

第 6 章 风险事业

假如将一程序员置于悬崖边，给他绳子和滑翔机，他会怎样从悬崖上下来呢？是沿绳子爬下来呢？还是乘滑翔机呢？还是干脆直接跳下来呢？是沿绳子爬下来还是使用滑翔机我们说不太准，但可以肯定，他不会跳下来，因为那太危险了。可是当程序员有几种可能的实现方案时，他们却经常只考虑空间和速度，而完全忽视了风险性。如果程序员处于这样的悬崖边而又忽视了风险性，只考虑选择到达崖底最有效的途径的话，结果又将如何呢？

程序员忽视风险性，至少有两个原因：

一是因为他们盲目地认为，不管他们怎样实现编码，都不会有错误。没有任何程序员会说：“我准备编写快速排序程序，并打算在程序中有三个错误。”程序员并没有打算出错，而后来错误出现了，他们也并不特别吃惊。

我认为程序员忽视风险性的第二个原因也是主要原因：在于从来没有人教他们这样去问问题：“该设计有多大的风险性？该实现有多大的风险性？有没有更安全的方法来写这个表达式？能否测试一下该设计？”要想问出这些问题，首先必须从思想上放弃这样的观点：不管作出哪种选择，最后总能得到无错代码。即使该观点是正确的，可是什么时候能得到无错代码呢？是由于使用安全的编码，在几天或几周之后就可以得到无错代码呢？还是由于忽视了风险性，出现很多错误而需要经过数月的调试和修改之后才能得到无错代码呢？

因此本章将讨论在某些普通的编码实践中所存在的一些风险，以及如何做才能减少甚至消除这些风险。

long 的位域有多长

美国国家标准协会（ANSI）委员会查看了运行在众多平台上的各种 C 语言。他们发现：尽管人们认为 C 语言是可移植语言，但实际上并非如此。不仅不同系统的标准库不同，而且预处理程序和语言本身在许多重要方面也不相同。ANSI 委员会对大多数问题进行了标准化，使程序员可以写出可移植的代码，但是，ANSI 标准忽视了一个非常重要的方面，它没有定义象 char、int 和 long 这样一些内部数据类型。ANSI 标准将这些重要的实现细节留给编译程序的研制者来决定，标准本身并没有具体定义这些类型。

例如，某一个 ANSI 标准的编译程序可能具有 32 位的 int 和 char。它们在缺省状态下是有符号的；而另一个 ANSI 标准的编译程序可能有 16 位的 int 和 char，缺省状态下是无符号的。尽管如此不同，然而，这两个编译程序可能都严格附合 ANSI 标准。

请看下面的代码：

```
char ch;
.....

ch = 0xff;
if (ch == 0xff )
.....
```

我的问题是 if 语句中的表达式求值为真还是为假呢？

正确的回答是：不知道。因为这完全依赖于编译程序。如果在缺省时字符是无符号的，则表达式肯定为真。但对于字符为有符号的编译程序而言，如 80x86 和 680x0 的编译程序，则每次测试都会失败，这是由 C 语言的扩充规则决定的。

在上面的代码中，字符 ch 与整型数 0xff 进行比较。根据 C 语言的扩充规则，编译程序必须首先将 ch 转换为整型 int，两者类型一致后再进行比较。关键在于：如果 ch 是有符号的，则在转换中要进行符号位扩充，其值将从 0xff 扩充为 0xffff（假设 int 是 16 位）。这就是测试失败的原因。

上面是为证明作者观点而设计的例子。读者可能会说，那不是一段有实际意义的代码。但是，在下面的常用代码中也存在着同样的问题。

```
char * pch;  
.....  
if ( *pch == 0xff )  
.....
```

在该定义中，char 类型不唯一，位域不正确。例如，以下位域的值域是多少？

```
int reg:3;
```

仍然是不知道。即使将 reg 定义为整型 int，这就隐含了它是有符号的，但根据所使用的不同编译程序，reg 既可以是有符号的，也可以是无符号的。如果要使 reg 明确地成为有符号的整型或无符号的整型，必须使用 signed int 或 unsigned int。

short, int, long 究竟有多大，ANSI 标准没有给出。而将其留给编译程序的研制者来决定。

ANSI 委员会成员并非对错误定义数据类型的问题视而不见。实际上，他们考查了大量的 C 语言实现并得出结论：由于各编译程序之间的类型定义是如此之不同，以至于定义严格的标准将会使大量现存代码无效。而这恰恰违背了他们的一个指导原则：“现存代码是非常重要的”。他们的目的并不是要建立更好的语言，而是给现存的语言制定标准，只要有可能，他们就要保护现存的代码。

对类型进行约束也将违背委员会的另外一个指导原则：“保持 C 语言的活力，即使不能保证它具有可移植性，也要使其速度快。”因此，如果实现者感到有符号字符对于给定的机器来说更有效、那么就使用有符号字符。同样，根据硬件实现者可以将 int 选择为 16 位、32 位或别的位数、这就是说，在缺省状态下，用户并不知道是具有有符号的位域还是无符号的位域。

内部类型在其规格说明中存在着一个不足之处，在今后升级或改变编译程序时、或移到新的目标环境时、或与其他单位共享代码时、甚至在改变工作并发现所用编译程序的规则全部改变时，这个不足就会体现出来。

这并不意味着用户不能安全使用内部类型、只要用户不对 ANSI 标准没有明确说明的类型再作假设。用户就可以安全使用内部类型。

例如，你可以用易变的 char 数据类型，只要它能提供 0 到 127 的值，这是有符号字符

和无符号字符域的交集。所以，当代码写为：

```
char * strcpy( char * pchTo, char * pchFrom )
{
    char *pchStart = pchTo;
    while(( *pchTo ++ = *pchFrom++ )!= ' \0 ' )
        NULL;
    Return( pchStart );
}
```

时，它在任何编译程序上都可以工作，因为没有对域作假定。而以下代码就不可以：

```
/* strcmp -- 比较两个字符串
 *
 * 如果 strLeft<strRight, 返回一个负值
 * 如果 strLeft==strRight, 返回 0
 * 如果 strLeft>strRight, 返回一个正值
 */
int strcmp( const char *strLeft, const char *strRight )
{
    for( NULL; *strLeft == *strRight; strLeft ++ ,strRight ++ )
    {
        if( strLeft == ' \0 ' )      /* 是否与最后的结束字符相匹配？*/
            return (0) ;
    }
    return ( (*strLeft<*strRight) ?-1:1 ) ;
}
```

