

附录 C 练习答案

本附录给出本书中所有练习的答案。

第 1 章

1) 编译程序会查获优先顺序错。因为它把表达式解释为：

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

换句话说，编译程序把它看作是将表达式的值赋给 ch，因而认为你把“==”错误的键为“=”，并向你发出可能有复制错误的警告。

2a) 查获偶然“八进制错误”的最简单方法是扔掉可选择的编译开关，这个开关导致编译程序在偶然遇到八进制常量时出错。取而代之的是使用十进制或十六进制。

2b) 为了查获程序员将“&&”误键入为“&”(或“||”误键为“|”)的情况，编译程序采用了与查获将“==”误键为“=”的同样测试。当程序员在 if 语句中或复合条件中使用了“&”(或“|”)，并且没有明确地将结果与 0 进行比较时，编译程序将产生一个错误。所以见到下面这条语句会产生一个警告。

```
if ( u & 1 )           /* u 是奇数吗？*/
```

而下面这条语句则不会产生警告信息。

```
if( (u & 1) != 0 )     /* u 是奇数吗？*/
```

2c) 警告一个无意而误成为注释的最简单的方法是，当编译发现注释的第一个字符是字母或（时，发出一个警告。这样的测试将查获下面两个可疑情况：

```
quot = numer/*pdenom;
```

```
quot = number/*( pointer expression );
```

为了避免发出警告，你可以通过将“/”与“*”之间用空格或括号分开，使你的意图更明确。

```
quot = numer / *pdenom;
```

```
quot = number / (*pdenom);
```

```
/*注意：本注释将产生一个警告 */
```

```
/* 本注释不产生警告 */
```

```
/*----- 警告勿忧 -----*/
```

2d) 编译查出可能存在的优先级顺序错的方法是，寻找在同一个不含括号的表达式中的“有麻烦的运算符对”。例如，当程序员偶然将“<<”和“+”运算符一起使用时，编译程序会发现优先级顺序错，对下面的代码发出警告：

```
word = bHigh << 8 + bLow;
```

但是，由于下面的语句含有括号，因此编译程序不发出警告信息：

```
word = ( bHigh << 8 ) + bLow;
```

```
word = bHigh << ( 8 + bLow );
```

如果不专设注释则可写警告式注释：“如果两个运算符具有不同的优先级顺序并没被括号括起，那么就要发出一个警告。”这样的注释太贫，但你在思想上要明白这点。开发一个好的启发式注释，需要在计算机上运行大量的代码直到最后产生有用的结果。你肯定不希望对下面这些常见的惯用语也产生警告信息：

```
word = bHigh * 256 + bLow ;
if ( ch == ' ' || ch == '\t' || ch == '\n' )
```

3) 当编译程序发现两个连续的 if 语句其后跟有一个 else 语句时，编译程序就会发出可能有悬挂 else 的警告信息：

if(expression 1)	if(expression 1)
if(expression 2)	if(expression 2)
.....	
else	else
.....	

为了避免编译程序发出警告信息，可以用括号将内层 if 语句括起：

if(expression1)	if(expression1)
{	{
if(expression2)	if(expression2)
.....	
}	else
else	
.....	}

4) 将常量和表达式置于比较操作的左边是很有意义的，它提供了自动检查错误的有一个方法。但时，这种方法必须有一个操作数是常量或表达式作为前提，如果两个操作数都是变量，这个方法就不起作用了。请注意，程序员在写代码的时候，一定要学会并记住使用这一技术。

通过使用编译开关，编译程序将警告每一种可能的赋值错。特别是对于没有经验的程序员，编译开关更显得特别有作用。

如果有编译开关，就一定要使用；如果没有，就把常量和表达式放在比较式的左边。

5) 为了防止误定义的预处理的宏产生不可预料的结果，编译（实际是预处理）程序应该具有一个开关允许程序员可以把无定义的宏用于错误情况。由于 ANSI 编译程序及支持老的 #ifdef 预处理指令，又支持新预处理的 defined 一元算子，那么就几乎没有必要将无定义的宏“定义”为 0。以下代码将会产生错误：

```
/* 建立目标等式 */
```

```

# if          INTEL8080
.....
#elif        INTEL80x86
.....
#elif        MC6809
.....
#elif        MC680x0
.....

#endif

```

因此，应写为如下代码：

```

/* 建立目标等式 */
#if          defined ( INTEL8080 )
.....
#elif        defined ( INTEL80x86 )
.....
#elif        defined ( MC6809 )
.....
#elif        defined ( MC680x0 )
.....

#endif

```

如果在 `# ifdef` 语句中使用了无定义的宏，此开关不会给出警告，因为这是有意安排的。

第二章

1) `ASSERTMSG` 宏的一种可能的实现是使它产生两个作用：一个是确认表达式，另一个是当断言否定时显示一个字符串。例如，若要打印 `memcpy` 的消息，应如以下形式调用 `ASSERTMSG`：

```

ASSERTMSG( pbTo >= pbFrom + size || pbFrom >= pbTo + size,
           "memcpy: the blocks overlap" );

```

下面是 `ASSERTMSG` 宏的实现。你应将 `ASSERTMSG` 的定义放在头文件中，再将 `_AssertMsg` 例程放在一个方便的源文件内。

```

#ifdef DEBUG
void _AssertMsg( char* strMessage );           /* 原型 */
#define ASSERTMSG( f, str ) \
    if( f ) \
        NULL \

