

第 7 章 编码中的假象

写小说，就希望每一页都能吸引读者，使读者激动、吃惊、悬念！决不能使读者感到厌烦。因此在每一页都要撒些胡椒粉，描述一些场景来吸引读者、如果小说写成：“罪犯走近乔并刺伤了他”，读者就会睡觉了。为了使读者感兴趣，就要使得当描述到乔听到身后“咚！咚！咚！”的脚步声时，读者也能感觉到乔是怎样的恐惧；当“咚！咚”的脚步声慢慢地越来越远的时候，读者也能感觉到乔的手在冒汗；当脚步声加速，罪犯朝乔逼近的时候，读者也能理解到乔是怎样的惊慌。最重要的是读者保持着悬念，乔能不能逃脱？……

在小说中使用惊奇和悬念很重要也很必要。但是如果把它们放到代码中，那就糟糕了。当写代码时，“情节”应该直观，以便别的程序员能预先清楚地知道将要发生的一切。如果用代码表述罪犯走近乔并刺伤了他，那么写成“罪犯走近乔并刺伤了他”最恰当了。该代码简短、清楚、并讲述了所发生的一切。但是由于某些原因，程序员拒绝写简捷清楚的代码，却极力主张使用具有技巧的、比较精炼的、异乎寻常的编码方法，最好不要这样。

但是直观的代码并不意味着是简单的代码，直观的代码可以使你沿着一条明确无奇的路径从 A 点到达 B 点。必要的时候直观的代码可能也很复杂。

因此，本章将考察导致产生不直观代码的编程风格。例子都很巧妙、有技巧，但是并非显而易见，当然，这些程序都会引起一些微妙的错误。

要注意到底引用了什么

下面的代码是上一章所给的 memchr 的无错版本：

```
void *memchr(void *pv, unsigned char ch, size_t size)
{
    unsigned char *pch = (unsigned char *)pv;
    while(size-- > 0)
    {
        if(*pch == ch)
            return(pch);
        pch++;
    }
    return(NULL);
}
```

大多数程序员玩弄的一种游戏是“我如何使得代码更快？”的游戏。这并不是坏游戏，但是正如我们从这本书所感到的那样：如果过份地热衷于这种游戏，那就是坏事。

例如如果在上面的例子上玩这个游戏的话，你就会问自己：“如何使循环加快？”只有三种可能的途径：删除范围检查、删除字符测试、或删除指针递增，好象删除哪一步骤都不行，但是如果愿意放弃传统的编码方式并进行大胆尝试的话是可以删除的。

看一下范围检查，之所以需要该检查仅仅是因为：当在存储器的头 size 个字节内没有找到要找的字符 ch 时，就要返回 NULL。要删除该检查，只要简单地保证总可以找到 ch 字符就可以了。这可以通过下面的方法来实现：在被查找的存储区域后面的第一个字节上存放字符 ch。这样，若待查存储区域内无字符此时，就可以找到后存入的这个 ch 字符：

```
void* memchr(void *pv, unsigned char ch, size_t size)
{
    unsigned char *pch = (unsigned char *)pv;
    unsigned char *pchPlant;
    unsigned char chSave;
    /* pchPlant 指向要被查寻的存储区域后面的第一个字节
     * 将 ch 存储在 pchPlant 所指的字节内来保证 memchr 肯定能挂到 Ch
     */
    pchPlant = pch+size;
    chSave = *pchPlant;
    *pchPlant = ch;
    while(*pch != ch)
        pch++;
    *pchPlant = chSave;
    return((pch == pchPlant)?NULL : pch);
}
```

巧妙吗？正确吗？通过用 ch 覆盖 pchPlant 指向的字符，可以保证 memchr 总能找到 ch，这样就可以删除范围检查，使循环的速度加倍。但是，这样坚挺、可靠吗？

这个 memchr 的新版看上去似乎坚挺，特别是它还仔细地把 pchPlant 原来所指的要被覆盖的字符保存起来，但是 memchr 的这个版本还是有问题。对于初学者来讲，请考虑下面几点：

- 如果 pcPlant 指向只读存储器，那么在*pchPlant 处存放字符 ch 就不起作用，因此当在 size+1 范围内没有发现 ch 时，函数将返回无效指针。
- 如果 pchPlant 指向被映射到 I/O 的存储器，那么将 ch 存储在*pchPlant 处就难以预计会发生什么事情，从使得软盘停止（或开始）工作到工业机器人狂暴地挥舞焊枪都有可能。
- 如果 pch 指向 RAM 最后的 size 个字节，pch 和 size 都是合法的，但 pchPlant 将指向不存在的或是写保护的存储空间。将 ch 存储在*pchPlant 处就可能会引起存储故障，或是不做任何动作。此时如果在 size+1 个字符内没有找到字符 ch，函数就会失败。
- 如果 pchPlant 指向的是并行进程共享的数据，那么当一个进程在*pchPlant 处存储 ch 时，就可能错改另一个进程要引用的存储空间。