这段代码，由于最后一行的比较操作而失去了可移植性。只要用户使用了“<”操作符或其它要用有符号信息的操作符，就迫使编译程序产生不可移植的代码。修改 strcmp 很容易，只须声明 strLeft 和 strRight 为无符号字符指针，或直接将其填在比较式中：

```
( *( unsigned char *) strLeft < *(unsigned char *)strRight )
```

记住一个原则不要在表达式中使用“简单的”字符。由于位域也有同样的问题，因此也有一个类似的原则：任何时候都不要使用“简单的”位域。

如果仔细阅读分析 ANSI 标准，就可以导出可移植类型集的定义。这些可移植类型可在多个编译程序上以多种数制工作。

char	0 to 127
signed char	-127 to127 (not -128)
unsigned char	0 to 255
	大小未定，但不小于 8 个字位

short	-32767 to 32767 (not -32768)
signed short	-32767 to 32767
unsigned short	0 to 65535
	大小未定，但不小于 16 个字位
long	-2147483647 to 2147483647 (not -2147483648)
signed long	-2147483647 to 2147483647
unsigned long	0 to 4294967295
	大小未定，但不小于 32 个字位
int i:n	0 to $2^{(n-1)}-1$
signed int i:n	$-(2^{(n-1)}-1)$ to $2^{(n-1)}-1$
unsigned int i:n	0 to $2^{(n)}-1$
	大小未定，至少有 n 个字位

可移植类型最值得注意之处是：它们只考虑了三种最通用的数制：壹的补码、贰的补码和有符号的数值。

现在我们可以不必为写可移植代码担心了。处理该问题就象人们为自己厨房操作台挑选贴面瓷砖一样，大多数人都愿意挑选自己喜欢的，将来的房屋买主也能容忍的贴面瓷砖，这样到时候就不必为了卖房屋来拆除、更换贴面瓷砖了。读者也应以同样的方式来考虑可移植代码，在大多数情况下，写可移植性代码与写非可移植性代码一样容易。为了避免将来的重复劳动，最好写可移植代码。

使用有严格定义的数据类型

尽量用可移植的数据类型

有些程序员可能认为使用可移植的类型比使用“自然的”类型效率低。例如，假定 int 类型对目标硬件其物理字长是最有效的。这就意味着这种“自然的”位数可能大于 16 位，所保持的值可能大于 32767。

现在假定用户的编译程序使用的是 32 位的 int，且题目要求 0 至 40,000 的值域。那么，是考虑到机器可以在 int 内有效地处理 40,000 个值而使用 int 呢，还是坚持使用可移植类型，而用 long 代替 int 呢？

答案是如果机器使用的是 32 位 int。那么也可以使用 32 位 long，这两者产生的代码即使不相同也很相似（事实证明是如此），因此要使用 long。用户即便担心在将来必须支持的机器上使用 long 效率可能会低一些，也应该坚持使用可移植类型。

数据上溢或下溢

有这样一些代码，表面看起来很正确。但是由于实现上存在着微妙的问题，执行却失败了，这是最严重的错误。“简单字符”就是这种性质的错误。下面的代码也具有这样的错误，这段代码用作初始化标准 `tolower` 宏的查寻表。

```
char chToLower[ UCHAR_MAX+1 ];
void BuildToLowerTable( void ) /* ASCII 版本*/
{
    unsigned char ch;
    /* 首先将每个字符置为它自己 */
    for (ch=0; ch <= UCHAR_MAX;ch++)
        chToLower[ch] = ch;
    /* 现将小写字母放进大写字母的槽子里 */
    for( ch = 'A'; ch <= 'Z'; ch++ )
        chToLower[ch] = ch + 'a' - 'A';
}
.....

#define tolower(ch) (chToLower[(unsigned char)(ch)])
```

尽管代码看上去很可靠，实际上 `BuildToLowerTable` 很可能使系统挂起来。看一下第一个循环，什么时候 `ch` 大于 `UCHAR_MAX` 呢？如果你认为“从来也不会”，那就对了。如果你不这样认为，请看下面的解释。

假设 `ch` 等于 `UCHAR_MAX`，那么循环语句理应执行最后一次了。但是就在最后测试之前，`ch` 增加为 `UCHAR_MAX+1`，这将引起 `ch` 上溢为 0。因此，`ch` 将总是小于等于 `UCHAR_MAX`，机器将进行无限的循环。

通过查看代码，这个问题还不明显吗？

变量也可能下溢，那将会造成同样的困境。下面是实现 `memchr` 函数的一段代码。它的功能是通过查寻存储块，来找到第一次出现的某个字符。如果在存储块中找到了该字符，则返回指向该字符的指针，否则，返回空指针。象上面的 `BuildToLowerTable` 一样，`memchr` 的代码看上去似乎是正确的，实际上却是错误的。

```
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 );
}

```

循环什么时候终止？只有当 size 小于 0 时，循环才会终止。可是 size 会小于 0 吗？不会，因为 size 是无符号值，当它为 0 时，表达式--size 将使其下溢而成为类型 size_t 定义的最大无符号位。

这种下溢错误比 BuldToLowerTable 中的错误更严重。假如，memchr 在存储块中找到了字符，它将正确地工作，即使没有找到字符，它也不致使系统悬挂起来。而坚持查下去，直到在某处找到了这个字符并返回指向该字符的指针为止。然而，在某些应用中也可能产生非常严重的错误。

我们希望编译程序能对“简单字符”错误和上面两种错误发出警告。但是几乎没有任何编译程序对这些问题给出警告。因此，在编译程序的销售商说有更好的编译代码生成器之前，程序员将依靠自己来发现上溢和下溢错误。

但是，如果用户按照本书第 4 章的建议逐条跟踪代码，那么这三种错误就都能发现。用户将会发现，*pch 在与 0xff 比较之前已经转换为 0xffff，ch 上溢为 0，size 下溢为 0xffff。由于这些错误太微妙，可能用户花几小时时间仔细阅读代码，也不会发现上溢错，但是如果查看在调试状态下该程序的数据流，就能很容易地发现这些错误。

经常反问：“这个变量表达式会上溢或下溢吗？”

量体裁衣

在下面的代码中还可以看到另一个常见的上溢例子，它将整数转换为相应的 ASCII 表示：

```

void IntToStr( int i, char *str )
{
    char *strDigits;
    if( i < 0 )
    {
        *str++ = ' - ';
        i = -i;    /* 把 i 变成正值 */
    }
    /* 反序导出每一位数值 */
    strDigits = str;
    do
        *str++ = i%10 + '0';
    while( (i/=10) > 0 );
}