```

```

        else \
            _AssertMsg( str )
#else
    #define ASSERTMSG( f, str )    NULL
#endif
在另外一个文件中有：
#ifdef DEBUG
    void _AssertMsg( char* strMessage )
    {
        fflush( stdout );
        fprintf( stderr, "\n\n Assertion failure in %s \n", strMessage );
        fflush( stderr );
        abort();
    }
#endif

```

2) 如果你的编译程序支持一个这样的开关，它通知编译程序将所有相同的字符串分配在同一个位置上，那么最简单的办法就是不要这个开关。如果允许这个选择，你的程序即或声明了 73 个文件名的副本，编译程序只分配一个字符串。这种方法的缺点是，它不仅“覆盖”了断言字符串，还将源文件中所有等长的字符串都“覆盖”了，只是不希望有的多余行为。

另一种办法是改变 ASSERT 宏的实现，有意识的只引用整个文件中相同文件名的字符串。唯一的困难是如何建立文件名的字符串，但是即使这不成问题，你也应该把实现细节隐藏在一个新的 ASSERTFILE 宏中，这个宏只在源程序文件的开始处使用一次：

```

#include <stdio.h>
.....

#include <debug.h>
ASSERTFILE( __FILE__ )    /* 加 */
.....

void* memcpy( void* pvTo, void* pvFrom, size_t size )
{
    byte* pbTo = (byte*)pvTo;
    byte* pbFrom = (byte*)pvFrom;
    ASSERT( pvTo != NULL && pvFrom != NULL );    /* 没有变更 */
    .....
}

```

下面是实现 ASSERTFILE 宏的代码和相应的 ASSERT 版本。

```

#ifdef DEBUG

```

```

#define ASSERTFILE(str) static char strAssertFile[] = str;
#define ASSERT(f)          \
    if( f )                \
        NULL              \
    else                    \
        _Assert( strAssertFile, _LINE_ )
#else
#define ASSERTFILE(str)
#define ASSERT(f) NULL
#endif

```

使用该版本的 ASSERT，可以获得大量的存储空间。例如，本书的测试应用程序很小，但是使用上面新给的代码，这些程序可以节省 3K 的数据空间。

3) 使用该断言的问题是测试包含了应保留在函数非调试版本中代码。非调试代码将进入一个无限循环，除非在执行 do 循环中，ch 碰巧等于执行符。所以函数应写成如下形式：

```

void getline( char* pch )
{
    int ch;          /* ch 必须是 int 类型 */
    do
    {
        ch = getchar();
        ASSERT( ch != EOF );
    }
    while( ( *pch++ = ch ) != ' \n' );
}

```

4) 查出不许修改的开关语句中所存在的错误，有一个很简单的方法，这就是将断言加到 default（缺省）分支来证实 default 分支是唯一处理那些应该处理的分支。在某些情况下，不能引用 default 分支，因为所有可能的情况都被明确地处理了。如果发生上述情况，请使用以下代码：

```

.....
default:
    ASSERT(FALSE);    /* 此处从不可达 */
    break;
}

```

5) 表中屏蔽码与相对应的模式之间有一个关系，模式应该总是屏蔽码的子集，或者说，

一旦被屏蔽，不能有任何指令与该模式相匹配。下面的 CheckIdInst 程序用来证实模式是屏蔽码的子集：

```
void CheckIdInst( void )
{
    identity *pid, *pidEarlier;
    instruction inst;
    for( pid = &idInst[0]; pid->mask != 0; pid++ )
    {
        /* 模式肯定是屏蔽码的子集 */
        ASSERT( (pid->pat & pid->mask) == pid->pat );
        .....
    }
}
```

6) 使用断言来证实 inst 没有任何有疑问的设置：

```
instruction* pcDecodeEOR( instruction inst, instruction* pc, opcode* popc )
{
    /* 我们是否错误地得到了 CMPM 或 CMPA.L 指令？ */
    ASSERT( eamode(inst) != 1 && mode(inst) != 3 );
    /* 如果为非寄存器方式，则只允许绝对字和长字方式 */
    ASSERT( eamode(inst) != 7 || ( eareg(inst) == 0 || eareg( inst ) == 1 ) );
    .....
}
```

7) 选择备份算法的关键是要选择一个不同的算法。例如，为了证实 qsort 是可以工作的，你可以扫描排序后的数据，以验证次序是正确的（扫描并不是排序，应把它看作不同的算法）。为了验证二分查找工作正常，就用线性扫描来看一下两种查找的结果是否相同。最后，为了验证 itoa 函数正确，将该函数返回的字符串重新转换为整数，然后与原来传递给 itoa 的整数进行比较，它们应该相等。

当然，除非你在为航天飞机、放射工厂、或其他一些一旦出错，可能威胁生命的情况编码，否则，你可能不想为你写的每一段代码都用备份算法。但是，对于应用中所有较重要的部分都应该使用备份算法。

第 3 章

1) 通过用不同的调试值来破坏两类存储空间，能容易的区分某个程序是使用了未初始化数据还是继续使用已释放的数据。例如，利用 bNewGarbage, fNewMemoery 可以破坏新的未初始化的存储空间，使用 bFreeGarbage, FreeMemory 可以破坏已释放的存储空间：

```
#define bNewGarbage      0xA3
```

```
#define bFreeGarbage      0xA5
```

fResizeMemory 建立这两类无用数据，你可以使用上面的两个值，或者，你也可以建立两个别的值。

2) 查获“溢出”错的一个方法是，定期地对跟在每一个已分配块后面的字节进行检查，证实这些字节并没有被修改。尽管这种测试听起来很直观，但是它却要求你记住所有的字节，而且它还忽略了你可能会再没有分配给你的存储块里进行读操作这样一个潜在的问题。幸运的是，还有一个简单的方法来实现这个测试，只不过要为你为每一个分配的块再分配一个额外的字节。

