# VS2019调试工具使用报告

班级：信息类11班

学号：2151294

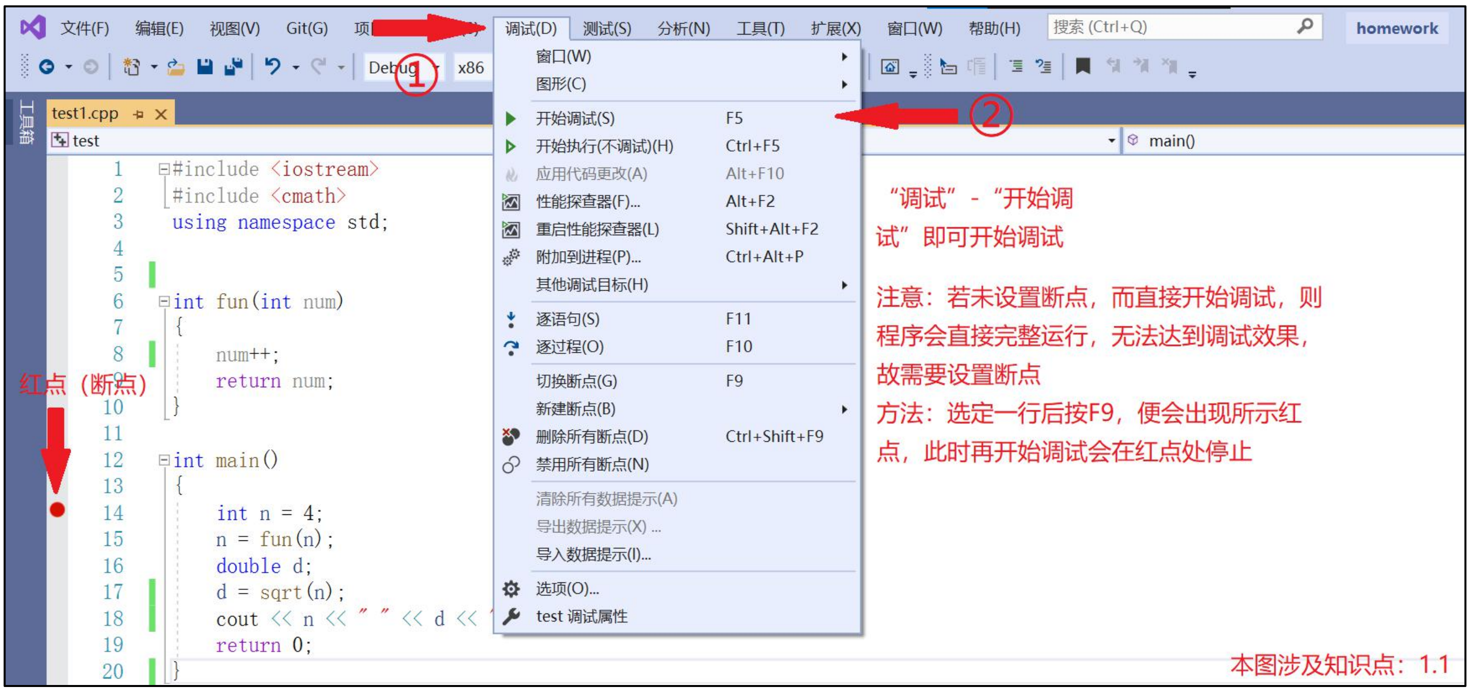
姓名：马威

完成日期：2021.12.30

## 1. 小程序1：

```
#include <iostream>
#include <cmath>
using namespace std;

int fun(int num)
{
    num++;
    return num;
}

int main()
{
    int n = 4;
    n = fun(n);
    double d;
    d = sqrt(n);
    cout << n << " " << d << " #" << endl;
    return 0;

}
```

### 1.1. 图 1



```
#include <iostream>
#include <cmath>
using namespace std;

int fun(int num)
{
    num++;
    return num;
}

int main()
{
    int n = 4;
    n = fun(n);
    double d;
    d = sqrt(n);
    cout << n << " " << d << "
    return 0;
}
```