```

```

        *str= ' /0 ' ;
        ReverseStr( strDigits );    /* 将数字的次序转为正序 */
    }

```

若该代码在二进制补码机器上运行，当 i 等于最小的负数（例如，16 位机器的 -32768）时就会出现这个问题。原因在于表达式 $i = -i$ 中的 $-i$ 上；即上溢超出了 `int` 类型的范围。然而，真正的错误在于程序员实现代码的方式上：程序员没有完全按照他自己的设计来实现代码，而只是近似实现了他的设计。

在设计中要求：“如果 i 是负的，加入一个负号，然后将 i 的无符号数值部分转换成 ASCII。”而上面的代码并没有这么做。它实际执行了：“如果 i 是负的，加入一个负号，然后将 i 的正值也就是带符号的数值部分转换为 ASCII。”就是这个有符号的数字引起了所有的麻烦。如果完全根据算法并使用无符号数，代码会执行得很好。可以将上述代码分为两个函数，这样做十分有用。

```

void IntToStr( int i, char *str )
{
    if( i < 0 )
    {
        *str++ = ' - ' ;
        i = -i;
    }
    UnsToStr(( unsigned )i, str );
}

void UnsToStr( unsigned u, char *str )
{
    char * strStart = str;
    do
        *str++ = (u%10) + '0' ;
    while(( u/=10 )>0 );
    *str= '\0' ;
    ReverseStr( strStart );
}

```

在上面的代码中， i 也要取负，这与前面的例子相同，为什么它就可以正常工作呢？这是因为：如果 i 是最小负数 -32768，二进制补码形式表示为 0x8000，然后通过将所有位倒装（即 0 变 1）再加 1 来取负，从而得到 $-i$ 为 0x8000，若为有符号数，则表示 -32768，若为无符号数，则表示 32768。按定义，由二进制补码表示的任意数，通过将其每一位倒装再加 1，可以得到该数的负值。因此 0x8000 表示的是最小负数 -32768 的负值，即 32768，因此应解释为无符号数。

至此,代码正确了,但并不美观。上面的代码容易让人产生错觉。根据可移植类型。-32768并不是有效的可移植整型值,因此通过在 IntToStr 中适当的位置插入断言,就可以排除所有的混乱。

```
void IntToStr( int i, char *str )
{
    /* i 是否超出范围? 使用 LongToStr ... */
    ASSERT( i>=-32768 && i<= 32767 );
```

通过使用上面的断言,既可以避免与某种数制相关的问题,又可以促使其他的程序员编写更容易移植的代码。不管怎样,都要记住:

尽可能精确地实现设计, 近似地实现设计就可能出错

每个函数只完成它自己的任务

我曾经考察了字符窗口代码,这是为 Microsoft 基于字符的 DOS 应用而设计的类窗口库,我之所以这样做,是因为使用该库的 PC-Word 和 PC-Works 两个小组都感到该库代码庞大,执行缓慢,而且不稳定。我刚开始考察该代码时就发现了程序员并没有按照他们原来的设计实现代码,而这恰恰违反了编写无错代码的另一条指导原则。

首先介绍一下背景。

字符窗口的的基本设计非常简单:用户将视频显示看作一些窗口的集合,每个窗口可以有它自己的子窗口。设计一个表示整个显示的根窗口,它可以具有菜单框、下拉式菜单、应用文档窗口、对话等子窗口。每一个子窗口又可能具有其自己的子窗口。例如,对话窗口可能具有为 OK 键和 Cancel 键而设立子窗口,还可能包含一个列表框窗口,其中又可能具有用作滚动条的子窗口。

为了表示窗口层次结构,字符窗口使用了二叉树结构。一个分支指向称为 "children" 的子窗口;另一个分支指向有相同父母的称为 "siblings" 的窗口:

```
typedef struct WINDOW
{
    struct WINDOW * pwndChild;    /* 如果没有孩子,则为 NULL */
    struct WINDOW * pwindSibling; /* 如果没有兄弟姐妹,则为 NULL */
    char *strWndTitle;
    /* ... */
} window;                        /* 命名: wnd, *pwnd */
```

可以查阅任何算法书籍,从中找到处理二叉树的有效方法。可是当我阅读了字符窗口代码中有关向树中插入子窗口的代码时,我有点吃惊了,该代码如下所示:

```
/* 指向最顶层窗口列表,例如象菜单框和主文件窗口
*/
```

```

static window * pwndRootChildren = NULL;
void AddChild( window * pwndParent, window * pwndNewBorn )
{
    /* 新窗口可能只有子窗口 ... */
    ASSERT( pwndNewBorn->pwndSibling == NULL );
    if( pwndParent == NULL )
    {
        /* 将窗口加入到顶层根列表 */
        pwndNewBorn->pwndSibling = pwndRootChildren;
        pwndRootChildren = pwndNewBorn;
    }
    else
    {
        /* 如果是父母的第一个孩子，那么开始一个链，
         * 否则加到现存兄弟链的末尾处
         */
        if( pwndParent -> pwndChild == NULL )
            pwndParent -> pwndChild = pwndNewBorn;
        else
        {
            window *pwnd = pwndParent -> pwndChild;
            while( pwnd -> pwndSibling != NULL )
                pwnd = pwnd -> pwndSibling;
            pwnd -> pwndSibling = pwndNewBorn;
        }
    }
}

```

尽管实际上是将窗口结构设计为二叉树结构，但并不是按这种结构实现的。由于根窗口（表示整个显示的窗口）没有同级窗口也没有标题，而且也不会有移动、隐藏、删除的操作，在 window 结构中只有指向菜单框和应用子窗口的 pwndChild 字段才是有意义的。因此有人认为声明完整 window 结构是浪费，而用指向顶层窗口的简单指针 pwndRootChildren 来代替 wndRoot 结构。

用一个指针代替 wndRoot，可能会在数据空间内节省一些字节，可是在代码空间内的代价巨大。象 AddChild 这样的例用就不得不处理两种不同的数据结构：根层窗口树的链表和窗口树本身，而不是处理简单的二叉树。当各个例程以窗口指针作为参数时，情况更糟，它不得不检查表示显示窗口的专用 NULL 指针，而这种情况很多。难怪 PC-Word 和 PC-Works 两个小组担心代码庞大。

