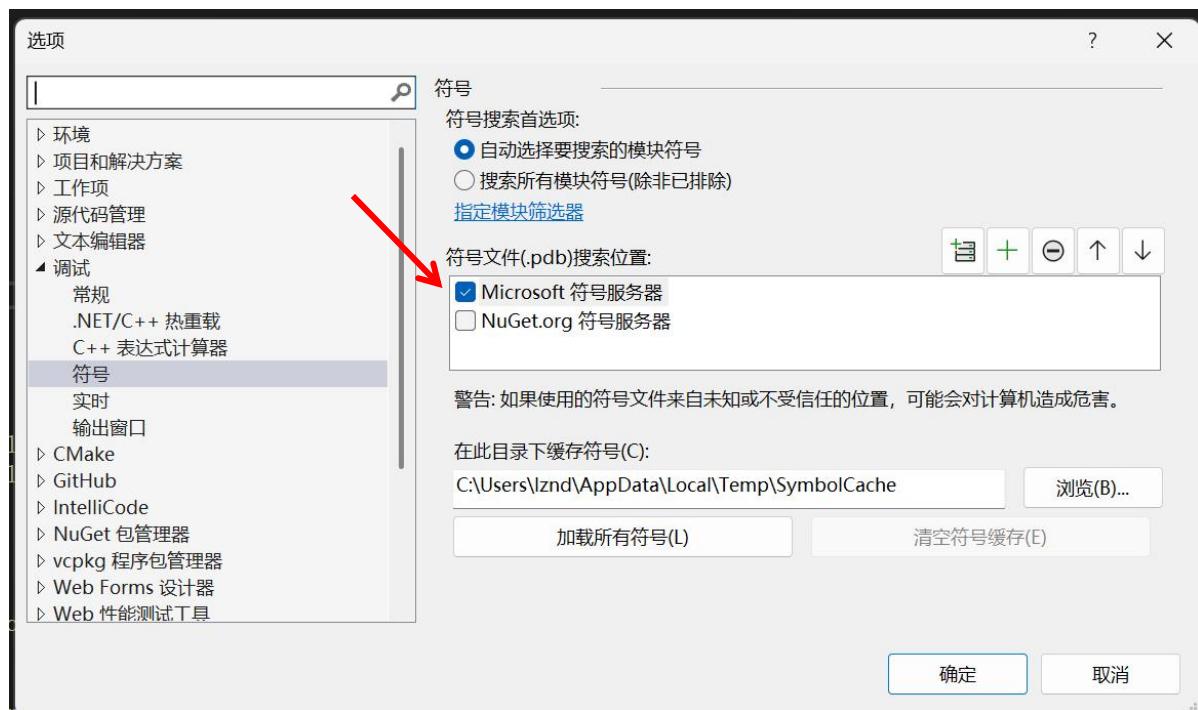
1.2 在函数中单步执行语句（知识点1.4）

1.2.2 每个语句单步执行，进入函数内部

默认设置之中不会进入系统函数内部，想要进入要修改设置

1. 工具>选项>调试>符号 勾选Microsoft符号服务器

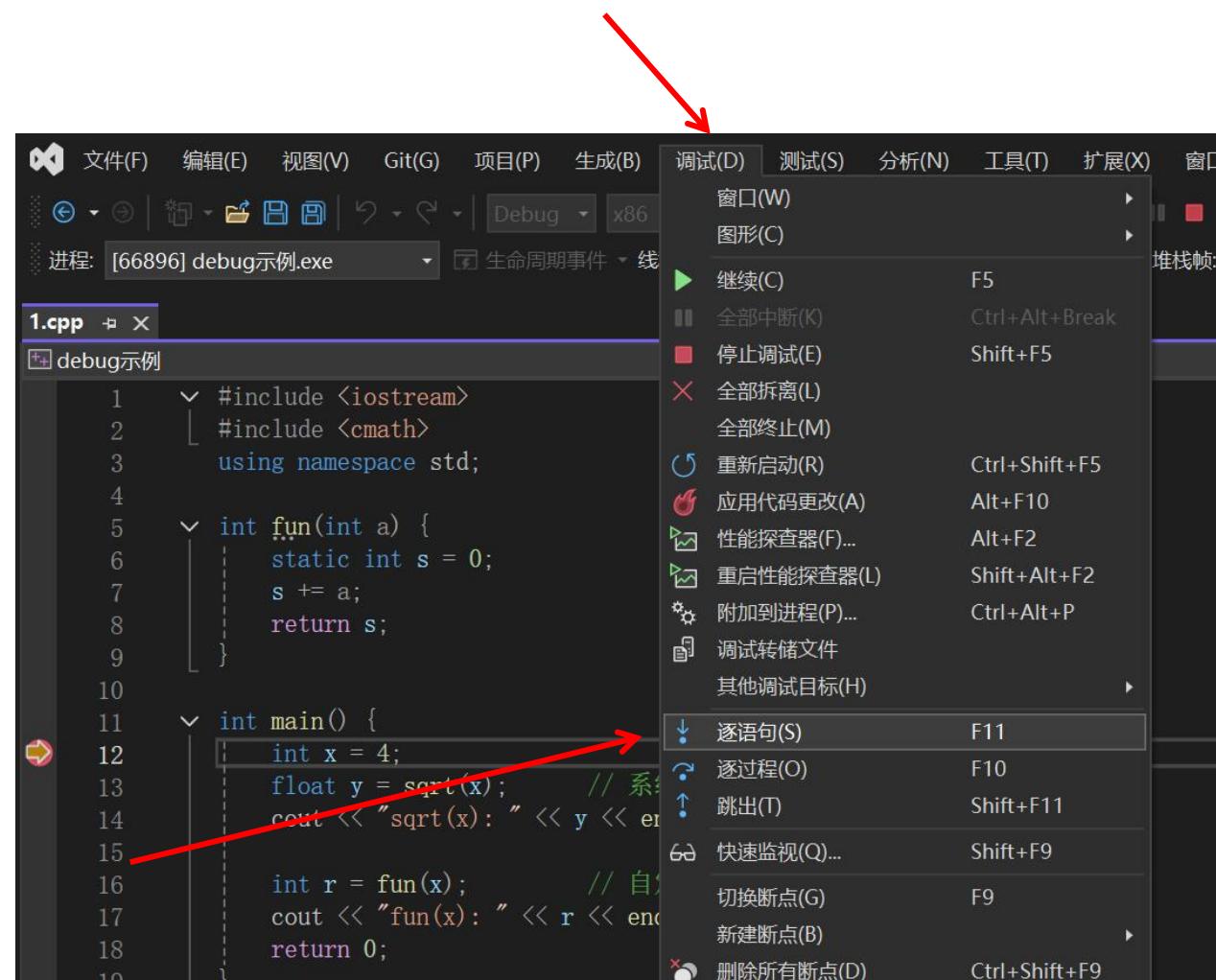
2. 工具>选项>调试>常规 勾选启用源服务器支持，取消勾选仅启用我的代码



1.2 在函数中单步执行语句（知识点1.2，1.6）

1.2.2 每个语句单步执行，进入函数内部

在菜单栏选择“调试”，在出现的菜单中选择“逐语句”，或者直接按F11，在遇到自定义函数的时候，会进入内部。



The screenshot shows the debugger at the beginning of the 'fun' function. A red arrow points from the 'Step Into (S)' menu option to the current line of code, which is highlighted in yellow. The code editor window shows the same C++ code as the previous screenshot.

