
Visual Studio 2022 调试工具的基本使用方法实验报告

班级：软件工程

作者姓名：陈君

学号：2250420

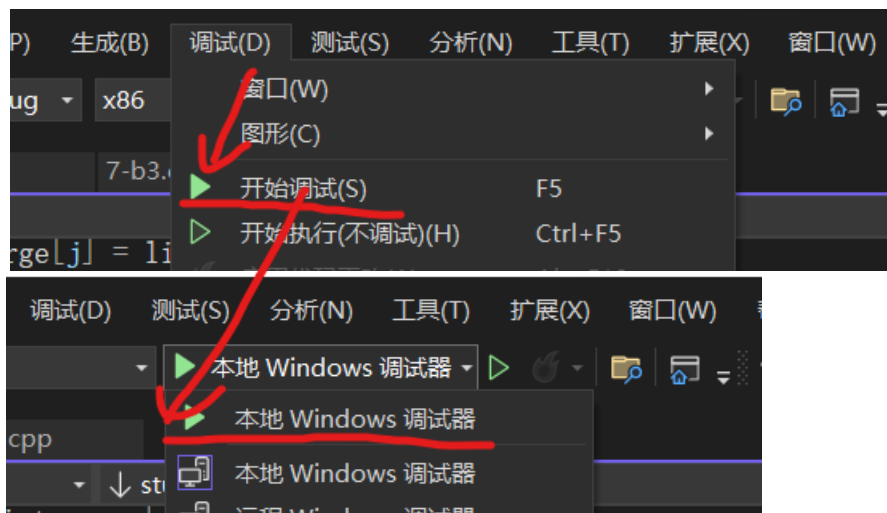
日期：1.3

1. VS2022下调试工具的基本使用方法

1.1. 如何开始调试和结束调试

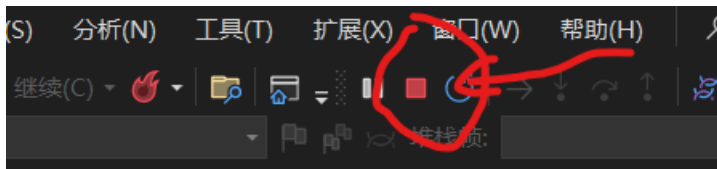
开始调试

可以按F5，或者在调试菜单中点击开始调试选项，或者点击本地windows调试器。



结束调试

点击右上角终止符号。



1.2. 在一个函数中每个语句单步执行

1、设置断点

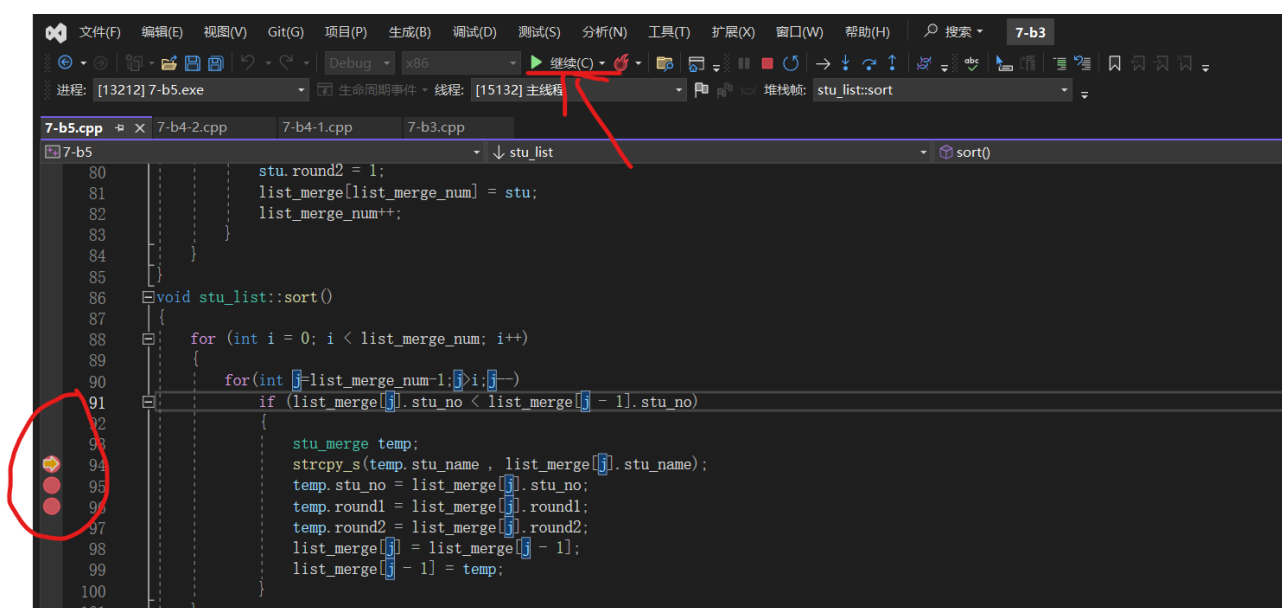
```

84  }
85  }
86  void stu_list::sort()
87  {
88      for (int i = 0; i < list_merge_num; i++)
89      {
90          for(int j=list_merge_num-1;j>i;j--)
91              if (list_merge[j].stu_no < list_merge[j - 1].stu_no)
92              {
93                  stu_merge temp;
94                  strcpy_s(temp.stu_name , list_merge[j].stu_name);
95                  temp.stu_no = list_merge[j].stu_no;
96                  temp.round1 = list_merge[j].round1;
97                  temp.round2 = list_merge[j].round2;
98                  list_merge[j] = list_merge[j - 1];
99                  list_merge[j - 1] = temp;
100              }
101      }

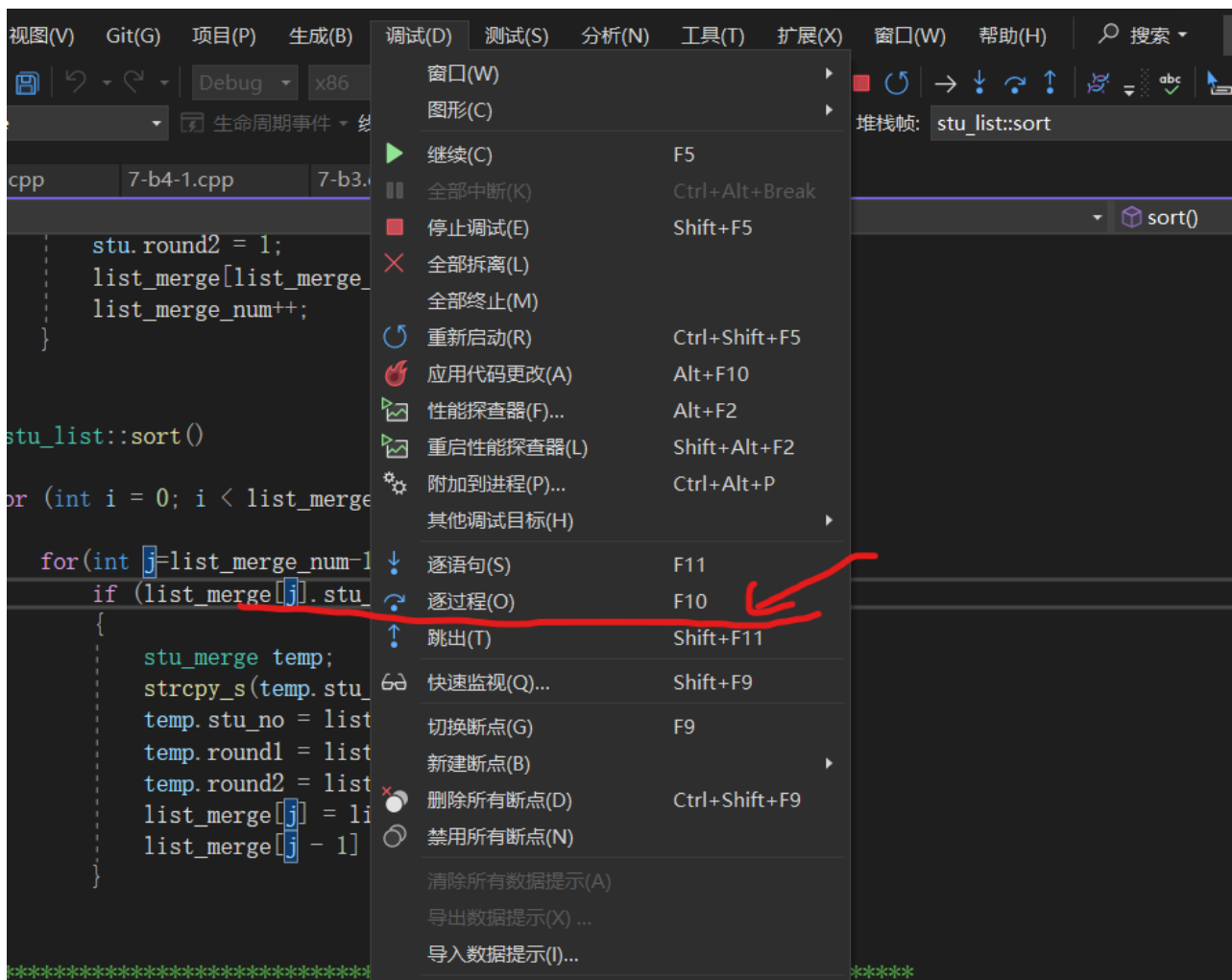
```