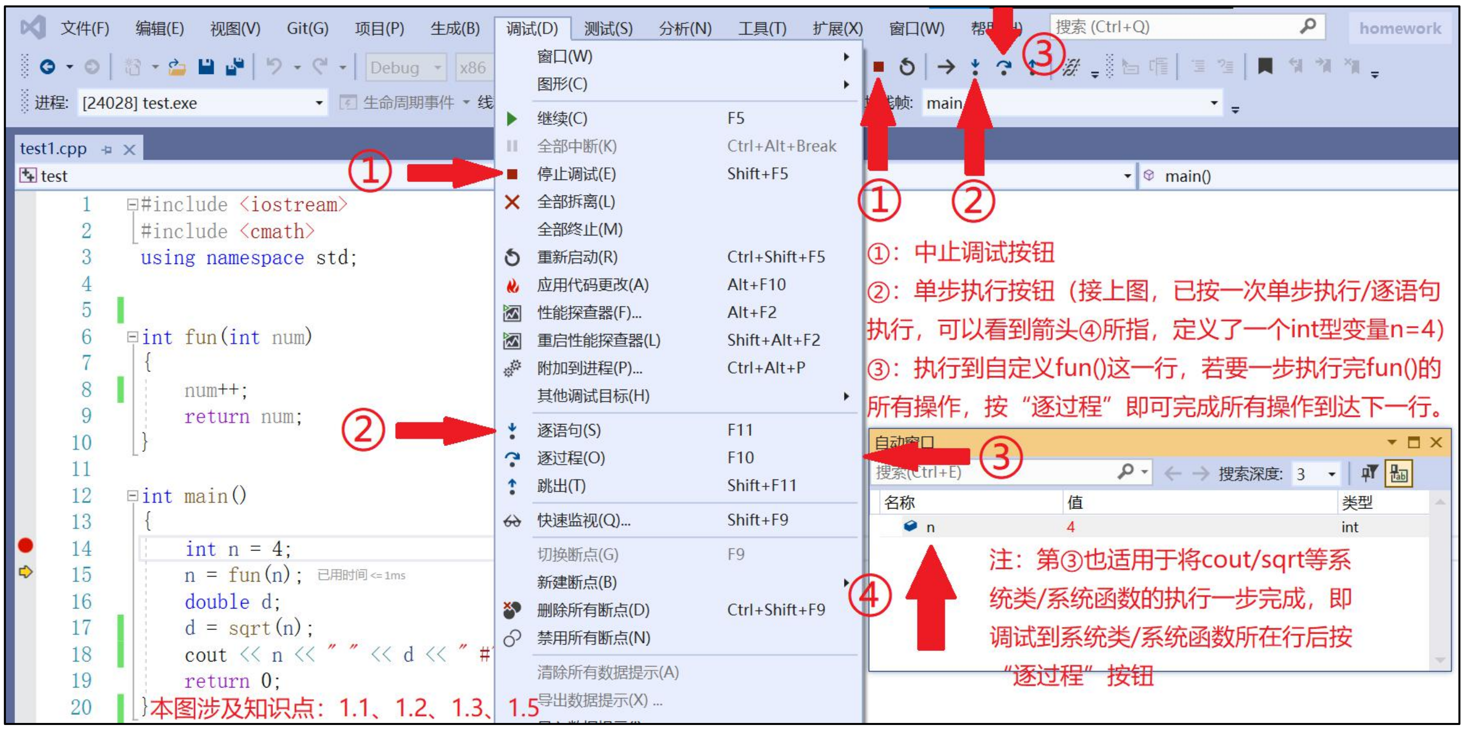
“调试” - “开始调试”即可开始调试

注意：若未设置断点，而直接开始调试，则程序会直接完整运行，无法达到调试效果，故需要设置断点

方法：选定一行后按F9，便会出现所示红点，此时再开始调试会在红点处停止

本图涉及知识点：1.1

## 1.2. 图 2



```
#include <iostream>
#include <cmath>
using namespace std;

int fun(int num)
{
    num++;
    return num;
}

int main()
{
    int n = 4;
    n = fun(n);
    double d;
    d = sqrt(n);
    cout << n << " " << d << " #
    return 0;
}
```

| 名称 | 值 | 类型 |
|---|---|---|
| n | 4 | int |

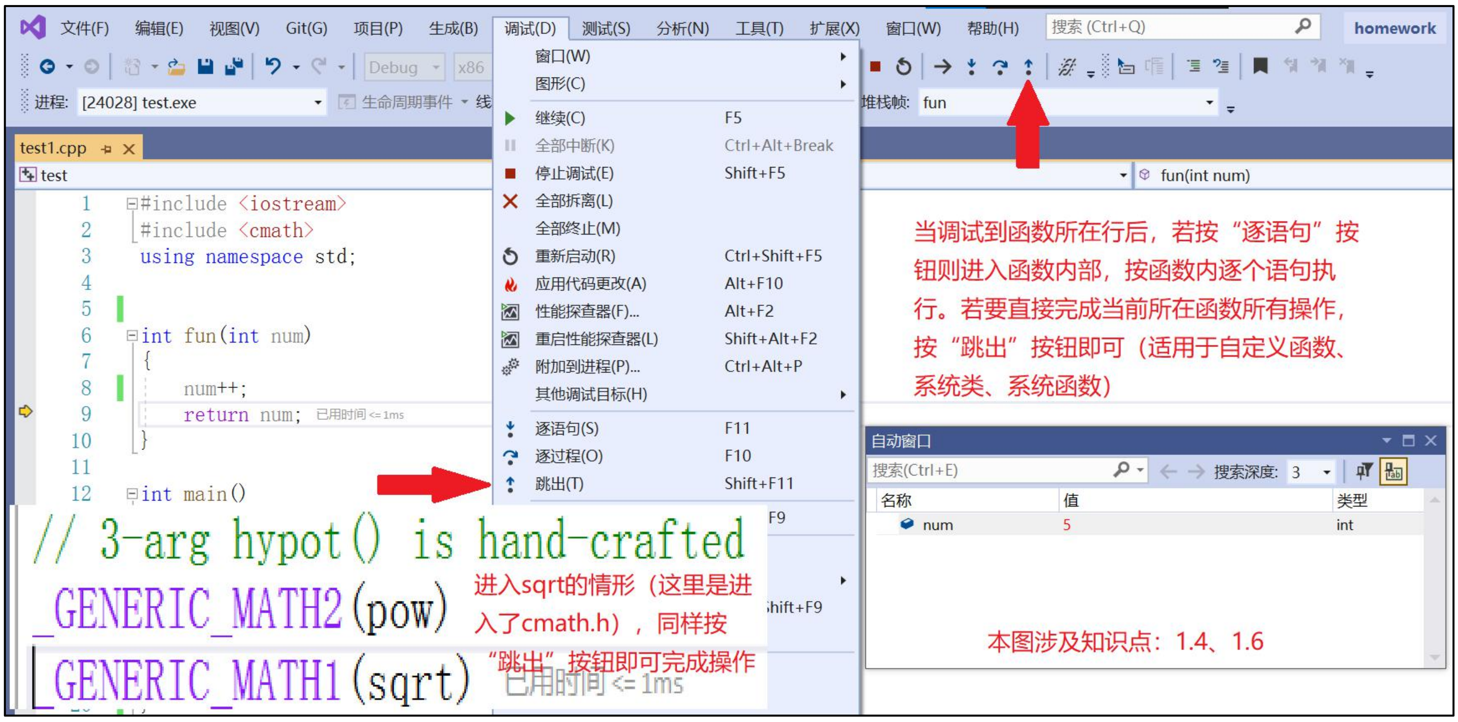
①：中止调试按钮
②：单步执行按钮（接上图，已按一次单步执行/逐语句执行，可以看到箭头④所指，定义了一个int型变量n=4）
③：执行到自定义fun()这一行，若要一步执行完fun()的所有操作，按“逐过程”即可完成所有操作到达下一行。

