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）

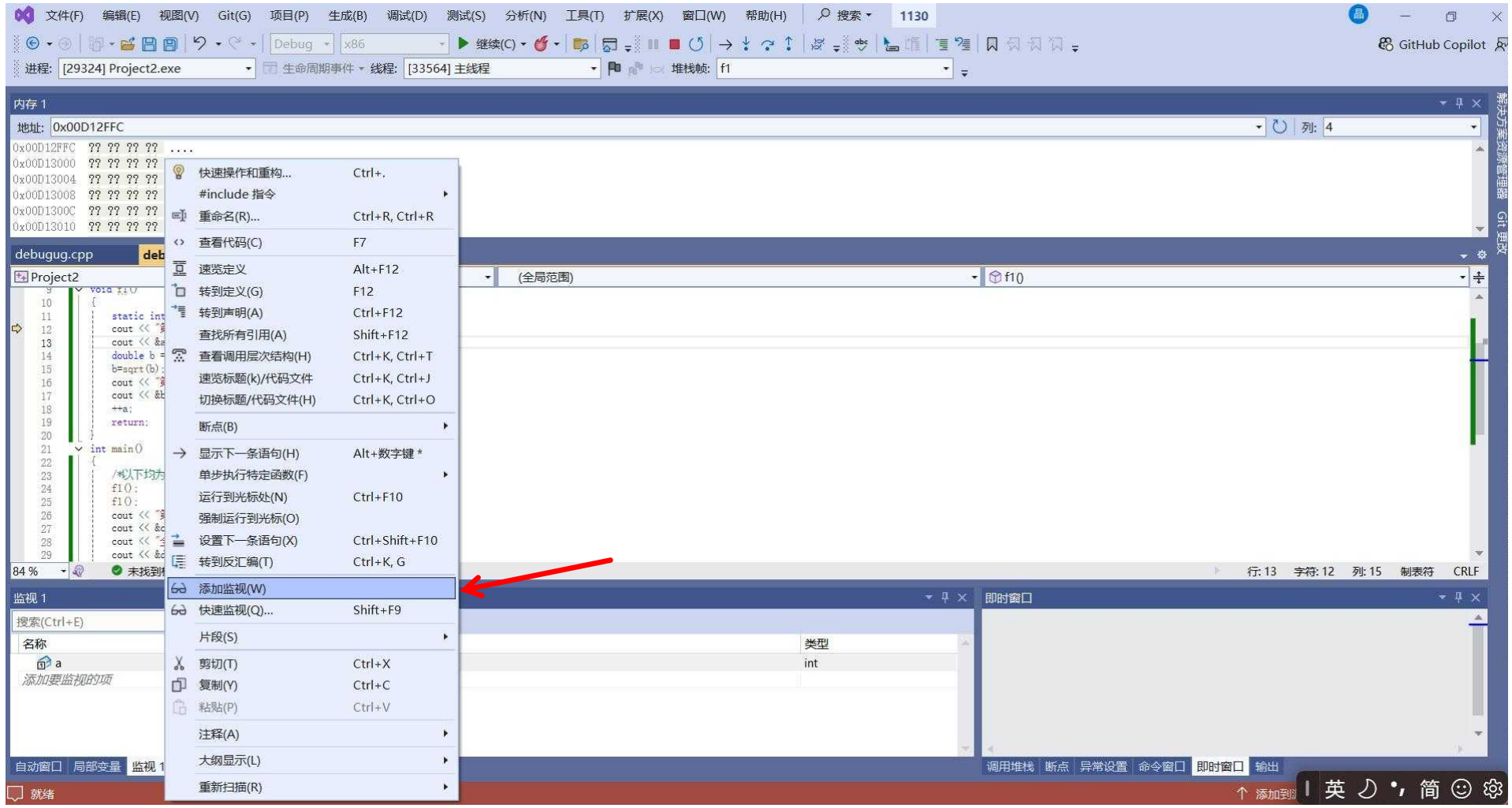
- 在碰到 cout/sgrt等系统类/系统函数和自定义函数的调用语句(例如在 main 中调用自定义的 fun 函数)时，按**F11**会进入函数内部单步执行；按**F10**可以直接不进入到函数的内部单步执行；如果已经进入，则按**Shift+F11**可以跳出并返回自己的函数。

