



《高级语言程序设计》实验报告

实验报告名称 Visual Studio 2022 调试工具的基本使用方法

姓 名	<u>林继申</u>
学 号	<u>2250758</u>
学 院	<u>新生院</u>
专 业	<u>工科试验班(信息类)</u>
班 级	<u>信息类07班</u>
教 师	<u>沈 坚</u>

完成日期：2023 年 6 月 3 日

1. VS2022下调试工具的基本使用方法

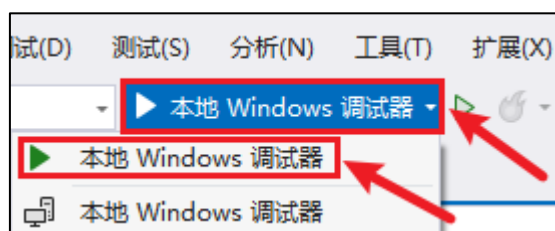
1.1. 开始调试和结束调试

1.1.1. 开始调试

按下F5键，或者选择“调试”菜单中的“开始调试”选项，或者点击工具栏上的“本地Windows调试器”按钮。这将启动程序，并进入调试器。



(上图涉及1.1相关内容)



(上图涉及1.1相关内容)

1.1.2. 结束调试

按下Shift+F5键，或者选择“调试”菜单中的“停止调试”选项，或者点击工具栏上的“停止调试”按钮。这将终止程序，并退出调试器。



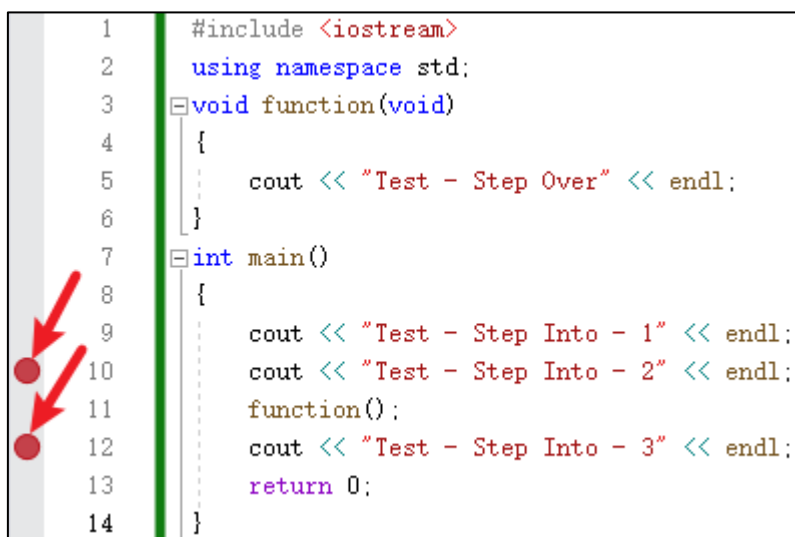
(上图涉及1.1相关内容)



(上图涉及1.1相关内容)

1.2. 在一个函数中每个语句单步执行

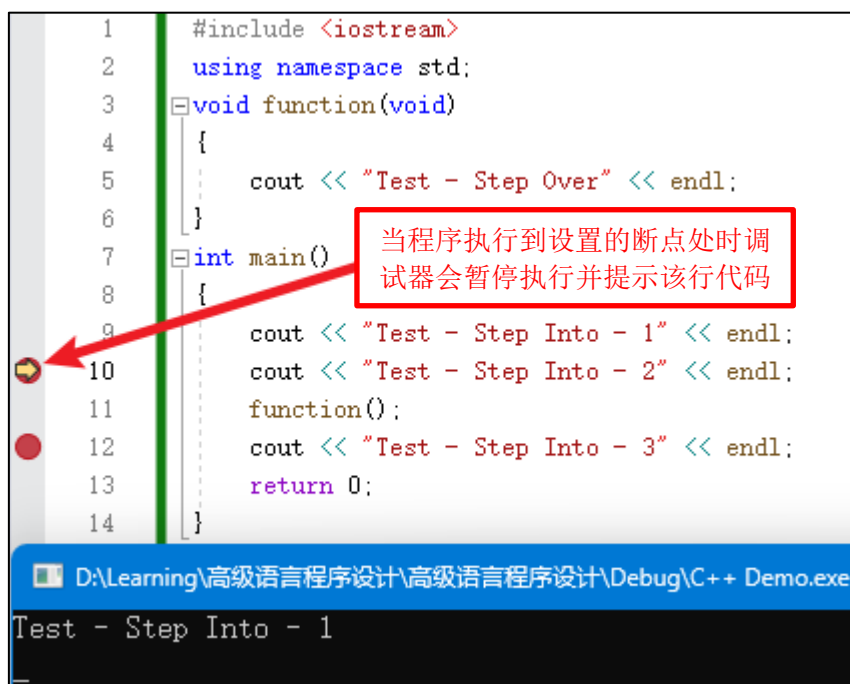
①在想要调试的函数内设置一个断点。在函数的左侧边栏单击代码行号旁边的空白区域，出现一个红色的圆点，表示设置了断点。断点是希望在该位置停止执行代码的地方。



(上图涉及1.2相关内容)

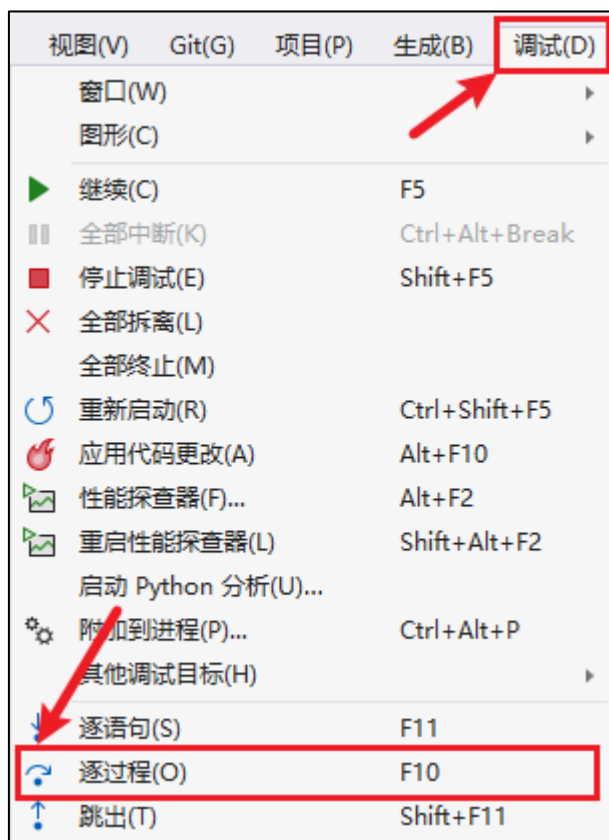
②开始调试。按下F5键，或者选择“调试”菜单中的“开始调试”选项，或者点击工具栏上的“本地Windows调试器”按钮。这将启动程序，并进入调试器。

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。可以使用调试菜单中的选项或调试工具栏上的按钮来控制执行过程。



(上图涉及1.2相关内容)

逐过程执行：按下F10键，或者选择“调试”菜单中的“逐过程”选项，或者点击调试工具栏上的“逐过程”按钮。这将逐过程执行代码。

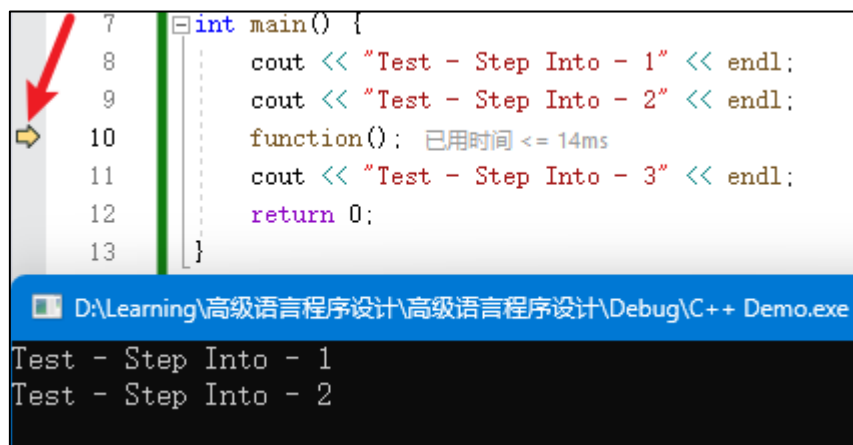


(上图涉及1.2相关内容)



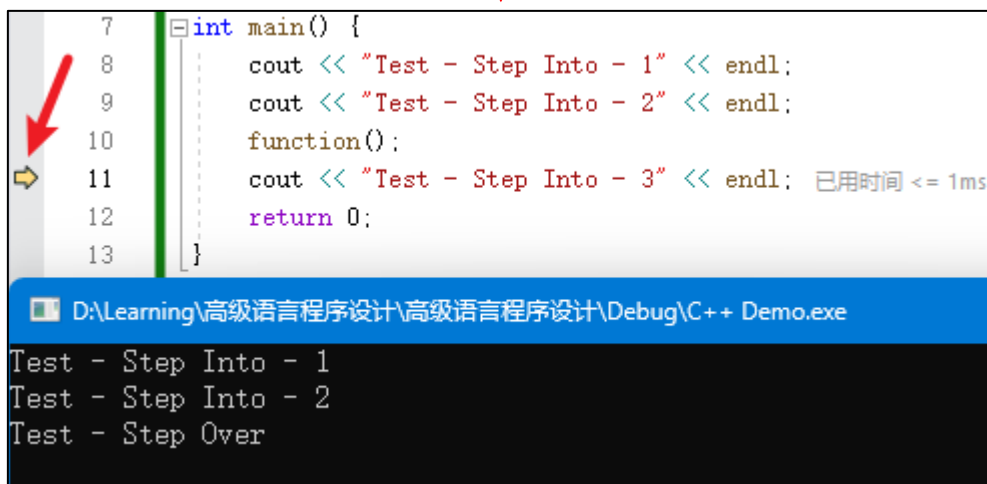
(上图涉及1.2相关内容)