The screenshot shows the debugger inside the 'fun' function. A red arrow points from the current line of code to the text '进入fun函数内部' (Enter fun function). The code editor window shows the same C++ code as the previous screenshots, with the current line of code highlighted in yellow.

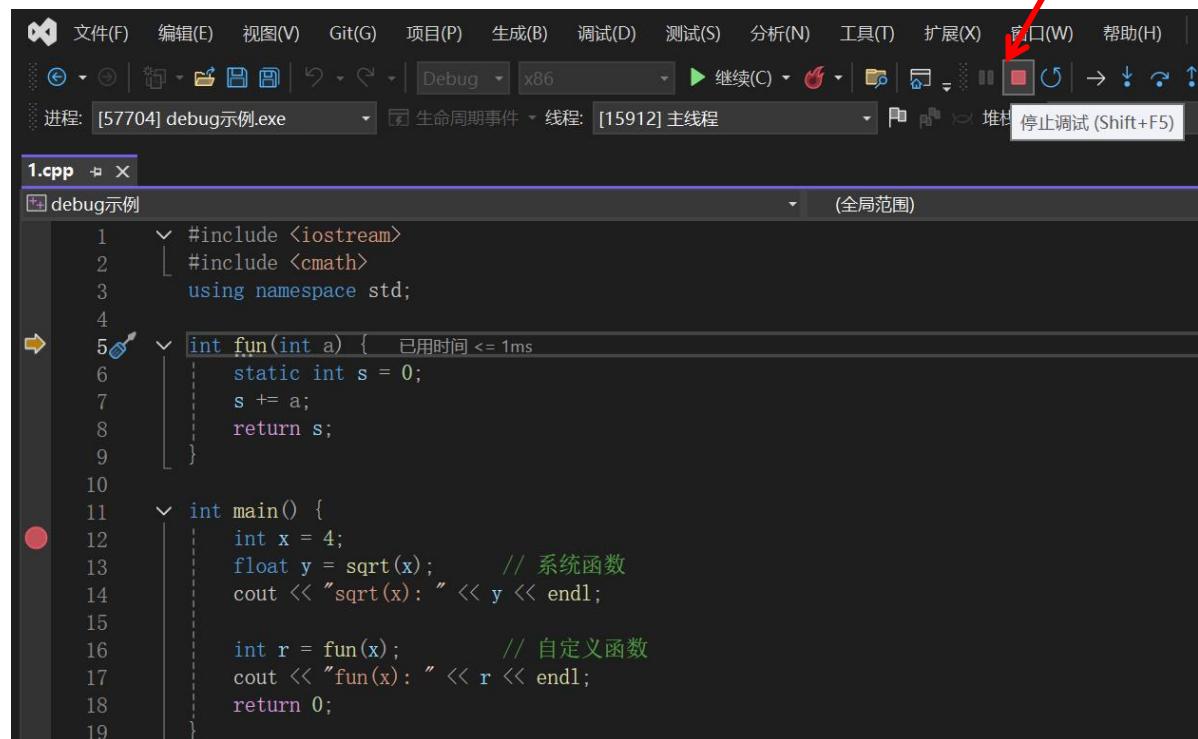
进入fun函数内部

1.3 结束调试（知识点1.1）

方法1：点击菜单栏中红色的停止按钮

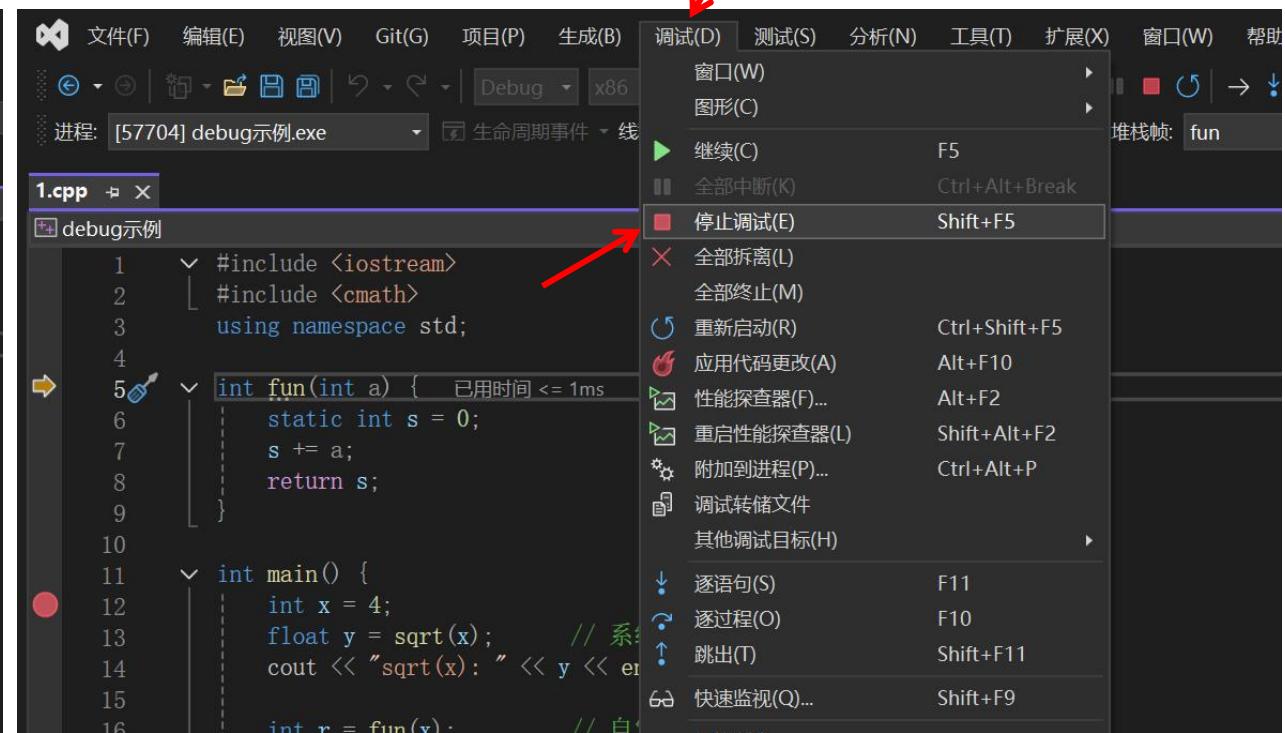
方法2：按shift+F5即可结束调试

方法3：在菜单栏选择“调试”，在出现的菜单中选择“停止调试”



