编程精粹

——— Microsoft 编写优质无错 C 程序秘诀

Writing Clean Code

——— Microsoft Techniques for Developing Bug-free C Programs

Steve Maguire 著

姜静波 佟金荣 译

麦中凡 校

电子工业出版社

这份电子书籍由 PC Home 俱乐部、C++ Bulider 讨论区数位网友分别整理完成，基本上完全按照所据译本原貌，极少数文字为通顺起见稍作修改。由于并非一人整理完成，书中例程各章节代码书写风格可能稍有不同，如指针声明，以下两种写法都存在：

```
void* pv;          //  ‘*’号与类型说明符相连
void *pv;          //  ‘*’号与变量名相连
```

抱歉为阅读带来了麻烦。

如果各位在阅读这份电子书籍时发现错误，请E-mail至wizard@citiz.net ，我们会尽快加以修正。

原文书名：

《Writing Clean Code —— Microsoft Techniques for Developing Bug-free C Programs》

Steve maguire 著

Microsoft Press 出版

所据译本：

《编程精粹 —— Microsoft 编写优质无错 C 程序秘诀》

姜静波、佟金荣 译，麦中凡 校

电子工业出版社 出版

整理：

Solmyr：序、某些背景、命名约定、引言、第 1、2、3、8 章、后记、参考文献

iliad： 第 4、5 章

lavos： 第 6 章、附录 A

warz： 第 7 章

chief： 附录 B、C

校对、格式编排：Solmyr

目　　录

（注：上述页码是以原书为基准，与本电子版本没有什么关系）

献给我的妻子 Beth，

以及我的双亲 Joseph 和 Julia Maguire

－－－－为了他们的爱和支持

序

1986 年，在为几家小公司咨询和工作了 10 年之后为了获得编写 Macintosh 应用程序的经验，我特意到 Microsoft 公司工作，参加了 Macintosh 开发小组。这个小组负责 Microsoft 的图形电子表格应用程序的开发。

当时，我还不能肯定想象的代码是什么样子的，我想也许应该既引入入胜又雅致吧！但我看到的代码却很平常，与我以往见到的其它代码没有什么不同。要知道，Excel 有一个相当漂亮的用户界面 ——— 它比当时其它基于字符的电子表格软件更容易使用，更加直观。但使我感受更深的是产品中包含的一个多功能调试系统。

该系统旨在自动地问程序员和测试者进行错误报警。其工作方式非常象波音 747 驾驶仓内向驾驶员报告故障的报警灯。该调试系统主要用来对代码进行监视，它并不过多地对代码进行测试。虽然现在该调试系统采用的概念已不再新鲜了，但当时它们的广泛使用程度以及该系统有效的查错能力还是吸引了我，使我深受启发。没过多久，我就发现 Microsoft 的大部分项目都有多功能的内部调试系统，而 Microsoft 的程序员都高度重视代码中的错误及其产生原因。

在做了两年 Macintosh Excel 之后，我离开了该开发小组，去帮助另一个代码错误数目超常的小组。在开发 Excel 的两年中，我发现 Microsoft 虽然壮大了两倍，但许多老项目组熟知的概念并没有随着公司的壮大而传到新项目组。新程序员不象我加入 Microsoft 之前时的老程序员一样对容易引起错误的编码习惯特别重视，而只有一般的注意。

在我转到新项目组六个月之后，有一次我对一个程序员伙伴提到：“应该把编写无错代码的某些概念写成文字，使那些原理能在新项目组传开”。这时另一位程序员对我说：“你不要总是想着写文档，为什么你不把这一切都写下来？为什么你不写本书，问问 Microsoft 出版社是否愿意出版呢？毕竟这些信息不是谁的专利，其作用不过是为了使程序员更加重视错误。”

当时我对这个建议并没有多想，主要原因是没有时间，而且以前我也没有写过书。以前我所做过与写书最有关系的事情，不过是在 80 年代初协助别人主办 Hi-Res 杂志的程序设计专栏，这与写书毕竟不是一回事。

正如您所见到的，这本书还是写出来了。理由很简单：1990 年，由于错误越来越多，Microsoft 取消了一个尚未公布的产品。现在，错误越来越多已经不是什么新鲜事情，Microsoft 的几个竞争者都因此取消过一些项目。但 Microsoft 因为这种原因取消项目，还

是头一次。最近，随着接连不断地出现产品错误。管理人员终于开始叫嚷“受不了啦”，并采取了一系列的措施企图将错误率下降到原先的水平。尽管如此，仍然没有人将这些错误因由记录下来。

现在，Microsoft 已经比我刚进公司时大了九倍。很难设想，倘若没有准确的指南，公司怎样才能将出错率降低到原来的水平。尤其是在 Windows 和 Macintosh 的应用越来越复杂的情况下，更是如此。

以上就是我最终决定写这本书的原因

Microsoft 出版社已经同意出版这本书。

情况就是这样。

我希望您会喜欢这本书，我力图使本书不那么枯燥并尽量有趣。

Steve Maguire
西雅图，华盛顿
1992.10.22

致谢

我要感谢 Microsoft 出版社中帮助本书问世的所有人，尤其是在我写作过程中始终手把手地教我的两个人。首先我要感谢我的约稿编辑 Mike Halvorson，他使我能够按照自己的进度完成了这本书，并且耐心地解答了我这个新作者提出的许多问题。我还要特别感谢我的责任编辑 Erin O’Connor 女士，她用了许多额外时间及早地对我写出的章节提出了反馈意见。没有他们的帮助，就不会有这本书。Erin 还鼓励我以幽默的风格写这本书。她对文中的小俏皮话发笑当然不会使我不快。

我还要感谢我的父亲 Joseph Maguire 是他在 70 年代中期把我引入早期的微机世界：Altair、IMSAI 和 Sol-20，使我与这一行结缘。我要感谢我于 1981～83 年在 Valpar International 工作时的伙伴 Evan Rosen，他是对我职业生涯最有影响的几个人之一，他的知识以及洞察力在本书中都有所体现。还有 Paul Davis，在过去的 10 年里，我与他在全国各地的许多项目中都有过愉快的合作，他也无疑的塑造了我的思维方式。

最后感谢花时间阅读本书草稿，并提供技术反馈意见的所有人。他们是：Mark Gerber、Melissa Glerum、Chris Mason、Dave Moore、John Rae-Grant 和 Alex Tilles。特别感谢 Eric Schlegel 和 Paul Davis，他们不仅是本书草稿的评审者，而且在本书内容的构思上对我很有帮助。

命名约定

本书采用的命名约定和 Microsoft 所使用的“匈牙利式”命名约定差不多。该约定是由生于匈牙利布达佩斯的 Charles Simonyi 开发的，它通过在数据和函数名中加入额外的信息以增进程序员对程序的理解。例如：

```
char ch;          /* 所有的字符变量均以 ch 开始  */
byte b;           /* 所有的字节均冠以 b      */
long l;           /* 所有的长字均冠以 l      */
```

对于指向某个数据类型的指针，可以先象上面那样建立一个有类型的名字，然后给该名字加上前缀字母 P：

```
char* pch;        /* 指向 ch 的指针以 p 开始 */
byte* pb;         /* 同理 */
long* pl;
void* pv;         /* 特意显用的空指针 */
char** ppch;      /* 指向字符指针的指针 */
byte** ppb;       /* 指向字节指针的指针 */
```

匈牙利式名字通常不那么好念，但在代码中读到它们时，确实可以从中得到许多的信息。例如，当你眼看到某个函数里有一个名为 pch 的变量时，不用查看声明就立即知道它是一个指向字符的指针。

为了使匈牙利式名字的描述性更强. 或者要区分两个变量名，可以在相应类型派生出的基本名字之后加上一个以大写字母开头的“标签”。例如，strcpy 函数有两个字符指针参数：一个是源指针，另一个是目的指针。使用匈牙利式命名约定，其相应的原型是：

```
char* strcpy(char* pchTo, char* pchFrom);        /* 原型 */
```

在上面的例子中，两个字符指针有一个共同的特点 —— 都指向以 0 为结尾的 C 的字符串。因此在本书中，每当用字符指针指向字符串时，我们就用一个更有意义的名子 str 来表示。因此，上述 strcpy 的原型则为：

```
char* strcpy(char* strTo, char* strFrom)         /* 原型 */
```

本书用到另一个类型是 ANSI 标准中的类型 size_t。下面给出该类型的一些典型用法：

```
size_t sizeNew, sizeOld;                         /* 原型 */
void* malloc(size_t size);                           /* 原型 */
void* realloc(void* pv, size_t sizeNew);         /* 原型 */
```

函数和数组的命名遵循同样的约定，名字由相应的返回类型名开始，后跟一个描述的标签。例如：

```
ch = chLastKeyPressed;                    /* 由变量得一字符 */
ch = chInputBuffer[];                     /* 由数组得一字符 */
ch = chReadKeyboard;                  /* 由函数得一字符 */
```

如果利用匈牙利式命名方法，mall~和 reali~可以写成如下形式：

```
void* pvNewBlock(size_t size);                    /* 原型 */
void* pvResizeBlock(void* pv, size_t sizeNew);    /* 原型 */
```

由于匈牙利式命名方法旨在增进程序员对程序的理解，所以大多数匈牙利式名字的长度都要超过 ANSI 严格规定 6 个字母的限制。这就不妙，除非所用的系统是几十年前设计的系统，否则这 6 个字母的限制只当是历史的遗迹。

以上内容基本上没有涉及到匈牙利式命名约定的细节，所介绍的都是读者理解本书中所用变量和函数名称意义的必需内容。如果读者对匈牙利式命名约定的详细内容感兴趣，可以参考本书末尾参考文献部分列出的 Simonyi 的博士论文。

某些背景

本书用到了一些读者可能不太熟悉的软件和硬件系统的名称。下面对其中最常见的几个系统给出简单的描述

Macintosh Macintosh 是 Apple 公司的图形窗口计算机，公布于 1984 年。它是最先支持“所见即所得”拥户界面的流行最广的计算机。

Windows Windows 是 Microsoft 公司的图形窗口操作系统。Microsoft 公司 1990 年公布了 Windows 3.0，该版本明显好于其早期版本。

Excel Excel 是 Microsoft 公司的图形电子表格软件，1985 年首次在 Macintosh 上公布，随后在进行了大量的重写和排错工作后，被移植到 Windows 上。多年来 Macintosh Excel 和 Windows Excel 共用一个名字，但程序所用的代码并不相同。

在本书中，找多次提到曾经当过 Macintosh Excel 程序员这一经历。但应该说明的是，我的大部分工作是将 Windows Excel 的代码移到 Macintosh Excel 上或者是实现与 Windows Excel 相似的特征。我与该产品现在的惊人成功并没有特别的关系。

我为 Macintosh Excel 所做的唯一真正重要的贡献是说服 Microsft 放弃掉 Macintosh Excel，而直接利用 Windows Excel 的源代码构造 Macintosh 的产品。Macintosh 2.2 版是第一个基于 Windows Excel 的版本，它享用了 Windows Excel 80%的源代码。这对 Macintosh Excel 的用户意义重大，因为用了 2.2 版以后他们会感至 Macintosh Excel 的功能和质量都有了一个很大的飞跃。

Word Word 是 Microsoft 公司的字处理应用软件。实际上，Word 有三种版本：基于字符并在 MS－DOS 上运行的 DOS 版；Macintosh 版；Windows 版。到目前为止，这三种版本的产品虽然是用不同的源代码做出的，但它们非常相象，用户改用任何一个都不会有什么困难。由于 Excel 利用共享代码获得了成功，Microsoft 已经决定 Word 的高版本也将

	采用共享代码进行开发。
80x86	80x86 是 MS-DOS 和 Windows 机器常用的 Intel CPU 系列。
680x0	680x0 是各种 Macintosh 所用的 Motorola CPU 系列。

引言

几年前在一次偶然翻阅 Donald Knuth 所著《TEX：The Program》一书时，序言中的一段话深深触动了我：

我确信 TEX 的最后一个错误已经在 1985 年 11 月 27 日被发现并排除掉了。但是如果出于目前尚不知道的原因，TEX 仍然潜伏有错误，我非常愿意付给第一个发现者$20.48 元。（这一金额已是以前的两倍。我打算在本年内再增加一倍。你看我是多么自信！）

我对 Knuth 是否曾经付给某人$20.48 甚至$40.96 元不感兴趣，这并不重要。重要的是他对他的程序所具有的那种自信。那么据你所知，究竟有多少程序员会严肃地声称他们的程序完全没有错误？又有多少敢把这一声称印刷在书上，并准备为错误的发现者付钱呢？

如果程序员确信测试组已经发现了所有的错误，那么他也许敢作这种声明。但这本身就是一个问题。每当代码被打包装送给程序经销商之前，人们在胸前划着十字带着最好的愿望说：“希望测试已经发现了所有的错误”。这一情景我已见过多次了。

由于现代的程序员已经放弃了对代码进行彻底测试的职责，他们没法知道代码中是否有错。管理人员也不会公布测试情况，只是说：“别操那个心，测试人员会为你作好测试的”。更为微妙的是，管理人员希望程序员自己进行代码的测试。同时，他们希望测试员作得更彻底些，因为毕竟这是他们的本职工作。

正如你在本书中将会看到的那样，编写无错代码的技术或许有几百种，程序员能用，但测试人员却无法使用，因为这些技术和代码的编写直接相关。

两个关键的问题

本书介绍的所有决窍是当发现错误时，不断地就以下两个问题追问自己的结果：

- 怎样才能自动地查出这个错误？
- 怎样才能避免这个错误？

第一个问题可能使读者认为本书是有关测试的书，其实不是。当编辑程序发现语法错误时，它是在做测试吗？不，不是。编辑程序只是在自动地检查代码中的错误。语法错误只是程序员可以使用的自动查错方法查出的一种最基本的错误类型。本书将详尽介绍自动向程序员提示错误的方法。

编写无错代码的最好方法是把防上错误放在第一位。关于这个问题，同样也有许多的技巧。某些技巧与常用的编码惯例有关，但它们不是象“每个人都违背原则”或“没有人违背该原则”这样泛泛地考虑问题，而是对相应的细节进行详细的讨论。要记住，在任何时候跟在大多数人的后面常常是所能选择的最坏一条路。因此在成为别人的追随者之前一定要确定这样做确实有意义，而且不要仅仅因为其它的什么人如此自己也如此。

本书的最后一章讨论编写无错代码应持正确态度的重要性。如果没有正确的态度，发现错误和防止错误就好比在冬季大开着窗户给房间加热，虽然也能达到目的，但要浪费大量的能量。

本书除第 4 章和第 8 章以外都配有练习。但要注意，这些练习并不是要测验读者对相应内容的理解。实际上更多的是作者想在该章的正文中阐述却没能放进去的要点。其它的练习为的是让读者思考与该章内容有关的一些问题，打开思路，琢磨一下以前未曾考虑过的概念。

无论哪种情况，都是想通过练习再补充一些新的技巧和信息，因此值得一读。为使读者了解作者的意图，本书在附录 C 中提供了所有练习的答案。大部分章节还给出了一些课题，但这些课题没有答案，因为这种课题通常是任务，而不是问题。