我提出 AddChild 问题并不想讨论设计问题，而是想指出这种实现方法至少违反了编写无错代码指导原则中的三条原则。前两条原则前面叙述过了：“不要接受具有特殊意义的参数”，例如 NULL 指针；“按照设计来实现而不能近似地实现。”第三个新的原则是：努力使每个函数一次就完成任务。

如何理解这条新原则呢？假如 AddChild 有一个任务，要在现有窗口中增加子窗口，而上面的代码具有三个单独的插入过程。常识告诉我们如果三段代码而不是一段代码来完成一个任务，则很可能有错。比如做脑外科手术，如果一次能做好，那就不能做三次，写代码也是这个道理。如果写个函数，发现是多次做一个任务，就要停下来问问自己，是否能用一段代码来完成这个任务。

有时也需要这样的函数，它执行的功能要做两次，例如第 2 章的 memset 快速版本（请回顾一下，它具有两个独立的填充循环，一个快速的，一个慢速的）。如果能够肯定自己理由充分的话，也可以打破这个原则。

改进 AddChild 的第一步非常容易，删掉“优化”，按照原来的设计实现。用一个指向窗口结构的指针 pwndDisplay 来代替 pwndRootChildren，窗口结构表示屏幕显示，将 pwndDisplay 传递给 AddChild，而不是将 NULL 传递给 AddChild，就不需要有处理根窗口的专用代码：

```
/* 在程序初始化过程中分配根层窗口
 * pwndDisplay 将被设置为指向根层窗口
 */
window* pwndDisplay = NULL;
void AddChild( window *pwndParent, window *pwndNewBorn )
{
    /* 新窗口可能只有子窗口 */
    ASSERT( pwndNewBorn -> pwndSibling == NULL );
    /* 如果是父母的第一个孩子，那么开始一个链，
     * 否则加到现存兄弟链的末尾处
     */
    if( pwndParent -> pwndChild == NULL)
        pwndParent -> pwndChild = pwndNewBorn;
    else
    {
        window * pwnd = pwndParent -> pwndChild;
        while( pwnd -> pwndSibling != NULL )
            pwnd = pwnd -> pwndSibling;
        pwnd -> pwndSibling = pwndNewBorn;
    }
}
```

上面的代码不仅改进了 AddChild (和其它每个与树结构相适应的函数), 而且将原来版本中根窗口要反向插入的错误也更正了。

一个“任务”应一次完成

为什么窗口是层次结构的？

为什么需要有层次结构的窗口, 一个最主要的原因就是为了简化象移动、隐蔽、删除窗口这样一些操作。如果移动对话框, OK 和 Cancel 框还在原来的位置吗? 或者说, 如果将某个窗口隐蔽起来, 它的子窗口还可见吗? 显然, 这并不是所期望的。通过支持子窗口, 可以说: “移动这个窗口” 并且所有与之相关的窗口都将紧随着移动。

避免无关紧要的 if、&& 和 “但是” 分支

AddChild 最后这个版本比前面的版本要好一些, 但它仍然是由两段代码来完成同一任务的。if 语句的出现标志着可能有同一任务两次重复执行的情况发生, 尽管两次执行的方式不同。但 if 语句的出现应在人们的头脑中敲起警钟。确实, 有一些场合需要合法使用 if 语句来执行一些有条件的操作, 但大多数情况下, 这是草率设计粗心实现的结果。因为将充满例外情况的设计组织在一起比停止并导出不含例外情况的模型要容易得多。

例如, 在窗口结构中, 可以有两种方法遍历兄弟链: 一种方法是进入指向窗口结构的循环。从一个窗口步进到另一窗口, 即是以窗口为中心的算法; 另一种方法是进入指向指针的循环, 从一个指针步进到另一个指针, 即是以指针为中心的算法。上述 AddChild 实现的是以窗口为中心的算法。

由于当前版本的 AddChild 要扫描兄弟链列表来附加新的窗口, 因此对第一个指针要有特殊的处理。附加窗口实际上是在前一个窗口的 “下一窗口指针” 域内建立一个指向新窗口的指针。要注意, 从一个窗口步进到另一窗口, 前一个窗口的指针可能是兄弟指针, 也可能是父子指针。特殊处理能够确保修改正确的指针。

但是如果使用以指针为中心的模型, 则总是指向 “下一窗口指针”。而 “下一窗口指针” 是父子指针还是兄弟指针无关紧要, 因此没有什么特别的处理。为了便于理解, 请看以下代码。

```
void AddChild(window* pwndParent, window* pwndNewBorn )
{
    window **ppwindNext;
    /* 新窗口可能没有兄弟窗口 ... */
    ASSERT( pwndNewBorn -> pwndSibling == NULL );
    /* 使用以指针为中心的算法
     * 设置 ppwindNext 指向 pwndParent -> pwndChild
     * 因为 pwndParent -> pwndChild 是链中第一个 “下一个兄弟指针”
```

```

    */
    ppwndNext = &pwdParent->pwdChild;
    while( *ppwndNext != NULL )
        ppwndNext = &( *ppwndNext )->pwdSibling;
    *ppwndNext = pwdNewBorn;
}

```

上面的代码是经典哑头（dummy header）链表插入算法的变形，这个算法因为不需要任何特殊代码来处理空列表而著名。

不必担心上面的 AddChild 代码会违反以前提出的原则，即准确地实现设计而不能近似地实现设计。该代码没有按人们通常想的那样实现设计，但它确实真正地实现了设计。就好象我们观察眼镜片一样，这个镜片是凸的还是凹的呢？两个答案可能都是对的，这就要看是怎样去观察它了。对于 AddChild 来说，使用以指针为中心的算法可以不必为特殊情况编写代码。

不用担心最后版本 AddChild 的效率。它产生的代码将比以前任一版本产生的代码少得多。甚至循环部分的代码也可能比以前版本产生的代码好。不要因为加上了*和&，而认为循环要比以前复杂，其实不然（编译一下这两个版本，看一下结果）。

避免无关紧要地 if 语句

“?:” 运算符也是一种 if 语句

C 程序设计员必须正视：“?:” 运算符不过是 if-else 语句的另外一种表示形式。因为我们经常看到一些程序员判断时只用“?:”，从不明确使用 if-else 语句。在 Excel 的对话框处理代码中就有这样的例子，它包含下面的函数，这个函数的功能是确定检查框的下个状态：

```

/* uCycleCheckBox —— 返回对话框地下个状态
 *
 * 给出了当前设置，uCur 返回对话框所应具有的下个状态
 * 这既处理只有 0 和 1 又处理在 2, 3, 4, 2 …… 三个状态
 * 中循环的三状态检查框
 */
unsigned uCycleCheckBox(unsigned uCur)
{
    return( (uCur<=1)?(1-uCur):(uCur==4)?2:(uCur+1) );
}