最后一点尤其会引起麻烦，因为有许多方式都可以引起系统瘫痪。如果你调用 memchr

来查寻已分配了的存储空间，却不料破坏了存储管理程序的某个数据结构，这将如何是好呢？如果并行进程是代码连接或中断处理之类的例程，那么最好不要调用存储管理程序，否则系统可能会瘫痪。如果调用 memchr 扫描全局数组并且步入了由另一个任务引用的交界变量，那又该如何呢？如果程序的两个实例要并行地查找共享数据时，那又会怎样呢？有很多情况都会使程序死掉。

当然，你还不能体验到 memchr 引起的微妙错误，因为只要不修改关键的存储区，它就会工作得很好。但像 memchr 这样的函数一旦引起了错误，要孤立这些错误就象在大海里捞针一样的困难。这是因为：执行 memchr 的进程工作得很好，而另一个进程却因为存储区损坏而崩溃，此时，就没有理由怀疑是 memchr 引起的。这样错误就很难发现。

现在你就知道了，为什么要买价值\$50,000 的电路仿真器了。因为它们记录从开始到崩溃前的每一个周期、每一条指令、和计算机引用的每一段数据。可能要花几天时间才能艰难地读完仿真器的输出，但是如果坚持而且不盲目地处理这些输出结果的话，应该能找到错误之所在。

早已有警句：不要引用不属于你的存储区。我们又何必如上例那样忍受痛苦绞尽脑汁呢？注意，“引用”意味着不仅要读而且要写。读未知的存储区可能不会和别的进程产生不可思议的相互作用，但是，如果引用了已保护的存储区、不存在的存储区、或者映射到 I/O 存储区的话，程序将会迅速死掉。

只引用属于你自己的存储空间

拿车钥匙的贼还是贼

很奇怪有些程序员，他们从不引用不属于他们自己的存储空间。但他们却觉得编写象下面 FreeWindowsTree 例程这样的代码是很正确的：

```
void FreeWindowsTree(windows *pwndRoot)
{
    if(pwndRoot != NULL)
    {
        window *pwnd;
        /* 释放 pwndRoot 的子窗口 ..... */
        for(pwnd = pwndRoot->pwndChild;pwnd != NULL;pwnd = pwnd->pwndSibling)
            FreeWindowTree(pwnd);
        if(pwndRoot->strWndTitle != NULL)
            FreeMemory(pwndRoot->strWndTitle);
        FreeMemory(pwndRoot);
    }
}
```

请看一下 for 循环,看出什么问题了吗?当 FreeWindowsTree 释放 pwndSibling 链表中的每个子窗口时,先释放 pwnd,然后 for 循环在控制赋值时又引用已释放的块:

```
pwnd = pwnd->pwndSibling;
```

但是一旦 pwnd 被释放,那么 pwnd->pwndSibling 的值是什么呢?当然是一堆垃圾。但是某些程序员并不接受这个事实,刚刚存储区还好好的,并且也没做什么影响它的事,它仍应该是有效的呀!也就是说,除了释放 pwnd 之外没做别的什么事情。

我从不明白为什么某些程序员会认为引用已经释放的存储区是允许的,这与你使用过去的钥匙进入曾经住过的公寓或开走曾经属于你的汽车又有什么区别呢?你之所以不能安全引用释放的存储空间是因为正如我们在第 3 章中讲的,存储管理程序可能已将这块释放的空间连到空闲链上了,或已将它用于别的私有信息了。

只有系统才能拥有空闲的存储区,程序员不能拥有

数据的权限

在你所阅读的程序设计手册中可能没有讲到这个问题,但是,在代码中的每一条数据都隐含地有一个与之相联系的读写权限。该权限没有明文出处,也没在声明变量时显式地给出,而是在设计子系统和函数的界面时隐含地声明的。

例如,实际上在调用某个函数的程序员和写这个函数的程序员之间有个隐式的约定:

假设我是调用者,你是被调用者,如果我向你传递一个指向输入的指针,那么你就同意将输入当作常量并且承诺不对其进行写操作。同样,如果我向你传递一个指向输出的指针,你就同意把它当作只写对象来处理并承诺不对其进行读操作。最后,无论指针指向输入还是指向输出,你都同意严格限制对保存这些输出的存储空间的引用。

回过来说,我这个调用者同意把只读输出当作常量并已承诺不对它们进行写操作。此外,还同意严格限制对保存这些输出空间的引用。

换句话说:“你不要搞乱我的事情,我也不搞乱你的事情。”要牢记:任何时候,只要你违反了隐含的读写权限,那么就冒着中断代码的危险,因为编写这些代码的程序员坚信每个程序员都应遵守这些约定。调用象 memchr 这样的函数程序员不应担心在一些特殊的情况下,memchr 会运转异常。

仅取所需

上一章,我们给出了 UnsToStr 函数的一种实现方法,它如下所示:

```
/* UnsToStr——将无符号值转换为字符串 */  
void UnsToStr(unsigned u,char *str)  
{  
    char *strStart = str;  
    do
```

```

        *str++ = (u % 10) + '0';
    while((u/=10)>0);
    *str = '\0';
    ReverseStr(strStart);
}