规则或者建议

本书的编排类似于 Brian Kernighan 和 P. J. Plauger 所写的程序设计经典著作《The Elements of Programming Sytle》。该书出于 William Strunk Jr.和 E. B. White 所写的重要经典著作《The Elements of Style》。这两本书采用同样的基本概念表达方法：

- 给出一个例子；
- 指出该例子中的某些问题所在；
- 用一般的准则改进该例子。

确实，这是个程式，而且是使读者感到舒服的程式，因此本书同样采用了这一程式。作者力图使本书读起来是一种享受，尽管它有着公式的性质。希望读者会觉得本书很有趣。

本书还给出一些似乎不应违背的“一般准则”。我们的第一条准则是：

每条准则都有例外

由于准则是用来说明一般情况的，所以本书一般并不指明准则的例外情况，而把它留给读者。我相信，当读者读到某个准则时，肯定会怀疑道：“噢，当……时，不是这样的……”。如果某个人对你说：“不能闯红灯”，虽然这是一条准则，但你肯定能够举出一种特殊情况，在这种情况下闯红灯倒是个正确的行动。这里关键是要记住准则只是在一般情况下才有意义，因此只有理由十分充足时，才可以违背准则。

关于本书代码的说明

本书的所有代码都是按 ANSI C 写的，并且通过了 MS-DOS、Microsoft Windows 和 Apple Macintosh 上五个流行编译程序的测试：

Microsoft C/C++ 7.0	Microsoft 公司
Turbo C/C++ 3.0	Borland 国际公司
Aztec 5.2	Manx 软件系统公司
MPW C 3.2	Apple 计算机公司
THINK C 5.0	Symantec 公司

还有一个问题：如果读者想从本书中摘取代码用在自己的程序中，那要小心。因为为了说明书中的论点，许多例子都有错误。另外，书中用到的函数虽然名字和功能都与 ANSI C 的标准库函数相同，但已对相应的界面进行了一些小的修改。例如 ANSI 版 memchr 函数的界面是：

```
void* memchr(const void* s, int c, size_t n);
```

这里 memchr 的内部将整数 c 当作 unsigned char 来处理。在本书的许多地方，读者都会看到字符类型被显式地声明为 unsigned char，而不是 int：

```
void* memchr(const void* pv, unsigned char ch, size_t size);
```

ANSI 标准将所有的字符变元都声明为 int，是为了保证其库函数同样可以用于 ANSI 标准之前编写的非原型程序，这时程序使用 extern 声明函数。由于在本书中只使用 ANSI C，所以不必考虑这些向下兼容的细节而可以用更加精确的类型声明以改进程序的清晰程度并用原型进行强类型检查（详见第 1 章）。

“提到 Macintosh 了吗？”

出于某种原因，一本书如果没有提到 PDP11，Honeywell 6000，当然还有 IBM 360，就不会被认真对待。因此，我在这里也提到了它们。仅此而已，读者在本书中再也不会见到这些字眼、读者见到最多的是 MS-DOS，Microsoft Windows，特别还有 Apple Macintosh，因为近年来我一直为这些系统编写代码。但是应该注意，本书中的任何代码都不受这些特定的系统约束。它们都是用通用的 C 编写的，应该能够工作于任何的 ANSI C 编译程序下。因此即使读者使用的系统本书没有提及，也不必担心这些操作系统的细节会产生障碍。

应该提到的是在大多数的微机系统中，用户都可以通过 NULL 指针进行读写，破坏栈框架并在内存甚至是其它应用程序的内存区中留下许多的无用信息，而硬件并没有什么反应，听任用户为所欲为。之所以提到这一点，是因为如果读者习惯于认为通过 NULL 指针进行写操作会引起硬件故障的话，那么可能会对本书中的某些语句感到迷惑不解。遗憾的是，目前微机上的保护型操作系统仍不普及，破坏内存的隐患必须通过硬件保护（通常它也不能提供充足的保护）之外的方法才能发现。

有错误就有错误

不必为本书的读者定义什么叫错误，相信读者都知道什么是错误。但是错误可以分为两类：一类是开发某一功能时产生的错误，另一类是在程序员认为该功能已经开发完成之后仍然遗留在代码中的错误。

例如在 Microsoft 中，每个产品都由一些绝不应该含有错误的原版源代码构成。当程序员给产品增加新功能时，并不直接改变相应的原版源代码，改变的是其副本。只有在这些改变已经完成，并且程序员确信相应代码中已经没有错误时，才将其合并到原版源代码中。因此从产品质量看，在实现指定功能过程中不论产生多少个错误都没有关系，只要这些错误在相应代码被并入原版源代码之前被删除掉就行。

所有的错误都有害，但损害产品最危险的错误是已经进入原版源代码中的错误。因此，本书中提到的错误指的就是这些已经进入原版源代码中的错误。作者并不指望程序员在键入计算机之前总是写出没有错误的代码，但确信防止错误侵入原版源代码是完全可能的。尤其是程序员在使用了本书提供的秘诀之后，更是如此。

第 1 章　假想的编译程序

读者可以考虑一下倘若编译程序能够正确地指出代码中的所有问题，那相应程序的错误情况会怎样？这不单指语法错误，还包括程序中的任何问题，不管它有多么隐蔽。例如，假定程序中有“差 1”错误，编译程序可以采用某种方法将其查出，并给出如下的错误信息

```
-> line 23: while (i<=j)
    off by one error: this should be '<'
```

又如，编译程序可以发现算法中有下面的错误：

```
-> line 42: int itoa(int i, char* str)
    algorithm error: itoa fails when i is -32768
```

再如，当出现了参数传递错误时，编译程序可以给出如下的错误信息：

```
-> line 318: strCopy = memcpy(malloc(length), str, length);
    Invalid argument: memcpy fails when malloc returns NULL
```

好了，要求编译程序能够做到这一程度似乎有点过分。但如编译程序真能做到这些，可以想象编写无错程序会变得多么容易。那简直是小事一桩，和当前程序员的一般作法真没法比。

假如在间谍卫星上用摄像机对准某个典型的软件车间.就会看到程序员们正弓着身子趴在键盘上跟踪错误；旁边，测试者正在对刚作出的内部版本发起攻击，轮番轰炸式地输入人量的数据以求找出新的错误。你还会发现，测试员正在检查老版本的错误是否溜进了新版本。可以推想，这种查错方法比用上面的假想编译程序进行查错要花费大得多的工作量、确实如此，而且它还要有点运气。

运气？

是的，运气。测试者之所以能够发现错误，不正是因为他注意到了诸如某个数不对、某个功能没按所期望的方式工作或者程序瘫痪这些现象吗？再看看上面的假想编译程序给出的上述错误：程序虽然有了“差 1”错误，但如果它仍能工作，那么测试者能看得出来吗？就算看得出来，那么另外两个错误呢？

这听起来好象很可怕但测试人员就是这样做的大量给程序输入数据，希望潜在的错误能够亮相。“噢，不！我们测试人员的工作可不这么简单，我们还要使用代码覆盖工具、自动的测试集、随机的“猴”程序、抽点打印或其他什么的”。也许是这样，但还是让我们来看看这些工具究竟做了些什么吧！覆盖分析工具能够指明程序中哪些部分未被测试到，测试人员可以使用这一信息派生出新的测试用例。至于其它的工具无非都是“输入数据、观察结果”这一策略的自动化。

请不要产生误解，我并不是说测试人员的所作所为都是错误的。我只是说利用黑箱方法所能做的只是往程序里填数据，并看它弹出什么。这就好比确定一个人是不是疯子一样。问一些问题，得到回答后进行判断。但这样还是不能确定此人是不是疯子。因为我们没法知道其头脑中在想些什么。你总会这样地问自己：“我问的问题够吗？我问的问题对吗……”。

因此，不要光依赖黑箱测试方法。还应该试着去模仿前面所讲的假想编译程序，来排除运气对程序测试的影响，自动地抓住错误的每个机会。

考虑一下所用的语言

你最后一次看推销字处理程序的广告是什么时候？如果那个广告是麦迪逊大街那伙人写的，它很可能是这么说：“无论是给孩子们的老师写便条还是为下期的《Great American Novel》撰稿，WordSmasher 都能行，毫不费劲！WordSmasher 配备了令人吃惊的 233000 字的拼写字典，足足比同类产品多 51000 个字。它可以方便地找出样稿中的打字错误。赶快到经销商那里去买一份拷贝。WordSmasher 是从圆珠笔问世以来最革命性的书写工具!”。

用户经过不断地市场宣传熏陶，差不多都相信拼写字典越大越好，但事实并非如此。象 em、abel 和 si 这些词，在任何一本简装字典中都可以查到、但在 me、able 和 is 如此常见的情况下您还想让拼写检查程序认为 em、abel 和 si 也是拼写正确的词吗？如果是，那么当你看到我写的 suing 时，其本意很可能是与之风马牛不相及的 using。问题不在于 suing 是不是一个真正的词而在于它在此处确实是个错误。

幸运的是，某些质量比较高的拼写检查程序允许用户删去象 em 这类容易引起麻烦的词。这样一来，拼写检查程序就可以把原来合法的单词看成是拼写错误。好的编译程序也应该能够这样 ——— 可以把屡次出错的合法的 C 习惯用法看成程序中的错误。例如，这类编译程序能够检查出以下 while 循环错放了一个分号：

```
/* memcpy 复制一个不重叠的内存块 */
void* memcpy(void* pvTo, void* pvFrom, size_t size)
{
byte* pbTo = (byte*)pvTo;
byte* pbFrom = (byte*)pvFrom;
while(size-->0);
    *pbTo++ = *pbFrom++;
return(pvTo);
}
```

我们从程序的缩进情况就可以知道 while 表达式后由的分号肯定是个错误，但编译程序却认为这是一个完全合法的 while 语句，其循环体为空语句。由于有时需要空语句，有时不需要空语句，所以为了查出不需要的空语句，编译程序常常在遇到空语句时给出一条可选的警告信息，自动警告你可能出了上面的错误。当确定需要用空语句时，你就用。但最好用 NULL 使其明显可见。例如：

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

```
    NULL;
Return(pchStart);
}
```

由于 NULL 是个合法的 C 表达式，所以这个程序没有问题。使用 NULL 的更大好处在于编译程序不会为 NULL 语句生成任何的代码，因为 NULL 只是个常量。这样，编译程序接受显式的 NULL 语句，但把隐式空语句自动地当作错误标出。在程序中只允许使用一种形式的空语句，如同为了保持文字的一致性，文中只想使用 zero 的一种复数形式 zeroes，因此要从拼写字典中删除另一种复数形式 zeros。

另一个常见的问题是无意的赋值。C 是一个非常灵活的语言，它允许在任何可以使用表达式的地方使用赋值语句。因此如果用户不够谨慎，这种多余的灵活性就会使你犯错误。例如，以下程序就出现了这种常见的错误：

```
if(ch = '\t')
ExpandTab();
```

虽然很清楚该程序是要将 ch 与水平制表符作比较，但实际上却成了对 ch 的赋值。对于这种程序，编译程序当然不会产生错误，因为代码是合法的 C。

某些编译程序允许用户在 && 和 || 表达式以及 if、for 和 while 构造的控制表达式中禁止使用简单赋值，这样就可以帮助用户查出这种错误。这种做法的基本依据是用户极有可能在以上五种情况下将等号＝＝偶然地健入为赋值号＝。

这种选择项并不妨碍用户作赋值，但是为了避免产生警告信息，用户必须再拿别的值，如零或空字符与赋值结果做显式的比较。因此，对于前面的 strcpy 例子，若循环写成：

```
while(*pchTo++ = *pchFrom++)
NULL;
```

编译程序会产生警告信息一所以要写成；

```
while( (*pchTo++ = *pchFrom++)!=' \0' )
NULL;
```

这样做有两个好处。第一，现代的商用级编译程序不会为这种冗余的比较产生额外的代码，可以将其优化掉。因此，提供这种警告选择项的编译程序是可以信赖的。第二，它可以少冒风险，尽管两种都合法，但这是更安全的用法。

另一类错误可以被归入“参数错误”之列。例如，多年以前，当我正在学 C 语言时，曾经这样调用过 fputc：

```
fprintf(stderr, "Unable to open file %s. \n", filename);
……
fputc(stderr, '\n');
```

这一程序看起来好象没有问题，但 fputc 的参数次序错了。不知道为什么，我一直认为流指针（stderr）总是这类流函数的第一个参数。事实并非如此，所以我常常给这些函数传递过去许多没用的信息。幸好 ANSI C 提供了函数原型，能在编译时自动地查出这些错误。由于 ANSI C 标准要求每个库函数都必须有原型所以在 stdio.h 头文件中能够找到 fputc 的

原型。fputc 的原型是：

```
int fputc(int c, FILE* stream);
```

如果在程序中 include 了 stdio.h，那么在调用 fputc 时，编译程序会根据其原型对所传递的每个参数进行比较。如果二者类型不同，就会产生编译错误。在上面的错误例于中，因为在 int 的位置上传递了 FILE* 类型的参数，所以利用原型可以自动地发现前一个 fputc 的错误。

ANSI C 虽然要求标准的库函数必须有原型，但并不要求用户编写的函数也必须有原型。严格地说，它们可以有原型，也可以没有原型。如果用户想要检查出自己程序中的调用错误，必须自己建立原型，并随时使其与相应的函数保持一致。

最近我听到程序员在抱怨他们必须对函数的原型进行维护。尤其是刚从传统 C 项目转到 ANSI C 项目时，这种抱怨更多。这种抱怨是有一定理由的，但如果不用原型，就不得不依赖传统的测试方法来查出程序中的调用错误。你可以扪心自问，究竟哪个更重要，是减少一些维护工作量，还是在编译时能够查出错误？如果你还不满意，请再考虑一下利用原型可以生成质量更好的代码这一事实。这是因为：ANSI C 标准使得编译程序可以根据原型信息进行相应的优化。

在传统 C 中，对于不在当前正被编译的文件中的函数，编译程序基本上得不到关于它的信息。尽管如此，编译程序仍然必须生成对这些函数的调用，而且所生成的调用必须奏效。编译程序实现者解决这个问题的办法是使用标准的调用约定。这一方法虽然奏效，但常常意味着编译程序必须生成额外的代码，以满足调用约定的要求。但如果使用了“要求所有函数都必须有原型”这一编译程序提供的警告选择项，由于编译程序了解程序中每个函数的参数情况，所以可以为不同的函数选择它认为最有效率的调用约定。

空语句、错误的赋值以及原型检查只是许多 C 编译程序提供的选择项中的一小部分内容，实际上常常还有更多的其它选择项。这里的要点是：用户可以选择的编译程序警告设施可以就可能的错误向用户发出警告信息，其工作的方式非常类似于拼写检查程序对可能的拼写错误的处理

Peter Lynch，据说是 80 年代最好的合股投资公司管理者，他曾经说过：投资者与赌徒之间的区别在于投资者利用每一次机会，无论它是多么小，去争取利益；而赌徒则只靠运气。用户应该将这一概念同样应用于编程活动，选择编译程序的所有可选警告设施，并把这些措施看成是一种无风险高偿还的程序投资。再不要问：“应该使用这一警告设施吗？而应该问：“为什么不使用这一警告设施呢？”。要把所有的警告开关都打开，除非有极好的理由才不这样做。

使用编译程序所有的可选警告设施

增强原型的能力

不幸的是，如果函数有两个参数的类型相同，那么即使在调用该函数时互换了这两个参

数的位置，原型也查不出这一调用错误。例如，如果函数 memchr 的原型是：

```
void* memchr(const void* pv, int ch, int size);
```