```

我曾经和那些不想两次嵌套使用“?:”编写 uCycleCheckBox 的程序员一起工作过，可是当要在下面显式使用 if 语句的代码上写上他们的名字之前，尽管由大多数编译程序但非最好的编译程序产生的这两个函数的代码几乎相同，他们还是转向了 COBOL。

```

unsigned uCycleCheckBox(unsigned uCur)
{
    unsigned uRet;
    if(uCur <= 1)
    {
        if(uCur != 0)    /* 处理 0, 1, 0 …… 循环 */
            uRet = 0;
        else
            uRet = 1;
    }
    else
    {
        if(uCur == 4)    /* 处理 2, 3, 4, 2 …… 循环 */
            uRet = 2;
        else
            uRet = uCur + 1;
    }
    return(uRet)
}

```

尽管有些编译程序确实为嵌套“?:”版本产生了比较好的代码，但是实际上并好不了多少。如果用户的目标机上已经有了效率很高的编译程序，不妨试一试下面的代码。做个比较。

```

unsigned uCycleCheckBox( unsigned uCur )
{
    unsigned uRet;
    if( uCur <= 1 )
    {
        uRet = 0;        /* 处理 0, 1, 0 …… 循环 */
        if( uCur == 0 )
            uRet = 1;
    }
    else
    {
        cuRet = 2; /* 处理 2, 3, 4, 2 …… 循环 */
        if( uCur != 4 )
            uRet = uCur + 1;
    }
    return( uRet );
}

```

```
}
```

仔细看看 uCycleCheckBox 的三个版本，尽管知道它们可能要做什么，但并不一目了然。如果输入 3 将返回几？你能很容易得出答案是 4 吗？我可不能。这些具有两个简单循环的函数，实现方法十分清楚，但却难以掌握。

使用“?:”运算符所存在的问题是：由于它很简单，容易使用，看起来好象是产生高效代码的理想方法，因此程序员就不再寻找更好的解决方法了。更严重的是，程序员会将 if 版本转换为“?:”版本来获得“较好”的解决方法，而实际上“?:”版本并不好。这就好象想通过将 100 美元的钞票换成 10,000 美分来获得更多的钱一样，钱并没有增多，却变重了，使用起来不方便了。如果程序员将时间花在寻求替代算法上，而不是花在以某个稍微不同的方式实现同一个算法上，那么可能会提出下面这种更直接的实现方法：

```
unsigned uCycleCheckBox( unsigned uCur )
{
    ASSERT( uCur >= 0 && uCur <= 4 );
    If( uCur == 1 )          /* 重新开始第一个循环？*/
        return( 0 );
    if( uCur == 4 )          /* 重新开始第二个循环？*/
        return( 2 );
    return( uCur + 1 );      /* 这时没有任何特殊处理 */
}
```

或许有人会提出下面这种列表解决方法：

```
unsigned uCycleCheckBox( unsigned uCur )
{
    static const unsigned uNextState[] = {1,0,3,4,2 };
    ASSERT( uCur >= 0 && uCur <= 4 );
    return ( uNextState[uCur] );
}
```

通过避免使用“?:”可以得到更好的算法，而不仅仅是看上去好一些的方法。列表版本与以前的版本相比较，哪个更好理解呢？哪个产生的代码最好呢？哪个更容易第一次实现就正确呢？你应该从中领悟到一些道理。

避免使用嵌套的“?:”运算符

消除代码的冗余性

在实现中，有时得支持特殊情况。为了避免特殊情况遍及整个函数，我们把处理特殊情况的代码独立出来。这样，维护人员在以后的维护中就不会无意识地将其遗漏而导致出现错误。

前面已经给出了实现 IntToStr 的两个版本，下面给出的是经常出现在 C 程序设计教程中的 IntToStr 代码（在教程中称为 itoa）：

```
void IntToStr( int i, char *str )
{
    int iOriginal = i;
    char* pch;
    if( iOriginal < 0 )
        i = -i;          /* 把 i 变成正值 */
    /* 反导出字符串 */
    pch = str;
    do
        *pch++ = i % 10 + '0';
    while(( i/=10 ) > 0 );
    if( iOriginal < 0 )    /* 不要忘记负号 */
        *pch++ = ' - ';
    *pch = '\0';
    ReverseStr(str);      /* 将子字符串次序从逆序转为正序 */
}
```

请注意，代码中有两个 if 语句，并且测试的是同一种特殊状况。既然可以很容易将两个代码体写在一个 if 语句内。我们就要问“为什么不那么做呢？”

有时，重复测试没有发生在 if 语句内，而发生在 for 或 while 循环语句的条件内。例如下面给出另一种实现 memchr 函数的代码：

```
void* memchr( void *pv, unsigned char ch, size_t size )
{
    unsigned char *pch = (unsigned char *)pv;
    unsigned char *pchEnd = pch + size;
    while( pch<pchEnd && *pch != ch )
        pch ++;
    return( ( pch < pchEnd ) ? pch : NULL );
}
```

再与下面的 memchr 版本做一下比较：

```
void* memchr( void *pv, unsigned char ch, size_t size )
{
    unsigned char *pch = ( unsigned char *)pv;
    unsigned char *pchEnd = pch + size;
    while( pch < pchEnd )
    {
```

```

        if( *pch == ch )
            return ( pch );
        pch ++ ;
    }
    return( NULL );
}

```

哪个看上去更好一些呢？第一个版本要比较 pch 和 pchEnd 两次，第二个版本只比较一次。哪个更清楚呢？更重要的是哪个第一次执行就可能正确呢？

由于第二个版本只在 while 条件中进行块范围检查，所以它更容易理解并且准确地实现了函数的功能。第一个版本的唯一长处是当需要将程序打印出来时，可以节省一些纸。

每种特殊情况只能处理一次

不返回太危险

上面给出的 memchr 两个版本正确吗？你是否看出这两个版本具有同样一个细小错误？提示一下：当 pv 指向存储区的最后 72 个字节，并且 size 也是 72 时，memchr 将要查找存储区的什么范围呢？如果答案是“存储区的全部范围，反复不断地查找。”那么你的回答就是对的。由于在程序中使用了有风险的惯用语，memchr 陷入了无限的循环之中。

有风险的惯用语就是这样一些短语或表达式，它们看上去似乎能够正确地工作，但实际上在某些特殊场合下，它们并不能正确执行。C 语言就是具有这样一些惯用语的语言，最好的办法是：无论什么时候，只要有可能就尽量避免使用这些惯用语。在 memchr 中有风险的惯用语是：

```

    pchEnd = pch + size;
    while( pch < pchEnd )
    ...

