对VS2022调试工具基本使用方法的学习

班级：10071706

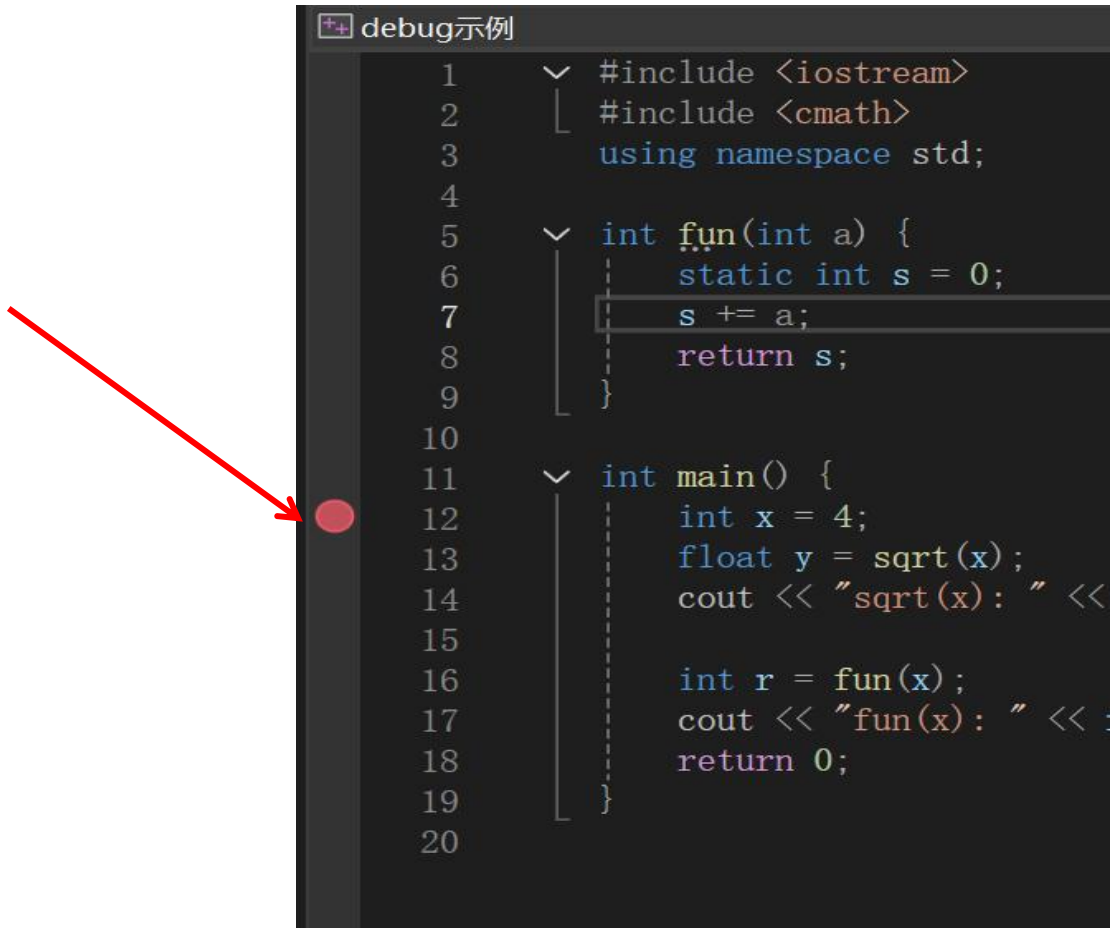
学号：2452654

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。



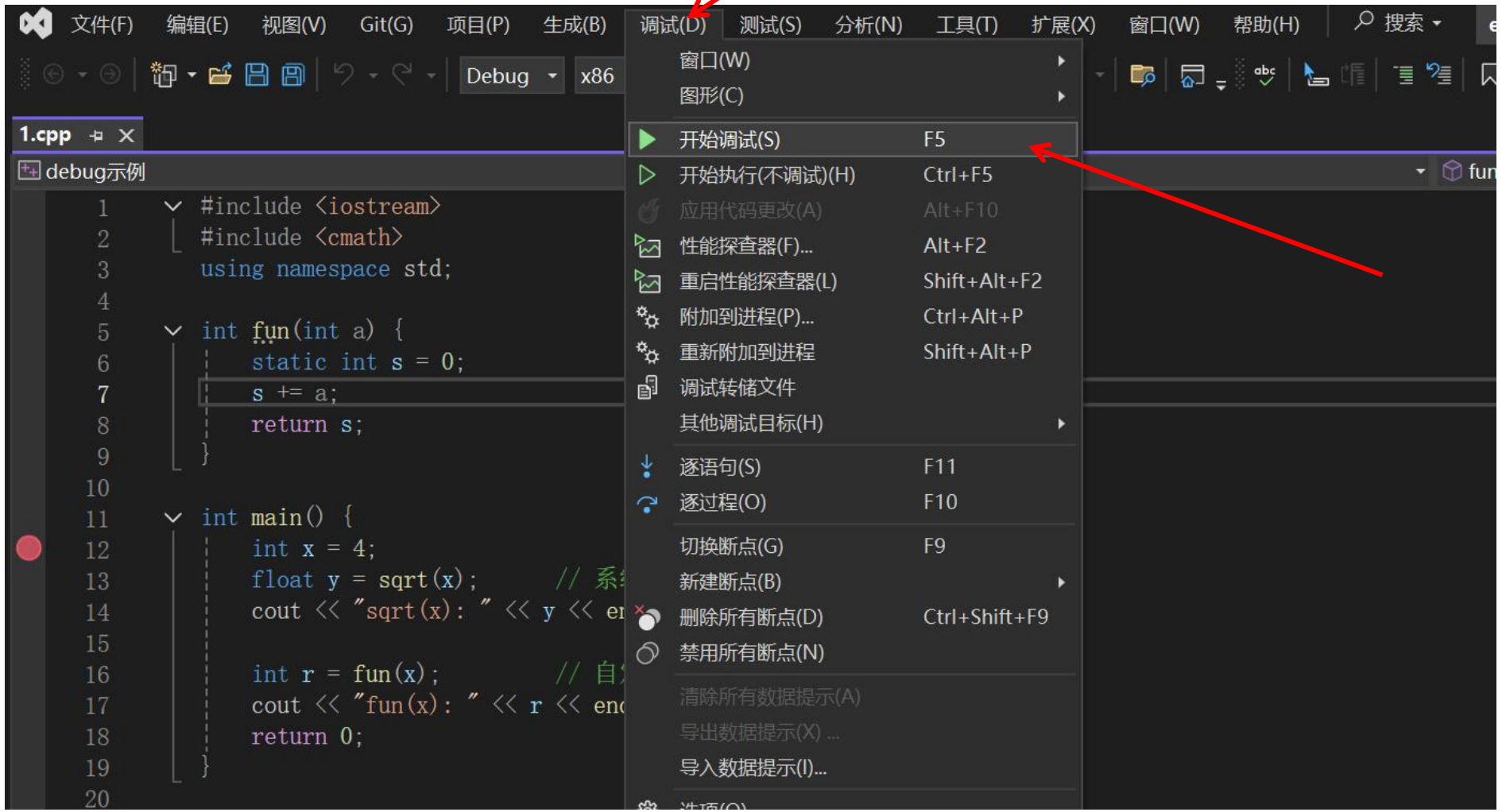
第一部分构造的源代码

```
#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）

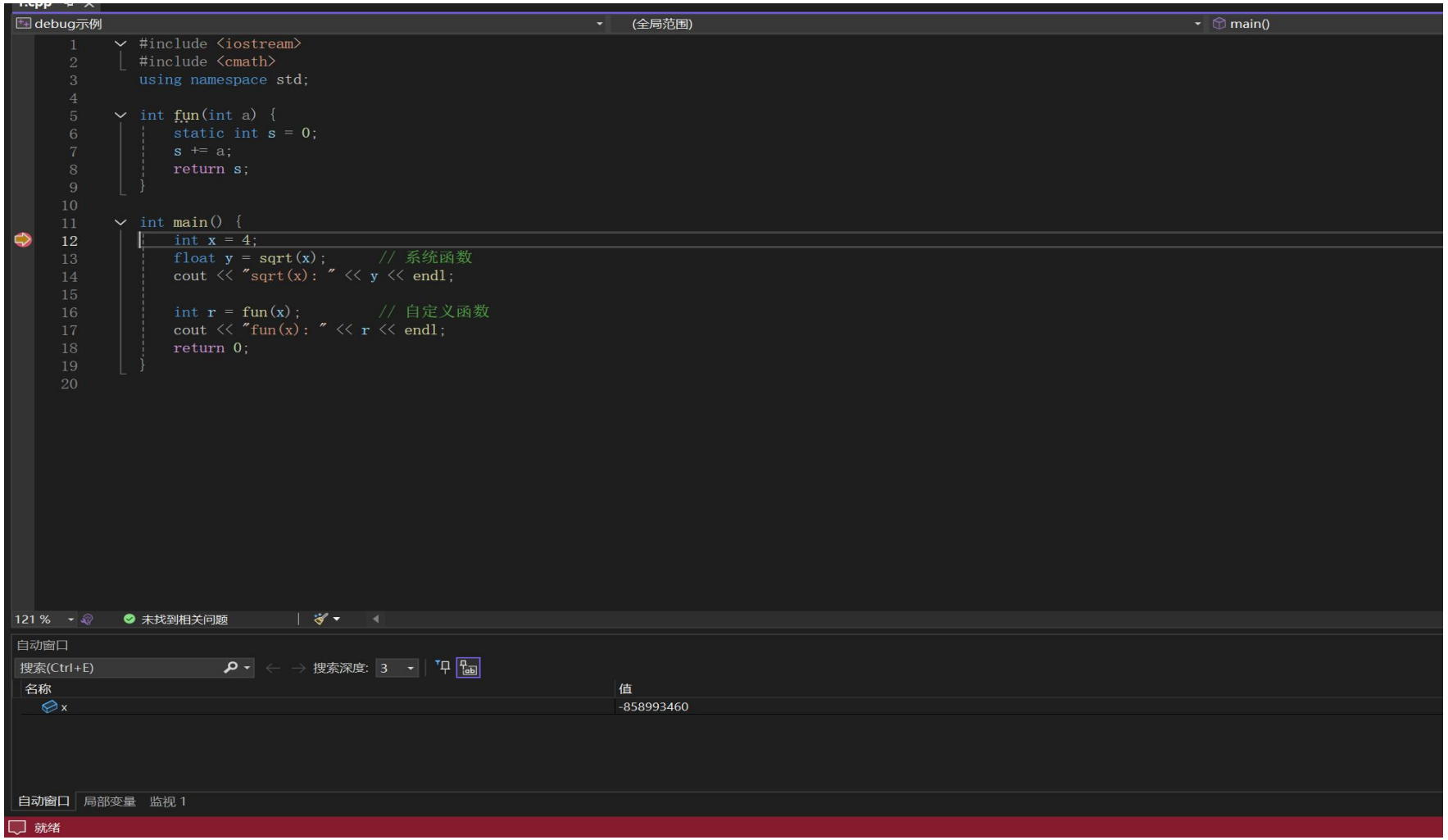
方法1：在菜单栏选择“调试”，在出现的菜单中选择“开始调试”

方法2：按F5



1.1 开始调试（知识点1.1）

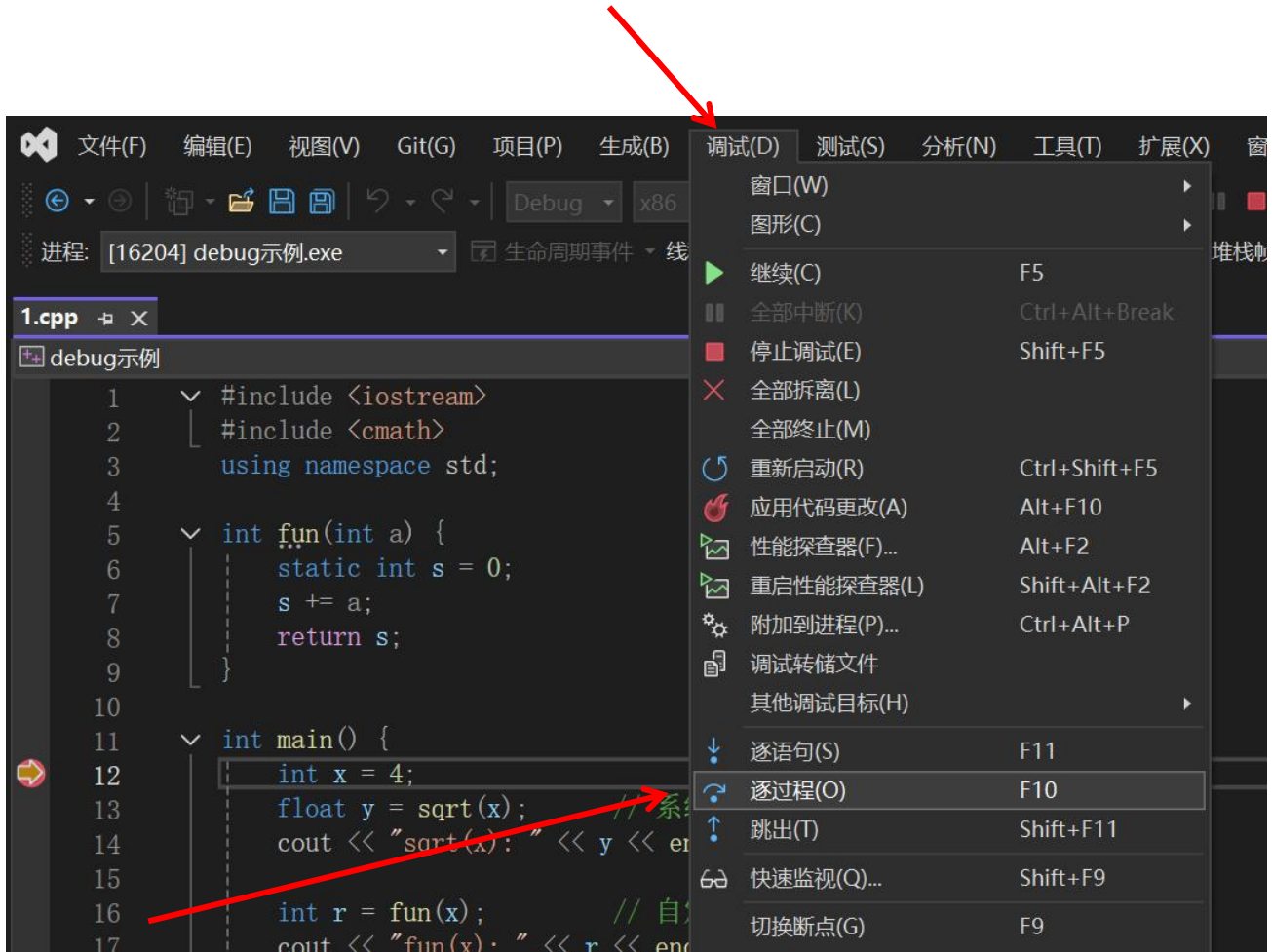
进入调试模式