逐过程执行适用于不关心函数内部的具体执行过程，只关心函数的整体结果。如果函数调用是不关心的或是第三方库的函数，可以快速跳过该函数，节省调试时间。



按下 F10 逐过程执行

当调试器遇到一个函数调用时，不会进入该函数而是跳过整个函数，停在函数调用的下一行



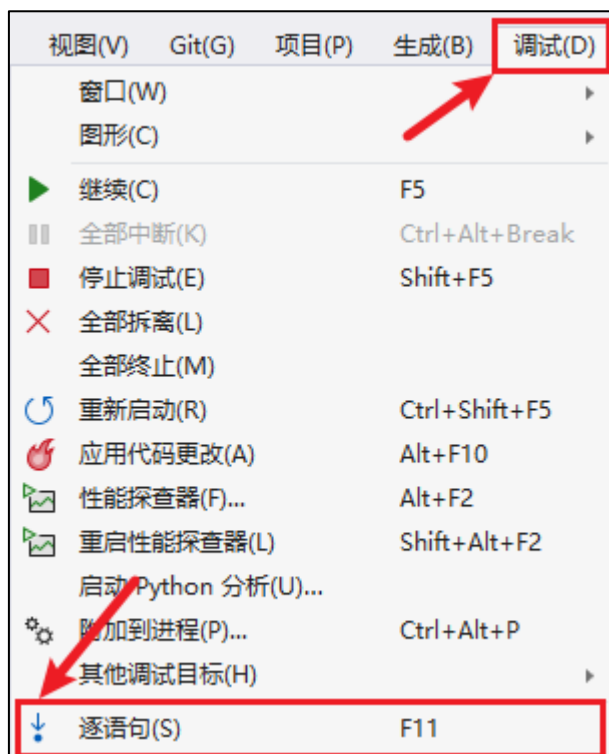
```

7  int main() {
8      cout << "Test - Step Into - 1" << endl;
9      cout << "Test - Step Into - 2" << endl;
10     function();
11     cout << "Test - Step Into - 3" << endl; 已用时间 <= 1ms
12     return 0;
13 }
    
```

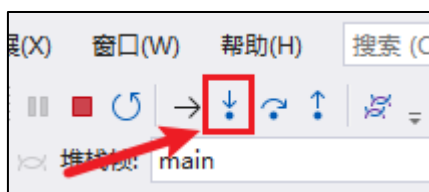
Test - Step Into - 1
Test - Step Into - 2
Test - Step Over

(上图涉及1.2相关内容)

逐语句执行：按下F11键，或者选择“调试”菜单中的“逐语句”选项，或者点击调试工具栏上的“逐语句”按钮。这将逐语句执行代码。

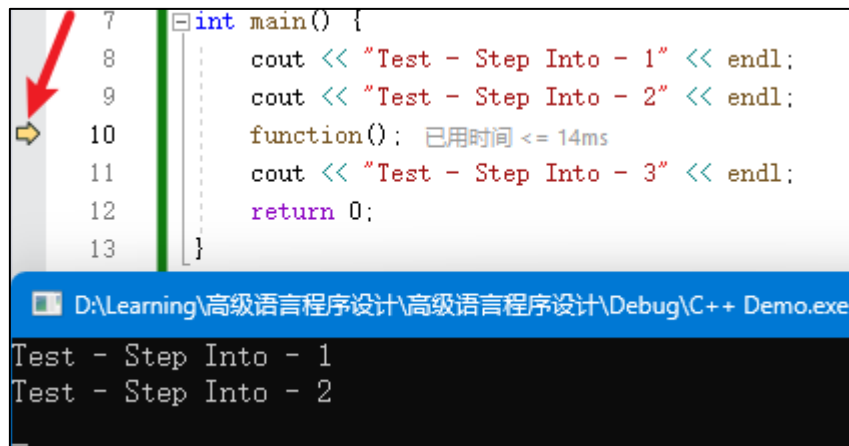


(上图涉及1.2相关内容)



(上图涉及1.2相关内容)

逐语句执行适用于查看函数内部的具体执行过程，逐步跟踪函数内部的语句执行。如果函数调用是自己的代码或标准库的函数，可以查看函数的内部实现并逐步执行。



```

7  int main() {
8      cout << "Test - Step Into - 1" << endl;
9      cout << "Test - Step Into - 2" << endl;
10     function(); 已用时间 <= 14ms
11     cout << "Test - Step Into - 3" << endl;
12     return 0;
13 }

```

D:\Learning\高级语言程序设计\高级语言程序设计\Debug\C++ Demo.exe

Test - Step Into - 1
Test - Step Into - 2

按下 F11 逐语句执行

当调试器遇到一个函数调用时，会进入该函数并停在函数内的第一行



```

3  void function(void)
4  { 已用时间 <= 3ms
5      cout << "Test - Step Over" << endl;
6  }
7  int main() {
8      cout << "Test - Step Into - 1" << endl;
9      cout << "Test - Step Into - 2" << endl;
10     function();
11     cout << "Test - Step Into - 3" << endl;
12     return 0;
13 }

```

D:\Learning\高级语言程序设计\高级语言程序设计\Debug\C++ Demo.exe

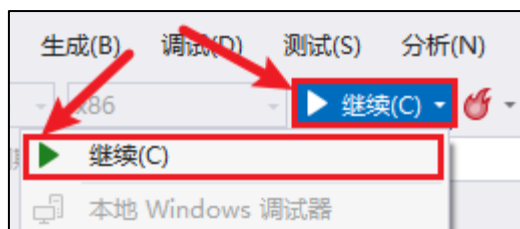
Test - Step Into - 1
Test - Step Into - 2

(上图涉及1.2相关内容)

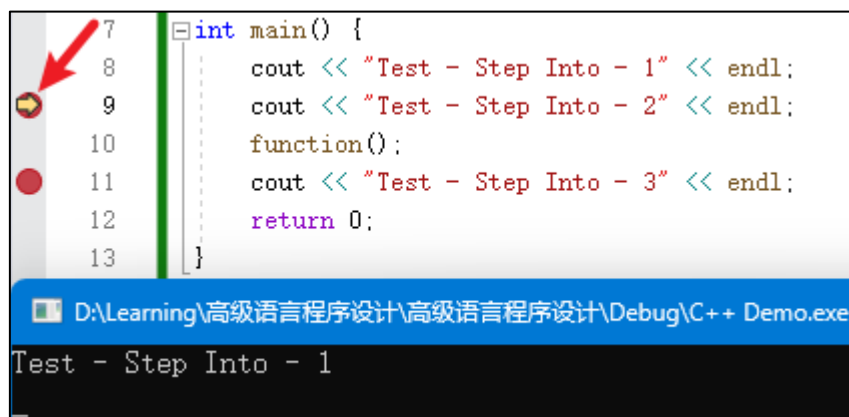
继续执行：按下F5键，或者选择“调试”菜单中的“继续”选项，或者点击调试工具栏上的“继续”按钮。这将继续执行代码，直到下一个断点或程序结束。



(上图涉及1.2相关内容)



(上图涉及1.2相关内容)



按下 F5 继续执行

按下 F5 键将继续执行代码直到下一个断点或程序结束



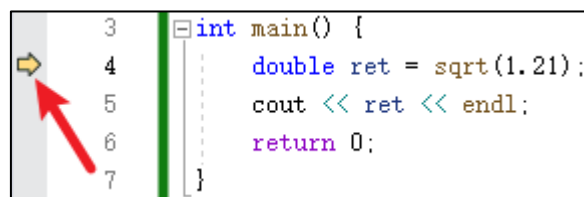
(上图涉及1.2相关内容)

④结束调试。按下Shift+F5键，或者选择“调试”菜单中的“停止调试”选项，或者点击工具栏上的“停止调试”按钮。这将终止程序，并退出调试器。

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

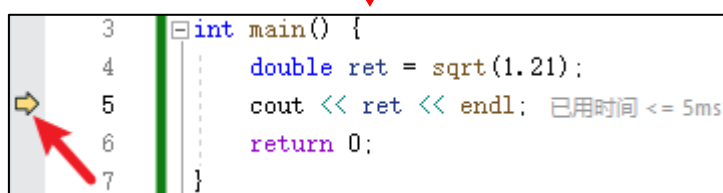
在调试模式下，当碰到cout/sqrt等系统类/系统函数时，可以通过“逐过程执行”一步完成这些系统

类/系统函数的执行而不要进入到这些系统类/函数的内部单步执行。具体方法为：按下F10键，或者选择“调试”菜单中的“逐过程”选项，或者点击调试工具栏上的“逐过程”按钮。



按下 F10 逐过程执行

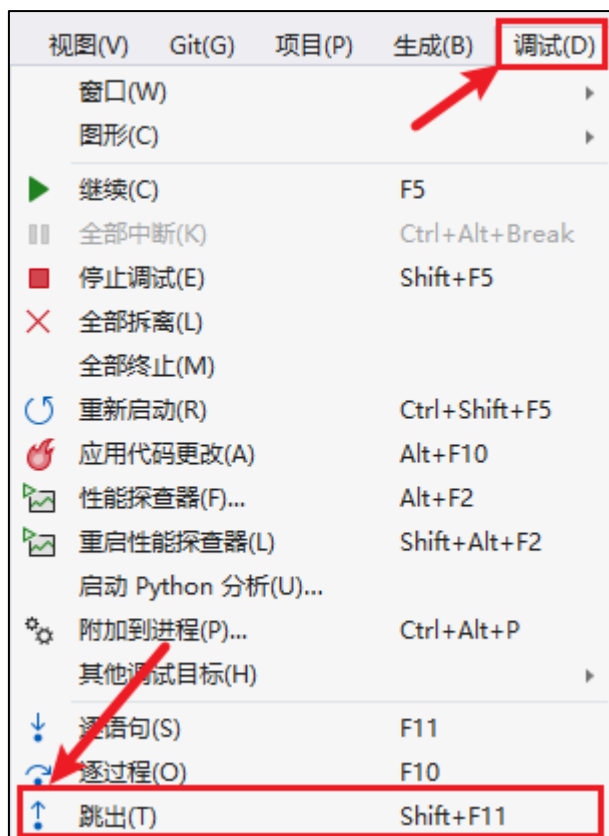
通过逐过程执行一步完成这些系统类/系统函数的执行而不要进入到这些系统类/函数内部单步执行



