







致谢:
南京大学计算机系
可视媒体研究组

断点

- 断点是一个信号，它通知调试器，在某个特定点上暂时将程序挂起，程序处于中断模式，所有程序实体（函数，变量和对象）均保留在内存中。
- 在断点模式下，可以检查程序实体的位置和状态，以确定是否存在冲突或bug。

断点标志符号

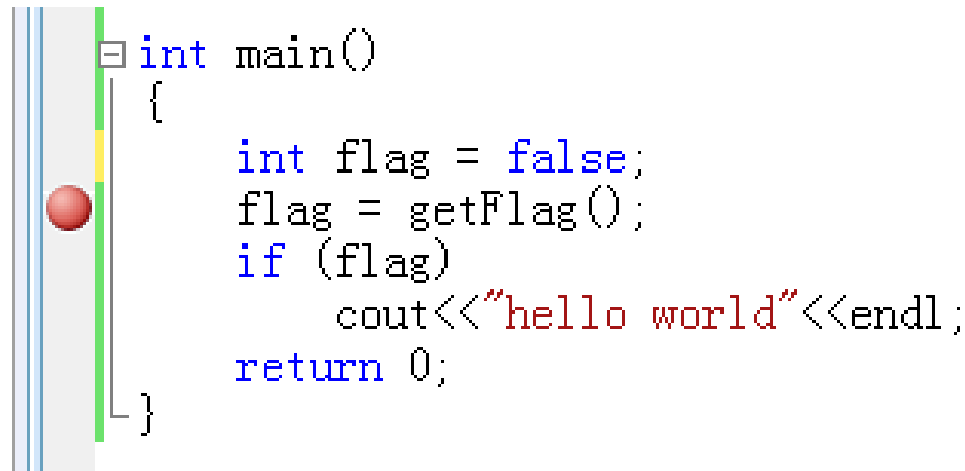
标志符号	说明
	普通断点，活动/禁用，实心表示断点启用，空心表示断点已禁用（常用）
	高级断点，活动/禁用，“+”表示断点至少有一个附加到它的高级功能，如条件、命中次数或筛选器（常用）
	跟踪点，活动/禁用，命中此点则执行指定的操作，但不中断程序的执行。
	高级跟踪点。活动/禁用，“+”表示跟踪点至少有一个附加到它的高级功能，例如条件、命中次数或筛选器
	断点错误或跟踪点错误。“X”指示由于出现错误而未能设置断点或跟踪点。
	断点警告或跟踪点警告。感叹号指示由于临时情况而未能设置断点或跟踪点。通常情况下，这意味着还没有加载断点或跟踪点位置处的代码。加载代码后，将启用断点，标志符号也将更改。

基本断点操作

- 设置简单断点
- 设置函数断点
- 删除断点
- 启用或禁用断点
- 编辑断点位置
- 从“调用堆栈”窗口针对函数调用设置断点

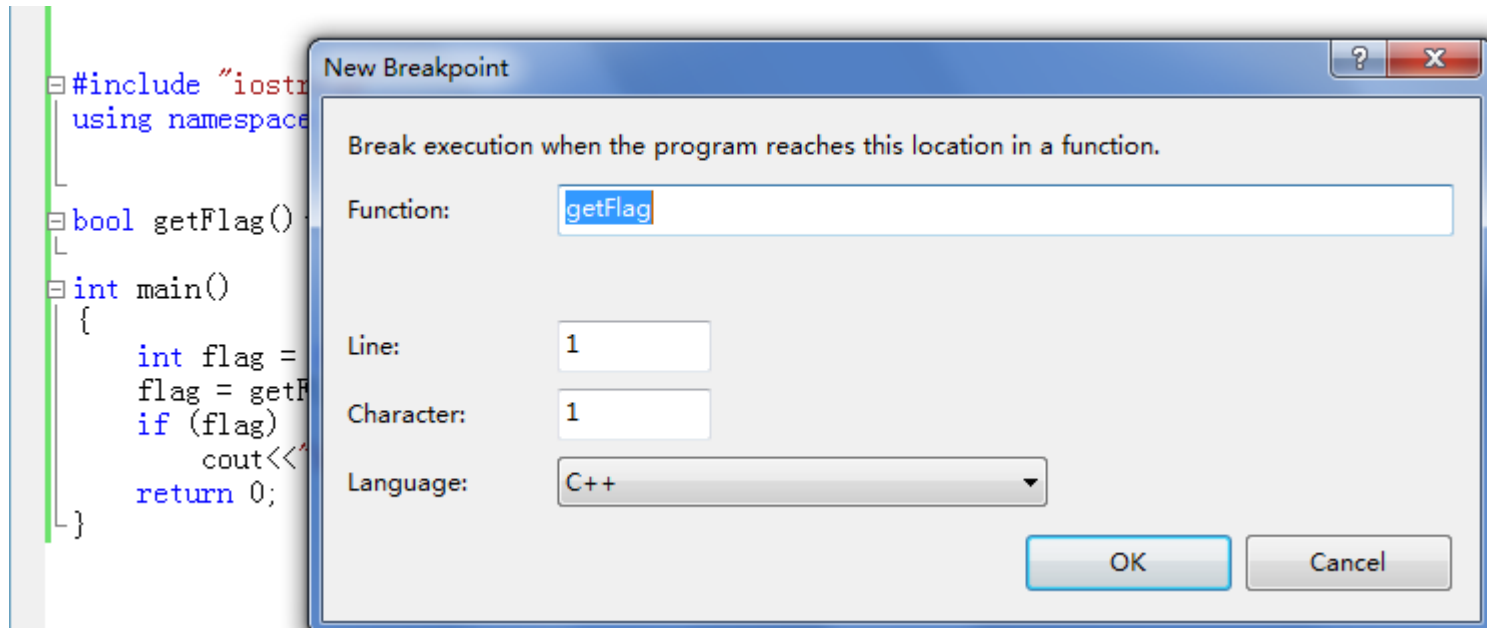
设置简单断点（常用）

- 方法一：鼠标点击代码行左侧灰色区域
- 方法二：鼠标光标移动到代码行，按下F9（亦可通过VS2008中“调试”菜单设定）
- 方法三：在代码行上右击，在弹出的菜单中选择“断点”之后再选择“插入断点”



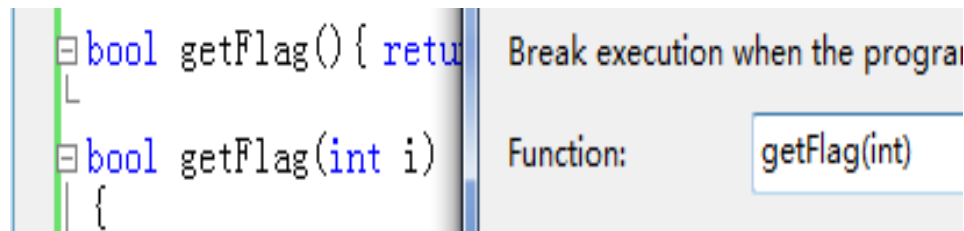
设置函数断点

- 步骤一：将光标移动到函数名上
- 步骤二：从“调试”菜单中选择“新建断点”，再单击“在函数处中断”，出现“新建断点”对话框（或直接使用快捷键Ctrl + B）



设置函数断点

- 步骤三： 点击确定，断点将被设置在函数的开始处。如果要在函数中的其他位置设置断点，可以通过编辑“新建断点”对话框中的“行”和“列”字符框中的值。
- 重载函数设置断点：
 - 可以通过指定参数以设置断点，例如函数有`bool f()` 和 `bool f(int i)`, 对后者在“新建断点”对话框中可使用 `f(int)`，若不指定则两个函数均设置断点