例如，当你调用 fNewMemory 时需分配 36 字节，你实际上要分配 37 字节，并且在那个额外的存储单元内存储一个已知的“调试字”。类似地，当 fResizeMemory 调用 realloc 时，你可以分配和设置一个额外的字节。为了查获溢出错，应该在 sizeofBlock, fValidPointer, FreeBlockInfo, NoteMemoryRef 和 CheckMemoryRefs 中加入断言来证实还没有接触到调试位。

下面是实现该代码的一种方法。首先，你要定义 bDebugByte 和 sizeofDebugByte：

```
/* bDebugByte 是一个奇异的值，它存储在该程序的 DEBUG 版本的每一个被
 * 分配存储块的尾部，sizeofDebugByte 是加到传给 malloc 和 realloc 的
 * size 上，使分配的空间大小正确。
 */
#define bDebugByte 0xE1
#ifdef DEBUG
    #define sizeofDebugByte 1
#else
    #define sizeofDebugByte 0
#endif
```

下一步，你应该在 fNewMemory 和 fResizeMemory 中用 sizeofDebugByte 来调整对 malloc 和 realloc 的调用，如果分配成功，就用 bDebugByte 来填充那些额外字节：

```
flag fNewMemory( void** ppv, size_t size )
{
    byte** ppb = ( byte** )ppv;
    ASSERT( ppv != NULL && size != 0 );
    *ppb = (byte*)malloc( size + sizeofDebugByte );    /* 变更了 */
#ifdef DEBUG
    {
        *( *ppb + size ) = bDebugByte;                /* 加 */
        memset( *ppb, bGarbage, size );
        .....
```

```

flag fResizeMemory( void** ppv, size_t sizeNew )
{
    byte** ppb = ( byte** )ppv;
    byte* pbResize;
    .....

    pbResize = (byte*)realloc(*ppv, sizeNew + sizeofDebugByte); /* 变更了 */
    if( pbResize != NULL )
    {
        #ifdef DEBUG
        {
            *( pbResize + sizeNew ) = bDebugByte;          /* 加 */
            UpdateBlockInfo( *ppv, pbResize, sizeNew );
            .....

```

最后,将以下断言插入到 sizeofBlock、fValidPointer、FreeBlockInfo、NoteMemoryRef 和 CheckMemoryRefs 例程中,这些例程在附录 B 中给出。

```

/* 保证在块的上界之外什么也没有写入 */
ASSERT( *( pbi->pb + pbi->size ) == bDebugByte );

```

做了这些改动之后,存储子系统就可以查获那些写到所分配的存储块上界之外的溢出错误了。

3) 查获不该悬挂的指针错有许多方法。一个可能的解就是,更改 FreeMemory 的调试版本使它不真正地释放这些存储块,而是为已分配的块建立一个释放链,(这些存储块,对于系统来讲,它们是已分配的,对于用户程序来讲,它们已被释放了)。以这种方式修改 FreeMemory 将是“释放的”存储块在调用 CheckMemoryRefs 来确认子系统之前不被重新分配。CheckMemoryRefs 通过获取 FreeMemory 的“释放”链和真正释放所有这些存储块,使存储系统有效。

虽然该方法可以查获不该悬挂的指针,但是,除非你的程序遇到了这类错误,一般不要使用这种方法。因为这种方法违反了“调试代码时附加了额外信息的代码,而不是不同的代码”原则。

4) 为了使指针所引用的对象大小有效,必须考虑两种情况:一种情况是指针指向整个块;另一种情况是指针指向块内的部分分配空间。对于第一种情况,可以采取最严格的测试来证实指针引用了块的开头,块的大小与 sizeofBlock 函数的返回值相匹配。对于第二种情况,测试应弱一些:即指针只要指在块内,大小没有超出块的结尾就可以了。

因此,如不使用 NoteMemoryRef 程序来表示部分分配块和完整块,可以使用两个函数来表示两类块,这可以通过下面的方式来实现:给已有的 NoteMemoryRef 函数增加一个参数 size,用扩充以后的 NoteMemoryRef 函数标识部分分配块 建立一个新函数 NoteMemoryBlock

来表示完整块，如下所示：

```
/* NoteMemoryRef ( pv , size )
 *
 * NoteMemoryRef 将 pv 所指的存储块标志为被引用的。注意：pv 不必指向一个
 * 存储块的开始；它可以指向一个已分配存储块内的任意位置，但是在该存储块
 * 内至少要剩有“size”个字节。注意：如果有可能，就使用 NoteMemoryBlock ——
 * 它更可靠。
 */
void NoteMemoryRef( void* pv, size_t size );
/* NoteMemoryBlock( pv, size )
 *
 * NoteMemoryBlock 将 pv 所指的存储块标志为被引用。注意：pv 必须
 * 指向一个存储块的开始，该存储块长度恰好为“size”个字节。
 */
void NoteMemoryBlock( void* pv, size_t size );
这些函数可以查获在练习中给出的错误。
```

5) 为了改进附录 B 中的完整性检查，应该首先将 BLOCKINFO 结构中的引用标志改为引用计数，然后更改 ClearMemoryRef 和 NoteMemoryRef，使其对计数器进行处理，这是很明显的。可是，怎样来修改 CheckMemoryRefs，使得当某些有多个引用的情况时，它只为这些块作断言检查而不为别的存储块作断言检查呢？