```

其中，pchEnd 指向存储区中被查找的最后一个字符的下一个位置，因此它可用在 while 表达式中。程序员觉得这样使用很方便，如果所指的存储位置存在的话，则该程序会工作得很好，但是如果恰好查找到存储器的结尾处，那么所指的位置就不存在了（一个例外情况是：如果使用 ANSI C，总可以计算出某个数组之外的第一个元素的地址。ANSI C 支持这个性能）。

作为改正错误的第一步尝试，将上面的代码改写为如下所示：

```

    pchEnd = pch + size - 1;
    while ( pch <= pchEnd )
    ...

```

但是，这还不能正确工作。还记得前面讲过的 BuildToLowerTable 中 UCHAR_MAX 上溢错误吗？这里也有同样的错误。现在 pchEnd 可能指向一个合法的存储位置，但是，由于每一次 pch 增加到 pchEnd + 1 时都要上溢，因此循环将不会终止。

当你可用指针也可用计数器时，使用计数器作为控制表达式是覆盖一个范围的安全方法：

```
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 );
}
```

上面的代码不仅正确，而且所产生的代码可能比前面各个版本产生的代码都要好，因为它不必初始化 pchEnd。由于 size 在减 1 之前必须先复制，以便与 0 进行比较，所以人们通常认为 size-- 版本比 pchEnd 版本要大一些并且要慢一些。可是，实际上对于许多编译程序来说，size-- 版本恰巧更快些、小些。这取决于编译程序是如何使用寄存器的。对于 80x86 编译程序言，还要取决于所使用的是哪种存储模型。不管怎样，在大小和速度方面，size-- 版本与 pchEnd 版本的差别很小，并不值得引起注意。

下面给出另一个惯用语，实际上在前面已经提过了。有些程序员可能会极力主张重写循环表达式，用 --size 代替 size--：

```
while ( --size >= 0 )
.....
```

这种变化的合理一面是：上面这种表达式不产生比以前的表达式更坏的代码。在某些情况下，可能会产生稍好一点的代码。但它的唯一问题是：如果盲目奉行的话，错误将会象苍蝇见到牲畜一样向代码突袭而来。

为什么呢？

如果 size 是无符号值（象 memchr 中的一样），根据定义，将总是大于或等于 0，循环将永远执行下去，因此表达式不能正常工作。

如果 size 是有符号数，表达式也不能正常工作。如 size 是 int 类型并且以最小的负值 INT_MIN 进入循环，它先被减 1，那么就会产生下溢，使得循环执行大量的次数。

相反，无论怎样声明 size 都能使 “size-- > 0” 正确工作。这是个小小的、但又很重要的差别。

程序员使用 “-- size > 0” 的唯一原因是想加快速度。让我们仔细看一下，如果真的存在速度问题，那么进行这种改进就好象用指甲刀剪草坪一样，可以这么做，但没有什么效果。如果不存在速度问题，那为什么又要冒这样的风险呢？这就好象没有必要让草坪的所有草叶都一样长，没有必要让每行代码效率都最优一样，要认识到最重要的是总体效果。

在某些程序员看来，放弃任何可能获得效率的机会似乎近似于犯罪。但是，当读完本书以后，你会得到这样的思想即使效率可能会稍稍低一点，也要使用安全的设计和实现来系统地减少风险性。用户不应该注意是否在某个地方又增加了一些循环，而应集中注意力来看是否在试图节省某些循环时而偶然引入了错误。用投资方面的一句术语来说就是：赢利并不能证明冒险是正确的。

使用移位操作来实现乘、除、求模 2 的幂是另外一种有风险的惯用语。它属于“浪费效率”类。例如，第 2 章给出的 memset 快速版本有如下几行代码：

```
pb = ( byte * )longfill(( long * )pb, 1, size/4 );
size = size % 4
```

可以肯定，有一些程序员在读到上面的代码时会想：“效率多低”。他们可能会将除操作和求模操作写成如下的形式：

```
pb = ( byte * )longfill(( long * )pb, 1, size >> 2 );
size = size & 3
```

移位操作比除法或求模要快，这在大多数机器上是对的。但是，象用 2 的幂去除或去求模无符号值（如 size）的这样的操作，已经优化好了，即使商用计算机也是如此，没有必要再手工优化这些无符号表达式。

那么，有符号表达式又将怎样呢？显式的优化是否值得呢？既值得也不值得。

假定有如下的有符号表达式：

```
midpoint = ( upper + lower ) / 2;
```

当有符号数的值为负值时，将其移位与进行有符号除法所得的结果不同，因此二进制补码的编译程序不将除法优化为移位。如果我们知道上面表达式中的 upper + lower 总是正值，就可以采用移位将表达式改写成如下所示，这个代码要好一些：

```
midpoint = ( unsigned )( upper + lower ) >> 1
```

因此，优化有符号表达式是值得的。可是，移位是否是最好的方法呢？不是。下面代码所示的强制转换方法同样也很好，并且比移位法要安全得多。请在编译程序上试一下：

```
midpoint = ( unsigned )( upper + lower )/2;
```

上面的代码不是告诉编译程序要做什么，而是将需要进行优化的信息传递给编译程序。通过告诉编译程序所求得的结果是无符号的，来调知它可以进行移位。现在来比较一下两种优化，那个更容易理解？那个更具有可移植性？那个更可能在第一次执行就正确呢？

多年来，我发现了许多由于程序员使用移位来进行有符号值的除法，而有符号值又不能确保为正值而引起的错误；发现了许多方向移错了的移位错误；发现了许多移位位数错了的移位错误；甚至发现了由于不小心将表达式“a=b+c/4”转换为“a=b+c>>2”而引入的优先级错。但我却不曾发现过以键入‘/’和‘4’来实现除以 4 时会发生错误。

C 语言还有许多其它的有风险的惯用语。有个最好的方法来找到自己经常使用的有风险的惯用语，这就是检查以前出现的每一个错误，再问一下自己：“怎样来避免这些错误？”然后建立个人的风险惯用语表从而避免使用这些惯用语。