2、开始调试，方法与1.1相同。

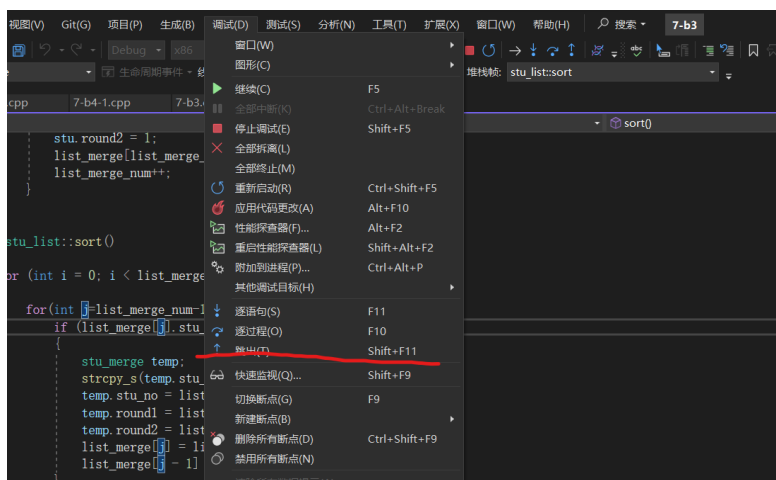
3、当程序执行到设置的断点处时，调试器会暂停执行并提示该行代码。可以使用调试工具栏上的继续按钮来控制执行过程。