删除断点

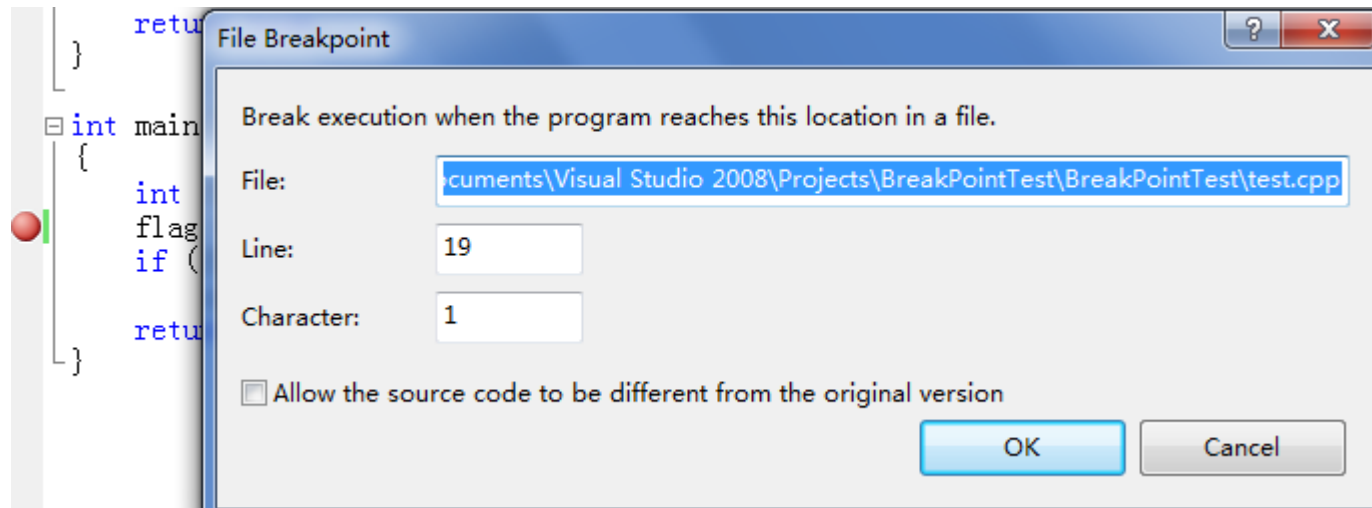
- 方法一：直接用鼠标单击代码行左侧的断点标记
- 方法二：在菜单栏中选择“调试”->“删除所有断点”
- 方法三：使用快捷键 **ctrl + shift + F9**

启用或禁用断点

- 禁用单个断点
 - 在有断点的代码行上右键，然后选择“断点”->“禁用断点”
- 启用单个断点
 - 在有禁用断点的代码行上右键，然后选择“断点”->“启用断点”
- 禁用/启用所有断点
 - 从菜单中选择“调试”->“禁用所有断点"或"启用所有断点"
- 断点被禁用后，在调试工具中仍然存在，但在程序运行时不起作用，启用后可恢复原来功能。

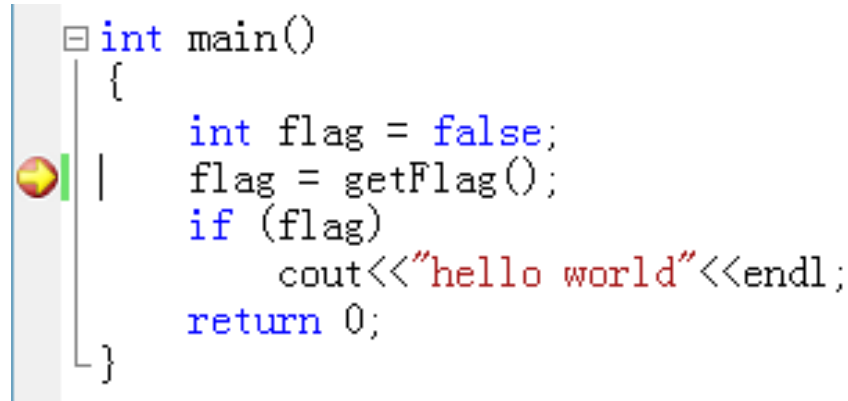
编辑断点位置

- 步骤一：右击断点，在快捷菜单中选择“位置”
- 步骤二：在弹出对话框中，编辑“文件”以更改断点所在文件，编辑“行”以更改文件中断点所在行号，编辑“字符”以更改该行上断点所在水平位置



从“调用堆栈”窗口针对函数调用设置断点

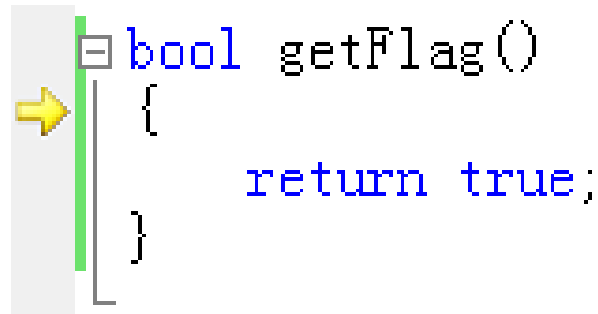
- 步骤一：执行到程序中的断点处



```
int main()
{
    int flag = false;
    flag = getFlag();
    if (flag)
        cout<<"hello world"<<endl;
    return 0;
}
```

A screenshot of a code editor showing the `main` function. A breakpoint, represented by a yellow circle with a red dot, is set on the line `flag = getFlag();`. A green vertical bar is positioned to the left of the code, and a yellow arrow points to the breakpoint.

- 步骤二：按F11(step into)进入函数体，若按F10 (step over)则直接跳过函数调用而不进入函数体

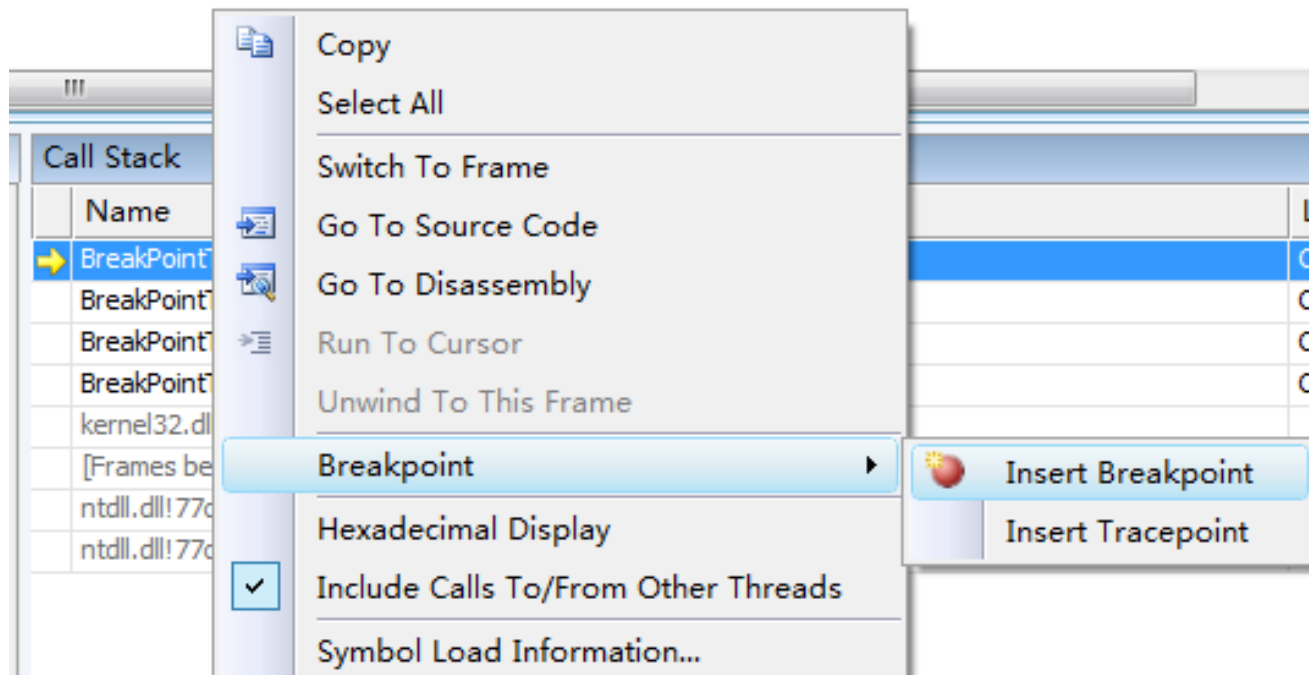


```
bool getFlag()
{
    return true;
}
```

