

第 3 章 为子系统设防

在上一章中我们说过瓶颈子程序是加入断言的绝佳之处,因为它可以使我们用很少的代码就能够进行很彻底的错误检查。这就好象一个足球场,虽然可以有 50000 个球迷来看球,但如果检票人员站在球场的入口,那么只需要几个检票人员就够了。程序中也有这样的入口,这就是子系统的调用点。

例如,对于文件系统,用户可以打开文件、关闭文件、读写文件和创建文件。这是五个基本的文件操作,这些操作通常需要大量复杂代码的支持。有了这些基本的操作,用户就可以通过对它们的调用来完成相应的文件操作,而不必操心文件目录、自由存储空间映射或者特定硬件设备(如磁盘驱动器、磁带驱动器或联网设备)的读写等实现细节。

又如,对于内存管理程序,用户可以分配内存、释放内存,有时还可以改变分配了的内存的大小。这些操作同样需要许多代码的支持。

通常,子系统都要对其实现细节进行隐藏,所隐藏的实现细节可能相当复杂。在进行实现细节隐藏的同时,子系统为用户提供了一些关键的入口点。程序员通过调用这些关键的入口点来实现同子系统的通讯。因此如果在程序中使用这样的子系统并且在其调用点加上了调试检查,那么不用花很大力气就可以进行许多的错误检查。

例如,假如要求你为标准的 C 运行时间库编写 malloc、free 和 realloc 子程序(有时必须做这件事情),那么你可能会在代码中加上断言。你可能进行了彻底的测试,并已编写了极好的程序员指南。尽管如此,我们知道在使用这些程序时,用户还是会遇到问题。那么为了对用户有所帮助,我们可以作些什么呢?

这里给出的建议是:当子系统编写完成之后,要问自己:“程序员什么情况下会错误地使用这个子系统,在这个子系统中怎样才能自动地检查出这些问题?”在正常情况下,当开始编码排除设计中的危险因素时就应该问过了这个问题。但不管怎样,还应该再问一次。

对于内存管理程序。程序员可能犯的错误是:

- 分配一个内存块并使用其中未经初始化的内容;
- 释放一个内存块但继续引用其中的内容;
- 调用 realloc 对一个内存块进行扩展,因此原来的内容发生了存储位置的变化,但程序引用的仍是原来存储位置的内容;
- 分配一个内存块后即“失去”了它,因为没有保存指向所分配内存块的指针;
- 读写操作越过了所分配内存块的边界;
- 没有对错误情况进行检查。

这些问题并不是臆想出来的,它们每时每刻都存在。更糟的是,这些问题都具有不可再现的特点,所以很难发现。出现一次,就再也看不到了。直到某一天,用户因为被上面某个常见问题搞得一筹莫展而怒气冲冲地打电话来“请”你排除相应的错误时,才会被再次发现。

确实,这些错误都很难发现。但是,这并不是说我们没有什么可以改进的事情了。断言确实很有用,但要使断言发挥作用就必须使其能够被执行到。对于我们上面列出的问题,内

存管理程序中的断言能够查出它们吗？显然不能。

在这一章中，将介绍一些用来肃清子系统中错误的其它技术。使用这些技术，可以免除许多麻烦。本章虽然以 C 的内存管理程序为例进行阐述，但所得到的结论同样适用于其它的子系统，无论是简单的链表管理程序，还是个多用户共享的正文检查工具都适用。

若隐若现，时有时无

通常，解决上述问题的方法是直接在子系统中加上相应的测试代码。但是出于两个理由，本书并没有这么做。第一个理由是我不要让例子中到处都是 malloc、free 和 realloc 的实现代码。第二个理由是用用户有时得不到所用子系统的源代码。我之所以会这么说，是因为在用来测试本书例子的六个编译程序中，有两个提供了标准的源代码。

由于用户可能得不到子系统的源代码，或者即使能够得到，这些源代码的实现也未必都相同，所以本书不是直接在子程序的源代码中加上相应的测试代码，而是利用所谓的“外壳”函数把内存管理程序包装起来，并在这层包装的内部加上相应的测试代码。这就是在得不到子系统源代码的情况下所能采用的方法。在编写外壳函数时，将采用本书前面介绍过的命名约定。

下面我们先讨论 malloc 的外壳函数。它的形式如下：

```
/* fNewMemory —— 分配一个内存块 */
flag fNewMemory(void** pv, size_t size)
{
    byte** ppb = (byte**)pv;
    *ppb = (byte*)malloc(size);
    return(*ppb != NULL);          /* 成功 */
}
```

该函数看起来比 malloc 要复杂，这主要是其指针参数 void**带来的麻烦。但如果你看到程序员调用这一函数的方法，就会发现它比 malloc 的调用形式更清晰。有了 fNewMemory，下面的调用形式：

```
if( (pbBlock) = (byte*)malloc(32) != NULL )
    成功 —— pbBlock 指向所分配的内存块
else
    不成功 —— pbBlock 等于 NULL
```

就可以被代替为：

```
if( fNewMemory(&pbBlock, 32) )
    成功 —— pbBlock 指向所分配的内存块
else
    不成功 —— pbBlock 等于 NULL
```

后一种调用形式与前一种功能相同。FNewMemory 和 malloc 之间的唯一不同，是前者把

调用“成功”标志与内存块分开返回，而后者则把这两个不同的输出结果合在一个参数中返回。无论上面哪种调用形式，如果分配成功，pbBlock 都指向所分配的内存块；如果分配失败，则 pbBlock 为 NULL。

在上一章中我们讲过，对于无定义的特性，要么应该将其从程序中消去，要么应该利用断言验证其不会被用到。如果把这一准则应用于 malloc，就会发现这个函数的行为在两种情况下无定义，必须进行相应的处理。第一种情况，根据 ANSI 标准，请求 malloc 分配长度为零的内存块时，其结果无定义。第二种情况，如果 malloc 分配成功，那么它返回的内存块的内容无定义，它们可以是零，还可以是内容随机的无用信息，不得而知。

对于长度为零的内存块，处理方法非常简单，可以使用断言对这种情况进行检查。但是对于另一种情况，使用断言能够检查出所分配内存块的内容是否有效吗？不能，这样做毫无意义。因此，我们别无选择，只能将其消去。消去这个无定义行为的明显方法，是使 fNewMemory 在分配成功时返回一个内容全部为零的内存块。这样虽然可以解决问题，但对于一个正确的程序来说，所分配内存块的初始内容并不应该影响程序的执行结果，所以这种不必要的填零增加了交付程序的负担，因此应该避免。

不必要的填充还可能隐瞒错误。

假如在为某一数据结构分配内存时，忘了对其某个域进行初始化（或者当维护程序扩展该数据结构时，忘了为新增的域编写相应的初始化代码）就会出现错误。但是如果 fNewMemory 把这些域填充为零或者其它可能有用的值，就可能隐瞒了这一错误。