（下面将结合具体的程序调试过程演示说明知识点一、二）

本图涉及知识点1.1

第1步：按F10进入main函数，开始调试。

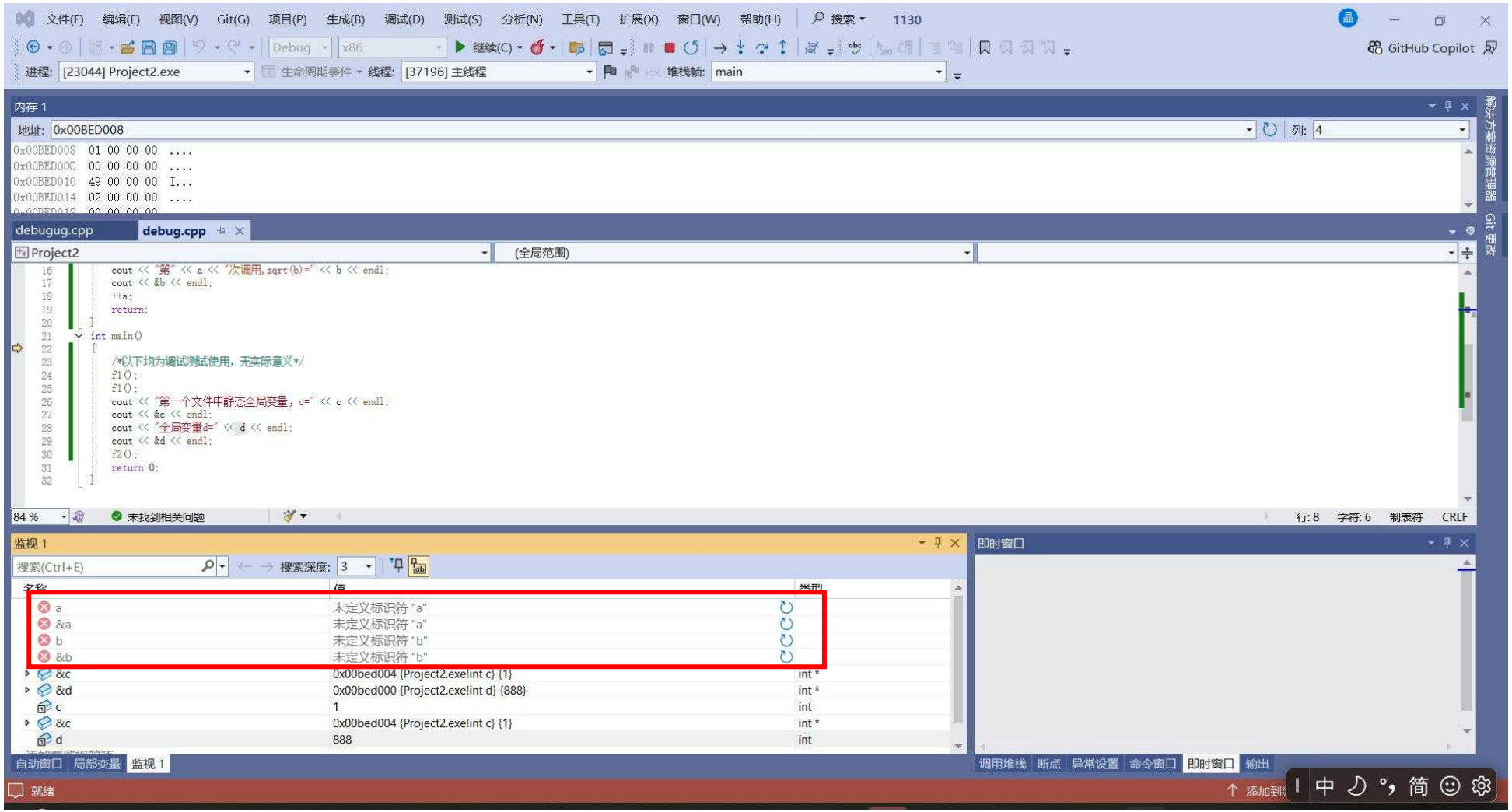
1-1：给所有变量添加监视，便于观察生存期和作用域的变化



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

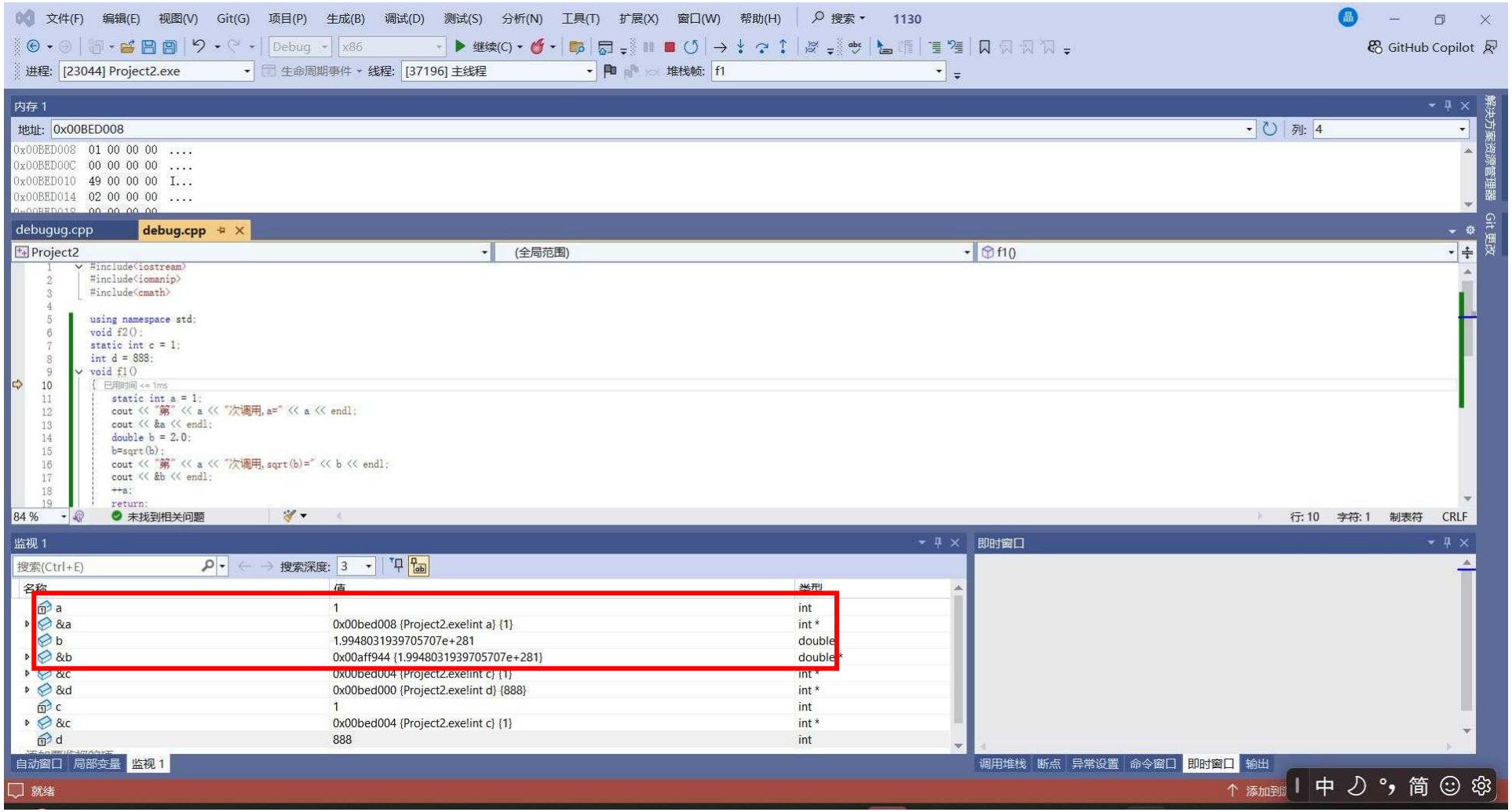


如图所示可以看到，一开始进入程序，f1（）函数未调用，故在f1（）函数中的局部自动变量呈未定义状态。而静态局部变量a、静态全局变量c、外部全局变量d生存期已开始。