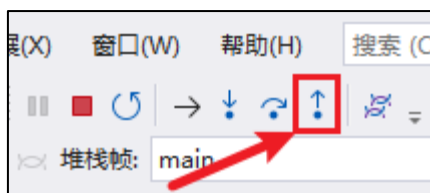
(上图涉及1.3相关内容)

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

在调试模式下，如果已进入cout/sqrt等系统类/系统函数的内部，可以通过“跳出”选项跳出并返回自己的函数。具体方法为：按下Shift+F11键，或者选择“调试”菜单中的“跳出”选项，或者点击调试工具栏上的“跳出”按钮。这将跳出当前系统类/系统函数的内部并返回自己的函数。



(上图涉及1.4相关内容)



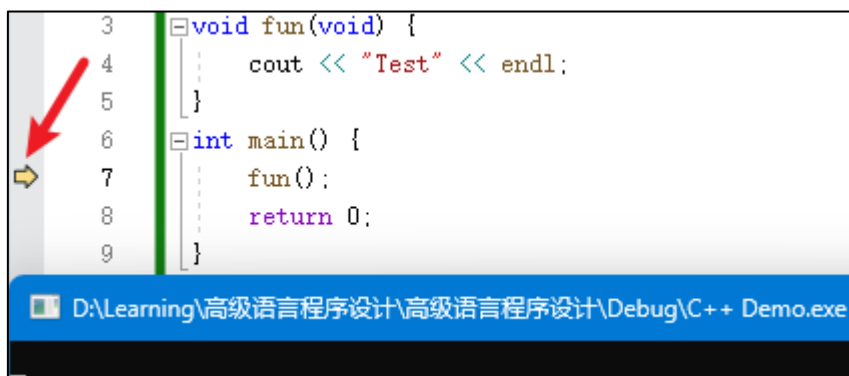
(上图涉及1.4相关内容)

在Visual Studio 2022中,对于像cout/sqrt等系统类/系统函数,默认设置为不能直接进入这些系统类/系统函数的内部单步执行。

可以通过在“选项”>“调试”>“常规”菜单中取消“启用“仅我的代码””选项,来直接进入这些系统类/系统函数的内部单步执行。

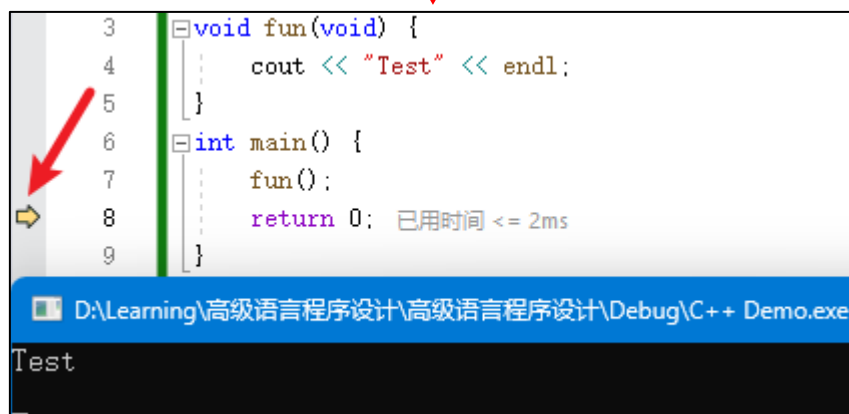
1.5. 在碰到自定义函数的调用语句时,一步完成自定义函数的执行而不要进入到这些自定义函数的内部单步执行

在碰到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,可以通过“逐过程执行”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。具体方法为:按下F10键,或者选择“调试”菜单中的“逐过程”选项,或者点击调试工具栏上的“逐过程”按钮。



按下 F10 逐过程执行

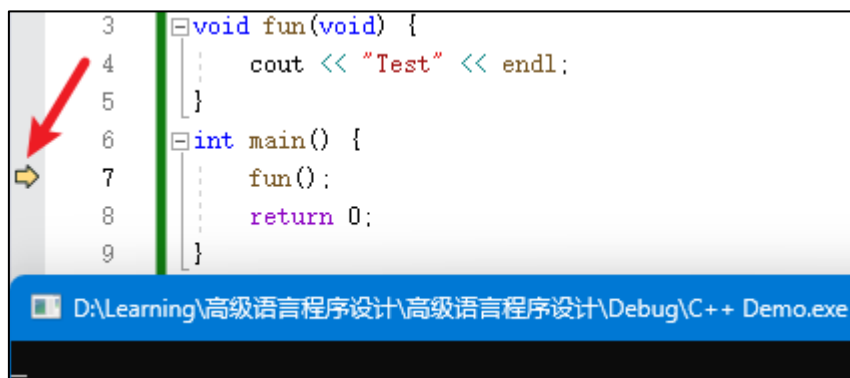
通过“逐过程执行”一步完成自定义函数的执行而不要进入到这些自定义函数的内部单步执行



(上图涉及1.5相关内容)

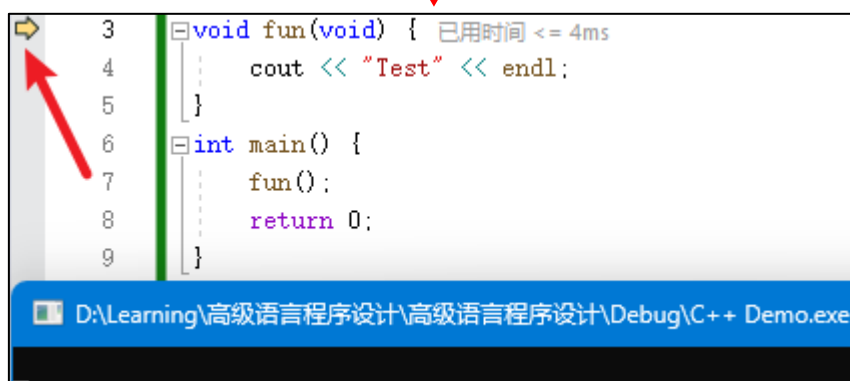
1.6. 在碰到自定义函数的调用语句时，转到被调用函数中单步执行

在碰到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，可以通过“逐语句执行”转到被调用函数中单步执行。具体方法为：按下F11键，或者选择“调试”菜单中的“逐语句”选项，或者点击调试工具栏上的“逐语句”按钮。这将逐语句执行代码。



按下 F11 逐语句执行

通过“逐语句执行”转到被调用函数中单步执行



（上图涉及1.6相关内容）

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

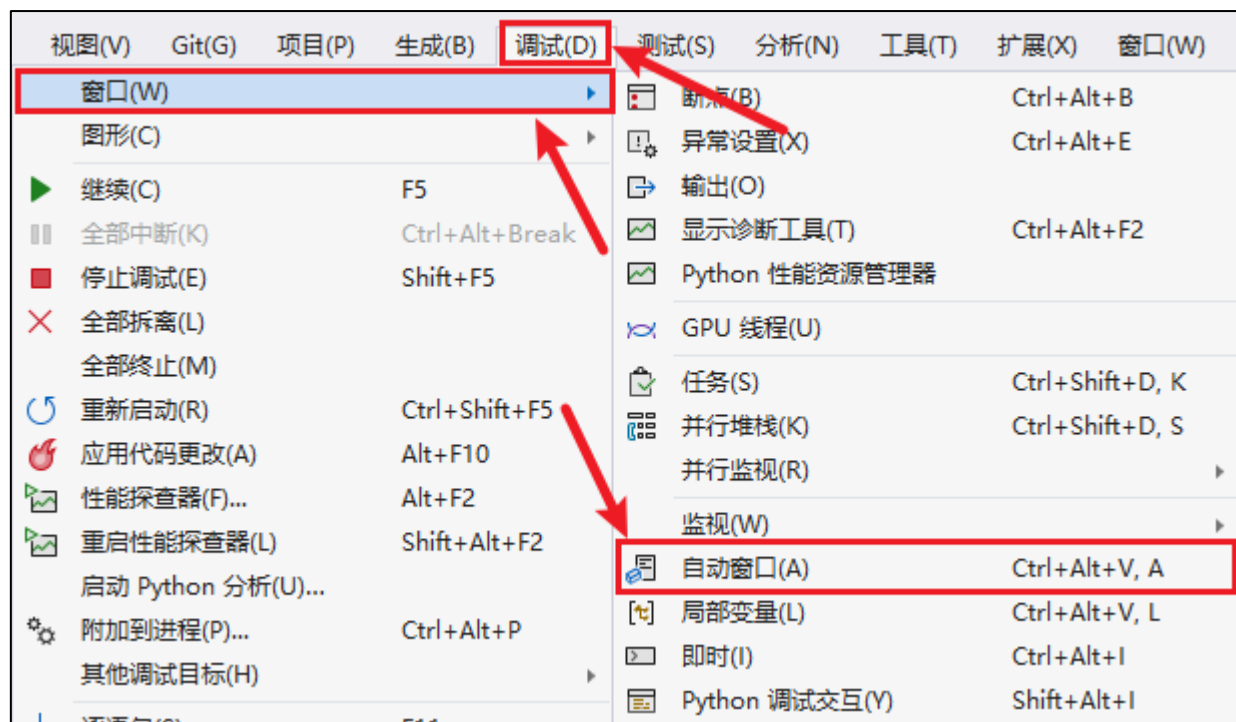
①在想要调试的函数内设置一个断点。在函数的左侧边栏单击代码行号旁边的空白区域，出现一个红色的圆点，表示设置了断点。