不管怎样，我们还是不希望所分配内存块的内容无定义，因为这样会使错误难以再现。那么如果只有当所分配内存块中的无用信息碰巧是某个特定值时才出错，会产生什么样的结果呢？这就会在大部分的时间内发现不了错误，而程序却会由于不明显的原因不断地失败、我们可以想象一下，如果每个错误都是在某个特定的时刻才发生，要排除程序中的所有错误会多难。要是这样，程序（和测试人员）非发疯不可。暴露错误的关键是消除错误发生的随机性。

确实，如何做到这一点要取决于具体的子系统及其所涉及到的随机特性。但对于 malloc 来说，通过对其所分配的内存块进行填充，就可以消除其随机性。当然，这种填充只应该用在程序的调试版本中。这样既可以解决问题，又不影响程序的发行代码。然而必须记住，我们不希望隐瞒错误，所以用来填充内存块的值应该离奇得看起来象是无用的信息，但又应该能够使错误暴露。

例如对于 Macintosh 程序，可以使用值 0xA3。选定这个值是向自己发问以下问题的结果：什么样的值可以使非法的指针暴露出来？什么样的值可以使非法的计数器或非法的索引值暴露出来？如果新分配的内存块被当做指令执行会怎样？

在一些 Macintosh 机上，用户使用奇数的指针不能引用 16 或 32 位的值。由此可知，新选择的填充值应该是奇数。另外，如果非法的计数器或索引值较大。就会引起明显的延迟，或者会使系统的行为显得不正常，从而增大发现这类错误的可能性。因此，所选择的填充值应该是一个字节能够表示的、看起来很奇怪的较大奇数。我选择 0xA3 不仅因为它能够满足上述的要求，而且因为它还是一条非法的机器语言指令。因此如果该内存块被莫名其妙地

执行到，程序会立即瘫痪。此时如果是在系统调试程序的控制下，就会产生“undefined A-Line trap”错误。最后一点似乎有点象大海捞针，发现错误的可能性极小。但我们为什么不应该利用每个机会，不管它奏效的可能性有多么小，去自动地进行查错呢？

机器不同，所选定和填充值也可能不同。例如在基于 Intel 80x86 的机器上，指针可以是奇数，所以填充值是否奇数并不重要。但填充值的选择过程是类似的，即先来确定在什么样的情况下未经初始化的数据才会被用到，然后再千方百计使相应的情况出现。对于 Microsoft 应用，填充值可以选为 0xCC。因为它不仅较大，容易发现，而且如果被执行，能使程序安全地进入调试程序。

在 fNewMemory 中加上内存块大小的检查和内存块的填充代码之后，其形式如下：

```
#define bGarbage 0xA3
flag fNewMemory(void** ppv, size_t size)
{
    byte** ppb = (byte**)ppv;
    ASSERT(ppv!=NULL && size!=0);
    *ppb = (byte*)malloc(size);
    #ifdef DEBUG
    {
        if( *ppb != NULL )
            memset(*ppb, bGarbage, size);
    }
    #endif
    return(*ppb != NULL);
}
```

fNewMemory 的这个版本不仅有助于错误的再现，而且常常可以使错误被很容易地发现。如果在调试时发现循环的索引值是 0xA3A3，或者某个指针的值是 0xA3A3A3A3，那么显然它们都是未经初始化的数据。不止一次，我在跟踪一个错误时，由于偶然遇到了 0xA3 某种不期望的组合，结果又发现了另一个错误。

因此要查看应用中的子系统，以确定其引起随机错误的设计之处。一旦发现了这些地方，就要通过改变设计的方法把它们排除。或行在它们的周围加上相应的调试代码，最大限度地减少错误行为的随机性。

要消除随机特性 —— 使错误可再现

冲掉无用的信息

free 的外壳函数形式如下：

```
void FreeMemory(void* pv)
```

```

{
    free(pv);
}

```

根据 ANSI 标准, 如果给 free 传递了无效的指针, 其结果无定义。这似乎很合理, 可是怎样才能知道 pv 是否有效呢? 又怎样才能得出 pv 指向的是一个已分配内存块的开始地址呢? 结论是没法做到, 至少在得不到更多信息的情况下做不到。

事情还可能变得更糟。

假定程序维护一颗某种类型的树, 其 deletenode 程序调用 FreeMemory 进行结点的释放。那么如果 deletenode 中有错, 使其释放相应结点时没有对邻接分配结点中的链指针进行相应的修改, 会产生什么样的结果? 很明显, 这会使树结构中含有一个已被释放了的自由结点。但这又怎么样呢? 在大多数的系统中, 这一自由结点仍将被看作有效的树结点。

这一结果应该不会使人感到特别地惊讶。因为当调用 Free 时, 就是要通知内存管理程序该块内存空间已经不再需要, 所以为什么还要浪费时间搞乱它的内容呢?

从优化的角度看, 这样做很合理。可是它却产生了一个不好的副作用, 它使已经被释放了的无用内存信息仍然包含着好象有效的数据。树中有了这种结点, 并不会使树的遍历产生错误, 而导致相应系统的失败。相反, 在程序看来, 这颗树似乎没什么问题, 是颗有效的树。怎样才能够发现这种问题? 除非你的运气同 lotto 数卡牌戏的获胜者一样好, 否则很可能就发现不了。

“没问题”, 你可能会说, “只要在 freememory 中加上一些调试代码, 使其在调用 Free 之前把相应内存块都填上 bGarbage 就行了。那样的话, 相应内存块的内容看起来就象无用信息一样, 所以树处理程序遇到自由结点时就会跳出来”。这倒是个好主意, 但你知道要释放的内存块的大小吗? 唬, 不知道。

你可能要举手投降了, 承认完全被 FreeMemory 击败了。不是吗? 既没办法利用断言检查 pv 的有效性, 又没办法破坏被释放内存块的内容, 因为根本就不知这个内存块究竟有多大。

但是不要放弃努力, 让我们暂时假定有一个调试函数 sizeofBlock, 它可以给出任何内存分配块和大小。如果有内存管理程序的源代码, 编写一个这样的函数可能并不费事。即使没有内存管理程序的源代码, 也不必着急, 在本章稍后的内容中, 我们将介绍一种 sizeofBlock 的实现方法。

还是让我们假定已经有了 sizeofBlock 函数。利用这个函数, 在释放之前可以破坏掉相应内存块的内容:

```

void FreeMemory(void* pv)
{
    ASSERT(pv != NULL);
    #ifdef DEBUG
    {
        memset(pv, bGarbage, sizeofBlock(pv) );
    }
}