A screenshot of a code editor showing the `getFlag` function. A breakpoint, represented by a yellow circle with a red dot, is set on the line `return true;`. A green vertical bar is positioned to the left of the code, and a yellow arrow points to the breakpoint.

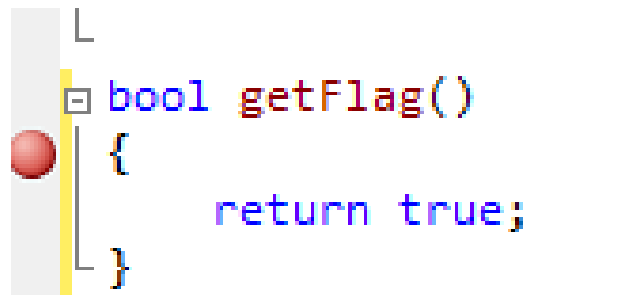
从“调用堆栈”窗口针对函数调用设置断点

- 步骤三：在调用堆栈上选择函数，右键选择“断点”->“插入断点”，即可在函数内部插入断点



在函数体内添加断点另一方法

- 直接在函数体的第一条语句处添加一个断点(F9)
- 在程序中若调用了此函数都会在这个断点处停下来。
- 前提：能够方便地找到此函数的定义，否则使用在调用点执行step into(F11)更方便。

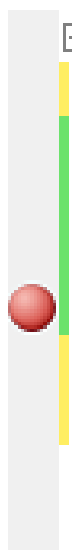


高级断点设置

- 指定条件断点
- 指定命中次数
- 指定跟踪点操作

指定条件断点

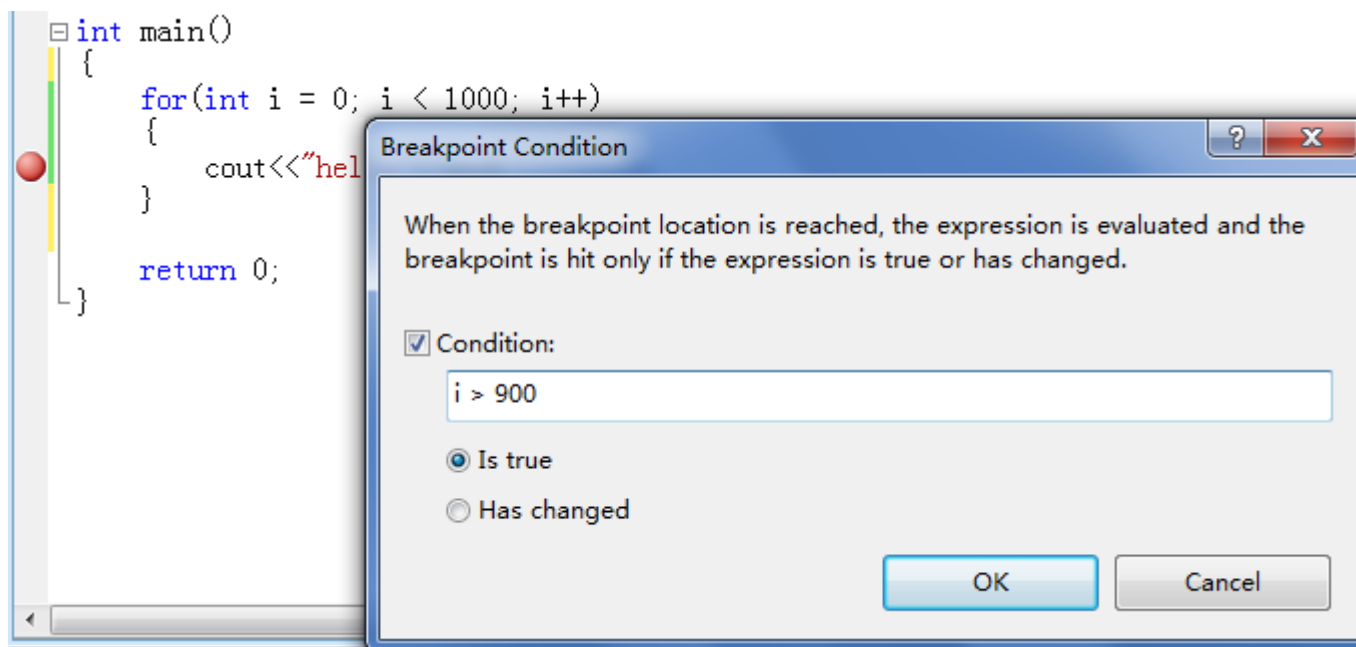
- 如果有一段代码，每次都在执行900次后才有可能出现bug，那么此时为了节省前900次循环的调试时间，可以使用条件断点。假设对于如下代码，在执行900次后，cout函数会崩溃，那么如何去定位崩溃呢？



```
int main()
{
    for(int i = 0; i < 1000; i++)
    {
        cout<<"hello"<<endl;
    }
    return 0;
}
```