那么在调用该函数时，即使互换其字符 ch 和大小 size 参数，编译程序也不会发出警告信息。

但是如果在相应界面和原型中使用了更加精确的类型，就可以增强原型提供的错误检查能力。例如，如果有了下面的原型：

```
void* memchr(const void* pv, unsigned char ch, size_t size);
```

那么在调用该函数时弄反了其字符 ch 和大小 size 参数，编译程序就会给出警告错误。

在原型中使用更精确类型的缺陷是常常必须进行参数的显式类型转换，以消除类型不匹配的错误，即使参数的次序正确。

lint 并不那么差

另一种检查错误更详细、更彻底的方法是使用 lint，这种方法几乎不费什么事。最初，lint 这个工具用来扫描 C 源文件并对源程序中不可移植的代码提出警告。但是现在大多数 lint 实用程序已经变得更加严密，它不但可以检查出可移植性问题，而且可以检查出那些虽然可移植并且完全合乎语法但却很可能是错误的特性，上一节那些可疑的错误就属于这一类。

不幸的是，许多程序员至今仍然把 lint 看作是一个可移植性的检查程序，认为它只能给出一大堆无关的警告信息。总之，lint 得到了不值得麻烦的名声。如果你也是这样想的程序员，那么你也许应该重新考虑你的见解。想一想究竟是哪一种工具更加接近于前文所述的假想编译程序，是你正使用的编译程序，还是 lint？

实际上，一旦源程序变成了没有 lint 错误的形式，继续使其保持这种状态是很容易做到的。只要对所改变的部分运行 lint，没有错误之后再把其并入到原版源代码中即可。利用这种方法，并不要进行太多的考虑，只要经过一、二周就可以写出没有 lint 错误的代码。在达到这个程度时，就可以得到 lint 带来的各种好处了。

使用 lint 来查出编译程序漏掉的错误

但我做的修改很平常

一次在同本书的一个技术评审者共进午餐时，他问我本书是否打算包括一节单元测试方面的内容。我回答说：“不”。因为尽管单元测试也与无错代码的编写有关，但它实际上属于

另一个不同的类别，即如何为程序编写测试程序。

他说：“不，你误解了。我的意思是你是否打算指出在将新做的修改并入原版源代码之前，程序员应该实际地进行相应的单元测试。我的小组中的一位程序员就是因为在进行了程序的修改之后没有进行相应的单元测试，使一个错误进入到我们的原版源代码中。”

这使我感到很惊奇。因为在 Microsoft，大多数项目负责人都要求程序员在合并修改了的源代码之前，要进行相应的单元测试。

“你没问他为什么不做单元测试吗？”，我问道。

我的朋友从餐桌上抬起头来对我说：“他说他并没有编写任何新的代码，只是对现有代码进行了某些移动。他说他认为没必要再进行单元测试”。

这种事情在我的小组中也曾经发生过。

它使我想起曾经有一个程序员在进行了修改之后，甚至没有再编译一次就把相应的代码并入了原版源代码中。当然，我发现了这一问题，因为我在对原版源代码进行编译时产生了错误。当我问这个程序员怎么会漏掉这个编译错误，他说：“我做的修改很平常，我认为不会出错”，但他错了。

这些错误本来都应该不会进入原版源代码中，因为二者都可以几乎毫不费力地被查出来。为什么程序员会犯这种错误呢？是他们过高地估计了自己编写正确代码的能力。

有时，似乎可以跳过一些设计用来避免程序出错的步骤，但走捷径之时，就是麻烦将至之日。我怀疑会有许多的程序员甚至没有对相应的代码进行编译，就“完成”了某一功能。我知道这只是偶然情况，但绕过单元测试的趋势正在变强，尤其是作简单的改动。

如果你发现自己正打算绕过某个步骤。而它恰恰可以很容易地用来查错，那么一定要阻止自己绕过。相反，要利用所能得到的每个工具进行查错。此外，单元测试虽然意味着查错，但如果你根本就不进行单元测试也是枉然。

如果有单元测试，就进行单元测试

小结

你认识哪个程序员宁愿花费时间去跟踪排错，而不是编写新的代码？我肯定有这种程序员，但至今我还没有见过一个。对于我认识的程序员，如果你答应他们再不用跟踪下一个错误，他们会宁愿一辈子放弃享用中国菜。

当你写程序时，要在心中时刻牢记着假想编译程序这一概念，这样就可以毫不费力或者只费很少的力气利用每个机会抓住错误。要考虑编译程序产生的错误、lint 产生的错误以及单元测试失败的原因。虽然使用这些工具要涉及到很多的特殊技术，但如果不花这么多的功夫，那产品中会有多少个错误？

如果想要快速容易地发现错误，就要利用工具的相应特性对错误进行定位。错误定位得越早，就能够越早地投身于更有兴趣的工作。

要点：

- 消除程序错误的最好方法是尽可能早、尽可能容易地发现错误，要寻求费力最小的自动查错方法。
- 努力减少程序员查错所需的技巧。可以选择的编译程序或 lint 警告设施并不要求程序员要有什么查错的技巧。在另一个极端，高级的编码方法虽然可以查出或减少错误，但它们也要求程序员要有较多的技巧，因为程序员必须学习这些高级的编码方法。

练习：

1) 假如使用了禁止在 while 的条件部分进行赋值的编译程序选择项，为什么可以查出下面代码中的运算优先级错误？

```
While(ch = getchar() != EOF)
……
```

2) 看看你怎样使用编译程序查出无意使用的空语句和赋值语句。值得推荐的办法是进行相应的选择，使编译程序能够对下列常见问题产生警告信息。怎样才能消除这些警告信息呢？
   a) if(flight == 063)。这里程序员的本意是对 63 号航班进行测试，但因为前面多了一个 0 使 063 成了八进制数。结果变成对 51 号航班进行测试。
   b) If(pb != NULL & pb != 0xff)。这里不小心把&&键入为&，结果即使 pb 等于 NULL 还会执行*pb != 0xff。
   c) quot = numer/*pdenom。这里无意间多了个*号结果使/*被解释为注释的开始。
   d) word = bHigh<<8 + bLow。由于出现了运算优先级错误，该语句被解释成了：word = bHigh << (8+bLow)

3) 编译程序怎样才能对“没有与之配对的 else”这一错误给出警告？用户怎样消除这一警告？

4) 再看一次下面的代码：

```
if(ch == ‘\t’)
    ExpandTab();
```

除禁止在 if 语句中使用简单赋值的方法之外，能够查出这个错误的另一种众所周知的方法是把赋值号两边的操作数颠倒过来：

```
if(‘\t’ == ch)
    ExpandTab();
```

这样如果应该键入＝＝时健入了＝，编译程序就会报错，因为不允许对常量进行赋

值。这个办法彻底吗？为什么它不象编译程序开关自动化程度那么高？为什么新程序员会用赋值号代替等号？

5) C的预处理程序也可能引起某些意想不到的结果。例如，宏UINT_MAX定义在limit.h中，但假如在程序中忘了 include 这个头文件，下面的伪指令就会无声无息地失败，因为预处理程序会把预定义的UINT_MAX 替换成0：

```
……
#if UINT_MAX > 65535u
……
#endif
```

怎样使预处理程序报告出这一错误？

课题：

为了减轻维护原型的工作量，某些编译程序会在编译时自动地为所编译的程序生成原型。如果你用的编译程序没有提供这一选择项，自己写一个使用程序来完成这一工作。为什么标准的编码约定可以使这个使用程序的编写变得相对容易？

课题：

如果你用的编译程序还不支持本章（包括练习）中提及的警告设施，那么促进相应的制造商支持这些设施，另外要敦促他们除了允许用户设定或者取消对某些类错误的检查之外，还要提供有选择地设定或取消一些特定的警告设施，为什么要这样做呢？

第 2 章　自己设计并使用断言

利用编译程序自动查错固然好，但我敢说只要你观察一下项目中那些比较明显的错误，就会发现编译程序所查出的只是其中的一小部分。我还敢说，如果排除掉了程序中的所有错误那么在大部分时间内程序都会正确工作。

还记得第 1 章中的下面代码吗？

```
strCopy = memcpy(malloc(length), str, length);
```

该语句在多数情况下都会工作得很好，除非 malloc 的调用产生失败。当 malloc 失败时，就会给 memcpy 返回一个 NULL 指针。由于 memcpy 处理不了 NULL 指针，所以出现了错误。如果你很走运，在交付之前这个错误导致程序的瘫痪，从而暴露出来。但是如果你不走运，没有及时地发现这个错误，那某位顾客就一定会“走运”了。

编译程序查不出这种或其他类似的错误。同样，编译程序也查不出算法的错误，无法验证程序员所作的假定。或者更一般地，编译程序也查不出所传递的参数是否有效。

寻找这种错误非常艰苦，只有技术非常高的程序员或者测试者才能将它们根除并且不会引起其他的问题。

然而假如你知道应该怎样去做的话，自动寻找这种错误就变得很容易了。

两个版本的故事

让我们直接进入 memcpy，看看怎样才能查出上面的错误。最初的解决办法是使 memcpy 对 NULL 指针进行检查，如果指针为 NULL，就给出一条错误信息，并中止 memcpy 的执行。下面是这种解法对应的程序。

```
/* memcpy ——— 拷贝不重叠的内存块 */
void memcpy(void* pvTo, void* pvFrom, size_t size)
{
void* pbTo = (byte*)pvTo;
void* pbFrom = (byte*)pvFrom;
if(pvTo == NULL || pvFrom == NULL)
{
    fprintf(stderr, “Bad args in memcpy\n”);
    abort();
}
```

```
while(size-->0)
    *pbTo++ == *pbFrom++;
return(pvTo);
}
```

只要调用时错用了 NULL 指针，这个函数就会查出来。所存在的唯一问题是其中的测试代码使整个函数的大小增加了一倍，并且降低了该函数的执行速度。如果说这是“越治病越糟”，确实有理，因为它一点不实用。要解决这个问题需要利用 C 的预处理程序。

如果保存两个版本怎么样？一个整洁快速用于程序的交付；另一个臃肿缓慢件（因为包括了额外的检查），用于调试。这样就得同时维护同一程序的两个版本，并利用 C 的预处理程序有条件地包含或不包含相应的检查部分。

```
void memcpy(void* pvTo, void* pvFrom, size_t size)
{
void* pbTo = (byte*)pvTo;
void* pbFrom = (byte*)pvFrom;
#ifdef DEBUG
if(pvTo == NULL || pvFrom == NULL)
{
    fprintf(stderr, “Bad args in memcpy\n”);
    abort();
}
#endif
while(size-->0)
    *pbTo++ == *pbFrom++;
return(pvTo);
}
```

这种想法是同时维护调试和非调试（即交付）两个版本。在程序的编写过程中，编译其调试版本，利用它提供的测试部分在增加程序功能时自动地查错。在程序编完之后，编译其交付版本，封装之后交给经销商。

当然，你不会傻到直到交付的最后一刻才想到要运行打算交付的程序，但在整个的开发工程中，都应该使用程序的调试版本。正如在这一章和下一章所建，这样要求的主要原因是它可以显著地减少程序的开发时间。读者可以设想一下：如果程序中的每个函数都进行一些最低限度的错误检查，并对一些绝不应该出现的条件进行测试的活，相应的应用程序会有多么健壮。

这种方法的关键是要保证调试代码不在最终产品中出现。

既要维护程序的交付版本，又要维护程序的调试版本

利用断言进行补救

说老实话 memcpy 中的调试码编得非常蹩脚，且颇有点喧宾夺主的意味。因此尽管它能产生很好的结果，多数程序员也不会容忍它的存在，这就是聪明的程序员决定将所有的调试代码隐藏在断言 assert 中的缘故。assert 是个宏，它定义在头文件 assert.h 中。assert 虽然不过是对前面所见＃ifdef 部分代码的替换，但利用这个宏，原来的代码从 7 行变成了 1 行。

```
void memcpy(void* pvTo, void* pvFrom, size_t size)
{
void* pbTo = (byte*)pvTo;
void* pbFrom = (byte*)pvFrom;
assert(pvTo != NULL && pvFrom != NULL);
while(size-->0)
    *pbTo++ == *pbFrom++;
return(pvTo);
}
```

aasert 是个只有定义了 DEBUG 才起作用的宏，如果其参数的计算结果为假，就中止调用程序的执行。因此在上面的程序中任何一个指针为 NULL 都会引发 assert。

assert 并不是一个仓促拼凑起来的宏，为了不在程序的交付版本和调试版本之间引起重要的差别，需要对其进行仔细的定义。宏 assert 不应该弄乱内存，不应该对未初始化的数据进行初始化，即它不应该产主其他的副作用。正是因为要求程序的调试版本和交付版本行为完全相同，所以才不把 assert 作为函数，而把它作为宏。如果把 assert 作为函数的话，其调用就会引起不期望的内存或代码的兑换。要记住，使用 assert 的程序员是把它看成一个在任何系统状态下都可以安全使用的无害检测手段。

读者还要意识到，一旦程序员学会了使用断言，就常常会对宏 assert 进行重定义。例如，程序员可以把 assert 定义成当发生错误时不是中止调用程序的执行，而是在发生错误的位置转入调试程序。assert 的某些版本甚至还可以允许用户选择让程序继续运行，就仿佛从来没有发生过错误一样。

如果用户要定义自己的断言宏，为不影响标准 assert 的使用，最好使用其它的名字。本书将使用一个与标准不同的断言宏，因为它是非标准的，所以我给它起名叫做 ASSERT，以使它在程序中显得比较突出。宏 assert 和 ASSERT 之间的主要区别是 assert 是个在程序中可以随便使用的表达式，而 ASSERT 则是一个比较受限制的语句。例如使用 assert，你可以写成：

```
if(assert(p != NULL), p->foo!=bar)
……
```

但如果用 ASSERT 试试就会产生语法错误。这种区别是作者有意造成的。除非打算在表达式环境中使用断言，否则就应该将 ASSERT 定义为语句。只有这样，编译程序才能够在它

被错误地用到表达式时产生语法错误。记住，在同错误进行斗争时每一点帮助都有助于错误的发现。我们为什么要那些自己从来用不着的灵活性呢？

下面是一种用户自己定义宏 ASSERT 的方法：

```
#ifdef DEBUG
void _Assert(char* , unsigned);      /* 原型 */
#define ASSERT(f)            \
if(f)                        \
    NULL;                \
else                         \
    _Assert(__FILE__ , __LINE__)
#else
#define ASSERT(f)            NULL
#endif
```

从中我们可以看到，如果定义了 DEBUG，ASSERT 将被扩展为一个 if 语句。if 语句中的 NULL 语句让人感到很奇怪，这是因为要避免 if 不配对，所以它必须要有 else 语句。也许读者认为在_Assert 调用的闭括号之后需要一个分号，但并不需要。因为用户在使用 ASSERT 时，已经给出了一个分号.