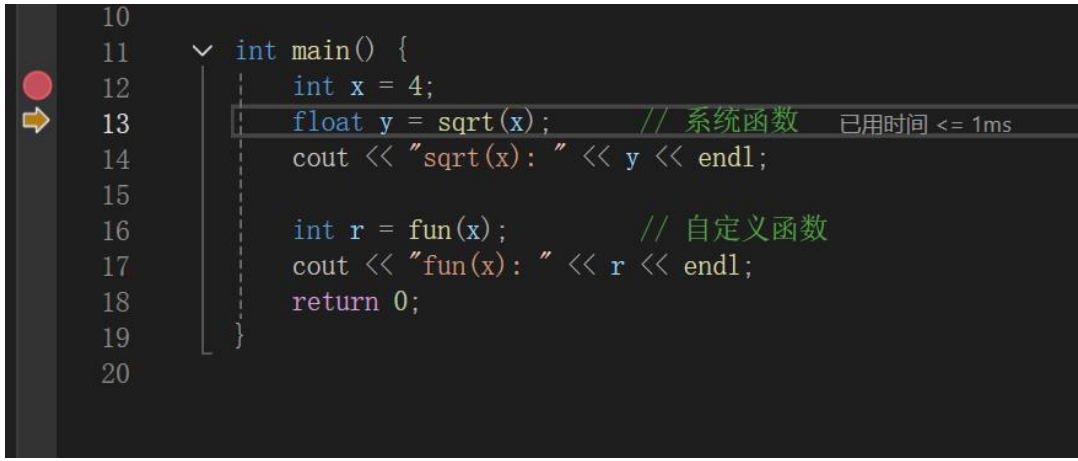


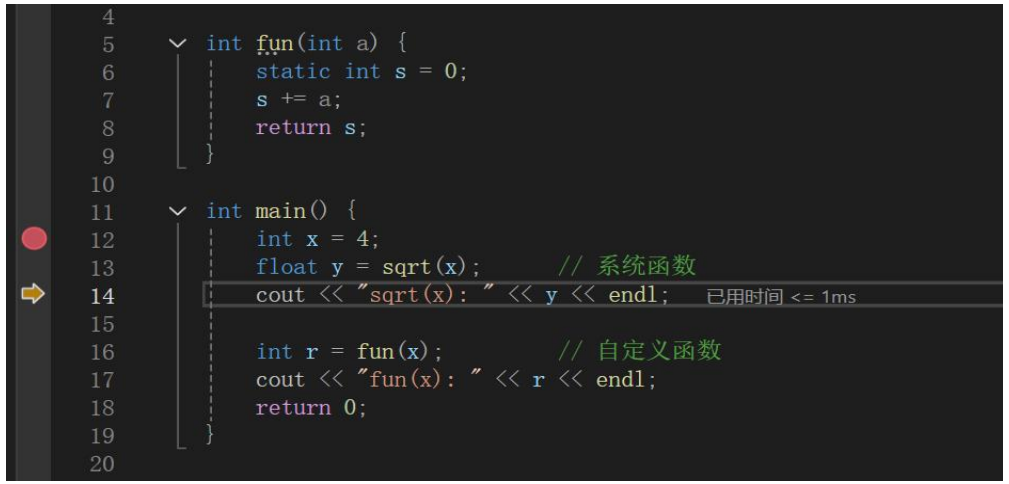
1.2 在函数中单步执行语句（知识点1.2，1.3）

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。



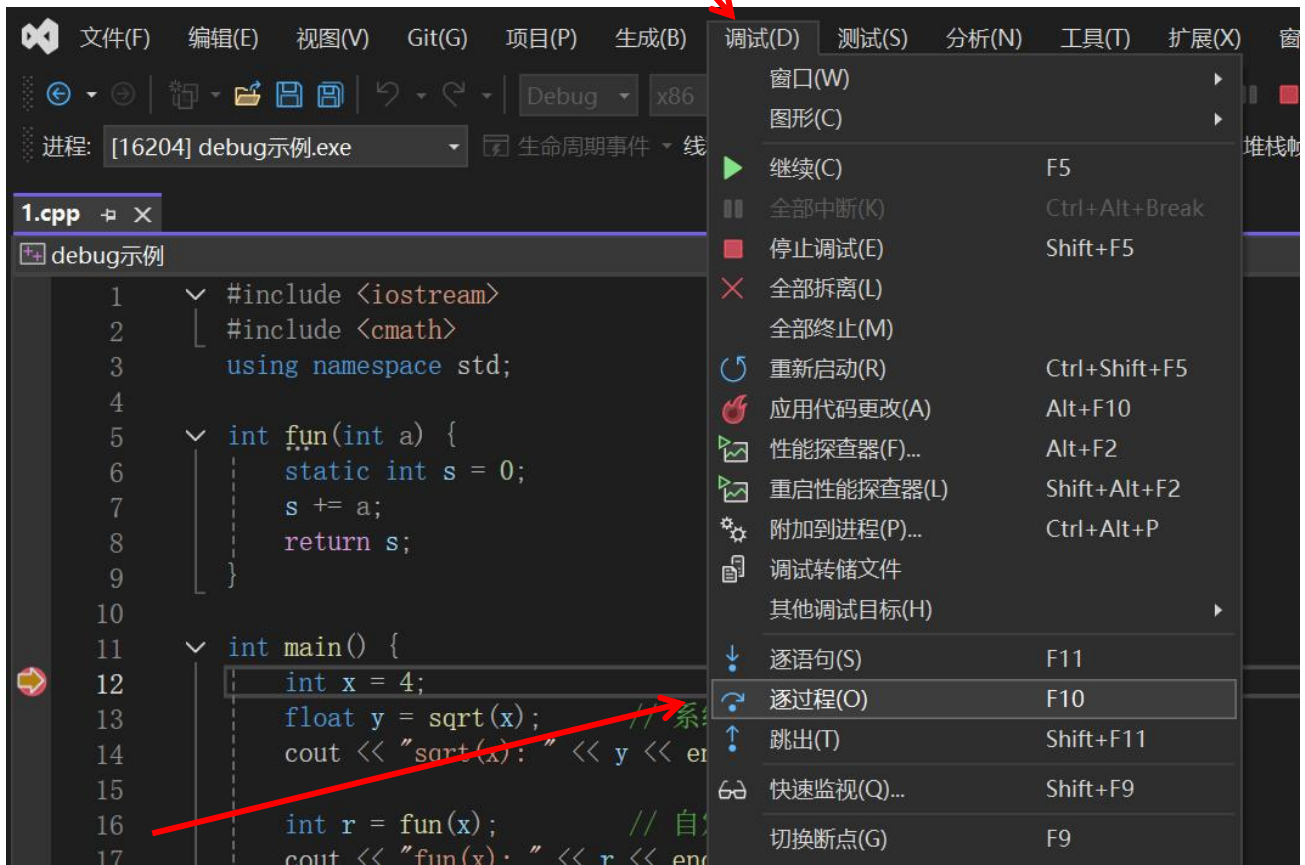


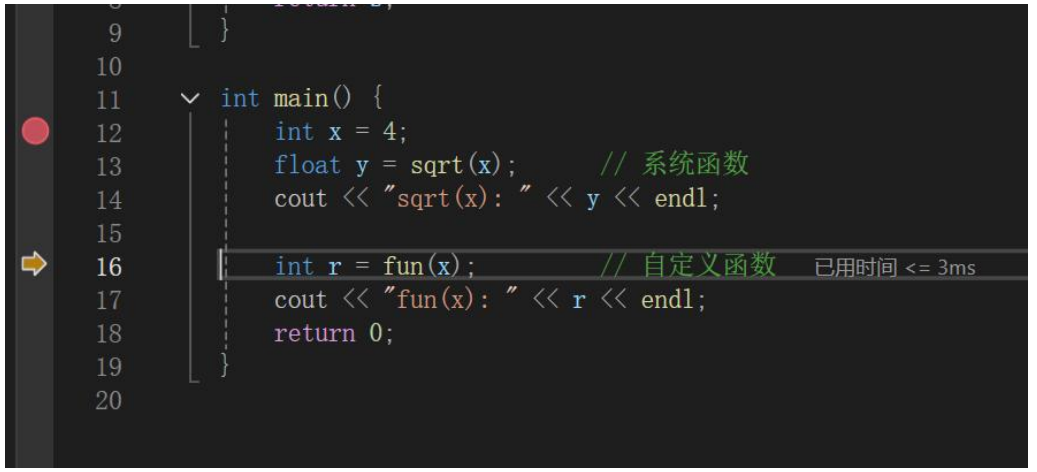


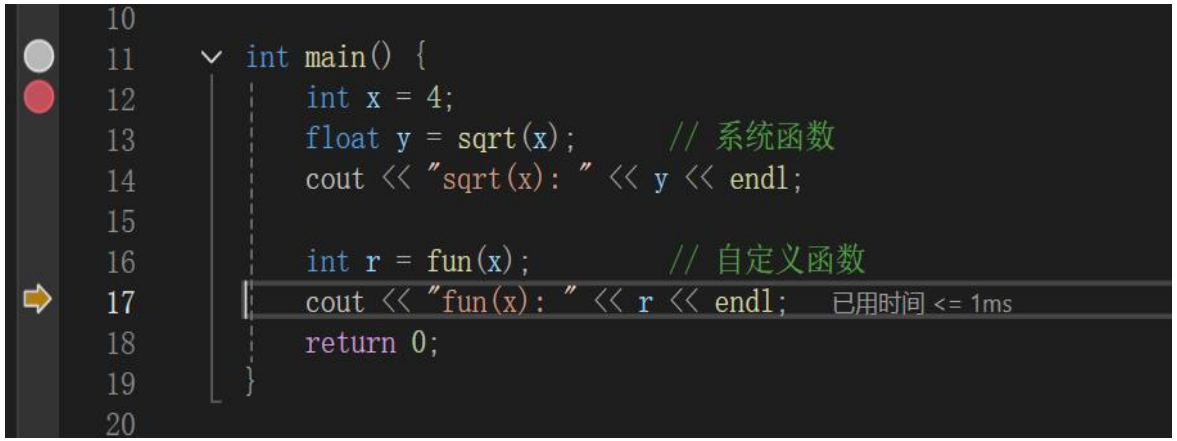
1.2 在函数中单步执行语句（知识点1.2，1.5）

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。



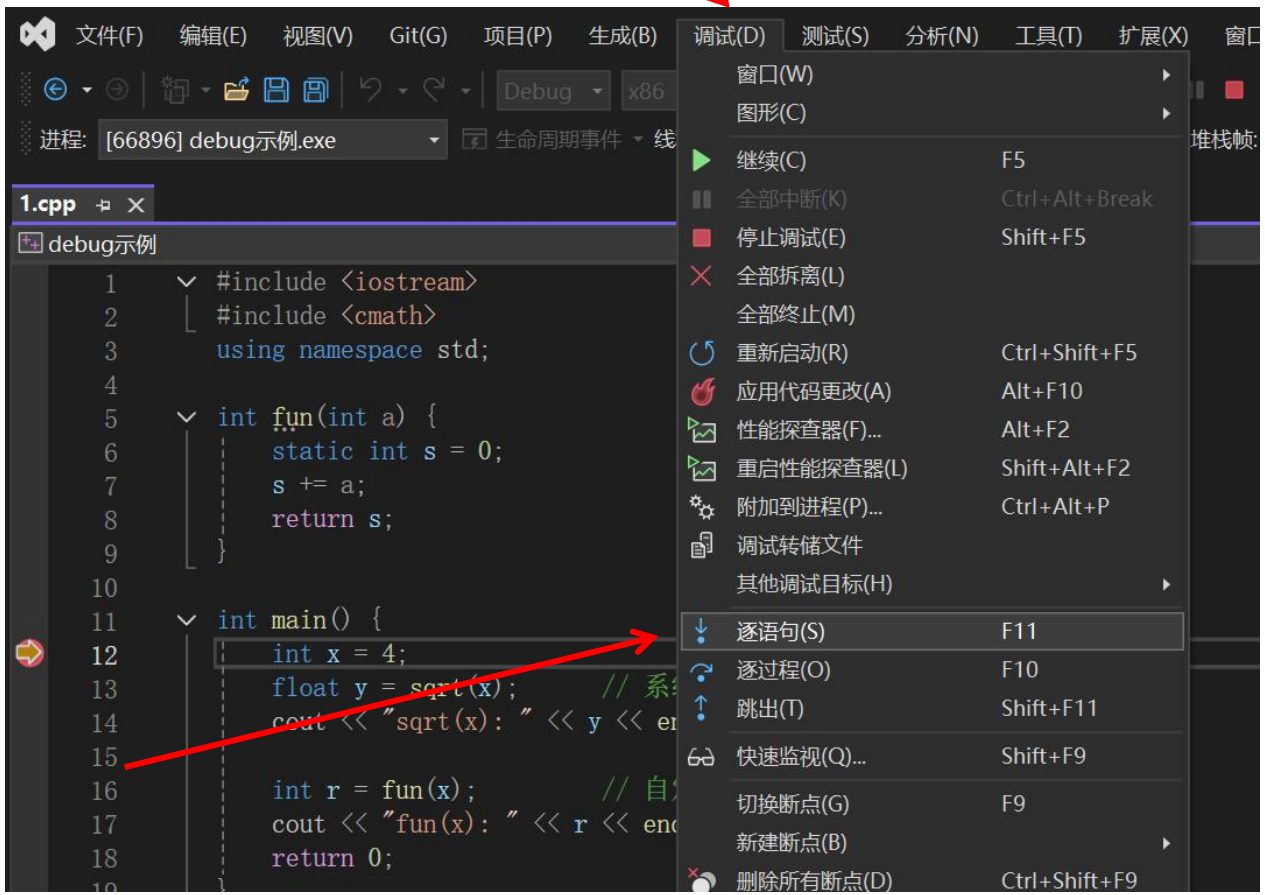


