# 多余的空语句

空语句一般来说是无害的，如：

ival = v1 + v2;; // 没什么问题，只是有点多余而已

但是如果 if 或else 后面跟了个额外的分号的话可能会造成不好的后果：

1. // 不好的情况: 循环体是while后面那条空语句
2. **while** (iter != svec.end()) ; // 循环体变成了后面的空语句
3. ++iter; // 这已经不是while循环体的一部分了

# switch语句的default标签后的语句在什么时候会跑到？

**没有任何一个case标签命中的时候**，会执行default后面的语句

# 有case标签命中的时候，会跑到default吗？

如果前面的case标签后面没有break，那就会。

# switch内部的变量定义

**原则**：

如果在某处一个带有初值的变量位于作用域之外，在另一处该变量位于作用域之内，则从前一处跳转到后一处的行为是非法的。

**解答**：

在switch内部，能跳过的是初始化，不能跳过的不是变量的定义。

**关于内存的分配：**

大部分编译器会在进入一个文件块之前就把所有的内存分配好，不需要运行到变量的定义语句才分配，所以只要是在一个作用域内部定义一个变量，则不管这段代码会不会运行到，编译器都会给你分配空间。而变量的初始化就不一样了，编译器会产生代码，因此不能跳过。

**需要注意的**：

一些类会调用默认初始化函数，因此即使在定义的时候没有赋给初值也是被初始化了的。

看下面的代码，注意！ **case** **true** 和 **case** **false** 属于同一作用域，因为每个 **case**  下面都没有自己的花括号：

* 1. **switch**（flag）{
  2. **case** **true**:
  3. // 初始化后，编译器才会给你产生代码，所以不能跳过
  4. // 因为程序的执行流程可能跳过下面的file\_name 和 ival的初始化语句，因此不合法
  5. string file\_name; // 错误: 这是一个隐式初始化；
  6. **int** ival = 0;  // 错误: 这是一个显示初始化；
  7. // ” 定义” 只是分配空间，这在进入文件块之前就分配好了，所以跳过”定义”也没问题
  8. **int** jval;  // 正确: 定义，非初始化，
  9. **break**;
  10. **case** **false**:
  11. // ok: jval 虽然在作用于内，但是它没有被初始化
  12. jval = next\_num();  // 正确: 给 jval 赋一个值
  13. **if** (file\_name.empty())  // file\_name 在作用于内，但是没有被初始化
  14. // ...
  15. }

看下面的代码，注意！ **case** **true** 和 **case** **false** 不属于同一作用域，因为每个 **case**  下面都有自己的花括号：

1. **case** **true**:
2. {
3. // 这个是对的，有了花括号，因此file\_name只属于case true标签下的局部变量
4. string file\_name = get\_file\_name();
5. // ...
6. }
7. **break**;
8. **case** **false**:
9. // 错误: file\_name变量 未定义（file\_name是 casetrue标签的局部变量）
10. **if** (file\_name.empty())

综上所述，在需要为某个case分支定义并初始化一个变量，我们应该加上花括号，保证后面所有的case标签都在变量的作用域之外

# 为什么可以在while循环内部定义变量？不会重复定义吗？

因为定义在while循环内部、while循环条件部分的变量每次迭代都经历从创建到销毁的过程。

# for循环的init-statement 有什么限制？

init-statement可以定义多个变量，但是只能有一个声明语句，因此所有变量的基础类型必须相同：

下面这样是错误的：

1. // 错误: init-statement 里有两个声明语句(int 和 long)
2. **for**(**int** i = 0, **long** j = 6; i < 6; ++i, --j)
3. cout<<"i:"<<i<<"  "<<"j: "<<j<<endl;

把long去掉，顺利编译通过：

1. **for**(**int** i = 0, j = 6; i < 6; ++i, --j)
2. cout<<"i:"<<i<<"  "<<"j: "<<j<<endl;

# for循环里的哪些部分可以省略？

for (init-statement; condition; expression)

statement

init-statement, condition, or expression都可以省略

# 用范围for语句将vector<int> v = {0,1,2,3,4,5,6,7,8,9};的值加倍

* 1. vector<**int**> v = {0,1,2,3,4,5,6,7,8,9};
  2. **for** (auto &r : v)  // for each element in v
  3. r \*= 2;  // double the value of each element in v

注意，要通过范围for语句修改容器的值，需要将其声明为引用。

# 为什么不能通过范围for语句来修改容器的元素？

上面用范围for语句将vector<int> v = {0,1,2,3,4,5,6,7,8,9};的值加倍的语句，用传统for循环等价于：

1. **for** (auto beg = v.begin(), end = v.end(); beg != end; ++beg)
2. {
3. auto &r = \*beg; // r must be a reference so we can change the element
4. r \*= 2;  // double the value of each element in v
5. }

从上面的代码可以看出，在范围for语句中，预存了end()的值，一旦在序列中添加（删除）元素，end函数的值可能就变得无效了。

# do while语句的语法形式是怎样的？

do

statementwhile (condition);

while后面的分号表示语句结束

# 用do while语句的时候需要注意什么？

do while语句不允许在条件不符定义变量，因为假设可以在条件部分定义变量，则变量的使用出现在定义之前，这显然是不对的

1. **do** {
2. // . . .
3. mumble(foo);
4. } **while** (**int** foo = get\_foo()); // 错误: 条件部分不允许定义变量！

# try catch 中，如何确定命中哪个异常？

这要看throw的是哪个异常了，来看两个例子：

在下面的代码中，因为抛出的是 **类型为 const char\* 的异常**，因此，当捕获该异常时，我们必须在 catch 块中使用 const char\*。

1. #include <iostream>
2. **using** **namespace** std;
4. **double** division(**int** a, **int** b)
5. {
6. **if**( b == 0 ) {
7. **throw** "Division by zero condition!";
8. }
9. **return** (a/b);
10. }
12. **int** main ()
13. {
14. **int** x = 50;
15. **int** y = 0;
16. **double** z = 0;
18. **try** {
19. z = division(x, y);
20. cout << z << endl;
21. }**catch** (**const** **char**\* msg) {
22. cerr << msg << endl;
23. }
24. **return** 0;
25. }

而在下面的代码中，我们抛出的是一个runtime\_error，所以在catch块中使用runtime\_error来捕获该异常

1. #include<iostream>
2. #include<stdexcept>
3. **using** **namespace** std;
5. **double** division(**int** a, **int** b)
6. {
7. **if**( b == 0 )  {
8. **throw** runtime\_error("Division by zero condition!");
9. }
10. **return** (a/b);
11. }
13. **int** main ()
14. {
15. **int** x = 50;
16. **int** y = 0;
17. **double** z = 0;
19. **try** {
20. z = division(x, y);
21. cout << z << endl;
22. }**catch** (**const** **char**\* msg) {
23. cerr << msg << endl;
24. }**catch** (runtime\_error err){
25. cout<<"I'm in runtime\_error\n";
26. cerr << err.what() << endl;
27. }
28. **return** 0;
29. }

# 在多层嵌套的try – catch调用中，处理流程是怎样的？

catch的匹配过程和函数调用链刚好相反，和递归是一个原理：

1. 先在首先抛出异常的函数中寻找有无匹配的异常，若有则处理，没有的话则终止该函数，在上一层中继续找；
2. 如果上一层还是没找到，再跳到更上一层；
3. 如果到最后还是没找到匹配的异常，程序会转到名为terminate的标准函数库，这个函数的会怎么处理异常情况，每个系统都不一样，一般是会导致程序异常退出。