```

```

    }
    #endif
    free(pv);
}

```

该函数中的调试代码不仅对所释放内存块的内容进行了废料填充，而且在调用 `sizeofBlock` 时，还顺便对 `pv` 进行了确认。如果该指针不合法，就会被 `sizeofBlock` 查出（该函数当然可以做到这一点，因为它肯定了解每个内存分配块的细节）。

既然 `NULL` 是 `free` 的合法参数（根据 ANSI 标准，此时 `free` 什么也不做），为什么还要使用断言来检查 `pv` 是否为 `NULL`，这不是很奇怪吗？这样做的原因应该是在意料之中：我不赞成只为了实现方便，就允许将无意义的 `NULL` 指针传递给函数。这一断言就是用来对这种用法进行确认。当然，你也许有不同的观点，所以可能想把该断言去掉。但我要说的是，用户不必盲目地遵守 ANSI 标准。其他人认为 `free` 应该接受 `NULL` 指针，并不意味着你也得接受这一想法。

`realloc` 是释放内存并产生无用信息的另一个函数。下面给出它的外壳函数，它与 `malloc` 的外壳函数 `fNewMemory` 很类似：

```

flag fResizeMemory(void** ppv, size_t size)
{
    byte** ppb = (byte**)ppv;
    byte* pbResize;
    pbResize = (byte*)realloc(*ppv, sizeNew);
    if( *pbResize != NULL )
        *ppv = pbResize;
    return(*pbResize != NULL);
}

```

同 `fNewMemory` 一样，`fResizeMemory` 也返回一个状态标志，该标志表明对相应内存块大小的改变是否成功。如果 `pbBlock` 指向的是一个已经分配了的内存块，那么可以这样改变其大小。

```

if( fResizeMemory(&pbBlock, sizeNew) )
    成功 —— pbBlock 指向新的内存块
else
    不成功 —— pbBlock 指向老的内存块

```

读者应该注意到了，同 `realloc` 不一样，`fResizeMemory` 在操作失败的情况下并不返回空指针。此时，新返回的指针仍然指向原有的内存分配块，并且块内的内容不变。

有趣的是，`realloc` 函数（`fResizeMemory` 也是如此）既要调用 `free`，又要调用 `malloc`。执行时究竟调用哪个函数，取决于是要缩小还是扩大相应内存块的大小。在 `FreeMemory` 中，相应内存块的内容在被释放之前即被冲掉；而在 `fNewMemory` 中，在调用 `malloc` 之后新分配的内存块即被填上看起来很怪的“废料”。为了使 `fResizeMemory` 比较健壮，这两件事情都

必须做。因此，该函数中要有两个不同的调试代码块：

```
flag fResizeMemory(void** ppv, size_t sizeNew)
{
    byte** ppb = (byte**)ppv;
    byte* pbResize;
    #ifdef DEBUG          /* 在此引进调试局部变量 */
    size_t sizeOld;
    #endif
    ASSERT(ppb!=NULL && sizeNew!=0);
    #ifdef DEBUG
    {
        sizeOld = sizeofBlock(*ppv);
        /* 如果缩小，冲掉块尾释放的内容 */
        if(sizeNew<sizeOld)
            memset((*ppv)+sizeNew, bGarbage, sizeOld-sizeNew);
    }
    #endif
    pbResize = (byte*)realloc(*ppv, sizeNew);
    if(pbResize != NULL)
    {
        #ifdef DEBUG
        {
            /* 如果扩大，对尾部增加的内容进行初始化 */
            if(sizeNew > sizeOld)
                memset(pbResize+sizeOld, bGarbage, sizeNew-sizeOld);
        }
        #endif
        *ppv = pbResize;
    }
    return( pbResize != NULL );
}
```

为了做这两件事在该函数中似乎增加了许多额外的代码。但仔细看过就会发现，其中的大部分内容都是虚的。如花括号、`#ifdef` 伪指令和注解。就算它确实增加了许多的额外代码，也不必杞人忧天。因为调试版本本来就不必短小精悍，不必有特别快的响应速度，只要能够满足程序员和测试者的日常使用要求就够了。因此，除非调试代码会变得太大、太慢而没法使用，一般在应用程序中可以加上你认为有必要的任何调试代码。以增强程序的查错能力。

重要的是要对子系统进行考查,确定建立数据和释放数据的各种情况并使相应的数据变成无用信息。

冲掉无用的信息,以免被错误地使用

用`#ifdef`来说明局部变量很难看!

看看 `sizeOld`, 一个只用于调试的局部变量。虽然将 `sizeOld` 的说明括在`#ifdef` 序列中, 使程序变得很难看, 但这却非常重要。因为在该程序的交付版本中, 所有的调试代码都应该被去掉。我当然知道如果去掉这个`#ifdef` 伪指令, 相应的程序会变得更加可读, 而且程序的调试版本和交付版本会同样地正确。但这样做的唯一问题是在其交付版本中, `sizeOld` 虽被说明, 但却没被使用。

在程序的交付版本中声明但不使用 `sizeOld` 变量, 似乎没有问题。但事实并非如此, 这样做会引起严重的问题。如果维护程序员没有注意到 `sizeOld` 只是一个调试专用的变量, 而把它用在了交付版本中, 那么由于它未经初始化, 可能就会引起严重的问题。将 `sizeOld` 的声明用`#ifdef` 伪指令括起来, 就明确地表明了 `sizeOld` 只是一个调试专用的变量。因此, 如果程序员在程序的非调试代码 (即使是`#ifdef`) 中使用了 `sizeOld`, 那么当构造该程序的交付版本时就会遇到编译程序错误。这等于加了双保险。

使用`#ifdef` 指令来除去调试用变量虽然使程序变得很难看, 但这种用法可以帮助我们消除一个产生潜在错误的根源。

产生移动和震荡的程序

假定程序不是释放掉树结构的某个结点, 而是调用 `fResizeMemory` 将该结点扩大, 以适应变长数据结构的要求。那么当 `fResizeMemory` 对该结点进行扩展时, 如果移动了该结点的存储位置, 就会出现两个结点: 一个是在新位置的真实结点, 另一个是原位置留下的不可用的无用信息结点。

这样一来, 如果编写 `expandnode` 的程序员没有考虑到当 `fResizeMemory` 在扩展结点时会引起相应结点的移动这种情况, 会出现什么问题呢? 相应树结构的状态会不会仍然不变, 即该结点的邻接结点仍然指向虽然已被释放但看起来似乎仍然有效的原有内存块? 扩展之后的新结点会不会漂浮在内存空间中, 没有任何的指针指向它? 事实确实会这样, 它可能产生看起来好象有效但实际上是错误的树结构, 并在内存中留下一块无法访问到的内存块。这样很不好。