注：第③也适用于将cout/sqrt等系统类/系统函数的执行一步完成，即调试到系统类/系统函数所在行后按“逐过程”按钮

本图涉及知识点：1.1、1.2、1.3、1.5

ì

## 1.3. 图 3



```
#include <iostream>
#include <cmath>
using namespace std;

int fun(int num)
{
    num++;
    return num;  已用时间 <= 1ms
}

int main()
```

```
// 3-arg hypot() is hand-crafted
_GENERIC_MATH2(pow)
_GENERIC_MATH1(sqrt)  已用时间 <= 1ms
```

| 名称 | 值 | 类型 |
|---|---|---|
| num | 5 | int |

当调试到函数所在行后，若按“逐语句”按钮则进入函数内部，按函数内逐个语句执行。若要直接完成当前所在函数所有操作，按“跳出”按钮即可（适用于自定义函数、系统类、系统函数）

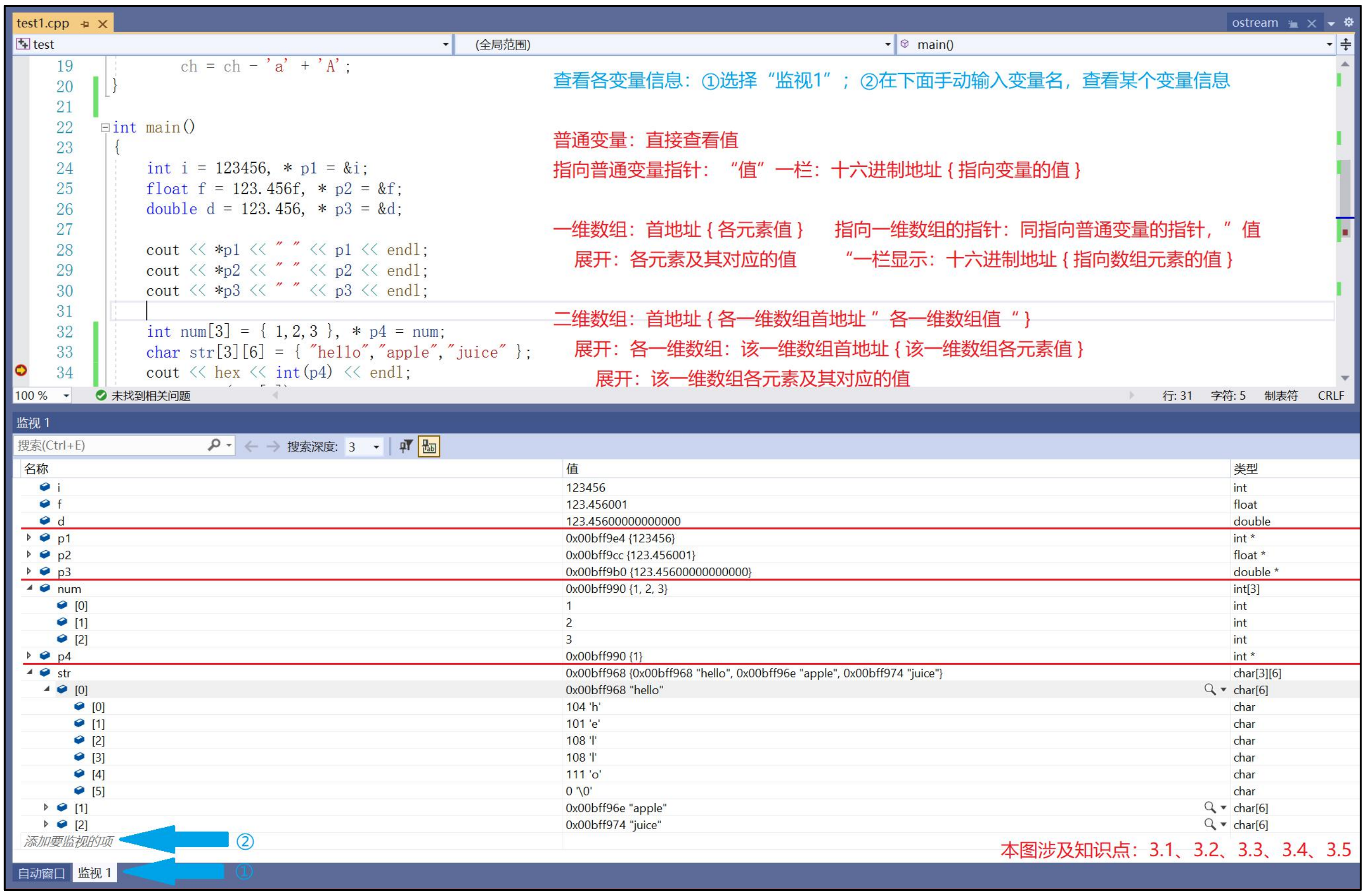
进入sqrt的情形（这里是进入了cmath.h），同样按“跳出”按钮即可完成操作