解决这个问题的一個方法是：改进 NoteMemoryRef 例程，是它除了具有指向存储块的指针外，还保持一个标尺存储块的标签 ID。NoteMemoryRef 可以将标签保存在 BLOCKINFO 结构中，随后作 CheckMemoryRefs 并用标签来检验引用计数器。下面是进行了这些变化以后的代码。前面的注释请参见附录 B 中的原版函数：

```
/* 块标签是为引用保存各种类型分配块的表 */
typedef enum
{
    tagNone,          /* ClearMemoryRefs 将所有块置为 tagNone */
    tagSymName,
    tagSymStruct,
    tagListNode,      /* 这些块必须有两种引用 */
    .....
}blocktag;

void ClearMemoryRefs( void )
{
```

```

    blockinfo* pbi;
    for( pbi = pbiHead; pbi != NUL; pbi = pbi->pbiNext )
    {
        pbi->nReferenced = 0;
        pbi->tag = tagNone;
    }
}

void NoteMemoryRef( void* pv, blocktag tag )
{
    blockinfo* pbi;
    pbi = pbiGetBlockInfo( (byte*)pv );
    pbi->nReferenced++;
    ASSERT( pbi->tag == tagNone || pbi->tag == tag );
    pbi->tag = tag;
}

void CheckMemoryRefs( void )
{
    blockinfo* pbi;
    for( pbi = pbiHead; pbi != NULL; pbi = pbi->pbiNext )
    {
        /* 简单的检查块的集成性。若以下断言引发则意味着管理块信
        * 息的调试代码错了，或者可能有一新的存储抹去了数据结
        * 构。这两种情况都是错误。
        */
        ASSERT( pbi->pb != NULL && pbi->size != 0 );
        /* 检查失去或漏掉的内存，若全无引用则意味着 app 要么丢失了该块的
        * 踪迹，要么没有使所有全局指针都计入 NoteMemoryRef。某些
        * 类型的块可以有多个对它们的引用。
        */
        switch( pbi->tag )
        {
            default:
                ASSERT( pbi->nReferenced == 1 );
                break;
            case tagListNode:

```

```

        ASSERT( pbi->nReferenced == 2 );
    break;
    .....
}
}
}
}

```

6) DOS、Windows 和 Macintosh 的开发者通常使用下面的方法来测试内存空间耗尽条件。他们使用一个工具来任意占用存储空间直到应用申请的存储空间出错为止。尽管这种方法可以起作用，但是并不精确，它会引起程序某个地方要求的分配失败。如果要测试一个孤立的特征，这种技术并不十分有用。一个更好的方法是，在存储管理程序中建立存储器溢出的模拟程序。

但请注意，存储错仅仅是资源错误的一种类型，还有磁盘错、出纸错、电话线路忙碌出错等各种错误。因此，需要一个故意制造资源短缺的通用工具。

一个解决办法是：建立 failureinfo 结构，在该结构中包含有通知如何做错误处理机制的信息。程序员和测试员在外部测试中填入 failureinfo 结构，然后，再演示他们的特征。

(Microsoft 应用经常使用 debug-only (只调试) 对话，它允许测试员用这样的系统，象 Excel 一类的应用中有宏语言，有一种 debug-only 宏能允许测试员将这一过程自动化)。

为了声明存储管理器的故障结构，应使用如下的代码：

```
failureinfo fiMemory;
```

为了在 fNewMemory 或 fResizeMemory 中模拟内存耗尽错，应将四行调试代码加到每个函数中：

```

flag fNewMemory( void** ppv, size_t size )
{
    byte** ppb = ( byte** )ppv;
    #ifdef DEBUG
        if( fFakeFailure( &fiMemory ) )
            return( FALSE );
    #endif
    .....

flag fResizeMemory( void** ppv, size_t sizeNew )
{
    byte** ppb = ( byte** )ppv;
    byte* pbResize;
    #ifdef DEBUG
        if( fFakeFailure( &fiMemory ) )

```

```

        return( FALSE );
    #endif
    .....

```

这样在代码中设置了故障机制，为了使其起作用，要调用 SetFailures 函数来初始化 failureinfo 结构：

```
SetFailures( fiMemory, 5, 7 );
```

用 5 和 7 调用 SetFailures 是告诉故障系统，在得到 7 个连续的故障之前要成功地调用系统 5 次。对 SetFailures 的两个常见的调用是：

```
SetFailures( &fiMemory, UINT_MAX, 0 ); /* 不要伪造任何故障 */
```

```
SetFailures( &fiMemory, 0, UINT_MAX ); /* 总是伪造故障 */
```

用 SetFailures，可以写出一次又一次调用同一段代码的单元测试，它是每次要用不同的值调用 SetFailures 来模拟所有可能的错误模式。通常将第二个“失败”值保持为 UINT_MAX，第一个“成功”值计数从 0 到某个很大的数，逐渐试探它。这个数大到能测试出所有的内存耗尽条件。

最后，当要多次调用内存（或磁盘等等）系统时，你肯定希望不出故障；特别是在某个调试代码内分配资源时，常常如此。下面两个可嵌套函数暂时允许故障机制失灵：

```
DisableFailures( &fiMemory );
```

… 进行分配 …

```
EnableFailures( &fiMemory );
```

下面的代码是建立四个函数的故障机制：

```
typedef struct
```

```

{
    unsigned nSucceed;    /* 在出故障之前有 # 次成功 */
    unsigned nFail;       /* # 次失败 */
    unsigned nTries;      /* 已被调用 # 次 */
    int lock;             /* 如 lock>0, 该机制不工作 */
}failureinfo;