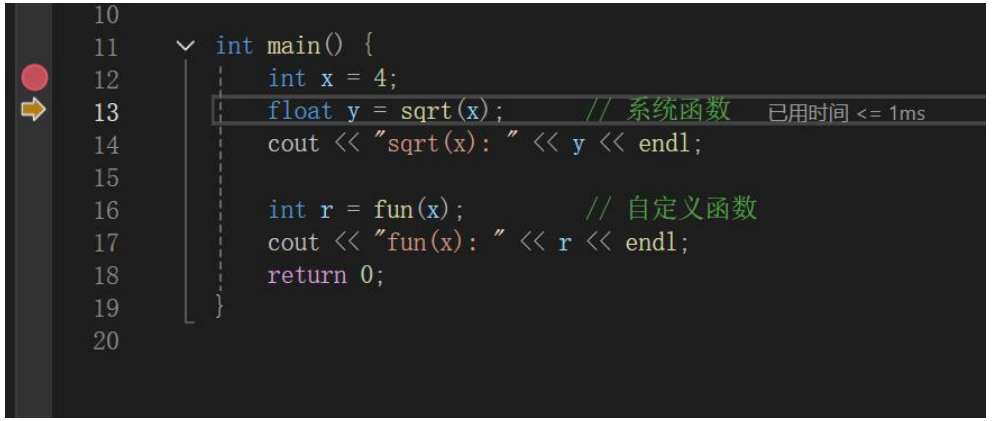


1.2 在函数中单步执行语句（知识点1.2，1.4）

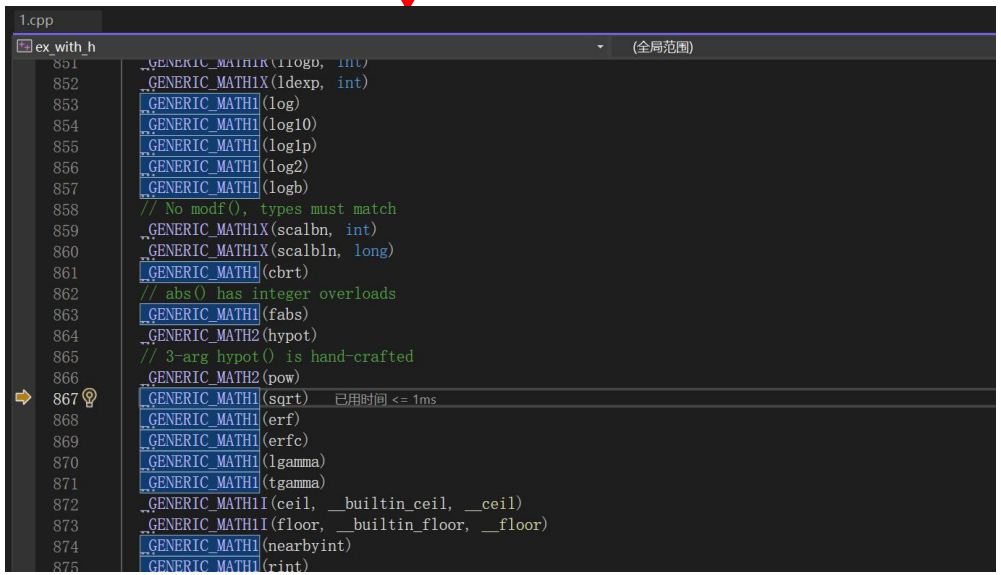
1.2.2 每个语句单步执行，进入函数内部

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。





进入sqrt函数内部



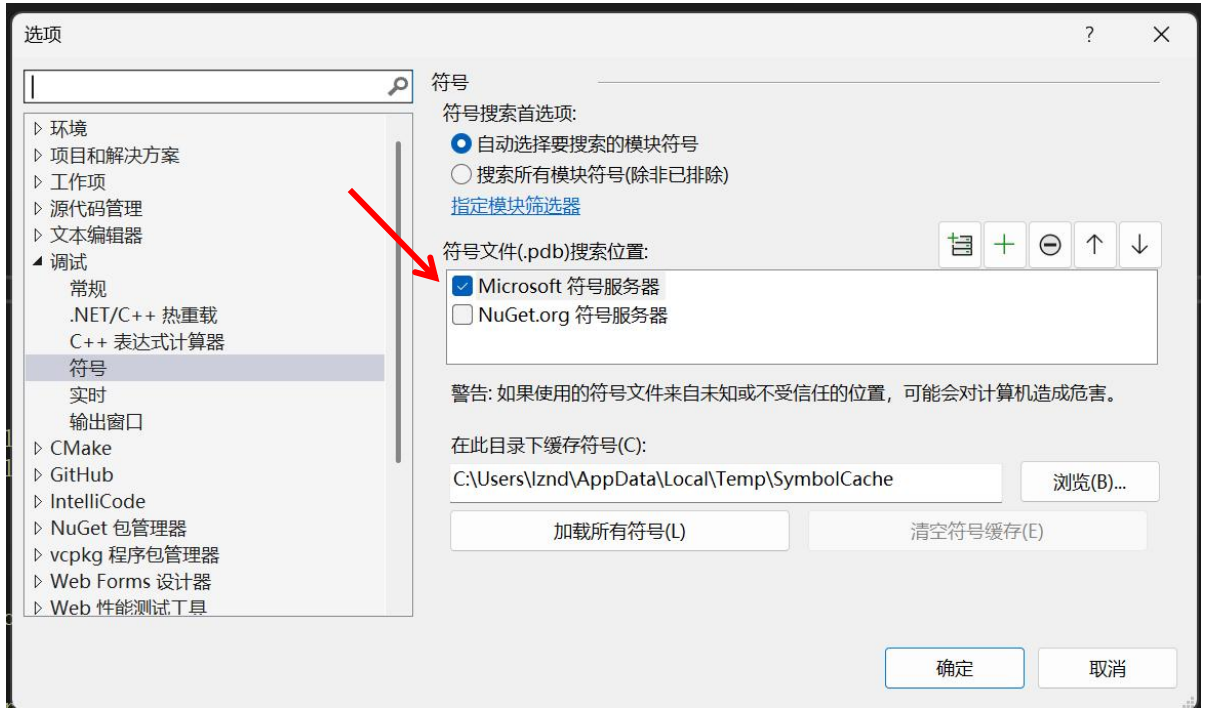
1.2 在函数中单步执行语句（知识点1.4）

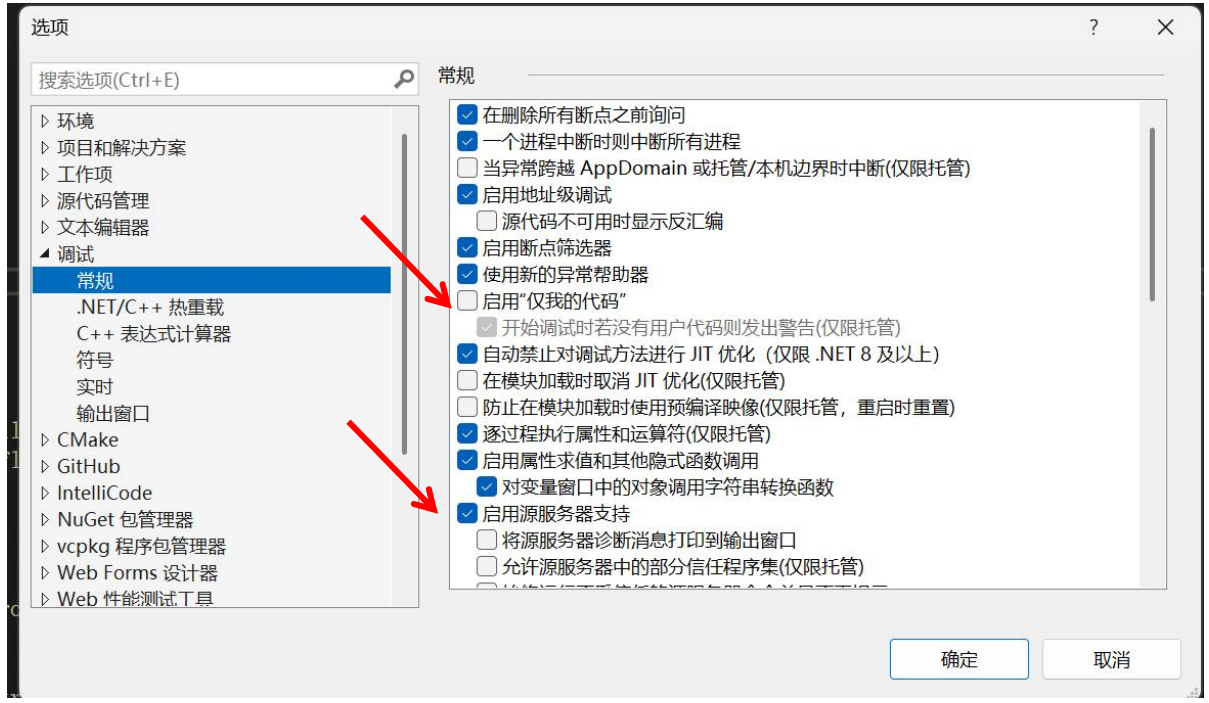
1.2.2 每个语句单步执行，进入函数内部

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