我们可以想到通过修改 `fResizeMemory`, 使其在扩展内存块引起存储位置移动的情况下, 冲掉原有的块内容。要达到这一目的, 只需简单地调用 `memset` 即可:

```
flag fResizeMemory(void** ppv, size_t sizeNew)
{
    .....
}
```

```

    pbResize = (byte*)realloc(*ppb, sizeNew);
    if(pbResize != NULL)
    {
        #ifdef DEBUG
        {
            /* 如果发生移动，冲掉原有的块内容 */
            if(pbResize != *ppb)
                memset(*ppb, bGarbage, sizeOld);
            /* 如果扩大，对尾部增加的内容进行初始化 */
            if(sizeNew > sizeOld)
                memset(pbResize+sizeOld, bGarbage, sizeNew-sizeOld);
        }
        #endif
        *ppb = pbResize;
    }
    return( pbResize != NULL );
}

```

很遗憾，这样做不行。即使知道原有内存块的大小和位置，也不能破坏原有内存块的内容，因为我们不知道内存管理程序会对被其释放了的内存空间进行如何的处理。对于被释放了的内存空间，有些内存管理程序并不对其做些什么。但另外一些内存管理程序，却用它来存储自由空间链或者其它的内部实现数据。这一事实意味着一旦释放了内存空间，它就不再属于你了，所以你不应该再去动它。如果你动了这部分内存空间，就有破坏整个系统的危险。

举一个非常极端的例子，有一次当我正在为 Microsoft 的内部 68000 交叉汇编程序增加新功能时，Macintosh Word 和 Excel 的程序员请求我去帮助他们查明一个长期以来总是使系统偶然失败的错误。检查这个错误的难点在于虽然它并不经常发生，但却总是发生，因此引起了人们的重视。我不想谈过多的细节，但折腾了几周之后我才找到了使这个错误重现的条件，而找出该错误的实际原因却只用了三天的时间。

找出使这个错误重现的条件花了我很长时间，但我还是不清楚是什么原因引起了这个错误。每当我查看相应的数据结构时，它们看起来似乎都完全没有问题。我没想到这些所谓完全没有问题的数据结构，实际上竟是早先调用 `realloc` 遗留下的无用信息！

然而，真正的问题还不在于发现这个错误的准确原因花了我多长的时间，而在于为了找出使这个错误重现的条件花了那么多的时间。`realloc` 在扩大内存块时不但确实会移动相应内存块的位置，而且原有的内存块必须被重新分配并被填写上新的数据。在汇编程序中，这两种情况都很少发生。

这使我们得出了编写无错代码的另一个准则：“不要让事情很少发生。”因此我们需要确定子系统中可能发生哪些事情，并且使它们一定发生和经常发生。如果发现子系统中有极罕

见的行为，要千方百计地设法使其重现。

你有过跟踪错误跟到了错误处理程序中,并且感到“这段错误处理程序中的错误太多了，我敢肯定它从来都没有被执行过”这种经历吗？肯定有，每个程序员都有过这种经历。错误处理程序之所以往往容易出错，正是因为它很少被执行到。

同样，如果不是 `realloc` 扩大内存块时使原有存储位置发生移动这种现象很罕见，这一汇编程序中的错误在几个小时内就可以被发现，而用不着要耗上几年。可是，怎样才能使 `realloc` 经常地移动内存块呢？回答是做不到，至少在相应操作系统没有提供支持的情况下做不到。尽管如此，但我们却能够模拟 `realloc` 的所作所为。如果程序员调用 `fResizeMemory` 扩大了某个内存块，那么可以通过先建一个新的内存块，然后再把原有内存块的内容拷贝到这个新块中，最后释放掉原有内存块的方法，准确地模拟出 `realloc` 的全部动作。

```
Flag fResizeMemory(void** ppv, size_t sizeNew)
{
    byte** ppb = (byte**)ppv;
    byte* pbResize;
#ifdef DEBUG
    size_t sizeOld;
#endif
    ASSERT(ppb!=NULL && sizeNew!=0);
#ifdef DEBUG
    {
        sizeOld = sizeofBlock(*ppv);
        /* 如果缩小，先把将被释放的内存空间填写上废料
         * 如果扩大，通过模拟 realloc 的操作来迫使新的内存块产生移动
         * （不让它在原有的位置扩展）如果新块和老块的长度相同，不
         * 做任何事情
         */
        if(sizeNew < sizeOld)
            memset((*ppv)+sizeNew, bGarbage, sizeOld-sizeNew);
        else if(sizeNew > sizeOld)
        {
            byte* pbNew;
            if( fNewMemory(&pbNew, sizeNew) )
            {
                memcpy(pbNew, *ppv, sizeOld);
                FreeMemory(*ppv);
                *ppv = pbNew;
            }
        }
    }
#endif
}
```

```

    }
}
#endif
pbResize = (byte*)realloc(*ppb, sizeNew);
.....
}

```

在上面的程序中，所增加的新代码只有在相应的内存块被扩大时，才会被执行。通过在释放原有内存块之前分配一个新的内存块，可以保证只要分配成功，相应内存块的存储位置就会被移动。如果分配失败，那么所增加的新代码相当于一个很大的空操作指令。

但是，请注意上面程序中所增加的新代码不仅使相应内存块不断地移动而，还顺便冲掉了原有内存块的内容。当它调用 `FreeMemory` 释放原有的内存块时，该内存块的内容即被冲掉。

现在你可能会想 既然上面的程序是用来模拟 `realloc` 的,那它为什么还要调用 `realloc` 呢？而且在所增加的代码中加入一条 `return` 语句，例如：

```

if( fNewMemory(&pbNew, sizeNew) )
{
    memcpy(pbNew, *ppb, sizeOld);
    FreeMemory(*ppb);
    *ppb = pbNew;
    return(TRUE);
}

```

不是可以提高其运行速度吗？

我们为什么不这样做呢？我们可做到这点,但切记不要这么做,因为它是个不良的习惯。要记住调试代码是多余的代码，而不是不同的代码。除非有非常值得考虑的理由，永远应该执行原有的非调试代码，即使它在加入了调试代码之后已经变得多余。毕竟查出代码错误的最好方法是执行代码，所以要尽可能地执行原有的非调试代码。

有时在我向程序员解释这些概念时，他们会反驳说：“总是移动内存块正如水远不移动内存块一样有害，你已经走到了另一个极端。”他们确实非常机敏，因此有必要解释一下。

假如在程序的调试版本和交付版本中都总是做某件事情，那么它确实如同永远不做一样有害。但在这个例子中，`fResizeMemory` 实际上并不紧张，尽管其调试版本是那样不屈不挠地对内存块进行移动，就好象吃了安非他明一样。

如果某事件很少发生并没有什么问题，只要在程序的交付版本和调试版本中不少发生就行。

如果某件事甚少发生的话，设法使其经常发生

保存一个日志，以唤起你的注意

从调试的端点看，内存管理程序的问题是当第一次创建内存块时知道其大小，但随后几乎马上就会失去这一信息，除非在某个地方保存了一个有关的记录。我们已经看到函数 `sizeofBlock` 的价值很大，但如果能够知道已分配内存块的数目及其在内存中的具体存储位置，用处会更大。假如能够知道这些信息，那么不管指针的值是什么，我们都能够确定它是否有效。如果能这样，该有多大的用处，尤其是对于函数参数的确认。