```

```
void SetFailures( failureinfo* pfi, unsigned nSucceed, unsigned nFail )
```

```

{
    /* 如果 nFail 是 0, 则要求 nSucceed 为 UINT_MAX */
    ASSERT( nFail != 0 || nSucceed == UINT_MAX );
    pfi->nSucceed = nSucceed;
    pfi->nFail = nFail;
    pfi->nTries = 0;
    pfi->lock = 0;
}

```

```

void EnableFailures( failureinfo* info )
{
    ASSERT( pfi->lock > 0 );
    pfi->lock--;
}

void DisableFailures( failureinfo* pfi )
{
    ASSERT( pfi->lock >= 0 && pfi->lock < INT_MAX );
    pfi->lock++;
}

flag fFakeFailure( failureinfo* pfi )
{
    ASSERT( pfi = NULL );
    if( pfi->lock > 0 )
        return( FALSE );
    if( pfi->nTries != UINT_MAX )      /* 勿使 nTries 溢出 */
        pfi->nTries++;
    if( pfi->nTries <= pfi->nSucceed )
        return( FALSE );
    if( pfi->nTries - pfi->nSucceed <= pfi->nFail )
        return( TRUE );
    return( FALSE );
}

```

第 4 章

第四章没有练习。

第 5 章

1) 与 malloc 一样，由于 strdup 的错误返回值是有假象的 NULL 指针，易于失察，因此，strdup 具有一个危险的界面。作为一个不易出错的界面应将错误条件与指向输出的指针分开，使错误条件更清晰。如下代码就是这样的界面：

```
char* strDup;           /* 指向复制串的指针 */
if( fStrDup( &strDup, strToCopy ) )
    成功 —— strDup 指向新串
else
    失败 —— strDup 为 NULL
```

2) getchar 的界面比 fGetChar 界面要好，它将返回一个错误代码而不是一个 TRUE 和 FALSE 的是否“成功”的值。例如：

```
/* errGetChar 可能返回错误 */
typedef enum
{
    errNone = 0 ,
    errEOF ,
    errBadRead ,
    .....
}error;

void ReadSomeStuff( void )
{
    char ch;
    error err;
    if( ( err = errGetChar(&ch) ) == errNone )
        成功 —— ch 得到下一个字符
    else
        失败 —— err 具有错误类型
    .....
}
```

这个界面之所以比 fGetChar 的界面好，是因为它允许 errGetChar 返回多种错误条件(和多种对应的成功条件)。如果你不关心返回错误的具体情况，可以取消局部变量 err，回到 fGetChar 的界面形式：

```
if( errGetChar(&ch) == errNone )
    成功 —— ch 得到下一个字符
else
    失败 —— 不关心是什么错误类型
```

3) strncpy 函数有一个麻烦的问题，该函数的性能不稳定：有时 strncpy 用一个空字符终止一个指定的字符串，有时就不是这样。strncpy 与别的通用字符串函数列在一起，程

序员可能会错误地断定 `strncpy` 函数本身是一个通用函数，其实它并不是。由于它具有异常的性能，事实上 `strncpy` 不应在 ANSI 标准中，但是，由于它在 ANSI C 的预处理实现中广泛使用，所以也可以说它在 ANSI 标准中。

4) C++ 的 `inline` (内联) 函数指明符非常有价值，它允许用户定义象宏一样有效的函数，然而还没有宏“函数”对参数求值时所带来的那些麻烦的副作用。

5) C++ 新的 & 引用参数有一个严重的问题，它隐藏了一个事实，即通过引用来传递变量，而不是通过值，这可能会引起混乱。例如，假设你重新定义了 `fResizeMemory` 函数，使用了引用参数。程序员可以写：

```
if( fResizeMemory( pb, sizeNew ) )  
    resize 是成功的
```

但是要注意，不熟悉这个函数的程序员不会认为在调用期间 `pb` 可能会改变。你认为这是否会影响程序的维护呢？

与此相联系的是，C 程序员经常对他们函数中的形式参数进行操作，因为他们知道这些参数是通过值传递的，而不是通过引用。但是，考虑一下维护人员要修改函数中的错误，就不能这样写。如果这些程序员没有注意到声明中的 `&`，他可能就修改了参数，而且没有意识到这个变更并非局部于这个函数。

6) `strcmp` 的界面所存在的问题是，该函数的返回值在调用点导致产生了难理解的代码。为了改进 `strcmp`，设计界面时应使返回值对于那些即使不熟悉该函数的程序员也很容易理解。

有一种界面，它对现在的 `strcmp` 作了较小的改动。它不是对不相等的字符串返回某个正值或负值，而是迫使程序员将所有的比较都改为和 0 比较。修改 `strcmp` 使它返回三个定义良好的命名常量：

```
if( strcmp( strLeft, strRight ) == STR_LESS )  
if( strcmp( strLeft, strRight ) == STR_GREATER )  
if( strcmp( strLeft, strRight ) == STR_EQUAL )  
另一种可能的界面是，每一类比较都用单独的函数：  
if( fStrLess( strLeft, strRight ) )  
if( fStrGreater( strLeft, strRight ) )  
if( fStrEqual( strLeft, strRight ) )
```