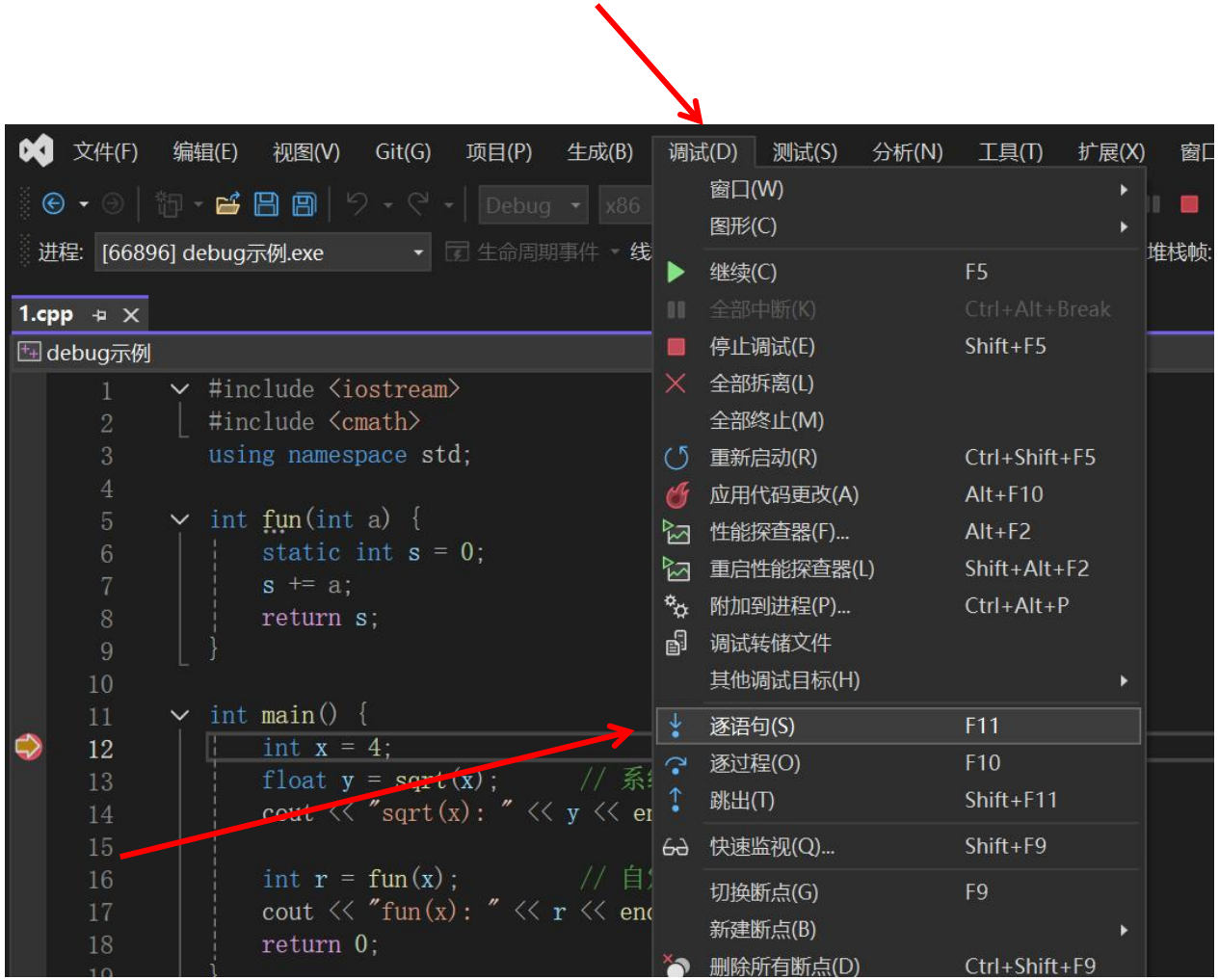




1.2 在函数中单步执行语句（知识点1.2，1.6）

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。



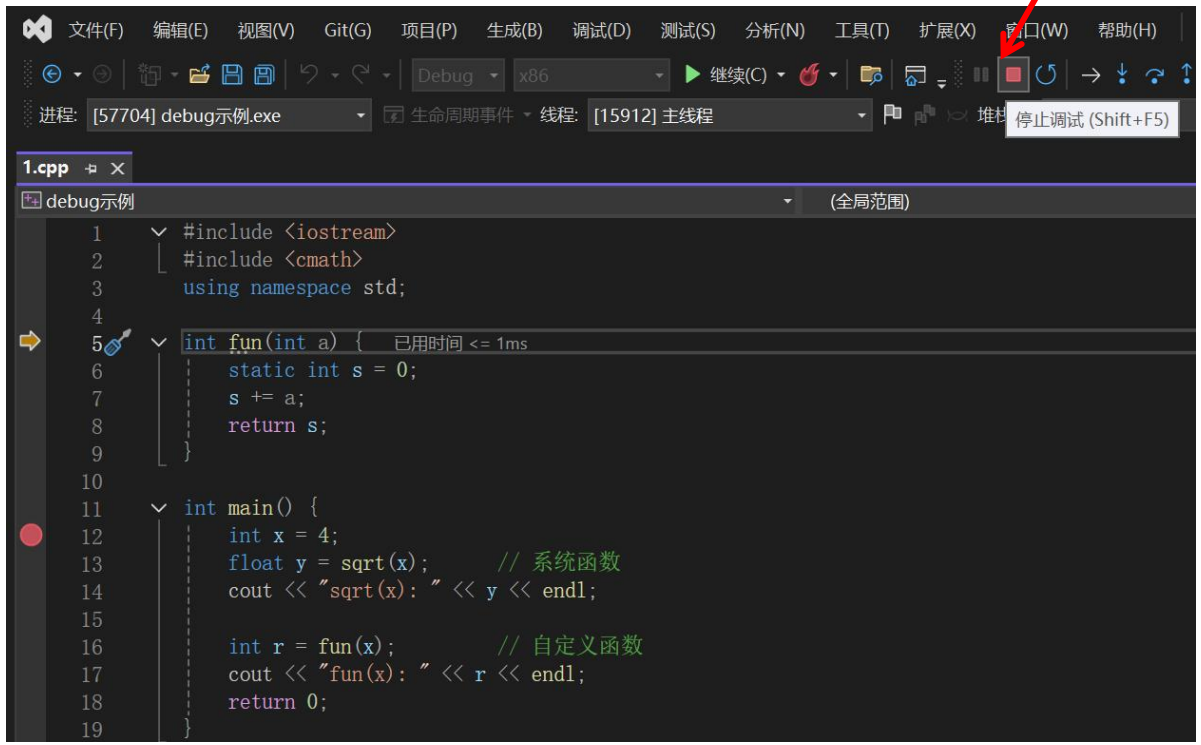
进入fun函数内部

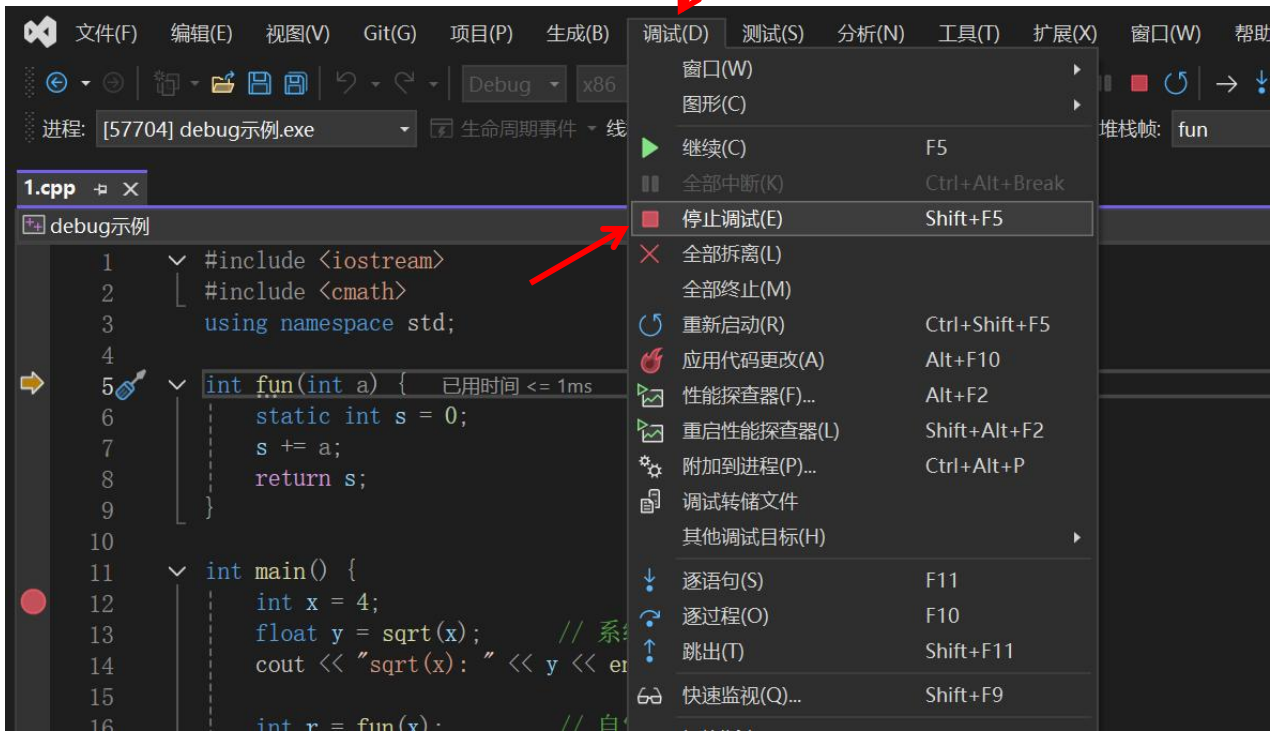
1.3 结束调试（知识点1.1）

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”





2.1 查看形参/自动变量的变化情况（知识点2.1）

main.cpp

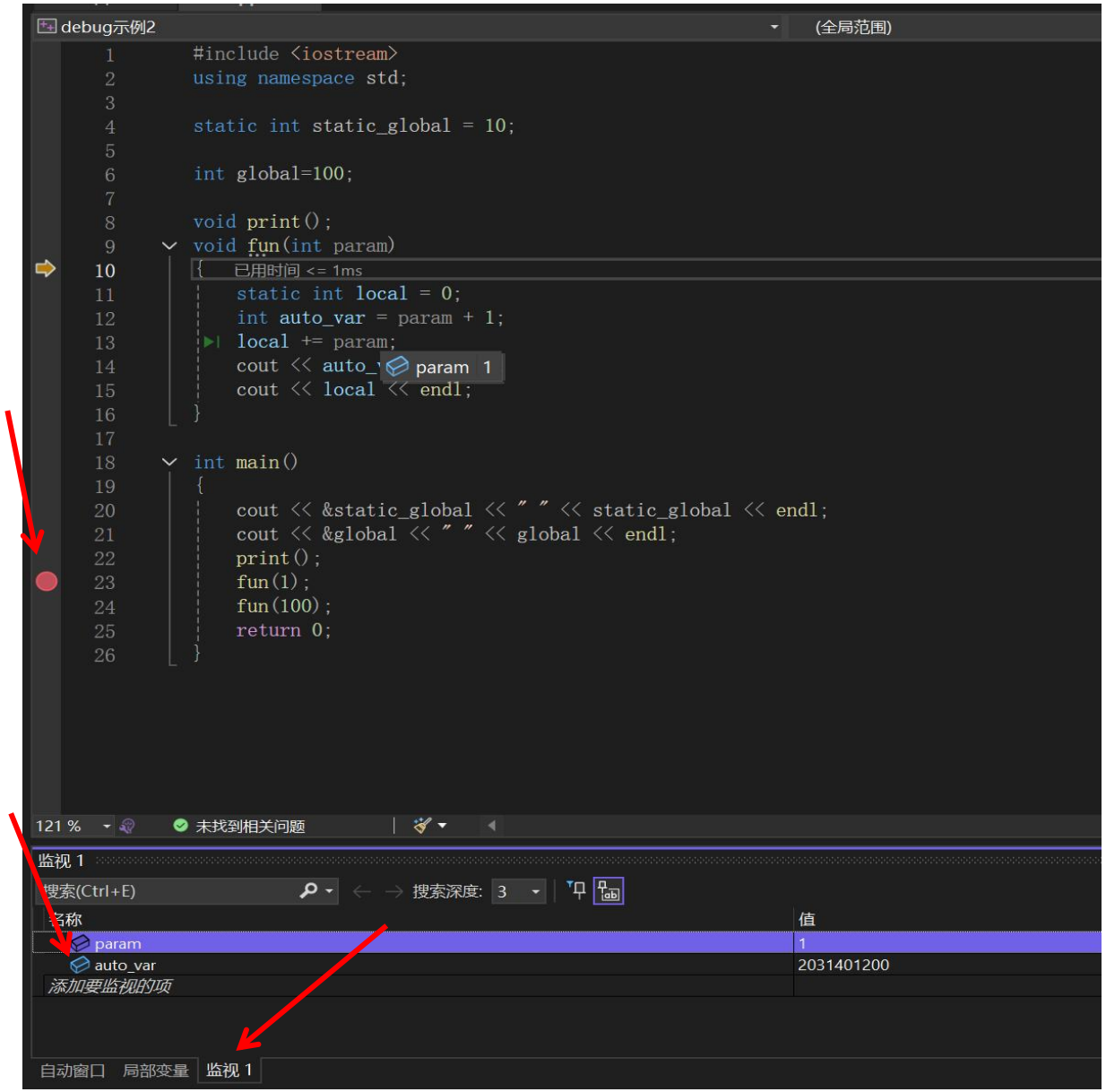
```
#include <iostream>
using namespace std;