1.cpp

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 int fun(int a) { // 已用时间 <= 1ms
6     static int s = 0;
7     s += a;
8     return s;
9 }
10
11 int main() {
12     int x = 4;
13     float y = sqrt(x); // 系统函数
14     cout << "sqrt(x): " << y << endl;
15
16     int r = fun(x); // 自定义函数
17     cout << "fun(x): " << r << endl;
18     return 0;
19 }
```



2.1 查看形参/自动变量的变化情况（知识点2.1）

main.cpp

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

static int static_global = 10;//静态全局变量

int global=100;//全局变量

void print();
void fun(int param)//形参
{
    static int local = 0;//局部静态变量
    int auto_var = param + 1;//自动变量
    local += param;
    cout << auto_var << endl;
    cout << local << endl;
}

int main()
{
    cout << &static_global << " " << static_global
    << endl;
    cout << &global << " " << global << endl;
    print();
    fun(1);
    fun(100);
    return 0;
}
```

other.cpp

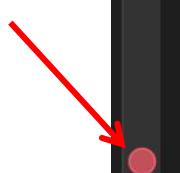
```
#include <iostream>
using namespace std;

static int static_global = 20;//静态全局变量

extern int global;

void print()
{
    global += 1;
    cout << &static_global << " " << static_global << endl;
    cout << &global << " " << global << endl;
}
```

先设置断点，并开始调试



```
16
17
18 int main()
19 {
20     cout << &static_global << " " << static_global
21     << endl;
22     cout << &global << " " << global << endl;
23     print();
24     fun(1);
25     fun(100);
26 }
```

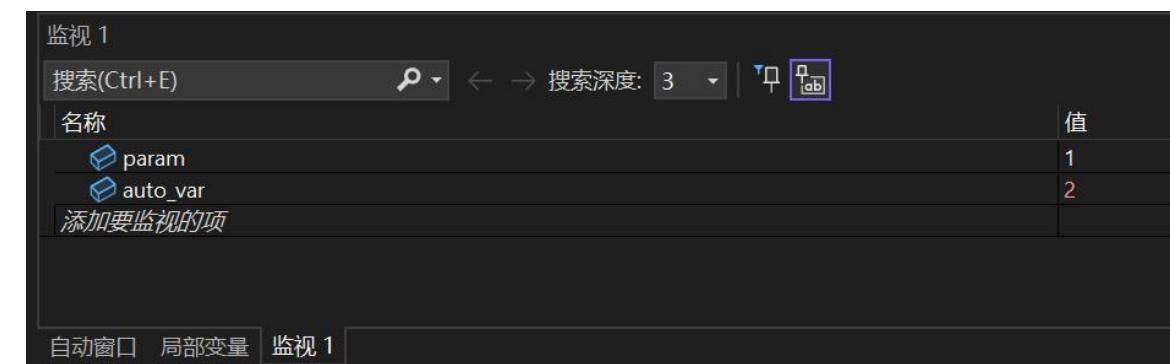
2.1 查看形参/自动变量的变化情况（知识点2.1）

The screenshot shows a debugger interface with the following details:

- Code View:** The code is named "debug示例2". It includes global variables `static_global` and `global`, and local variables `local` and `auto_var`. The `fun` function takes an integer parameter `param` and prints the values of `auto_var` and `local`.
- Breakpoint:** A red arrow points to a breakpoint at the start of the `main` function.
- Watch Window:** The "监视 1" (Watch 1) window at the bottom shows the variable `param` with a value of 1 and the variable `auto_var` with a value of 2031401200. A red arrow points to the row for `auto_var`.

按F11进入fun()函数内部，
打开下方监视窗口，输入形参名
param和自动变量名auto_var，此
时param值为1，auto_var值不可信。

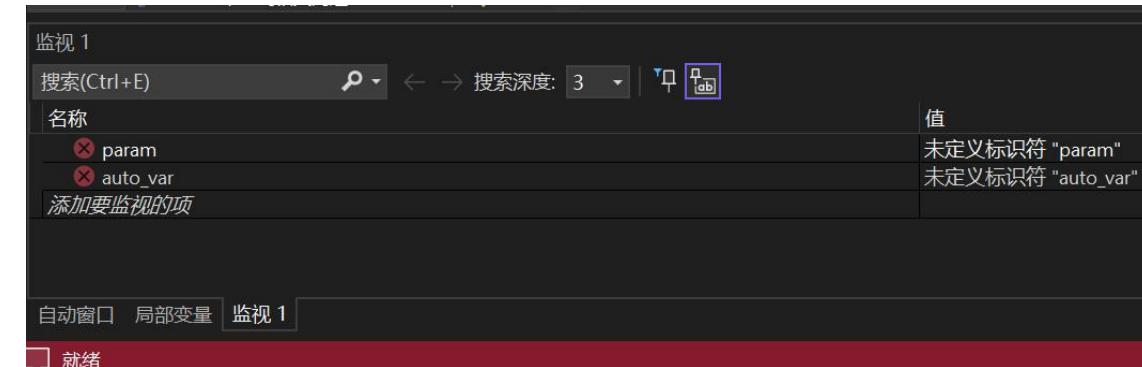
按F10运行函数后auto_var
被赋值为2。



2.1 查看形参/自动变量的变化情况（知识点2.1）

到第二个fun函数的位置按F11进入内部，如果前面没有进入fun（1）的内部，两个都会显示未定义，如果进入过，就会保持执行fun（1）后的值

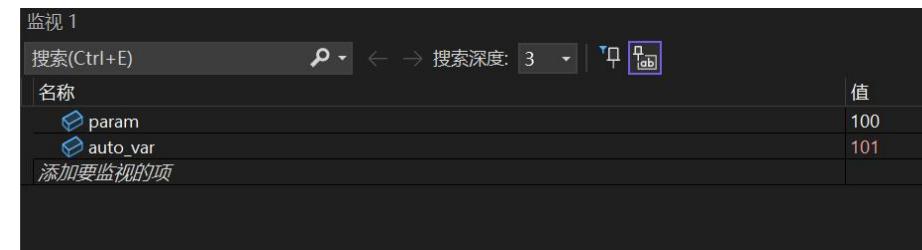
```
10
20     cout << &static_global << " " << static_global;
21     cout << &global << " " << global << endl;
22     print();
23     fun(1);
24     fun(100);    已用时间 <= 2ms
25
26 }
```



```
10
11 {
12     static int local = 0;
13     int auto_var = param + 1;    已用时间 <= 1ms
14     local += param;
15     cout << auto_var << endl;
16     cout << local << endl;
17 }
18
19 int main()
20 {
21     cout << &static_global << " " << static_global << endl;
22     cout << &global << " " << global << endl;
23     print();
24     fun(1);
25     fun(100);
26     return 0;
}
```