第二种界面的优点是，可以通过在已有的 `strcmp` 函数上使用宏来实现。把 `<=` 和 `>=` 这样的比较定义为宏可以大大提高可读性。结果是，提高了可读性，在空间和速度方面也没有损失。

```
#define fStrLess(strLeft, strRight) ( strcmp(strLeft, strRight) < 0 )  
#define fStrGreater(strLeft, strRight) ( strcmp(strLeft, strRight) > 0 )
```

```
#define fStrEqual(strLeft, strRight)    ( strcmp(strLeft, strRight) == 0 )
```

第 6 章

1) “简单的”1 位位域的可移植范围为 0，这没什么用。位域确实有非 0 状态，却不知道这是什么值：该值可以是 -1 或 1，这取决于所使用的编译程序在缺省状态下是带符号的位域呢还是不带符号的好的位域。如果将所有的比较都限制为与 0 进行比较，那么就可以安全地使用位域的两种状态。如果假设 psw.carry 是个简单的 1 位位域，则可以安全地写如下的代码：

```
if( psw.carry == 0 )           if( !psw.carry )
if( psw.carry != 0 )           if( psw.carry )
```

但是，下面的语句是有风险的，因为它们依赖于所使用的编译程序：

```
if( psw.carry == 1 )           if( psw.carry == -1 )
if( psw.carry != 1 )           if( psw.carry != -1 )
```

2) 返回布尔值的函数就像“简单的”1 位位域一样，没办法安全的预言“TRUE”返回的值将是什么。可以依赖于：FALSE 是 0。但是程序员经常把非 0 值作为“TRUE”的返回值，当然，这并不等于常量 TRUE。如果你假设 fNewMemory 返回一个布尔值，那么就可以安全地写成下面的代码：

```
if( fNewMemory( ... ) == FALSE )
if( fNewMemory( ... ) == FALSE )
```

甚至更好的代码：

```
if( fNewMemory( ... ) )
if( fNewMemory( ... ) )
```

但是，下面的代码是有风险的，因为它假设 fNewMemory 将不会返回除了 TRUE 之外的任何非零值：

```
if( fNewMemory( ... ) == TRUE )    /* 有风险 */
```

记住一个很好的规则：不要将布尔值与 TRUE 进行比较。

3) 如果将 wndDisplay 声明为一个全局窗口结构，你给它一个别的窗口结构没有的特殊属性：全局性。这看上去似乎是一个次要的细节，但是它可能会引入一个没有预料到的错误。例如，假设你想写一个释放窗口和所有子窗口的例程，下面的函数就实现了这一功能：

```
void FreeWindowTree( window* pwndRoot )
{
    if( pwndRoot != NULL )
    {
```

```

window *pwnd, *pwndNext;
ASSERT( fValidWindow( pwndRoot ) );
for( pwnd = pwndRoot->pwndChild; pwnd != NULL; pwnd = pwndNext )
{
    pwndNext = pwnd->pwndSibling;
    FreeWindowTree( pwnd );
}
if( pwndRoot->strWndTitle != NULL )
    FreeMemory( pwndRoot->strWndTitle );
FreeMemory( pwndRoot );
}
}

```

但是要注意，如果要释放每一个窗口，就可以安全地传递 `pwndDisplay`，因为它指向已分配的窗口结构。但是，不能传递 `&wndDisplay`，因为该代码将释放 `wndDisplay`，这是不可能的，因为 `wndDisplay` 是一个全局的窗口结构。为了使得有 `&wndDisplay` 的代码能够正确工作，必须在最后调用 `FreeMemory` 之前插入：

```
if( pwndRoot != &wndDisplay )
```

如果这么做了，代码就要依靠全局数据结构了。哟嗬！

要想在代码中没有错误，有一个最好的方法，这就是在实现中避免任何古怪的设计。

4) 第二版代码比第一版代码所冒的风险更大一些，这有几个原因。由于在第一版代码中 `A`、`D` 和 `expression` 都是公共代码，不管 `f` 的值是什么，它们都要被执行和测试。而在第二版中，和 `A`、`D` 有关的每一个表达式都将分别测试，除非它们是相同的，否则要冒漏掉某一个分支的风险。（如果为了与 `B` 或 `C` 联用方便而专门对两个 `A` 和两个 `D` 分别进行不同的优化，那么两个 `A` 和两个 `D` 将不同）。

在第二版中，还有一个问题，当程序员修改错误或改进代码时，很难保证两个 `A` 和两个 `D` 同步。尤其当两个 `A` 和两个 `D` 本来就不相同时就更是如此了。因此，除非计算 `f` 的代价太昂贵以至于用户都能观察出来，否则的话都使用第一版。在此请记住另外一条很有用的规则：通过最大限度地增加公共代码的数量来使代码差异减到最少。

5) 使用相似的名字是危险的，例如象 `S1` 和 `S2`，当你想键入 `S2` 时很容易误键为 `S1`。更糟的是，在编译这样的代码时，可能不会发现这个错误。使用相似的名字，使得很难发现名字颠倒错误：

```

int strcmp(const char* s1, const char* s2)
{
    for( NULL; s1==s2; s1++, s2++ )
    {

```

```

        if( *s1 == '\0' )           /* 与末端匹配吗？ */
            return(0;
    }
    return( (*(unsigned char*)s2 < *(unsigned char*)s1) ? -1 : 1 );
}

```

以上代码是错误的，最后一行的测试方向反了，由于名字本身没有含义，所以这个错误很难发现。但是，如果使用描述性的、有区别的名字，如 `sLeft` 和 `sRight`，上述两类错误的出现次数会自动下降，代码更好读。