②开始调试。按下F5键，或者选择“调试”菜单中的“开始调试”选项，或者点击工具栏上的“本地Windows调试器”按钮。这将启动程序，并进入调试器。

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。此时可以查看形参/自动变量的名称、当前值和类型。

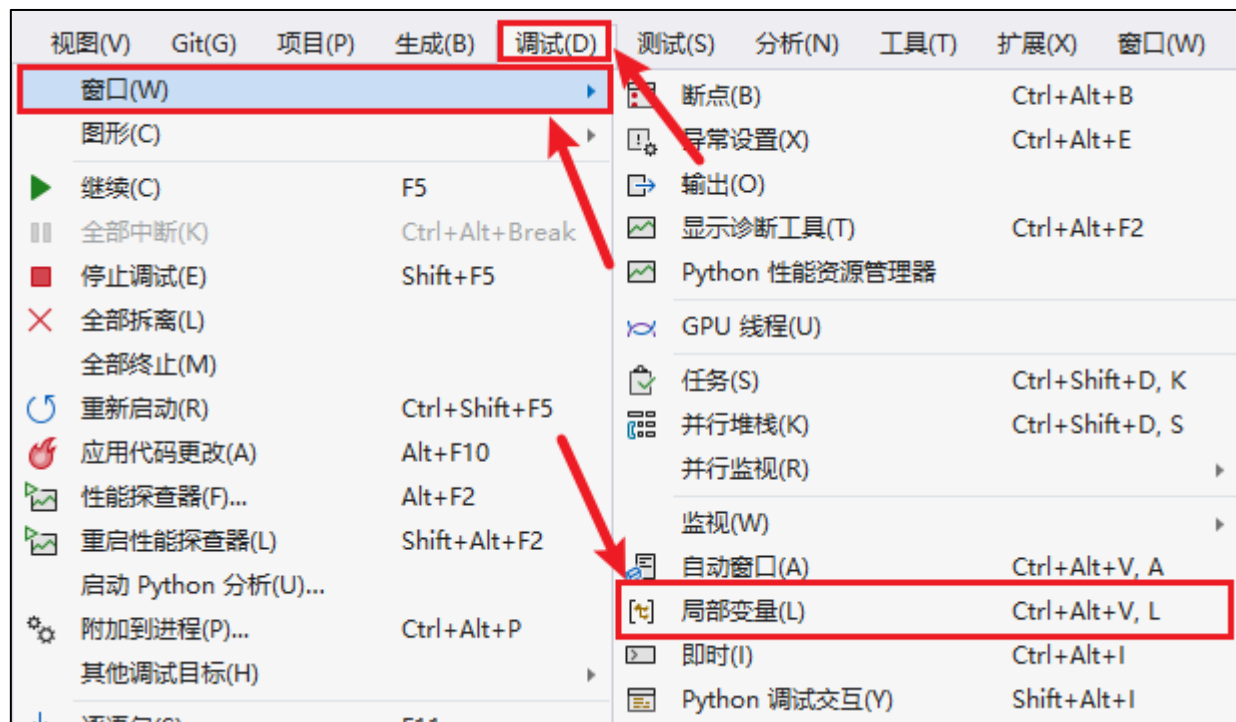
在“自动窗口”窗口中查看变量：在调试模式下打开“自动窗口”窗口的具体方法为在菜单栏中选择“调试”>“窗口”>“自动窗口”，快捷键为Ctrl+Alt+V,A。“自动窗口”窗口显示当前作用域内的自动变

量，在“自动窗口”窗口中，可以看到变量的名称、当前值和类型。



(上图涉及2.1相关内容)

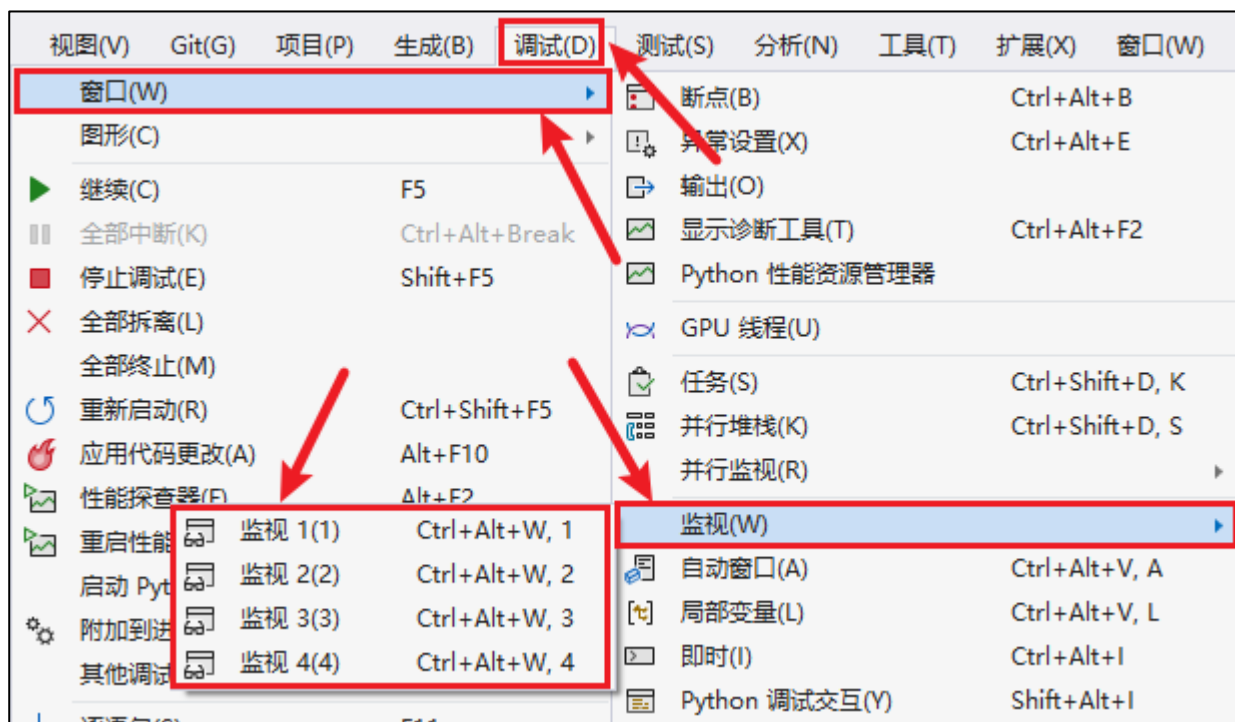
在“局部变量”窗口中查看变量：在调试模式下打开“局部变量”窗口的具体方法为在菜单栏中选择“调试”>“窗口”>“局部变量”，快捷键为Ctrl+Alt+V, L。“局部变量”窗口显示当前活动函数的所有局部变量，在“局部变量”窗口中，可以看到变量的名称、当前值和类型。



(上图涉及2.1相关内容)

在“监视”窗口中查看变量：在调试模式下打开“监视”窗口的具体方法为在菜单栏中选择“调试”>

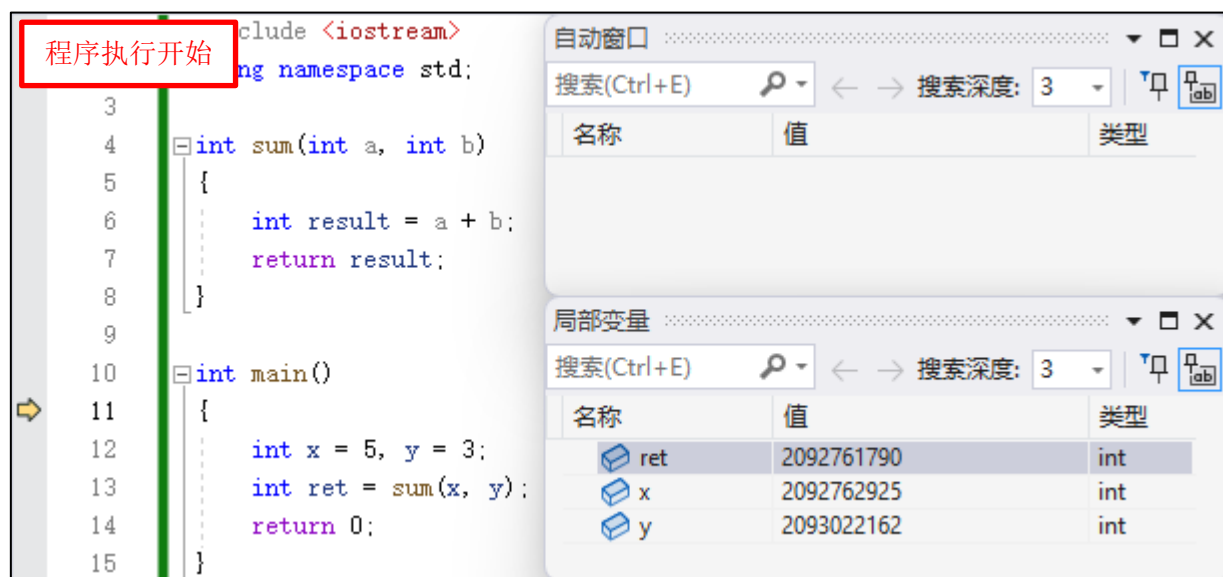
“窗口”>“监视”>“监视1/监视2/监视3/监视4”，快捷键为Ctrl+Alt+W, 1/2/3/4。“监视”窗口允许手动添加要监视的特定变量，在“监视”窗口中，可以看到被监视变量的名称、当前值和类型。



(上图涉及2.1相关内容)

下面是一个示例程序。在示例程序的执行过程中，每一步的自动变量和形参的变化过程如下：在main函数中，int型自动变量x和y分别被赋值为5和3；在调用sum函数时，int型形参a和b被赋值为x和y的值，即5和3；在sum函数中，int型自动变量result被赋值为a+b的结果，即8；sum函数返回result的值8，并赋值给int型自动变量ret；main函数返回0。