本图涉及知识点：1.4、1.6

## 2. 小程序 2

```cpp
#include <iostream>
using namespace std;

void fun(int* NUMBERS, char& ch)
{
    int i = 0;
    const char String[] = "12345!@#$%", * p5 = String;
    const char* p6 = "67890^&*()";

    if (ch >= 'a' && ch <= 'z')
        ch = ch - 'a' + 'A';

    int* p = NUMBERS;
    (*p)++;
    p++;
    (*p)++;
}

int main()
{
    int i = 123456, * p1 = &i;
    float f = 123.456f, * p2 = &f;
    double d = 123.456, * p3 = &d;

    cout << *p1 << " " << p1 << endl;
    cout << *p2 << " " << p2 << endl;
    cout << *p3 << " " << p3 << endl;

    int num[3] = { 1,2,3 }, * p4 = num;
    char str[3][6] = { "hello","apple","juice" };

    char c = 'f';
    fun(num, c);
    cout << c << endl;
    return 0;

}
```

## 2.1. 图 1



test1.cpp | test | (全局范围) | main()

```cpp
        ch = ch - 'a' + 'A';
}

int main()
{
    int i = 123456, * p1 = &i;
    float f = 123.456f, * p2 = &f;
    double d = 123.456, * p3 = &d;

    cout << *p1 << " " << p1 << endl;
    cout << *p2 << " " << p2 << endl;
    cout << *p3 << " " << p3 << endl;

    int num[3] = { 1,2,3 }, * p4 = num;
    char str[3][6] = { "hello","apple","juice" };
    cout << hex << int(p4) << endl;
```