```

上面的代码是 `UnsToStr` 的直接实现，但是，有些程序员觉得这样做法不舒服，因为代码以反向顺序导出数字，却要建立正向顺序的字符串。因此需要调用 `ReverseStr` 来重排数字的顺序。这样似乎很浪费。如果你打算以反向顺序导出数字，为什么不建立反向顺序的字符串从而可以取消对 `ReverseStr` 的调用呢？为什么不可以这样呢：

```

void UnsToStr(unsigned u,char *str)
{
    char *pch;
    /* u 超出范围吗？使用 UlongToStr... */
    ASSERT(u<= 65536);
    /* 将每一位数字自后向前存储
     * 字符串足够大以便能存储 u 的最大可能值
     */
    pch = &str[5];
    *pch = '\0';
    do
        *--pch = u%10 + '0';
    while((u/=10)>0);
    strcpy(str,pch);
}

```

某些程序员对上面的代码感到很满意，因为它更有效并且更容易理解。它之所以更有效是因为 `strcpy` 比 `ReverseStr` 更快，特别是对于那些可把“调用”生成为内联指令的编译程序来说就更是这样。代码之所以更容易理解是因为 C 程序员对 `strcpy` 要更熟悉一些。当程序员见到 `ReverseStr` 时，就好象听到他们的朋友住进医院的消息一样，都会迟疑一下。

这又说明什么呢？如果 `UnsToStr` 真是那么完美，我说这些干嘛！当然，它并不完美，事实上 `UnsToStr` 有个严重的缺陷。

告诉我，`str` 所指的存储空间多大？你并不知道。对于 C 语言接口程序来说，这并不罕见。在调用者和实现者之间有个无言的原则，这就是 `str` 将指向足够大的存储区来存放 `u` 的正文表示。`UnsToStr` 假定 `str` 指向转换 `u` 的最大可能值所需的足够存储空间，但 `u` 并不常是最大值。因而，调用者写出如下代码：

```

DisplayScore()
{
    char strScore[3];

```

```

        UnsToStr(UserScore, strScore);
    }

```

由于 UserScore 不会产生小于三个字符（两位数字加一位空字符）的字符串，因此，程序员将 strScore 定义为三个字符的数组是完全合理的，然而，UnsToStr 假设 strScore 是 6 个字符的数组，并且破坏了存储区内 strScore 后面的三个字节。在上面的例子中，如果所用的机器具有向下增长的栈，那么 UnsToStr 将损坏结构的后向指针，或损坏返回给 DisplayScore 调用者的地址，或对两者都有损坏。这时，机器很可能瘫痪，所以应该注意这个问题。但是，如果 strScore，不只是局部变量的话，可能不会注意到 UnsToStr 破坏了存储器中跟在 strScore 后面的变量。

我相信会有程序员争辩：将 strScore 定义成恰好保存最大字符串是有风险的。这的确有风险，但仅当程序员写出象 UnsToStr 最后版本一样的代码之时。事实上，没有必要象上面那样施展伎俩：因为可以通过在局部缓冲区中建立字符串，安全有效地实现 UnsToStr，然后将最终产物复制到 str：

```

void UnsToStr(unsigned u, char *str)
{
    char strDigits[6];
    char *pch;
    /* u 超出范围了吗？使用 UlongToStr... */
    ASSERT(u <= 65536);
    pch = &strDigits[6];
    *pch = '\0';
    do
        *--pch = u % 10 + '0';
    while((u/=10)>0);
    strcpy(str, pch);
}

```

需要记住的是：除非 str 已在别处定义，象 str 那样的指针不会指向被用作工作空间缓冲的存储区。为了提高效率，象 str 这样的指针是通过引用传递输出而不是通过值传递输出的。

指向输出的指针不是指向工作空间缓冲区的指针

私有数据自己管

当然，还会有程序员认为在 UnsToStr 中调用 strcpy 效率太低。毕竟 UnToStr 是要创建一个输出串。那么当你通过返回一个指向你已经建立的字符串的指针来节省一些循环时，为什么不将它拷贝到另一个缓冲区呢？

```

char *strFromUns(unsigned u)
{
    static char strDigits = "?????"; /* 5 个字符+ '\0' */
    char *pch;
    /* u 超出范围了吗？使用 UlongToStr ... */
    ASSERT(u<=65535);
    /* 将每位数字自后向前存储在 strDigits 中 */
    pch = &strDigits[5];
    ASSERT(*pch == '\0');
    do
        *--pch = u%10 + '0';
    while((u/=10)>0);
    return(pch);
}

```

上面的代码与上一节所给的代码几乎相同，所不同的只是将 `strDigits` 声明为静态的，这样即使在 `strFromUns` 返回以后分配给 `strDigits` 的存储区仍然保存。

设想一下：如果要实现将两个无符号值转换成字符串的函数，你就会写成：

```

strHighScore = strFromUns(HighScore);
...
strThisScore = strFromUns(Score);

