

同济大学

高级语言程序设计课程实验报告



实验名称:	VS2022调试工具的使用
学 院:	计算机科学与技术学院
专 业:	计算机科学与技术
学 号:	
姓 名:	
完成日期:	2024年12月8日

# 一、VS2022调试工具的基本使用方法 及

## 二、用 VS2022的调试工具查看各种生存期/作用域变量

- 开始调试主要有两种方法：

1. 设置**断点**，按**F5**：直接执行到断点处，然后进行逐句调试或逐过程调试即可。（如上图）
2. 按**F10**：从main函数的第一句开始，等待逐句调试或逐过程调试即可。

- 结束调试主要有一种方法：

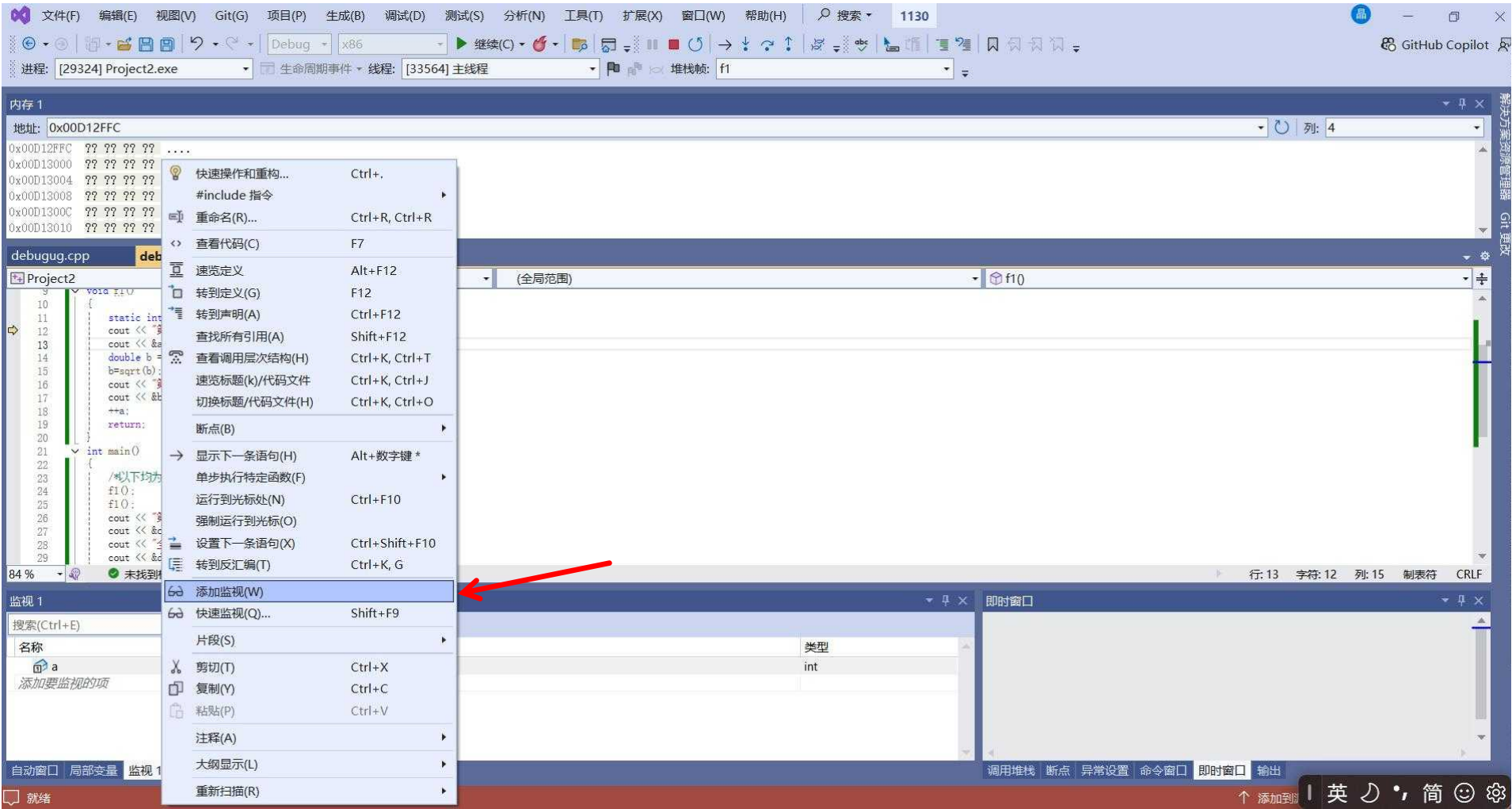
1. 快捷键**Shift+F5**：跳出调试过程。（或者如上图找对应的图标结束调试）

- 在碰到 cout/sqrt等系统类/系统函数和自定义函数的调用语句(例如在 main 中调用自定义的 fun 函数)时，按**F11**会进入函数内部单步执行；按**F10**可以直接不进入到函数的内部单步执行；如果已经进入，则按**Shift+F11**可以跳出并返回自己的函数。

(下面将结合具体的程序调试过程演示说明知识点一、二)

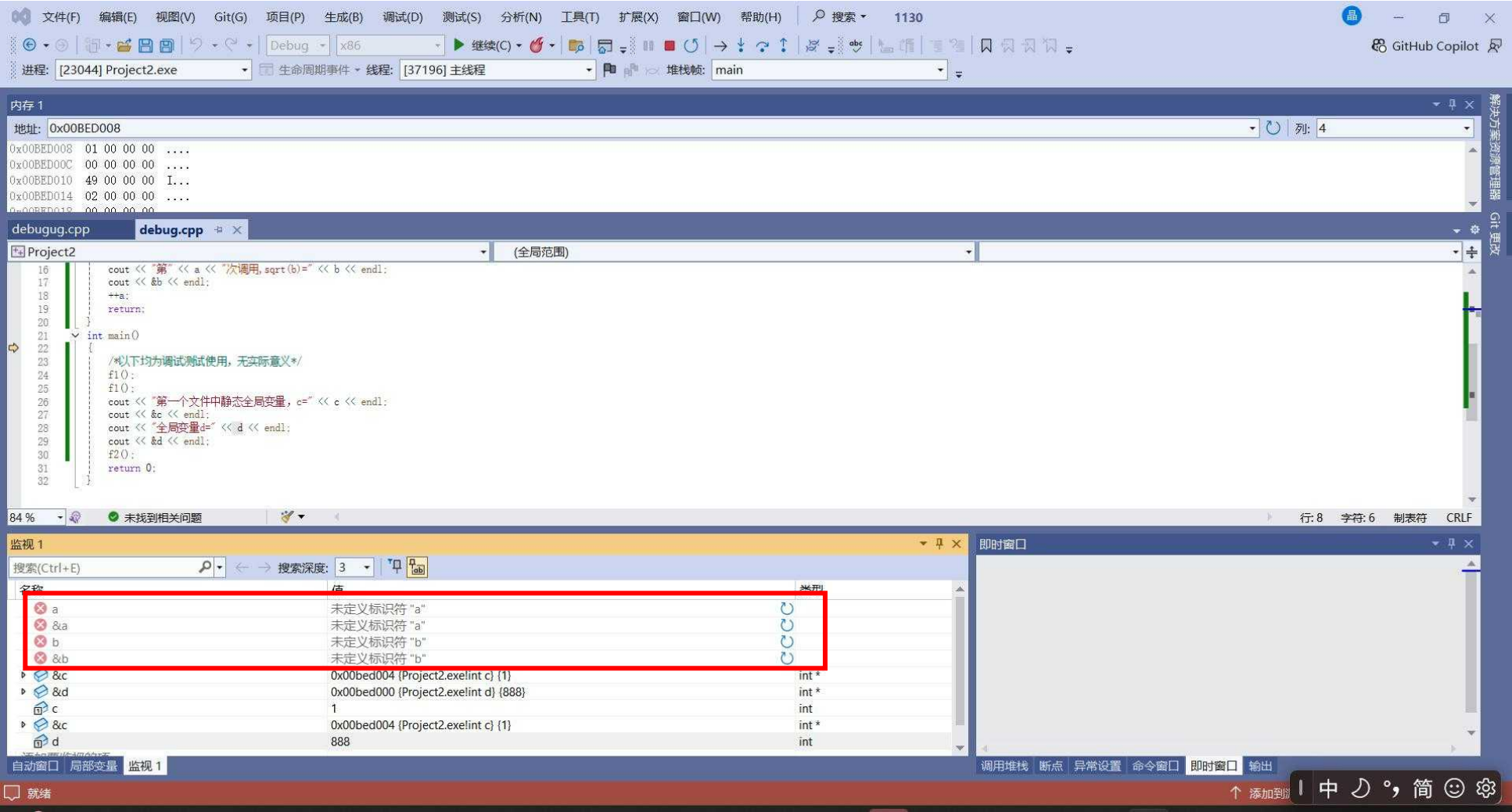
第1步：按F10进入main函数，开始调试。

1-1：给所有变量添加监视，便于观察生存期和作用域的变化



光标选中  
要被监视  
的变量，  
右击鼠标，  
选择“添  
加监视”