避免使用有风险的语言惯用语

不要过高地估计代价

1984 年, 当 Apple 研制出 Macintosh 的时候, Microsoft 公司是少数几个采用 Macintosh 产品的公司之一。很显然, 采用其它公司的产品对 Microsoft 公司来说, 既有益也有害。采用了 Macintosh 就意味着随着 Macintosh 本身的发展, Microsoft 必须也要不断地发展相应的产品。因此, Microsoft 公司的程序员就必须经常使用工作环境 (work-arounds) 以使 Macintosh 正常工作。但当 Apple 第一次对 Macintosh 操作系统进行版本升级时出现了问题。在早期的测试中发现新版的操作系统使 Microsoft 的产品不能正常工作。为了简化问题, Apple 请求 Microsoft 删除过时的工作环境 (work-arounds) 以保持与最新操作系统一致。

但是, 删除 Excel 中的工作环境 (work-arounds) 就意味着要重写手工优化的汇编语言过程。在代码中要增加 12 个循环。由于这个过程很关键, 围绕着是否要重写这个问题展开了讨论。一部分人认为要与 Apple 保持一致, 另一部分人则要保持速度。

最后, 程序员将一个计数器加到函数中, 让 Excel 运行了三个小时, 并观察该函数被调用的频度。该函数被调用了大概 76, 000 次, 只会使 3 小时增加到 3 小时 0.1 秒。

这个例子恰好又说明了: 关心局部效率是不值得的。如果你很注重效率的话, 请集中于全局效率和算法的效率上, 这样你才会看到努力的效果。

不一致性是编写正确代码的障碍

请看下面的代码, 它包含了最简单一类的错误——优先级错:

```
word = high << 8 + low ;
```

该代码原意是用两个 8 位字节组合成一个 16 位的字, 但是由于 + 操作符比移位操作符的优先级要高, 因此, 该代码实际实现的是把 high 移动了 8 + low 位。程序员一般不将移位操作符和算术操作符混合使用。如果只用移位类操作符或只用算术类操作符就可以完成, 那么为什么还要将移位操作符和算术操作符混合起来呢?

```
word = high << 8 | low;    /* 移位解法 */
```

```
word = high * 256 + low;    /* 算术解法 */
```

这些式子难以理解吗? 它们的效率低吗? 当然不是。这两种解法差别很大, 但这两种解法都是正确的。

若程序员在写表达式时只用一类操作符, 那么出现错误代码的概率就要小一些, 因为凭直觉, 同一类操作符的优先顺序容易掌握。当然也有例外, 但是作为一条原则, 这是对的。有多少程序员, 脑子里想着先加后除, 却写成下面的表达式呢?

```
midpoint = upper + lower / 2;
```

由于程序员学过数理逻辑课程, 熟悉象 $f(A, B, C) = AB + C$ 这样的函数, 因此记住移位操作符的优先级顺序不会有什么困难。大多数程序员都知道顺序 (从高到低) 是 “~”

“&”、“/”。很容易想到可在“~”和“&”之间插入移位操作符，因为它没有“~”约束得那么紧（想一想 $\sim A < 2$ ），但是它却比“&”的优先级要高（想一想幂运算和乘法运算就可推知了）。

程序员可能清楚知道各类操作符的优先级，但是在他们混合使用各类操作符时，很容易出现问题。因此，第一条原则是：如果有可能，就不要把不同类型的操作符混合使用。第二条原则是：如果必须将不同类型操作符混合使用，就用括号把它们隔离开来。

你已经看到了第一条原则如何使代码免于出错。请看下面的 while 循环，这在前面给出过了，从这个例子又可以看到第二条原则是如何避免代码出错的：

```
while ( ch = getchar( ) != EOF )
    .....
```

上面的代码将赋值操作符与比较操作符混合使用，从而引入了一个优先级错。可以通过重写没有操作符混合使用的循环来改正上面的错误，但是结果看上去很糟：

```
do
{
    ch = getchar ();
    if( ch == EOF )
        break;
    .....;
}while ( TRUE );
```

在这种情况下，最好打破第一条原则，使用第二条原则，用括号将操作符分隔开来：

```
while ( (ch = getchar()) != EOF )
    ..... ;
```

不能毫无必要地将不同类型地操作符混合使用，如果必须将不同类型地操作符混合使用，就用括号把它们隔离开来

不查优先级表

要插入括号的时候，有些程序员总要先查优先级表，再来确定是否有必要插入括号，如果没有必要就不插。对于这样的程序员，要提醒他：“如果必须通过查表才能确定优先顺序的话，那就太复杂了，简单一些嘛。”这就意味着可以在不需要括号的地方插入括号。这样做不仅正确，而且显然可使任何人不经查表就可判断优先级了。

避免和错误联系在一起

在上一章，我们讨论了在设计函数时尽量避免返回错误值，以免程序员错误地处理或漏掉这些返回值（例如 tolower，当 ch 不是大写字符时，它返回-1）。本章，我们又要谈论这个话题，“不要调用返回错误的函数”，这样，就不会错误地处理或漏掉由其它人设计的函数

所返回的错误条件。有时必须要调用这种函数，在这种情况下，必须在调试系统中走查这段错误处理代码，从而确保该函数正确地工作。

但要强调一点，如果自始至终程序反复处理同样的错误条件，就将错误处理部分独立出来。最简单的方法，也是每个程序员都知道的方法，就是将错误处理放在一个子过程中，这样做效果很好。但在某些情况下，还可以做得比这更好。

例如，字符窗口具有可为一个窗口在六、七处换名的代码，如下述：

```
if( fResizeMemory( &hwnd->strWndTitle, strlen(strNewTitle)+1 ) )
    strcpy( hwnd->strWndTitle, strNewTitle );
else
    /* 不能为窗口名分配空间 …… */
```

在存储区具有足够的空间来存放新名字的情况下，上面的代码更改了窗口的名字，如果空间不够，它将保持窗口的当前名并设法对错误进行处理。问题是，怎样处理这个错误呢？向用户报警？不言语，悄悄地留下原来的名字？将新名字截取下来复制到当前名字上？这几种解决方法都不理想，特别是当代码作为通用子过程的一部分时，就更不理想。

上面所述的只是不想让代码失败的多种情况中的一种情况。要为一个窗口重新命名，总是办得到的。

上面代码所存在的问题是，不能保证有足够的存储空间来存放新的窗口名字。但是，如果愿意超量分配名字空间，这个问题就很容易解决。例如，在一个典型的字符窗口应用中，只有少数窗口需要重新命名，这些名字所占的存储空间都不大，即使名字都是最大长度，我们就为存放最长的名字分配足够的空间，而不是仅仅分配当前名字长度所需的空间。于是，重命名窗口就变成如下的简单代码：

```
strcpy( hwnd -> strWndTitle, strNewTitle );
```