4、如果需要逐过程执行代码，则按F10，或者调试中的逐过程。

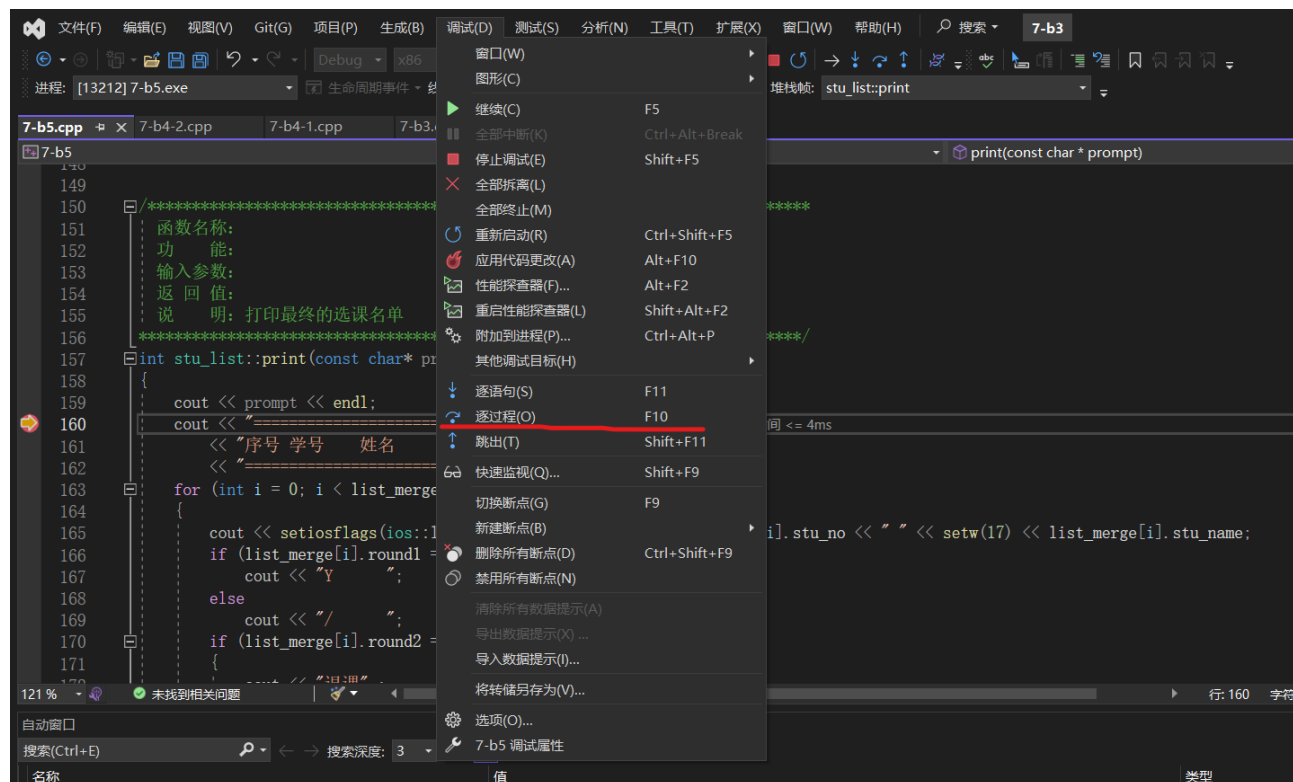


5、如果需要逐语句执行：按下F11，这个模式不会跳过这个函数中的函数而是继续执行计算机实际执行的指令



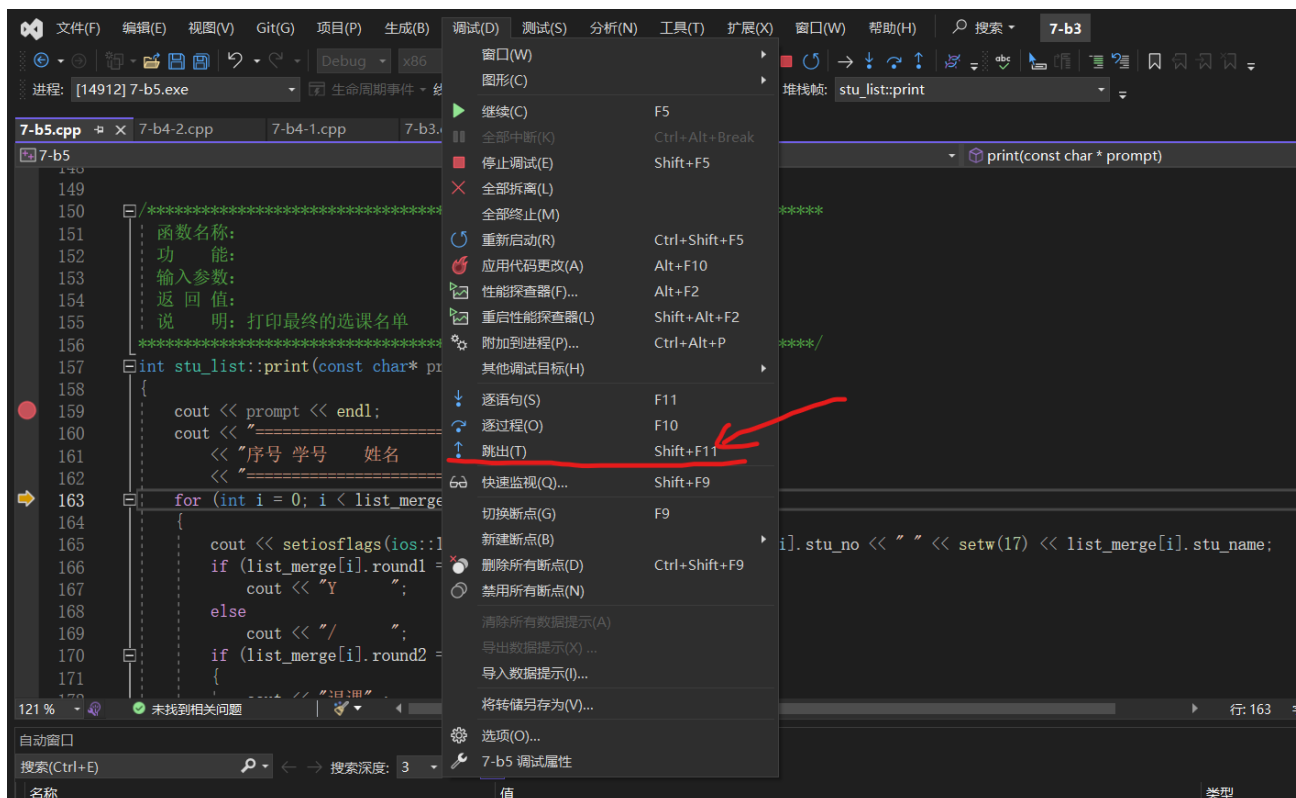
1.3. 在碰到cout/sqrt等系统类/系统函数时，一步完成这些系统类/系统函数的 执行而不要进入到这些系统类/系统函数的内部单步执行