当 ASSERT 失败时，它就使用预处理程序根据宏__FILE__和__LINE__所提供的文件名和行号参数调用_Assert。_Assert 在标准错误输出设备 stderr 上打印一条错误消息，然后中止：

```
void _Assert(char* strFile, unsigned uLine)
{
fflush(stdout);
fprintf(stderr, “\nAssertion failed: %s, line %u\n”,strFile, uLine);
fflush(stderr);
abort();
}
```

在执行 abort 之前，需要调用 fflush 将所有的缓冲输出写到标准输出设备 stdout 上。同样，如果 stdout 和 stderr 都指向同一个设备，fflush stdout 仍然要放在 fflush stderr 之前，以确保只有在所有的输出都送到 stdout 之后，fprintf 才显示相应的错误信息。

现在如果用 NULL 指针调用 memcpy，ASSERT 就会抓住这个错误，并显示出如下的错误信息：

```
Assertion failed: string.c , line 153
```

这给出了 assert 与 ASSERT 之间的另一点不同。标准宏 assert 除了给出以上信息之外，还显示出已经失败了的测试条件。例如对这个问题，我通常所用编译程序的 assert 会显示出如下信息：

```
Assertion failed: pvTo != NULL && pbFrom != NULL
```

File string.c , line 153

在错误消息中包括测试表达式的唯一麻烦是每当使用 assert 时，它都必须为_Assert 产生一条与该条件对应的正文形式打印消息。但问题是，编译程序要在哪儿存储这个字符串呢？Macintosh、DOS 和 Windows 上的编译程序通常在全局数据区存储字符串，但在 Macintosh 上，通常把最大的全局数据区限制为 32K，在 DOS 和 Windows 上限制为 64K。因此对于象 Microsoft Word 和 Excel 这样的大程序，断言字符串立刻会占掉这块内存。

关于这个问题存在一些解决的办法，但最容易的办法是在错误信息中省去测试表达式字符串。毕竟只要查看了 string.c 的第 153 行，就会知道出了什么问题以及相应的测试条件是什么。

如果读者想了解标准宏 assert 的定义方法，可以查看所用的编译系统的 assert.h 文件。ANSI C 标准在其基本原理部分也谈到了 assert 并且给出了一种可能的实现。P. J. Plauger 在其“The Standard C library”一书中也给出了一种略微不同的标准 assert 的实现。

不管断言宏最终是用什么样方法定义的，都要使用它来对传递给相应函数的参数进行确认。如果在函数的每个调用点都对其参数进行检查，错误很快就会被发现。断言宏的最好作用是使用户在错误发生时，就可以自动地把它们检查出来。

要使用断言对函数参数进行确认

“无定义”意味着“要避开”

如果读者停下来读读 ANSI C 中 memcpy 函数的定义，就会看到其最后一行说：“如果在存储空间相互重叠的对象之间进行了拷贝，其结果无定义”。在其它的书中，对此的描述有点不同。例如在 P. J. Plauger 和 Jim Brodie 的“Standard C”中相应的描述是：“可以按任何次序访问和存储这两个数组的元素”。

总之，这些书都说如果依赖于以按特定方式工作的 memcpy，那么当使用相互重叠的内存块调用该函数时，你实际上是做了一个编译程序不同（包括同一编译程序的不同版本）、结果可能也不同的荒唐的假定。

确实有些程序员在故意地使用无定义的特性，但我想大多数的程序员都会很有头脑地避开任何的无定义特性。我们不应该效仿前一部分程序员。对于程序员来说，无定义的特性就相当于非法的特征，因此要利用断言对其进行检查。倘若本想调用 memmove，却调用了 memcpy，难道你不想知道自己搞错了吗？

通过增加一个可以验证两个内存块绝不重叠的断言，可以把 memcpy 加强如下：

```
/* memcpy ——— 拷贝不重叠的内存块 */
void memcpy(void* pvTo, void* pvFrom, size_t size)
{
void* pbTo = (byte*)pvTo;
void* pbFrom = (byte*)pvFrom;
```

```
ASSERT(pvTo != NULL && pvFrom != NULL);
ASSERT(pbTo>=pbFrom+size || pbFrom>=pbTo+size);
while(size-->0)
    *pbTo++ == *pbFrom++;
return(pvTo);
}
```

读者可能会认为上面的加强不大明显，怎么只用了一行语句就完成了重叠检查呢？其实只要把两个内存块比作两辆在停车处排成一行等候的轿车，就可以很容易明白其中的道理。我们知道，如果一辆车的后保险杠在另一辆车的前保险杠之前，两辆车就不会重叠。上面的检查实现的就是这个思想，那里 pbTo 和 pbFrom 是两个内存块的“后保险杠”。PbTo+size 和 pbFrom+size 分别是位于其相应“前保险杠”之前的某个点。就是这么简单。

顺便说一句如果读者还没有认识到重叠填充的严重性，只要考虑 pbTo 等于 pbFrom+1 并且要求至少要移动两个字节这一情况就清楚了。因为在这种情况下，memcpy 的结果是错误的。

所以从今以后，要经常停下来看看程序中有没有使用无定义的特性。如果程序中使用了无定义的特性就要把它从相应的设计中去掉，或者在程序中包括相应的断言，以便在使用了无定义的特性时，能够向程序员发出通报。

这种做法在为其他的程序员提供代码库（或操作系统）时显得特别重要。如果读者以前曾经为他人提供过类似的库，就知道当程序员试图得到所需的结果时，他们会利用各种各样的无定义特性。更大的挑战在于改进后新库的发行，因为尽管新库与老库完全兼容，但总有半数的应用程序在试图使用新库时会产生瘫痪现象，问题在于新库在其“无定义的特性”方面，与老库并不 100％兼容。

要从程序中删去无定义的特性
或者在程序中使用断言来检查出无定义特性的非法使用

不要让这种事情发生在你的身上

在 1988 年晚些时候，Microsoft 公司的摇钱树 DOS 版 Word 被推迟了三个月，明显地影响了公司的销售。这件事情的重要原因，是整整六个月来开发小组成员一直认为他们随时都可以交出 Word。

问题出在 Word 小组要用到的一个关键部分是由公司中另一个小组负责开发的。这个小组一直告诉 Word 小组他们的代码马上就可以完成，而且该小组的成员对此确信不疑。但他

们没有意识到的是在他们的代码中充斥了错误。

这个小组的代码与Word代码之间一个明显的区别是Word代码从过去到现在一直都使用断言和调试代码，而他们的代码却几乎没有使用断言。因此，其程序员没有什么好的办法可以确定其代码中的实际错误情况，错误只能慢慢地暴露出来。如果他们在代码中使用了断言，这些错误本该在几个月之前就被检查出来。

大呼“危险”的代码

尽管我们已经到了新的题目，但我还是想再谈谈 memcpy 中的重叠检查断言。对于上面的重叠检查断言：

```
ASSERT(pbTo>=pbFrom+size || pbFrom>=pbTo+size);
```

假如在调用 memcpy 时这个断言测试的条件为真，那么在发现这个断言失败了之后，如果你以前从来没见过重叠检查，不知道它是怎么回事，你能想到发生的是什么查错吗？我想我大概想不出来。但这并不是说上面的断言技巧性太强、清晰度不够，因为不管从那个角度看这个断言都很直观。然而直观并不等于明显。

请相信我的话，很少比跟踪到了一个程序中用到的断言，但却不知道该断言的作用这件事更令人沮丧的了。你浪费了大量的时间，不是为了排除错误，而只是为了弄清楚这个错误到底是什么。这还不是事情的全部，更有甚者程序员偶尔还会设计出有错的断言。所以如果搞不清楚相应断言检查的是什么，就很难知道错误是出现在程序中，还是出现在断言中。幸运的是，这个问题很好解决，只要给不够清晰的断言加上注解即可。我知道这是显而易见的事情，但令人惊奇的是很少又程序员这样做。为了使用户避免错误的危险，程序员们经历了各种磨难，但却没有说明危险到底是什么。这就好比一个人在穿过森林时，看到树上钉着一块上书“危险”红字的大牌子。但危险到底是什么？树要倒？废矿井？大脚兽？除非告诉人们危险是什么或者危险非常明显，否则这个牌子就起不到帮助人们提高警觉的作用，人们会忽视牌子上的警告。同样，程序员不理解的断言也会被忽视。在这种情况下，程序员会认为相应的断言是错误的，并把它们从程序中去掉。因此，为了使程序员能够理解断言的意图，要给不够清楚的断言加上注解。

如果在断言中的注解还注明了相应错误的其他可能解法，效果更好。例如在程序员使用相互重叠的内存块调用 memcpy 时，就是这样做的一个好机会。程序员可以利用注解指出此时应该使用 memmove，它不但能够正好完成你想做的事情，而且没有不能重叠的限制：

```
/* 内存块重叠吗？如果重叠，就使用 memmove */
ASSERT(pbTo>=pbFrom+size || pbFrom>=pbTo+size);
```

在写断言的注解时，不必长篇大论。一般的方法时使用经过认真考虑过的间断文句，它可能比用一整段的文字系统地解释出每个细节地指导性更强。但要注意，不要在注解中建议

解决问题的办法，除非你能确信它对其他程序员确有帮助。做注解的人当然不想让注解把别人引入歧途。

不要浪费别人的时间 ——— 详细说明不清楚的断言

不是用来检查错误的

当程序员刚开始使用断言时，有时会错误地利用断言去检查真正地错误，而不去检查非法的况。看看在下面的函数 strdup **中的两个断言：**

```
char* strdup(char* str)
{
char* strNew;
ASSERT(str != NULL);
strNew = (char*)malloc(strlen(str)+1);
ASSERT(strNew != NULL);
strcpy(strNew, str);
return(strNew);
}
```

第一个断言的用法是正确的，因为它被用来检查在该程序正常工作时绝不应该发生的非法情况。第二个断言的用法相当不同，它所测试的是错误情况，是在其最终产品中肯定会出现并且必须对其进行处理的错误情况。

你又做假定了吗？

有时在编程序时，有必要对程序的运行环境做出某些假定。但这并不是说在编程序时，总要对运行环境做出假定。例如，下面的函数 memset 就没对其运行环境做出任何的假定。因此它虽然未必效率很高，但却能够运行在任何的 ANSI C 编译程序之下：

```
/* memset ——— 用“byte”的值填充内存 */
void* memset(void* pv, byte b, size_t size)
{
byte* pb = (byte*)pv;
while(size-- > 0)
    *pb++ = b;
return(pv);
```

```
}
```

但是在许多计算机上通过先将要填充到内存块中的小值拼成较大的数据类型，然后用拼出的大值填充内存，由于实际填充的次数减少了，可使编出的 memset 函数速度更快，例如在 68000 上，下面的 mernset 函数的填充速度比上面的要快四倍。

```
/* longfill ——— 用“long”的值填充内存块。在填完了最后一个长字之后，
 * 返回一个指向所填第一个长字的指针。
*/
long* longfill(long* pl, long l, size_t size);          /* 原型 */
memset(void* pv, byte b, size_t size)
{
byte* pb = (byte*)pv;
if(size >= sizeThreshold)
    {
    unsigned long l;
    l = (b<<8) | b;          /* 用 4 个字节拼成一个长字 */
    l = (l<<16) | l;
    pb = (byte*)longfill((long*)pb, l, size/4);
    size = size%4;
    }
while(size-- > 0)
    *pb++ = b;
return(pv);
}
```

在上面的程序中可能除了对 sizeThreshold 所进行的测试之外，其它的内容都很直观。如果读者还不太明白为什么要进这一测试,那么可以想一想无论是将 4 个字节拼成一个 long 还是调用 longfill 函数都要花一定的时间。对 sizeThreshold 进行测试是为了使 memset 只有在用 long 进行填充速度更快时才进行相应的填充。否则就仍用 byte 进行填充。

这个 memset 新版本的唯一问题是它对编译程序和操作系统都做了一些假定。例如，这段代码很明显地假定 long 占用四个内存字节，该字节的宽度是八位。这些假定对许多计算机都正确，目前几乎所有的微机都毫无例外。不过这并不意味着因此我们就应该对这一问题置之不理，因为现在正确并不等于今后几年也正确。

某些程序员“改进”这一程序的方法，是把它写成下面这种可移植性更好的形式：

```
memset(void* pv, byte b, size_t size)
{
byte* pb = (byte*)pv;
if(size >= sizeThreshold)
    {
```

```
    unsigned long l;
    size_t sizeSize;
    l = 0;
    for(sizeSize=sizeof(long); sizeSize-->0; NULL)
        l = (l<<CHAR_BIT) | b;
    pb = (byte*)longfill((long*)pb, l, size/sizeof(long));
            size = size%sizeof(long);
    }
while(size-- > 0)
    *pb++ = b;
return(pv);
}
```

由于在程序中大量使用了运算符 sizeof，这个程序看起来移植性更好，但“看起来”不等于“就是”。如果要把它移到新的环境中，还是要对其进行考察才行。例如，如果在 Macintosh plush 或者其它基于 68000 的计算机上运行这个程序，假如 pv 开始指向的是奇数地址，该程序就会瘫痪。这是因为在 68000 上，byte*和 long*是不可以相互转换的类型，所以如果在奇数地址上存储 long 就会引起硬件错误。

那么到底应该怎么做呢?

其实在这种情况下，根本就不应该企图将 memset 写成一个可移植的函数。要接受其不可移植这一事实，不要对其进行改动。对于 68000，要避免上述的奇数地址问题，可以先用 byte 进行填充，填到偶数地址之后，再换用 long 继续填充。虽然将 long 对齐在偶数地址上已经可以工作了，但在各种基于 68020，68030 和 68040 的新型 Macintosh 上，如果使其对齐在 4 字节边界上，性能会更好。至于对程序中所做的其他假定，可以利用断言和条件编译进行相应的验证：

```
memset(void* pv, byte b, size_t size)
{
byte* pb = (byte*)pv;
#ifdef MC680x0
if(size >= sizeThreshold)
    {
    unsigned long l;
    ASSERT(sizeof(long)==4 && CHAR_BIT==8);
    ASSERT(sizeThreshold>=3);
    /* 用字节进行填充，直到对齐在长字边界上 */
    while( ((unsigned long)pb & 3) != 0 )
    {
        *pb++ = b;
```

```
        size--;
    }
    /* 现在拼装长字并用长字填充其他内存单元 */
    l = (b<<8) | b;          /* 用 4 个字节拼成一个长字 */
    l = (l<<16) | l;
    pb = (byte*)longfill((long*)pb, l, size/4);
    size = size%4;
    }
    #endif  /* MC680x0 */
while(size-- > 0)
    *pb++ = b;
return(pv);
}
```

正如读者所见，该程序中与具体机器相关的部分已被 MC680x0 预处理程序的定义设施括起。这样不仅可以避免这部分不可移植的代码被不小心地用到其它不同的目标机上，而且通过在程序中搜寻 MC680x0 这个字符串，可以找出所有与目标机有关的代码。

为了验证 long 占用 4 个内存字节、byte 的宽度是 8，还在程序中加了一个相当直观的断言。虽然暂时不太可能发生改变的，但谁知道以后会不会发生变化呢？

最后，为了在调用 longfill 之前使 pb 指向 4 字节边界上，程序中使用了一个循环。由于不管 size 的值如何，这个循环最终都会执行到 size 等于 3 的倍数，所以在循环之前还加了个检查 sizeThreshold 是否至少是 3 的断言（sizeThreshold 应该取较大的值。但它至少应该是 3，否则程序就不会工作）。

