

报告名称:

# VS2019调试工具的使用

高程1班

信09

1953729

吴浩泽

完成日期: 2020年12月13日

◆

◆

装

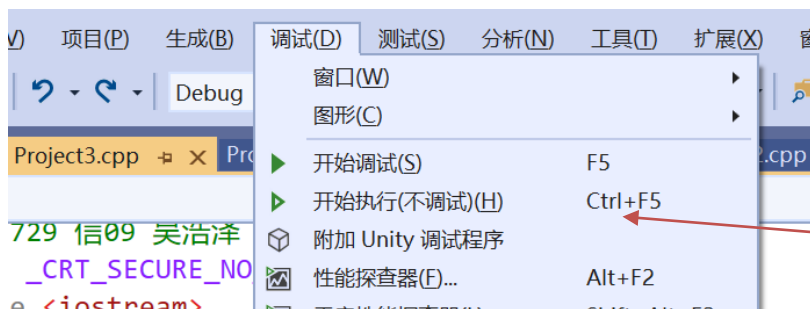
订

线

## 1. VS 2019下调试工具的基本使用方法

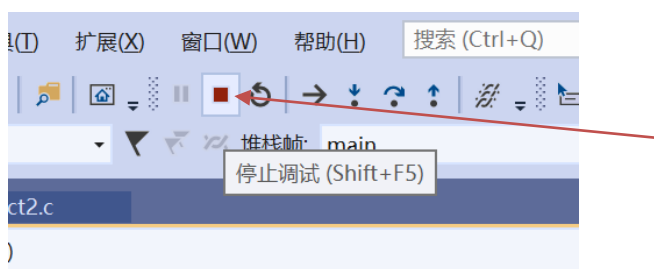


### 1.1 如何开始调试？如何结束调试？



在如图所示点击调试，或者按快捷键F5开始调试

如果不设置断点，程序调试之后就正常弹出exe, 正常结束，如果程序设置断点，则程序一直执行到断点所在位置停止，然后可以选择调试步骤，这里说一下结束的标志

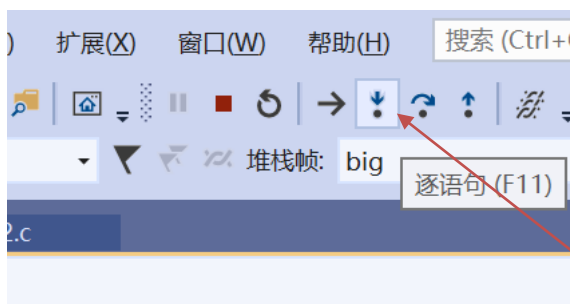


结束调试可以点如图所示按钮，也可以按快捷键Shift+F5，

### 1.2 如何在一个函数中每个语句单步执行？

```

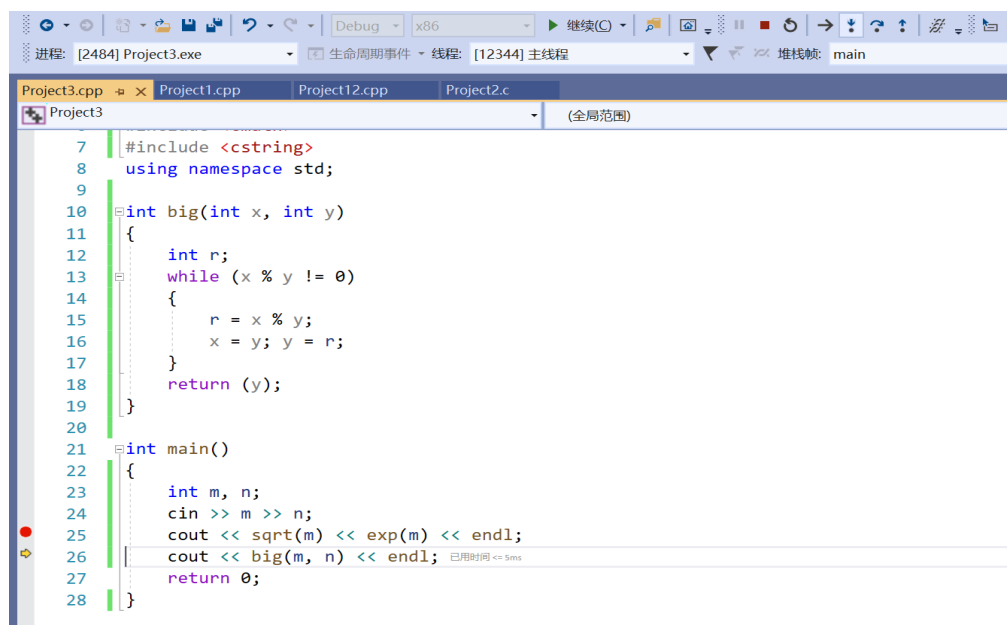
8
9 int big(int x, int y)
10 {
11     int r;
12     while (x % y != 0) 已用时间 <= 2ms
13     {
14         r = x % y;
15         x = y; y = r;
16     }
17     return (y);
18 }
19
20 int main()
21 {
22     int m, n;
23     cin >> m >> n;
24     cout << big(m, n) << endl;
25     return 0;
26 }
    
```