可以通过逐过程的调试功能一步执行这条指令，使用逐过程的方法如1.2中方法相同，使用F10，或者再调试选项中选择逐过程



1.4. 如果已进入cout/sqrt等系统类/系统函数的内部，跳出并返回自己的函数

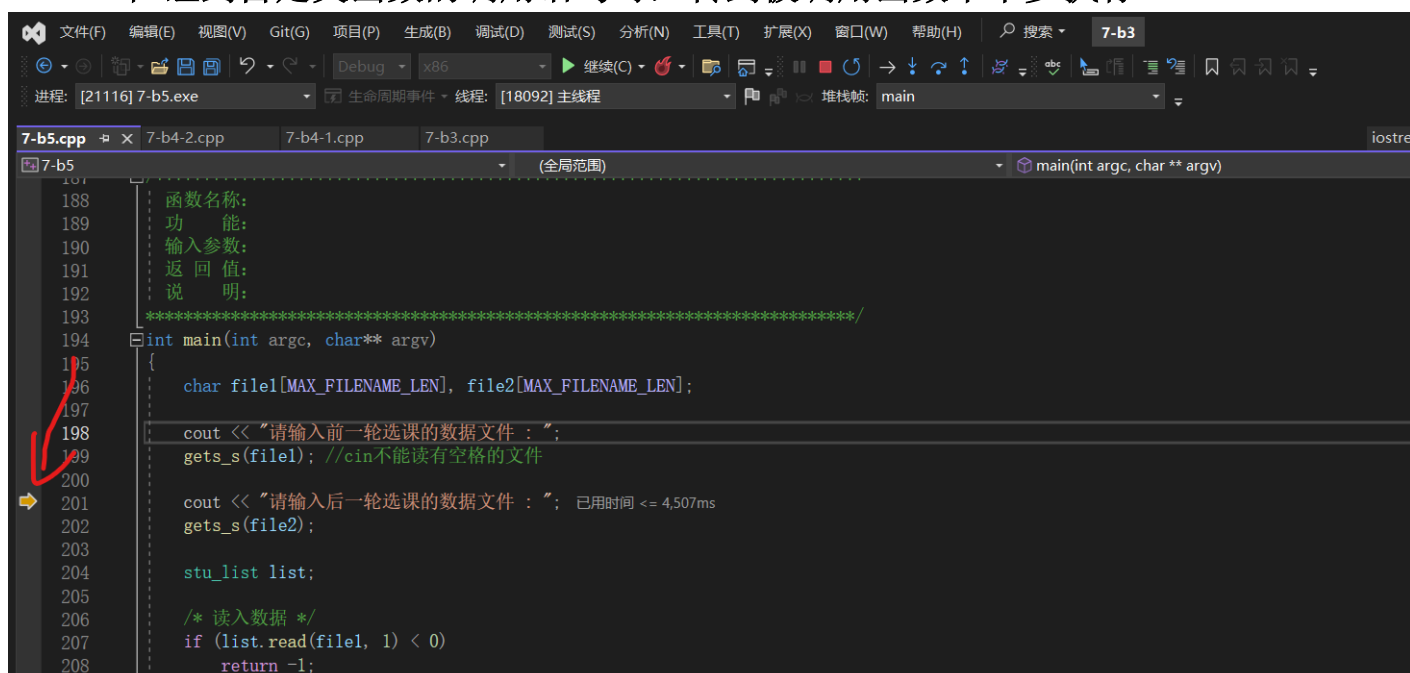
可以通过跳出的调试功返回自己的函数，使用逐过程的方法如1.2中方法相同，使用Shift+F11，或者再调试选项中选择跳出。



1.5. 在碰到自定义函数的调用语句时，一步完成自定义函数的执行而不要进入到这些自定义函数的内部单步执行

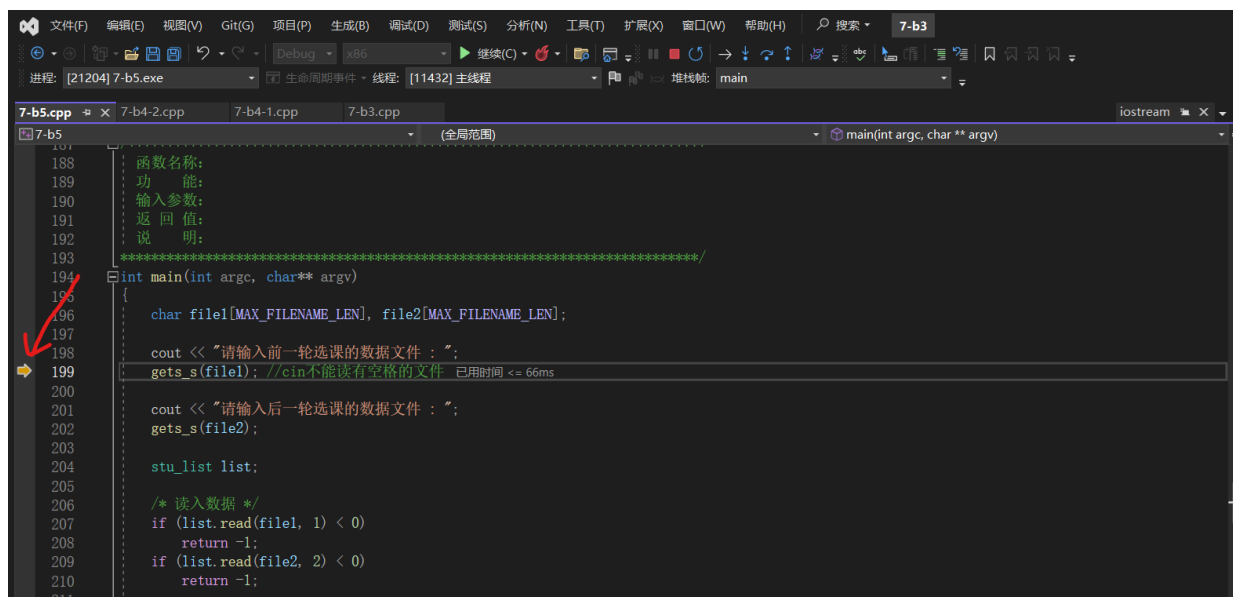
碰到这种情况可以使用逐过程的调试功能一步执行这条指令，使用逐过程的方法如1.2中方法相同，使用F10，或者在调试选项中选择逐过程。