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

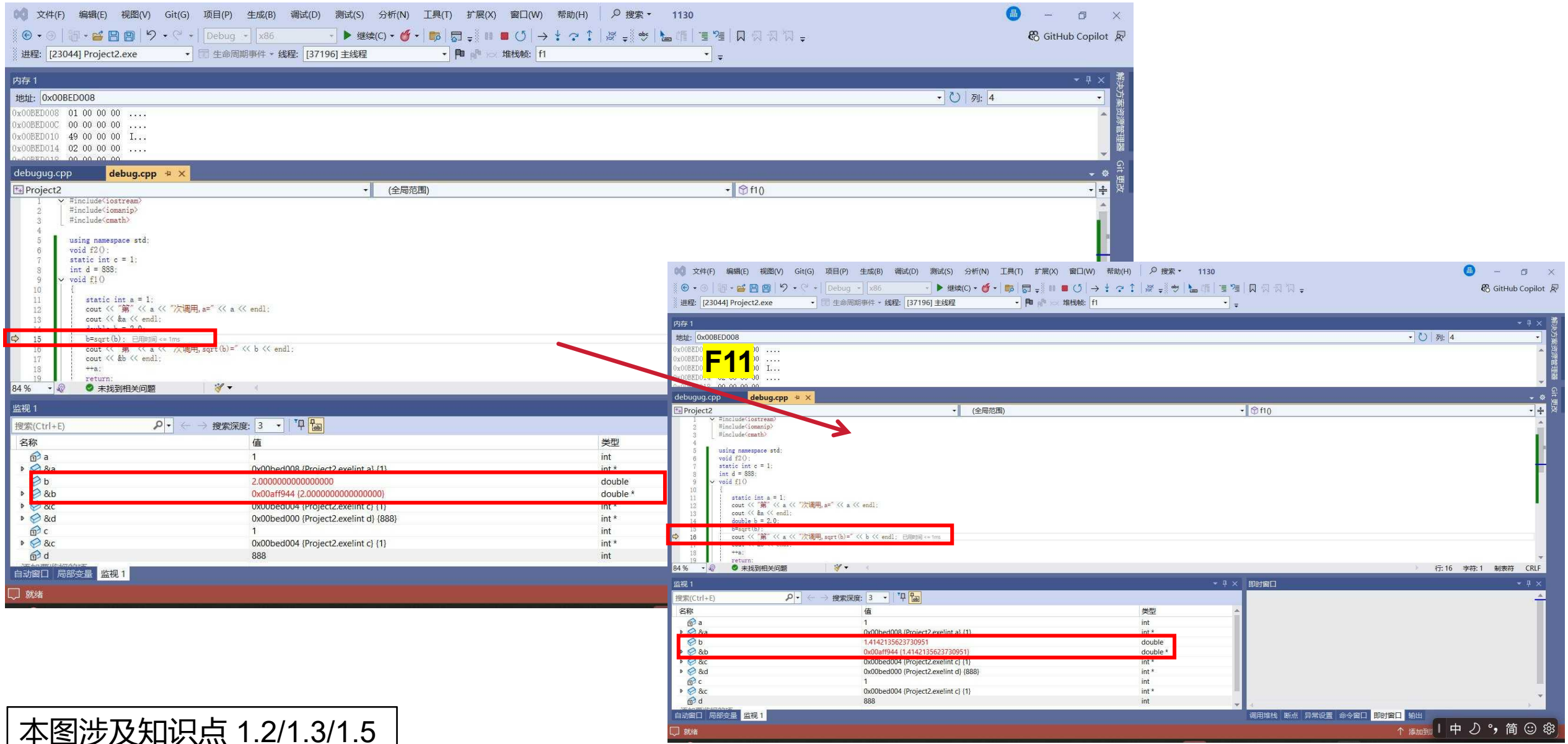
第2步：当箭头指向f1（）的调用语句时，按F11，进入函数。



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。

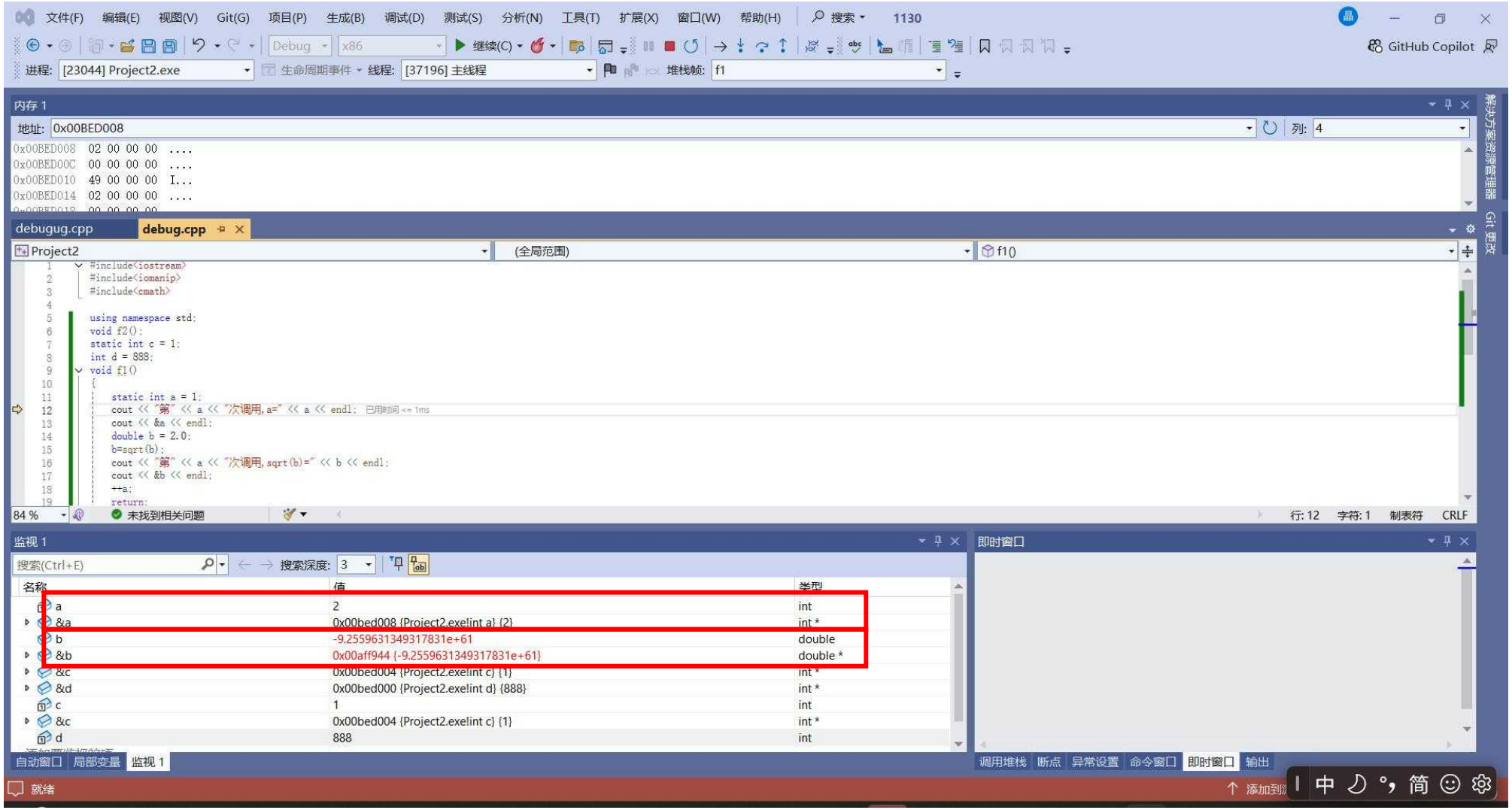


本图涉及知识点 1.2/1.3/1.5

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）

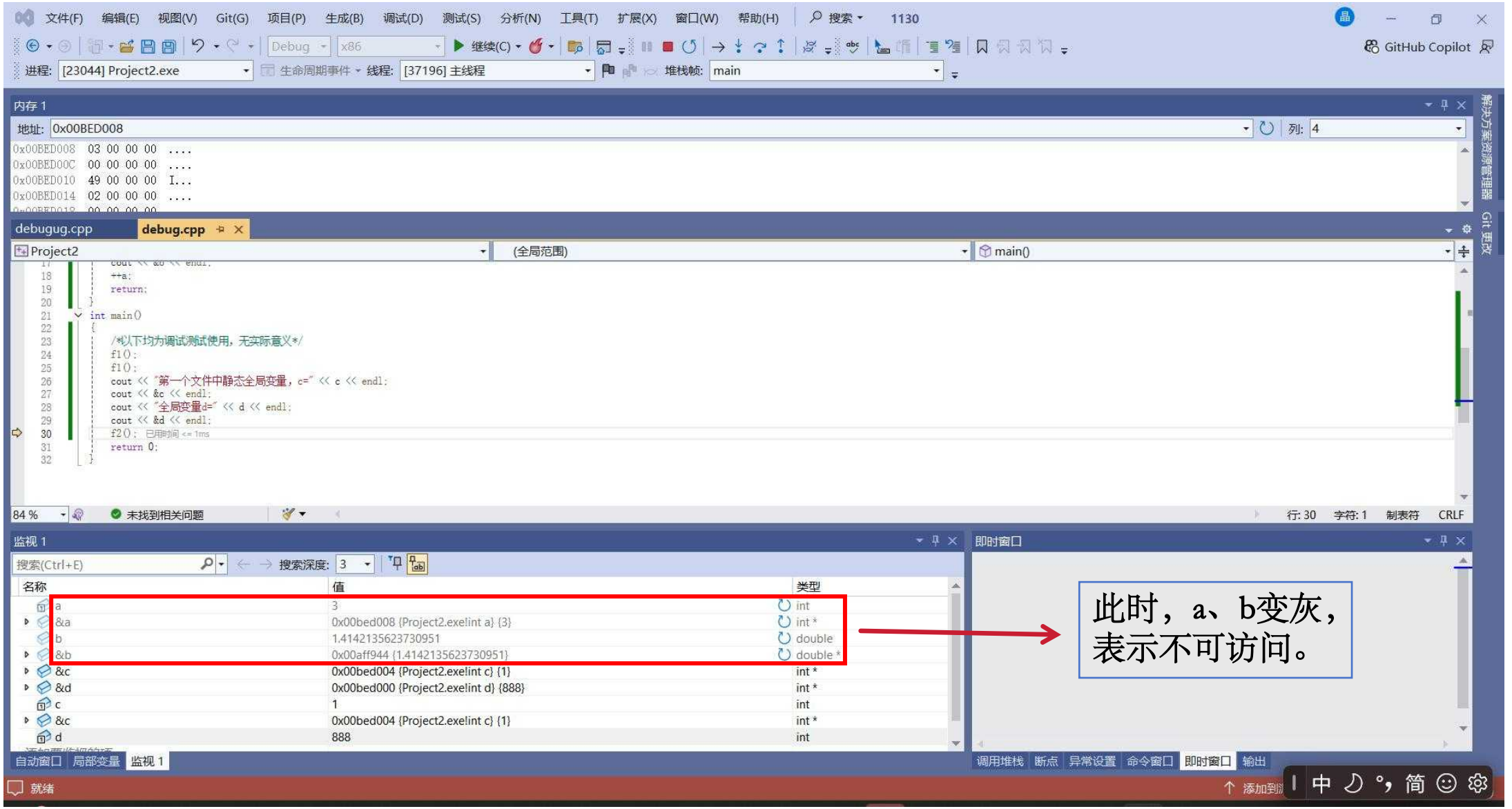
第5步：重复步骤2，再次进入f1（）函数。



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。

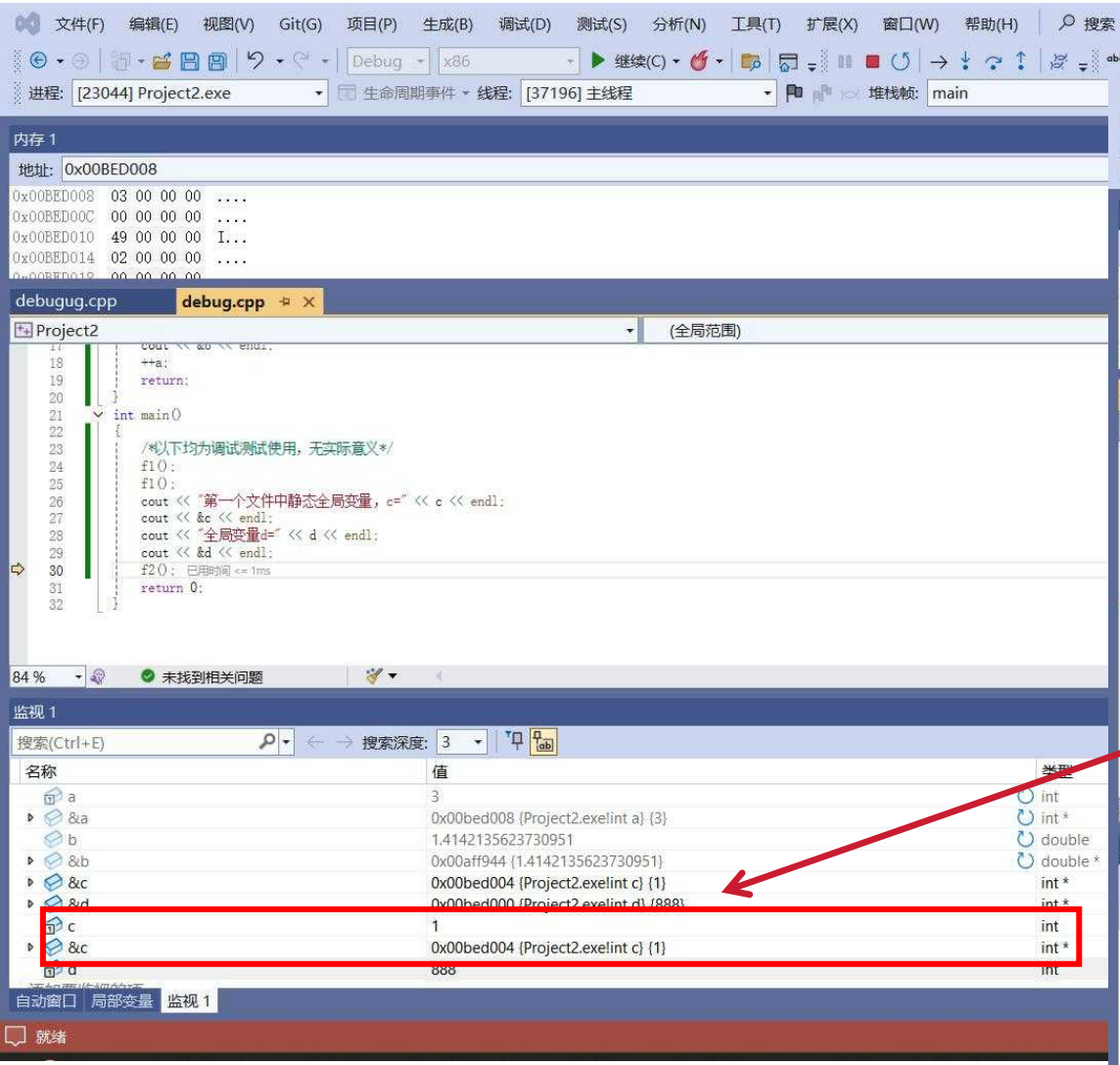
并且可以看到，此时b的赋值定义语句还未执行，b的值是随机的。可以推断得出，局部自动变量的存储空间当函数结束后会回收，下一次调用时候再重新分配，重新赋值。而静态局部变量不会。