查看各变量信息：①选择“监视1”；②在下面手动输入变量名，查看某个变量信息

普通变量：直接查看值

指向普通变量指针：“值”一栏：十六进制地址 { 指向变量的值 }

一维数组：首地址 { 各元素值 }　指向一维数组的指针：同指向普通变量的指针，” 值

展开：各元素及其对应的值　“一栏显示：十六进制地址 { 指向数组元素的值 }

二维数组：首地址 { 各一维数组首地址 ” 各一维数组值 “ }

展开：各一维数组：该一维数组首地址 { 该一维数组各元素值 }

展开：该一维数组各元素及其对应的值

100 %　未找到相关问题　行: 31　字符: 5　制表符　CRLF

监视 1

搜索(Ctrl+E)　搜索深度: 3

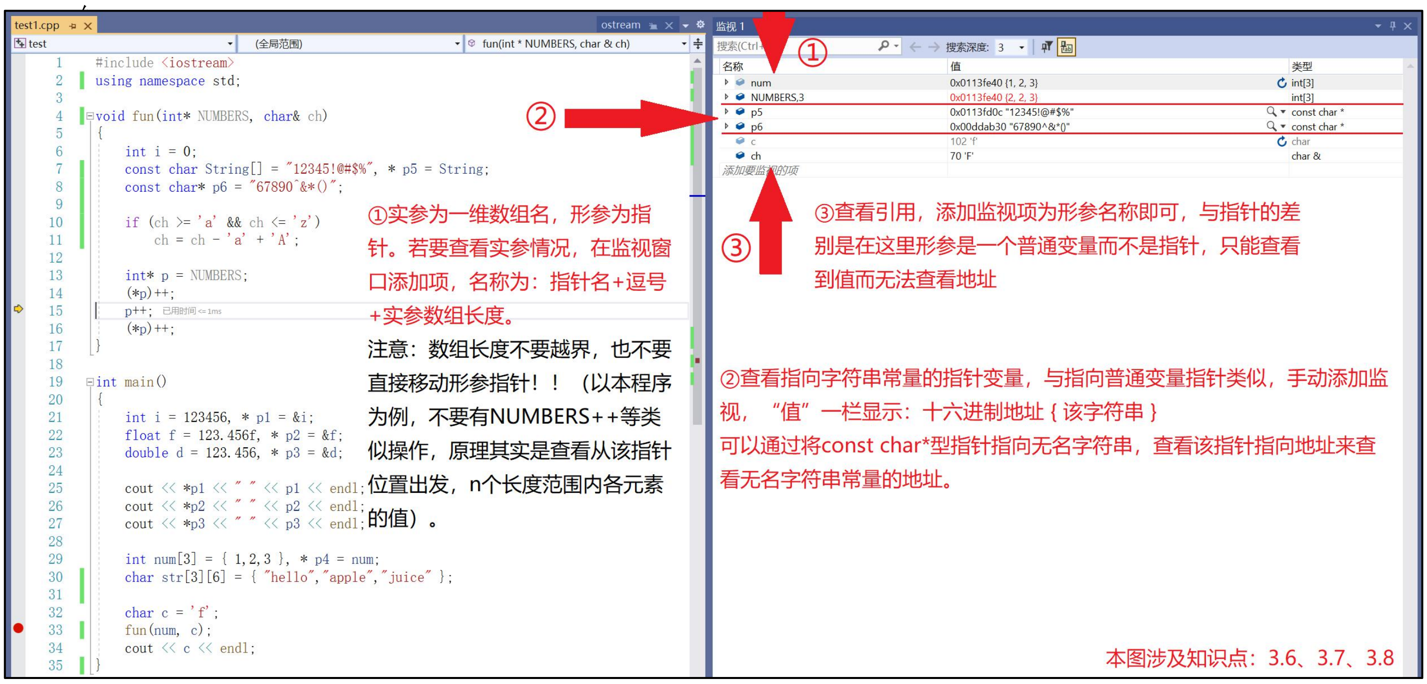
| 名称 | 值 | 类型 |
|---|---|---|
| i | 123456 | int |
| f | 123.456001 | float |
| d | 123.45600000000000 | double |
| p1 | 0x00bff9e4 {123456} | int * |
| p2 | 0x00bff9cc {123.456001} | float * |
| p3 | 0x00bff9b0 {123.45600000000000} | double * |
| num | 0x00bff990 {1, 2, 3} | int[3] |
| [0] | 1 | int |
| [1] | 2 | int |
| [2] | 3 | int |
| p4 | 0x00bff990 {1} | int * |
| str | 0x00bff968 {0x00bff968 "hello", 0x00bff96e "apple", 0x00bff974 "juice"} | char[3][6] |
| [0] | 0x00bff968 "hello" | char[6] |
| [0] | 104 'h' | char |
| [1] | 101 'e' | char |
| [2] | 108 'l' | char |
| [3] | 108 'l' | char |
| [4] | 111 'o' | char |
| [5] | 0 '\0' | char |
| [1] | 0x00bff96e "apple" | char[6] |
| [2] | 0x00bff974 "juice" | char[6] |
| 添加要监视的项 | | |

②

自动窗口　监视 1　①

本图涉及知识点：3.1、3.2、3.3、3.4、3.5

## 2.2. 图 2



test1.cpp | test | (全局范围) | fun(int * NUMBERS, char & ch)

```cpp
#include <iostream>
using namespace std;

void fun(int* NUMBERS, char& ch)
{
    int i = 0;
    const char String[] = "12345!@#$%", * p5 = String;
    const char* p6 = "67890^&*()";

    if (ch >= 'a' && ch <= 'z')
        ch = ch - 'a' + 'A';

    int* p = NUMBERS;
    (*p)++;
    p++;  已用时间 <= 1ms
    (*p)++;
}

int main()
{
    int i = 123456, * p1 = &i;
    float f = 123.456f, * p2 = &f;
    double d = 123.456, * p3 = &d;

    cout << *p1 << " " << p1 << endl;
    cout << *p2 << " " << p2 << endl;
    cout << *p3 << " " << p3 << endl;

    int num[3] = { 1,2,3 }, * p4 = num;
    char str[3][6] = { "hello","apple","juice" };

    char c = 'f';
    fun(num, c);
    cout << c << endl;
}
```

监视 1

搜索(Ctrl+E)　①　搜索深度: 3

| 名称 | 值 | 类型 |
|---|---|---|
| num | 0x0113fe40 {1, 2, 3} | int[3] |
| NUMBERS,3 | 0x0113fe40 {2, 2, 3} | int[3] |
| p5 | 0x0113fd0c "12345!@#$%" | const char * |
| p6 | 0x00ddab30 "67890^&*()" | const char * |
| c | 102 'f' | char |
| ch | 70 'F' | char & |
| 添加要监视的项 | | |

②

①实参为一维数组名，形参为指针。若要查看实参情况，在监视窗口添加项，名称为：指针名+逗号+实参数组长度。

注意：数组长度不要越界，也不要直接移动形参指针！！（以本程序为例，不要有NUMBERS++等类似操作，原理其实是查看从该指针位置出发，n个长度范围内各元素的值）。