假定我们有函数 `fValidPointer`，该函数有指针 `pv` 和大小 `size` 两个参数；当 `pv` 实际指向的内存分配块正好有 `size` 个字节时，该函数返回 `TRUE`。利用这一函数我们可以为常用的子程序编写出更加严格的专用版本。例如，如果发现内存分配块的部分内容常常被填掉，那么我们可以绕过对指针检查得不太严格的 `memset` 函数，而调用自己编写的 `FillMemory` 程序。该程序能够对其指针参数进行更加严格的确认：

```
void FillMemory(void* pv, byte b, size_t size)
{
    ASSERT(fValidPointer(pv, size));
    Memset(pv, b, size);
}
```

通过应用 `fValidPointer`，该函数可以保证 `pv` 指向的是一个有效的内存块。而且，从 `pv` 到该内存块的尾部至少会有 `size` 个字节。

如果愿意的话我们可以在程序的调试版本中调用 `FillMemory`，而在其交付版本中直接调用 `memset`。要做到这一点，只需在其交付版本中包括如下的全局宏定义：

```
#define FillMemory(pb, b, size) memset((pb), (b), (size))
```

这些内容已经有点离题了。

这里一直强调的是如果在程序的调试版本中保存额外的信息，就经常可以提供更强的错误检查。

到目前为止，我们介绍了在 `FillMemory` 和 `fResizeMemory` 中使用 `sizeofBlock` 填充内存块的方法。但这种方法同通过保存一个含有所有分配内存块信息的记录所能做到的相比，只是个相对“弱”的错误发现方法。

同前面一样，我们仍然假定会遇到最坏的情况：从相应的子系统本身，我们得不到关于分配内存块的任何信息。这意味着通过内存管理程序，我们得不到内存块的大小，不知道指针是否有效，甚至不知道某个内存块是否存在或者已经分配了多少个内存块。因此如果程序中需要这些信息，就必须自己提供出来。这就是说，在程序中得保存一个某种类型的分配日志。至于如何保存这个日志并不重要，重要的是在需要这些信息时就能够得到。

维护日志的一种可能方法是：当在 `fNewMemory` 中分配一个内存块时，为日志信息也分配一个内存块；当在 `fFreeMemory` 中释放一个内存块时，还要释放相应的日志信息；当在 `fResizeMemory` 中改变了内存块的大小时，要修改相应的日志信息，使它反映出相应内存块的新大小和新位置。显然，我们可以把这三个动作封装在三个不同的调试界面中：

```

/* 为新分配的内存块建立一个内存记录 */
flag fCreateBlockInfo(byte* pbNew, size_t sizeNew);
/* 释放一个内存块对应的日志信息 */
void FreeBlockInfo(byte* pb);
/* 修改现有内存块对应的日志信息 */
void UpdateBlockInfo(byte* pbOld, byte* pbNew, size_t sizeNew);

```

当然，只要它们不使相应系统的运行速度降低到无法使用的程度，这三个程序维护日志信息的方法就不很重要。读者在附录 B 中可以找到上述函数的实现代码。

对 FreeMemory 和 fResizeMemory 进行修改，使其调用适当的子程序非常简单。修改后的 FreeMemory 变成了如下形式：

```

void FreeMemory(void* pv)
{
    #ifdef DEBUG
    {
        memset(pv, bGarbage, sizeofBlock(pv));
        FreeBlockInfo(pv);
    }
    #endif
    free(pv);
}

```

在 fResizeMemory 中，如果 realloc 成功地改变了相应内存块的大小，那么就调用 UpdateBlockInfo（如果 realloc 失败，自然就没有什么要修改的内容）。fResizeMemory 的后一部分如下：

```

flag fResizeMemory(void** ppv, size_t sizeNew)
{
    .....
    pbResize = (byte*)realloc(*ppb, sizeNew);
    if(pbResize != NULL)
    {
        #ifdef DEBUG
        {
            UpdateBlockInfo(*ppb, pbResize, sizeNew);
            /* 如果扩大，对尾部增加的内容进行初始化 */
            if(sizeNew > sizeOld)
                memset(pbResize+sizeOld, bGarbage, sizeNew-sizeOld);
        }
        #endif
    }
}

```

```

        *ppb = pbResize;
    }
    return(pbResize != NULL);
}

```

fNewMemory 的修改相对要复杂一些，所以把它放到了最后来讨论。当调用 fNewMemory 分配一个内存块时系统必须分配两个内存块：一个用来满足调用者的请求，另一个用来存放相应的日志信息。只有在两个内存块的分配都成功时，fNewMemory 的调用才会成功。如果不这样规定就会使某些内存块没有日志信息。要求内存块必须有对应的日志信息非常重要，因为如果没有日志信息，那么在调用对指针参数进行确认的函数时，就会产生断言失败。

在下面的代码中我们将会看到，如果 fNewMemory 成功地进行了用户请求空间的分配，但相应日志内容所需内存的分配失败，该函数会把第一个内存块释放掉，并返回一个内存块分配失败标志。这样做可以使所分配的内存内容与相应的日志信息同步。

fNewMemory 的代码如下：

```

flag fNewMemory(void** ppv, size_t size)
{
    byte** ppb = (byte**)ppv;
    ASSERT(ppv!=NULL && size!=0);
    *ppb = (byte*)malloc(size);
    #ifdef DEBUG
    {
        if(*ppb != NULL)
        {
            memset(*ppb, bGarbage, size);
            /* 如果无法创建日志块信息，
             * 那么模拟一个总的内存分配错误。
             */
            if( !fCreateBlockInfo(*ppb, size) )
            {
                free(*ppb);
                *ppb = NULL;
            }
        }
    }
    #endif
    return(*ppb != NULL);
}

```

就是这样。

现在，我们有了相应内存系统的完整记录。利用这些信息，我们可以很容易地编写出象 `sizeofBlock` 和 `fValidPointer`（见附录 B）这样的函数，以及任何其它的有用函数。

保存调试信息，以便进行更强的错误检查

不要等待错误发生

直到目前为止，我们所做的一切努力只能帮助用户注意到错误的发生。这固然不错，但它还不能自动地发现错误。以前面讲过的 `deletenode` 函数为例，如果该函数调用函数 `FreeMemory` 释放某个结点时，在相应的树结构中留下了指向已释放内存空间的指针，那么在这些指针永远都不会被用到的情况下，我们能够发现这个问题吗？不，不能。又如，如果我们在函数 `fResizeMemory` 中忘了调用 `FreeMemory`，又会怎样？

```
.....  
  
if( fNewMemory(&pbNew, sizeNew) )  
{  
    memcpy(pbNew, *ppb, sizeOld)  
    /* FreeMemory(*ppb); */  
    *ppb = pbNew;  
}
```