经过了这些改动，很明显这个程序已不再可移植。原先所做的假定或者已经被消除，或通过断言进行了验证。这些措施使得该程序极少可能被不正确地使用。

消除所做的隐式假定，或者利用断言检查其正确性

光承认编译程序还不够

最近，Microsoft 的一些小组渐渐发现他们不得不对其代码进行重新的考察和整理，因为相当多的代码中充满了“ +2” 而不是“ +sizeof(int)”、与 0xFFFF 而不是 UINT_MAX 进行无符号数的比较、在数据结构中使用的是 int 而不是真正想用的 16 位数据类型这一类问题。

你也许会认为这是因为这些程序员太懒散，但他们却不会同意这一看法。事实上，他们认为有很好的理由说明他们可以安全地使用“ +2” 这种形式，即相应的 C 编译程序是由

Microsoft 自己编写的。这一点给程序员造成了安全的假象，正如几年前一位程序员所说：“ 编译程序组从来没有做使我们所有程序垮掉的改变”。

但这位程序员错了。

为了在 Intel 80386 和更新的处理器上生成更快更小的程序，编译程序组改变了 int 的大小（以及其他一些方面）。虽然编译程序组并不想使公司内部的代码垮掉，但是保持在市场上的竞争地位显然更重要。毕竟，这是那些自己做了错误假定的 Microsoft 程序员的过错。

不可能的事用也能发生？

函数的形参并不一定总是给出函数的所有输入数据，有时它给出的只是一个指向函数输入数据的指针。例如，请看下面这个简单的压缩还原程序：

```
byte* pbExpand(byte* pbFrom, byte* pbTo, size_t sizeFrom)
{
byte b, *pbEnd;
size_t size;
pbEnd = pbFrom + sizeFrom;        /* 正好指向缓冲区尾的下一个位置 */
while(pbFrom < pbEnd)
{
    b = *pbFrom++;
    if(b == bRepeatCode)
    {
        /* 在 pbTo 开始的位置存储“size”个“b” */
        b = *pbFrom++;
        size = (size_t)*pbFrom++;
        while(size-- > 0)
            *pbTo++ = b;
    }
    else
        *pbTo++ = b;
    }
    return(pbTo);
}
```

这个程序将一个数据缓冲区中的内容拷贝到另一个数据缓冲区中。但在拷贝过程中，它要找出所有的压缩字符序列。如果在输入数据中找到了特殊的字节 bRepeatCode，它就认为其后的下两个字节分别是要重复的还原字符以及该字符的重复次数。尽管这一程序显得有些过于简单，但我们还是可以把它们用在某些类似于程序编辑的场合下。那里，正文中常常包括有许多表示缩进的连续水平制表符和空格符。

为了使 pbExpand 更健壮，可以在该程序的入口点加上一个断言，来对 pbFrom、sizeFrom 和 pbTo 的有效性进行检查。但除此之外，还有许多其它可以做的事情。例如，还可以对缓冲区中的数据进行确认。

由于进行一次译码总需要三个字节，所以相应的压缩程序从不对两个连续的字符进行压缩。另外，虽然也可以对三个连续的字符进行压缩，但这样做并不能得到什么便宜。因此，压缩程序只对三个以上的连续字符进行压缩。

存在一个例外的情况。如果原始数据中含有 bRepeatCode，就必须对其进行特殊的处理。否则当使用加 pbExpand 时，就会把它误认为是一个压缩字符序列的开始。当压缩程序在原始数据中发现了 bRepeatCode 时，就把它再重复一次，以便和真正的压缩字符序列区别。

总之，对于每个字符压缩序列，其重复次数至少是 4，或者是 1。在后一种情况下，相应的重复字符一定是 bRepeatCode 本身。我们可以使用断言对这一点进行验证：

```
byte* pbExpand(byte* pbFrom, byte* pbTo, size_t sizeFrom)
{
byte b, *pbEnd;
size_t size;
ASSERT(pbFrom != NULL && pbTo != NULL && sizeFrom != 0);
pbEnd = pbFrom + sizeFrom;       /* 正好指向缓冲区尾的下一个位置 */
while(pbFrom < pbEnd)
{
    b = *pbFrom++;
    if(b == bRepeatCode)
    {
        /* 在 pbTo 开始的位置存储“size”个“b” */
        b = *pbFrom++;
        size = (size_t)*pbFrom++;
        ASSERT( size>=4 || (size==1 && b==bRepeatCode) );
        while(size-- > 0)
            *pbTo++ = b;
    }
    else
        *pbTo++ = b;
    }
```

```
    return(pbTo);
}
```

如果这一断言失败说明 pbFrom 指向的内容不对或者字符压缩程序中有错误。无论哪种情况都是错误，而且是不用断言就很难发现的错误。

利用断言来检查不可能发生的情况

安静地处理

假如你受雇为核反应堆编写软件，就必须对堆芯过热这一情况进行处理。

某些程序员解决这个问题的方法可以是自动地向堆芯灌水、插入冷却棒或者是能使反应堆冷却下来的一些其他什么方法。而且，只要程序已经控制了势态就不必向有关人员发出警报。

另一些程序员可能会选择另一种方法，即只要堆芯过热就向反应堆工作人员发出警报。虽然相应的处理仍由计算机自动进行，不同的是操作员总是知道这件事。

如果由你来实现这一程序，你会选择哪一种方法？

我想关于这一点，大家基本上不会有太多的异议，即总是应该向操作人员发出警报，这与计算机能够恢复反应堆的正常操作是两回事。堆芯不会无缘无故地出现过热现象，一定是发生了某种不同寻常的事情，才会引起这一故障。因此在计算机进行相应处理的同时，最好使操作人员搞清楚发生了什么事情以避免事故的再次发生。

令人惊奇的是，程序员，尤其是有经验的程序员编的程序通常都是这样：当某些意料不到的事情发生时，程序只进行无声无息的安静处理，甚至有些程序员会有意识地使程序这样做。也许你自己用的是另一种方法。

当然，我现在谈的是所谓的防错性程序设计。

在上一节中，我们介绍 pbExpand 程序。该函数使用的就是防错程序设计。但从其循环条件可以看出，下面的修改版本并没有使用防错性程序设计。

```
byte* pbExpand(byte* pbFrom, byte* pbTo, size_t sizeFrom)
{
byte b, *pbEnd;
size_t size;
pbEnd = pbFrom + sizeFrom;       /* 正好指向缓冲区尾的下一个位置 */
while(pbFrom != pbEnd)
{
    b = *pbFrom++;
    if(b == bRepeatCode)
    {
        /* 在 pbTo 开始的位置存储“size”个“b” */
```

```
            b = *pbFrom++;
            size = (size_t)*pbFrom++;
            do
                *pbTo++ = b;
            while(size-- != 0)
        }
        else
            *pbTo++ = b;
        }
        return(pbTo);
    }
```

虽然这一程序更精确地反应了相应的算法，但有经验的程序员很少会这样编码。否则好机会就来了，我们可以把他们塞进一辆既没有安全带又没有车门的双人 Cessna 车中。上面的程序使人感到太危险了。

有经验的程序员会这样想：“我知道在外循环中 pbFrom 绝不应该大于 pbEnd，但如果确实出现了这种情况怎样办呢？还是在这种不可能的情况出现时，让外循环退出为好。”

同样，对于内循环，即使 size 总应该大于或等于 1，但使用 while 循环代替 do 循环，可以保证进入内循环时一旦 size 为 0，不至于使整个程序瘫痪。

使自己免受这些“不可能”的打扰似乎很合理，甚至很聪明。但如果出于某种原因 pbFrom 被加过了 pbEnd，那么会发生什么事情呢？在上面这个充满危险的版本或者前面看到的防错性版本中，找出这一错误的可能性又有多大呢？当发生这一错误时，上面的危险版本也许会引起整个系统的瘫痪，因为 pbExpand 会企图对内存中的所有内容进行压缩还原。在这种情况下，用户肯定会发现这一错误。相反，对于前面的防错性版本来说，由于在 pbExpand 还没来得及造成过多的损害（如果有的话）之前，它就会退出。所以虽然用户仍然可能发现这一错误，但我看这种可能性不大。

实际的情况就是这样，防错性程序设计虽然常常被誉为有较好的编码风格，但它却隐瞒了错误。要记住，我们正在谈论的错误决不应该再发生，而对这些错误所进行的安全处理又使编写无错代码变得更加困难。当程序中有了一个类似于 pbFrom 这样的跳跃性指针。并且其值在每次循环都增加不同的量时，编写无错代码尤其困难。

这是否意味着我们应该放弃防错性程序设计呢？

答案是否定的。尽管防错性程序设计会隐瞒错误，但它确实有价值。一个程序所能导致的最坏结果是执行崩溃，并使用户可能花几个消失建立的数据全部丢掉。在非理想的世界中，程序确实会瘫痪，因此为了防止用户数据丢失而参去的任何措施都是值得的。防错性性程序设计要实现的就是这个目标。如果没有它，程序就会如同一个用纸牌搭起的房子，哪怕硬件或操作系统中发生了最轻微的变化，都会塌落。同时，我们还希望在进行防错性程序设计时，错误不要被隐瞒。

假定某个函数以无效的参数调用了 pbExpand，比如 sizeFrom 比较小并且数据缓冲区最

后一个字节的内容碰巧是 bRepeatCode。由于这种情况类似于一个压缩字符序列，所以加 pbExpand 将从数据缓冲区外多读 2 个字节，从而使 pbFrom 超过 pbEnd。结果呢？pbExpand 的危险版可能会瘫痪，但其防错性版本或许可以避免用户数据的丢失，尽管它也可能冲掉 255 个字节的未知数据。既然两者都想得到，既需要调试版本对错误进行报警，又需要交付版本对错误进行安全的恢复，那么可以一方面一如既往地利用防错性程序设计进行编码，另一方面在事情变糟地情况下利用断言进行报警：

```
byte* pbExpand(byte* pbFrom, byte* pbTo, size_t sizeFrom)
{
byte b, *pbEnd;
size_t size;
pbEnd = pbFrom + sizeFrom;        /* 正好指向缓冲区尾的下一个位置 */
while(pbFrom != pbEnd)
{
    b = *pbFrom++;
    ……
    }
    ASSERT(pbFrom == pbEnd);
    return(pbTo);
}
```

上面的断言只是用来验证该函数的正常终止。在该函数的交付版本中，相应的防错措施可以保证当出了毛病时，用户可以不受损失；而在该函数的调试版本中，错误仍然可以被报告出来。

但是在实际的编程中也不必过分拘泥于此。例如，如果每次循环 pbFrom 的内容总是增 1，那么要使 pbFrom 超过 pbEnd 从而引起问题，恐怕需要一束宇宙射线的偶然轰击才行。在这种情况下，相应的断言没什么用处，因此可以从程序中删去。在程序中究竟是否需要使用断言，要根据常识视具体的问题而定。最后应该说明的是，循环只是程序员通常用来进行防错性程序设计的一个方面。实际上，无论把这种程序设计风格用在哪里，在编码之前都要同自己：“在进行防错性程序设计时，程序中隐瞒错误了吗？”如果答案是肯定的，就要在程序中加上相应的断言，以对这些错误进行报警。

在进行防错性程序设计时，不要隐瞒错误

两个算法比一个算法好

为了捕捉程序中的错误，只对坏的输入和有漏洞的假定进行检查是不够的。正如调用函数可能给被调用函数传递无用信息一样，被调用函数也可能给调用函数返回无用信息。二者都是我们所不期望的。

由于无论 memcpy 还是 memset 只有一个返回参数，所以使它们返回无用信息的可能性极小。但对于更复杂的程序，也许就不太容易做出这一结论了。

比如，最近我为 Macintosh 程序员编写了一个开发工具的部分程序：68000 反汇编程序。在该程序的编写中，我对它能够运行得多快并不在意，关键是要工作正确。因此，我选择了简单的表格驱动算法来实现这一程序，因为它比较容易测试。在程序中，我还使用了断言，以便在测试期间捕捉到可能遗漏的错误。

如果读者以前曾经看过汇编语言参考书，那么很幸运，因为这种书通常都会精心描述出每条指令的详细情况，如每条指令对应的二进制形式。例如，如果在 68000 汇编语言参考手册中查阅 ADD 指令，就可以知道它有如下的二进制形式：

ADD：

15	14	13	12	11 10 9	8 7 6	5 4 3 2 1 0
1	1	0	1	Register	Op-Mode	Effective Address Mode | Register

我们可以忽略指令中的 Register 和 Mode 域，而只对其中明显标为 0 或 1 的二进制位感兴趣。在 ADD 的情况下，我们只对其高 4 位感兴趣。如果去掉该指令中没有明显标为 0 或 1 的其它进制位，然后检查其高 4 位是否为 1101 或 16 进制数 0xd，就可以知道该指令是否是 ADD 指令：

```
if( (inst & 0xf000) == 0xd000 )
它是条 ADD 指令 ……
```

用来进行带符号相除的 DIVS 指令模式中有 7 个被明显标为 0 或 1 的二进制位：

DIVS：

15	14	13	12	11 10 9	8	7	6	5 4 3 2 1 0
1	0	0	0	Register	1	1	1	Effective Address Mode | Register

同样，如果去掉该指令中没有被明显标为 0 或 1 的 Register 和 Mode 域，就可以知道该指令是否是 DIVS 指令：

```
if( (inst & 0xf1c0) == 0x81c0 )
它是条 DIVS 指令 ……
```

可以用这种先屏蔽后测试的办法来检查每条汇编指令，一旦确认为 ADD 或 DIVS 指令就可调用译码函数恢复刚才忽略的 Register 和 Mode 区域的内容。

这就是我设计的反汇编程序的工作方式。

自然，该程序并没有使用 142 个条件不同的 if 语句来实现对所有可能的 142 条指令进行检查而是使用一个含有屏蔽码、指令特征和译码函数的表格对每条指令进行检查。查表程序循环检查指令，如果匹配上某条指令，就调用相应的译码程序译出该指令的 Register 和 Mode 域。

下面给出这个表格的部分内容以及使用该表的部分代码：

```
/* idInst 是个屏蔽码和指令特征组成的表格，
* 其内容表示了不同类型指令的二进制位模式。
*/
static identity idInst[] =
{
{ 0xFF00,   0x0600,     pcDecodeADDI },     /* 屏蔽码、特征及函数 */
{ 0xF130,   0xD100, pcDecodeADDX },
{ 0xF000,   0xD000, pcDecodeADD },
{ 0xF000,   0x6000,     pcDecodeBcc },      /* 短转移 */
{ 0xF1C0,   0x4180,     pcDecodeCHK },
{ 0xF138,   0xB108,     pcDecodeCMPM },
{ 0xFF00,   0x0C00,     pcDecodeCMPI },
{ 0xF1C0,   0x81C0,     pcDecodeDIVS },
{ 0xF100,   0xB100,     pcDecodeEOR },
/* …… */
{ 0xFF00,   0x4A00, pcDecodeTST },
{ 0xFFF8,   0x4E58,     pcDecodeUNLK },
{ 0x0000,   0x0000,     pcDecodeError }
};

/* pcDisasm
* 反汇编一条指令，并将其填入操作码结构 opc 中。
* pcDisasm 返回一个修改过的程序计数器
*
* 典型用法：pcNext = pcDisasm(pc, &opc);
*/
instruction* pcDisasm(instruction* pc, opcode* popcRet)
{
identity* pid;
instruction inst = *pc;
for(pid=&idInst[0]; pid->mask!=0; pid++)
{
    if( (inst & pid->mask) ==  pid->pat )
        break;
}
return( pid->pcDecode(inst, pc+1, popcRet) );
}
```