现在按照进入过fun（1）的步骤继续，param被赋值为100，而auto_var值不可信，之后被赋值为101，这说明形参和自动变量每一次调用函数时都会重新初始化，不会保留上一次的值。

名称	值
param	100
auto_var	-858993460



2.2 查看静态局部变量的变化（知识点2.2）

按F11进入fun(1) 内部

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

static int static_global = 10;

int global=100;

void print();
void fun(int param)
{
    static int local = 0;
    int auto_var = param + 1;
    local += param;
    cout << auto_var << endl;
    cout << local << endl;
}

int main()
{
    cout << &static_global << " " << static_gl
    cout << &global << " " << global << endl;
    print();
    fun(1);
    fun(100);
    return 0;
}
```

静态局部变量local的初始值为0

名称	值
local	0

自动窗口 局部变量 监视 1

```
void fun(int param)
{
    static int local = 0;
    int auto_var = param + 1;
    local += param;
    cout << auto_var << endl; 已用时间 <= 1ms
    cout << local << endl;
}

int main()
{
    cout << &static_global << " " << static_global << endl;
    cout << &global << " " << global << endl;
    print();
    fun(1);
    fun(100);
    return 0;
}
```

监视 1

名称	值
local	1

local+=param后得到local=1

2.2 查看静态局部变量的变化（知识点2.2）

按F11进入fun(100) 内部

```
17 int main()
18 {
19     cout << &static_global << " " << static_global << endl;
20     cout << &global << " " << global << endl;
21     print();
22     fun(1);
23     fun(100); 已用时间 <= 1ms
24     return 0;
25 }
```



```
8 void print();
9 void fun(int param)
10 {
11     static int local = 0;
12     int auto_var = param + 1; 已用时间 <= 1ms
13     local += param;
14     cout << auto_var << endl;
15     cout << local << endl;
16 }
17
18 int main()
19 {
20     cout << &static_global << " " << static_global << endl;
21     cout << &global << " " << global << endl;
22     print();
23     fun(1);
24     fun(100);
25     return 0;
26 }
```

进入函数内部后，local没有重新赋值，而是保持fun(1)执行以后的值，并且在此基础上继续累加得到101。所以静态局部变量只会被定义一次，地址始终不变。

```
9 void fun(int param)
10 {
11     static int local = 0;
12     int auto_var = param + 1;
13     local += param;
14     cout << auto_var << endl;
15     cout << local << endl;
16 }
17
18 int main()
19 {
20     cout << &static_global << " " << static_global << endl;
21     cout << &global << " " << global << endl;
22     print();
23     fun(1);
24     fun(100);
25     return 0;
26 }
```



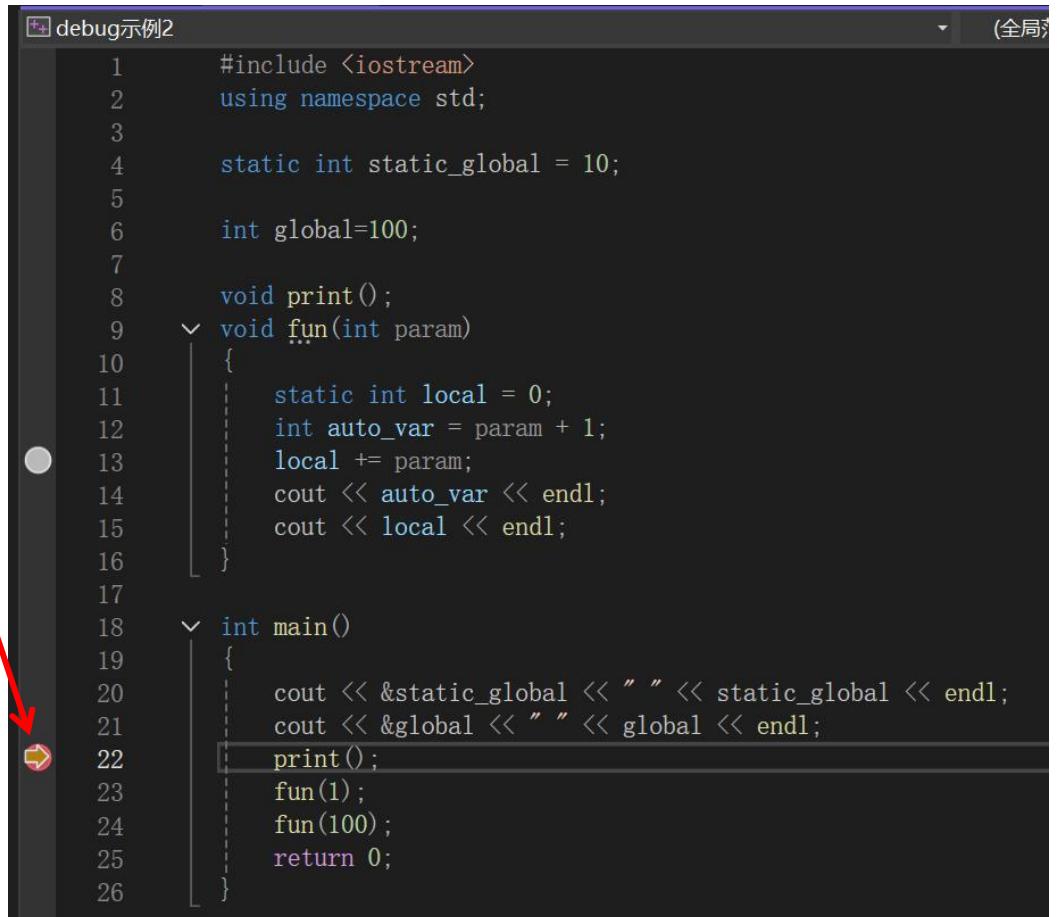
```
121 % 未找到相关问题 | 监视 1 | 搜索(Ctrl+E) | 搜索深度: 3 | 增加要监视的项
名称 | 值
local | 1
```



2.3 查看静态全局变量的变化（知识点2.3）

设置断点，开始调试，监视static_global及其地址

在这个源文件里static_global=10



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

static int static_global = 10;

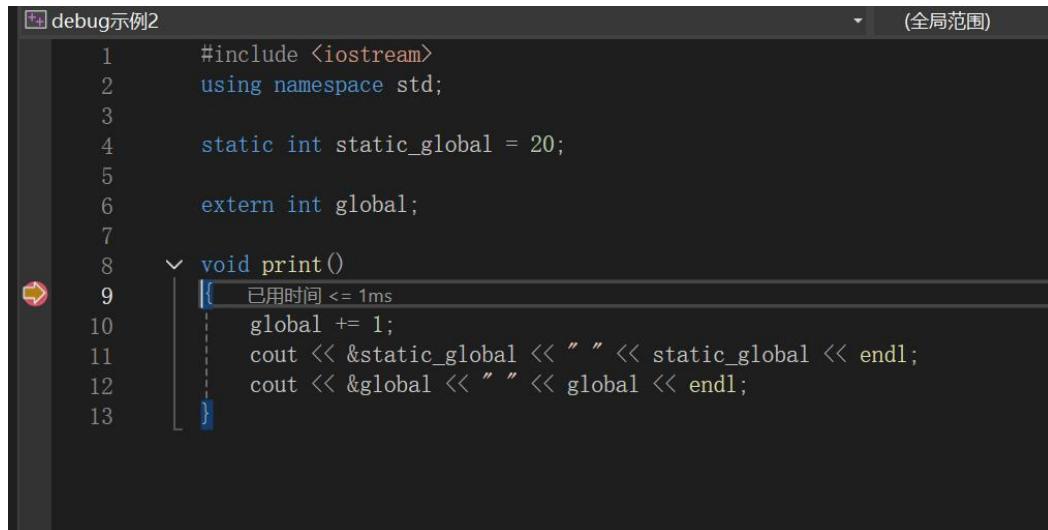
int global=100;

void print();
void fun(int param)
{
    static int local = 0;
    int auto_var = param + 1;
    local += param;
    cout << auto_var << endl;
    cout << local << endl;
}

int main()
{
    cout << &static_global << " " << static_global << endl;
    cout << &global << " " << global << endl;
    print();
    fun(1);
    fun(100);
    return 0;
}
```