第1步：按F10进入main函数，开始调试。  
1-2：观察各变量的值和地址变化

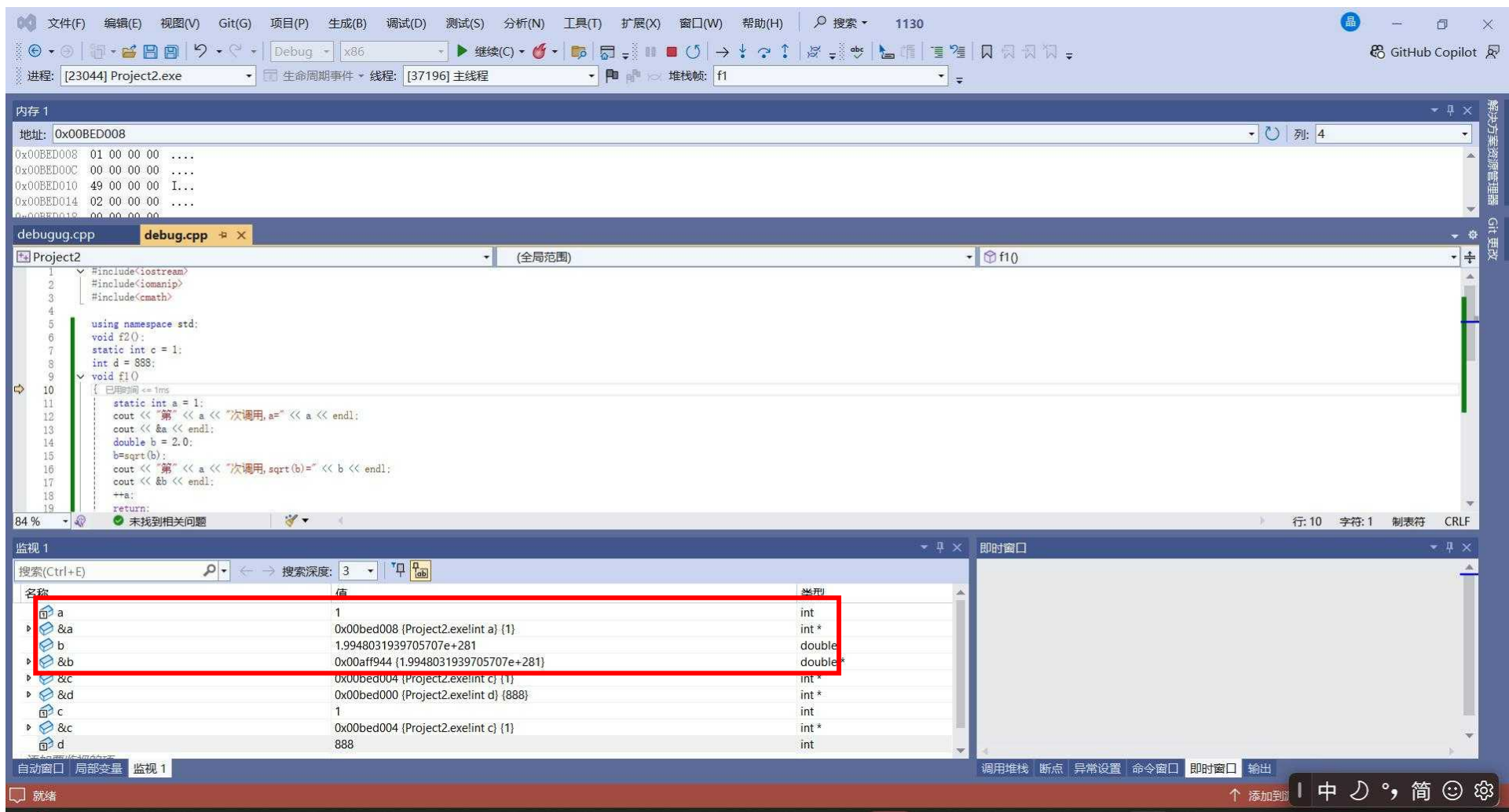


如图所示可以看到，一开始进入程序，f1（）函数未调用，故在f1（）函数中的局部自动变量呈未定义状态。而静态局部变量a、静态全局变量c、外部全局变量d生存期已开始。

查看各变量生存期/作用域的方法就是，对这些变量和其地址添加监视，从而能在单步调试中观察变量值和地址的变化。

本图涉及知识点1.1/1.2/2.1/2.2

第2步：当箭头指向f1（）的调用语句时，按F11，进入函数。



如图所示可以看到，一进入f1（），静态局部变量a和局部自动变量b就存在了。

本图涉及知识点 2.1/2.2



第3步：按F10单步执行，直到b=sqrt(b).按F10，不进入到函数的内部，直接得到返回值。

The image displays two screenshots of the Visual Studio IDE, illustrating the execution of a C++ program using the F10 (Step Over) debugging command.

**Left Screenshot:** The source code in `debug.cpp` is shown. Line 15, `b=sqrt(b);`, is highlighted with a red box. The 'Watch' window (监视 1) displays the following variables and their values:

名称	值	类型
a	1	int
&a	0x00bed008 (Project2.exe!int a) (1)	int *
b	2.0000000000000000	double
&b	0x00aff944 (2.0000000000000000)	double *
&c	0x00bed004 (Project2.exe!int c) (1)	int *
d	0x00bed000 (Project2.exe!int d) {888}	int *
c	1	int
&c	0x00bed004 (Project2.exe!int c) (1)	int *
d	888	int

**Right Screenshot:** After pressing F10, the execution has moved to line 16, `cout << "第" << a << "次调用, sqrt(b) = " << b << endl;`, which is also highlighted with a red box. The 'Watch' window (监视 1) now shows the updated value of b:

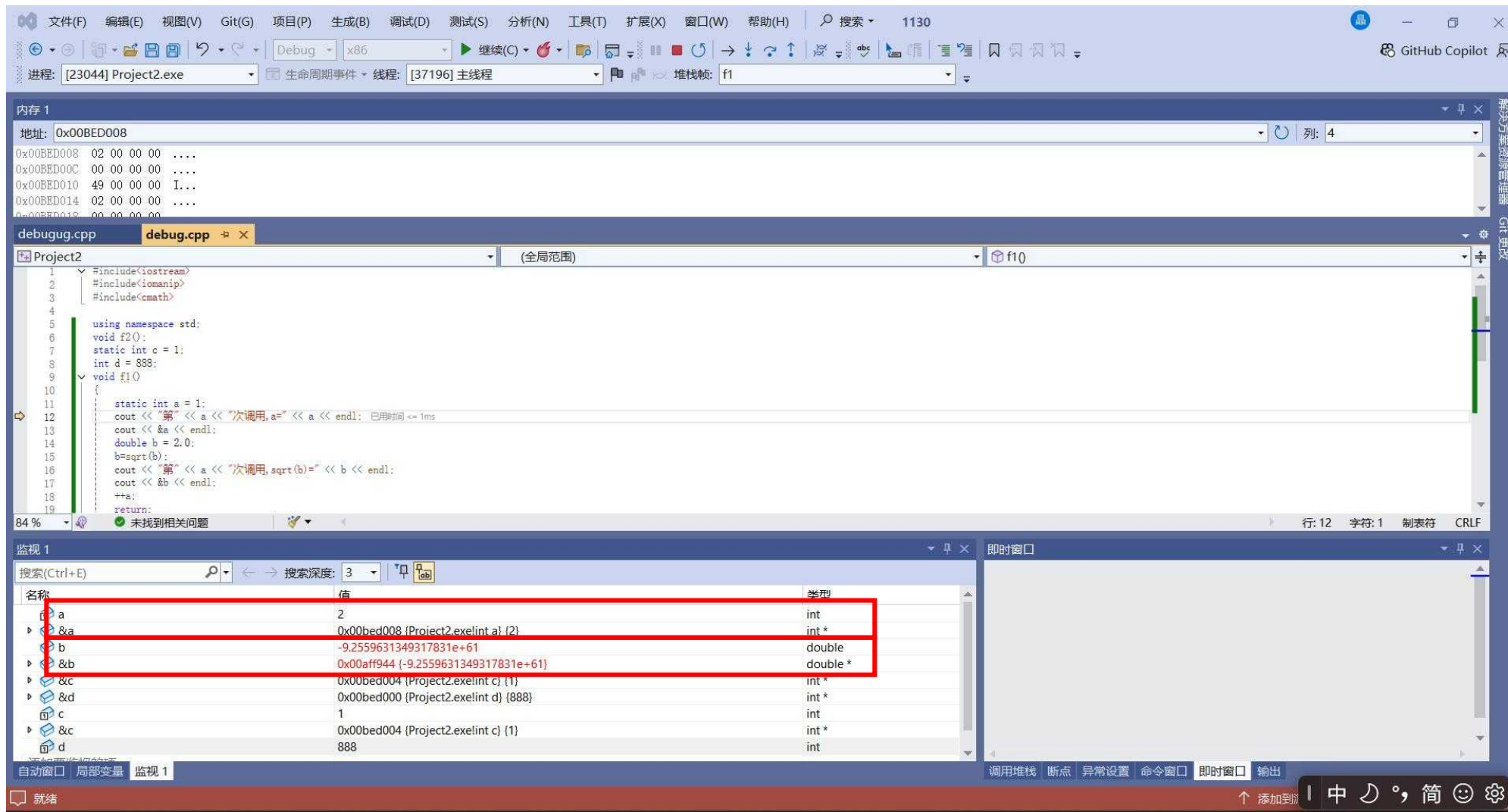
名称	值	类型
a	1	int
&a	0x00bed008 (Project2.exe!int a) (1)	int *
b	1.4142135623730951	double
&b	0x00aff944 (1.4142135623730951)	double *
&c	0x00bed004 (Project2.exe!int c) (1)	int *
d	0x00bed000 (Project2.exe!int d) {888}	int *
c	1	int
&c	0x00bed004 (Project2.exe!int c) (1)	int *
d	888	int

A red arrow points from the first screenshot to the second, indicating the progression of the debugging process.

本图涉及知识点 1.2/1.3/1.5

第4步：按Shift+F11退出f1（）函数，返回到main函数中去  
本步骤涉及知识点 1.4/1.6（因为系统函数退出操作一样的）

第5步：重复步骤2，再次进入f1（）函数。



如图所示可以看到，再次进入f1（）函数时，**静态局部变量**不会重新赋值，而**局部自动变量**的值会重新赋。

并且可以看到，此时b的赋值定义语句还未执行，b的值是随机的。可以推断得出，**局部自动变量**的存储空间当函数结束后会回收，下一次调用时候再重新分配，重新赋值。而**静态局部变量**不会。

第6步：重复步骤4，退出f1（）函数，回到main函数。再按F10单步执行到f2（）函数的调用语句。

文件(F) 编辑(E) 视图(V) Git(G) 项目(P) 生成(B) 调试(D) 测试(S) 分析(N) 工具(T) 扩展(X) 窗口(W) 帮助(H) 1130

进程: [23044] Project2.exe 生命周期事件 线程: [37196] 主线程 堆栈帧: main

内存 1

地址: 0x00BED008 列: 4

0x00BED008	03 00 00 00	....
0x00BED00C	00 00 00 00	....
0x00BED010	49 00 00 00	I...
0x00BED014	02 00 00 00	....
0x00BED018	00 00 00 00	....

debug.cpp

Project2 (全局范围) main()

```
17 cout << &a << endl;
18 ++a;
19 return;
20 }
21 int main()
22 {
23     /*以下均为调试测试使用, 无实际意义*/
24     f1();
25     f1();
26     cout << "第一个文件中静态全局变量, c=" << c << endl;
27     cout << &c << endl;
28     cout << "全局变量d=" << d << endl;
29     cout << &d << endl;
30     f2(); 已用时间 <= 1ms
31     return 0;
32 }
```

84 % 未找到相关问题 行: 30 字符: 1 制表符 CRLF

监视 1

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	3	int
&a	0x00bed008 (Project2.exe!int a) {3}	int *
b	1.4142135623730951	double
&b	0x00aff944 {1.4142135623730951}	double *
&c	0x00bed004 (Project2.exe!int c) {1}	int *
&d	0x00bed000 (Project2.exe!int d) {888}	int *
c	1	int
&c	0x00bed004 (Project2.exe!int c) {1}	int *
d	888	int