下面是调试模式下在“自动窗口”窗口和“局部变量”窗口中查看在示例程序执行过程中每一步形参/自动变量的变化情况。



进入 main 函数

```

1  #include <iostream>
2  using namespace std;
3
4  int sum(int a, int b)
5  {
6      int result = a + b;
7      return result;
8  }
9
10 int main()
11 {
12     int x = 5, y = 3;
13     int ret = sum(x, y);
14     return 0;
15 }
    
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
x	-858993460	int
y	-858993460	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
ret	-858993460	int
x	-858993460	int
y	-858993460	int

int 型自动变量 x 和 y 分别被赋值为 5 和 3

```

1  #include <iostream>
2  using namespace std;
3
4  int sum(int a, int b)
5  {
6      int result = a + b;
7      return result;
8  }
9
10 int main()
11 {
12     int x = 5, y = 3;
13     int ret = sum(x, y);
14     return 0;
15 }
    
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
ret	-858993460	int
x	5	int
y	3	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
ret	-858993460	int
x	5	int
y	3	int

调用 sum 函数时, int 型形参 a 和 b 被赋值为 x 和 y 的值, 即 5 和 3

```

1  #include <iostream>
2  using namespace std;
3
4  int sum(int a, int b)
5  {
6      int result = a + b;
7      return result;
8  }
9
10 int main()
11 {
12     int x = 5, y = 3;
13     int ret = sum(x, y);
14     return 0;
15 }
    
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	5	int
b	3	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	5	int
b	3	int
result	2093024741	int

定义 int 型自动变量 result

```

1  #include <iostream>
2  using namespace std;
3
4  int sum(int a, int b)
5  {
6      int result = a + b;
7      return result;
8  }
9
10 int main()
11 {
12     int x = 5, y = 3;
13     int ret = sum(x, y);
14     return 0;
15 }
    
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	5	int
b	3	int
result	-858993460	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	5	int
b	3	int
result	-858993460	int

int 型自动变量 result 被赋值为 a+b 的结果, 即 8

```

4  int sum(int a, int b)
5  {
6      int result = a + b;
7      return result; 已用时间
8  }
9
10 int main()
11 {
12     int x = 5, y = 3;
13     int ret = sum(x, y);
14     return 0;
15 }
    
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	5	int
b	3	int
result	8	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	5	int
b	3	int
result	8	int

sum 函数返回 result 的值 8

```

1  #include <iostream>
2  using namespace std;
3
4  int sum(int a, int b)
5  {
6      int result = a + b;
7      return result;
8  } 已用时间 <= 1ms
9
10 int main()
11 {
12     int x = 5, y = 3;
13     int ret = sum(x, y);
14     return 0;
15 }
    
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	5	int
b	3	int
result	8	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	5	int
b	3	int
result	8	int

sum 函数返回 8

```

1  #include <iostream>
2  using namespace std;
3
4  int sum(int a, int b)
5  {
6      int result = a + b;
7      return result;
8  }
9
10 int main()
11 {
12     int x = 5, y = 3;
13     int ret = sum(x, y);
14     return 0;
15 }
    
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
已返回 sum	8	int
ret	-858993460	int
x	5	int
y	3	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
已返回 sum	8	int
ret	-858993460	int
x	5	int
y	3	int

sum 函数返回 result 的值赋值给 int 型自动变量 ret, 即 8

```

1  #include <iostream>
2  using namespace std;
3
4  int sum(int a, int b)
5  {
6      int result = a + b;
7      return result;
8  }
9
10 int main()
11 {
12     int x = 5, y = 3;
13     int ret = sum(x, y);
14     return 0; 已用时间 <= 2
15 }
    
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
ret	8	int
x	5	int
y	3	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
ret	8	int
x	5	int
y	3	int

main 函数返回 0

```

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

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
ret	8	int
x	5	int
y	3	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
ret	8	int
x	5	int
y	3	int

(上图涉及2.1相关内容)

④结束调试。按下Shift+F5键，或者选择“调试”菜单中的“停止调试”选项，或者点击工具栏上的“停止调试”按钮。这将终止程序，并退出调试器。

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）

①在想要调试的函数内设置一个断点，并开始调试。

②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。此时可以查看静态局部变量的名称、当前值和类型。

下面是一个示例程序。在示例程序的执行过程中，每一步的静态局部变量的变化过程如下：第一次调用fun函数时，静态局部变量count被定义并初始化为0，count自增为1；第二次调用fun函数时，静态局部变量count的值保持为上一次的结果，即1，count自增为2。下面是调试模式下在“自动窗口”窗口、“局部变量”窗口和“监视”窗口中查看在示例程序执行过程中静态局部变量的变化情况。

第一次调用 fun 函数，静态局部变量 count 被定义并初始化为 0

```

4      void fun()
5      {
6          static int count = 0;
7          count++; 已用时间 <= 8ms
8      }
9
10     int main()
11     {
12         fun();
13         fun();
14         return 0;
15     }
                
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
count	0	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
count	0	int

监视 1

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
count	0	int

添加要监视的项

count 自增为 1

```

3      include <iostream>
4      using namespace std;
5
6      void fun()
7      {
8          static int count = 0;
9          count++;
10     } 已用时间 <= 3ms
11
12     int main()
13     {
14         fun();
15         fun();
16         return 0;
17     }
                
```

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
count	1	int

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
count	1	int

监视 1

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
count	1	int

添加要监视的项

静态局部变量 count 在所在函数体外，值保持上一次的结果不变，可以体现静态局部变量无论在函数体内外所占存储单元在程序执行过程中均不释放的特点

```

4 void fun()
5 {
6     static int count = 0;
7     count++;
8 }
9
10 int main()
11 {
12     fun();
13     fun(); 已用时间 <= 9ms
14     return 0;
15 }
    
```

名称	值	类型
count	1	int

第二次调用 fun 函数，静态局部变量 count 的值保持为上一次的结果，即 1

```

4 void fun()
5 {
6     static int count = 0;
7     count++; 已用时间 <= 7ms
8 }
9
10 int main()
11 {
12     fun();
13     fun();
14     return 0;
15 }
    
```

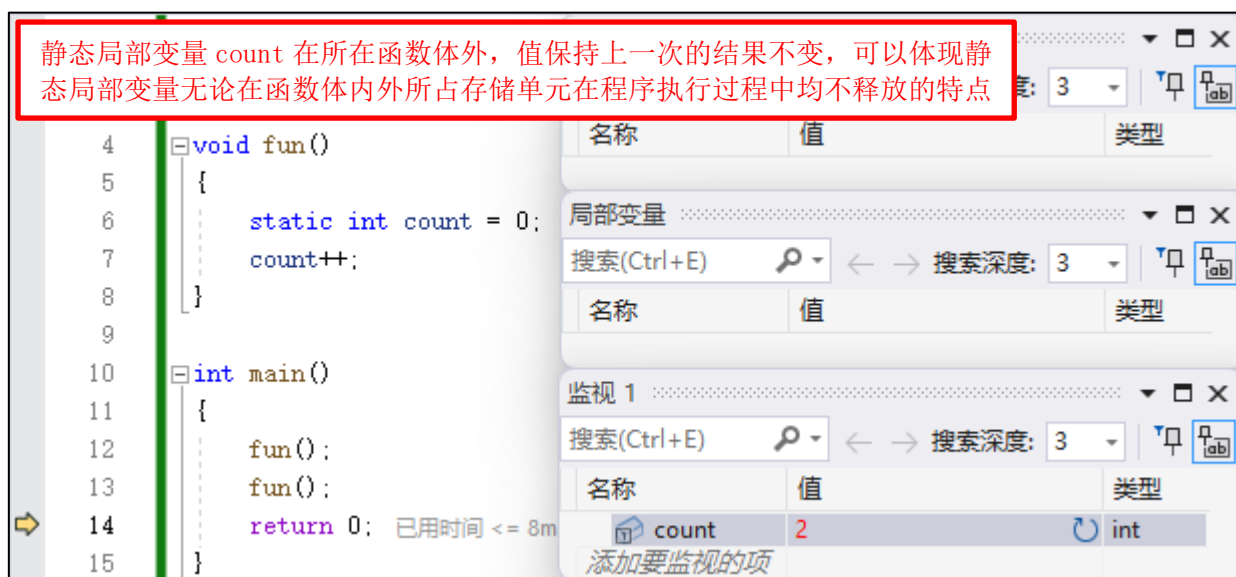
名称	值	类型
count	1	int

count 自增为 2

```

1 #include <iostream>
2 using namespace std;
3
4 void fun()
5 {
6     static int count = 0;
7     count++;
8 } 已用时间 <= 3ms
9
10 int main()
11 {
12     fun();
13     fun();
14     return 0;
15 }
    
```

名称	值	类型
count	2	int



(上图涉及2.2相关内容)