可以点如图所示按钮，也可以按快捷键F11, 就会每个语句单步执行

## 1.3 在碰到 cout/sqrt 等系统类/系统函数时，如何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？

这里说明一下：经过我的测试，我发现我是进不去系统类或者系统函数的，不管我是点单步执行F11还是逐过程，也就是说平常的sqrt, cout, 程序是不自己进入到系统函数里面的。



```

7 | #include <cstring>
8 | using namespace std;
9 |
10 | int big(int x, int y)
11 | {
12 |     int r;
13 |     while (x % y != 0)
14 |     {
15 |         r = x % y;
16 |         x = y; y = r;
17 |     }
18 |     return (y);
19 | }
20 |
21 | int main()
22 | {
23 |     int m, n;
24 |     cin >> m >> n;
25 |     cout << sqrt(m) << exp(m) << endl;
26 |     cout << big(m, n) << endl;
27 |     return 0;
28 | }
    
```

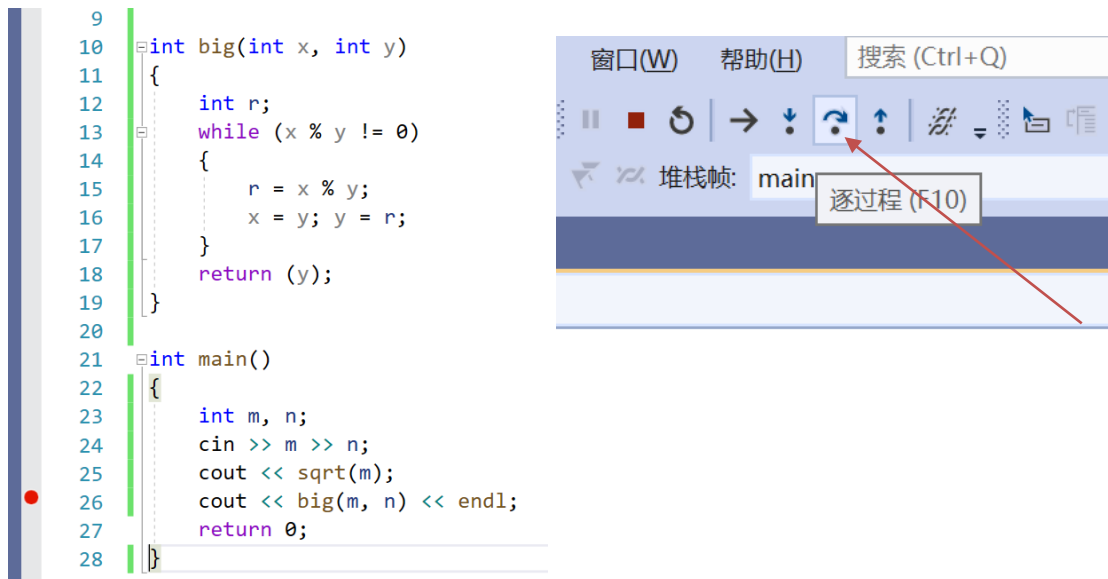
这里在第25行执行之后，无论是点逐语句，还是逐过程，程序都是一步到下面26行，不进入到系统函数中去的。