```

这会有什么错误吗？你能看出调用 `strFromUns` 来转换 `Score` 就损坏了 `strHighScore` 所指的字符串吗？

你可能争辩说错误在上面的这个代码中，而不是在 `strFromUns` 中。但是要记住我们在第 5 章中讲的：函数能正确地工作是不够的，它们还必须能防上程序员产生明显的错误。由于你和我都知道某些程序员将要犯类似上述的错误，我总可以证实 `strFromUns` 有个界面错。

即使程序员已经意识到 `strFromUns` 的字符串很脆弱，也会情不自禁地引入错误。假设他们调用了 `strFromUns` 然后调用另一个函数，而他们并不知道这个函数也调用 `strFromUns`，因此破坏了他们的字符串。或者，假设有多条代码的执行线，其中一条执行线调用 `strFromUns`，那么就有可能冲掉另外一个执行线仍在使用的字符串。

即使上述问题与 `strFromUns` 本身的问题比起来是次要的，但是，这些问题肯定要出现，随着项目的发展，还可能是多次出现。因此，当你决定在你的某个函数中插入对 `strFromUns` 的调用时，你必须做到下列两点：

- 确保你的调用者（以及你的调用者的调用者等等）中没有任何一个正在使用由 `strFromUns` 返回的字符串。换句话说，你必须验证没有任何一个这样的函数在可能调用你的函数的调用链上，并假定 `strFromUns` 的私有缓冲区是被保护的。
- 确保你不调用任何调用 `strFromUns` 的函数以防损坏作仍需要的字符串。当然这就

意味着你不能调用那些直接、间接地调用了 `strFromUns` 的函数。

如果你插入一个对 `strFromUns` 的调用而又不进行上面的两项检查那么你就冒着引入错误的风险。但是，设想一下当程序员改正错误和增加新特征时遵守上面的两种情况该有多么困难。每一次改变对你的函数调用的调用链，或修改你的代码所调用的函数，这些维护人员都必须重新检验上面的两种情况。你认为他们会做到吗？很难。那些程序员甚至都没有认识到他们应该检验上述条件。毕竟，他们只做改出错误、重组代码和增加新特征；那么他们对函数 `strFromUns` 该做什么呢？他们可能从未用过，甚至从未见过这个函数。

正是由于这样的设计使得在维护程序时很容易引入错误，因此象 `strFromUns` 这样的函数可能一而再地引起错误。当然，当程序员要孤立 `strFromUns` 的错误时，错误并不在 `strFromUns` 内而是在不正确地使用 `strFromUns` 的代码内。因此只咒骂 `strFromUns` 并不能解决真正的问题。程序员要改正这种特殊的错误，随着时间的流逝在程序中不再使用 `strFromUns`。

不要利用静态（或全局）量存储区传递数据

全局量问题

上述 `strFromUns` 例子说明了当借助指向静态存储区的指针返回数据时，你将面临的危险。例子没有说明每当你向非局部的缓冲区传递数据时，也存在着同样的危险。你可以改写 `strFromUns` 使它能在全局缓冲区，甚至在永久缓冲区内建立数值串，（永久缓冲区一般在程序开始处利用 `malloc` 建立），但是情况没有任何变化，因为程序员仍能连续两次调用 `strFromUns`，并且第二次调用会破坏第一次调用返回的字符串。

因此，经验方法是：除非你有绝对的必要，否则，不要向全局缓冲区传递数据。不要忘记，静态局部缓冲区和全局缓冲区一样。

在设计函数过程中，当需要向缓冲区传递数据时，安全的方法是让调用者分配一个局部（非静态）的缓冲区。

如果能够迫使函数调用者提供一个指向输出缓冲区的指针，那么就可以避免全部问题。

函数的寄生虫

向公共缓冲区传递数据是危险的，但是假若比较小心并且运气很好的话可能会摆脱危险。但是，编写依赖于别的程序内部处理的寄生函数不仅危险，而且也是不负责任的：如果宿主函数有了更改，寄生函数也就毁坏了。

我所知道的寄生函数的最好例子，来自一个广泛推广、移植的 `FORTH` 程序设计语言的标准程序。在 70 年代末 80 年代初，`FIG` (`FORTH Interest Group`) 试图通过提供公共的 `FORTH-77` 标准程序来刺激人们对 `FORTH` 语言的兴趣。那些 `FORTH` 程序定义了三个标准函数：`FILL`，它以字节为单位填充存储块；`CMOVE`，它用“头到头”的算法拷贝存储；`<CMOVE`，它用“尾到尾”的算法拷贝存储；`CMOVE` 和 `<CMOVE` 特意叫做“头到头”和“尾到尾”，这样，

当程序员需要拷贝覆盖的存储块时，知道该使用哪个函数)。

在那些 FORTH 程序中，CMOVE 是用优化的汇编语言写的，但为了具有可移植性，FILL 用 FORTH 语言本身编写。CMOVE 的代码和我们设计的代码差不多，转换为 C 语言如下：

```
/* CMOVE —— 用头到头的移动来转移存储 */
void CMOVE (byte *pbFrom,byte *pbTo,size_t size)
{
    while(size-- > 0 )
        *pbTo++ = *pbFrom++;
}
```