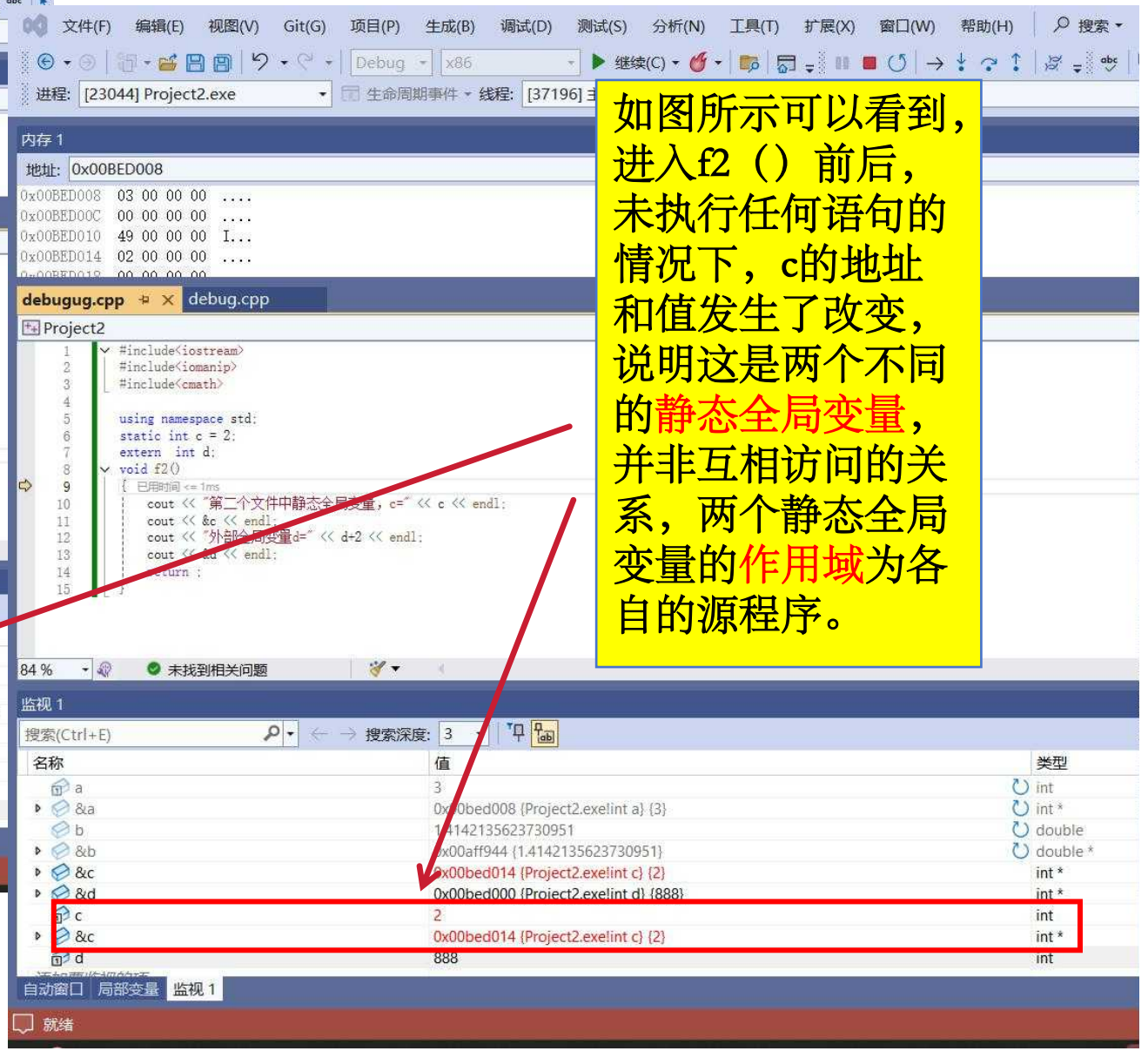
第6步：重复步骤4，退出f1（）函数，回到main函数。再按F10
单步执行到f2（）函数的调用语句。



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）

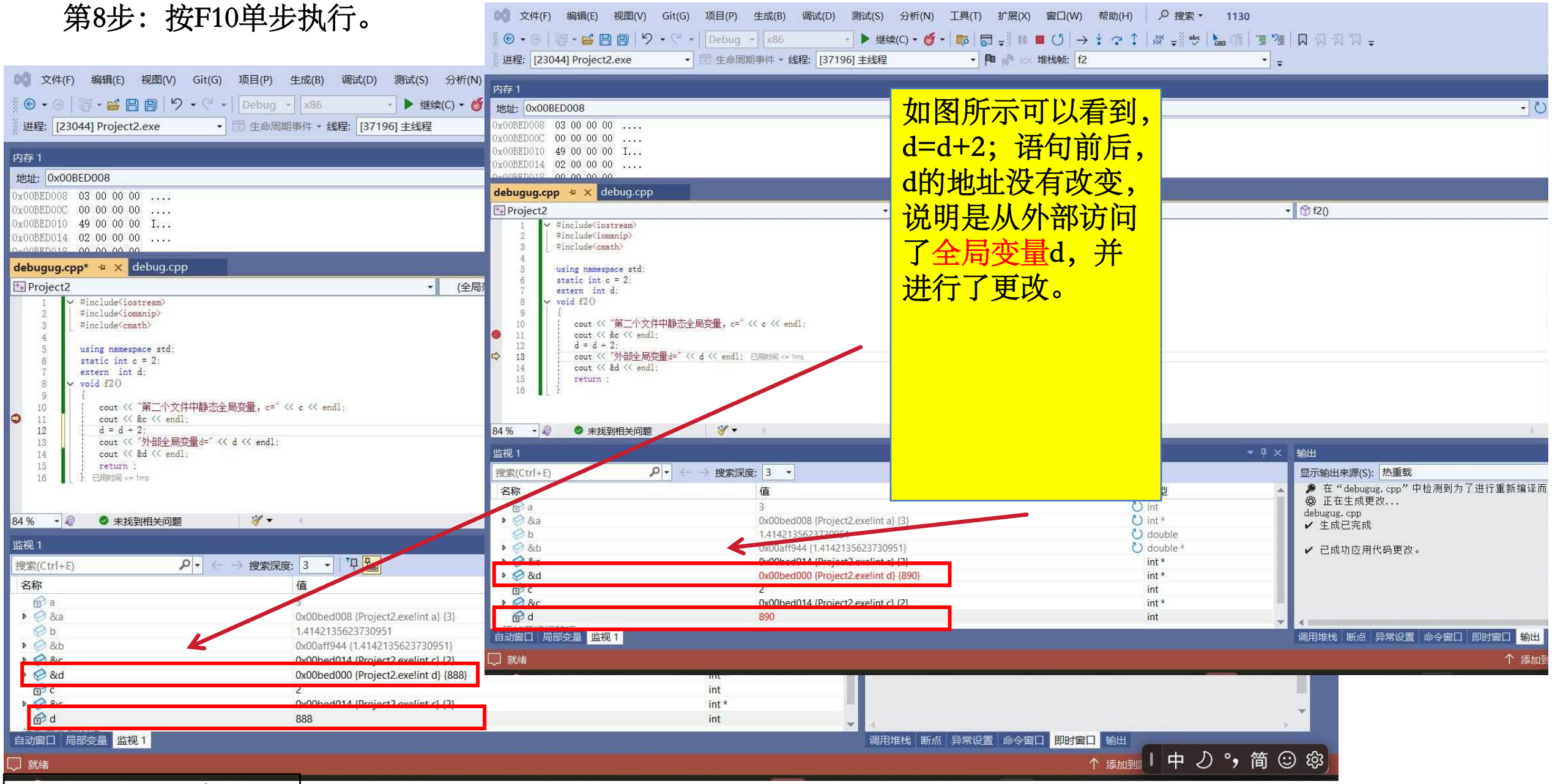




如图所示可以看到，进入f2（）前后，未执行任何语句的情况下，c的地址和值发生了改变，说明这是两个不同的静态全局变量，并非互相访问的关系，两个静态全局变量的作用域为各自的源程序。