③

③查看引用，添加监视项为形参名称即可，与指针的差别是在这里形参是一个普通变量而不是指针，只能查看到值而无法查看地址

②查看指向字符串常量的指针变量，与指向普通变量指针类似，手动添加监视，“值”一栏显示：十六进制地址 { 该字符串 }

可以通过将const char*型指针指向无名字符串，查看该指针指向地址来查看无名字符串常量的地址。

本图涉及知识点：3.6、3.7、3.8

**3. 小程序 3**

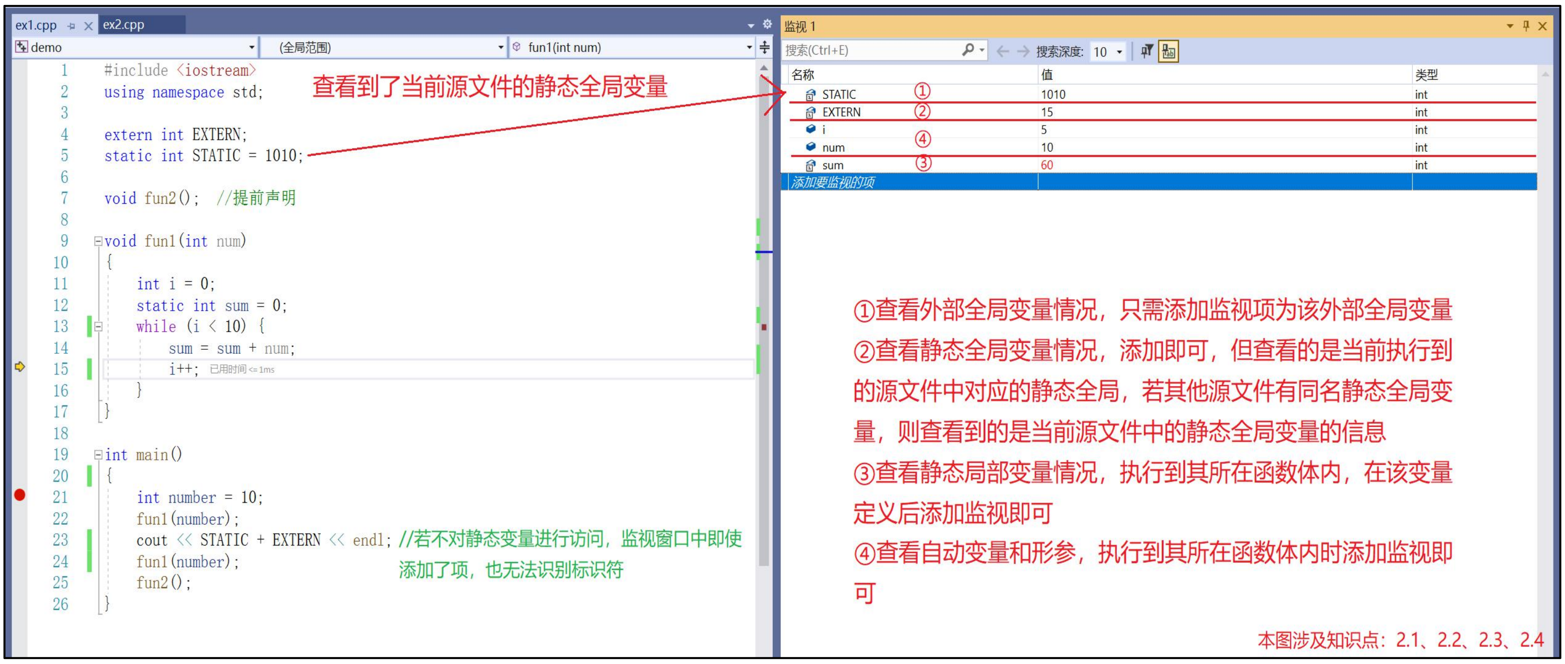
**ex1. cpp**

```
#include <iostream>
using namespace std;

extern int EXTERN;
static int STATIC = 1010;

void fun2();  //提前声明

void fun1(int num)
{
    int i = 0;
    static int sum = 0;
    while (i < 10) {
        sum = sum + num;
        i++;
    }
}

int main()
{
    int number = 10;
    fun1(number);
    cout << STATIC + EXTERN <<
endl;
    fun1(number);
    fun2();
    return 0;
}
```

**ex2. cpp**

```
#include <iostream>
using namespace std;

int EXTERN = 15;
static int STATIC = 2020;

void fun2()
{
    EXTERN++;
    cout << STATIC + EXTERN <<
endl;
}
```

## 3.1. 图 1



ex1.cpp　ex2.cpp　demo　(全局范围)　fun1(int num)

```cpp
#include <iostream>
using namespace std;

extern int EXTERN;
static int STATIC = 1010;

void fun2();  //提前声明

void fun1(int num)
{
    int i = 0;
    static int sum = 0;
    while (i < 10) {
        sum = sum + num;
        i++;  已用时间 <= 1ms
    }
}

int main()
{
    int number = 10;
    fun1(number);
    cout << STATIC + EXTERN << endl; //若不对静态变量进行访问，监视窗口中即使添加了项，也无法识别标识符
    fun1(number);
    fun2();
}
```

查看到了当前源文件的静态全局变量

监视 1

搜索(Ctrl+E)　搜索深度: 10

| 名称 | | 值 | 类型 |
|---|---|---|---|
| STATIC | ① | 1010 | int |
| EXTERN | ② | 15 | int |
| i | ④ | 5 | int |
| num | | 10 | int |
| sum | ③ | 60 | int |
| 添加要监视的项 | | | |