即时窗口

此时, a、b变灰, 表示不可访问。

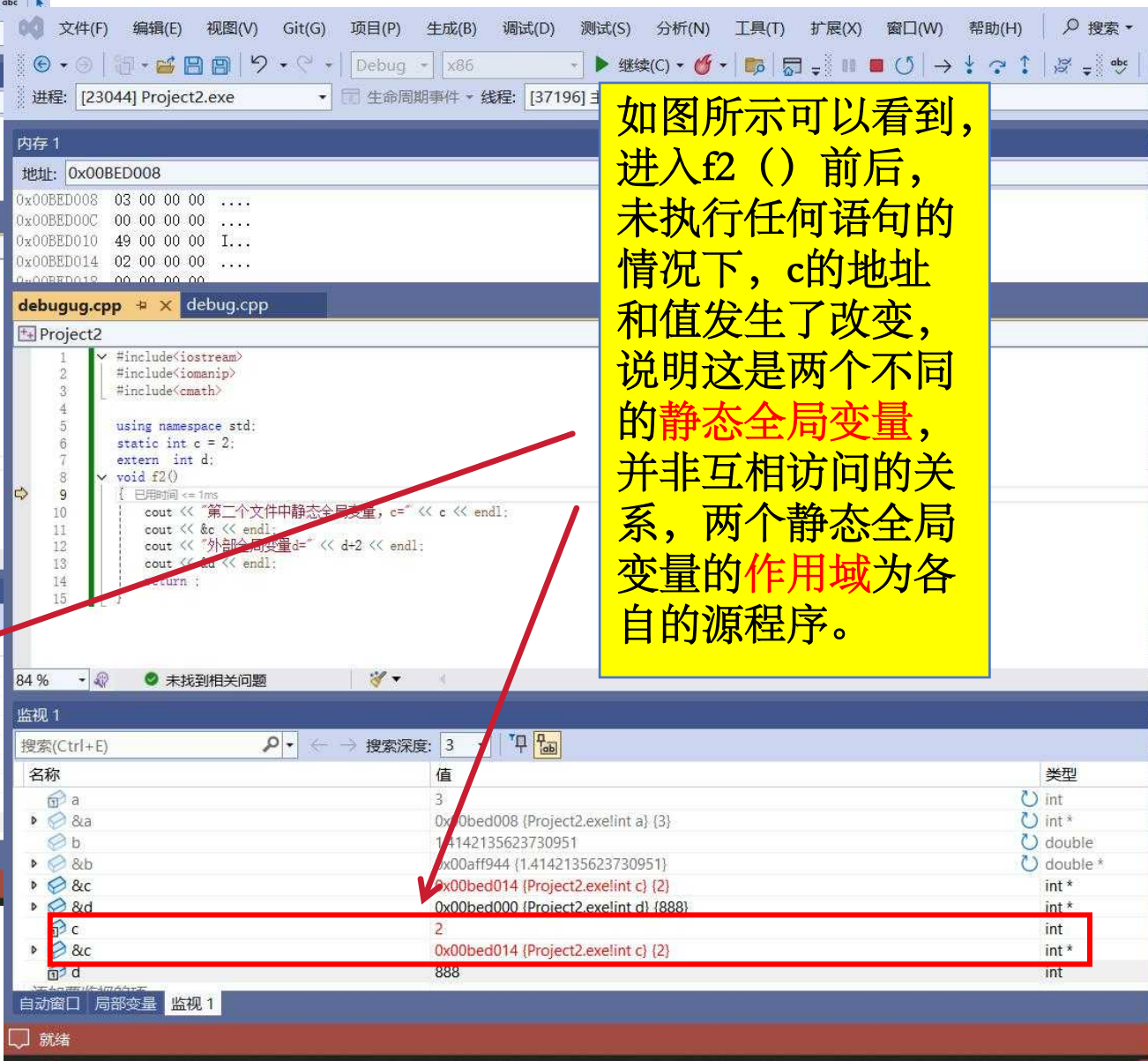
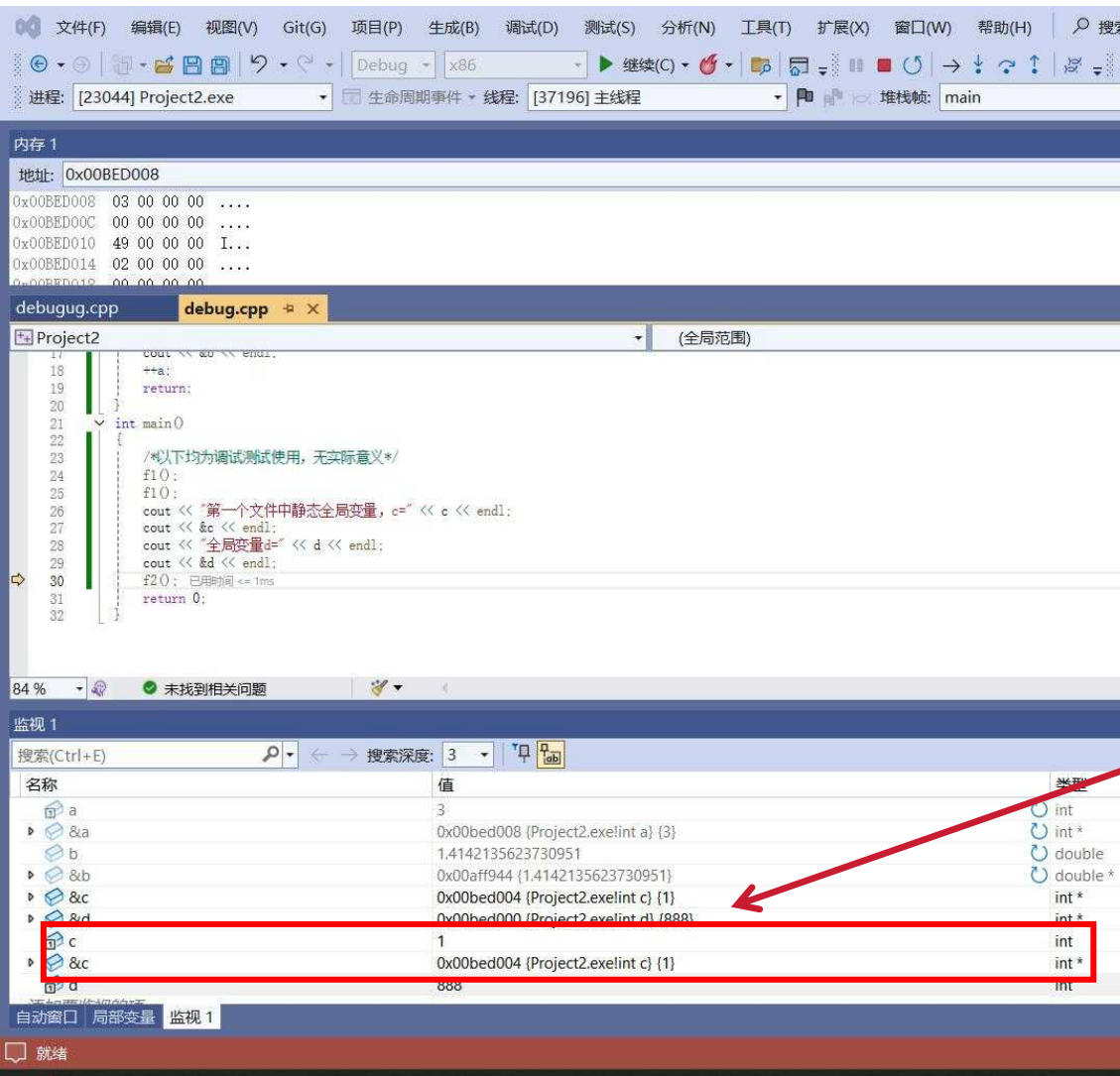
自动窗口 局部变量 监视 1 调用堆栈 断点 异常设置 命令窗口 即时窗口 输出

就绪 添加到 中 简 舒



第7步：按F11进入f2（）函数。（f2定义在另一个源程序中）

(PS：我多监视了一个c，其实没必要哈)



如图所示可以看到，进入f2（）前后，未执行任何语句的情况下，c的地址和值发生了改变，说明这是两个不同的静态全局变量，并非互相访问的关系，两个静态全局变量的作用域为各自的源程序。

本图涉及知识点 2.3

第8步：按F10单步执行。

The screenshot shows the Visual Studio IDE with the following components:

- Code Editor:** Displays the source code of `debugug.cpp`. The code includes `<iostream>`, `<iomanip>`, and `<cmath>`. It defines a static integer `c = 2` and an external integer `d`. The `f2()` function prints the value of `c`, increments `d` by 2, and prints the new value of `d`.
- Memory Window (内存 1):** Shows the memory address `0x00BED008` and its contents. A red arrow points from the `&d` variable in the code to this memory location.
- Variable Watcher (监视 1):** Displays the values of variables `a`, `b`, `c`, and `d`. Red boxes highlight the `&d` and `d` entries, with red arrows pointing to the memory window.
- Callout Box:** A yellow box with red text stating: "如图所示可以看到，d=d+2; 语句前后，d的地址没有改变，说明是从外部访问了全局变量d，并进行了更改。"
- Output Window:** Shows the program's output, including the message "第二个文件中静态全局变量，c= 2" and the updated value of `d`.

本图涉及知识点 2.4

第9步：按F10进入main函数。（然后可以继续按F10，运行完结束；也可以直接按F5，也可以直接运行完结束（因为本题没有设置断点））

● 变量的生存期、作用域、存储区

	生存期	作用域	存储区
自动变量	本函数	本函数	动态数据区
形参	本函数	本函数	动态数据区
静态局部变量	整个程序执行中	本函数	静态数据区
静态全局变量	整个程序执行中	本源程序文件	静态数据区
外部全局变量	整个程序执行中	全部源程序文件	静态数据去

## debug.cpp

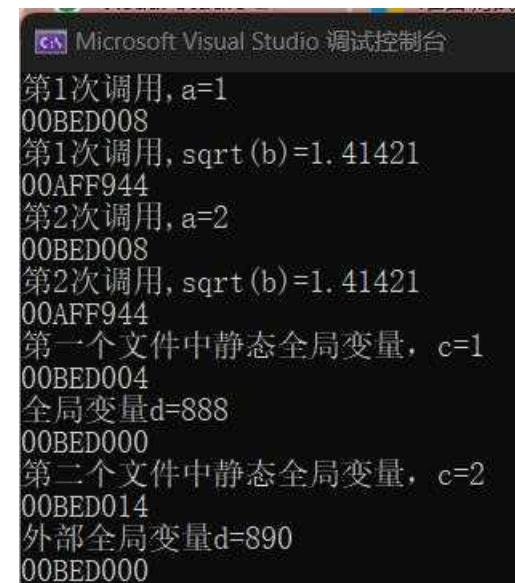
```
#include<iostream>
#include<iomanip>
#include<cmath>
using namespace std;
void f2();
static int c = 1;
int d = 888;
void f1()
{
    static int a = 1;
    cout << "第" << a << "次调用,a=" << a << endl;
    cout << &a << endl;
    double b = 2.0;
    b=sqrt(b);
    cout << "第" << a << "次调用,sqrt(b)=" << b << endl;
    cout << &b << endl;
    ++a;
    return;
}
int main()
{
    /*以下均为调试测试使用，无实际意义*/
    f1();
    f1();
    cout << "第一个文件中静态全局变量，c=" << c << endl;
    cout << &c << endl;
    cout << "全局变量d=" << d << endl;
    cout << &d << endl;
    f2();
    return 0;
}
```