本图涉及知识点 2.3

第8步：按F10单步执行。



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））

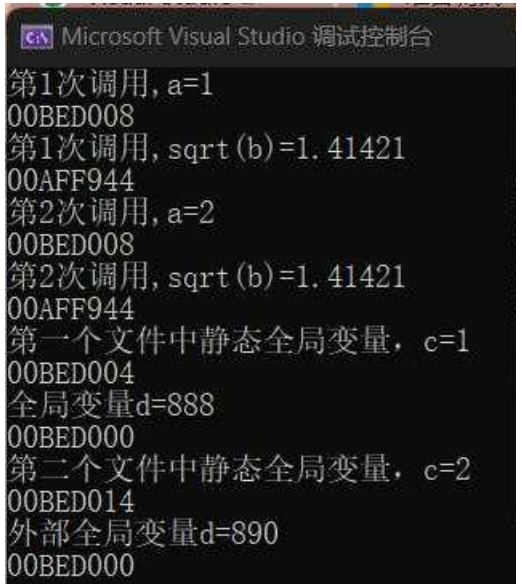
- 变量的生存期、作用域、存储区

	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

测试程序及运行结果

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;
}
```

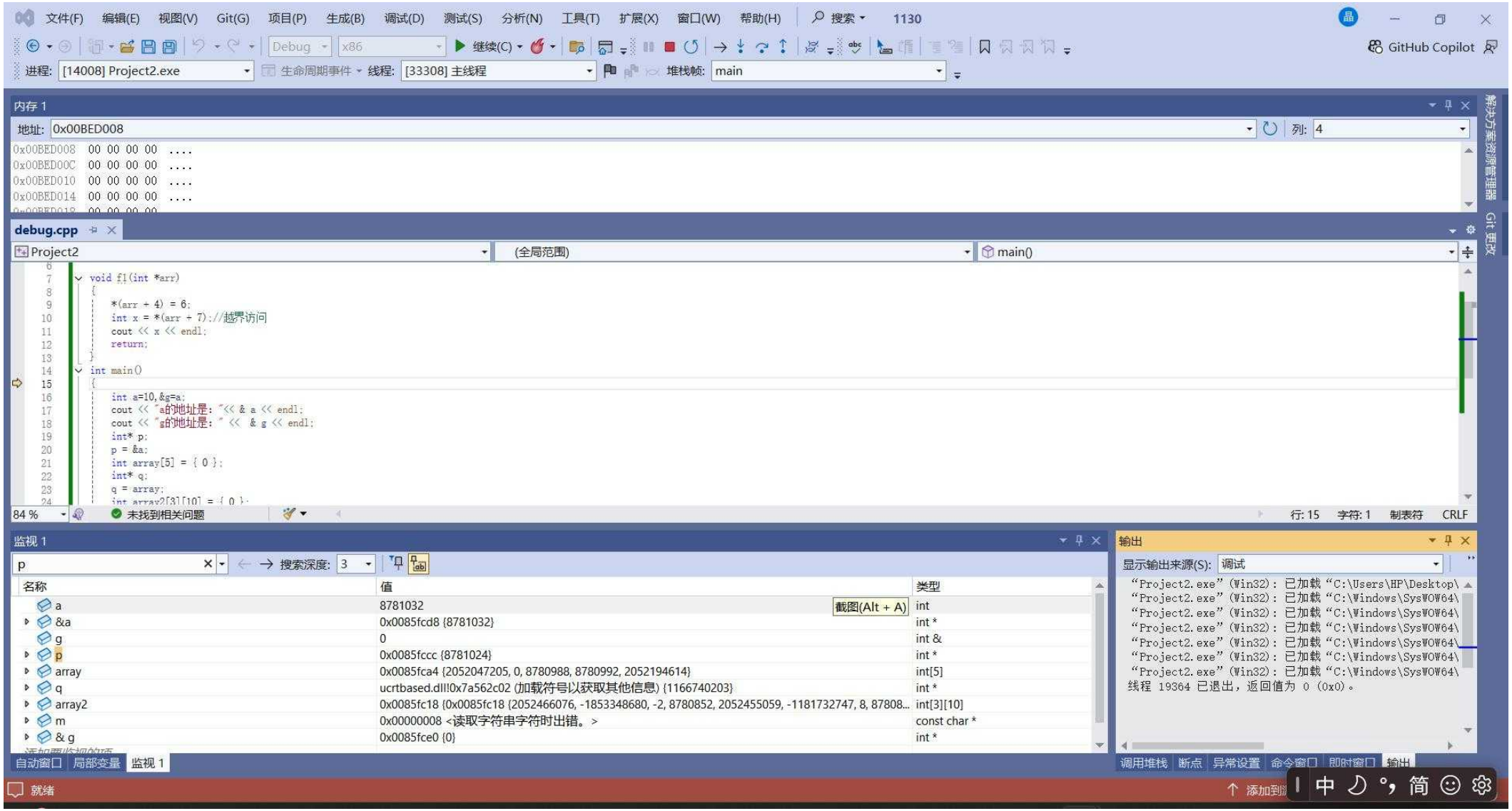


debugg.cpp

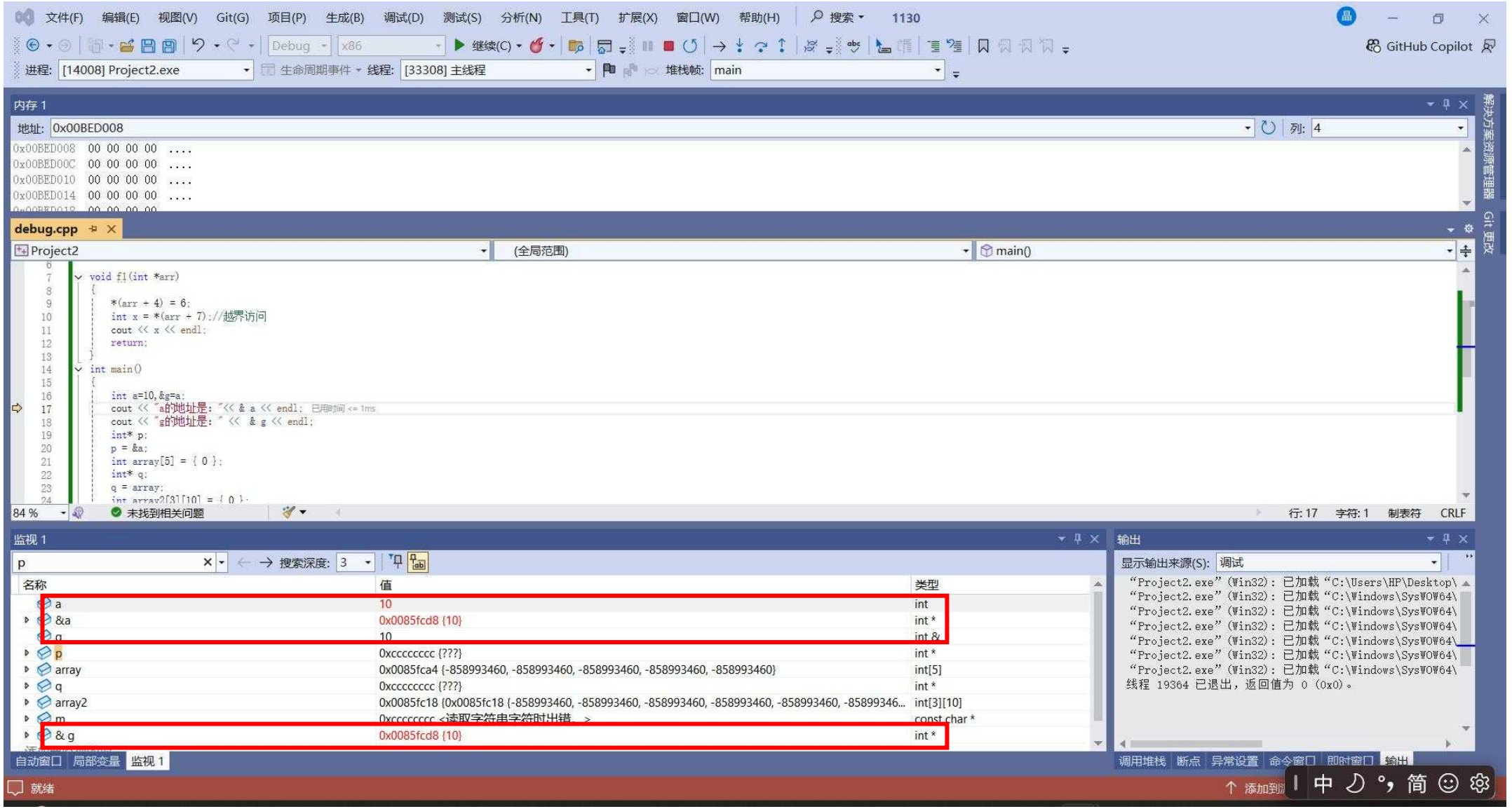
```
#include<iostream>
#include<iomanip>
#include<cmath>