结果会在该函数中产生一个难解的错误。说它难解，是因为表面看起来，什么问题都没有。但我们每次执行这段程序，就会“丢失”一块内存空间。因为在把 `pbNew` 赋给 `*ppb` 时，这个唯一指向该内存块的指针被冲掉了。那么该函数中的调试代码能够帮助我们查出这个错误吗？根本不能。

这些错误与前面讲的错误不同，因为它们不会引起任何不合法情况的发生。正如匪徒根本没打算出城，路障就没用了一样，在相应数据没被用到的情况下相应的调试代码也没用，因为它查不出这些错误。查不到错误并不意味着这些错误不存在，它们确实存在只不过我们没有看到它们——它们“隐藏”得很深。

要找出这些错误，就得象程序员一样，对错误进行“挨门挨户”的搜查。不要等待错误自己暴露出来，要在程序中加上能够积极地寻找这种问题的调试代码。

对于上面的程序我们遇到两种情况。第一种情况，我们得到一个指向已被释放了的内存块的“悬挂指针”；第二种情况，我们分配了一个内存块，但却没有相应的指针指向它。这些错误通常都很难发现，但是如果我们在程序中一直保存有相应的调试信息，就可以比较容易地发现它们。

让我们来看看人们是怎样检查其银行财务报告书中的错误：我们自己有一个拨款清单，银行有一个拨款清单。通过对这两个清单进行比较，我们就可以发现其中的错误、这种方法同样可以用来发现悬挂指针和内存块丢失的错误。我们可以对已知指计表（保存在程序的调试信息中）进行比较，如果发现指针所引用是尚未分配的内存块或者相应的内存块没有被任

何指针所指向，就肯定出了问题。

但程序员，尤其是有经验的程序员总是避免直接对存储在每个数据结构中的每个指针进行检查。因为要对程序中的所有数据结构以及存储在其中的所有指针进行跟踪，如果不是不可能的话，似乎也非常困难。实际的情况是，即使某些编写得很差的程序，也是为指针再单独分配相应的内存空间，以便于对其进行检查。

例如，68000 汇编程序可以为 753 个符号名分配内存空间，但它并没有使用 753 个全局变量对这些符号名进行跟踪，那样会显得相当的愚蠢。相反，它使用的是数组、散列表、树或者简单的链表。因此，尽管可能会有 753 个符号名，但利用循环可以非常简单地遍查这些数据结构，而且这也费不了多少代码。

为了对相应的指针表和对应的调试信息进行比较，我定义了三个函数。这三个函数可以同上节给出的信息收集子程序（读者在附录 B 中可以找到它们的实现代码）配合使用：

```
/* 将所有的内存块标记为“尚未引用” */
void ClearMemoryRefs(void);
/* 将 pv 所指向的内存块标记为“已被引用” */
void NoteMemoryRef(void* pv);
/* 扫描引用标志，寻找被丢失的内存块 */
void CheckMemoryRefs(void);
```

这三个子程序的使用方法非常简单。首先，调用 ClearMemoryRefs 把相应的调试信息设置成初始状态。其次，扫描程序中的全局数据结构，调用 NoteMemoryRef 对相应的指针进行确认并将其指向的内存块标记为“已被引用”。在对程序中所有的指针这样做了之后，每个指针都应该是有效的指针，所分配的每个内存块都应该标有引用标记。最后，调用 CheckMemoryRefs 验证某个内存块没有引用标记，它将引发相应的断言，警告用户相应的内存块是个被丢失了的内存块。

下面我们看看在本章前面介绍的汇编程序中，如何使用这些子程序对该汇编程序中使用的指针进行确认。为了简单起见，我们假定该汇编程序所使用的符号表是棵二叉树，其每个结点的形式如下：

```
/* “symbol”是一个符号名的结点定义。
 * 对于用户汇编源程序中定义的每个符号，
 * 都分配一个这样的结点
typedef struct SYMBOL
{
    struct SYMBOL* psymRight;
    struct SYMBOL* psymLeft;
    char* strName;          /* 结点的正文表示 */
    .....
}symbol;                  /* 命名方法：sym, *psym */
```

其中只给出了三个含有指针的域。头两个域是该结点的左子树指针和右子树指针，第三

个域是以零字符结尾的字符串。在我们调用 ClearMemoryRefs 时，该函数完成对相应树的遍历，并将树中每个指针的有关信息记载下来。完成这些操作的代码破封装在一个调试专用的函数 NoteSymbolRefs 中，该函数的形式如下：

```
void NoteSymbolRefs(symbol* psym)
{
    if(psym!=NULL)
    {
        /* 在进入下层结点之前先确认当前的结点 */
        NoteMemoryRef(psym);
        NoteMemoryRef(psym->strName);
        /* 现在确认当前结点的子树 */
        NoteSymbolRefs(psym->psymRight);
        NoteSymbolRefs(psym->psymLeft);
    }
}
```

该函数对符号表进行先序遍历，记下树中每个指针的情况。通常，符号表都被存储为中序树，因此相应地应该对其进行中序遍历。但我这里使用的是先序遍历，其原因是我想在引用 psym 所指内容之前，对其有效性进行确认，这就要求进行先序遍历。如果进行中序遍历或者后序遍历，就会在企图对 psym 进行确认之前引用到其指向的内容，从而可能在进行了多次的递归之后，使程序失败。当然，这样也可以发现错误。但跟踪一个随机的错误和跟踪一个断言的失败，你宁愿选择哪一个呢？

在为其它的数据结构编写了“Note-Ref”这一类的例程之后，为了便于在程序的其它地方进行调用，应该把它们合并为一个单独的例程。对于这个汇编程序，相应的例程可以有如下的形式

```
#ifdef DEBUG
void CheckMemoryIntegrity(void)
{
    /* 将所有的内存块标记为“尚未引用” */
    ClearMemoryRefs();
    /* 记载所有的已知分配情况 */
    NoteSymbolRefs(psymRoot);
    NoteMacroRefs();
    .....
    NoteCacheRefs();
    NoteVariableRefs();
    /* 保证每个指针都没有问题 */
    CheckMemoryRefs();
}
```

```
}  
#endif
```

最后一个是：“应该在什么时候调用这个例程？”显然，我们应该尽可能多地调用这个例程，但其实这要取决于具体的需要。至少，在准备使用相应的子系统之前，应该调用这一例程对其进行一致性检查。如果能在程序等待用户按键、移动鼠标或者拨动硬件开关期间，对相应的子系统进行检查，效果会更好。总之，要利用一切机会去捕捉错误。