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;
}
```

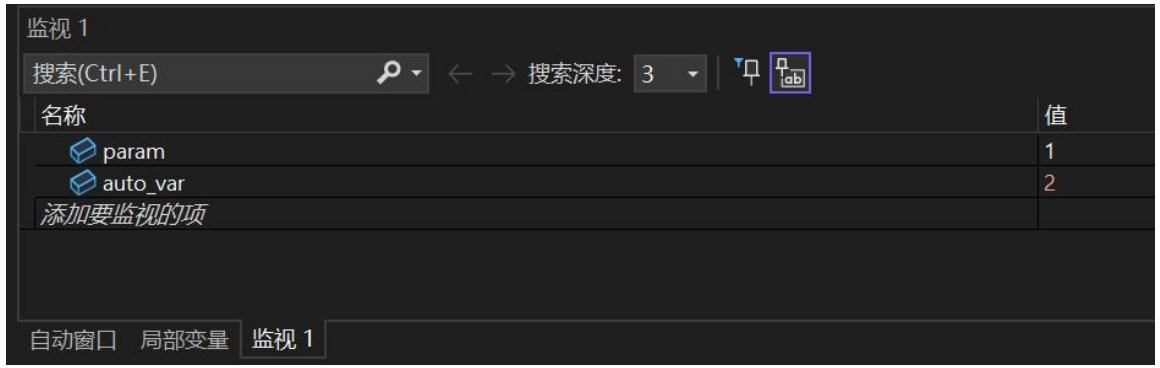
先设置断点，并开始调试

2.1 查看形参/自动变量的变化情况（知识点2.1）



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。

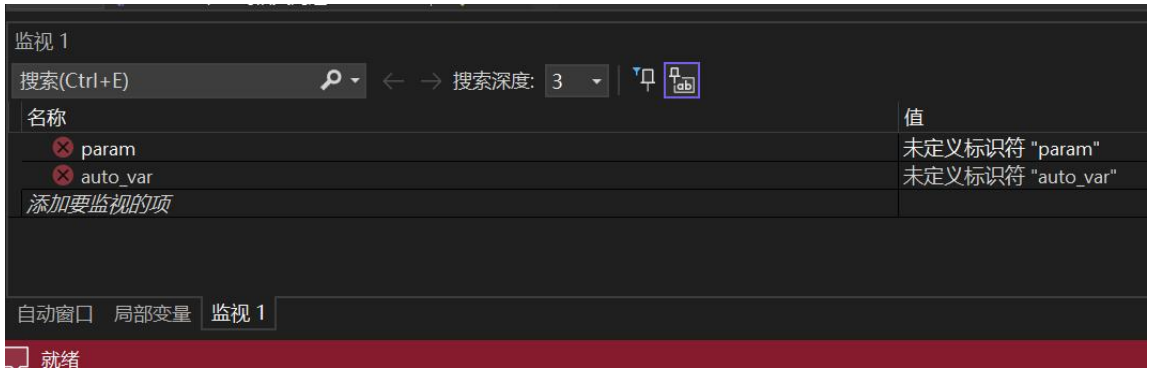
按F10运行函数后auto_var被赋值为2。



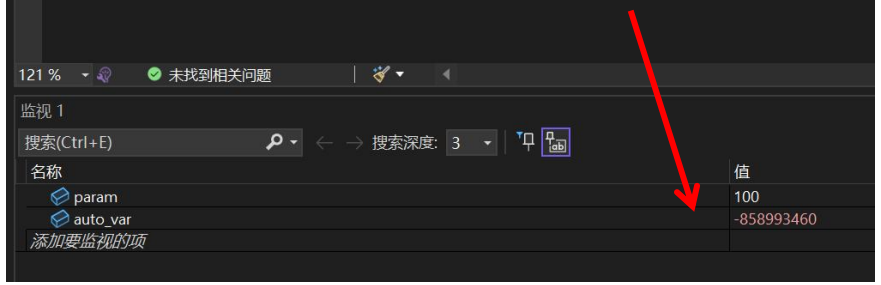
2.1 查看形参/自动变量的变化情况（知识点2.1）

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

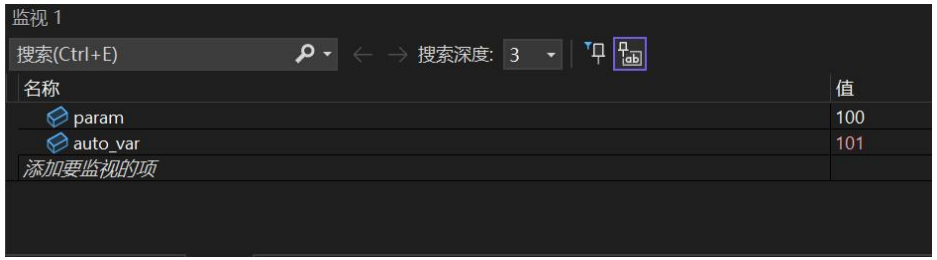
```
20      cout << &static_global << " " << stati
21      cout << &global << " " << global << en
22      print();
23      fun(1);
24      fun(100);    已用时间 <= 2ms
25      return 0;
26  }
```