6) ANSI 标准保证可以对所声明的数据类型的第一个字节寻址，但是，它不能保证能引用任何数据类型前面的字节；该标准也不能保证对 `malloc` 分配的存储块前面的字节寻址。

例如，某些 80x86 存储模型的指针式使用 `base:offset`（基地址：偏移量）来实现的，且只操纵无符号的偏移量。如果 `pchStart` 是指向所分配的存储块开始处的指针，则其偏移量为 0。如果你假设 `pch` 开始就超出 `pchStart+size` 的值，那么它决不会小于 `pchStart`，因为它的偏移量决不会小于 `pchStart` 的偏移量 0。

7a) 如果 `str` 包含若干%符号，则使用 `printf(str)`代替 `printf(“%s”, str)`就会出现错误，`printf` 将把 `str` 包含的%符号错误地解释为格式说明。使用 `printf(“%s”, str)`的麻烦是，由于它可以非常“明显地”被优化为 `printf(str)`，以至于粗心的程序员会在清理代码时引入错误。

7b) 使用 `f=1-f` 代替 `f=!f` 是有风险的，因为它假设 `f` 或者是 0 或者是 1。然而使用 `!f` 清楚地表明是个倒装标志，对所有 `f` 值都起作用。采用 `1-f` 的唯一理由是它能够产生比 `!f` 效率更高一点的代码，但是，要记住，局部效率的提高很少对程序的总体产生影响。使用 `1-f` 只能增加产生错误的风险。

7c) 在一个语句中使用多重赋值的风险性在于，可能会引起不希望的数据类型转换。在所给的例子中，程序员非常小心地将 `ch` 声明为 `int`，以便它能正确地处理 `getchar` 可以返回的 EOF 值。但是 `getchar` 返回的值却首先存在一个字符串中，要将值转换为 `char`，正是这个 `char` 被赋给了 `ch`，而不是 `getchar` 返回的 `int` 赋给了 `ch`。如果在系统上 EOF 的值为负，而编译程序的缺省值为无符号字符，那么错误就会很快地显现出来。但是，若编译程序的缺省值是有符号字符，EOF 可能被截取为字符，当重新转换为 `int` 时，可能恰好又一次等于 EOF。这并不意味着该代码工作正确。如果你看不出 EOF 的问题，你就丧失了区分 EOF 和 EOF 进位后所等价的字符的能力。

8) 在典型情况下，表格使得代码减少、速度加快，可用以简化代码，增加正确的概率。但是，当考虑到表中的数据时，又得出了相反的结论。首先，代码可能少了，但是表格占用了存储空间，总的来说，表格解法可能比非表格实现占用的存储空间要多。使用表格的另一个问题是具有风险性，你必须确保表格中的数据是完全正确的，有时很容易做到，比如

tolower 何 uCycleCheckBox 表格就是如此。但是，对于一些大表格象第 2 章反汇编程序中的表格，要保证表中的数据完全正确就很难了，因为很容易引入错误。所以得到了一条原则：除非你可以确保数据有效，否则不要使用表格。

9) 如果你使用的编译程序没有做一些象把乘法、除法转换为移位（在适当的时候）这样一些基本优化的话，那么必然有更糟的代码生成问题使你耽心，切勿着意通过移位来代替除法这样的微小改善。不要在提高效率的小技巧方面下功夫以克服差编译程序的局限性。相反，要保持代码的清晰性并找到一个好编译程序。

10) 为了确保总能保存用户的文件，在用户更改文件之前为其分配缓冲区。如果每个文件需要一个缓冲区的话，那么每次打开一个文件时都要分配一个缓冲区。如果分配失败了，就不打开文件，或将文件打开作为只读文件。但是，如果用一个缓冲区来处理所有打开的文件，那么可以在程序初始化时分配这个缓冲区。并且当在大多数时间内缓冲区悬挂着不做任何事，不要担心“浪费”存储空间。“浪费”存储空间，并确保可以保存用户的数据，这比让用户工作 5 小时，以后又由于不能分配缓冲区，数据不能保存要好的多。

第 7 章

1) 下面的代码对函数的两个输入参数 pchTo 和 pchFrom 都做了修改：

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

修改 pchTo 和 pchFrom 并没有违反与这两个参数有关的写权限，因为它们是通过值传递的，这就是说 strcpy 接受了复制的输入，因此允许 strcpy 修改它们。但是要注意，并不是所有的计算机语言（例如 FORTRAN）都是通过值来传递参数的，因此，虽然这个练习用 C 语言实现十分安全，但是，若用其它语言来实现，可能很危险。

2) strDigits 的问题是它被声明为静态指针，而不是静态缓冲区，如果所用编译程序的选择项指示编译程序把所有的字符串直接量作为常量处理，那么这个声明上的微小差别就会带来问题。支持“常量字符串直接量”选项的编译程序接受所有的字符串直接量，并把它们和程序中别的常量储存在一起。由于常量不会变更，因此这些编译程序一般都扫描所有的常量字符串，并删除复制的常量字符串。换句话说，如果 strFromUns 和 strFromInt 都将静

态指针声明为类似于“?????”的字符串，那么编译程序可能会分配一份（而不是两份）该字符串的拷贝。一些编译程序甚至更彻底，只要一个字符串和另一个字符串的尾部相匹配（例如“her”就和“mother”的尾部相匹配），就把它们组合起来存放。这样改变一个字符串就会改变其它的字符串。

解决这个问题的办法是将所有的字符串直接量作为常量处理，并限制程序代码只从它们中读出信息。如果要改变一个字符串，那么就声明一个字符缓冲区，而不是声明一个字符串指针：

```
char* strFromUns(unsigned u)
{
    static char strDigits[] = "?????";      /* 5 个字符 + '\0' */
    .....
```

