3　语义错误：试试断点法和输出变量法

出现了语义错误的的话，很可能不报错不警告，这时候解释器的报错机制就救不了你了。

的确，这样的错误最好的方法就是自己认真的去重新阅读分析代码，显然这是个苦差事。但是，如果掌握了断点和输出变量的用法，那这个过程就会轻松很多。

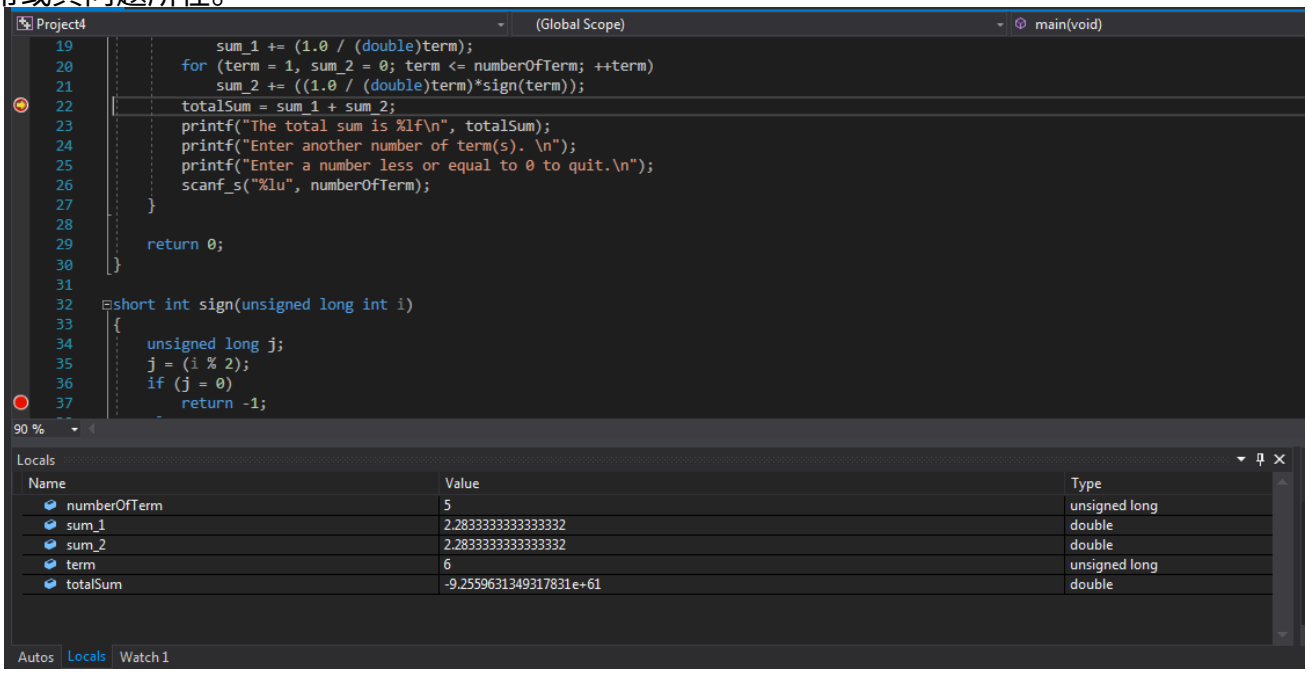
3.1　输出变量法

输出变量调试法就是，在程序中加入一些语句，把程序变量的值打印出来。通过分析这些变量的变化，可以知道程序的问题所在。

当然，你可以用 `disp()` 或 `printf()` 打印所要监视的变量。但是这并不是一个简单的办法。别忘了，Octave 自带一个属性，就是不加分号就打印变量。我们只要把要监视的变量后面的分号去掉就好了。

3.2　断点法是什么黑科技?

断点就是指在程序中设置一个位置。每次程序运行到这个位置时，就会暂时停下来。同时，程序会显示此时程序中各个变量的值。通过分析这些值，程序猿就能清楚的知道程序运行是否正常或其问题所在。



如图中用 MS Visual Studio 调试 C 语言，图中两个红点所在的那个行就是断点的位置，而有黄的箭头的那个断点是程序暂停的位置。而下方的窗格里有各个变量的类型和值。

其实说到这里，大家可能会发现，无论是输出变量法还是断点法，本质上都是使变量的值可视化。只不过，用变量输出法很可能会一下子打出来一坨，找到到底是第几步哪个位置出的毛病很折磨人，而让程序停下来看此时的变量，显然更方便。

3.3 自己制造断点

刚才所说的断点功能，主要是一些集成开发环境（IDE）和编辑器的插件提供的。（这两个词不认识不要怕，我应该会在下下次发文时进行解释）

“哇，这个‘断点’好厉害，Octave 里在哪开啊？快说快说！”

很可惜，Octave 貌似没有提供这样的黑科技。

可能有人看到这里就有打我的冲动了：“没有你说它干嘛啊？增加我们心理落差？”诶诶，各位好汉别冲动，我们可是有创造世界能力的电子信息科技工程师俗称程序猿啊。原来没有的东西，按照程序员的创造探知精神，应该是自己动手丰衣足食，用奇淫巧技造出来啊！