名称	值
static_global	10
&static_global	0x0029c004 {debug示例2.exe!int static_global} {10}
添加要监视的项	

按F11进入other.cpp
文件



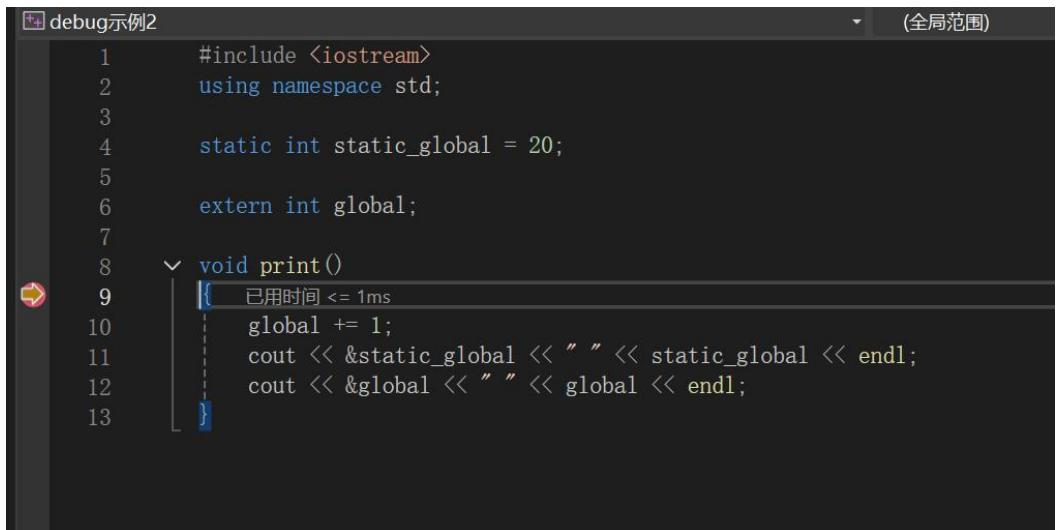
```
#include <iostream>
using namespace std;

static int static_global = 20;

extern int global;

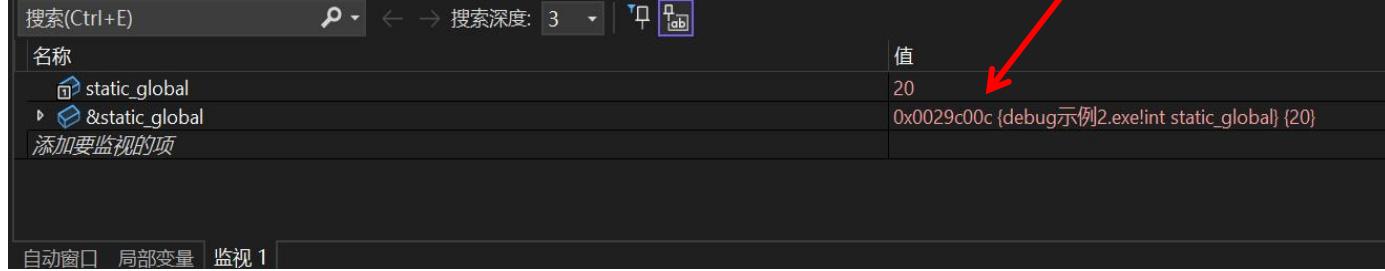
void print()
{
    已用时间 <= 1ms
    global += 1;
    cout << &static_global << " " << static_global << endl;
    cout << &global << " " << global << endl;
}
```

2.3 查看静态全局变量的变化（知识点2.3）



```
1 #include <iostream>
2 using namespace std;
3
4 static int static_global = 20;
5
6 extern int global;
7
8 void print()
9 {
10     已用时间 <= 1ms
11     global += 1;
12     cout << &static_global << " " << static_global << endl;
13     cout << &global << " " << global << endl;
14 }
```

结论：在other.cpp中，`static_global=20`,而且地址与main.cpp里的不一样，这说明两个不同源文件里同名的静态全局变量是两个独立的变量



名称	值
static_global	20
&static_global	0x0029c00c {debug示例2.exe!int static_global} {20}

2.4 查看外部全局变量的变化（知识点2.4）

设置断点，开始调试，监视全局变量global及其地址，此时global=100

```
13     local += param;
14     cout << auto_var << endl;
15     cout << local << endl;
16 }
17
18 int main()
19 {
20     cout << &static_global << " " << static_global << endl;
21     cout << &global << " " << global << endl;
22     print();
23     fun(1);
24     fun(100);
25     return 0;
26 }
```