建立详尽的子系统检查并且经常地进行这些检查

非确定性原理

我经常向程序员解释使用调试检查是怎么回事。在我解释的过程中，有时他或她会因为所加入的调试代码会对原有的代码产生妨碍，而对增加这种代码可能带来的不良后果的严重程度表示担忧。这又是一个与 Heisenberg 提出的“非确定性原理”有关的问题。如果读者对这一问题感兴趣，请继续读下去。

毫无疑问，所加入的调试代码会引起程序交付版本和调试版本之间的区别。但只要在加入调试代码时十分谨慎，并没有改变原有程序的内部行为，那么这种区别就不应该有什么问题。例如虽然 `fResizeMemory` 可能会很频繁地移动内存块，但它并没有改变该函数的基本行为。同样，虽然 `fNewMemory` 所分配的内存空间会比用户所请求的多（用于存放相应的日志信息），但这对用户程序也不应该有什么影响。（如果你指望请求分配 21 个字节，`fNewMemory` 或者 `malloc` 就应该恰好为你分配 21 个字节，那么无论有没有调试代码你都会遇到麻烦。因为要满足对齐要求，内存管理程序分配的内存总是要比用户请求的量多）

另一个问题是调试代码会增加应用程序的大小，因此需要占用更多的 RAM。但是读者应该记得，建立调试版本的目的是捕捉错误，而不是最大限度地利用内存。对于调试版本来说，如果无法装入最大的电子表格，无法编辑最大可能的文档或者没法做需要大量内存的工作也没有什么关系，只要相应的交付版本能够做到这些就可以。使用调试版本会遇到的最坏情况，是相对交付版本而言，运行不久便耗尽了可用的内存空间，使程序异常频繁地执行相应的错误处理代码；最好的情况，是调试版本很快就捉住了错误，几乎没有或者花费很少的调试时间。这两种极端情况都有价值。

一点就透

Robert Cialdini 博士在其“*Influence: How and Why people Agree to Things*”一书中指出：如果你是个售货员，那么当顾客来到你负责的男装部准备购买毛衣和套装时，你应该总是先给顾客看套装然后再给顾客看毛衣。这样做的理由是可以增加销售额，因为在顾客买了一件\$500 元的套装之后，相比之下，一件\$80 元的毛衣就显得不那么贵了。但是如果你先给顾客看毛衣，那么\$80 元一件的价格可能会使其无法接受，最后也许你只能卖出一件\$30 元的毛衣。任何人只要花 30 秒的时间想一想，就会明白这个道理。可是，又有多少人

花时间想过这一问题呢？

同样，一些程序员可能会认为，bGarbage 选为何值并不重要，只要从过去用过的数中随便挑一个就行了。另外一些程序员也可能会认为，究竟是按先序、中序还是后序对符号表进行递归遍历并不重要。但正如我们在前面指出的那样，有些选择确实比另外的一些选择要好。

如果可以随意地选择实现细节的话，那么在做出相应的选择之前，要先停下来花 30 秒钟考查一下所有的可能选择。对于每一种选择，我们都要问自己：“这种选择是会引起错误，还是会帮助发现错误？”如果对 bGarbage 的取值问过这一问题的话，你就会发现选择 0 会引起错误而选择 0xA3 之类的值则会帮助我们发现错误。

仔细设计程序的测试代码，任何选择都应该经过考虑

无需知道

在对子系统进行测试时，为了使用相应的测试程序，你可能遇到过需要了解这些测试程序各方面内容的情况。fValidPointer 的使用就是这样一个例子。如果你不知道有这样一个函数，就根本不会去使用它。然而，最好的测试代码应该是透明的代码，不管程序员能否感觉到它们的存在，它们都会起作用。

假定一个没有经验的程序员或者某个对项目不熟悉的人加入了项目组。在根本不知道 fNewMemory、fResizeMemory 和 FreeMemory 的内部有相应测试代码的情况下，他不是照样可以随意地在程序中使用这些函数吗？

那么如果他没有意识到 fResizeMemory 会引起内存块的移动，并因此在其程序中产生了类似于前述汇编程序中出现的错误，那么会发生什么现象呢？他需要因为执行了相应的一致性检查程序并产生了断言“illegal pointer”而对一致性检查程序的内容有所了解吗？

如果他创建并随即丢失了一个内存块，又会怎样呢？这时同样会执行相应的一致性检查程序并产生断言“lost memory”。也许，他甚至连什么叫做“lost memory”都不知道。但事实是，他并不需要知道这个，相应的检查就可以起作用。更妙的是，通过跟踪这一错误不用向有经验的程序员请教也可以学到与内存丢失有关的内容。

这就是精心设计子系统测试代码的好处——当测试代码将错误限制在一个局部的范围之内后，就通过断言把错误抓住并送到“广播室”，把正常的工作打断。对于程序员来说，这真是再好不过的反馈。

努力做到透明的一致性检查

我们交付的不是调试版本

在这一章中，我确实给内存管理程序加上了许多调试代码。对此，一些程序员可能会认为：“在程序中加入调试代码似乎很有用，但象这样把所有的检查都加上并且还包括了对日

志信息的处理，就太过分了。”我得承认，我也有过这种感觉。

以前我也对给程序加上这么多降低效率的调试代码很反感，但不久我就认识到了自己的错误。在程序的交付版本中加上这种调试代码是会断送它的市场前途，但我们并没有在其交付版本中增加任何的测试代码，这些代码只是被用在了它的调试版本中。确实，调试代码会降低调试版本的运行速度。但使你的零售产品瘫在用户那儿，或者为了帮助查错使你的调试版本运行得稍慢，哪一种情况更糟糕呢？我们不应该担心调试版本的效率，因为毕竟顾客不会使用程序的调试版本。

重要的是要在感情上区分程序的调试版本和交付版本。调试版本事用来发现错误的，而交付版本则是用来取悦顾客的。因此在编码时，对这两个版本所作的权衡也会相当不同。

记住，只要相应的交付版本能够满足顾客的大小和速度要求，就可以对调试版本做你想做的任何事情。如果为内存管理程序加上日志程序可以帮助你发现各种难于捕捉的错误，那么就会皆大欢喜。顾客可以得到一个充满活力的程序而你不用费很多的时间和精力就可以发现错误。

Microsoft 的程序员总是在其程序中加上相应的调试代码。例如，Excel 就含有一些内存子系统的测试程序（它们比我们这里介绍的还要详尽）。它有单元表格一致性检查程序；它有人为产生内存失败的机制，使程序员可以强制程序执行“内存空间耗尽”的错误处理程序；它还有许多的其它检查程序。这不是说 Excel 的交付版本从来没有错误，它确实有，但这些错误很少出现在通过了详尽的子系统检查的代码中。