## 测试程序及运行结果

### debugg.cpp

```
#include<iostream>
#include<iomanip>
#include<cmath>

using namespace std;
static int c = 2;
extern int d;
void f2()
{
    cout << "第二个文件中静态全局变量，c=" << c << endl;
    cout << &c << endl;
    d = d + 2;
    cout << "外部全局变量d=" << d << endl;
    cout << &d << endl;
    return ;
}
```





# 三、用 VS2022 的调试工具查看各种不同类型变量

第1步：按F10,进入main函数首句，进行逐步调试。（相比于设置断点再按F5，本测试程序更适合按F10进行逐步调试）

内存 1

地址: 0x00BED008 列: 4

0x00BED008 00 00 00 00 ....

0x00BED00C 00 00 00 00 ....

0x00BED010 00 00 00 00 ....

0x00BED014 00 00 00 00 ....

0x00BED018 00 00 00 00 ....

debug.cpp

Project2 (全局范围) main()

```
6 void f1(int *arr)
7 {
8     *(arr + 4) = 6;
9     int x = *(arr + 7); //越界访问
10    cout << x << endl;
11    return;
12}
13
14 int main()
15 {
16     int a=10,&g=a;
17     cout << "a的地址是: " << &a << endl;
18     cout << "g的地址是: " << &g << endl;
19     int* p;
20     p = &a;
21     int array[5] = { 0 };
22     int* q;
23     q = array;
24     int array2[3][10] = { 0 };
```

84 % 未找到相关问题

行: 15 字符: 1 制表符 CRLF

监视 1

名称	值	类型
a	8781032	int
&a	0x0085fcd8 (8781032)	int *
g	0	int &
p	0x0085fccc (8781024)	int *
array	0x0085fca4 (2052047205, 0, 8780988, 8780992, 2052194614)	int[5]
q	ucrtbased.dll!0x7a562c02 (加载符号以获取其他信息) (1166740203)	int *
array2	0x0085fc18 (0x0085fc18 (2052466076, -1853348680, -2, 8780852, 2052455059, -1181732747, 8, 87808...	int[3][10]
m	0x00000008 <读取字符串字符时出错。>	const char *
&g	0x0085fce0 {0}	int *

输出

显示输出来源(S): 调试

"Project2.exe" (Win32): 已加载 "C:\Users\HP\Desktop\

"Project2.exe" (Win32): 已加载 "C:\Windows\SysWOW64\

"Project2.exe" (Win32): 已加载 "C:\Windows\SysWOW64\

"Project2.exe" (Win32): 已加载 "C:\Windows\SysWOW64\

"Project2.exe" (Win32): 已加载 "C:\Windows\SysWOW64\

"Project2.exe" (Win32): 已加载 "C:\Windows\SysWOW64\

"Project2.exe" (Win32): 已加载 "C:\Windows\SysWOW64\

线程 19364 已退出, 返回值为 0 (0x0)。

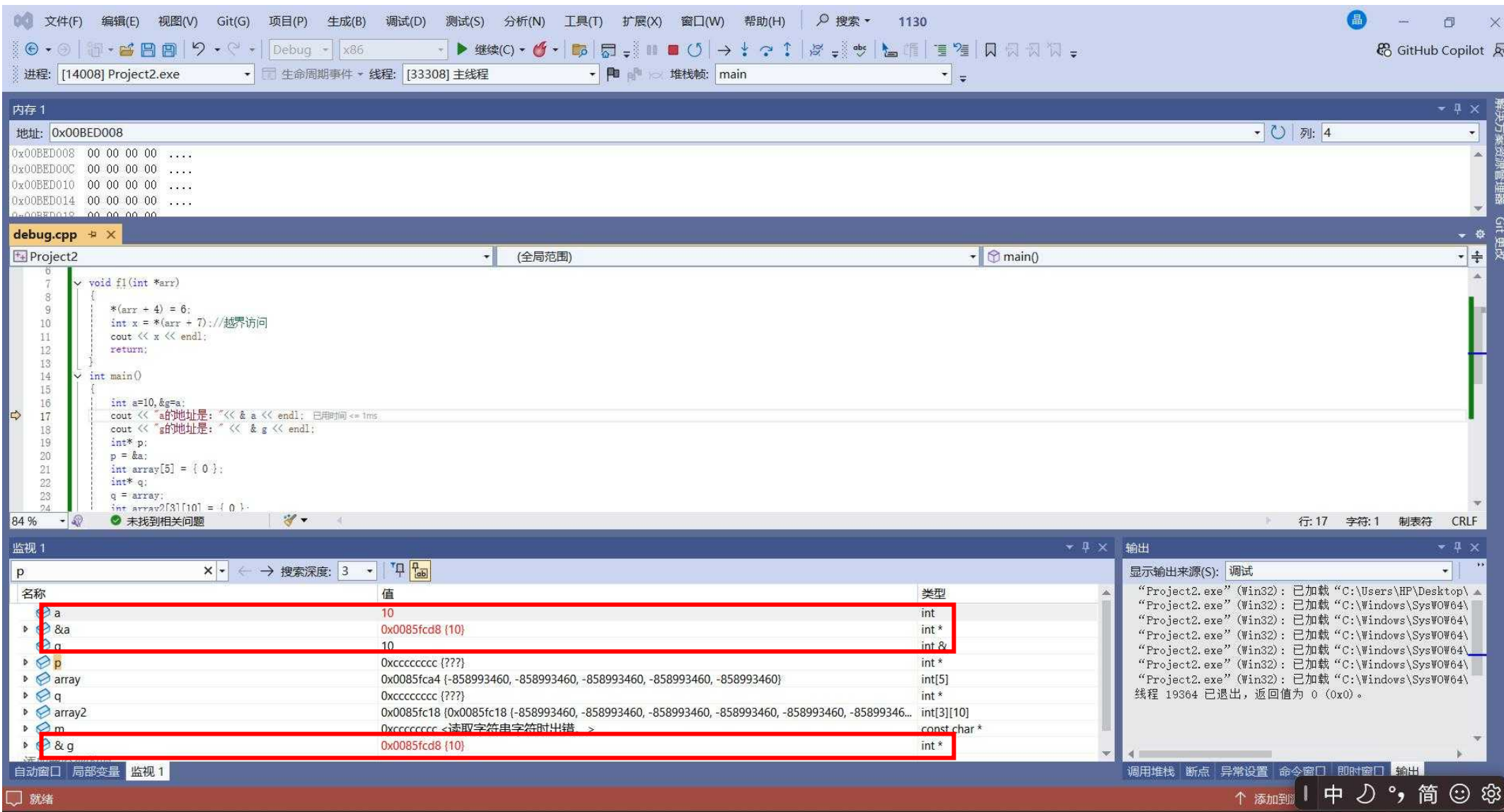
调用堆栈 断点 异常设置 命令窗口 即时窗口 输出

就绪

添加到 中 简 设置



第2步：按F10，逐步调试至简单变量定义、赋值语句后。即可在监视区看到相关的值和地址。（如何给变量添加监视已在前文提到过，在此不再赘述）



本图涉及知识点 3.1 (涵盖了double等一系列简单变量的查看方法)