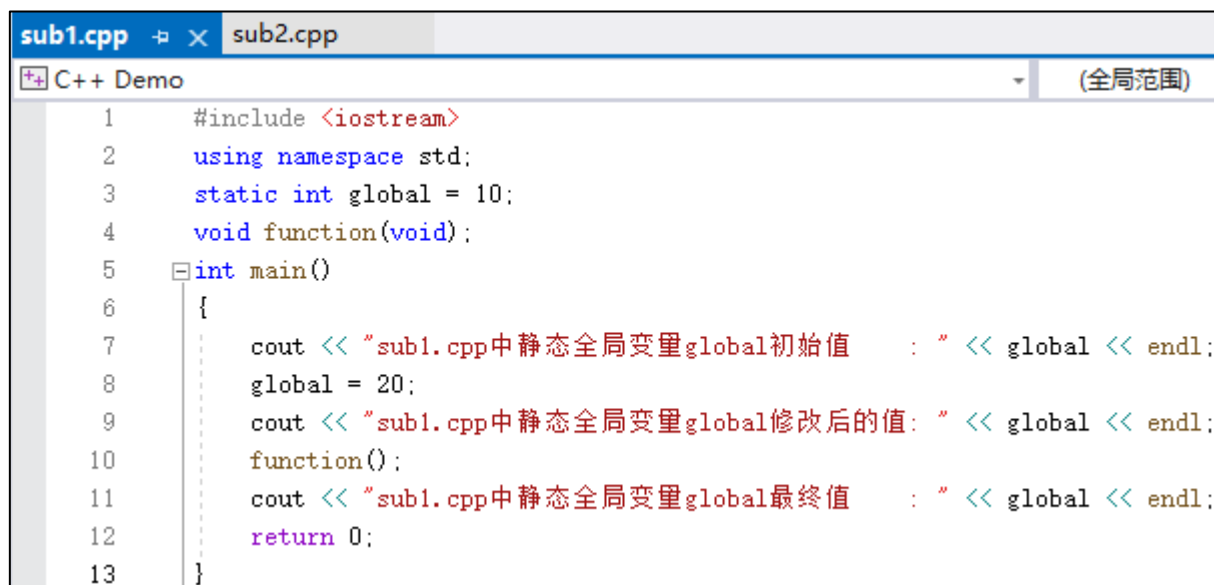
③结束调试。

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）

①在想要调试的函数内设置一个断点，并开始调试。

②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。此时可以查看两个源程序文件中同名的静态全局变量的名称、当前值和类型。

下面是一个由两个源文件组成，在每个源文件中有同名静态全局变量的示例程序。在示例程序的执行过程中，每一步的静态全局变量的变化过程如下：在sub1.cpp中，静态全局变量global被定义并初始化为10，在main函数中，静态全局变量global被赋值为20，在sub2.cpp中，静态全局变量global被定义并初始化为30，在function函数中，静态全局变量global被赋值为40，在function函数调用结束返回main函数后，静态全局变量global的值为20。



```

sub1.cpp  sub2.cpp  + x
C++ Demo (全局范围)
1  #include <iostream>
2  using namespace std;
3  static int global = 30;
4  void function()
5  {
6      cout << "sub2.cpp中静态全局变量global初始值    : " << global << endl;
7      global = 40;
8      cout << "sub2.cpp中静态全局变量global修改后的值: " << global << endl;
9  }
    
```

(上图涉及2.3相关内容)

示例程序的运行输出结果如下，可以说明静态全局变量只限本源程序文件的定义范围内使用，不同源程序文件中的静态全局变量允许同名。静态全局变量的作用域仅限于单个源文件，因此在不同的源文件中定义同名的静态全局变量会创建各自独立的实例。

```

Microsoft Visual Studio 调试控制台
sub1.cpp中静态全局变量global初始值    : 10
sub1.cpp中静态全局变量global修改后的值: 20
sub2.cpp中静态全局变量global初始值    : 30
sub2.cpp中静态全局变量global修改后的值: 40
sub1.cpp中静态全局变量global最终值    : 20
    
```

(上图涉及2.3相关内容)

下面是调试模式下在“监视”窗口中查看在示例程序执行过程中两个源程序文件中同名的静态全局变量的变化情况。

在 sub1.cpp 中，静态全局变量 global 被定义并初始化为 10

```

sub1.cpp  sub2.cpp
C++ Demo
1  #include <iostream>
2  using namespace std;
3
4  static int global = 10;
5
6  void function(void);
7
8  int main()
9  {
10     cout << "sub1.cpp中静态全局变量global初始值    : " << global << endl;
11     global = 20;
12     cout << "sub1.cpp中静态全局变量global修改后的值: " << global << endl;
13     function();
14     cout << "sub1.cpp中静态全局变量global最终值    : " << global << endl;
15     return 0;
16 }
    
```

监视 1

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
global	10	int

添加要监视的项

在 main 函数中，静态全局变量 global 被赋值为 20

```

1  #include <iostream>
2  using namespace std;
3
4  static int global = 10;
5
6  void function(void);
7
8  int main()
9  {
10     cout << "sub1.cpp中静态全局变量global初始值    : " << global << endl;
11     global = 20;
12     cout << "sub1.cpp中静态全局变量global修改后的值: " << global << endl;
13     function();
14     cout << "sub1.cpp中静态全局变量global最终值    : " << global << endl;
15     return 0;
16 }
    
```

名称	值	类型
global	20	int

在 sub2.cpp 中，静态全局变量 global 被定义并初始化为 30

```

1  #include <iostream>
2  using namespace std;
3
4  static int global = 30;
5
6  void function()
7  {
8     cout << "sub2.cpp中静态全局变量global初始值    : " << global << endl;
9     global = 40;
10     cout << "sub2.cpp中静态全局变量global修改后的值: " << global << endl;
11 }
    
```

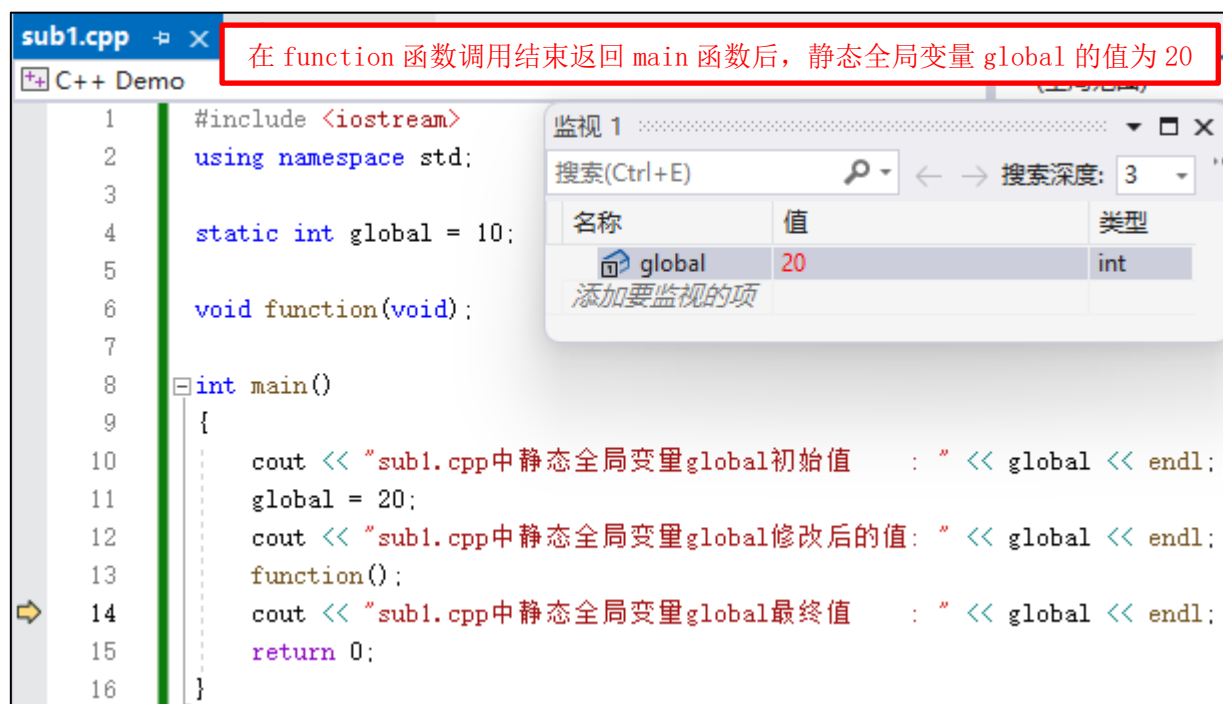
名称	值	类型
global	30	int

在 function 函数中，静态全局变量 global 被赋值为 40

```

1  #include <iostream>
2  using namespace std;
3
4  static int global = 30;
5
6  void function()
7  {
8     cout << "sub2.cpp中静态全局变量global初始值    : " << global << endl;
9     global = 40;
10     cout << "sub2.cpp中静态全局变量global修改后的值: " << global << endl;
11 }
    
```

名称	值	类型
global	40	int



(上图涉及2.3相关内容)

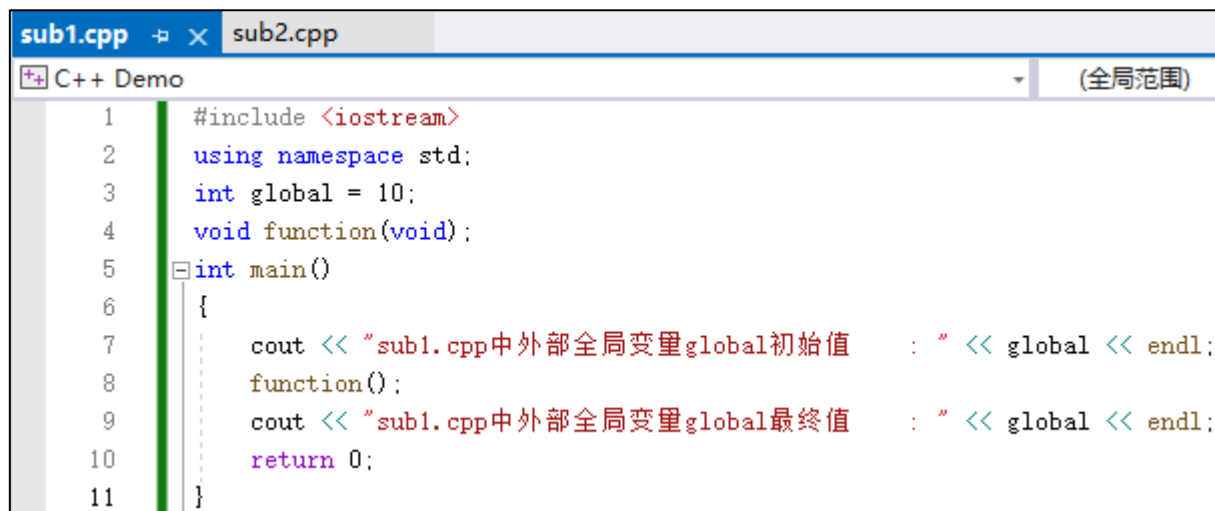
③结束调试。

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）

①在想要调试的函数内设置一个断点，并开始调试。

②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。此时可以查看外部全局变量的名称、当前值和类型。

下面是一个由两个源文件组成的示例程序。在示例程序的执行过程中，每一步的外部全局变量的变化过程如下：在sub1.cpp中，外部全局变量global被定义并初始化为10，在sub2.cpp的function函数中，外部全局变量global被赋值为20，在function函数调用结束返回main函数后，外部全局变量global的值为20。



```

1  #include <iostream>
2  using namespace std;
3  extern int global;
4  void function()
5  {
6      global = 20;
7      cout << "sub2.cpp中外部全局变量global修改后的值: " << global << endl;
8  }
    
```