而 FILL 的实现却令人惊异：

```
/* FILL 填充某一存储域 */
void FILL (byte *pb,size_t size,byte b)
{
    if(size>0)
    {
        *pb = b;
        CMOVE(pb,pb+1,size-1);
    }
}
```

FILL 调用 CMOVE 来实现它的功能，在弄清它是怎样工作之前，有点费解。这个实现方法要么是“巧妙的”，要么就是“粗劣的”，就看你怎么看了。如果你认为 FILL 是巧妙的，那么考虑一下：FORTH 可能需要将 CMOVE 实现为一个头到头的转移。但是，如果为了提高效率。而改写 CMOVE，用 long（长字）而不是 byte（字节）来移动存储，那又将如何呢？在我看来，上面的 FILL 程序是粗劣的，而不是巧妙的。

但是假设你不打算改变 CMOVE。你甚至可在 CMOVE 内写上注释：FILL 依赖于它的内部处理，以警告其他的程序员，但这样只解决了一半问题。

假定你做过用于控制四自由度的工厂机器人的控制代码，每个自由度都有 256 个位置。只要用映射到 I/O 存储器的四个字节，就可以设计出这个机器人，其中每个字节控制一个自由度。

为了保存一个自由度的位置，要将 0 到 255 之间的某个值写到存储器的相应位置内。为了检索一个自由度的当前位置（特别是当某个自由度的位置要移到一个新位置时尤其有用）要从相应的存储位置上读出相应值。

如果想将四个自由度复位到初始点（0，0，0，0）。从理论上讲，可以写成如下代码：

```
FILL(pbRobotDevice,4,0);    /* 将机器人复位到初始点状态 */
```

按前述方式的 FILL 定义，该代码是不能正常工作的。FILL 将给第一个自由度写上 0，其它三个自由度因之填入垃圾，导致机器人处于紊乱状态。为什么会这样呢？如果查看一下 FILL 的设计，就可以明白，它是将以前存储的字节拷贝到当前字节来实现填充的。但是，

当 FILL 读出第一个字节时，希望它是 0。可是由于读到的应是第一个自由度的当前位置，因此这个位置可能不是 0。因为在存储 0 到试图将该位置值读回之间这短短的若干分之一秒内，第一自由度可能还没有移动到位置 0 处。这个位置值可能是任意值，因此将第二自由度发送到某个不确定点。类似地，第三和第四自由度也将被发送到未知的地方。

为了使 FILL 正确地工作就必须保证 FILL 可以从存储器中读到刚写进存储的那个值。可是，对于映射到 I/O、ROM、被保护的存储、或空存储库的存储区来说，无法保证上面的要求。

我的观点是 FILL 之所以有错误，是因为它剽窃了别的函数的私有细节并滥用了这些知识。在除了 RAM 之外的其它形式的存储器上，FILL 都不能正确地工作，这还是次要问题，更主要的是它又一次证明了在任何时候只要不编写直观代码，就是自找麻烦！

不要写寄生函数

断言使程序员更加诚实

假如 CMOVE 用了断言来验证其参数的合法性（即源存储空间在被拷贝到目标存储空间之前不被破坏），那么编写 FILL 的程序员在第一次测试该代码时就碰到了断言。这样程序员就有两个选择：要么用合理的算法重写 FILL，要么从 CMOVE 中删除相应的断言。幸运的是几乎没有程序员为了使得糟糕的 FILL 程序能够工作而删除 CMOVE 的断言。

断言还能阻止 FreeWindowTree 中的自由存储空间错误进入项目的原版源代码。通过使用断言和第 3 章中的调试代码，当用有子窗口的窗口第一次测试 FreeWindowTree 时，就会引发相应的断言。除非碰上了想通过扣除断言来“排除”断言失败的个别程序员，大多数程序员为了消除断言失败都会修改 FreeWindowTree 本身。

物非所用

用一把螺丝刀来播开油漆罐的盖子，然后又用这把螺丝刀来搅拌油漆，这是家庭维护中最熟悉的举动之一，我有一大堆各种颜色的螺丝刀可以来证明这一点。但是，当人们知道这样会糟蹋螺丝刀，不应该这样做时，为什么还要用螺丝刀来搅拌油漆呢？原因就在于，之所以这样做是因为当时这样很方便，而且能够解决问题。当然，有一些程序设计手段也很方便并保证能工作，但是就象那把螺丝刀一样，它们没有发挥它们本来的作用。

例如下面的代码，它将比较的结果作为计算表达式的一部分

```
unsigned atou(char *str); /* atoi 的无符号版本 */
/* atoi 将 ASCII 字符串转换为整数值 */
int atoi(char *str)
{
    /* str 的格式为“[空白][+/-]数字” */
    while(isspace(*str))
```

```

        str++;
        if(*str == ' - ')
            return (-(int)atou(str+1));
        return ((int)atou(str + ( *str == ' + ' )));
    }