我们看到，函数 pcDisasm 并不很大。它使用的算法非常简单：先读入当前的指令，在表中查出其对应的内容；然后调用相应的译码程序在 popcRet 指向的结构 opcode 中填入相应的内容；最后返回一个修改后的程序计数器。由于并不是所有的 68000 指令长度都相同，所以必须对程序计数器进行相应的修改。如果有必要的话，在上面译码程序的参数中还可以包括指令的其它域，但仍然要把新的程序计数器值返回给 pcDisasm。

现在，我们再回到原先的问题。

通过类似于 pcDisasm 这样的函数，程序员很难知道其返回的数据是否有效。或许 pcDisasm 自己能够正确地识别指令，但其用到的译码程序却可能产生无用信息，而且这一问题很难发现。捕捉这种错误的一个方法是在每条指令对应的译码程序中都加上断言。这样做虽然也可以达到目的，但更好的方法是在 pcDisasm 中加上相应的断言，因为它是调用所有译码程序的关键之处。

问题是怎样才能做到这一点，怎样才能在以 pcDisasm 中检查出相应译码程序对结构 opcode 的填写是否正确呢？回答是必须编写相应的程序对该结构中填写的内容进行确认。怎么确认呢？这基本上是说我们必须写一个子程序，用 68000 的指令同结构 opcode 的填写内容进行比较。换句话说，必须再写一个反汇编程序。

这听起来好象有点令人发疯，真的需要这样吗？

还是让我们看看 Microsoft Excel 重新计算工具的做法吧。由于速度是电子表格软件成功的关键，所以为了保证绝不对其它无关单元中的公式重新计算，Excel 使用了一个相当复杂的算法。这样做的唯一问题是因为该算法过于复杂，所以对其进行修改难免会引进新的错误。Excel 的程序员当然不希望这种事情发生，所以他们又编写了一个只用在 Excel 调试版本的重新计算工具。当原来的重新计算工具完成了重新计算工作之后，再用这个重新计算工具对含有公式的所有单元进行一遍虽然缓慢但很彻底的重新计算。如果两次计算的结果不同，就会触发某个断言。

Microsoft Word 也遇到过类似的问题。由于字处理程序在进行页面布局时速度也很关键，所以 Word 的程序员用汇编语言编写了这部分程序，以便能够对其进行人工优化。这样一来，虽然速度上去了，但在防止程序有错方面却变得很糟。而且同不常发生变化的 Excel 重计算工具不同，Word 的页面布局程序需要随着 Word 新功能的增加而定期改变。因此为了能够自动地查出页面布局程序中的错误，Word 程序员为每个可进行人工优化的汇编语言程序都相应地写了一个 C 程序，如果两个版本产生的结果不一致，就触发某个断言。

同样，我们可以把上述方法用到我们的反汇编程序上来，即使用另一个只用作调试的反汇编程序来对第一个反汇编程序进行确认。

我不想让第 2 个反汇编程序 pcDisasmAlt 的实现细节打扰读者。简单地说，它是逻辑驱动的，而不是表格驱动的。它利用嵌套的 switch 语句不断地对指令中的有效位进行分离，直到分离出所需要的指令。下面的程序表明了利用 pcDisasmAlt 来确认第一个反汇编程序的方法：

```
instruction* pcDisasm(instruction* pc, opcode* popcRet)
{
```

```
identity* pid;
instruction inst = *pc;
instruction* pcRet;
for(pid=&idInst[0]; pid->mask!=0; pid++)
{
    if( (inst & pid->mask) ==  pid->pat )
        break;
}
pcRet = pid->pcDecode(inst, pc+1, popcRet);
#ifdef DEBUG
{
    opcode opc;
    /* 检查两个输出值的有效性 */
    ASSERT( pcRet == pcDisasmAlt(pc, &opc) );
    ASSERT( memcmp(popcRet, &opc, sizeof(opcode))==0 );
}
#endif
return(pcRet);
}
```

在正常的情况下，在现有代码中增加调试检查代码不应该对原有代码产生其它的影响，但在本程序中无法做到这一点。因为我们必须建立一个局部对象 pcRet，以便对 pid->pcDecode 返回的指针值进行确认。幸好这并没有违反“除了原有代码之外，还应该执行所加入的调试代码，而且加入了调试代码之后，仍然要执行原有的代码”这条基本的准则，因此这样做还可以接受。这条基本准则虽然说得再清楚不过了，但一旦开始使用断言和调试代码，有时仍会企图用所加入的调试代码取代原有代码的执行。在第 3 章中我们将看到一个这样的例子，但现在还是让我们说：“要抑制这一冲动”。虽然为了进行相应调试检查我们不得不对 pcDisasm 进行了相应的修改，但所加入的调试代码并没有代替原有代码的执行。

上面的做法并不意味着程序中的每个函数都得有两个版本，因为那无疑与浪费时间使每个函数部尽可能效率很高的做法一样荒谬。正确的做法是只对程序中的关键部分这样做。我确信大多数的程序都有必须做好的关键部分，例如电子表格软件中的重新计算程序、字处理程序中的页面布局程序、项目管理程序中的任务调度程序以及数据库中的搜索 / 抽取程序。另外，每个程序中用来保证用户数据不被丢失的部分也是其关键部分。

当编写代码时，要抓住一切机会对程序的结果进行验证（调用所有其它函数的瓶颈函数，是特别适于进行这种检查的好地方）。要尽可能地使用不同的算法，而目要使其不仅仅是同一算法的又一实现。通过使用不同的算法不仅可以发现算法实现中的错误，而且还增加了发现算法本身错误的可能性。

要利用不同的算法对程序的结果进行确认

嘿，这是怎么回事？

在本章的早些时候曾经说过，在定义宏 ASSERT 时必须谨慎从事。其中特别提到它不能移动内存的内容，不能调用其它的函数或者引起了其它不期望的副作用。既然如此，为什么下面的函数 pcDisasm 还使用了不符合上述要求的断言呢？

```
/* 检查两个输出值的有效性 */
ASSERT( pcRet == pcDisasmAlt(pc, &opc) );
ASSERT( memcmp(popcRet, &opc, sizeof(opcode))==0 );
```

之所以要禁止 ASSERT 调用其它的函数，是因为那样可能会对 ASSERT 周围的代码产生某种不可预料的影响。但在上面的代码中，调用其它函数的不是 ASSERT，而是作者，即 ASSERT 的使用者。因为我知道在 pcDisasm 中调用其它的函数很安全，不会引起问题，所以不用顾虑在该断言中使用函数调用。

一开始就要阻止错误的发生

到目前为止，我们一直忽略了指令中的 Register 和 Mode 域。那么如果这些位使对应指令与表中其它指令所对应的内容碰巧匹配上了，会发生什么事情呢？例如指令 EOR 的二进制形式如下：

EOR：

15	14	13	12	11 10 9	8	7 6	5 4 3 2 1 0
1	0	1	1	Register	1	Mode	Effective Address Mode | Register

而指令 CMPM 的二进制形式与其非常相似：

CMPM：

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	0	1	1	Register	1	Size	0	0	1	Register

注意，如果指令 EOR 的“Effective Address Mode”域为 001，那么看起来它就象条 CMPM 指令。因此如果 EOR 指令在 idInst 表中的位置比 CMPM 指令靠前，那么所有经过的 CMPM 指

令都会被错误地认为是 EOR 指令。

值得庆幸的是，由于 pcDisasm 和 pcDisasmAlt 使用的算法不同，所以在第一次对 CMPM 指令进行反汇编时，就会引起断言失败。原因是 pcDisasm 在 opcode 结构中填写的是 EOR 指令而 pcDisasmAlt 填写的则是我们所期待的正确指令 CMPM。因此在调试代码对两个结构进行内容比较时就会产生断言失败。这就是在调试函数中使用不同算法的威力。

但令人不快的是，只有在试图对 CMPM 指令进行反汇编的时候这一错误才会被发现。当然，如果测试人员使用的测试集内容足够详尽是可以发现这一错误了。然而，读者还记得我在第 1 章中曾经说过的话吗？我们追求的是尽可能早地自动查出程序中的错误。而且在查错时不应该依赖于其他的人。

因此虽然我们也可以把这一任务推给测试组，但不要那么做。尽管相当多的程序员认为测试者就是要为自己测试程序，但要知道，测试者的工作并不只是对你的程序进行测试，查出自己程序中的错误毕竟是你自己的工作。如果你不同意这一观点，那么请举出一个只因为有人进行错误检查就可以草率从事的其它工作来。既然没有，那为什么程序设计应该例外呢？如果你想要始终如一地编写出没有错误的代码，就必须采取措施负起责。所以，还是让我们从现在就开始做起吧。

在进行程序设计时，只要注意到程序中存在某些危险的因素，就可以问自己：“怎样才能自动地及早查出这个错误呢？”通过习惯性地就这个问题不断向自己发问，你会发现使程序更加健壮的各种方法。

例如要查出上面的错误，可以在该程序的初始化完成之后，立即在 main 函数中对该表进行扫描，通过查看其每项的内容来验证先前没有哪个入口不正确地截取了其它的指令。检查这种错误的表格检查程序虽然不长，但并不清晰：

```
void CheckIdInst(void)
{
identity *pid, *pidEarlier;
instruction inst;
for(pid=&idInst[0]; pid->mask!=0; pid++)
{
    for(pidEarlier=&idInst[0]; pidEarlier<pid; pidEarlier++)
    {
        inst = pid->pat | (pidEarlier->pat & pid->mask);
        if( (inst & pidEarlier->mask) == pidEarlier->pat )
        ASSERT( bitcount(pid->mask) < bitcount(pidEarlier->mask) );
    }
}
}
```

该程序通过将当前指令与表中存放在该指令前面的每条指令进行比较来检查这种错误。我们知道，每条指令都有其“不用考虑”的位，即那些在形成其指令特征时被屏蔽掉的位。

但是如果这些“不用考虑”的位碰巧使对应的指令与表中前面的指令匹配上了，会发生什么情况呢？在这种情况下，就会产生表中两个入口之间的冲突。那么对于可能产生冲突的入口，它们在表中的位置究竟应该谁先谁后呢？

答案很简单。如果表中两个入口都与同一条指令匹配那么必须把含 1 多的入口放在表中的前面。如果这还不够直观，请考虑一下指令 EOR 和 CMPM 的二进制形式。假如这两条指令所对应的入口对于同一条指令匹配，那么应该选择哪个入口作为“正确的”匹配，为什么？因为凡是指令二进制形式中被明确标为 0 或 1 的位，其对应的屏蔽位都是 1，所以通过对相应屏蔽码中 1 的个数进行比较就可以知道哪个入口更正确。

如果两条指令相互冲突，区别起来就更复杂。具体处理方法是先取出一个人口中的指令特征，然后强制使其那些“不用考虑”的位与表中前面每条指令的特征精确匹配。这一操作所产生的指令特征，就是上面程序中赋给变量 inst 的值。根据设计我们知道，所形成的指令 inst 必定与当前的指令匹配，因为它只改变了当前指令中那些与指令特征没关系的位。但除此之外如果它还同表中前面的其它指令匹配就会产生两个入口之间的冲突，因此必须进行相应屏蔽码的比较。

通过在程序的初始化过程中调用 CheckIdInst，一开始执行反汇编时就可以查出以上的冲突错误，而不必等到进行反汇编时才能发现。程序员应该在程序中寻找类似的初始检查，这样可以尽快地发现错误。否则，错误就会隐藏一段时间。

不要等待错误发生，要使用初始检查程序

一个告诫

一旦开始使用断言，也许就会发现程序中的错误会显著增加。人们对此如果没有心理准备就会感到惊惶失措。

我曾经重写了一个由几个 Microsoft 小组共享的代码，它有许多的错误，因为在编写原来的代码库时没有使用断言，但我在新版库中加上了断言。使我吃惊的是，当我把新版代码交给这些小组使用之后，一个程序员竟很生气地要求我把原来的代码库还给他。

我问他为什么？

他说：“安装这个库后出现了许多错误。”

“你是说新库引起了错误？”，我感到震惊。

“好象是这样，我们遇到了许多过去没有的断言。”

“ 你们检查过这些断言吗？”，我问道。

“ 检查过，它们在我们的程序中是错误的。这么多的断言不可能全没错误，而且我们也没有时间去跟踪这些根本不属于我们的问题。我想要回原来的库。”

我当然不认为他们没有问题。所以我请求他继续使用新库，直到发现某个断言有错为止。他虽然心里不高兴，但还是答应了我的请求。结果，他发现所有的错误都出在他们自己的项目，而不是我的新库中。

正因为我事前没有告诉他们新库中已加上断言，所以这个程序员才会感到惊慌，因为没有人想出错。但如果我告诉大家出现了断言失败是件好事，也许这个程序员就不会那么惊慌了。然而，对错误感到惊慌的并不止程序员。因为公司是通过项目尚未完成功能的数目和项目中出现的明显错误数目来进行项目评估的，所以每当这些数字显著增加时，项目组中的每个人都会变得精神紧张。因此，要让别人知道你增加断言的打算。否则，就得为他们准备一些 EXCEDRIN。

小结

在本章中，我们介绍了利用断言对程序中的错误进行自动检查的方法。虽然这一方法对及早查出程序中的“最后错误”非常有价值，但同其它的工具一样，断言也可能被过度使用，用户要自己灵活掌握断言的使用分寸。对于某些程序员来说，每次相除都利用断言检查分母是否为 0 可能很重要，但对其它的程序员来说，这可能很可笑。用户要自己做出恰当的判断。

在项目的整个生存期中，程序中都应该保留断言。在程序的交付之前不要把它们删去。在今后打算为程序增加新功能时，这些断言仍然有用。

要点

- 要同时维护交付和调试两个版本。封装交付的版本，应尽可能地使用调试版本进行自动查错。
- 断言是进行调试检查的简单方法。要使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是在最终产品中必须处理的。
- 使用断言对函数的参数进行确认，并且在程序员使用了无定义的特性时向程序员报警。函数定义得越严格，确认其参数就越容易。

- 在编写函数时，要进行反复的考查，并且自问：“我打算做哪些假定？”一旦确定了相应的假定，就要使用断言对所做的假定进行检验，或者重新编写代码去掉相应的假定。另外，还要问：“这个程序中最可能出错的是什么，怎样才能自动地查出相应的错误？”努力编写出能够尽早查出错误的测试程序。
- 一般教科书都鼓励程序员进行防错性程序设计，但要记住这种编码风格会隐瞒错误。当进行防错性编码时如果“不可能发生”的情况确实发生了，要使用断言进行报警。

练习

1) 假定你必须维护一个共享库并想在其中使用断言，但又不想发行这个库的源代码，那么怎样定义 ASSERTMSG 这个断言宏，才能在发生非法的情况下，显示一条有意义的信息来代替相区的文件名和行名？例如，memcpy 可能显示如下的断言：
   Assertion failed in memcpy: the blocks over lap
2) 每当使用 ASSERT，宏__FILE__就产生一个唯一的文件名字符串。这就是说，如果在同一个文件中使用了 73 个断言，编译程序就会产生 73 个完全相同的文件名字符串。怎样实现 ASSERT 宏，才能使文件名字符串在每个文件中只被定义一次？
3) 下面函数中的断言有什么问题？

```
/* getline ——— 将一个以\n 结尾的行读入缓冲区中 */
void getline(char* pch)
{
    int ch;      /* ch“必须”是 int */
do
    ASSERT( (ch = getchar()) != EOF );
While( (*pch++=ch) !=  ‘\n’  )
}
```