- ❖
- ❖ 1.4 如果已经进入到cout/sqrt 等系统类/系统函数的内部，如何跳出并返回自己的函数？



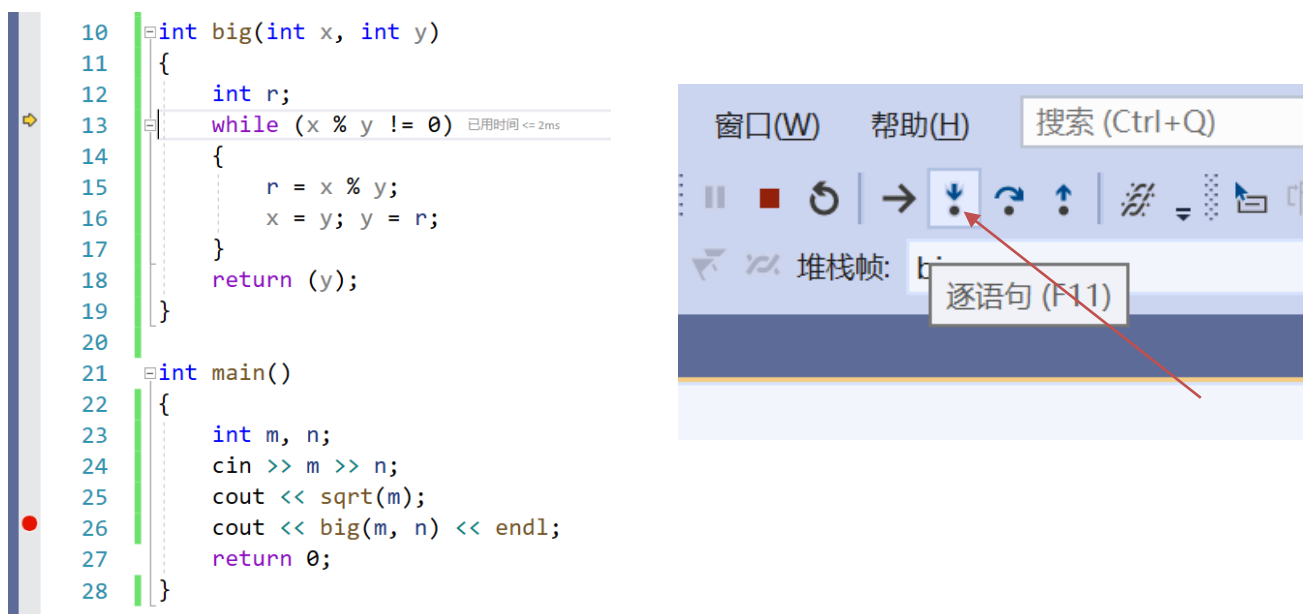
如果已经进入函数内部，想直接跳出函数，可以点如图所示按钮，或者按Shift+F11快捷键就可以跳出该函数，转到主函数

1.5 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，如何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？



可以点如图所示的逐过程，或者按F10快捷键，就可以直接执行，不进入对应的函数。

1.6 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，如何转到被调用函数中单步执行？



可以点如图所示的逐语句，或者按F11快捷键，就可以进入到对应函数执行。

## 2. 用VS 2019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



这里在调试的过程中点窗口下的局部变量，以及自动窗口，就会在下面显示出对应的界面显示出当即对应的形参或者自动变量的值，这些变量的值会随着程序的进行而随时改变



## 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）

```

9
10 int big(int x, int y)
11 {
12     int r;
13     static int a = 1;
14     while (x % y != 0)
15     {
16         r = x % y;
17         x = y; y = r; 已用时间 <= 1ms
18     }
19     a++;
20     return (y);
21 }
22
23 int main()
24 {
25     int m, n;
26     cin >> m >> n;
27     cout << big(m, n) << endl;
28     cout << big(n, m) << endl;
29     return 0;
30 }

```

名称	值	类型
a	1	int
r	4	int
x	4	int
y	6	int

```

10 int big(int x, int y)
11 {
12     int r;
13     static int a = 1;
14     while (x % y != 0)
15     {
16         r = x % y;
17         x = y; y = r;
18     }
19     a++;
20     return (y); 已用时间 <= 1ms
21 }
22
23 int main()
24 {
25     int m, n;
26     cin >> m >> n;
27     cout << big(m, n) << endl;
28     cout << big(n, m) << endl;
29     return 0;
30 }