```

上面的代码把(*str == ' + ')的测试结果加到字符串指针上，从而跳过可选的前导 ' + ' 号。因为按 ANSI 标准，任何关系操作的结果或者是 0 或者是 1，因此可以这样写代码。但是某些程序员可能没有意识到，ANSI 标准不是一本告诉你可以做什么和不可以做什么的规则书。你可以写出形式合格的代码，但却违反了它的意图。因此，不要因为允许写出如上那样的代码，就意味着应该写出这种代码。

但是，真正的问题与代码毫无关系，而与程序员的看法紧密相关。如果程序员觉得在计算表达式中使用逻辑求值非常好的话，他还会愿意采用什么别的安全性未知的捷径吗？

不要滥用程序设计语言

标准也会改变

当发行 FORTH-83 标准时，一些 FORTH 程序员发现他们的代码与其不一致了。原因是：在 FORTH-77 标准中，布尔值的结果定义为 0 和 1，由于各种原因，在 FORTH-83 标准中，将布尔值的结果改为 0 和 -1。当然，这种改变只破坏了那些依赖于“真”为 1 的代码。

并不只是 FORTH 程序员遇到了这种情况。

在 70 年代末到 80 年代初，UCSD Pascal 非常普及，如果你在微机上使用 Pascal 的话，多半是 UCSD Pascal。但是后来，许多 UCSD Pascal 程序员得到了编译程序的新版本，并且发现在新编译程序上，他们的代码不能工作。原因又是：编译程序的编写者，由于各种原因改变了“真”的值从而破坏了依赖于原先值的所有程序。

谁又能说得准将来的 C 标准不会发生更改呢？即使 C 不变改，C++或者别的派生语言是否会发生变化？

程序设计语言综合症

那些不知道 C 语言代码是如何转换为机器代码的程序员，经常试图通过使用简炼的 C 语句来提高机器代码的质量。他们认为，如果使用最少量的 C 语句，那么就应该得到最少量的机器代码。在 C 代码数量和相应的机器代码数量之间存在着一定的关系，但当把这种关系应用到单行代码时，这种关系便不适用了。

你还记得第 6 章的 uCycleCheckBox 函数吗？

```

unsigned uCycleCheckBox(unsigned uCur)
{
    return ((uCur<=1)?(uCur?0:1) : (uCur == 4)?2 : (uCur +1));
}

```

```
}
```

uCycleCheckBox 可以说是简炼 C 代码，但是正如我指出的那样，它产生了很糟的机器代码。再看上一节中给出的返回语句：

```
return((int)atou(str + (*str == ' + ')));
```

如果你使用的是优化得很好的编译程序，并且你的目标机不用任何分支指令即可生成 0/1 的结果，那么把比较的结果加到指针上，这条语句将产生相当好的代码。如果不具备上面描述的条件，编译程序很可能要作比较在内部将其扩展为 `?:` 操作，生成的代码就好象你写了如下所示的 C 代码一样：

```
return ((int)atou(str+((*str == ' + ')?1:0)));
```

由于“`?:`”操作只不过是化了妆的 if-else 语句，因此所得到的代码可能比下面明显、直观的代码更坏：

```
if(*str == ' + ') /* 跳过可选的 ' + ' 号 */  
    str++;  
return ((int)atou(str));
```

当然还有其它的方法来优化上面的代码。我曾经见到这样一些情况：程序员将一个两行的 if 语句用“`||`”操作符“改进”成一行：

```
if( (*sdr != ' + ') || str++ ) /* 跳过可选择的 ' + ' 号 */  
    return ((int)atou(str));
```

这样的代码之所以可以工作是因为 C 语言具有短路求值规则，但是将代码放在一行上并不能保证比使用 if 语句产生更好的机器代码 如果编译程序产生的 0 或 1 有副作用的话，使用“`||`”甚至会得到更坏的代码。

需要记住这个简单规则：把“`||`”用于逻辑表达式，把“`?:`”用于条件表达式，把 if 用于条件语句。这并不是说把它们混合起来就好，而往往出于使代码高效和可维护。

我的观点是：如果你总是使用稀奇古怪的表达式，以便把 C 代码尽量写在源代码的一行上，从而达到最好的瑜伽状态的话，你很可能患有可怕的“一行清”疾病（也称为程序设计语言综合症）。那么你就要做个深呼吸，反复地提醒自己：“多行源代码可能产生效率高的机器代码。多行源代码可能产生效率高的机器代码……。”

紧凑的 C 代码并不能保证得到高效的机器代码

切勿傲慢轻率

世上最令人厌烦的书，就是那些由专家撰写的、其内容充满了没有必要的技术术语的书。他们不说“该错误可能使你的系统暂停或失败”，而是说“这样的程序设计缺陷可能导致丧失对系统的控制或引起系统终止”。他们还用象“公理化程序验证”和“缺陷分类”这样的术语，好象程序员每天都要用到些术语似的。啧！啧！啧！啧！这些作者将他们要说明的信息隐藏在含糊难懂的术语中，这不仅没有帮助读者，反而使读者更糊涂了。不只是