```
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  }
```



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。



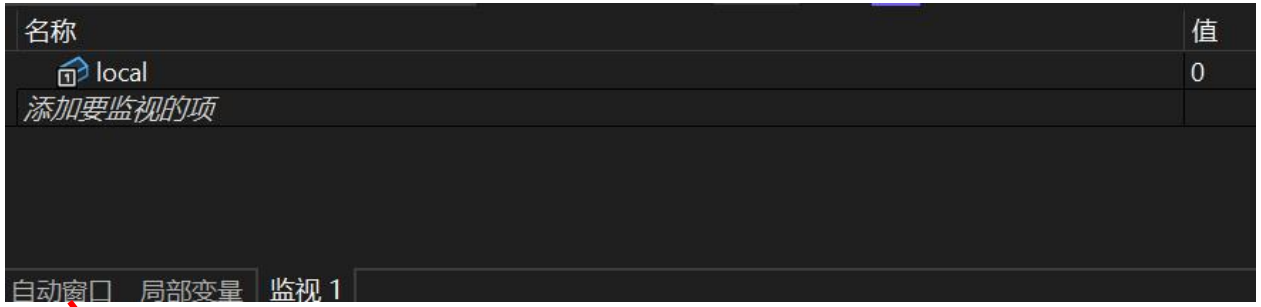
2.2 查看静态局部变量的变化（知识点2.2）

按F11进入fun（1）内部

debug示例2