```

名称	值	类型
a	3	int
r	2	int
x	4	int
y	2	int

可以看到静态局部变量如果是函数内的则只会在经过函数时显示出来，而其值保留上次的值。

## 2.3 查看静态全局变量的变化情况（两个源程序文件，有静态全局变量同名）

```

9 using namespace std;
10 static int a = 1;
11 int big(int, int);
12 int max1(int, int);
13 int main()
14 {
15     int m, n;

```

```

9
10 int max1(int x, int y)
11 {
12     int a = 1;
13     return x > y ? x : y; 已用时间 <= 1ms
14 }

```

名称	值	类型
a	1	int
x	6	int
y	4	int

可以看到静态全局变量和局部变量是有符号上的差别的，那么自动变量是不影响静态全局变量的值的。

## 2.4 查看外部全局变量的变化情况（两个源程序文件，一个定义，另一个有extern 说明）

```

8 using namespace std;
9 int a = 1;
10 int big(int, int);
11 int max1(int, int);
12 int main()
13 {
14     int m, n;
15     cin >> m >> n;
16     cout << big(m, n) << endl;
17     cout << max1(n, m) << endl;
18     cout << big(n, m) << endl;
19     return 0;
20 }

```

```

9 extern int a;
10
11 int max1(int x, int y)
12 {
13     a = 2;
14     return x > y ? x : y;
15 }

```

在extern引用了a之后a的值会变成第二个cpp里定义的值，外部全局变量的值发生了改变。

## 3. 掌握用VS 2019的调试工具查看各种不同类型变量的方法

### 3.1 char/int/float 等简单变量

```

12 int main()
13 {
14     int m, n;
15     char k, h;
16     float i, j;
17     int* p1;
18     char* p2;
19     float* p3;
20     cin >> m >> n;
21     k = 'A';
22     i = 3.14;
23     cout << m << k << i;

```

自动窗口		
搜索(Ctrl+E)		
搜索深度: 3		
名称	值	类型
i	3.14000010	float
k	65 'A'	char
m	2	int
n	3	int

几个简单变量的值可以如图所示查看。

## 3.2 指向简单变量的指针变量（如何查看地址、值？）

```
int m, n;
char k, h;
float i, j;
int* p1;
char* p2;
float* p3;
cin >> m >> n;
k = 'A';
i = 3.14;
p1 = &m;
p2 = &k;
p3 = &i;
cout << *p1 << *p2 << *p3;
```

▸ &i	0x00bffa8 {3.14000010}	float *
▸ &k	0x00bffc3 "A烫烫烫烫\x3"	char *
▸ &m	0x00bffd8 {2}	int *
▸ *p1	2	int
▸ *p2	65 'A'	char
▸ *p3	3.14000010	float
▸ p1	0x00bffd8 {2}	int *
▸ p2	0x00bffc3 "A烫烫烫烫\x3"	char *
▸ p3	0x00bffa8 {3.14000010}	float *

在中间这一栏里，前面是地址，后面是对应的值，都可以看到。

## 3.3 一维数组

```
2 int main()
3 {
4     int a[100];
5
6     int* p1;
7     char* p2;
8     float* p3;
9     for (int i = 0; i < 3; i++)
0         cin >> a[i];
1     cout << a[0] << a[1] << a[2];
2     return 0;
```

▸ a	0x00bcfa24 {4, 5, 6, -858993460, -858993460, -858993460, ...}	int[100]
▸ a[0]	4	int
▸ a[1]	5	
▸ a[2]	6	0x00bcfa24 {4, 5, 6, -858993460, -858993460, -858993460, ...}



## 3.4 指向一维数组的指针变量（如何查看地址、值？）

```
int main()
{
    int a[100] = {};
    int* p1;
    char* p2;
    float* p3;
    for (int i = 0; i < 3; i++)
        cin >> a[i];
    p1 = a;
    for (; p1 != 0; p1++)
        cout << *p1 << ' ';
    return 0;
}
```

*p1	4	int
a	0x006ffac8 {4, 5, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...}	int[100]
p1	0x006ffac8 {4}	int *
*p1	5	int
a	0x006ffac8 {4, 5, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...}	int[100]
p1	0x006ffacc {5}	int *
*p1	0	int
p1	0x00eff9b0 {0}	int *