但这也是冒风险的，因为这取决于程序员键入“?”标志的正确个数，并且假设尾部的空字符不会遭到破坏。使用“?”标志来占有空间并非是一种好思想，难道这个字符串真是5个“?”标志吗？如果你不能保证这一点，那么就明白了为什么应该使用不同的字符。

声明缓冲区的大小并做一次存储来替换断言是一个安全的实现：

```
char* strFromUns(unsigned u)
{
    static char strDigits[6];                /* 5 个字符 + '\0' */
    .....

    pch = &strDigits[5];
    *pch = '\0';                             /* 替换 ASSERT */
    .....
```

3) 使用 `memset` 来初始化相邻的区域，既非常冒险，又非常低效（相对于直接使用赋值而言）：

```
i = 0;          /* 置 i、j 和 k 为零 */
j = 0;
k = 0;
```

或者更简洁一些：

```
i = j = k = 0;  /* 置 i、j 和 k 为零 */
```

这些代码片断既可移植又高效，因此非常明显，甚至不再需要解释，而 `memset` 版本则是另一回事。

我不能肯定最初的程序员想通过使用 `memset` 得到什么，但可以肯定他没有得到什么好处。对于除了最优秀的编译程序以外的所有编译程序来说，调用 `memset` 的内存操作，比显示声明 `i`、`j` 和 `k` 的操作要昂贵的多，但是假设程序员使用的是一个优秀的编译程序，只要在编译时间知道要填充值的长度，这个编译程序就可以插入微小的填充，这时这个“调用”将蜕变成三个 `sizeof(int)` 的存储。这并不能使情况得到多大改进：代码依然假设编译程序

会把 i、j、k 相邻地分配到栈中，其中 k 存放在栈的最下面，代码还假设 i、j、k 互相紧连，没有任何其它多余的“垫”字节来调整变量长度以便于有效地存取。

又有谁说过变量非得放在主存储内呢？好的编译程序照例要作跨生命周期分析，将紧要信息放入寄存器并保持常驻其整个声明周期。例如，i 和 j 可能始终分配在寄存器中，根本方不到主存储器内；另一方面，k 必须分配在主存储器内，因为它的地址传给了 memset（你无法使用寄存器的地址），在这种情况下，i 和 j 依然未初始化，而 k 后面的 2*sizeof(int) 个字节将永远被错误地置为 0。

4) 当你调用或者跳到机器 ROM 的某个固定地址上时，将会面临两个危险。第一个危险是在你的机器上 ROM 可能不会有更改，但未来新型号硬件肯定会有某种改变，即使 ROM 的例程没有变更，硬件销售商有时通过驻留在 RAM 上的软件来修补 ROM 中的错误，其中修补程序是通过系统界面调用的。如果你绕过这些界面，那么也就绕过了这些修补程序。

5) 如果不需要 val，就不传递 val。所带来的问题是调用程序要对 DoOperation 的内部执行情况做个假设，就象 FILL 和 CMOVE 之间的关系一样。假定有程序员要改进 DoOperation 将其写为如下所示的代码，这样它就一直引用 val：

```
void DoOperation(operation op, int val)
{
    if( op < opPrimaryOps )
        DoPrimaryOps(op, val);
    else if( op < opFloatOps )
        DoFloatOps(op, val);
    else
        .....
}
```

当 DoOperation 引用不存在地 val 时会发生什么呢？这取决于你的操作系统。如果“val”是在栈结构的写保护部分时，代码可能会异常终止。

通过强行传递那些不再使用的变量所占据的位置，就可以使程序员难以对你的函数玩什么花样。例如，在文档中你可以写明：“每当你用 opNegAcc 来调用 DoOperation 时，就将 0 传递给 val。”一个有关存储位置的断言就可以使程序员不再折腾：

```
case opNegAcc :
    ASSERT( val == 0 );          /* 向 val 传递 0 */
    accumulator = -accumulator;
    break;
```

6) 该断言用来验证 f 是 TRUE 还是 FALSE。该断言不仅不清晰，而且更重要的是它没有必要在调试代码中如此复杂，这毕竟是以商品版本为基础得来的。该断言最好写成：

```
ASSERT( f==TRUE || f==FALSE );
```

7) 不要将所有的工作都放在一行代码上，声明一个函数指针，将一行代码一分为二，如下所示：

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

    return(pvTo);
}
```

8) 因为调用 print 例程的代码依赖于 print 代码的内存实现。如果一个程序员改变了 print 代码，并且没有意识到别的代码调用它是从入口电跳过 4 个字节实现的，则这个程序员修改代码后，可能就破坏了“print + 4”的调用者。如果你发现了这个问题就要将入口点的代码重写，加到例程中间，至少要使入口点呈现在维护人员眼前：

```
        move    r0, #PRINTER
        call    printDevice
        .....

printDisplay:  move    r0, #DISPLAY
printDevice:   .....                ;r0 == device ID
```

9) 当微型计算机还只有非常少量的只读存储器时，这种无意义的类型很受欢迎，因为每一个字节都很宝贵，而这种方式通常能节省一个或两个字节。后来，这就是一个坏习惯了。现在就把它看成很糟糕的习惯了。如果你小组中仍有人写这样的代码，让他们改正这个习惯，或让他们离开你的小组。你没有必要让这样的代码给你找麻烦。

第 8 章

第 8 章没有练习。