```
1   #include <iostream>
2   using namespace std;
3
4   static int static_global = 10;
5
6   int global=100;
7
8   void print();
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_gl
21      cout << &global << " " << global << endl;
22      print();
23      fun(1);
24      fun(100);
25      return 0;
26  }
```

静态局部变量local的初始值为0




```
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;   已用时间 <= 1ms
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

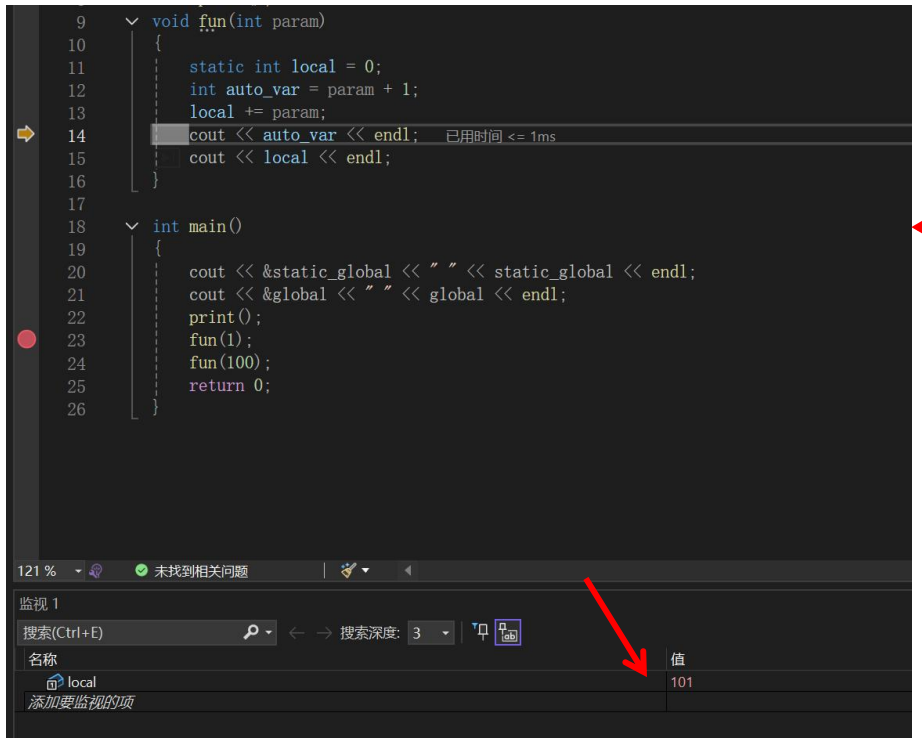
名称	值
local	1
添加要监视的项	

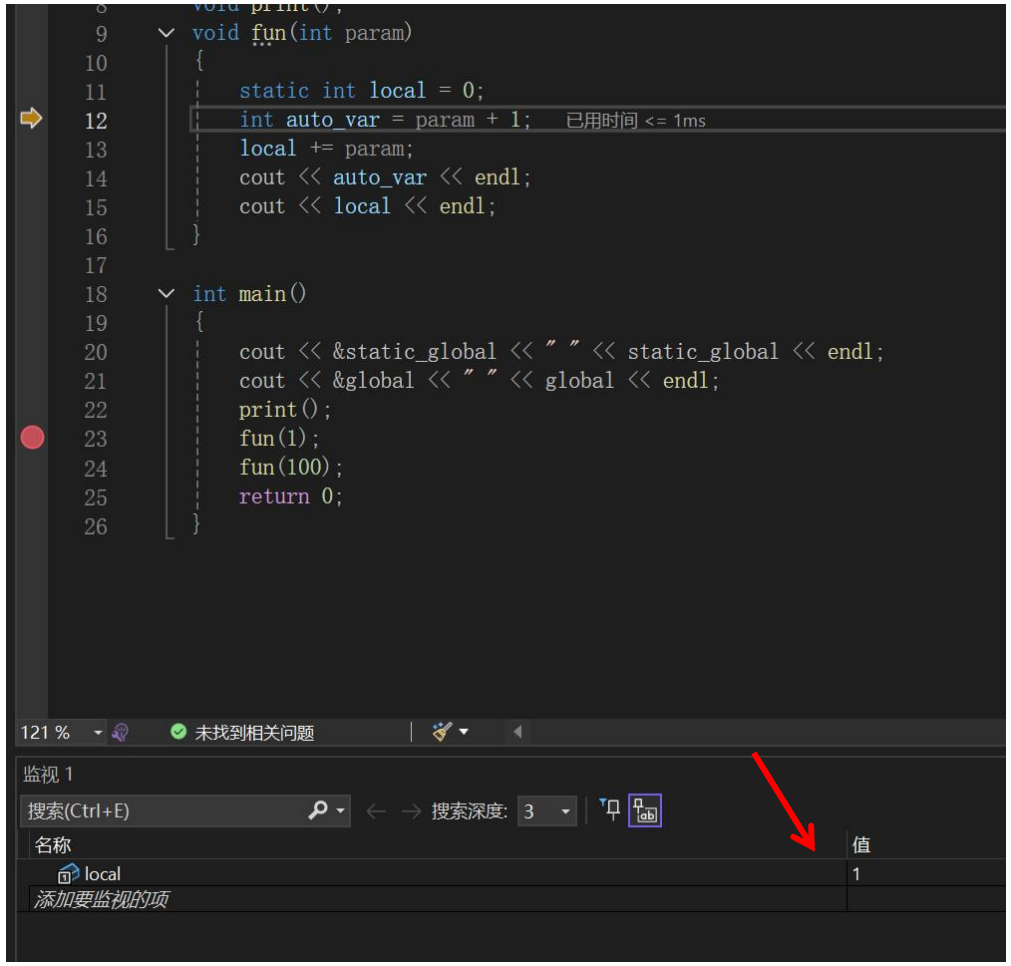
local+=param后得到local=1

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

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

debug示例2 (全局范围)

```
1   #include <iostream>
2   using namespace std;
3
4   static int static_global = 10;
5
6   int global=100;
7
8   void print();
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  }
```

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

debug示例2 (全局范围)

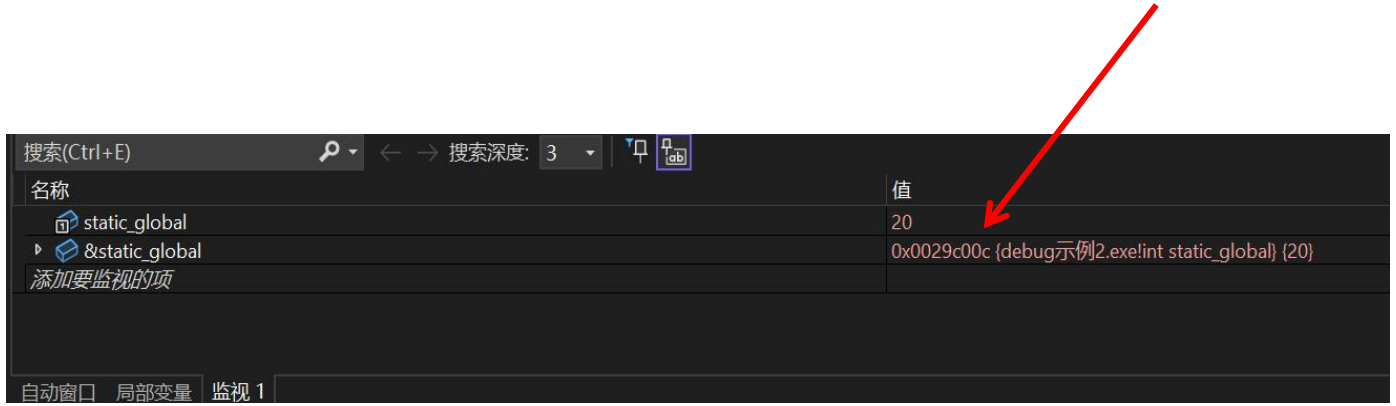
```
1   #include <iostream>
2   using namespace std;
3
4   static int static_global = 20;
5
6   extern int global;
7
8   void print()
9   {   已用时间 <= 1ms
10      global += 1;
11      cout << &static_global << " " << static_global << endl;
12      cout << &global << " " << global << endl;
13  }
```

2.3 查看静态全局变量的变化（知识点2.3）

debug示例2 (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   {   已用时间 <= 1ms
10      global += 1;
11      cout << &static_global << " " << static_global << endl;
12      cout << &global << " " << global << endl;
13  }
```

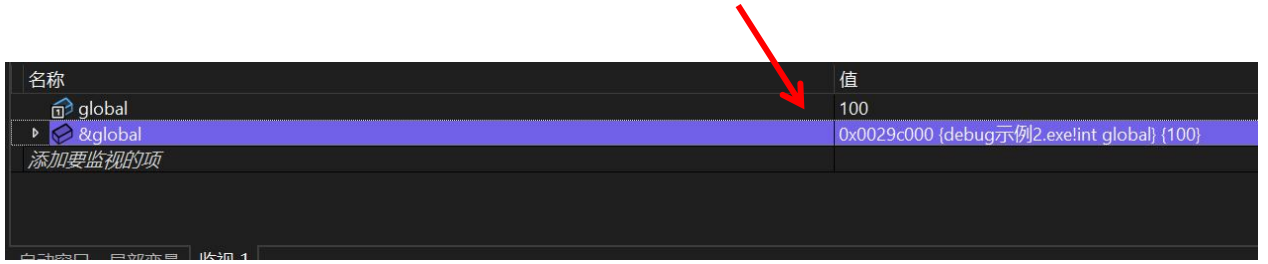
结论：在other.cpp中，static_global=20,而且地址与main.cpp里的不一样，这说明两个不同源文件里同名的静态全局变量是两个独立的变量



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

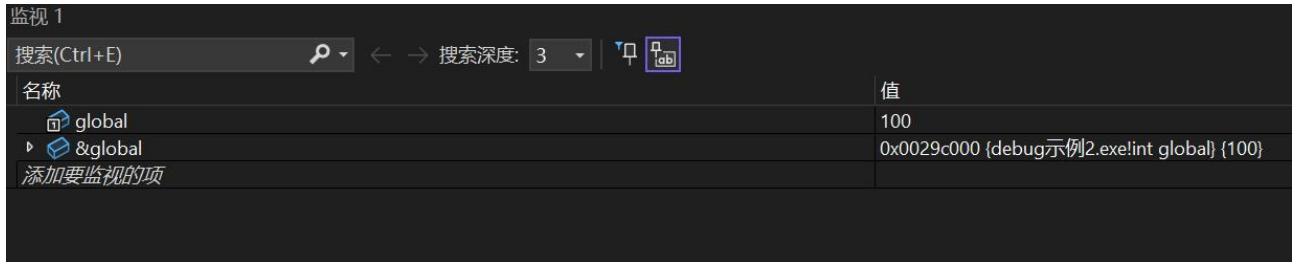
```
13      local += param;
14      cout << auto_var << endl;
15      cout << local << endl;
16  }
17
18  int main()
19  {
20      cout << &static_global << " " << static_global << endl;
21      cout << &global << " " << global << endl;
22      print();
23      fun(1);
24      fun(100);
25      return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



按F11进入other.cpp

```
debug示例2                                   (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   {   已用时间 <= 1ms
10      global += 1;
11      cout << &static_global << " " << static_global << endl;
12      cout << &global << " " << global << endl;
13  }
```

global的值和地址都不变



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

debug示例2 (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;   已用时间 <= 1ms
12      cout << &global << " " << global << endl;
13  }
```

监视 1

搜索(Ctrl+E) 搜索深度: 3

名称	值
global	101
&global	0x0029c000 {debug示例2.exe!int global} {101}
添加要监视的项	

按shift+F11退出other.cpp

```
17
18  int main()
19  {
20      cout << &static_global << " " << static_global << endl;
21      cout << &global << " " << global << endl;
22      print();   已用时间 <= 10ms
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	101
&global	0x0029c000 {debug示例2.exe!int global} {101}
添加要监视的项	

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
素
		}
		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  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
```

在监视窗口输入变量名c，n，f

监视 1

搜索(Ctrl+E)　搜索深度: 3

名称	值
c	-52 '?'
n	-858993460
f	-107374176.
添加要监视的项	

监视 1

搜索(Ctrl+E)　搜索深度: 3

名称	值
c	65 'A'
n	42
f	3.14000010
添加要监视的项	

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
```