书作者这么做，有一些程序员也热衷于编写含糊难懂的代码，他们认为只有代码含糊不清才能给人留下深刻的印象。

例如，看看下面的函数是怎样工作的：

```
void *memmove(void *pvTo,void *pv From,size_t size)
{
    byte *pbTo = (byte *)pvTo;
    byte *pbFrom = (byte *)pvFrom;
    ((pbTo < pbFrom)?(tailmove:headmove)(pbTo,pbFrom,size)
    return (pvTo);
}
```

如果我将它改写如下，该函数是否更好理解呢？

```
void *memmove(void *pvTo,void *pvFrom,size_t size)
{
    byte *pbTo = (byte *)pvTo;
    byte *pbFrom = (byte *) pvFrom;
    if(pvTo < pbFrom)
        tailmove(pbTo,pbFrom,size);
    else
        headmove(pbTo,pbFrom,size);
    return (pbTo);
}
```

第一个例子看起来不象合法的 C 语言程序，但实际上是。比较一下是很有好处的，第一个例子编译以后产生的代码比第二个例子所产生的代码要少得多。尽管如此，有多少程序员能理解第一个函数是怎样工作呢？如果他们必须维护该代码那又将如何呢？如果你写了正确的代码，但是没有人能够理解，那又有什么意义呢？如果不打算让别人看懂，你甚至可以用手工优化的汇编语言来编写这个函数。

下面的代码是使许多程序员费解的另一个例子：

```
while(expression)
{
    int i = 33;
    char str[20];
    ... 其他代码 ...
}
```

请迅速回答，是每一次循环都要初始化 i，还是仅仅第一次进入循环时对 i 进行初始化呢？你能不用思考就知道正确答案吗？如果你不能肯定，说明你训练有素，因为即使是专家级 C 程序员通常也要在脑子里浏览一下 C 语言的规则才能回答这个问题。

如果稍微修改一下，成为如下所示的代码。

```

While(expression)
{
    int i;
    char str[20];
    i = 33;
    ... 其它代码 ...
}

```

你对每次通过循环都要将 i 置为 33 还有什么疑问吗？在你的小组中还有程序员对此表示怀疑吗？当然没有。

和小说作家不一样，他们只有一类读者，而程序员却有两类读者：使用代码的用户和必须对代码进行更新的程序维护人员。程序员经常忘记这一点。我知道，忘掉用户的程序员并不多，但是根据我这些年读的程序来推测，程序员似乎忘记了他们的第二类读者：程序维护人员。

应该编写可维护的代码这一观点并不新奇，程序员知道应该编写这样的代码。可是，他们总是没有认识到，他们虽然整天编写可维护的代码，但是如果他们使用只有 C 语言专业人员才能理解的语言，那么这些代码实际上是不可维护的。根据定义，可维护的代码应该是维护人员可以很容易地理解并且在修改时不会引入错误的代码。不管怎样，程序维护人员一般都是该项目的新手而不是专家。

因此，当你考虑你的读者时，一定还要考虑到程序维护人员。下一次当你又想写下面的代码时：

```
strncpy(strDay,&" SunMonTueWedThuFriSat "[day*3],3);
```

你可以制止自己，并且以一种不让读者吃惊又很好理解的方式编写代码：

```

static char strDayNames[] = " SunMonTueWedThuFriSat ";
...
strncpy(strDay,&strDayNames[day*3],3);

```

为一般水平的程序员编写代码

谁在维护程序

在 Microsoft 公司，每个程序员编写新代码的数量，与他对所从事研制的产品内部情况的熟悉程度成正比，对产品比较熟悉的程序员，编写新代码的是多一些，而较少进行维护性的程序设计。当然，如果对项目了解很少那么就要花大量时间来阅读别人写的代码、修改别人的错误、对于已有特征作少量的局部性的增补。直观地看，这种安排很有意义。如果你不知道系统是怎样写的，那你就不能给系统增加重要的功能。

概括起来，这种安排的结果就是：一般来说，有经验的程序员编写出代码，新手维护代码。我并不是说不应该这样安排，这种安排是实用的而且就是这么作的。但是，只有在

有经验的程序员认识到，他们有责任使得他们所编写的代码，能够被程序维护人员和程序设计新手维护，这时这种安排才能行得通。

不要错误的理解我的意思，我并不是说你应该写初级的 C 程序以使程序设计新手能够理解你的代码，这样就和总是编写专家级 C 代码一样愚蠢了。我要说的是，当你能用普通程度语言表达清楚时，就应该避免使用困难的或神秘的 C。如果你的代码很容易理解，那么新手在维护时就不易引入错误，你也不必总是向他们解释代码是如何如何地工作了。