global的值和地址都不变

监视 1

搜索(Ctrl+E) ← → 搜索深度: 3

名称	值
global	100
&global	0x0029c000 {debug示例2.exe!int global} {100}

添加要监视的项

名称

名称	值
global	100
&global	0x0029c000 {debug示例2.exe!int global} {100}

添加要监视的项

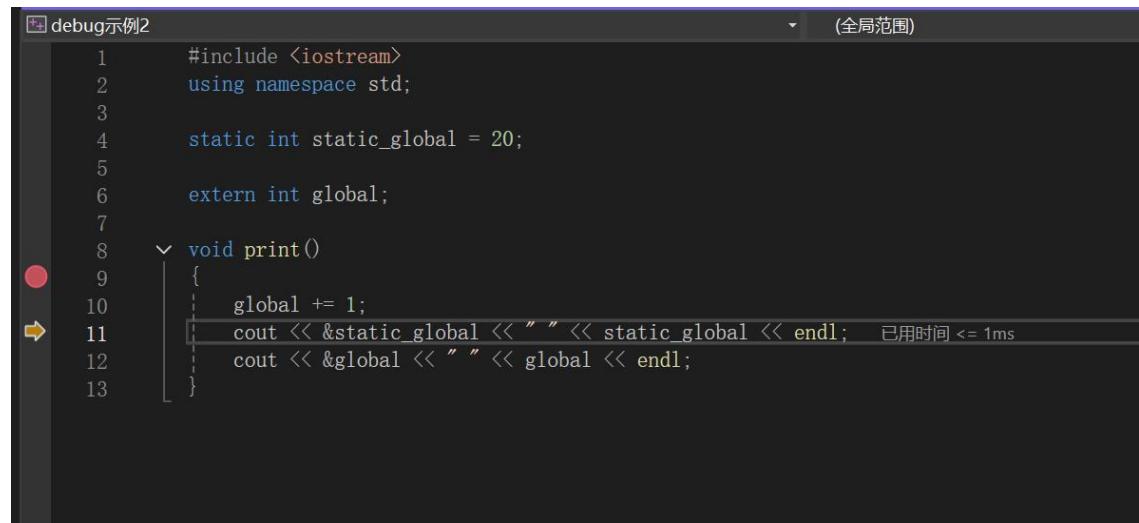
按F11进入other.cpp

(全局范围)

```
1 #include <iostream>
2 using namespace std;
3
4 static int static_global = 20;
5
6 extern int global;
7
8 void print()
9 {
10     global += 1;
11     cout << &static_global << " " << static_global << endl;
12     cout << &global << " " << global << endl;
13 }
```

2.4 查看外部全局变量的变化（知识点2.4）

按F10运行到图中所示位置时，global=101



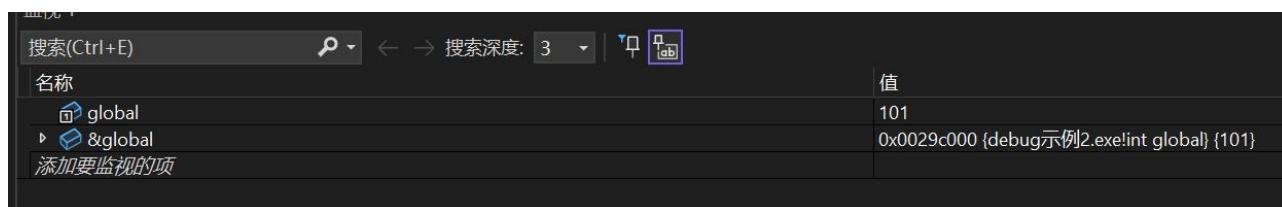
```
#include <iostream>
using namespace std;

static int static_global = 20;

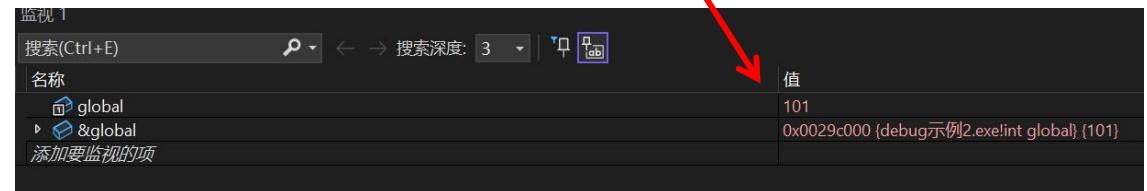
extern int global;

void print()
{
    global += 1;
    cout << &static_global << " " << static_global << endl;    已用时间 <= 1ms
    cout << &global << " " << global << endl;
}
```

global的值和地址都不变

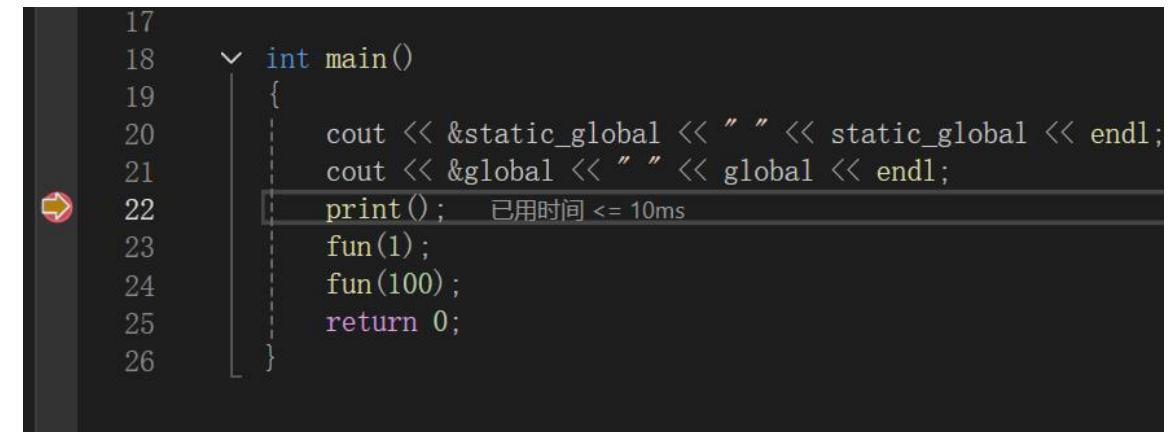


名称	值
global	101 0x0029c000 [debug示例2.exe!int global] {101}



名称	值
global	101 0x0029c000 [debug示例2.exe!int global] {101}

按shift+F11退出other.cpp



```
int main()
{
    cout << &static_global << " " << static_global << endl;
    cout << &global << " " << global << endl;
    print();    已用时间 <= 10ms
    fun(1);
    fun(100);
    return 0;
}
```