(上图涉及2.4相关内容)

示例程序的运行输出结果如下，可以说明外部全局变量在所有源程序文件中的函数均可使用（其他源程序文件中加extern说明），对外部全局变量的修改会影响到不同源程序文件。

```

Microsoft Visual Studio 调试控制台
sub1.cpp中外部全局变量global初始值      : 10
sub2.cpp中外部全局变量global修改后的值    : 20
sub1.cpp中外部全局变量global最终值        : 20
    
```

(上图涉及2.4相关内容)

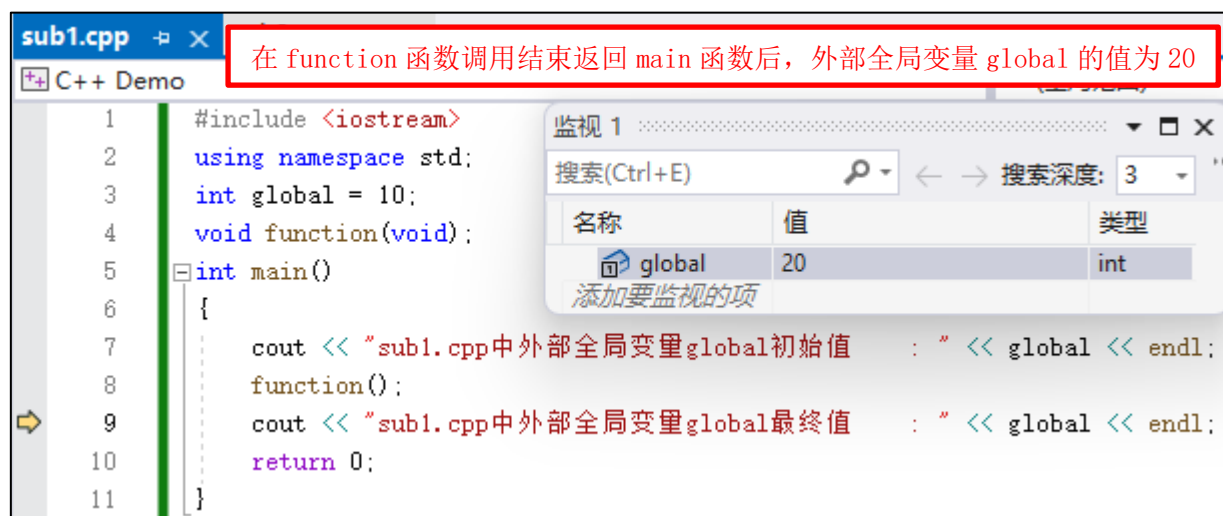
下面是调试模式下在“监视”窗口中查看在示例程序执行过程中外部全局变量的变化情况。

在 sub1.cpp 中，外部全局变量 global 被定义并初始化为 10

名称	值	类型
global	10	int

在 sub2.cpp 的 function 函数中，外部全局变量 global 被赋值为 20

名称	值	类型
global	20	int



(上图涉及2.4相关内容)

③结束调试。

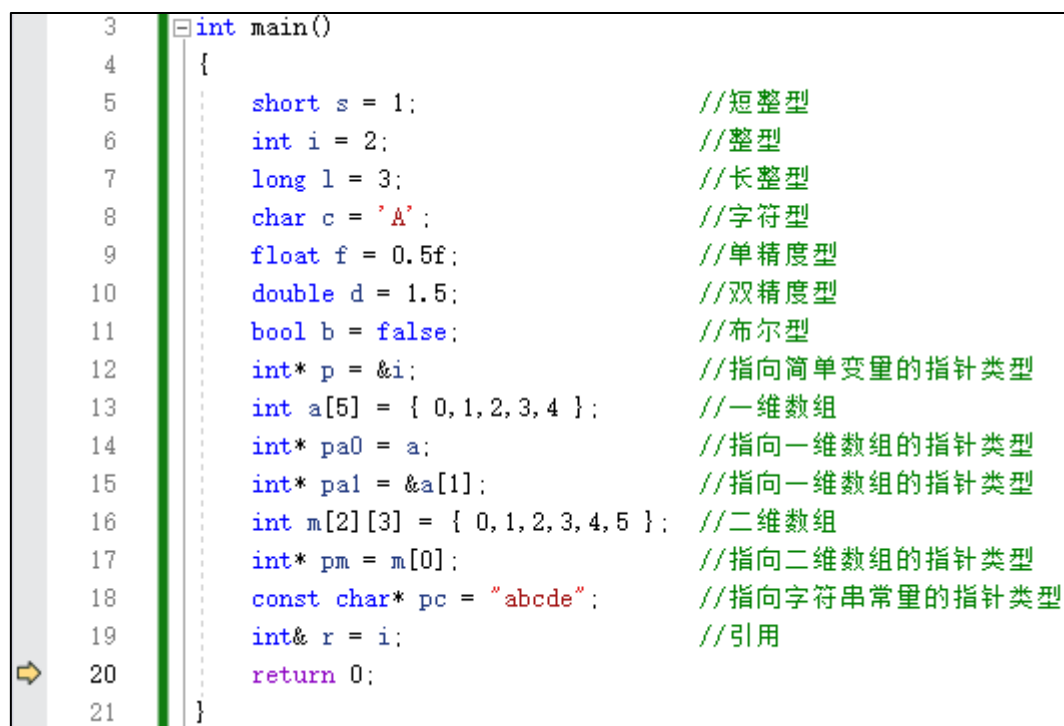
3. 用VS2022的调试工具查看各种不同类型变量的方法

①在想要查看变量值的代码行设置一个断点并开始调试。

②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。此时可以在“自动窗口”窗口、“局部变量”窗口和“监视”窗口查看不同类型变量的名称、当前值和类型，也可以将鼠标悬停在代码编辑器中的变量名上，以查看当前变量的值。

③结束调试。

下面是用VS2022的调试工具查看各种不同类型变量的示例程序。



(上图涉及3.1/3.2/3.3/3.4/3.5/3.7/3.8相关内容)

下面是“局部变量”窗口中的显示内容。

对于数组类型变量，可以单击数组类型变量名称左边的“>”按钮展开列表，来查看数组中每一个元素变量的当前值和类型。

局部变量		
搜索(Ctrl+E) 搜索深度: 3		
名称	值	类型
▲ a	0x012ff730 {0, 1, 2, 3, 4}	int[5]
[0]	0	int
[1]	1	int
[2]	2	int
[3]	3	int
[4]	4	int
b	false	bool
c	65 'A'	char
d	1.5000000000000000	double
f	0.5000000000	float
i	2	int
l	3	long
▲ m	0x012ff6f8 {0x012ff6f8 {0, 1, 2}, 0x012ff704 {3, 4, 5}}	int[2][3]
[0]	0x012ff6f8 {0, 1, 2}	int[3]
[0]	0	int
[1]	1	int
[2]	2	int
[1]	0x012ff704 {3, 4, 5}	int[3]
[0]	3	int
[1]	4	int
[2]	5	int
p	0x012ff798 {2}	int *
pa0	0x012ff730 {0}	int *
pa1	0x012ff734 {1}	int *
pc	0x006a7b30 "abcde"	const char *
pm	0x012ff6f8 {0}	int *
r	2	int &
s	1	short

(上图涉及3.1/3.2/3.3/3.4/3.5/3.7/3.8相关内容)

下面是“监视”窗口中的显示内容。

通过在“监视”窗口中，填写变量的取地址表达式“&[变量名]”，然后按Enter键，可以在调试模式下查看变量的地址。

对于“&[数组名]”，可以单击数组名左边的“>”按钮展开列表，来查看数组首地址和数组中每一个元素变量的当前值和类型。

监视 1		
搜索(Ctrl+E) 搜索深度: 3		
名称	值	类型
&ls	0x012ff7a4 {1}	short *
&i	0x012ff798 {2}	int *
&l	0x012ff78c {3}	long *
&c	0x012ff783 "A烫烫烫\t3"	char *
&f	0x012ff774 {0.5000000000}	float *
&d	0x012ff764 {1.5000000000000000}	double *
&b	0x012ff75b {false}	bool *
&p	0x012ff74c {0x012ff798 {2}}	int **
&a	0x012ff730 {0, 1, 2, 3, 4}	int[5] *
[0]	0	int
[1]	1	int
[2]	2	int
[3]	3	int
[4]	4	int
&pa0	0x012ff724 {0x012ff730 {0}}	int **
&pa1	0x012ff718 {0x012ff734 {1}}	int **
&m	0x012ff6f8 {0x012ff6f8 {0, 1, 2}, 0x012ff704 {3, 4, 5}}	int[2][3] *
[0]	0x012ff6f8 {0, 1, 2}	int[3]
[0]	0	int
[1]	1	int
[2]	2	int
[1]	0x012ff704 {3, 4, 5}	int[3]
[0]	3	int
[1]	4	int
[2]	5	int
&pm	0x012ff6ec {0x012ff6f8 {0}}	int **
&pc	0x012ff6e0 {0x006a7b30 "abcde"}	const char **
&r	0x012ff798 {2}	int *
添加要监视的项		