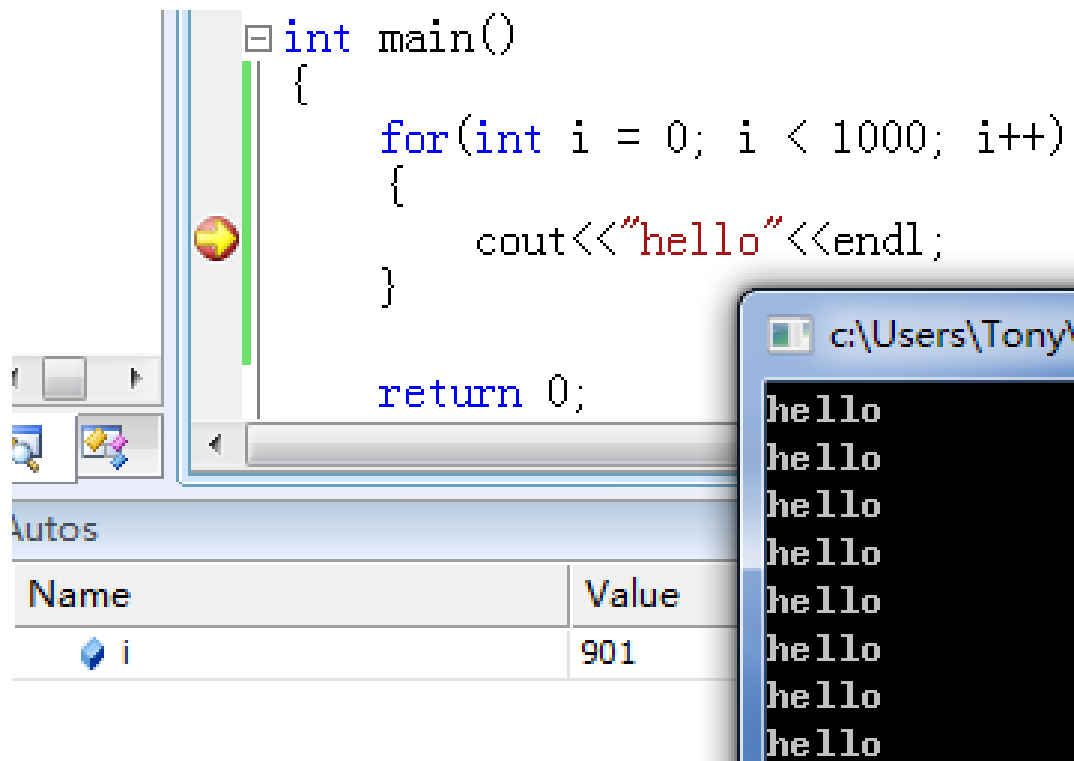
指定条件断点

- 右击断点，选择“条件”，弹出对话框，在“条件”文本框中编辑条件，此例需要的条件如下：



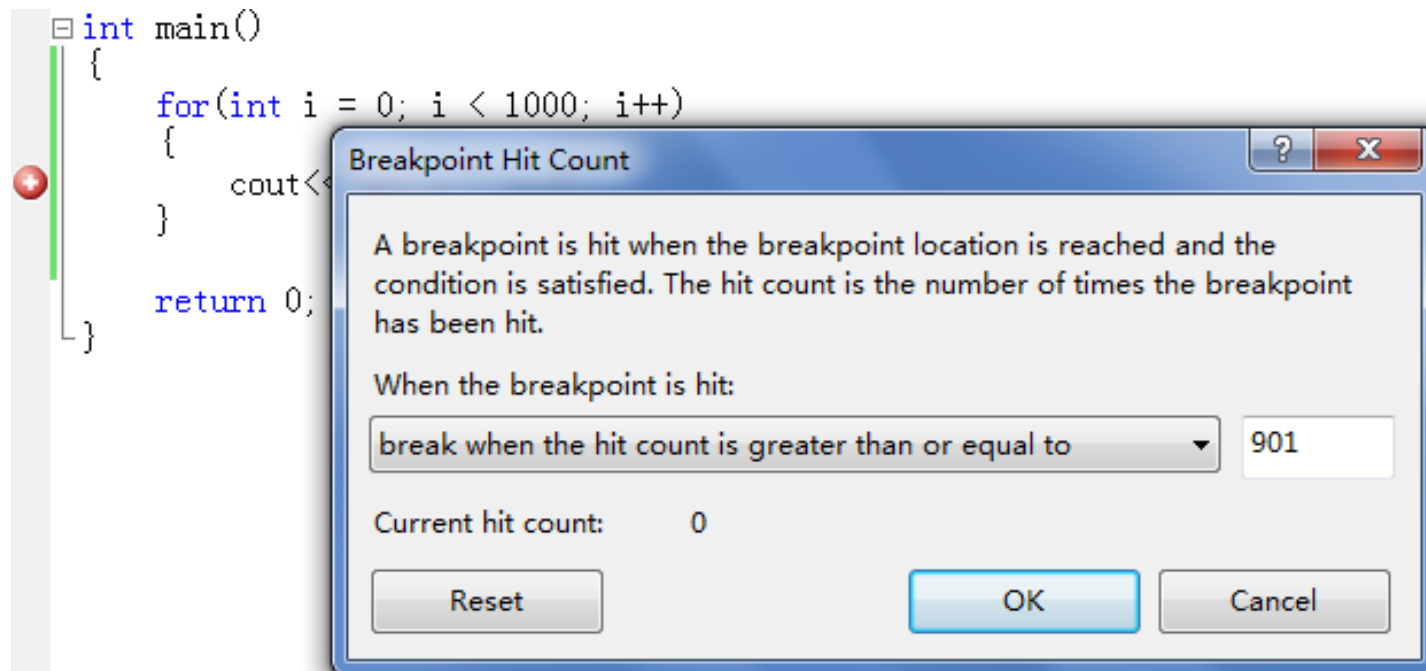
指定条件断点

- 可以看到在i为901时，程序被中断执行



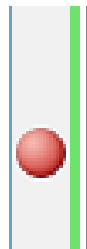
指定命中次数

- 上例中的情况，亦可用指定断点的命中次数来实现。右击断点，选择“命中次数”，弹出设定窗口



指定跟踪点操作

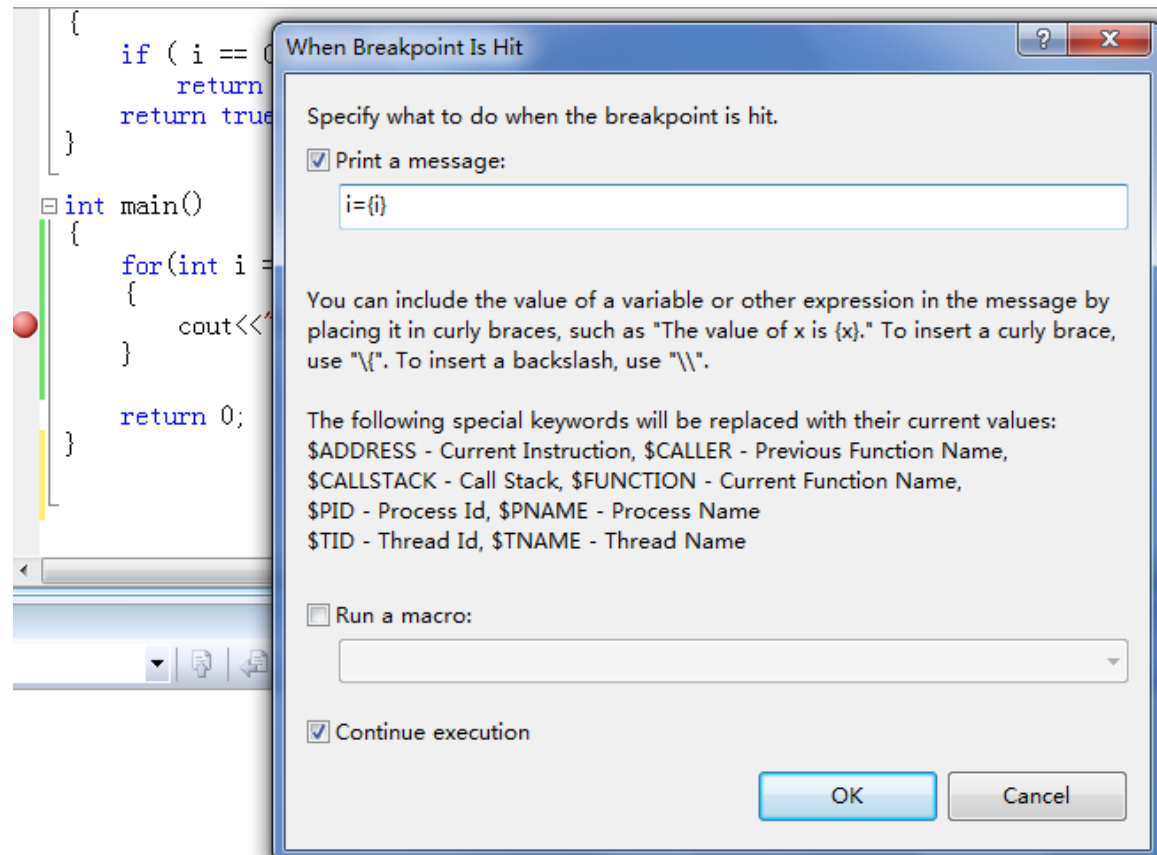
- “跟踪点”是一种特殊的断点，当它被命中时，它会触发一系列自定义操作。如果你想观察程序的行为，而又不想中断调试的时候，这个功能尤其有用。
- 继续使用前一个例子，在循环体中先设定普通断点



```
for(int i = 0; i < 1000; i++)  
{  
    cout<<"hello"<<endl;  
}
```