同样，虽然我们在这一章中给内存管理程序增加了许多的代码，但增加的所有代码都是用来构造 fNewMemory、FreeMemory 和 fResizeMemory，我们没给这些函数的调用程序增加任何东西，也没给 malloc、free 和 realloc 的内部支持代码（它们可以非常重要）增加任何东西。甚至增加调试代码所引起的速度下降，也并非如想象的那样糟糕。如果 Microsoft 公司的统计结果具有代表性的话，程序调试版本（充满了断言和子系统测试）的速度大约应该是相应交付版本的一半。

**不要把对交付版本的约束应用到相应的调试版本上
要用大小和速度来换取错误检查能力**

确有其事

为了发现更多的错误，过去 Microsoft 总是把其开发的应用程序的调试版本送给 β 测试者进行 β 测试。但当基于产品的 β 调试版本对产品进行评论的“Pre-release”周刊出现，并且说其程序虽然非常好，但就是慢得和鼻涕虫一样之后，他们不再至少是暂时不再提供产品的 β 调试版本。这个事实告诫我们不要把调试版本送到测试场所，或者在这样做之前要把调试版本中影响性能的内部调试检查代码全部清除。

小结

在这一章中，我们介绍了六种增强内存子系统的方法。这些方法虽然是针对内存子系统提出来的，但其中的观点同样适用于其它的子系统。大家可以想象得出，在程序自己具有详尽的确认能力之后错误要想悄悄地溜入这种程序，简直比登天还难。同样，假如在我前面讲过的汇编程序中用上了这些调试检查，那么通常要花上几年才能发现的 `realloc` 错误，在相应代码第一次编写的几个小时或者几天之内就可以被自动地发现。不管程序员的技术很高，还是没有经验，这些测试代码都能够抓住这个错误。

事实上，这些测试代码能够抓住所有的这类错误。而且是自动地抓住，不靠运气，也不靠技巧

这就是编写无错代码的方法。

要点：

- 考查所编写的子系统，问自己：“在什么样的情况下，程序员在使用这些子系统时会犯错误。”在子系统中加上相应的断言和确认检查代码，以捕捉难于发现的错误和常见的错误。
- 如果不能使错误不断重现，就无法排除它们。找出程序中可能引起随机行为的因素，并将它们从程序的调试版本中清除。把目前尚“无定义”的内存单元置成了某个常量值，就可能产生这种错误。在这种情况下，如果程序在该单元被正确地定义为某个值之前引用了它的内容，那么每次执行这部分错误的代码，都会得到同样的错误结果。
- 如果所编写的子系统释放内存（或者其它的资源），并因此产生了“无用信息”，那么要把它搅乱，使它真的象无用信息。否则，这些被释放了的数据就有可能仍被使用，而又不会被注意到。
- 类似地，如果在所编写的子系统中某些事情可能发生，那么要为该子系统加上相应的调试代码，使这些事情一定发生。这样可以增大查出通常得不到执行的代码中的错误的可能性。
- 尽力使所编写的测试代码甚至在程序员对其没有感觉的情况下亦能起作用。最好的测试代码是不用知道其存在也能起作用的测试代码。
- 如果可能的话，把测试代码放到所编写的子系统中，而不要把它放到所编写子系统的外层。不要等到进行了系统编码时，才考虑其确认方法。在子系统设计的每一步，都要考虑“如何对这一实现进行详尽的确认”这一问题。如果发现这一设计难于测试或者不可能对其进行测试，那么要认真地考虑另一种不同的设计，即使这意味着用大小或速度作代价去换取该系统的测试能力也要这么做。
- 在由于速度太慢或者占用的内存太多而抛弃一个确认测试程序之前，要三思而后行。切记，这些代码并不是存在于程序的交付版本中。如果发现自己正在想：“这个测试程序太慢、太大了”，那么要马上停下来问自己：“怎样才能保留这个测试程序，并使它既快又小？”

练习

- 1) 如果在进行代码测试时偶然碰到了 0xA3 的某种组合构成的数据，那么这一数据可能是未经过初始化的数据，或者是已被释放了的数据。怎样才能修改相应的调试代码，使我们可以比较容易地确定所发现的数据是哪一类？
- 2) 程序员编写的代码有时会对所分配内存块上界之外的内存单元进行填充。请给出增加相应的内存子系统检查，使其能够对这类错误报警的方法。
- 3) 虽然 CheckMemoryIntegrity 程序被用来对悬挂指针错误进行检查，但在有些情况下，该程序检查不出这种错误。例如，假定一个函数调用了 FreeMemory，但由于该函数的错误，某个指针被悬挂起来，即该指针指向的内存块已被 FreeMemory 释放掉。现在我们进一步假定在该指针被确认之前，某个函数调用 fNewMemory 对这块刚释放不久的内存块进行了再分配。这样一来，刚才被悬挂起来的指针又指向了新分配的内存块。但是，这个内存块已经不是原来那个内存块了。因此，这是个错误。但对于 CheckMemoryIntegrity 来说却一切都很正常，并没有什么不合法。假如这个错误在你的项目中比较常见，那么怎样增强该程序才能使其查出这个问题呢？
- 4) 利用 NoteMemoryRef 程序，我们可以对程序中的所有指针进行确认。但是，我们如何对所分配内存块的大小进行确认呢？例如，假定指针指向的是一个含有 18 个字符的字符串，但所分配内存块的长度却小于 18。或者在相反的情况下，程序认为所分配的内存块有 15 个字节，但相应的日志信息表明为其分配了 18 个字节。这两种情况都是错误的。怎样加强相应的一致性检查程序，使其能够查出这种问题？
- 5) NoteMemoryRef 可以使我们把一个内存块标为“已被引用”，但利用它我们无法知道引用该内存块的指针数目是否超过了其应有的数目。例如，双向链表的每个结点只应该有两个引用。一个是前向指针，另一个是后向指针。但在大多数的情况下，每个内存块只应该有一个指针在引用着它。如果有多个指针同时引用一个内存块，那么一定是程序中什么地方出了错误。如何改进相应的一致性检查程序，使其对某些内存块允许多个指针对其进行同时的引用；但对另外一些内存块仍不允许多个指针对其进行同时的引用，并在这种情况发生时，引发相应的断言？
- 6) 本章自始至终所谈的都是为了帮助程序员检查错误，可以在相应的内存系统中加上调试代码。但是，我们可不可以增加对测试者有所帮助的代码呢？测试者知道程序经常会对错误情况进行不正确的处理，那么如何为测试者提供模拟“内存空间耗尽”这一条件的能力呢？

课题：

考查你项目中的主要子系统，看看为了检查出与使用这些子系统有关的常见错误，可以实现哪种类型的调试检查？

课题：

如果没有所用操作系统的调试版本，那么尽可能买一个。如果买不到，就利用外壳函数自己写一个。如果你助人为乐，那么请使所编出的代码（以某种方式）可以为其它的开发者所用。