### 第2步补充说明：

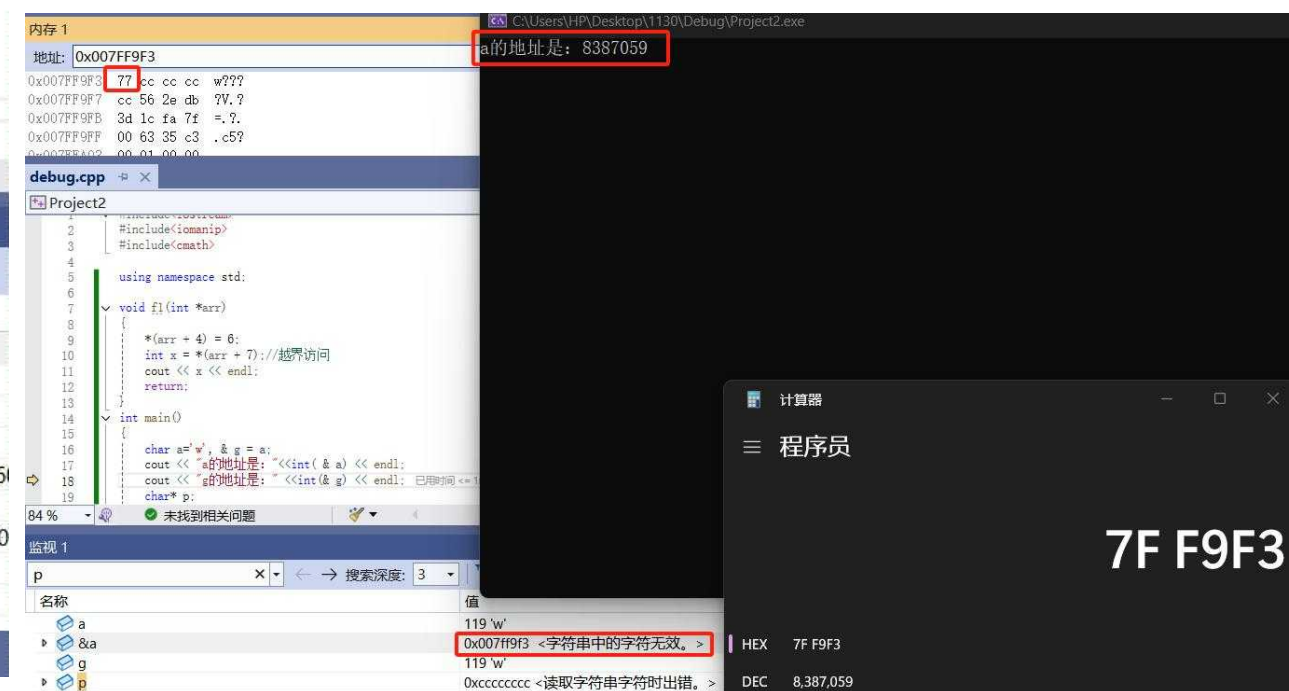
上图中还涉及了简单变量的引用，即知识点3.8

对char型简单变量作出特殊说明（其余简单变量的处理就按照上图代码处理即可）：

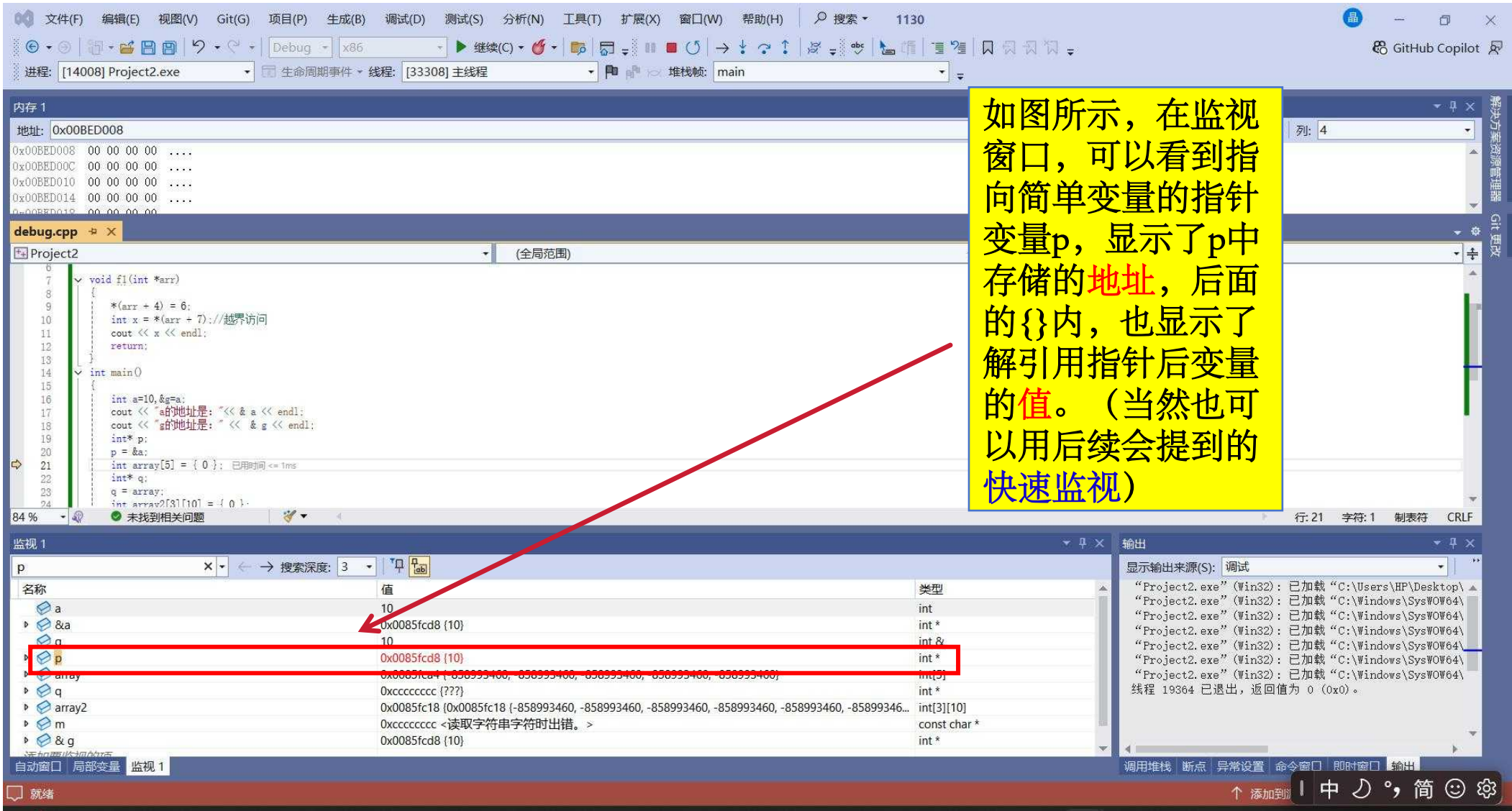
在尝试过程中，char型简单变量不能直接cout<<&a输出char型变量的地址。因为char型变量取地址就类似字符串指针的作用了。打印输出的时候会按照字符串方式输出。在监视界面和输出界面都会出错。

(错误情况展示:)

(解决方案: 转int型输出地址即可 'w' ASCII码为77H)