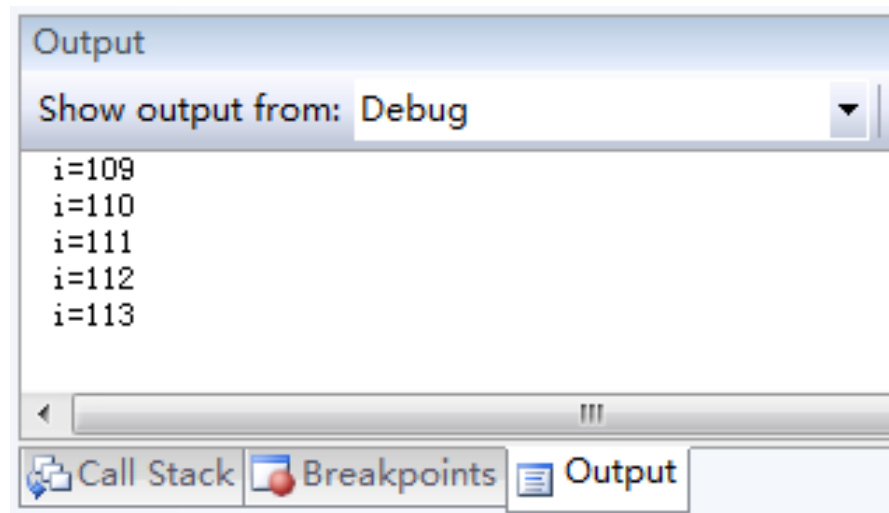
指定跟踪点操作

- 在断点上右击，选择“命中断点时”，弹出对话框



指定跟踪点操作

- 在弹出对话框中，可以设置命中该断点时，触发的事件，本例中是打印出*i*当前的值。
- 本例中，我们同时选择了底端的“继续执行”选项，说明不希望程序中断，而是继续运行。与断点唯一不同是，自定义跟踪信息都将被输出。



常用调试操作

- 操作一：在菜单中选择“调试”->“继续”，程序会运行到下一个断点处暂停或到程序结束。
- 操作二：如要仔细观察代码执行情况，可采用F10或者F11单步调试找到精确错误处。其中F10(step over)是跳过函数调用，F11(step over)是进入函数体调用。一般先用F10，确定函数结果是否正确，如不正确则使用F11进入函数体调试。还可以用Shift+F11跳出函数体。

常用调试操作

- 操作三：运行到当前光标下，在某一行代码，右击鼠标，选择“运行到光标”，程序就会执行到鼠标的地方，快捷键(Ctrl + F10)
- 操作四：locals窗口，可以查看运行到断点处局部作用域的所有变量的值，若变量的值有变化会用红色标出
- 操作五：watch窗口，在watch栏中输入变量的名，可以查看运行到断点处内存中变量的值，通常在local窗口中为列出自己感兴趣的变量时使用。

常用调试操作

- 操作六：Autos窗口，列出在当前程序执行的局部所相关的变量，在当前局部作用域中变量较多时可以使用该窗口。
- 操作六：在Debug状态下，若未出现想要的调试窗口时，可以选择菜单“调试”->“窗口”，可以选出watch, locals等窗口。
- 操作七：可以在程序的关键位置打印出关键的变量，这样有助于判断问题是出现在之前还是之后

常用的调试快捷键总结

熟练地使用常用的快捷键比每次用鼠标点按钮要方便准确得多，大家要牢记下面的快捷键。所有的快捷键在相应的菜单下都有写出，随时可查看。

未进入调试时：

- **F5**：开始调试，若遇到断点会停止。
- **Ctrl+F5**：开始运行，若遇到断点不会停止。
- **F9**：在光标所在行设置或取消一个简单断点。

常用的调试快捷键总结（续）

进入调试后：

- **F5**：继续执行程序直到遇到下一个断点。
- **Shift+F5**：停止调试。
- **Ctrl+Shift+F5**：重新开始调试。
- **F10**：Step over，执行下一行语句，若有函数调用不会进入函数体。
- **F11**：Step into，执行下一行语句，若有函数调用会进入函数体。
- **Shift+F11**：Step out，跳出当前函数。

断点调试示例1： 计算平均分

```
int main()
{
    int scores[10];
    cout<<"input socres: ";
    for (int i = 0; i < 10; ++i)
        cin>> scores[i];

    int sum = 0;
    int max = scores[0];
    int min = scores[0];
    for (int i = 0; i < 10; ++i)
    {
        if (scores[i] > max) max = scores[i];
        if (scores[i] < min) min = scores[i];
        sum += scores[i];
    }

    int sumr = sum - max - min;
    double avg = sumr / 8;

    cout<<"Canceled one of the max scores: "<<max <<endl;
    cout<<"Canceled one of the min scores: "<<min <<endl;
    cout<<"The average score is: "<<avg <<endl;

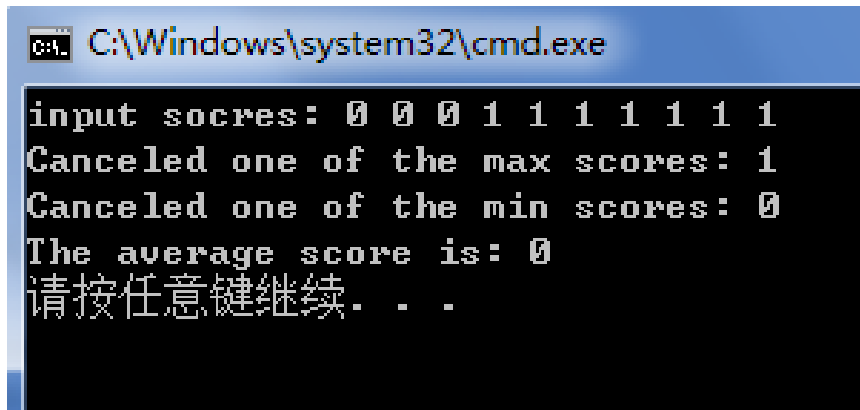
    return 0;
}
```

断点调试示例1：计算平均分

前一页代码是输入10个分数，去掉一个最高分一个最低分后再计算平均值的标准写法。并且作者考虑到，计算平均值时使用到除法，会出现小数所以在定义平均值时特地使用了

```
double avg = sumr / 8;
```