1.6. 在碰到自定义函数的调用语句时，转到被调用函数中单步执行



在这种情况下需要逐语句执行：按下F11，这个模式不会跳过这个函数中的函数而是继续执行计算机实际执行的指令

按下F11后跳转到了merge函数中

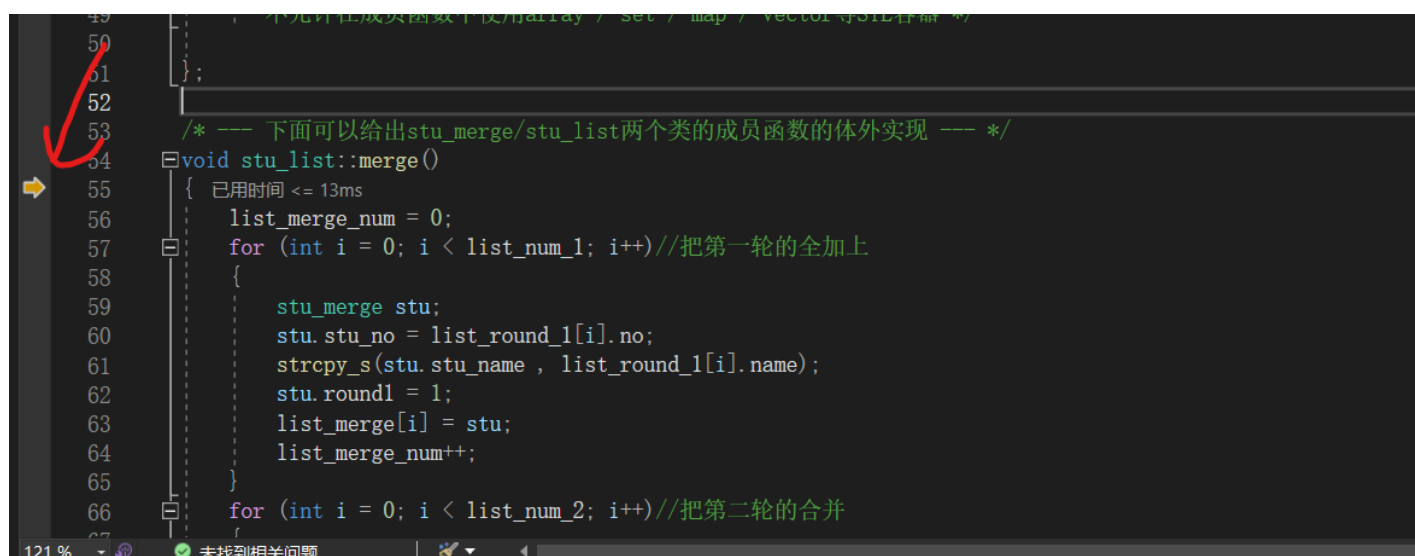


2. 用VS2022的调试工具查看各种生存期/作用域变量的方法

2.1. 查看形参/自动变量的变化情况

在调试过程中，当程序执行到断点时，在屏幕的下方会有显示自动窗口，在自动窗口中会显示当前的形参和自动变量

如果没有自动窗口可以在调试选项->窗口->自动变量中调出自动变量窗口。



