案例参考《C 专家编程》 2.4 节 47 页

在 C 语言中，很容易写出一些能够轻松通过编译，但在运行时却产生一堆垃圾的代码。本节所描述的 Bug 就是一个非常好的例子。在任何语言中，都可能出现这样情况（比如除数为零），但很少有语言能像 C 语言那样提供如此丰富而意外的机会。

Sun 的 Pascal 编译器最近进行了“国际化”，也就是说进行了改进，使之能够（改进的成果之一）按照当地的日期格式在源代码列表中打印日期。比如在法国，日期可能以 Lundi 6 Avril 1992 这样的形式出现。其工作过程如下：编译器首先调用 stat()得到 UNIX 格式的源文件修正时间，然后调用 localtime()将其转换为 tm 结构，最后调用 strftime()函数，把 tm 结构转换为以当地日期格式表示的 ASCII 字符串。

令人不快的是，这里存在一个 Bug，症状就是表示日期的字符串被破坏。按照预想，打印出的日期应该如下：

```
lundi 6 Avril 1992
```

但结果却成了这么一种损坏了的形式：

```
Lui*7&' Y sxxdj @ ^F
```

这个函数仅有四条语句，而且在所有情况下传递给函数的参数都是正确的。下面是源代码，看看你能不能找出字符串破坏的问题所在。

```
/*  将源文件的 timestamp 转换为表示当地格式日期的字符串 */
char * localized_time(char * filename)
{
struct tm  *tm_ptr;
struct stat  stat_block;
char  buffer[120];

/* 获得源文件的 timestamp，格式为 time_t */
stat(filename, &stat_block);

/* 把 UNIX 的 time_t 转换为 tm 结构，里面保存当地时间 */
tm_ptr = localtime(&stat_block.st_mtime);

/* 把 tm 结构转换成以当地日期格式表示的字符串 */
strftime(buffer, sizeof(buffer), "%a %b %e %T %Y", tm_ptr);

return buffer;
}
```

变量的作用域和生命周期！！！！

看出来了吗？时间到！问题就出现在函数的最后一行，也就是返回 buffer 的那行。buffer 是一个自动分配内存的数组，是该函数的局部变量。当控制流离开声明自动变量(即局部变量）的范围时，自动变量便自动失效。这就意味着即使返回一个指向局部变量的指针（比如此例），当函数结束时，由于该变量已被销毁，谁也不知道这个指针所指向的地址的内容是什么。

在 C 语言中，自动变量在堆栈中分配内存。第 6 章会详细讲述这方面的内容。当包含自动变量的函数或代码块退出时，它们所占用的内存便被回收，它们的内容肯定会被下一个所调用的函数覆盖。这一切取决于堆栈中先前的自动变量位于何处，活动函数声明了什么变量，写入了什么内容等。原先自动变量地址的内容可能被立即覆盖，也可能稍后才被覆盖，这就是日期破坏问题难以被发现的原因。

1. 返回一个指向字符串常量的指针。例如：

```
char * func() { return "Only works for simple strings"; }
                                                    /* 只适用于简单的字符串*/
```

这是最简单的解决方案，但如果你需要计算字符串的内容，它就无能为力了，在本例中就是如此。如果字符串常量存储于只读内存区但以后需要改写它时，你也会有麻烦。

2. 使用全局声明的数组。例如：

```
char *fun() {
    ...
my_global_array[i] =
    ...
return my_golbal_array;
}
```

这适用于自己创建字符串的情况，也很简单易用。它的缺点在于任何人都有可能在任何时候修改这个全局数组，而且该函数的下一次调用也会覆盖该数组的内容。

3. 使用静态数组。例如：

```
char * func() {
static char buffer[20];
    ...
return buffer;
}
```

这就可以防止任何人修改这个数组。只有拥有指向该数组的指针的函数（通过参数传递给它）才能修改这个静态数组。但是，该函数的下一次调用将覆盖这个数组的内容，所以调用者必须在此之前使用或备份数组的内容。和全局数组一样，大型缓冲区如果闲置不用是非常浪费内存空间的。

4. 显式分配一些内存，保存返回的值。例如：

```
char * func() {
char * s = malloc(120);
...
return s;
}
```

这个方法具有静态数组的优点，而且在每次调用时都创建一个新的缓冲区，所以该函数以后的调用不会覆盖以前的返回值。它适用于多线程的代码（在某一时刻具有超过一个的活动线程的程序）。它的缺点在于程序员必须承担内存管理的责任。根据程序的复杂程度，这项任务可能很容易，也可能很复杂。如果内存尚在使用就释放或者出现“内存泄漏”（不再使用的内存未回收），就会产生令人难以置信的 Bug。人们非常容易忘记释放已分配的内存。

5. 也许最好的解决方案就是要求调用者分配内存来保存函数的返回值。为了提高安全性，调用者应该同时指定缓冲区的大小（就像标准库中 fgets()所要求的那样）。

```
void func( char * result, int size) {
    ...
strncpy(result, "That'd be in the data segment, Bob", size);
}

buffer = malloc(size);
func(buffer, size);
    ...
free(buffer);
```

如果程序员可以在同一代码块中同时进行“malloc”和“free”操作，内存管理是最为轻松的。这个解决方案就可以实现这一点。

为了避免“日期破坏”问题，注意 lint 程序会对下面这样最简单的例子发出警告：

```
return local_array;
```

建议使用方法 3 和方法 5