4) 当程序员为枚举类型增加新元素时，有时会忘记在相应的 switch 语句中增加新的 case 条件。怎样才能使用断言帮助查出这个问题？
5) CheckIdInst 能够验证 idInst 表中的内容次序正确，但表中还可能发生其它的问题。由于表中有许多数，很容易输入不正确的屏蔽码或指令特征。如何增强 CheckIdInst 的功能，使其能够自动地查出这种输入错误。
6) 如前所述，当指令 EOR 的“Effective Address Mode”域是 001 时，它就真的变成了一条 CMPM 指令。EOR 指令中还有其它的限制，例如其 MODE 域中的两位绝不能是 11（这会使它变成一条 CMPA L 指令），而且如果其“Effective Address Mode”域是 111，那么其“Effective Address Register”域必须是 000 或 001。由于对这些 EOR 的非法组合不应该调用 pcDecodeEOR，那么怎样为其加上可以查出表中这些错误的断言呢？

7） 怎样利用不同的算法对 qsort 函数进行验证？怎样对二分查找程序进行验证？又怎样验证 itoa 函数呢？

课题：

同编写所用操作系统的公司联系，促使有关人员为程序员提供一个调试版本。（顺便说一句，这是个尽赚不赔的买卖。因为开发操作系统的公司总是希望人们为其操作系统编写应用程序，这样可以使其操作系统产品更容易走向市场。

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。第一个理由是我不想让例子中到处都是 malloc、free 和 realloc 的实现代码。第二个理由是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**)ppv;
*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，忘了为新增加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

应该是用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，并不意味你也得接受这一想法。

relloc 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(*ppb, sizeNew);
    if( *pbResize != NULL )
        *ppb = 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 relloc 不一样，fResizeMemory 在操作失败的情况下并不返问空指针。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(*ppb);
    /* 如果缩小，冲掉块尾释放的内容 */
    if(sizeNew<sizeOld)
        memset((*ppb)+sizeNew, bGarbage, sizeOld-sizeNew);
}
#endif
pbResize = (byte*)realloc(*ppb, 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
        *ppb = 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，所以你也不应该再去动它。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(*ppb);
        /* 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((*ppb)+sizeNew, bGarbage, sizeOld-sizeNew);
            else if(sizeNew > sizeOld)
            {
            byte* pbNew;
            if( fNewMemory(&pbNew, sizeNew) )
            {
                memcpy(pbNew, *ppb, sizeOld);
                FreeMemory(*ppb);
                *ppb = pbNew;
            }
            }
        }
        #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，所分配的每个内分块都应该标有引用标记。最后，调用 CheckMemroyRefs 验证某个内存块没有引用标记，它将引发相应的断言，警自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)
{
    /* 在进入到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
```

最后一个问题是：“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，如果无法装人最大的电子表格，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凋试代码，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

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) 本章自始自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。

第 4 章　对程序进行逐条跟踪

前面我们讲过，发现程序中错误的最好方法是执行程序。在程序执行过程中，通过我们的眼睛或者利用断言和子系统一致性检查这些自动的测试工具来发现错误。然而，虽然断言和子系统检查都很有用，但是如果程序员事先没有想到应该对某些问题进行检查也不能保证程序不会遇到这些问题。这就好比家庭安全检查系统一样。

如果只在门和窗户上安装了警报线，那么当窃贼从天窗或地下室的入口进入家中时，就不会引起警报。如果在录像机、立体声音响或者其它一些窃贼可能盗取的物品上安装了干扰传感器，而窃贼却偷取了你的 Barry Manilow 组合音响，那么他很可能会不被发现地逃走。这就是许多安全检查系统的通病。因此，唯一保证家中物品不被偷走的办法是在窃贼有可能光顾的期间内呆在家里。防止错误进入程序的办法也是这样，在最有可能出现错误的时候，

必须密切注视。

那么什么时候错误最有时能出现呢？是在编写或修改程序的时候吗？确实是这样。虽然现在程序员都知道这一点，但他们却并不总能认识到这一点的重要性，并不总能认识到编写无错代码的最好办法是在编译时对其进行详尽的测试。

在这一章中，我们不谈为什么在编写程序时对程序进行测试非常重要，只讲在编写程序时对程序进行有效测试的方法。

增加对程序的置信

最近，我一直为 Microsoft 的内部 Macintosh 开发系统编写某个功能。但当我对所编代码进行测试时，发现了一个错误。经过跟踪，确定这个错误是出在另一个程序员新编的代码中。使我迷惑不解的是，这部分代码对其他程序员的所编代码非常重要，我想不出他这部分代码怎么还能工作。我来到他的办公室，以问究竟

“我想，在你最近完成的代码中我发现了一个错误”。我说道。“你能抽空看一下吗？”他把相应的代码装入编辑程序，我指给他看我认为的问题所在。当他看到那部分代码时不禁大吃一惊。

“你是对的，这部分代码确实有错。可是我的测试程序为什么没有查出这个错误呢？”

我也对此感到奇怪。“你到底用什么方法测试的这部分代码？”，我问道。

他向我解释了他的测试方法，听起来似乎它应该能够查出这个错误。我们都感到很费解。“让我们在该函数上设置一个断点对其进行逐条跟踪，看看实际的情况到底怎样”，我提议道。

我们给该函数设置了一个断点。但当找们按下运行键之后，相应的测试程序却运行结束了，它根本就没有碰上我们所设置的断点。没过多久，我们就发现了测试程序没有执行该函数的原因 ——— 在该函数所在调用链上几层，一个函数的优化功能使这个函数在某种情况下面跳过了不必要的工作。

读者还记得我在第 1 章中所说的黑箱测试问题吗？测试者给程序提供大量的输入，然后通过检查其对应的输出来判断该程序是否有问题。如果测试者认为相应的输出结果没有问题，那么相应的程序就被认为没有问题。但这种方法的问题是除了提供输入和接受输出之外，测试者再没有别的办法可以发现程序中的问题。上述程序员漏掉错误的原因是他采用了黑箱方法对其代码进行测试，他给了一些输入，得到了正确的输出，就认为该代码是正确的。他没有利用程序员可用的其他工具对其代码进行测试。

同大多数的测试者不同，程序员可以在代码中设置断点，一步一步地跟踪代码的运行，观察输入变为输出的过程。尽管如此，但奇怪的是很少有程序员在进行代码测试时习惯于对其代码进行逐条的跟踪。许多程序员甚至不耐烦在代码中设置一个断点，以确定相应代码是否被执行到了。

还是让我们回到这一章开始所谈论的问题上：捕捉错误的最好办法是在编写或修改程序时进行相应的检查。那么，程序员测试其程序的最好办法是什么呢？是对其进行逐条的跟踪，

对中间的结果进行认真的查看。对于能够始终如一地编写出没有错误程序的程序员，我并不认识许多。但我所认识的几个全都有对其程序进行逐条跟踪的习惯。这就好比你在家时夜贼光临了 ——— 除非此时你睡着了，否则就不会不知道麻烦来了。

作为一个项目负责人，我总是教导许多程序员在进行代码测试时，要对其代码进行遍查，而他们总是会吃惊地看着我。这倒不是他们不同意我的看法，而是因为进行代码遍查听起来太费时间了。他们好容易才能赶得上进度，又哪有时间对其代码进行逐条的跟踪呢？幸好这一直观的感受是错误的。是的，对代码进行逐条的跟踪确实需要时间，但它同编写代码相比，只是其一小部分。要知道，当实现一个新函数时，你必须为其设计出函数的外部界面，勾画出相应的算法并把源程序全部输入到计算机中。与此相比，在你第一次运行相应的的程序时，为其设置一个断点，按下“步进”键检查每行的代码又能多花多少时间呢？并不太多，尤其是在习惯成自然之后。这就好比学习驾驶一辆手扳变速器的轿车，一开始好象不可能，但练习了几天以后，当需要变速时你甚至可以无意识地将其完成。同样，一旦逐条地跟踪代码成为习惯之后，我们也会不加思索地设置断点并对整个过程进行跟踪。可以很自然地完成这一过程，并最后检查出错误。

不要等到出了错误再对程序进行逐条的跟踪

代码中的分支

当然有些技术可以使我们更加有效地对代码进行逐条的跟踪。但是如果我们只对部分而不是全部的代码进行逐条跟踪，那么也不会取得特别好的效果。例如，所有的程序员都知道错误处理代码常常有错，其原因是这部分代码极少被测试到，而且除非你专门对这部分代码进行测试，否则这些错误就不会被发现。为了发现错误处理程序中的错误，我们可以建立使错误情况发生的测试用例，或者在对代码进行逐条跟踪时可以对错误的情况进行模拟。后一种方法通常费时较少。例如，考虑下面的代码中断：

```
pbBlock = (byte*)malloc(32);
if( pbBlock == NULL )
{
    处理相应的错误情况;
    ……
}
……
```

通常在逐条跟踪这段代码时，malloc 会分配一个 32 字节的内存块，并返回一个非 NULL 的指针值使其中的错误处理代码被绕过。但为了对该错误处理代码进行测试，可以再次逐条跟踪这段代码并在执行完下行语句之后，立即用跟踪程序命令将 pbBlock 置为 NULL 指针值：

```
pbBlock = (byte*) malloc(32);
```

虽然 malloc 可能分配成功，但将 pbBlock 置为 NULL 指针就相当于 malloc 产生了分配

失败，从而使我们可以步进到相应的错误处理部分。（注意：在改变了 pbBlock 的值之后，malloc 刚分配的的内存块即被丢失，但不要忘了这只是在做测试!）除了要对错误情况进行逐条的跟踪之外，对程序中每一条可能的路径都应该进行逐条的跟踪。程序中具有多条代码路径的明显情况是 if 和 switch 语句，但还有一些其它的情况：&&，||和？：运算符，它们每个都有两条路径。

为了验证程序的正确性，至少要对程序中的每条指令逐条跟踪一遍。在做完了这件事之后，我们对程序中不含错误就有了更高的置信。至少我们知道对于某些输入，相应的程序肯定没错。如果测试用例选择得好，代码的逐条跟踪会使我们受益非浅。

对每一条代码路径进行逐条的跟踪

大的变动怎么样？

过去程序员问过这样的问题：“如果我增加的功能与许多地方的代码都有关系怎么办？那对所有增加的新代码进行逐条的跟踪不是太费时间了吗？”假如你是这么想的，那么我不妨问你另一个问题：“如果你做了这么大的变动，在进行这些改动时可能不引进任何的问题吗？“

习惯于对代码进行逐条跟踪会产生一个有趣的负反馈回路。例如，对代码进行逐条跟踪的程序员很快就会学会编写较小的容易测试的函数，因为对于大函数进行逐条的跟踪非常痛苦。（测试一个 10 页长的的函数比测试 10 个一页长的函数要难得多）程序员还会花更多的时间去考虑如何使必需做的大变动局部化，以便能够更容易地进行相应的测试。这些不正是我们所期望的吗？没有一个项目的负责人喜欢程序员做大的变动，它们会使整个项目太不稳定。也没有一个项目负责人喜欢大的、不好管理的函数，因为它们常常不好维护。

如果发现必须做大的变动，那么要检查相应的改变并进行判断。同时要记住，在大多数情况下，对代码进行逐条跟踪所花的时间要比实现相应代码所花的时间少得多。

数据流 ——— 程序的命脉

在我编写的第 2 章中介绍的快速 memset 函数之前，该函数的形式如下（不含断言）：

```
void* memset( void *pv, byte b, size _tsize )
```

```
{
    byte pb=(byte*)pv;
    if( size >= sizeThreshold )
    {
        unsigned long l;
        /* 用 4 个字节拼成一个长字 */
        l = (b<<24) | (b<<16) | (b<<8) | b;
        pb = (byte*)longfill( (long*)pb, l, size/4 );
        size = size % 4;
    }
    while( size-- > 0 )
        *pb++ = b;
    return(pv);
}
```

这段代码看起来好象正确，其实有个小错误。在我编完了上述代码之后，我把它用到了一个现成的应用程序中，结果没有问题，该函数工作得很好。但为了确信该函数确实起作用了，我在该函数上设置了一个断点并重新运行该应用程序。在进入代码跟踪程序得到了控制之后我检查了该函数的参数：其指针参数值看起来没问题，大小参数亦如此，字节参数值为零。这时我感到使用字节值 0 来测试这个函数真是太不应该，因为它使我很难观察到许多类型的错误，所以我立即把字节参数的值改成了比较奇怪的 0x4E。

我首先测试了 size 小于 sizeThreshold 的情况，那条路径没有问题。随后我测试了 size 大于或等于 sizeThreshold 的情况，本来我想也不会有什么问题。但当我执行了下条语句之后：

```
l = (b<<24) | (b<<16) | (b<<8) | b;
```

我发现 l 被置成了 0x00004E4E，而不是我所期望的值 0x4E4E4E4E。在对该函数进行汇编语言的快速转储之后，我发现了这一错误，并且知道了为什么在有这个错误的情况下该应用程序仍能工作。

我用来编译该函数的编译程序将整数处理为 16 位。在整数为 16 位的情况下，b<<24 会产生什么样的结果呢？结果会是 0。同样 b<<16 所产生的结果也会是 0。虽然这个程序在逻辑上并没有什么错误，但其具体的实现却是错的。之所以该函数在相应应用程序中能够工作，是因为该应用程序使用 memset 来把内存块填写为 0，而 0<<24 则仍是 0，所以结果正确。

我几乎立即就发现了这个错误，因为在把它搁置在一边继续往下走查之前，我又多花了一点时间逐条跟踪了这部分代码。确实，这个错误很严重，最终一定会被发现。但要记住，我们的目标是尽可能早地查出错误。对代码进行逐条跟踪可以帮助我们达到这个目标。

对代码进行逐条跟踪的真正作用是它可以使我们观察到数据在函数中的流动。如果在对代码进行逐条跟踪时密切地注视数据流，就会帮助你查出下面这么多的错误：

- 上溢和下溢错误；

- 数据转换错误；
- 差 1 错误；
- NULL 指针错误；
- 使用废料内存单元错误（0xA3 类错误）；
- 用 = 代替 == 的赋值错误；
- 运算优先级错误；
- 逻辑错误。

如果不注重数据流，我们能发现所有这些错误吗？注重数据流的价值在于它可以使你以另一种非常不同的观点看待你的代码。你也许没能够注意到下面程序中的赋值错误：

```
if( ch = '\t' )
    ExpandTab();
```

但当你对其进行逐条跟踪，密切注视其数据流时，很容易就会发现 ch 的内容被破坏了。

当对代码进行逐条跟踪时，要密切注视数据流

为什么编译程序没有对上述错误发出警告？

在我用来测试本书中程序的五个编译程序中尽管每个编译程序的警告级别都被设置到最大，但仍没有一个编译程序对于 b<<24 这个错误发生警告。这一代码虽然是合法的 ANSI C，但我想象不出在什么情况下这一代码实际能够完成程序员的意图。既然如此，为什么不给出警告呢？

当你遇到这种错误，要告诉相应编译程序的制造商，以使该编译程序的新版本可以对这种错误送出警告。不要低估作为一个花了钱的顾客你手中的权利。

你遗漏了什么东西吗？

使用源级调试程序的一个问题是在执行一行代码时可能会漏掉某些重要的细节。例如，假定在下面的代码中错误地将 && 输入了 & ：