结论：多文件中外部全局变量是同一个变量，是共享的

3.1 查看char/int/float等简单变量（知识点3.1）

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

void process_array(int* arr_ptr, int size) { // 形参是指针
    cout << "数组首地址(形参): " << arr_ptr << endl;
    for (int i = 0; i < size; ++i) {
        cout << arr_ptr[i] << " "; // 通过指针访问数据
    }
    cout << endl;
}//3.6

int main()
{
    char c = 'A';
    int n = 42;
    float f = 3.14F;//3.1

    int* p = &n;
    char* pc = &c;//3.2

    int array1[5] = { 1, 2, 3, 4, 5 };//3.3

    int* p_array1 = array1;//3.4

    int array2[2][3] = { {1, 2, 3}, {4, 5, 6} };//3.5

    process_array(array1, 5);//3.6

    const char* str = "Hello";//3.7

    int& ref = n;//3.8

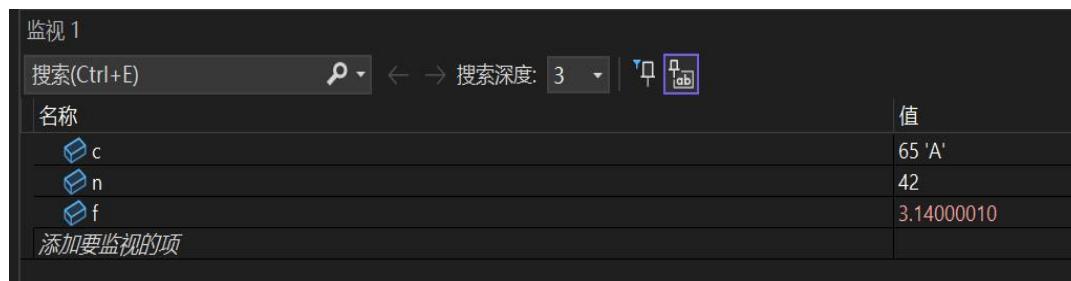
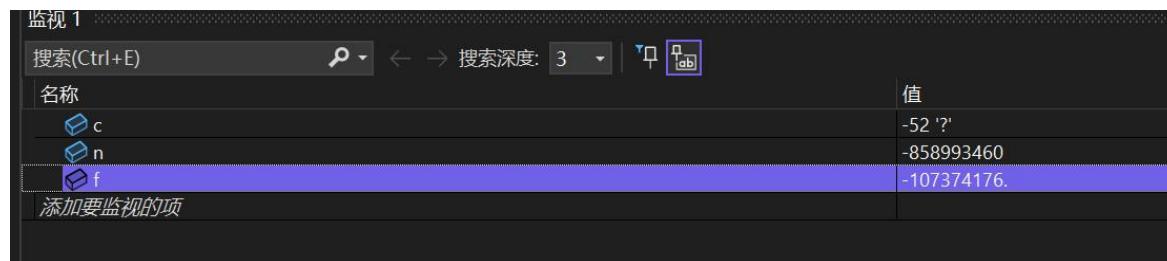
    int* danger = array1 + 10;//3.9
    return 0;
}
```

第三部分构造的源代码

设置断点，开始调试

```
11
12     v  int main()
13     {
14         char c = 'A';
15         int n = 42;
16         float f = 3.14F;//3.1
17
18         int* p = &n;
19         char* pc = &c;//3.2
20
21 }
```

在监视窗口输入变量名c, n, f



逐步运行后得到各个变量的值

3.2 查看指向简单变量的指针变量（知识点3.2）

```
11
12     int main()
13     {
14         char c = 'A';
15         int n = 42;
16         float f = 3.14F;//3.1
17
18         int* p = &n;    已用时间 <= 1ms
19         char* pc = &c;//3.2
20
21         int array1[5] = { 1, 2, 3, 4, 5 };//3.3
22
```

在监视窗口输入p,
*p, pc, *pc, 此
时还没有定义

名称	值
p	0xffffffff {???
*p	<无法读取内存>
pc	0xffffffff <读取字符串字符时出错。>
*pc	<无法读取内存>
添加要监视的项	



逐步运行可以查看指针的值（地址）
和对应的变量的值

```
12     int main()
13     {
14         char c = 'A';
15         int n = 42;
16         float f = 3.14;//3.1
17
18         int* p = &n;
19         char* pc = &c;//3.2
20
21         int array1[5] = { 1, 2, 3, 4, 5 };//3.3   已用时间 <= 1ms
22
```

搜索(Ctrl+E) 搜索深度: 3

名称	值
p	0x012ff7e8 {42}
*p	42
pc	0x012ff7f7 <字符串中的字符无效。>
*pc	65 'A'
添加要监视的项	

3. 3 查看一维数组（知识点3. 3）

```
17  
18     int* p = &n;  
19     char* pc = &c;//3. 2  
20  
21     int array1[5] = { 1, 2, 3, 4, 5 };//3. 3    已用时间 <= 1ms  
22  
23     int* p_array1 = array1;//3. 4  
24
```

在监视窗口输入array1和array1[2]分别查看整个一维数组和内部1的元素，此时未被赋值

监视 1	
名称	值
array1	0x012ff7a8 {-858993460, -858993460, -858993460, -858993460, -858993460}
array1[2]	-858993460



运行后得到一维数组的值及其内部单个元素的值

名称	值
array1	0x012ff7a8 {1, 2, 3, 4, 5}
array1[2]	3

3. 4 查看指向一维数组的指针变量（知识点3. 4）

```
20  
21     int array1[5] = { 1, 2, 3, 4, 5 };//3. 3  
22  
23     int* p_array1 = array1;//3. 4    已用时间 <= 1ms  
24
```

在监视窗口输入p_array1,*p_array1,p_array1[3]

监视 1	
名称	值
p_array1	0xffffffff {??}
*p_array1	<无法读取内存>
p_array1[3]	<无法读取内存>
添加要监视的项	



运行后得到各个值，p_array1是一维数组起始地址，
*p_array1是数组第一个值，p_array1[3]是数组第四个值

监视 1	
名称	值
p_array1	0x00affd68 {1}
*p_array1	1
p_array1[3]	4
添加要监视的项	

3. 5 查看二维数组（知识点3. 5）

```
21     int array1[5] = { 1, 2, 3, 4, 5 };//3.3
22
23     int* p_array1 = array1;//3.4
24
25     int array2[2][3] = { {1, 2, 3}, {4, 5, 6} };//3.5    已用时间 <= 1ms
26
27     const char* str = "Hello";//3.7
```

在监视窗口输入array2[0],array2[1][2],array2

监视 1	
名称	值
array2[0]	0x0073fb5c {-858993460, -858993460, -858993460}
array2[1][2]	-858993460
array2	0x0073fb5c {0x0073fb5c {-858993460, -858993460, -858993460}, 0x0073fb68 {-858993460, -858993460, -858993460}}
添加要监视的项	

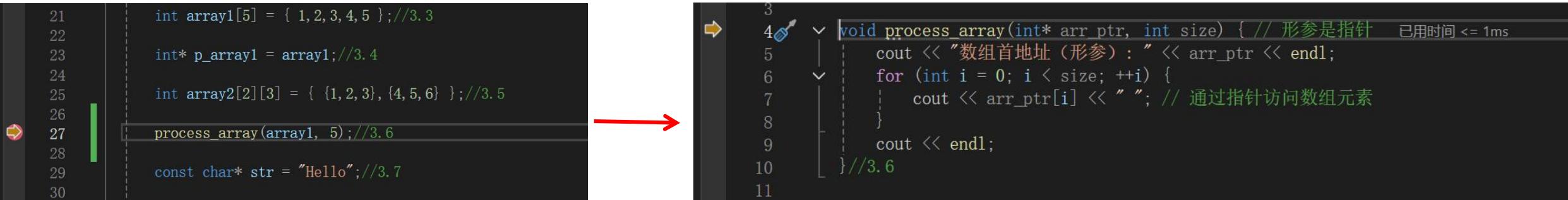


运行后得到各个值，array2[0]是数组第1行的值，array2[1][2]是第2行第3列的值，array2是整个二维数组的值

监视 1	
名称	值
array2[0]	0x0073fb5c {1, 2, 3}
array2[1][2]	6
array2	0x0073fb5c {0x0073fb5c {1, 2, 3}, 0x0073fb68 {4, 5, 6}}
添加要监视的项	

3. 6 在函数中查看实参数组的地址和值（知识点3. 6）

按F11进入函数内部