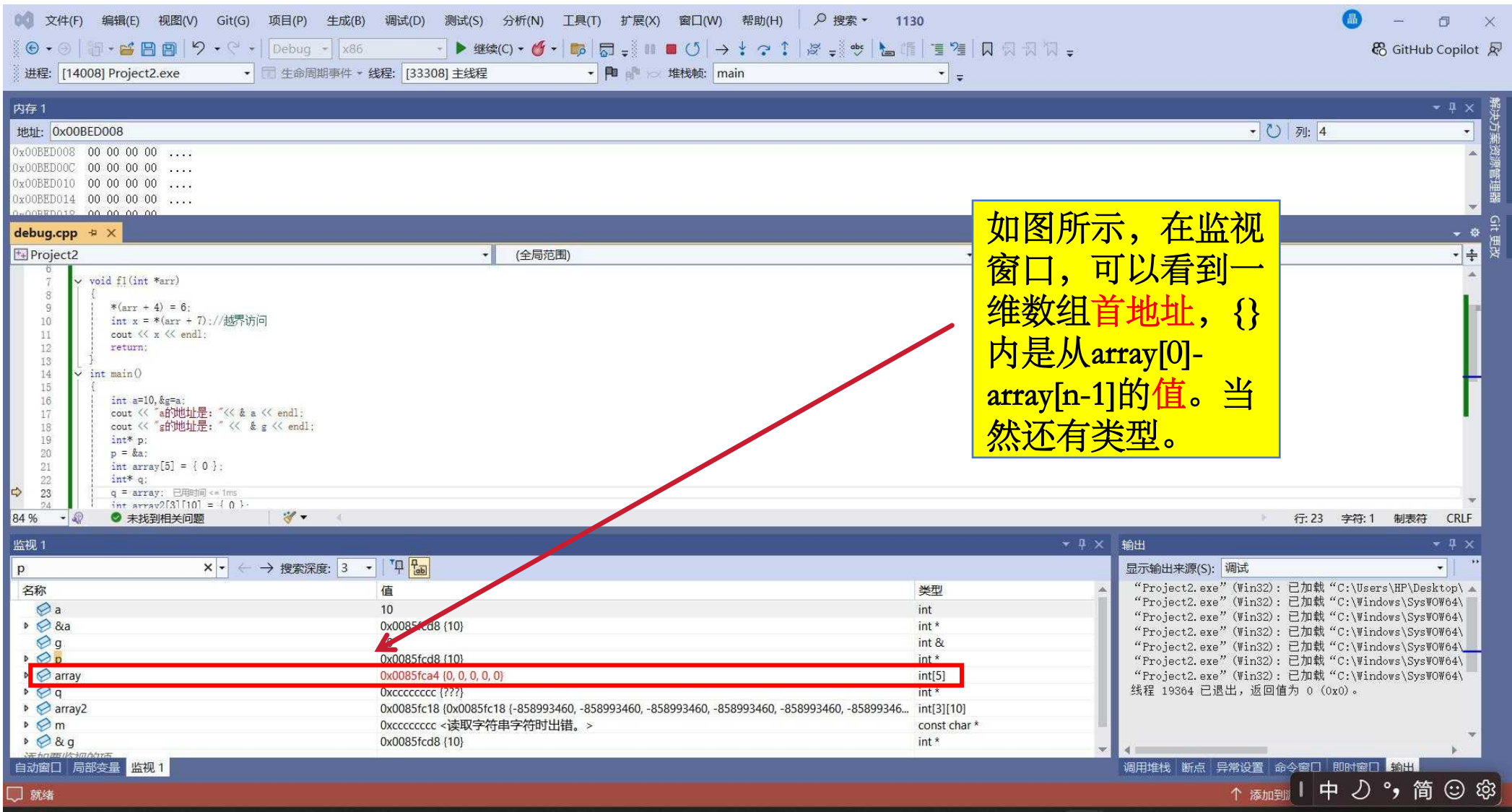
第3步：按F10,逐步调试至指向简单变量的指针变量定义、赋值语句后。即可在监视区看到相关的值和地址。



本图涉及知识点 3.2



第4步：按F10,逐步调试至一维数组定义、赋值语句后。即可在监视区看到相关的值和地址。



如图所示，在监视窗口，可以看到一维数组首地址，{}内是从array[0]-array[n-1]的值。当然还有类型。

本图涉及知识点 3.3

第5步：按F10,逐步调试至指向一维数组的指针变量定义、赋值语句后。即可在监视区看到相关的值和地址。

内存 1

地址: 0x00BED008

0x00BED008 00 00 00 00 ....

0x00BED00C 00 00 00 00 ....

0x00BED010 00 00 00 00 ....

0x00BED014 00 00 00 00 ....

0x00BED018 00 00 00 00 ....

debug.cpp

```
7 void f1(int *arr)
8 {
9     *(arr + 4) = 6;
10    int x = *(arr + 7); //越界访问
11    cout << x << endl;
12    return;
13 }
14 int main()
15 {
16     int a=10,&g=a;
17     cout << "a的地址是: " << &a << endl;
18     cout << "g的地址是: " << &g << endl;
19     int* p;
20     p = &a;
21     int array[5] = { 0 };
22     int* q;
23     q = array;
24     int array2[3][10] = { 0 }; 已用时间 <= 1ms
```

监视 1

名称	值	类型
a	10	int
&a	0x0085fcd8 {10}	int *
g	10	int &
p	0x0085fcd8 {10}	int *
array	0x0085fca4 {0, 0, 0, 0, 0}	int[5]
q	0x0085fca4 {0}	int *
array2	0x0085fc18 {0x0085fc18, -858993460, -858993460, -858993460, -858993460, -858993460, -858993460, -858993460, -858993460, -858993460}	int[3][10]
m	0xc0000000 <读取字符串字符时出错。>	const char *
&g	0x0085fcd8 {10}	int *

如图所示，在监视窗口，可以看到指向一维数组的指针变量q，显示了array的首地址，后面的{ }内，也显示了解引用指针后变量的值，即array[0]。（当然也可以用后续会提到的快速监视去查看array数组其他位置的值和地址）

本图涉及知识点 3.4



第6步：按F10,逐步调试至二维数组定义、赋值语句后。即可在监视区看到相关的值和地址。

如图所示，在监视窗口，可以看到二维数组首地址 `array2[0][0]`。由于二维数组可以看成元素为一维数组的一维数组，所以 `{}` 内的地址是 `array2[0]-array2[1]` 的地址，后面跟的就是对应一维数组元素的值。

内存 1

地址	值
0x00BED008	00 00 00 00 ....
0x00BED00C	00 00 00 00 ....
0x00BED010	00 00 00 00 ....
0x00BED014	00 00 00 00 ....
0x00BED018	00 00 00 00 ....

debug.cpp

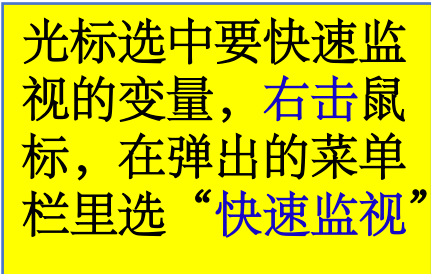
```
17 cout << "a的地址是: " << a << endl;
18 cout << "g的地址是: " << g << endl;
19 int* p;
20 p = &a;
21 int array[5] = { 0 };
22 int* q;
23 q = array;
24 int array2[3][10] = { 0 };
25
26 fl(array); 已用时间 <= 1ms
27
28 const char* m;
29 m = "Hello";
30 return 0;
31 }
```

监视 1

名称	值	类型
a	10	int
&a	0x0085fcd8 (10)	int *
g	10	int &
p	0x0085fcd8 (10)	int *
array	0x0085fca4 {0, 0, 0, 0, 0}	int [5]
q	0x0085fca4 (0)	int *
array2	0x0085fc18 {0x0085fc18 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, 0x0085fc40 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, 0x0085fc68 {...}}	int [3][10]
m	0x00000000 <读取字符串字符时出错。>	const char *
&g	0x0085fcd8 (10)	int *