小结

我们已经考察了一些有争议的编码实践，其中大部分初看上去都很好。但是，正如我们已经看到的，看一遍，甚至看五遍，你可能都没有警觉到那些巧妙代码产生的微妙的副作用。因此建议：如果你发现自己编写的代码用了较多技巧，那么停止编写代码并寻找别的解决方法。你的“技巧”也许很好，但是如果你确实觉得它有些费解，那就是你的直觉在告诉你，情况不妙。听凭你的直觉，如果你认为你的代码确有技巧的话，那么，这实际上是在对自己讲，尽管这个算法应该直观而实际并非如此，但它却产生了正确的结果。那么这个算法的错误同样也会不明显。

因此，编写直观的代码才是真正的聪明人。

要点：

- 如果你要用到的数据不是你自己所有的，那怕是临时的，也不要对其执行写操作。尽管你可能认为读数据总是安全的，但是要记住，从映射到 I/O 的存储区读数据，可能会对硬件造成危害。
- 每当释放了存储区人们还想引用它，但是要克制自己这么做。引用自由存储区极易引起错误。
- 为了提高效率，向全局缓冲区或静态缓冲传递数据也是很吸引人的，但是这是一条充满风险的捷径。假若你写了一个函数，用来创建只给调用函数使用的数据，那么就将数据返回给调用函数，或保证不意外地更改这个数据。
- 不要编写依赖支持函数的某个特殊实现的函数。我们已经看到，FILL 例程不该象给出的那样调用 CMOVE，这种写法只能作为坏程序设计的例子。
- 在进行程序设计的时候，要按照程序设计语言原来的本意清楚、准确地编写代码。避免使用有疑问的程序设计惯用语，即使语言标准恰好能保证它工作，也不要使用。请记住，标准也在改变。
- 如果能用 C 语言有效地表示某个概念，那么类似地，相应的机器代码也应该是有效的。逻辑上讲似乎应该是这样，可是事实上并非如此。因此在你将多行 C 代码压缩为一行代码之前，一定要弄清楚经过这样的更改以后，能否保证得到更好的机器代码。
- 最后，不要象律师写合同那样来编写代码。如果一般水平的程序员不能阅读和理

解你的代码，那就说明你的代码太复杂了，使用简单一点的语言。

练习：

- 1) C 程序设计员经常修改传递给函数的参数。为什么这种做法没有违反输入数据的写权限呢？
- 2) 前面已经介绍了有关下面 `strFromUns` 函数的主要缺陷（复习一下，它将非保护缓冲区里的数据返回），除此之外，`strDigits` 的声明方式还有什么错误吗？

```
char *strFromUns(unsigned u)
{
    static char strDigits = "?????"; /* 串长为 5 个 char + '0' */
    char *pch;
    /* u 超出范围吗？使用 UlongToStr */
    ASSERT( u <= 65535);
    /* 将每一位数字自后向前存储在 strDigits 中 */
    pch = &strDigits[5];
    ASSERT(*pch == '\0');
    do
        *--pch = u%10 + '0';
    while((u/= 10)>0);
    return (pch);
}
```

- 3) 在我阅读一本杂志上的代码时，我注意到了有这样一个函数，它用 `memset` 函数将三个局部变量置为 0，如下所示：

```
void DoSomething(...)
{
    int i;
    int j;
    int k;
    memset(&k,0,3*sizeof(int));
    .....
```

这样的代码在某些编译程序上可以运行，但是为什么要避免使用这种技巧呢？

- 4) 尽管计算机在只读存储器中存有部分操作系统的程序，但假如为了避免不必要的内部操作，你绕过了系统界面而直接调用 ROM 过程，为什么这又是有风险的呢？
- 5) 传统上，C 允许程序员向函数传递参数的个数比函数期望接收的参数个数少。某些程序员利用这个特征来优化调用，这些调用并不要求全部参数。例如：

...

```

DoOperation(opNegAcc);      /* 不需要传递 val */
...
void DoOperation(operation op,int val)
{
    switch(op)
    {
        case opNegAcc:
            accumulator = - accumulator;
            break;
        case opAddVal:
            accumulator +=val;
            break;
        ...
    }
}

```

尽管这样优化仍能工作但为什么要避免这么做呢？

- 6) 下面的断言是正确的，但是，为什么要改写它呢？

```
ASSERT((f&1)==f);
```

- 7) 请研究使用以下代码的 memmove 的另一版本：

```
((pbTo<pbFrom)?tailmove:headmove)(pbTo,pbFrom,size);
```

怎样改写 memmove 使它既保持上面代码的效率，又更容易理解？

- 8) 下面的汇编语言代码给出了调用函数的常用捷径。如果你使用了这段代码，为什么说自找麻烦呢？

```

        move r0,#PRINTER
        call print+4
...
print:  move r0,#DISPLAY          ;(4 字节指令)
...                                ; ro == 设备标识符 ID

```

- 9) 下面的汇编语它给出了另一个技巧。这个代码与上个练习的代码具有同样的问题，除了上述问题外，为什么你还应该避免使用这个技巧？

```

        instClear R0 = 0x36A2      ;“clear ro”是 16 进制指令
...
        call print+2              ;输出到打印机
...
print:  move r0,#instClearR0       ;(4 字节指令)
        comp r0,#0                ;0 == PRINTER , non-0 == DISPLAY
...

```