①查看外部全局变量情况，只需添加监视项为该外部全局变量

②查看静态全局变量情况，添加即可，但查看的是当前执行到的源文件中对应的静态全局，若其他源文件有同名静态全局变量，则查看到的是当前源文件中的静态全局变量的信息

③查看静态局部变量情况，执行到其所在函数体内，在该变量定义后添加监视即可

④查看自动变量和形参，执行到其所在函数体内时添加监视即可

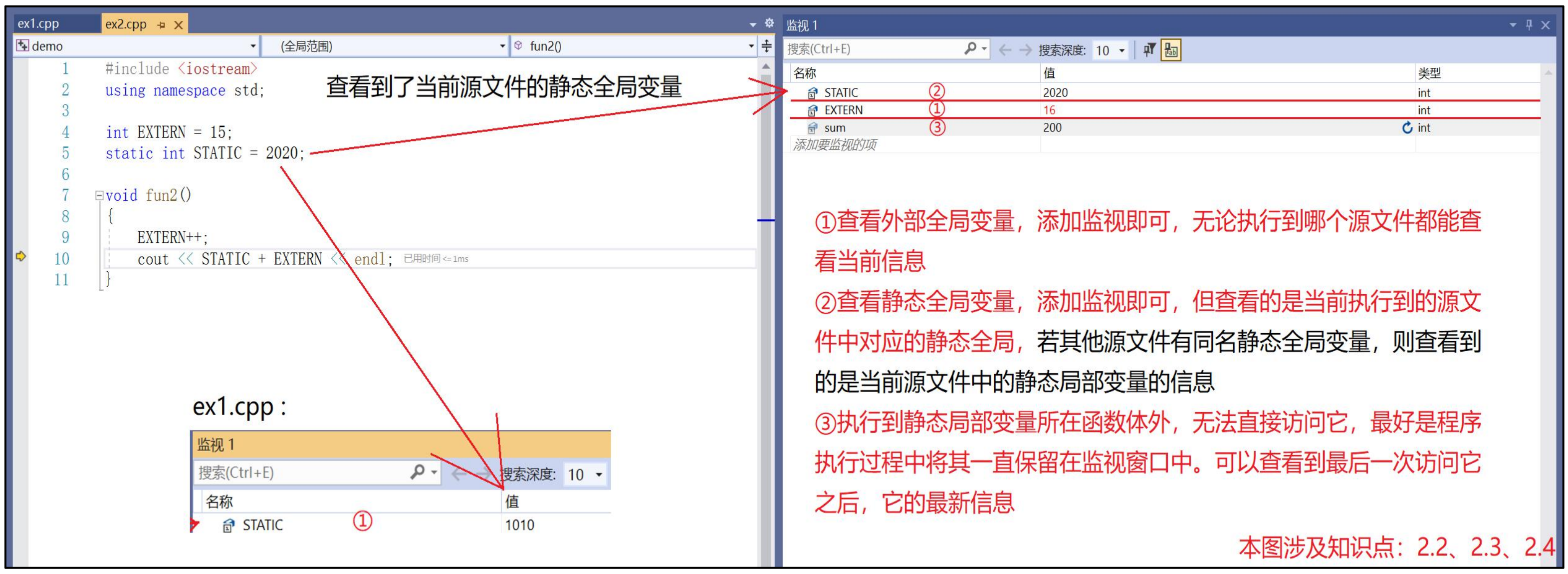
本图涉及知识点：2.1、2.2、2.3、2.4

讠

---

## 3.2. 图 2

纟



ex1.cpp　ex2.cpp　demo　(全局范围)　fun2()

```cpp
#include <iostream>
using namespace std;

int EXTERN = 15;
static int STATIC = 2020;

void fun2()
{
    EXTERN++;
    cout << STATIC + EXTERN << endl;  已用时间 <= 1ms
}
```

查看到了当前源文件的静态全局变量

监视 1

搜索(Ctrl+E)　搜索深度: 10

| 名称 | | 值 | 类型 |
|---|---|---|---|
| STATIC | ② | 2020 | int |
| EXTERN | ① | 16 | int |
| sum | ③ | 200 | int |
| 添加要监视的项 | | | |