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）



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）

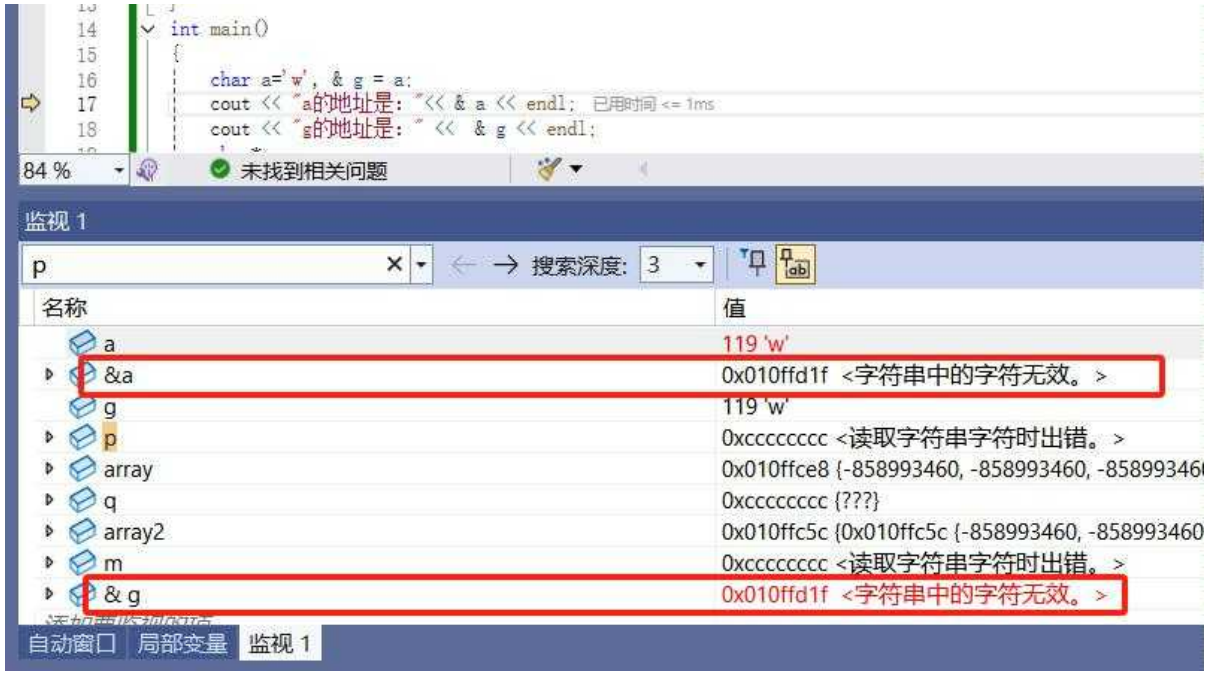
第2步补充说明：

上图中还涉及了简单变量的引用，即知识点3.8

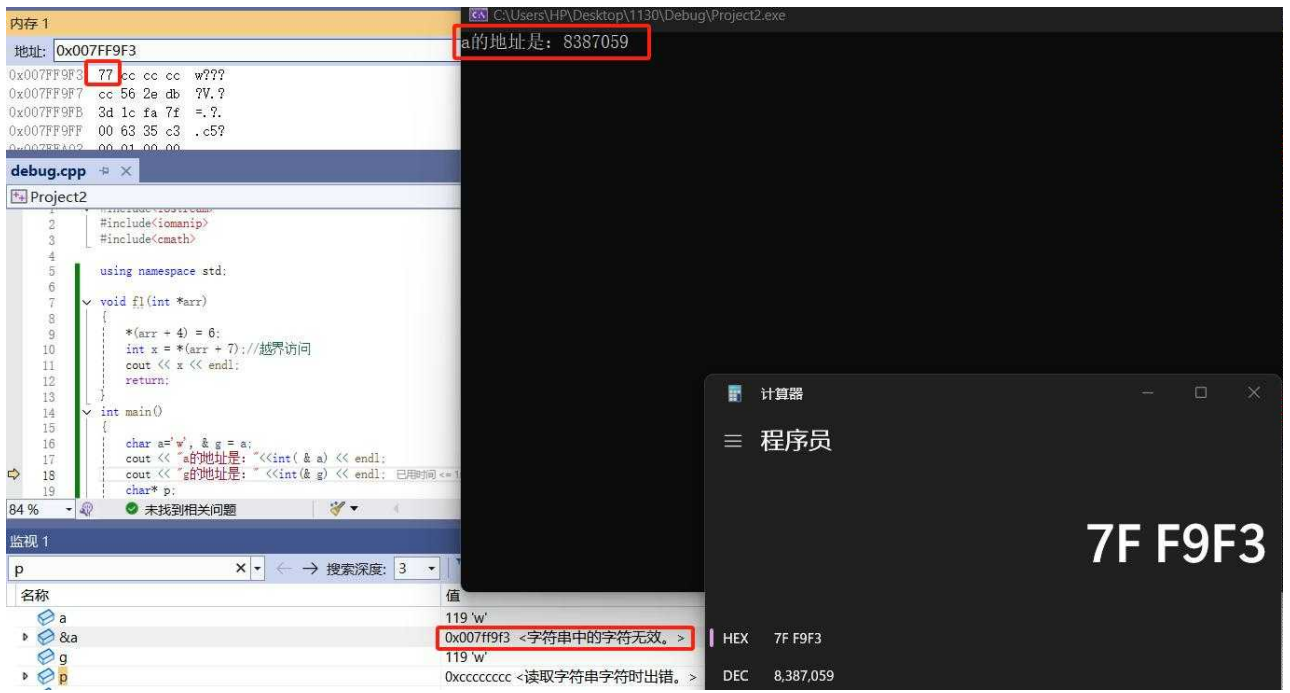
对char型简单变量作出特殊说明（其余简单变量的处理就按照上图代码处理即可）：

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。

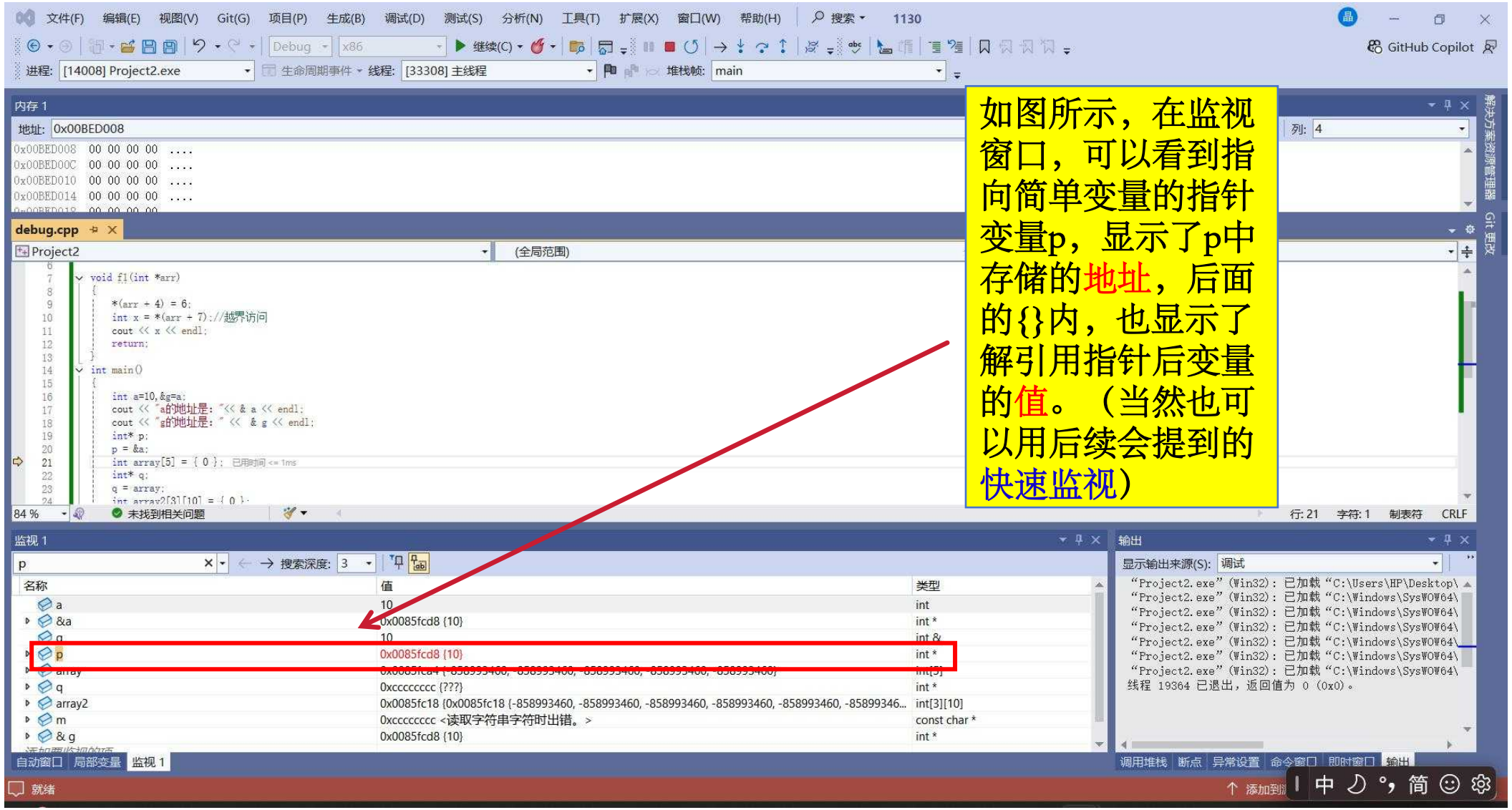