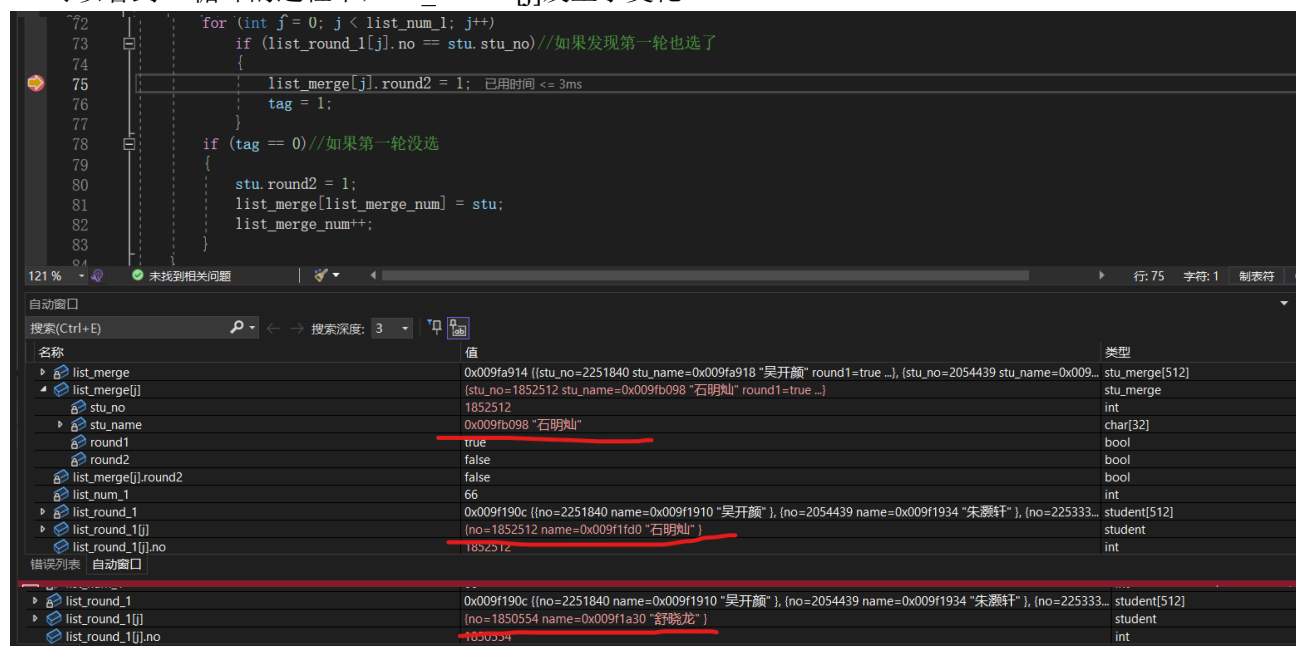
具体使用方法：

开始调试后，自动窗口会实时显示走到当前程序变量的值，每走一步都可以看到形参，或自动变量的变化，每一步发生变化的值，VS编译器会将这个值标红。



以这个7-b5为例

可以看到for循环的过程中，list_round1[j]发生了变化。

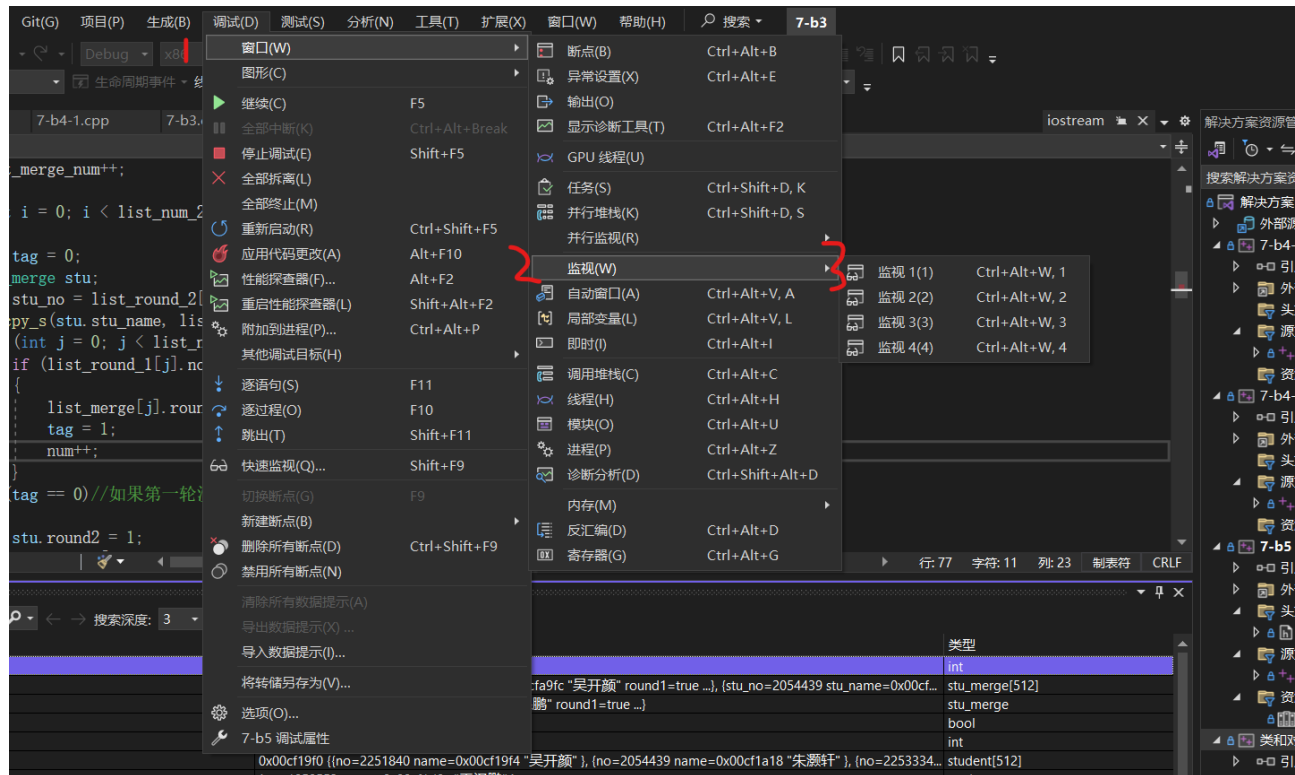


2.2. 查看静态局部变量的变化情况（该静态局部变量所在的函数体内/函数体

外)

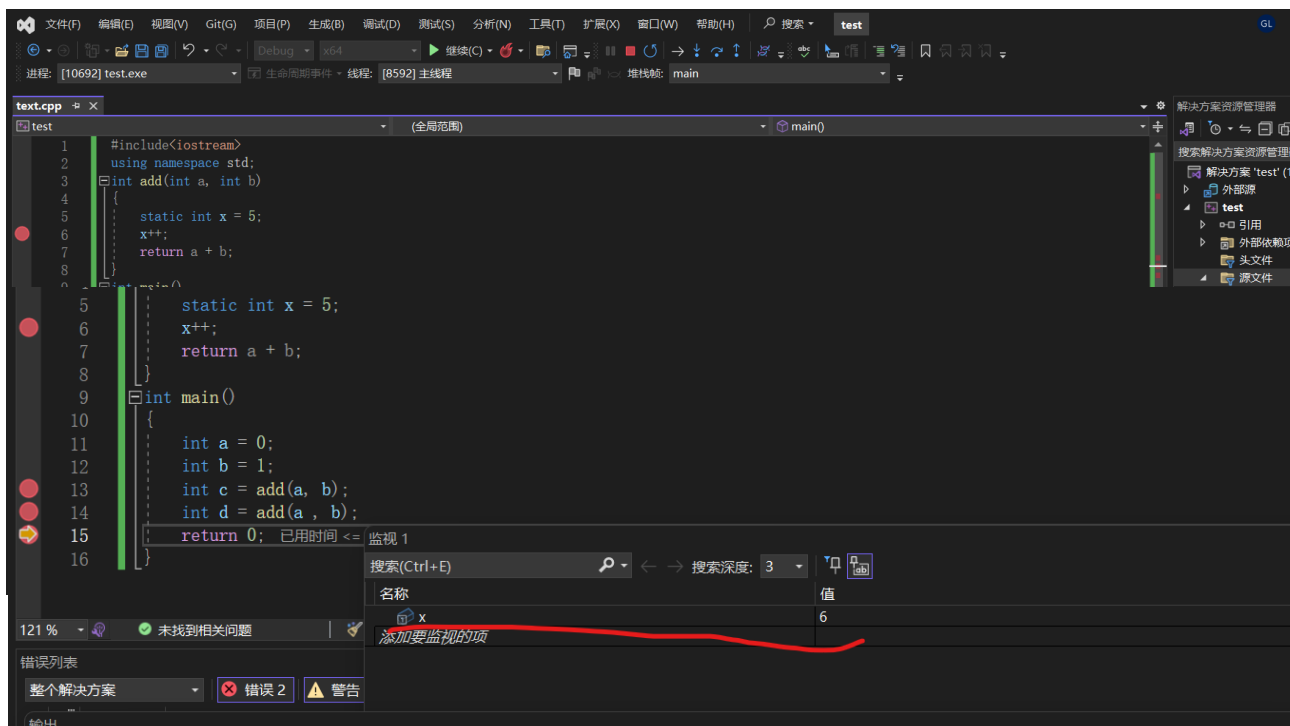
查看静态局部变量的变化情况可以使用调试中监视的功能，在监视中添加需要监视的变量。