还可改得更好，将实现细节隐藏在 RenameWindow 函数中，使用断言来验证所分配的名字空间有足够大，它可以存放可能的任何名字：

```
void RenameWindow( window *hwnd, char *strNewTitle )
{
    ASSERT( fValidWindow(hwnd) );
    ASSERT( strNewTitle != NULL );
    ASSERT( fValidPointer( hwnd->strWndTitle, strlen(strNewTitle) - 1 ) );
    strcpy( hwnd->strWndTitle, strNewTitle );
}
```

这种方法的缺点是：当超量分配名字空间时，就会浪费存储区。但同时，由于不需要任何错误处理代码，又复得了代码空间。现在的问题是权衡数据空间和代码空间，并根据运行的具体情况决定哪个更重要。假如程序中有数千个窗口需要重命名，你可能就不会超量分配窗口名字空间了。

避免调用返回错误的函数

小结

至此，本章所讲的：程序设计是“风险事业”的含义已经很清楚了。本章的所有观点都集中在如何把有风险的编码转换成编写在空间、速度、甚至在无错方面都堪与之匹敌的代码上。

但是，不要倡留在本章给出的各点上，在实践中要不断地总结出自己的新观点，并在编码时严格地遵守这些原则。你是否周密思考了每一个编码习惯？是否因为看到别的程序员采用了某个编码习惯，于是自己也采用？刚刚入门的程序员经常认为用移位实现除法是一种“技巧”，而有经验的程序员则认为这是十分显然的，没有什么可值得疑虑的，哪个正确呢？

要点：

- 在选择数据类型的时候要谨慎。虽然 ANSI 标准要求所有的执行程序都要支持 char, int, long 等类型，但是它并没有具体定义这些类型。为了避免程序出错，应该只按照 ANSI 的标准选择数据类型。
- 由于代码可能会在不理想的硬件上运行，因此很可能算法是正确的而执行起来却有错。所以要经常详细检查计算结果和测试结果的数据类型范围是否上溢或下溢。
- 在实现某个设计的时候，一定要严格按照设计去实现。如果在编写代码时只是近似地实现所提出的要求，那就很容易出错。
- 每个函数应该只有一个严格定义的任务，不仅如此，完成每个任务也应只有一种途径。假如不管输入什么都能执行同样的代码，那就会大大降低那些不易被发现的错误所存在的概率。
- if 语句是个警告信号，说明代码所做的工作可能比所需要的要多。努力消除代码中每一个不必要的 if 语句，经常反问自己：“怎样改变设计从而删掉这个特殊情况？”有时可能要改变数据结构，有时又要改变一下考察问题的方式，就象透镜是凸的还是凹的问题一样。
- 有时 if 语句隐藏在 while 和 for 循环的控制表达式中。“?:”操作符是 if 语句的另外一种形式。
- 警惕有风险的语言惯用语，注意那些相近但更安全的惯用语。特别要警惕那些看上去象是好编码的惯用语，因为这样的实现总体效率很少有显著的影响，但却增加了额外的风险性。
- 在写表达式时，尽量不要把不同类型的操作符混合起来，如果必须混合使用，用括号把它们分隔开来。
- 特殊情况中的特殊情况是错误处理。如果有可能，应该尽量避免调用可能失败的函数，假如必须调用返回错误的函数，将错误处理局部化以便所有的错误都汇集到一点，这将增加在错误处理代码中发现错误的机会。
- 在某些情况下，取消一般的错误处理代码是有可能的，但要保证所做的事情不会失败。这就意味着在初始化时要对错误进行一次性处理或是从根本上改变设计。

练习：

- 1) “简单的”一位位域的可移植范围是什么？
- 2) 如果布尔量的值用“简单的”一位位域来表示的话，返回布尔值的函数应该是怎样的？
- 3) 从 AddChild 的第二个版本到最后版本，都使用全局变量 pwndDisplay 来指向已分配的表示整个显示的窗口结构。如果不这样，也可以声明一个全局的窗口结构：wndDisplay。尽管这样也可以，但为什么不这样做呢？
- 4) 假如有个程序员提出；为了提高效率是否应该将下面的循环

```
while( expression )
{
    A;
    if( f )
        B;
    else
        C;
    D;
}
```

改写成为：

```
if( f )
    while( expression )
    {
        A;
        B;
        D;
    }
else
    while( expression )
    {
        A;
        C;
        D;
    }
```

上面的 A 和 D 代表语句集。第二个版本速度要快一些，但是，与第一个版本比较起来它有什么风险？

- 5) 如果你阅读了 ANSI 标准的话，将会发现这样一些函数，它具有若干个名字几乎相同的参数。例如：

```
int strcmp( const char *s1, const char *s2 );
```

为什么说这样的函数是有风险的?应如何消除风险?

- 6) 象下面这样的条件循环是有风险的:

```
while( pch ++ <= pchEnd )
```

但使用类似的递减循环为什么还有风险呢?

```
while( pch -- >= pchStart )
```

- 7) 一些程序员为了提高效率或使问题变得简洁,采用了下面的简化方法。为什么应该避免这样做呢?

a) 用 `printf( str )` 代替 `printf( "%s",str);`

b) 用 `f=1-f` 代替 `f=!f;`

c) 用

```
int ch; /* ch 必须是整数 */
```

```
.....
```

```
ch = *str ++ = getchar( );
```

```
.....
```

代替两个独立的赋值语句。

- 8) `uCycleCheckBox`, `tolower` 和第 2 章中的反汇编程序都使用了表驱动算法,使用表的优点及风险性是什么?
- 9) 假设你的编译程序不能为无符号 2 的幂的算术运算自动提供移位操作符,那么除风险性问题和不可移植性问题之外,为什么在明确的优化中仍然应该避免使用移位和“与”操作?
- 10) 程序设计中的一个重要原则是:决不能丢失用户的数据。假设你正在做 `WordSnasher` 项目,需要编写“保存文件”子例程,这时,为了保存用户文件,必须给用户文件分配一个临时数据缓冲区。问题是如果你无法分配缓冲区,就不能保存文件,从而违反了上述原则。怎样做才能保证保存用户文件呢?

课题:

将所有你能想到的有风险的语言特点列成一个表格,列出使用每一特征的优缺点。随后,针对表中的每一项,分析在什么情况下,你宁愿冒险而使用该特征。