（错误情况展示：）



（解决方案：转int型输出地址即可‘w’ASCII码为77H）

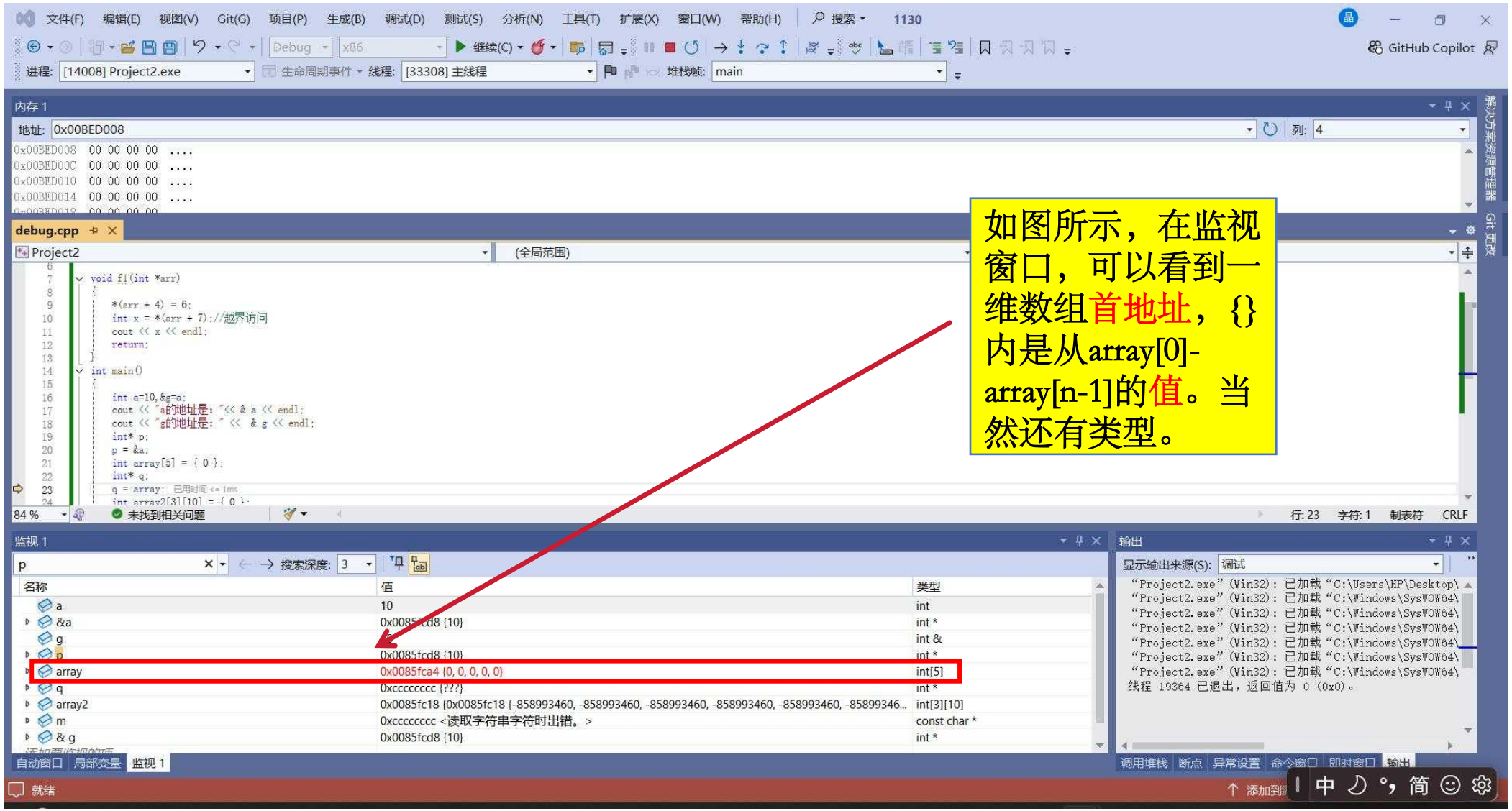


第3步：按F10,逐步调试至指向简单变量的指针变量定义、赋值语句后。即可在监视区看到相关的值和地址。



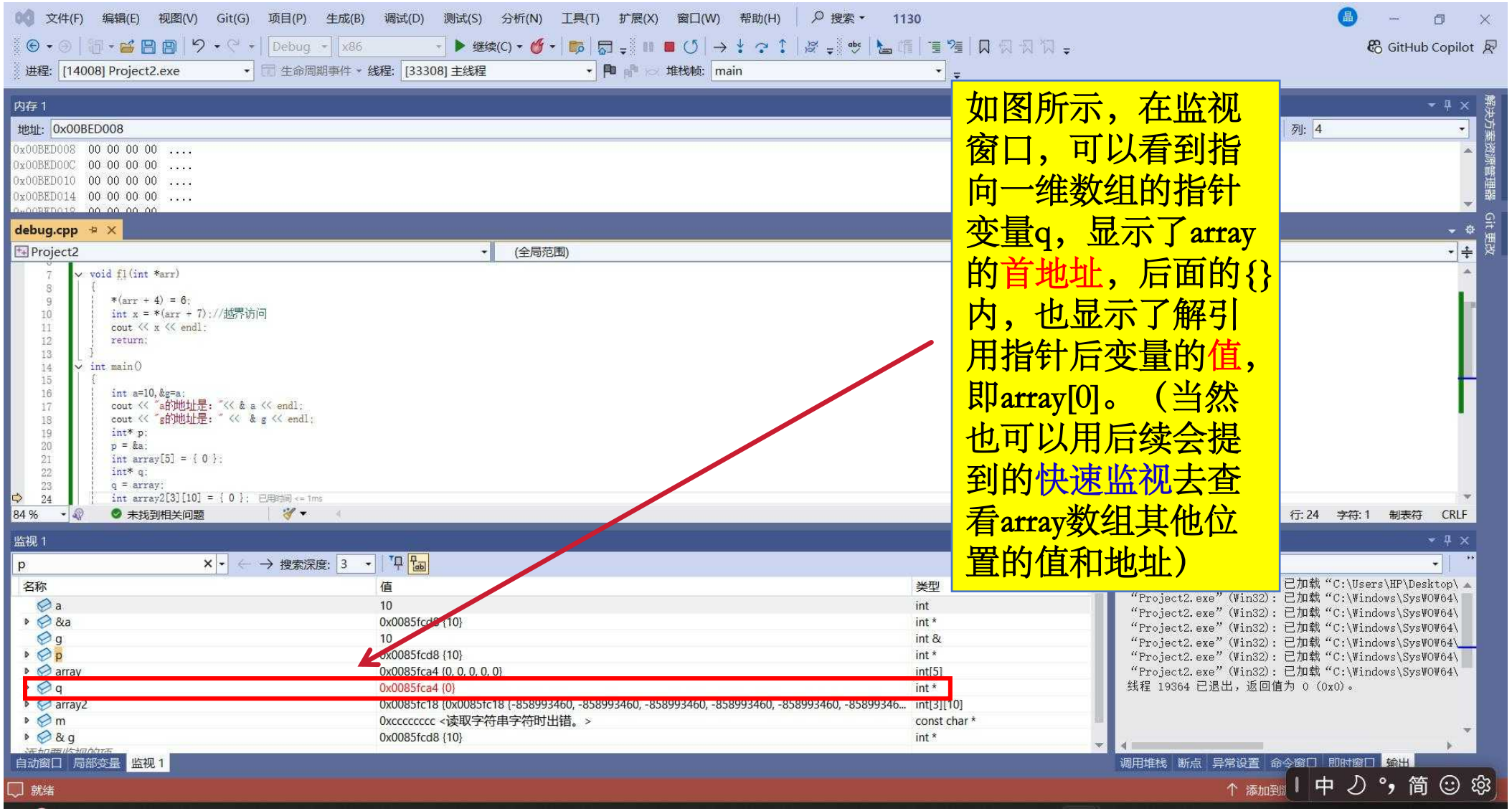
本图涉及知识点 3.2

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。



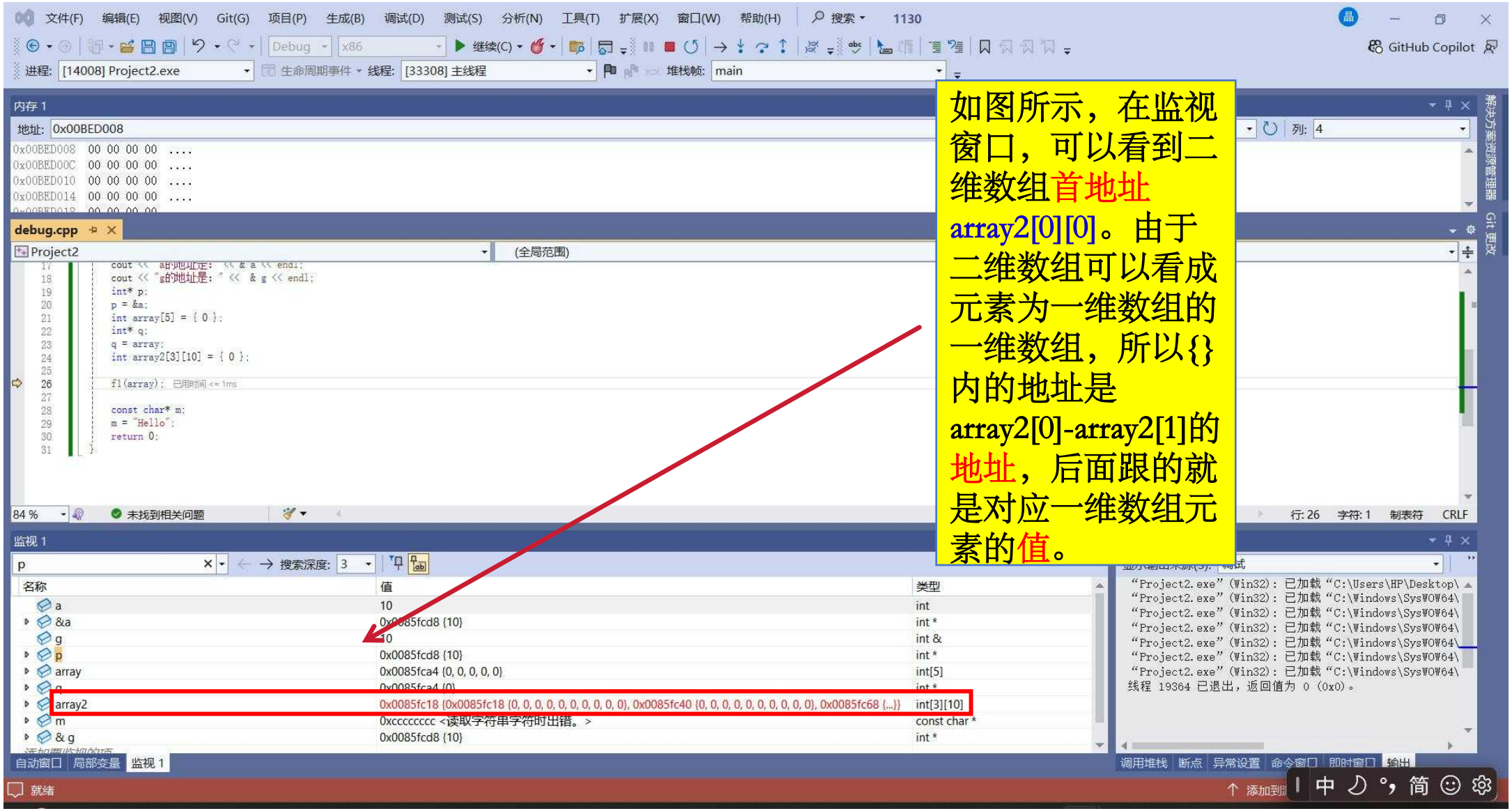
本图涉及知识点 3.3

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。