## 3.5 二维数组（包括数组名仅带一个下标的情况）

```
int main()
{
    int a[5][3] = {};
    int* p1;
    for (int i = 0; i < 3; i++)
        cin >> a[0][i];
    cout << a[0] << endl;
    cout << a[0][1]; 已用时间 <= 4ms
    return 0;
}
```

a	0x0054fba0 {0x0054fba0 {4, 5, 6}, 0x0054fbac {0, 0, 0}, 0x005...	int[5][3]
a[0]	0x0054fba0 {4, 5, 6}	int[3]
a[0][1]	5	int

仅带一个下标就是该行的首元素的地址，该行的首地址。

3.6 实参是一维数组名，形参是指针的情况，如何在函数中查看实参数组的地址、值？

The screenshot shows a C++ IDE with the following code:

```

9 void print(int*, int n);
10 int main()
11 {
12     int a[5] = { 0,1,2,3 };
13     print(a, 3);
14     return 0;
15 }
16
17
18 void print(int* p, int n)
19 {
20     for (int i = 0; i < n; i++)
21     {
22         cout << *p;
23         p++;
24     }
25 }

```

Below the code, the 'Stack' window is open, showing the following stack frames:

- Project1.exe!print(int \* p, int n) 行 22
- Project1.exe!main() 行 13
- [外部代码]
- kernel32.dll! [下面的框架可能不正确和/或缺失，没有为 kernel32.dll 加载符号]

这里可以在堆栈帧进行调整，通过选择main()就可以看到a的地址和值，而正常进入到函数里，看函数的变量的值就选对应的函数

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
a	0x00affaa4 {0, 1, 2, 3, 0}	int[5]

自动窗口下的变量由i, n, \*p变成了a。

## 3.7 指向字符串常量的指针变量（能否看到无名字符串常量的地址？）

```

10 int main()
11 {
12     const char* p = "China";
13     while (*p != '\0')
14     {
15         cout << *p;
16         p++;
17     }
18     cout << endl;
19     p = "China";
20     cout << p;
21     return 0;

```

89 % 未找到相关问题

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
*p	67 'C'	const char
p	0x00f7abcc "China"	const char *

这里可以看到p的值即为China的首地址，然后p会移动，p的值也就是地址会改变。

## 3.8 引用（引用与指针是否有区别？有什么区别？）

首先先说一下，引用和指针是有区别的，区别的本质在于指针还是在传值，只不过传的值是地址，而引用对函数传的是址，直接传的实参变量的地址。如果说指针传的地址，那么是一个实参指针指向了实参的地址，给函数的是实参指针的值，而实参指针间接反应的是实参变量的值，但是引用直接简化了，它传的不在需要其他变量，而是直接传实参变量。

```

void swap(int* p1, int* p2)
{
    int t;
    t = *p1;
    *p1 = *p2;
    *p2 = t;
}

int main()
{
    int* p1, * p2;
    int i = 2, j = 3;
    p1 = &i;
    p2 = &j;
    cout << i << j;
    swap(p1, p2);
    cout << i << j;
    return 0; 已用时间 <= 4ms
}

```

```

void swap(int& i, int& j)
{
    int t;
    t = i;
    i = j;
    j = t;
}

int main()
{
    int i = 2, j = 3;
    cout << i << j << endl;
    swap(i, j);
    cout << i << j;
    return 0;
}

```

▸ &i	0x00ffa90 {2}	int *
▸ &j	0x00ffa84 {3}	int *
▸ i	2	int
▸ j	3	int
▸ p1	0x00ffa90 {2}	int *
▸ p2	0x00ffa84 {3}	int *
▸ &j	0x00ffa84 {2}	int *
▸ i	3	int
▸ j	2	int
▸ p1	0x00ffa90 {3}	int *
▸ p2	0x00ffa84 {2}	int *

i	3	int &
j	3	int &
t	2	int

i	3	in
j	2	in

可以看到，对应的值确实是i, j的值，而不再是通过指针变量的地址形式改变实参的值，引用是直接改变。

### 3.9 使用指针时出现了越界访问

```

10 int main()
11 {
12     char a[6] = "China";
13     char* p = a;
14     while (*p != '\0')
15     {
16         cout << *p;
17         p++;
18     }
19     p = p + 10;
20     cout << *p;
21     return 0;
22 }

```

19 p = p + 10;  
20 cout << \*p; 已用时间 <= 3ms  
21 return 0;  
22 }

89 % 未找到相关问题

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
*p	41 'j'	char
p	0x008ffd53 "t龔"	char *