首先，仔细想一下，断点法比输出变量法多的，就是能让程序暂停。那么我们只要用去分号的方法，就能实现显示变量的功能了。

好了，断点的第二个功能已经有了。至于第一个让程序停下来的功能么，实现方法就是，在我们想让程序停下的地方，暂时加上一个 `input("");`。注意不用把这个 `input();` 的返回值赋给任何变量，但是必须给 `input();` 的括号里放一个字符串作为参数，否则这个函数就会因为缺乏参数而无法运行。如果想不到什么好的句子，就在括号里打个引号使其生成一个空字符串就行了。

这样做的原理就是，`input();` 这个自带函数会让程序暂时停下来，然后等待用户输入一些东西。只有用户输入了什么并按下回车后，`input();` 如愿以偿的拿到了它想要的输入，然后程序就会再次运行。如果你用这种方法暂停了程序，只需要随便输入点什么敲回车就能让程序重新跑起来。实际上，直接敲回车也是可以的，因为这样会输入一个回车符或者换行符。

这样，自制断点就大功告成了！

3.4 示例

举一个例子来说明一下断点应该怎么用。下面是一个用于计算迭加的程序。

```
function result = mySum(upperLim)
%程序有错误
    addedNum = 1;
    sum = 0;
    while addedNum < upperLim
        sum += addedNum;
        addedNum += 1;
    endwhile
    result = sum;
endfunction
```

该程序一个运行示例如下

```
>> mySum(10)
ans = 45
```

但是，如果用公式自己算的话，从 1 加到 10 的结果应该是

$$1+2+\cdots+10=10\times\frac{1+10}{2}=55$$

显然出错了。我们用自制断点法，去掉程序中的分号，然后在循环的最后加上一个 input，

然后现在代码应该长这样

```
function result = mySum(upperLim)
% 断点调试版本
    addedNum = 1
    sum = 0
    while addedNum < upperLim
        sum += addedNum
        addedNum += 1
        input("Paused. Press enter to continue."); % 断点在此
    endwhile
    result = sum
endfunction
```

现在建议再开一个窗口同样打开这份 PDF，然后对照着刚才的代码看下面的分析。如果你的 PDF 阅读器不支持二开，网页浏览器可以一战。

重新再运行一次，等到第一次停下来时，屏幕上出现这样的内容

```
>> mySum(10)
addedNum = 1
sum = 0
sum = 1
addedNum =  2
Paused. Press enter to continue.
```

运行结果中的第 2、3 行显然对应着源代码的 3、4 行。此时还没有进入循环。第 4 行又出现了一次 `sum`，这对应的是第 6 行，此时已经进入循环，并且第一次给 `sum` 加值了。当然从这里我们也可以看到，我们第一次给 `sum` 加了个 `0` 导致它的值没变，做了个无用功。这算不上是个 bug，但也确实写得不怎么样。所以最好把 `addedNum` 的初始值设成 1。

然后接下来就看着程序运行，每停一次，就看一下是不是正常，然后按回车再运行。

在第 10 和 11 次停的附近，我们会发现一些异常情况

```
...
sum =  36
addedNum =  9
Paused. Press enter to continue.
sum =  45
```

```
6  addedNum =  10
7  Paused. Press enter to continue.
8  result =  45
9  ans =  45
```

可以看到，在上面第 2 行，sum 是 36，第 3 行时，addedNum 已经递加为 9，然后程序暂停了一下。接着到第 5 行，sum 被加上了 addedNum 从而变成了 45，addedNum 也在第 9 行递加为 10。现在，只要 sum 再加一个 addedNum，就得到正确答案了。

暂停后，却直接出现了 result 的值，说明已经退出了循环。显然此时已经不满足循环的判断条件了。看一下源代码中第 5 行，

```
1      while currentNum  < upperLim     % 第5行
```

while 的判断条件是被加值小于最大值，而不能相等。这就是为什么到加不了最后的 10。

也许有人会说这个错误太简单，没必要这样大费周章。其实这不过是一个例子而已。但是在思维受阻时，与其对着屏幕干瞪眼，不如用这种方法。还有，如果以后我们需要调试大型的复杂程序，断点会非常的高效。

3.5 注意

断点并非没有坏处，最明显的就是如果你交 Coursework 时，如果不把刚才去掉的分号补上，或者多加的 input 删去的话就凉了，你的 Coursework 会被扣掉相当可观的分数。在实际的开发中，调试完程序也必须把所有断点去除。

此外，断点也只是我们的辅助工具，不能代替我们自己对程序的分析思考。程序一出错，不分析代码，就打上一大堆断点的行为，实际上就是依赖电脑的帮助进行 debug，放弃了对程序的思考。这样会使初学者的水平长期得不到提高。不说这些空话，就比如考试的时候写代码，如果没有自己纠错的能力，还能在试卷上打断点吗？

今天大课上，Manish（2018）有一句话说的很好，虽然记不得完全的原话，还是想分享给大家。

“*The aim of this module, is to training your thought and logic, but not how to use functions. Because we want you to be a programmer, rather than just a user.*”

4 运行错误

从上面对运行错误的原因的分析就能看出，运行错误没法像上面两个错误一样，那么容易被我们自己控制。关于防止用户瞎输入的方法的确有些复杂，也不是初学者应该过多关注度东西，我们大可到学 C 语言时再想它。

4.1 递归导致的内存不足

当然，比较常见而且能控制的一种运行错误，就是递归导致的内存不足。