(上图涉及3. 1/3. 2/3. 3/3. 4/3. 5/3. 7/3. 8相关内容)

3. 1. short/int/long/char/float/double/bool等简单变量

如示例程序和“局部变量”窗口中的显示内容所示，简单变量的名称在“名称”栏中显示，当前值在“值”栏中显示，类型在“类型”栏中显示。

3. 2. 指向简单变量的指针变量

如示例程序和“局部变量”窗口中的显示内容所示，指向简单变量的指针变量的名称在“名称”栏中显示，指针变量所指向的值的地址和当前值在“值”栏中显示（当前值显示在花括号“{ }”中，地址显示在花括号“{ }”前），类型在“类型”栏中显示。

3. 3. 一维数组

如示例程序和“局部变量”窗口中的显示内容所示，一维数组的名称在“名称”栏中显示，数组首地址和数组元素的当前值在“值”栏中显示（数组元素的当前值显示在花括号“{ }”中，用逗号分隔，数组首地址显示在花括号“{ }”前），类型在“类型”栏中显示。

3.4. 指向一维数组的指针变量

如示例程序和“局部变量”窗口中的显示内容所示，指向一维数组的指针变量的名称在“名称”栏中显示，指针变量所指向的一维数组元素的值的地址和当前值在“值”栏中显示（一维数组元素的当前值显示在花括号“{ }”中，一维数组元素的地址显示在花括号“{ }”前），类型在“类型”栏中显示。

3.5. 二维数组

如示例程序和“局部变量”窗口中的显示内容所示，二维数组的名称在“名称”栏中显示，二维数组首地址和数组元素的当前值在“值”栏中显示（二维数组首地址显示在外层花括号“{ }”前，一维数组首地址显示在外层花括号“{ }”中，用逗号分隔，一维数组元素的当前值显示在一维数组首地址后的内层花括号“{ }”中，用逗号分隔），类型在“类型”栏中显示。

二维数组名仅带一个下标是代表组成二维数组的一维数组首地址，和指向一维数组的指针变量表示相同，指针变量的名称在“名称”栏中显示，组成二维数组的一维数组首地址和指针变量所指元素的当前值在“值”栏中显示（指针变量所指元素的当前值显示在花括号“{ }”中，组成二维数组的一维数组首地址显示在花括号“{ }”前），类型在“类型”栏中显示。

3.6. 实参是一维数组名，形参是指针时，在函数中查看实参数组的地址与值

下面是一个实参是一维数组名，形参是指针时，在函数中查看实参数组的地址与值的示例程序。在示例程序的执行过程中，可以在function函数查看实参数组的地址与值。

```

1      #include <iostream>
2      using namespace std;
3      void function(int* p)
4      {
5          p++;
6          p++;
7      }
8      int main()
9      {
10         int a[3] = { 0,1,2 };
11         function(a);
12         return 0;
13     }
    
```

（上图涉及3.6相关内容）

下面是调试模式下在“局部变量”窗口中查看在示例程序执行过程中实参数组的地址与值的变化情况。

```

3 void function(int* p)
4 {
5     p++;
6     p++;
7 }
8 int main()
9 {
10     int a[3] = { 0, 1, 2 };
11     function(a);
12     return 0;
13 }
    
```

局部变量

名称	值	类型
a	0x0098f998 {0, 1, 2}	int[3]

实参是一维数组名，在局部变量窗口中实参数组的地址与值

```

3 void function(int* p)
4 {
5     p++; 已用时间 <= 13ms
6     p++;
7 }
8 int main()
9 {
10     int a[3] = { 0, 1, 2 };
11     function(a);
12     return 0;
13 }
    
```

局部变量

名称	值	类型
p	0x0098f998 {0}	int *

形参是指针，在局部变量窗口中实参数组的地址与值

```

3 void function(int* p)
4 {
5     p++;
6     p++; 已用时间 <= 4ms
7 }
8 int main()
9 {
10     int a[3] = { 0, 1, 2 };
11     function(a);
12     return 0;
13 }
    
```

局部变量

名称	值	类型
p	0x0098f99c {1}	int *

指针变量 p 自增，在局部变量窗口中实参数组的地址与值

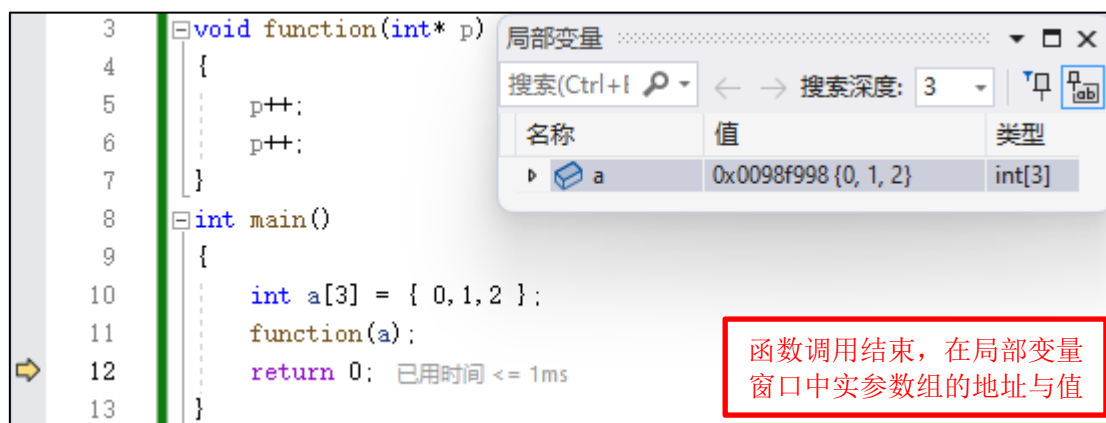
```

3 void function(int* p)
4 {
5     p++;
6     p++;
7 } 已用时间 <= 4ms
8 int main()
9 {
10     int a[3] = { 0, 1, 2 };
11     function(a);
12     return 0;
13 }
    
```

局部变量

名称	值	类型
p	0x0098f9a0 {2}	int *

指针变量 p 自增，在局部变量窗口中实参数组的地址与值



(上图涉及3.6相关内容)

3.7. 指向字符串常量的指针变量

如示例程序和“局部变量”窗口中的显示内容所示，指向字符串常量的指针变量的名称在“名称”栏中显示，指针变量所指向的值的地址和当前值在“值”栏中显示（能看到无名字符串常量的地址），类型在“类型”栏中显示。

3.8. 引用

如示例程序和“局部变量”窗口中的显示内容所示，引用的名称在“名称”栏中显示，当前值在“值”栏中显示，类型在“类型”栏中显示。

引用和指针有如下区别：

①引用和指针在定义和语法上的区别：

引用是C++中的一种变量别名，用于引用现有变量。引用必须在定义时初始化，并且一旦初始化后，就无法更改引用的目标。引用使用&符号进行声明和初始化；指针是一个变量，用于存储另一个变量的地址。指针的值可以在任何时候修改为指向其他变量或空地址。指针使用*符号进行声明和初始化。

②引用和指针在内存管理上的区别：

引用在声明时不会分配新的内存空间，它只是现有变量的别名。因此，引用不能为空，并且必须在声明时初始化为有效的变量；指针在声明时会分配内存空间来存储变量的地址。指针可以为空，也可以指向有效的变量。

③引用和指针在用法和语义上的区别：

引用提供了一种简洁的方式来操作变量，而不需要使用指针运算符。它可以像普通变量一样使用，不需要额外的解引用操作；指针提供了更多的灵活性，可以进行指针算术运算，可以在运行时动态地指向不同的对象。指针需要使用解引用操作符*来访问所指向的变量。

④引用和指针在重新赋值上的区别：

引用在初始化后，不能重新赋值为引用其他的变量。一旦引用初始化完成，它将一直引用同一个对象；指针可以通过重新赋值来指向不同的变量或空地址。可以将指针重新赋值为其他变量的地址。

⑤引用和指针在参数传递上的区别：

引用可以用作函数的参数，通过引用传递可以修改函数外部的变量。引用传递用于函数参数的传递具有高效性和便捷性；指针也可以用作函数的参数，通过指针传递可以修改函数外部的变量。指针传递通常用于需要在函数内部修改指针指向的变量或者传递指针数组等场景。

引用和指针在某些情况下可以互相替代，但它们的语义和使用方式有一些不同。在选择使用引用还是指针时，需要根据具体的需求和编程场景来进行判断。

3.9. 使用指针时出现越界访问

下面是一个使用指针时出现越界访问的示例程序。在示例程序的执行过程中，每一步的指针变量的变化过程如下：在main函数中，定义int型指针变量p，并将其初始化为指向int型一维数组a中第三个元素a[2]的地址；执行p++操作，int型指针变量p向后移动一个整数大小的偏移量，由于p指向数组a中的最后一个元素，所以这里出现越界访问，会导致指针变量p指向未定义的内存位置。

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[3] = { 0, 1, 2 };
6      int* p = &a[2];
7      p++;
8      return 0;
9  }
```

(上图涉及3.9相关内容)

下面是调试模式下在“局部变量”窗口中查看在示例程序执行过程中指针变量的变化情况与使用指针时出现越界访问的情况。

在main函数中，定义int型指针变量p，并将其初始化为指向int型一维数组a中第三个元素a[2]的地址

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[3] = { 0, 1, 2 };
6      int* p = &a[2];
7      p++;
8      return 0; 已用时间 <= 3m
9  }
```

局部变量

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	0x006ffce8 {0, 1, 2}	int[3]
[0]	0	int
[1]	1	int
[2]	2	int
p	0x006ffcf4 {-858993460}	int *
	-858993460	int

执行p++操作，int型指针变量p向后移动一个整数大小的偏移量，由于p指向数组a中的最后一个元素，所以这里出现越界访问，会导致指针变量p指向未定义的内存位置

(上图涉及3.9相关内容)