但是，输入如下值时，平均值却为0



```
C:\Windows\system32\cmd.exe
input socres: 0 0 0 1 1 1 1 1 1 1
Canceled one of the max scores: 1
Canceled one of the min scores: 0
The average score is: 0
请按任意键继续. . .
```

断点调试示例1：计算平均分

- 我们知道，对于 0 0 0 1 1 1 1 1 1 1 这样的输入，去掉最低分和最高分后，平均值应该是0.75而不是0，分析代码，可能发生错误的地方有以下几处
 - 最大值、最小值计算
 - 10个分数的总和计算
 - 去掉最高分最低分
 - 平均值计算

断点调试示例1： 计算平均分

- 我们在可能出现错误的地方设置好断点

```
int main()
{
    int scores[10];
    cout<<"input socres: ";
    for (int i = 0; i < 10; ++i)
        cin>> scores[i];

    int sum = 0;
    int max = scores[0];
    int min = scores[0];
    for (int i = 0; i < 10; ++i)
    {
        if (scores[i] > max) max = scores[i];
        if (scores[i] < min) min = scores[i];
        sum += scores[i];
    }







    int sumr = sum - max - min;
    double avg = sumr / 8;

    cout<<"Canceled one of the max scores: "<<max <<endl;
    cout<<"Canceled one of the min scores: "<<min <<endl;
    cout<<"The average score is: "<<avg <<endl;

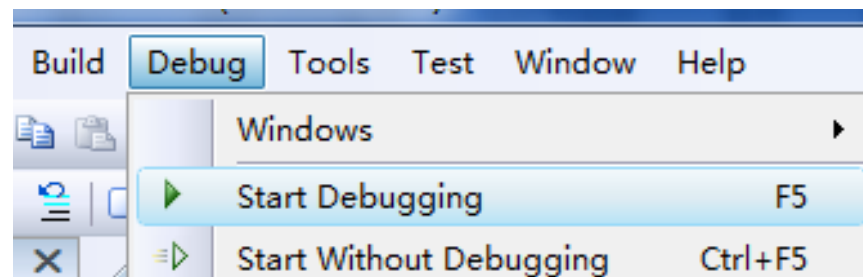
    return 0;
}
```

断点调试示例1： 计算平均分

- 在watch窗口中设置好要观察的变量


Watch 1		
Name	Value	Type
 i	0	int
 max	0	int
 min	0	int
 sum	0	int
 sumr	-858993460	int
 avg	-9.2559631349317831e+061	double

- 按下“开始调试”按钮




断点调试示例1：计算平均分

- 程序运行到循环中，被中断



```
for (int i = 0; i < 10; ++i)
{
    if (scores[i] > max) max = scores[i];
    if (scores[i] < min) min = scores[i];
    sum += scores[i];
}
```

- 如果，此时你能确定循环肯定正确，那么你可以取消循环中的断点，再按下F5继续往下执行



```
for (int i = 0; i < 10; ++i)
{
    if (scores[i] > max) max = scores[i];
    if (scores[i] < min) min = scores[i];
    sum += scores[i];
}

int sumr = sum - max - min;
double avg = sumr / 8;
```

- 程序执行完循环后，到下一断点时停下。同时要注意观察watch窗口中变量的值

断点调试示例1： 计算平均分

- watch窗口中的变量值如下所示，佐证了到目前为止程序都是正确的判断

Watch 1		
Name	Value	Type
i	10	int
max	1	int
min	0	int
sum	7	int
sumr	-858993460	int
avg	-9.2559631349317831e+061	double

- 按F10，继续单步调试，去掉最高分最低分后，sumr为6

Watch 1		
Name	Value	Type
i	10	int
max	1	int
min	0	int
sum	7	int
sumr	6	int
avg	-9.2559631349317831e+061	double

断点调试示例1：计算平均分

- 再用F10，执行计算平均值的语句，再观察watch窗口中的变量，此时，avg值竟然显示为0！

Watch 1		
Name	Value	Type
◆ i	10	int
◆ max	1	int
◆ min	0	int
◆ sum	7	int
◆ sumr	6	int
◆ avg	0.0000000000000000	double

断点调试示例1：计算平均分

- 此时，我们基本可以确定程序中bug的位置，就在平均值计算语句附近（心情好激动~~）。仔细检查平均值计算语句

```
double avg = sumr / 8;
```

avg没有问题，是double类型，唯一可能发生问题的就是sumr变量，检查sumr变量的定义

```
int sumr = sum - max - min;
```

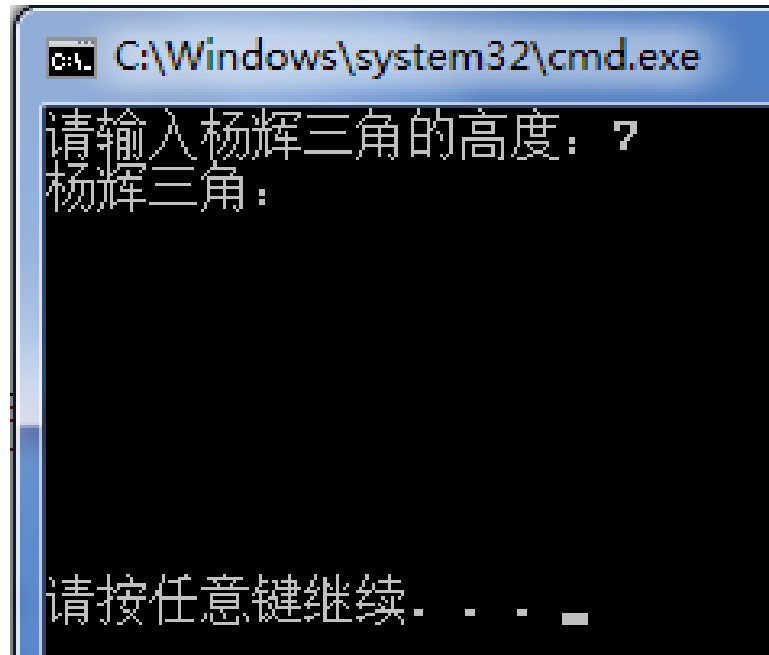
此时，问题已经浮现，sumr为整型数，sumr/8使用的是整型数除法，直接抛弃小数位！改为double即可，至此，一个完整的断点调试工作结束。

断点调试示例2： 杨辉三角

```
int a[M][N] = {0}, m, n, i, j;
cout << "请输入杨辉三角的高度：";
cin >> m;
n = m * 2;
for (i = 0; i < m; ++i)
{
    for (j = m-i; j < m+i; j += 2)
    {
        if (i == 0)
            a[i][j] = 1;
        else
            a[i][j] = a[i-1][j-1] + a[i-1][j+1];
    }
}
//输出杨辉三角部分请自行想象...
```

断点调试示例2：杨辉三角

- 前一页是生成杨辉三角的代码，作者深谙空间换时间的潜规则，开辟了 $m \times 2m$ 大小的空间用于存放杨辉三角，代码非常简洁，但是在输出时，却意外地发现，什么都没有，怎么回事！？