本图涉及知识点 3.4

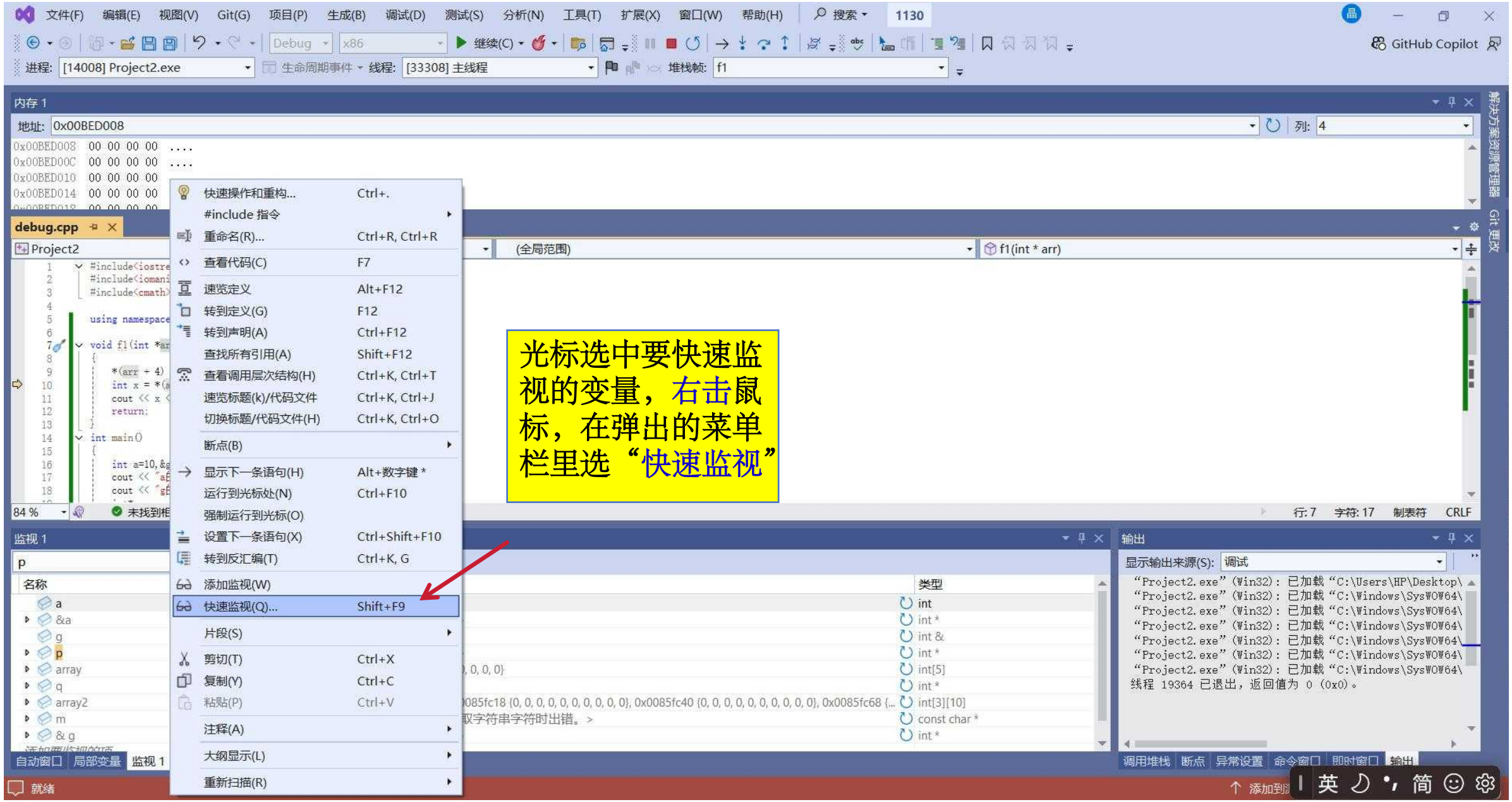
第6步：按F10,逐步调试至二维数组定义、赋值语句后。即可在监视区看到相关的值和地址。



本图涉及知识点 3.5

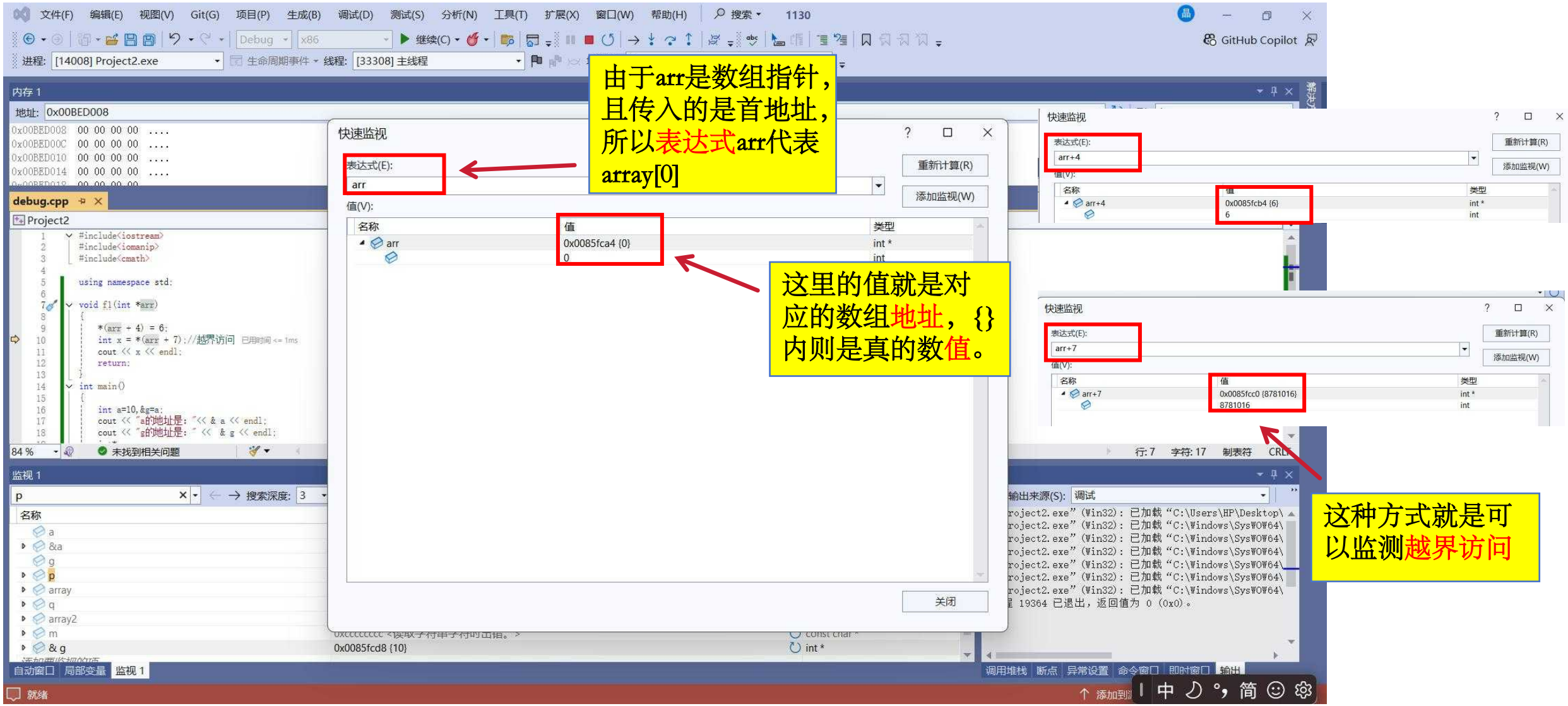
第7步：按F11,进入实参是一维数组名，形参是指针的函数中。在函数中查看实参数组的地址、值。

7-1：添加快速监视



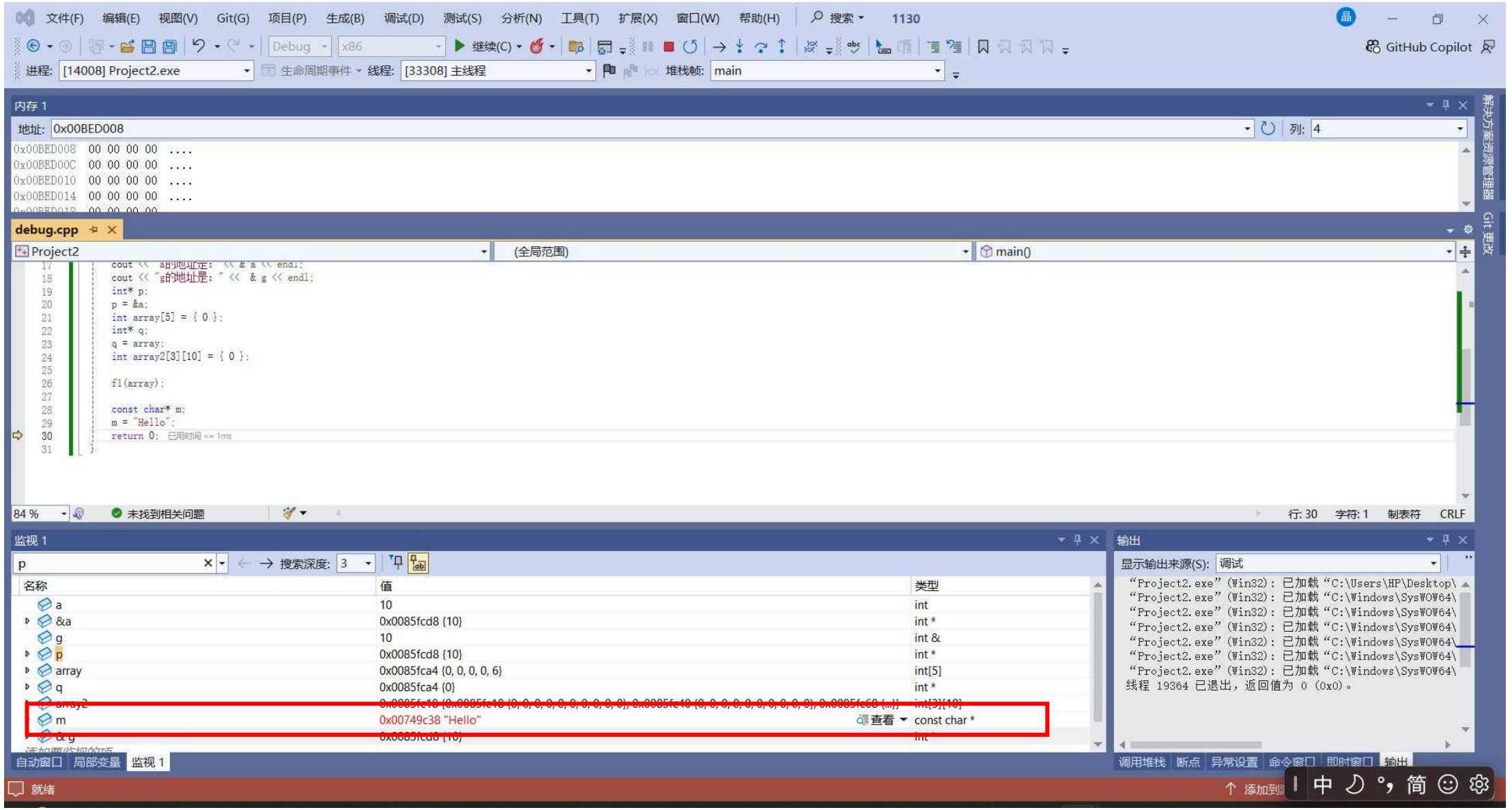
第7步：按F11,进入实参是一维数组名，形参是指针的函数中。在函数中查看实参数组的地址、值。

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;
}
```