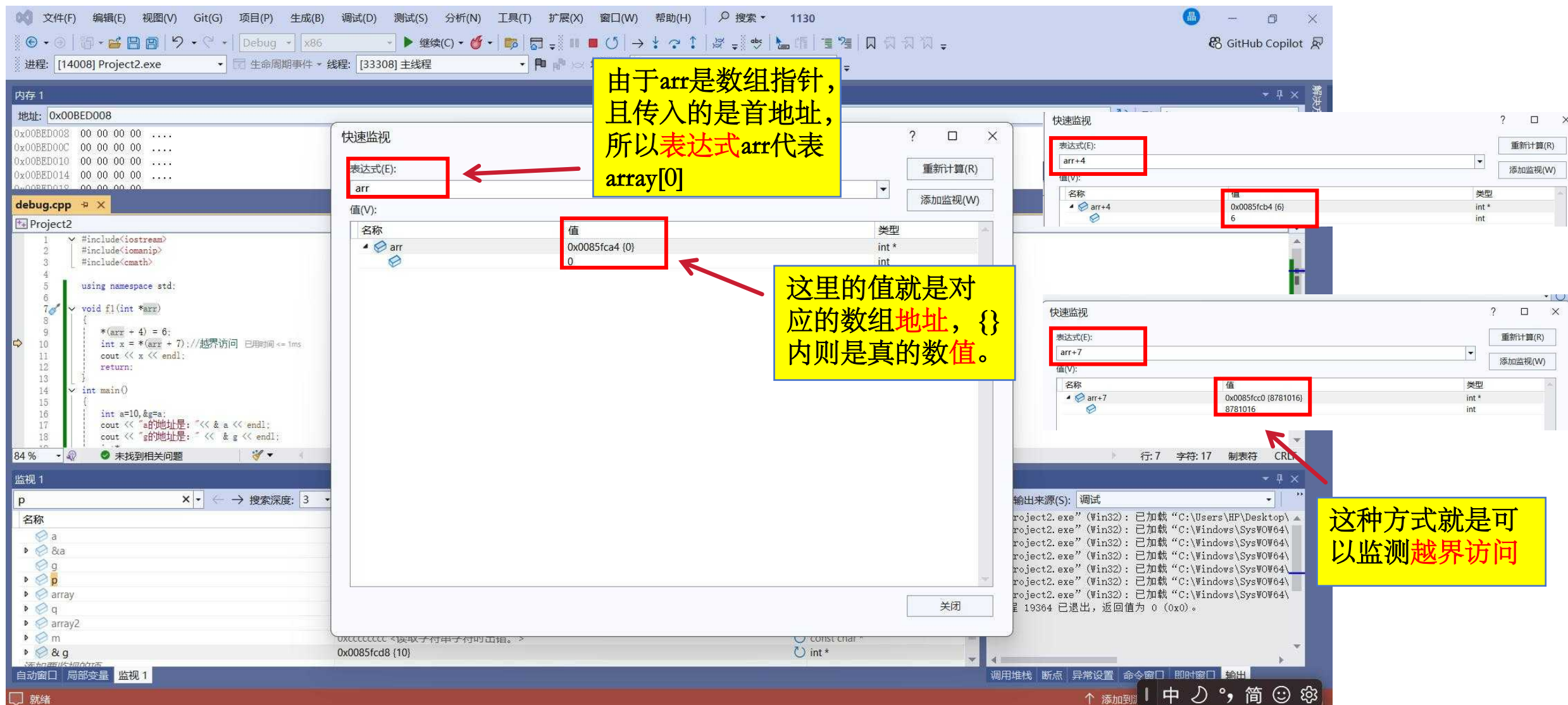
本图涉及知识点 3.5

## 7-1: 添加快速监视



第7步：按F11,进入实参是一维数组名，形参是指针的函数中。在函数中查看实参数组的地址、值。

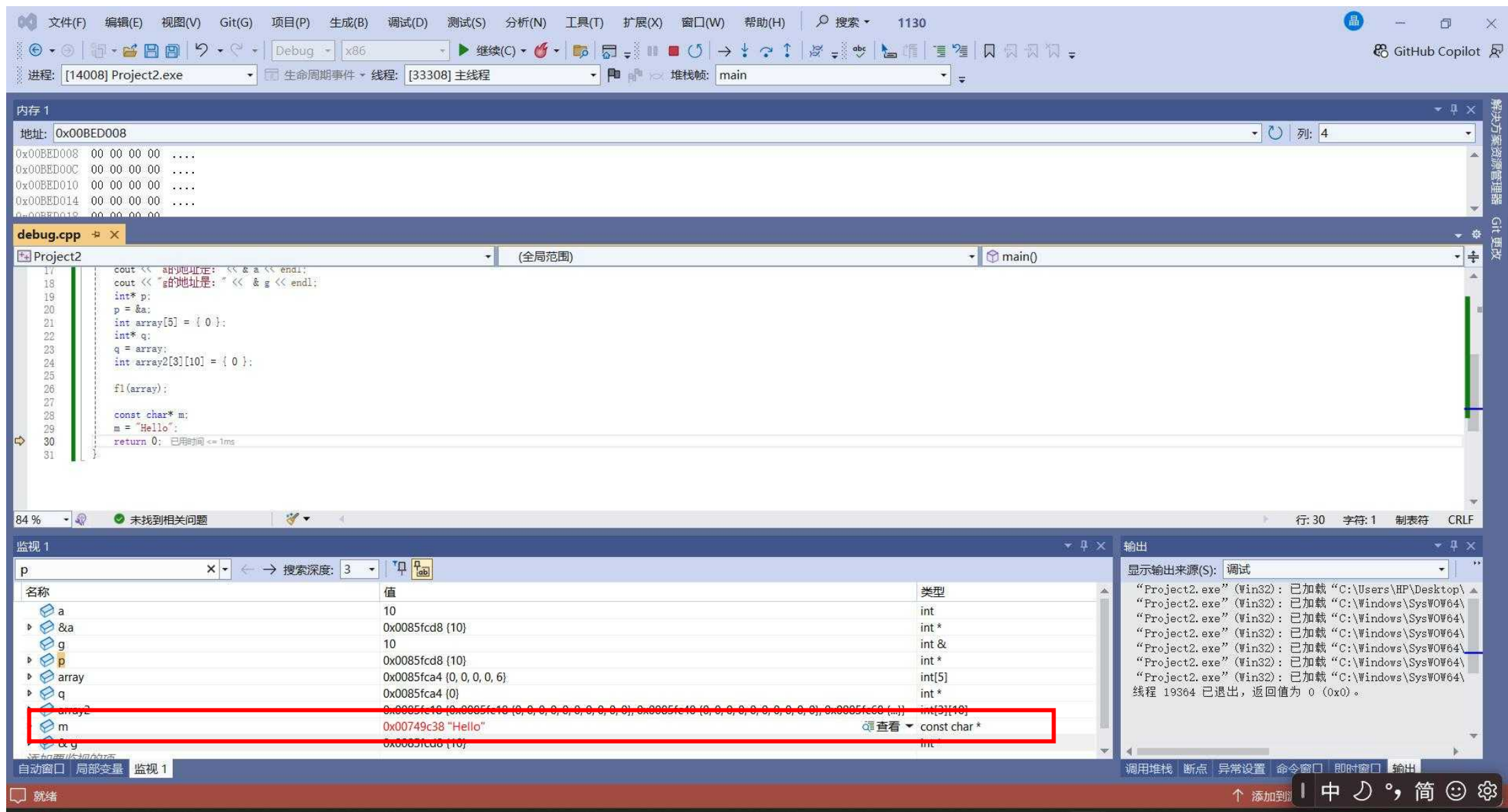
7-2: 在快速监视窗口查看需要查看的数组的地址、值。



本图涉及知识点 3.6/3.9



第8步：按F10,逐步调试至字符串常量的指针变量定义、赋值语句后。即可在监视区看到相关的值和地址。



本图涉及知识点 3.7 (可以看到无名字符串常量的地址, 同样应用快速监测即可)

- 补充说明----引用和指针的区别：

区别	引用	指针
定义和初始化	必须在声明时初始化，不能重新引用到另一个对象。	可以声明后初始化，也可以重新指向另一个对象。
使用	直接使用，不需要解引用操作符*	使用时需要加解引用操作符*
内存占用	不占用额外内存，只是变量的别名	需要占用内存
类型转换	无	有
可修改性	可以修改引用的变量	可以修改指针指向的对象，也可以修改指针指向的地址



# 测试程序

```
#include<iostream>
#include<iomanip>
#include<cmath>
```

```
using namespace std;
```

```
void f1(int *arr)
{
    *(arr + 4) = 6;
    int x = *(arr + 7);//越界访问
    cout << x << endl;
    return;
}
int main()
{
    int a=10, & g = a;
    cout << "a的地址是: " << int( & a) << endl;
    cout << "g的地址是: " << int(& g) << endl;
    int* p;
    p = &a;
    int array[5] = { 0 };
    int* q;
    q = array;
    int array2[3][10] = { 0 };

    f1(array);

    const char* m;
    m = "Hello";
    return 0;
}
```