在监视窗口输入p，*p，pc，*pc，此时还没有定义

名称	值
p	0xcccccccc {???}
*p	<无法读取内存>
pc	0xcccccccc <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

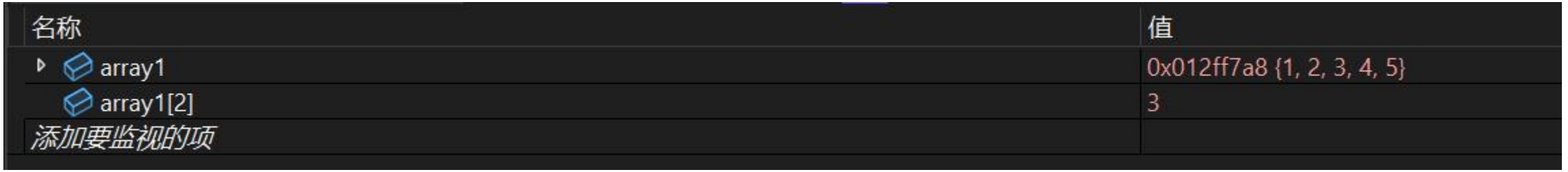
名称	值
p	0x012ff7e8 {42}
*p	42
pc	0x012ff7f7 <字符串中的字符无效。>
*pc	65 'A'
添加要监视的项	

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

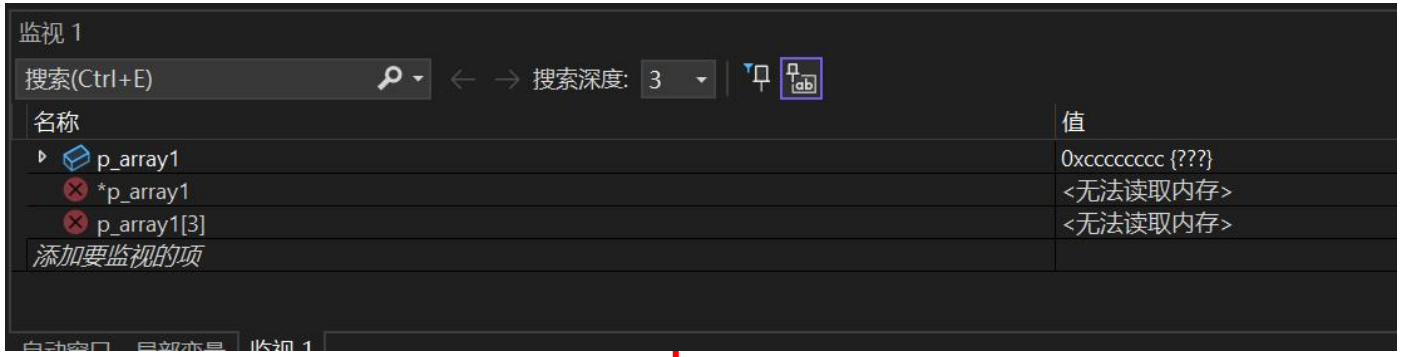
运行后得到一维数组的值及其内部单个元素的值



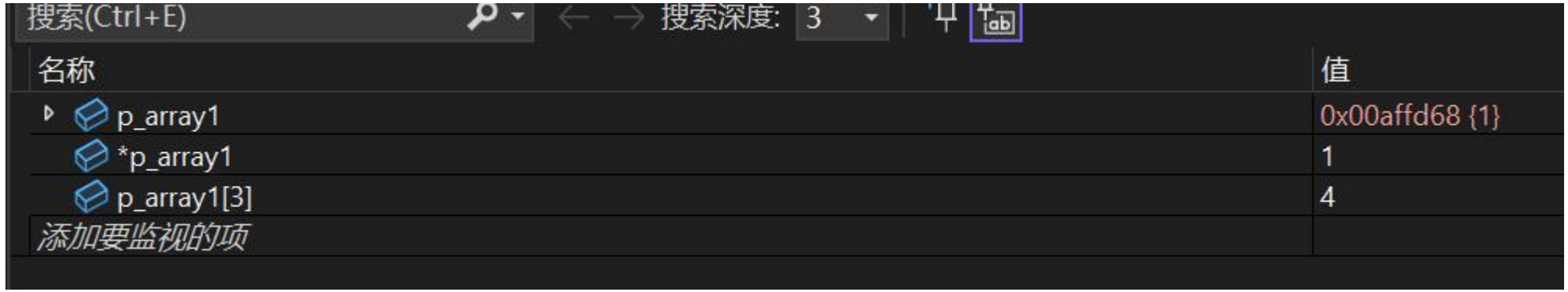
3.4 查看指向一维数组的指针变量（知识点3.4）

```
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_array[3]



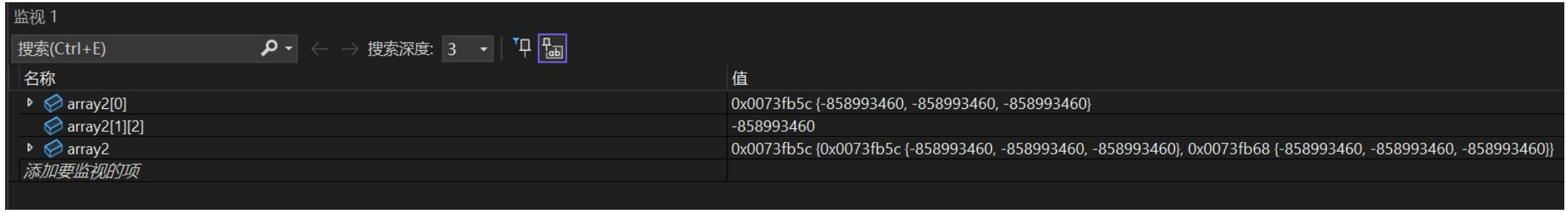
运行后得到各个值，p_array1是一维数组起始地址，*p_array1是数组第一个值，p_array1[3]是数组第四个值



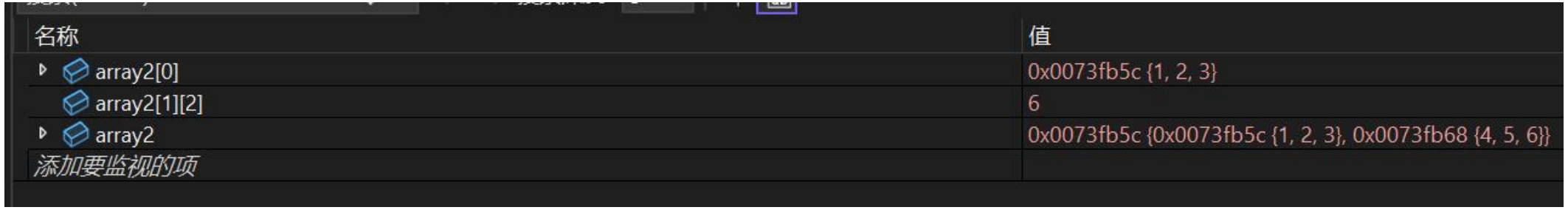
3.5 查看二维数组（知识点3.5）

```
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



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

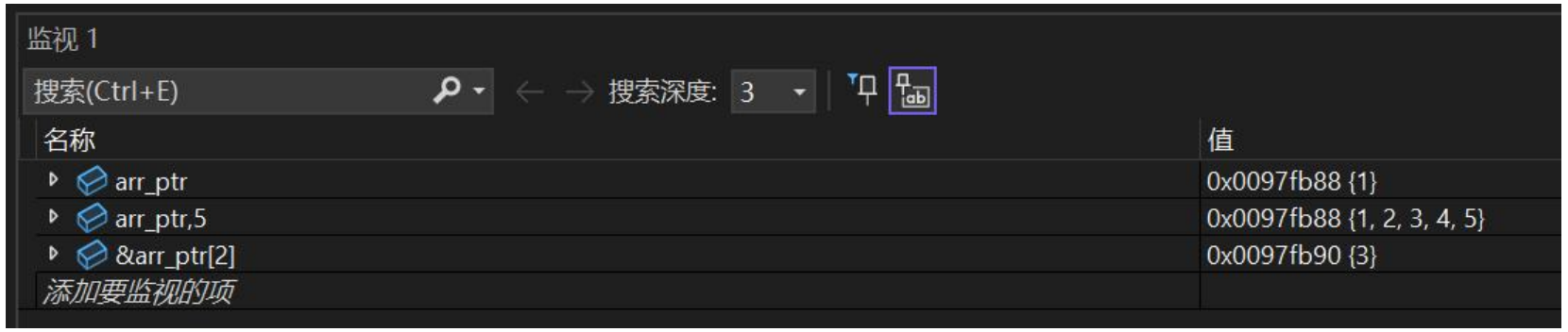


3.6 在函数中查看实参数组的地址和值（知识点3.6）

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      const char* str = "Hello";//3.7
30
```

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



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'
添加要监视的项	

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。

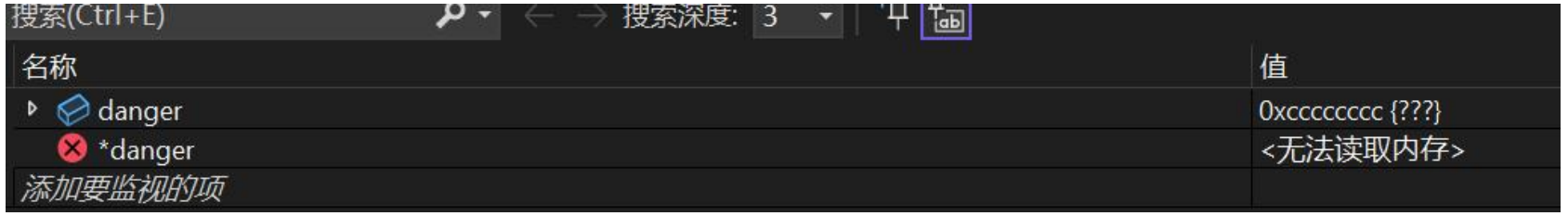
名称	值
ref	<无法读取内存>
&ref	0xcccccccc {???}
n	42
&n	0x0097fbc8 {42}
添加要监视的项	

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  }
```



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。