如何打开监视：



例子：

监视静态局部变量x的值的变化



可以看到在函数体外也能够查看x的值

2.3. 查看静态全局变量的变化情况（两个源程序文件，有同名静态全局变量）

同样可以用监视的方法查看

```

2.cpp
1 #include<iostream>
2 using namespace std;
3 static int x = 10;
4 int add(int a, int b)
5 {
6     x++;
7     return a + b;
8 }
9 int main()
10 {
11     int a = 0;
12     int b = 1;
13     x++;
14     int c = add(a, b);
15     int d = add(a, b);
16     return 0;
17 }

text.cpp
1 #include<iostream>
2 using namespace std;
3 static int x = 5;
4 int add(int a, int b)
5 {
6     x++;
7     return a + b;
8 }
9 int main()
10 {
11     int a = 0;
12     int b = 1;
13     x++;
14     int c = add(a, b);
15     int d = add(a, b);
16     return 0;
17 }
    
```

在Cpp1中监视的x初值为5

监视 1

名称	值
x	6

添加要监视的项

在cpp2中监视的x的初值为10.

监视 1

名称	值
x	11

添加要监视的项

2.4. 查看外部全局变量的变化情况（两个源程序文件，一个进行定义，另一个进行extern说明）

仍然使用监视的方式查看两边的变量的值的变化。

下面是一个例子：

The screenshot shows two source files in Visual Studio:

- text.cpp:**

```

1  #include<iostream>
2  using namespace std;
3  int x = 5;
4  int add(int a, int b)
5  {
6      int a = 0;
7      int b = 1;
8      x++;
9  }
10 int main()
11 {
12     int c = add(a, b);
13     int d = add(a, b);
14     return 0;

```
- teat2.cpp:**

```

1  #include<iostream>
2  using namespace std;
3  extern int x ;
4  int add(int a, int b)
5  {
6      x++;
7      return a + b;
8  }

```

The variable **x** is being monitored, and its value is shown as **6** in the '监视' (Watch) window.

The screenshot shows the same two source files in Visual Studio, but with a different view of the code:

- text.cpp:**

```

1  #include<iostream>
2  using namespace std;
3  int x = 5;
4  int add(int a, int b)
5  {
6      int a = 0;
7      int b = 1;
8      x++;
9  }
10 int main()
11 {
12     int c = add(a, b);
13     int d = add(a, b);
14     return 0;

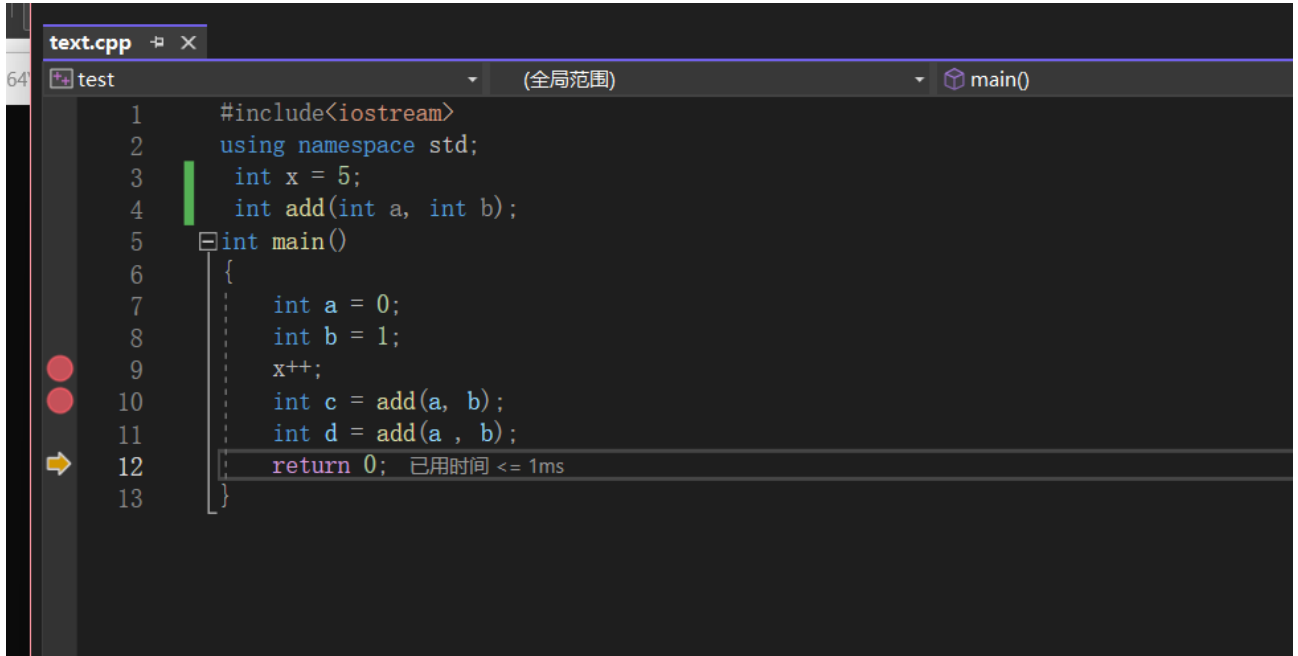
```
- teat2.cpp:**

```

1  #include<iostream>
2  using namespace std;
3  extern int x ;
4  int add(int a, int b)
5  {
6      x++;
7      return a + b;
8  }

```

The variable **x** is being monitored, and its value is shown as **6** in the '监视' (Watch) window.



```

1  #include<iostream>
2  using namespace std;
3  int x = 5;
4  int add(int a, int b);
5  int main()
6  {
7      int a = 0;
8      int b = 1;
9      x++;
10     int c = add(a, b);
11     int d = add(a, b);
12     return 0; 已用时间 <= 1ms
13 }
    
```