ex1.cpp：

监视 1

搜索(Ctrl+E)　搜索深度: 10

| 名称 | | 值 |
|---|---|---|
| STATIC | ① | 1010 |

①查看外部全局变量，添加监视即可，无论执行到哪个源文件都能查看当前信息

②查看静态全局变量，添加监视即可，但查看的是当前执行到的源文件中对应的静态全局，若其他源文件有同名静态全局变量，则查看到的是当前源文件中的静态局部变量的信息

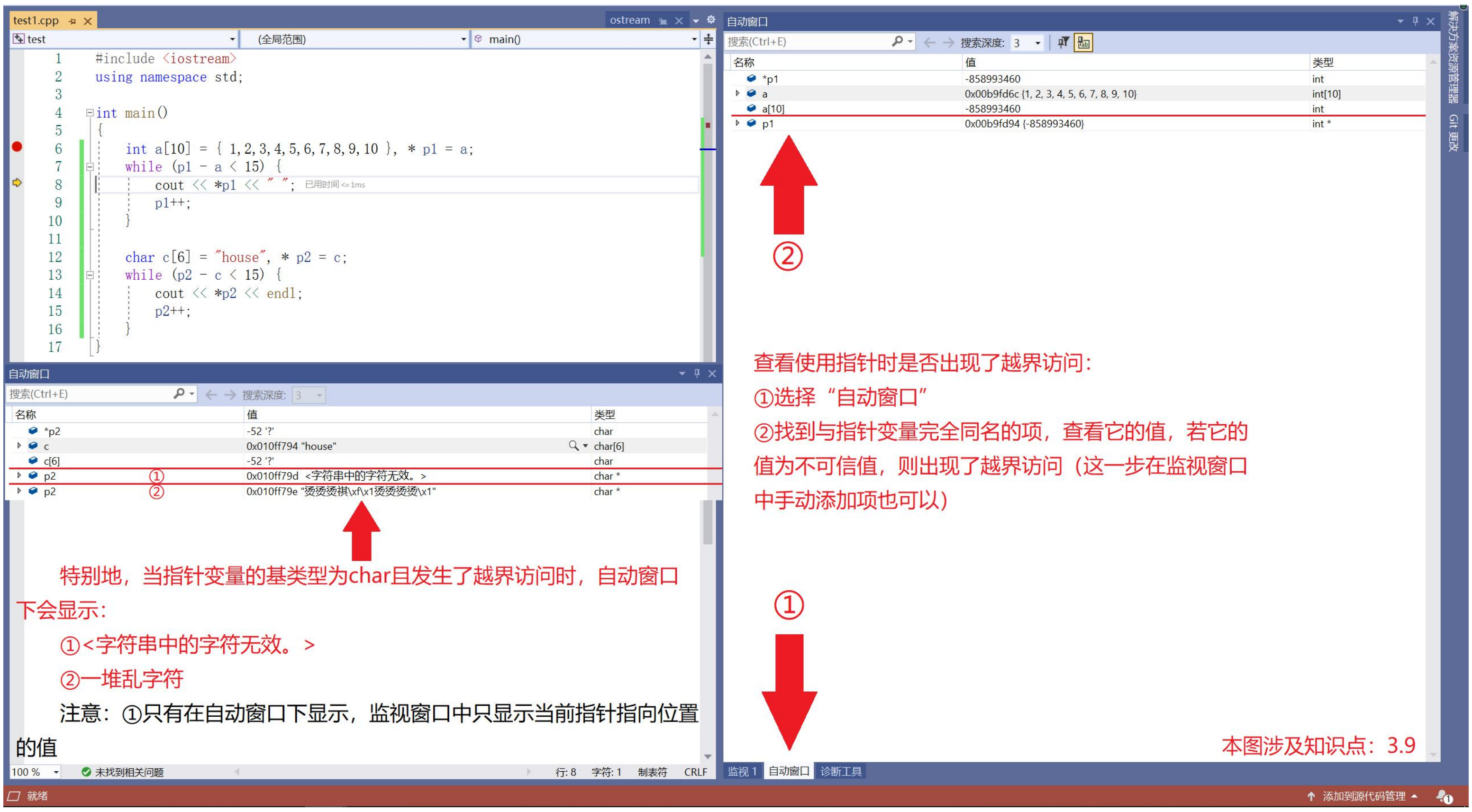
③执行到静态局部变量所在函数体外，无法直接访问它，最好是程序执行过程中将其一直保留在监视窗口中。可以查看到最后一次访问它之后，它的最新信息

本图涉及知识点：2.2、2.3、2.4

## 4. 小程序 4

```
#include <iostream>
using namespace std;

int main()
{
    int a[10] = { 1,2,3,4,5,6,7,8,9,10 }, * p1 = a;
    while (p1 - a < 15) {
        cout << *p1 << " ";
        p1++;
    }

    char c[6] = "house", * p2 = c;
    while (p2 - c < 15) {
        cout << *p2 << endl;
        p2++;
    }
}
```

### 4.1. 图 1



test1.cpp　test　(全局范围)　main()　ostream

```
#include <iostream>
using namespace std;

int main()
{
    int a[10] = { 1,2,3,4,5,6,7,8,9,10 }, * p1 = a;
    while (p1 - a < 15) {
        cout << *p1 << " ";  已用时间 <= 1ms
        p1++;
    }

    char c[6] = "house", * p2 = c;
    while (p2 - c < 15) {
        cout << *p2 << endl;
        p2++;
    }
}
```

自动窗口　搜索(Ctrl+E)　搜索深度: 3

| 名称 | 值 | 类型 |
|---|---|---|
| *p2 | -52 '?' | char |
| c | 0x010ff794 "house" | char[6] |
| c[6] | -52 '?' | char |
| p2 ① | 0x010ff79d <字符串中的字符无效。> | char * |
| p2 ② | 0x010ff79e "烫烫烫祺\xf\x1烫烫烫烫\x1" | char * |

特别地，当指针变量的基类型为char且发生了越界访问时，自动窗口下会显示：

①<字符串中的字符无效。>

②一堆乱字符

注意：①只有在自动窗口下显示，监视窗口中只显示当前指针指向位置的值

自动窗口　搜索(Ctrl+E)　搜索深度: 3

| 名称 | 值 | 类型 |
|---|---|---|
| *p1 | -858993460 | int |
| a | 0x00b9fd6c {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} | int[10] |
| a[10] | -858993460 | int |
| p1 | 0x00b9fd94 {-858993460} | int * |

②

查看使用指针时是否出现了越界访问：

①选择“自动窗口”

②找到与指针变量完全同名的项，查看它的值，若它的值为不可信值，则出现了越界访问（这一步在监视窗口中手动添加项也可以）

①

本图涉及知识点：3.9