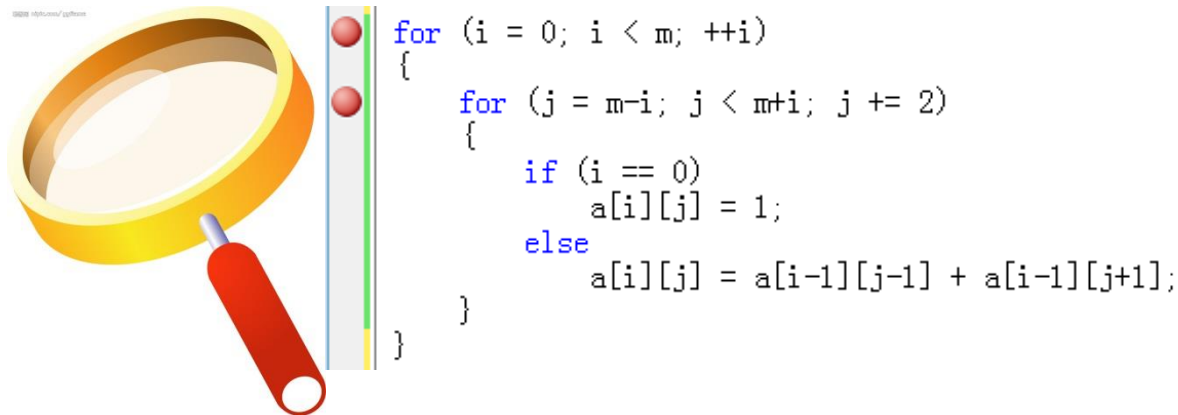


```
C:\Windows\system32\cmd.exe
请输入杨辉三角的高度：?
7
杨辉三角：

请按任意键继续...
```

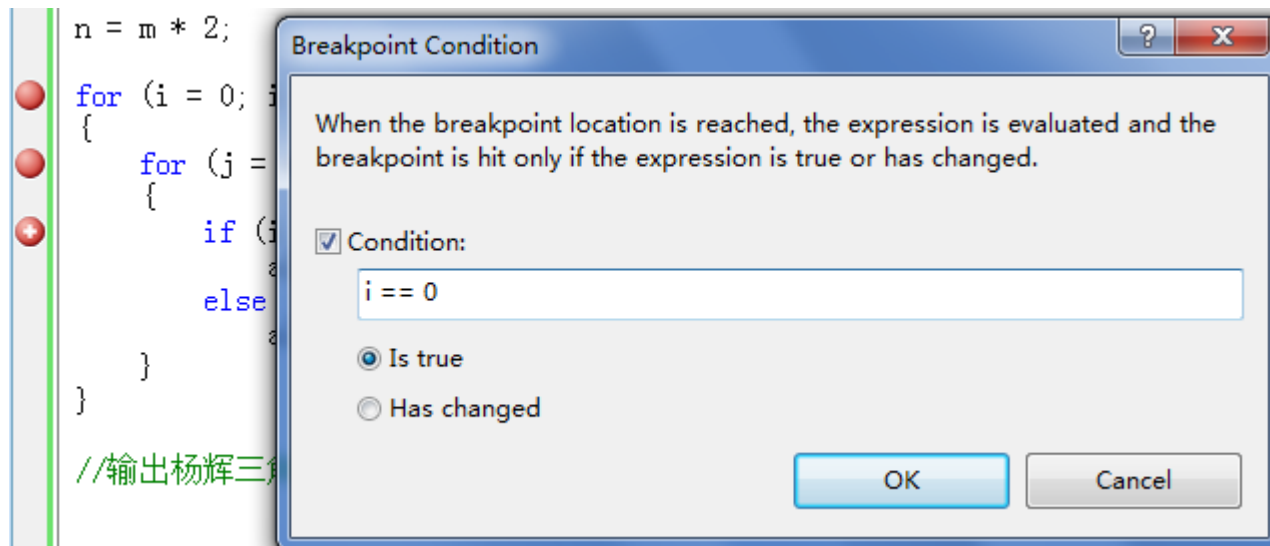
断点调试示例2：杨辉三角

- 有了示例1的经验后，我们不假思索地祭出断点调试的大旗，先找可能发生错误的地方！（本例中假设打印杨辉三角的程序没有错误）
 - 唯一可能隐匿错误的就是那个双循环，拿起放大镜仔细检查



断点调试示例2：杨辉三角

- 一番忙活，F10，F11按到键盘要爆，结果发现两个循环都毫发无损地执行了。就在你三观都快崩溃时，扫地阿姨提醒你，该用条件断点啦。
- 再检查代码，发现还有if判断语句，会不会是if语句体没有执行呢，设置一个条件断点试试吧~



断点调试示例2：杨辉三角

- 按照我们设置的条件断点，只要i等于0，那么程序就会一本正经地中断停下来，等待你检验。出乎我们意料地是，在我们按下F5后（先去掉两个for循环那的断点）程序一路绿灯从头跑到尾，丝毫没有停下来的迹象。

Bingo!

我们差不多找到问题的位置了，if判断语句没有执行。

断点调试示例2：杨辉三角

- 会不会是if语句的条件没有满足呢？第一重for循环for (i = 0; i < m; ++i) 已经将i设置为0，那么问题肯定发生在第二重循环，这时已经有90%的把握判断是第二个for循环没有执行了。
- 仔细检查， $j = m - i$ ， $j < m + i$ ，在i等于0时，for循环条件不满足，没有进入循环体。而第二个for循环是用于生成杨辉三角在第i层的数字，应该从m-i-1开始，将第二层for循环改为

for (j = m-i-1; j < m+i; j += 2)

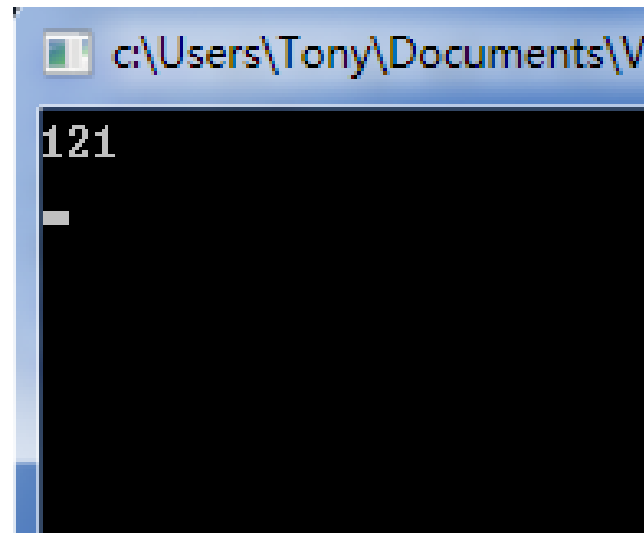
问题解决，又可以看到智慧与美貌并存的杨辉三角了！

断点调试示例3： 循环大逃亡

```
int main()
{
    double x1 = 0, x2 = 0;
    double num;
    cin >> num;
    x2 = num;
    double result = fabs(x1 - x2);
    while (result >= 1e-8)
    {
        x1 = x2;
        x2 = (2*x1 + num / (x1 * x1)) / 3;
    }
    cout << num << "的立方根是: " << x2 << endl;
    return 0;
}
```

断点调试示例3：循环大逃亡

- 前一页的代码是用迭代法来计算数字的立方根，你美滋滋地按下 `ctrl + F5`（根据契约精神，你给程序一个输入，程序给你一个输出），但是等过了夏天，等到了秋天，输入还在，输出没了。。。