通过例子可以看到两个cpp中的全局变量是同一个变量，在调试过程中值的变化是一致的。

3. 用VS2022的调试工具查看各种不同类型变量的方法

可以在局部变量，自动变量，监视三种窗口下查看这些变量

下面以局部变量为例

```

text.cpp  x
test      (全局范围)  main()
1
2   using namespace std;
3   int x = 5;
4   int add(int a, int b);
5   int main()
6   {
7       int a = 1;
8       short b = 2;
9       char c = 'a';
10      float f = 1.0;
11      double d = 2.0;
12      int* p1 = &a; //指针
13      int num[5] = { 1, 2, 3, 4, 5 }; //一维数组
14      int* p2 = num; //指向一维数组的指针
15      int array[2][2] { {1, 2}, {3, 4} }; //二维数组
16      const char* s = "hello"; //指向字符串常量
17      int* p3 = array[0]; //指向二维数组的指针
18      int& r = a; //引用
19      return 0;
20  }
    
```

局部变量	
搜索(Ctrl+E) 搜索深度: 3	
名称	值
a	1
array	0x0000006b97eff848 (0x0000006b97eff848 {1, 2}, 0x0000006b97eff850 {3, 4})
array[0]	0x0000006b97eff848 {1, 2}
array[1]	0x0000006b97eff850 {3, 4}
b	2
c	97 'a'
d	2.0000000000000000
f	1.00000000
num	0x0000006b97eff7f8 {1, 2, 3, 4, 5}
num[0]	1
num[1]	2
num[2]	3
num[3]	4
num[4]	5
p1	0x0000006b97eff734 (1)
p2	0x0000006b97eff7f8 (1)
p3	0x0000006b97eff848 (1)
r	1
s	0x00007ff6708d9bd8 "hello"
	104 'h'

第三大题中所提到的变量通过局部变量的窗口可以查找。

3.2: 查看地址: 在 {} 之前的一部分为这个指针所指的地址, {} 中的值为指向地址的值。

3.4: 查看地址: 在 {} 之前的一部分为这个指针所指的地址, {} 中的值为指向地址的值。通过比较发现指向一维数组的指针变量实际上指向的是一维数组第一个元素的地址, 他的值也是第一个元素的值

3.5: 二维数组就是一维数组结构的叠加, 有两层的 {}, 第一层花括号中是1这

个元素的首地址，和3这个元素的首地址，第二层 {} 中是二维数组中元素的值。

3.6: 如何查看数组地址和值：可以在函数中对指针进行递增这样可以在局部变量窗口看到数组的地址和值

```

3      int x = 5;
4      int add(int* num)
5      {
6          num++;
7          num++;
8          return *num;
9      }
10     int main()
11     {
12         int a = 1;

```

```

1  #include<iostream>
2  using namespace std;
3  int x = 5;
4  int add(int* num)
5  {
6      num++;
7      num++; 已用时间 <= 2ms
8      return *num;
9  }
10 int main()
11 {
12     int a = 1;
13     short b = 2;
14     char c = 'a';
15     float f = 1.0;
16     double d = 2.0;
17     int* p1 = &a; //指针
18     int num[5] = { 1, 2, 3, 4, 5 }; //一维数组
19     int* p2 = num; //指向一维数组的指针
20     int array[2][2] { {1, 2}, {3, 4} }; //二维数组
21     const char* s = "hello"; //指向字符串常量
22     int* p3 = array[0]; //指向二维数组的的指针
23     int& r = a; //引用

```

121 % 未找到相关问题

名称	值
num	0x000000e7058ff90c (2)

```

3   int x = 0;
4   int add(int* num)
5   {
6       num++;
7       num++;
8       return *num;
9   }
10  int main()
11  {
12      int a = 1;
13      short b = 2;
14      char c = 'a';
15      float f = 1.0;
16      double d = 2.0;
17      int* p1 = &a; // 指针
18      int num[5] = { 1, 2, 3, 4, 5 }; // 一维数组
19      int* p2 = num; // 指向一维数组的指针
20      int array[2][2] { { 1, 2 }, { 3, 4 } }; // 二维数组
21      const char* s = "hello"; // 指向字符串常量
22      int* p3 = array[0]; // 指向二维数组的的指针
23      int& r = a; // 引用

```

121 % 未找到相关问题

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值
num	0x000000e7058ff908 (1)

3.7: 查看字符串常量

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值
a	1
array	0x000000e7058ff958 {0x000000e7058ff958 {1, 2}, 0x000000e7058ff960 {3, 4}}
b	2
c	97 'a'
d	2.0000000000000000
f	1.00000000
num	0x000000e7058ff908 {1, 2, 3, 4, 5}
p1	0x000000e7058ff844 {1}
p2	0x000000e7058ff908 {1}
p3	0x000000e7058ff958 {1}
r	1
s	0x00007ff6554f9bd8 "hello"
	104 'h'

可以看到空字符的地址和值，值为' \0'

s	0x00007ff78c879bde ""
	0 '\0'

3.8: 引用与指针有区别，

在C++中，引用和指针都是用于处理内存地址的概念，但它们有一些重要的区别。以下是引用和指针的主要区别：

语法和声明：

引用： 引用是一个别名，通过使用 & 符号声明。例如：int a = 10; int &ref = a;

指针： 指针是一个变量，存储了一个内存地址，通过使用 * 符号声明。例如：int a = 10; int *ptr = &a;

空值 (NULL)：

引用： 引用必须在创建时被初始化，并且不能为NULL。因此，引用总是指向某个有效的对象。

指针： 指针可以在创建后被设置为NULL，表示它不指向任何有效的对象。

重指向：

引用： 一旦引用被初始化，就不能再引用其他对象。引用始终指向初始引用的对象。

指针： 指针可以在运行时被重新赋值，指向其他对象。指针的值可以改变。

空间占用：

引用： 引用不占用额外的内存空间，它只是被认为是已存在变量的别名。

指针： 指针占用额外的内存空间，存储了一个地址值。

3.9:越界访问

这个指针会被指向一个没有意义的地方，地址上的值也是不可信值