```
21 int array1[5] = { 1, 2, 3, 4, 5 };//3.3
22
23 int* p_array1 = array1;//3.4
24
25 int array2[2][3] = { {1, 2, 3}, {4, 5, 6} };//3.5
26
27 process_array(array1, 5);//3.6
28
29
30 const char* str = "Hello";//3.7
```

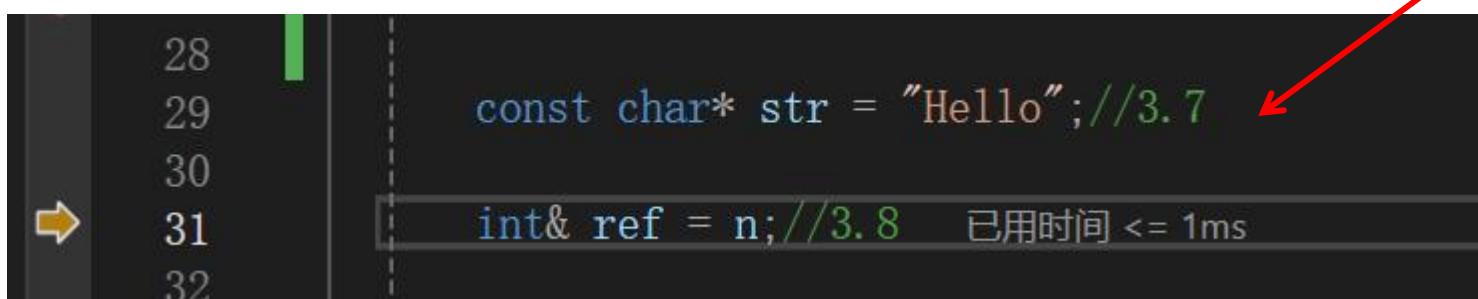
```
3 void process_array(int* arr_ptr, int size) { // 形参是指针 已用时间 <= 1ms
4     cout << "数组首地址(形参)：" << arr_ptr << endl;
5     for (int i = 0; i < size; ++i) {
6         cout << arr_ptr[i] << " ";
7     }
8     cout << endl;
9 }
10 }//3.6
11
```



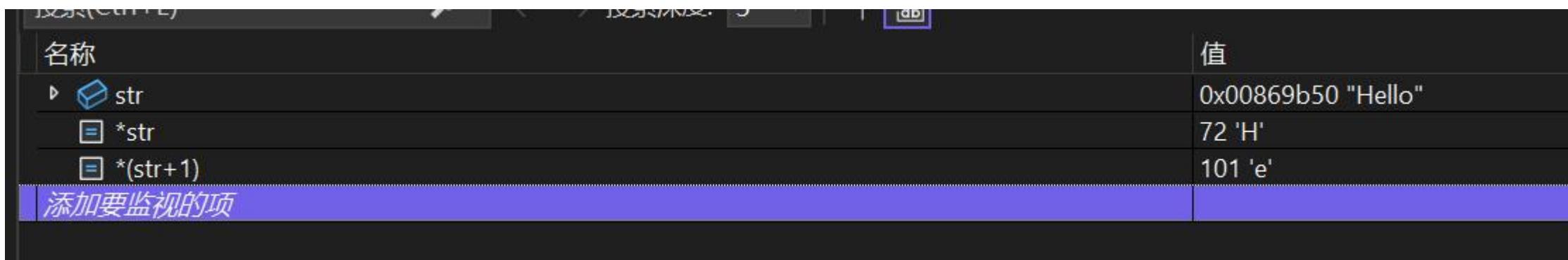
监视 1	
名称	值
arr_ptr	0x0097fb88 {1}
arr_ptr,5	0x0097fb88 {1, 2, 3, 4, 5}
&arr_ptr[2]	0x0097fb90 {3}
添加要监视的项	

在监视窗口输入arr_ptr,得到数组起始地址和第一个元素的值，输入arr_ptr,5得到整个数组元素的值，输入数字是多少就得到前多少个值，输入&arr_ptr[2]得到第三个元素的地址和值

3.7 查看指向字符串常量的指针变量（知识点3.7）



```
28
29     const char* str = "Hello";//3.7
30
31     int& ref = n;//3.8    已用时间 <= 1ms
32
```



名称	值
str	0x00869b50 "Hello"
*str	72 'H'
*(str+1)	101 'e'

在监视窗口输入str得到字符串的起始地址和字符串的值，可以看到无名字符串常量的地址；输入*str得到字符串的首字母，输入*(str+1) 得到第二个字母。

3.8 查看引用变量（知识点3.8）

```
29     const char* str = "Hello";//3.7
30
31     int& ref = n;//3.8    已用时间 <= 1ms
32
```

在监视窗口输入ref, &ref,n,&n,
运行后发现ref和n的地址和值都
相同。

名称	值
ref	<无法读取内存>
&ref	0xffffffff {??}
n	42
&n	0x0097fbc8 {42}
添加要监视的项	



名称	值
ref	42
&ref	0x0097fbc8 {42}
n	42
&n	0x0097fbc8 {42}
添加要监视的项	

引用和指针的区别：1.引用相当于变量的别名，而指针是存储变量地址的独立变量
2.引用没有独立的内存，指针会占存储空间
3.引用必须立刻初始化且不能改变绑定，指针可以先定义并且多次修改指向

3.9 使用指针时出现越界（知识点3.9）

```
31     int& ref = n;//3.8
32
33     int* danger = array1 + 10;//3.9    已用时间 <= 1ms
34     return 0;
35 }
```

搜索(Ctrl+E)		← → 搜索深度: 3	↑ Tab
名称		值	
► danger		0xffffffff {????}	
✗ *danger		<无法读取内存>	
添加要监视的项			

在监视窗口输入danger和*danger，运行这句话以后可以得到地址danger，但是由于越界，*danger的值是乱码。

监视(Ctrl+Shift+E)		↑ Tab
名称		值
► danger		0x0097fbb0 {9960392}
✗ *danger		9960392
添加要监视的项		