断点调试示例3：循环大逃亡

- 既然代码里面有循环，那么我们自然就是到循环头上去找啦，循环毛还得出在循环身上！
 - 状况一：循环没有执行
 - 状况二：循环一直执行

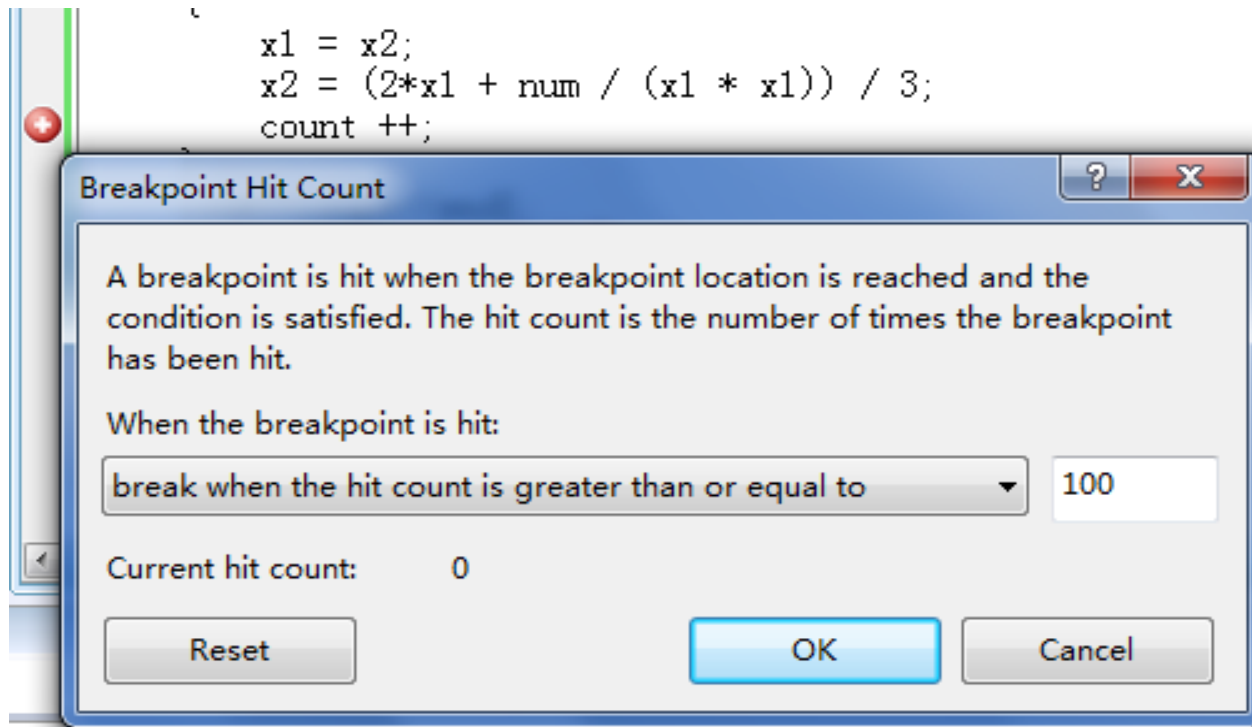
断点调试示例3：循环大逃亡

- 对于状况一，如果循环没执行，那么接下来的输出语句应该执行，所以状况一洗清嫌疑，无罪释放。
- 至于状况二，循环问题中的惯犯，一定要好好审查。对于程序中的循环次数，我们可以进行一个大致的估计，开立方这件小事，迭代一百次已经可以达到较高的精度了。我们设定一个循环计数器count

```
while (result >= 1e-8)
{
    x1 = x2;
    x2 = (2*x1 + num / (x1 * x1)) / 3;
    count ++;
}
```

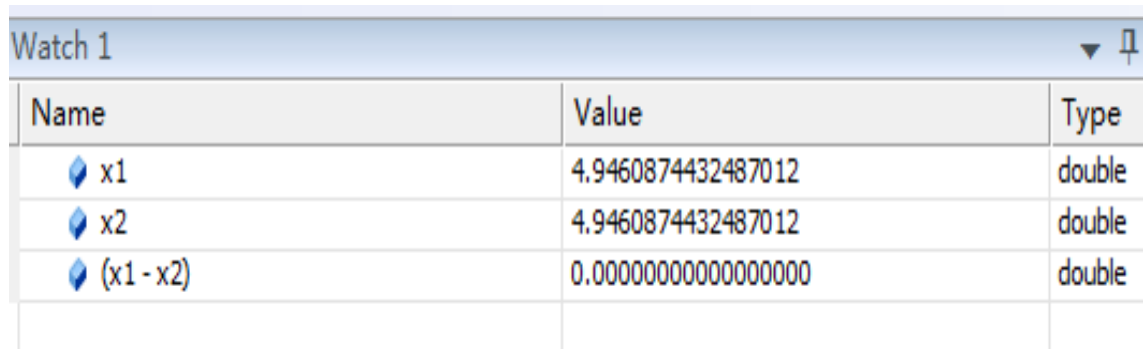
断点调试示例3：循环大逃亡

- 在count上设置命中次数断点，循环超过一百次即中断程序执行



断点调试示例3：循环大逃亡

- 此时，我们再检查`fabs (x1 - x2)`是否小于`1e-8`



The screenshot shows a debugger's 'Watch 1' window. It contains a table with three rows of data. Each row has a blue diamond icon next to the variable name. The first two rows show 'x1' and 'x2' both with the value '4.9460874432487012'. The third row shows '(x1 - x2)' with the value '0.0000000000000000'. All three variables are of type 'double'.

Name	Value	Type
x1	4.9460874432487012	double
x2	4.9460874432487012	double
(x1 - x2)	0.0000000000000000	double

- 而此时`fabs(x1 - x2)`的值早已为0，循环条件早已满足，为什么还不退出循环呢？

断点调试示例3：循环大逃亡

- 再检查循环条件

```
while (result >= 1e-8)
```

`result`被用来代替`fabs(x1 - x2)`，而`result`在循环体却一直没有被更新，导致循环一直无法退出，在`while`体中加入

```
result = fabs(x1 - x2)
```

这样，你和程序之间的契约关系又建立起来了！

实用编程技巧1

- 写完一个部分（如，一个函数，一个循环等）就进行编译，而不是写完整个程序才进行编译

```
#include <iostream>

using namespace std;

void Func();
{
    //...
}

int main()
{
    return 0;
}
```

常见问题

在编写Func函数后就可以找出它，
而不必等到main函数写完

实用编程技巧2

- 控制台中也可以粘贴/框选/复制



实用编程技巧3

- 在比较语句中，将常量置于左侧，以避免错误

```
if( a == 1 )  
{  
    //...  
}
```

常用比较语句

```
if( 1 == a )  
{  
    //...  
}
```

更好的写法

```
if( a = 1 )  
{  
    //...  
}
```

常见错误，不易发现

```
if( 1 = a )  
{  
    //...  
}
```

编译不通过，易于发现

实用编程技巧4

- 尽量推迟变量的定义

```
int iSum = 0;
for( int i = 0 ; i < iStuNum ; i++ )
{
    for( int j = 0 ; j < iScoreNum ; j++ )
    {
        iSum += aaiScores[i][j];
    }
    cout<<"Average = "<<iSum/((float)iScoreNum<<endl;
}
```

常见问题：此处忘记清零iSum

实用编程技巧4

- 尽量推迟变量的定义

```
for( int i = 0 ; i < iStuNum ; i++ )  
{  
    int iSum = 0;  
    for( int j = 0 ; j < iScoreNum ; j++ )  
    {  
        iSum += aaiScores[i][j];  
    }  
    cout<<"Average = "<<iSum/(float)iScoreNum<<endl;  
}
```

推迟iSum的定义

结束语

以上是简单的断点调试操作的介绍。调试代码是一件需要耐心和技巧的工作，同学们需要不断地去尝试自己动手调试代码，在实践中熟悉，掌握现有调试方法，发现新的调试技巧。