```
/* 如果该符号存在并且它有对应的正文名字，
 * 那么就释放这个名字
 */
if( psym != NULL & psym->strName != NULL )
{
    FreeMemory( psym->strName );
```

```
    psym->strName = NULL;
}
```

这段程序虽然合法但却是错误的。if 语句的使用目的是避免使用 NULL 指针 psym 去引用结构 symbol 的成员 strName，但上面的代码做的却并不是这件事情。相反，不管 psym 的值是否为 NULL 这段程序总会引用 strName 域。

如果使用源级调试程序对代码进行逐条跟踪，并在到达该 if 语句时，按了“步进”键，那么调试程序将把整个 if 语句当做一个操作来执行。如果发现了这个错误，你就会注意到即使在其表达式的左边是 FALSE 的情况下，表达式的右边仍会被执行。（或者，如果你很幸运，当程序间接引用了 NULL 指针时系统会出现错误。但并没有许多的台式计算机会这样做，至少在目前它们不这样做。）

记得我们以前说过：& ，|| 和 ？：运算符都有两条路径，因此要查出错误就必须对每条路径进行逐条的跟踪。源级调试程序的问题是用一个单步就越过了 && 、||和 ？： 的两条路径。有两个实用的方法可以解决这一问题。

第一个方法，只要步进到使用 && 和 || 运算符的复合条件语句，就扫描相应的一些条件，验证这些条件拼写无误然后使用调试程序命令显示条件中每个比较的结果。这样做可以帮助我们查出在某些情况下虽然整个表达式的计算结果正确，但该表达式中确实有错误这一情况。例如，如果你认为在这种情况下 || 表达式的第一部分应该是 TRUE，第二部分应该是 FALSE，但其结果恰恰相反。此时虽然整个表达式的计算结果虽然正确，但表达式中却有错误。观察表达式的各个部分可以发现这类问题。

第二个，也是更彻底的方法是在汇编语言级步进到复合条件语句和 ？：运算符的内部。是的，这要花费更多的工夫，但对于关键的代码，为了观看到中间的计算结果而对其内部的代码实际地走上一遍是很重要的。这同对 C 语句进行逐条的跟踪一样，一旦你习惯之后。对汇编语言指令进行逐条地跟踪也很快，只不过需要经过练习而已。

源级调试程序可能会隐瞒执行的细节
对关键部分的代码要进行汇编指令级的逐条跟踪

关掉优化？

如果所用的编译程序优化功能很强 ,那么对代码进行逐条的跟踪可能会是一个十分有趣的练习。因为编译程序在生成优化的代码时 ,可能会把相邻源语句对应的机器代码混在一块。对于这种编译程序 , 一条“ 单步” 命令跳过三行代码并非不常见 ; 同样 , 利用“ 单步” 指令执行完一行将数据从一处送到另一处的源语句之后却发现相应的数据尚未传送过去的情况也很常见。

为了对代码进行逐条跟踪容易一些，在编译调试版本时可以考虑关掉不必要的编译程序优化。这些优化除了扰乱所生成的机器代码之外，毫无用处。我听到过某些程序员反对关掉编译程序的优化功能他们认为这会在程序的调试版本和交付版本之间产生不必要的差别从而带来风险。如果担心编译程序会产生代码生成错误的话，这种观点还有点道理。但同时我们还应该想到，我们建立调试版本的目的是要查出程序中的错误，既然如此，如果关掉编译的优化功能可以帮助我们做到这点，那么就值得考虑。

最好的办法是对优化过的代码进行逐条的跟踪，先看看这样做的困难有多大，然后为了有效地对代码进行逐条跟踪，只关闭那些你认为必须关闭的编译程序优化功能。

小结

我希望我知道一种能够说服程序员对其代码进行逐条跟踪的方法，或者至少能够使他们尝试一个月。但是我发现，程序员一般说来都克服不了“那太费时间”这一想法。作为项目负责人的一个好处是对于这种事情你可以霸道一些，直到程序员认识到这样做并不费很多时间，并且觉得很值得这样做，因为出错率显著的下降了。

如果你还没有对你的程序进行逐条的跟踪，你会开始这样做吗？只有你自己才知道这个问题的答案。但我猜想当你拿起这本书并开始阅读的时候，准是因为你正被减少你或你领导的程序员的代码中的错误所困扰。这自然就归结为如下的问题：你是宁愿花少量的时间，通过对代码进行逐条的跟踪来验证它；还是宁愿让错误溜进原版源代码中，希望测试者能够注意到这些错误以便你日后对其进行修改。选择在你。

要点：

- 代码中不会自己生出错误来，错误是程序员编写新代码或者修改现有代码的产物。如果你想发现代码中的错误，没有哪个办法比在对代码进行编译时对其进行逐条跟踪更好。
- 虽然直观上你可能认为对代码进行走查会花费大量的时间，但这是不对的。刚开始进行代码的走查确实要多花一点时间，但当这一切习惯成自然之后并不会多花多少时间，你可以很快地走查一遍。
- 一定要对每一条代码路径进行逐条的跟踪，至少要跟踪一遍，尤其是对代码中的错误处理部分。不要忘记 &&、|| 和？：这些运算符，它们每个都有两条代码路径需要进行测试。

- 在某些情况下也许需要在汇编语言级对代码进行逐条的跟踪。尽管不必经常这样做，但在必要的时候不要回避这种做法。

课题：

如果看看第一章中的练习，你就会发现它们所涉及的都是编译程序能够自动为你检查出来的常见错误。重新考查一遍这些练习，这次问问自己：如果使用调试程序对相应的代码进行逐条跟踪，你会漏掉那些错误吗？

课题：

看着六个月以来对你的程序报告出来的错误，确定假如你在编写程序时对其进行了逐条跟踪的话，你会抓住多少个错误。

第 5 章　糖果机界面

Microsoft 雇员从公司得到的一个好处是可以随便享用免费的软饮料，如香味塞尔查矿泉水、牛奶加巧克力和软包装果汁等，管够。但讨厌的是，如果你想吃糖果，就得自己掏腰包。所以有时馋了，我就溜到自动售货机那儿。一次，我塞进几个 25 美分的硬币，然后按下选择键 4 和 5。但当售货机吐出茉莉香味的泡泡糖，而不是我想买的老奶奶牌花生黄油饼干时我愣住了。自然，售货机没错，是我错了，45 号是代表泡泡糖。看一眼售货机上花生黄油饼干的小标记，进一步证实了我的错误。标记上写着花生黄油饼干，21 号，45 美分。

这件事一直使我耿耿于怀，因为假如自动售货机的设计者多花 30 秒钟考虑一下他们的设计，就不会使我以及无数其他人遇到这种事情：买了不想买的东西。如果他们想过：“嗯，人们在向键盘塞钱时常常会想着 45 美分 ——— 我敢打赌，人们在向键盘塞钱时常常会把价钱错当选择号输入给售货机。因此，我们应该选用字母键，不应该使用数字键，以避免这种情况。

这样设计自动售货机并不会增加他的造价，也不会明显改变它的原有设计，但每当在键盘上敲入 45 美分时就会发现机器拒绝接受这种输入，提醒你敲入相应的字母代码。这种设计会引导人们去做正确的事情。

当我们设计函数的界面时，所面临的是相同的问题。不幸的是，程序员不常考虑其他程序员会怎样使用他所设计的函数。就像上面的糖果机界面一样，设计上的细微差别有可能非常容易引起错误，也可能非常容易避免错误。光使设计出的函数没有错误并不够，还必须使它们使用起来很安全。

很自然，getchar()会得到一个 int

标准的 C 库函数以及按照该模式编写的数以千计的其它函数，都有着上述糖果机式界面，容易使用户犯错误。就说 getchar 函数吧，我们有充足的理由说这个函数的界面是有风险的，其中最严重的问题是该函数的设计名鼓励程序员编写有错的代码。关于这一点，还是让我们看看 Brian Kernighan 和 Dennis Ritchie 在其“C 程序设计语言”一书中是怎么说的吧：

考虑以下的代码：

```
char c;
```

```
c = getchar();
if( c == EOF )
    ……
```

在不进行符号扩展的机器上，c 总是正数因为它是 char 类型而 EOF 却是负数，结果上面的测试条件总会失败。为了避免这一点，必须用 int 而不用 char 来保存 getchar 返回值的变量。

这不是说明即使有经验的程序员也必须小心谨慎地使用函数吗？按照 getchar 这样的函数名，将 c 定义成字符类型是很自然的事情，这就是程序员会遇到这个错误的原因。但 getchar 非得如此有害不可吗？该函数要做的工作并不复杂，不过是从某个设备上读入一个字符并返回可能的错误情况。

以下代码给出了另一个常见的问题：

```
/* strdup ——— 为一个字符串建立副本 */
char* strdup( char* str )
{
    char* strNew;
    strNew = (char*)malloc( strlen(str)+1 );
    strcpy( strNew, str );
    return( strNew );
}
```

这个函数在一般的情况下都会工作得很好，除非内存空间耗尽引起了 malloc 的失败。这时，它返回一个不指向任何内存单元的 NULL 指针。但当目的指针 strNew 为 NULL 时，鬼才知道 strcpy 会做些什么。strcpy 不论是失败，还是悄悄地冲掉内存中的信息都不是程序员所期望的。

程序员之所以在使用 getchar 和 malloc 时会遇到麻烦，是因为他们能够写出即使有缺陷但表面上仍能工作的代码。直到几个星期甚至几个月后，才会碰到一连串不易发生的事件而导致这些代码的失败，就像泰坦尼克号邮船沉没的灾难一样。getchar 和 malloc 都不能引导程序员写出正确的代码，都极易使程序员忽视错误情况。

getchar 和 malloc 的问题在于他们返回的值不精确。有时他们返回所要的有效数据，但另一些时候他们却返回不可思议的错误值。

假如 getchar 不返回奇怪的 EOF 值，把 c 声明为字符类型就是正确的，程序员也就不会遇到 Kernighan 和 Ritchie 所说的错误。同样，假如 malloc 不返回好象是内存指针的 NULL，程序员就不会忘记对相应的错误进行处理。问题在于不怕这些函数返回错误，而怕他们把错误隐藏在程序员极易忽视的正常返回值中。

如果我们重新设计 getchar，使他们分别返回两个不同输出怎么样？它可以根据是否成功读入一个新的字符而返回 TRUE 或 FALSE，并把读入的字符返回到一个通过引用传递给他的变量中：

```
flag fGetChar(char* pch);
```

通过这一界面我们可以很自然地写出

```
chat ch;
if( fGetChar( &ch ) )
    ch 中是下一个字符
else
    碰到了 EOF，ch 中是无用信息
```

这样一来，“char 还是 int”的问题就解决了。任何程序员，不管多么幼稚都不太可能偶然忘记测试它的错误返回值，比较一下 getchar 和 fgetchar 的返回值，你看出 getchar 强调的是所返回的字符而 fGetChar 强调的是错误情况吗？如果你的目标是编写出无错的代码，那么你认为应该强调哪一方面？

确实，这样一来在编写代码时就失去了下面的灵活性：

```
putchar( getchar() );
```

但你知道 getchar 的失败频度有多高吗？而几乎在所有的情况下，上面的代码都会产生错误。

一些程序员可能会想：“确实 fGetChar 的界面很安全，但却浪费了代码。因为在调用它时，必须多传一个参数。另外如果程序员没有传递 &ch 而传递了 ch 怎么办？当程序员使用 scanf 函数时，忘记相应的 & ，长期以来一直是一个出错的根源。”

问得好。

编译程序生成代码的好坏其实取决于具体的编译程序，有的编译程序生成稍多的代码，有的稍少，因为我们不必在每次调用该函数之后对函数的返回值和 EOF 进行比较。不管稍多也好稍少也罢，考虑到磁盘和存储器价格的暴跌，同时程序的复杂性及相应的错误率骤增，代码大小上的细微差别也许并不值得顾虑。

在于第二个问题 ——— 比如为 fGetChar 传递了字符而不是字符指针，在采用了第 1 章建议的函数原型之后也用不着担心。如果给 fGetChar 传递了非字符指针的其它参数，编译程序会自动地产生一条错误信息向你指明所犯的错误。

事实上把相互排斥的输出组合到单一返回值中的做法是从汇编语言继承下来的。对于汇编语言来说，只有有限的机器寄存器可以用来处理和传递数据。因此，在汇编语言环境中使用一个寄存器返回两个相互排斥的值既有效率常常又是必需的。然而用 C 编程是另一回事，尽管 C 可以使我们“更接近于机器”，但这并不是说我们应该把它当作高级的汇编语言来使用。

当设计函数的界面时，要选择使程序员第一次就能够写出正确代码的设计。不要使用引起混淆的双重意义的返回值 ——— 每个输出应该只代表一种数据类型，要在设计中显式地体现出这些要点，使用户很难忽视这些重要的细节。

要使用户不容易忽视错误情况
不要在正常地返回值中隐藏错误代码

只再多考虑一下

程序员总知道在什么时候把多个输出组合到单一的返回值中，所以实施上述的建议很容易 ——— 只要不那么做就行了。然而在其它的情况下，程序员设计的界面可能很好，但却象特洛伊木马一样会含有潜在的危险。观察一下改变内存块大小的以下代码：

```
pbBuf = (byte*)realloc( pbBuf, sizeNew );
if( pbBuf != NULL )
    使用初始化这个更大的缓冲区
```

你看出这段程序的错误了吗？如果没看出，也没什么关系 ——— 这个错误虽然很严重，但却很微妙，如果不给出一点暗示很少人会发现它。所以我们给出一个提示：如果 pbBuf 是指向将要改变其大小的内存块的唯一指针，那么当 realloc 的调用失败时会怎样？回答是当 realloc 返回时会把 NULL 填入 pbBuf，冲掉这个指向原有内存块的唯一指针。简而言之，上面的代码会产生内存块丢失的现象。

我们有多少次在要改变一个内存块的大小时，想到要把指向新内存块的指针存储到另一个不同的变量中？我想就象在大街上捡到 25 美分硬币一样，把新指针存储到不同的变量中肯定也很少见。通常人们在改变一个内存块的大小时，会希望仍用原来的变量指向新的内存块，这就是程序员常常掉进陷阱，写出上面代码的原因。

请注意，那些经常把错误值和有效数据混杂在一起返回的程序员，会习惯性地设计出象 realloc 这样的界面。理想情况下，realloc 应该返回一个错误代码，同时不管内存块扩大与否都要再返回一个指向相应内存块的指针。这是两个独立的输出。让我们再看看 fResizeMemory，它是我们在第 3 章中介绍过的 realloc 的外壳函数。去掉了其中的调试代码之后，它的形式如下：

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

上面代码中的 if 语句保证了原有指针绝不会被破坏。如果利用 fResizeMemory 重写本节开始例子中的 realloc 代码，就会得到：

```
if( fResizeMemory(&pbBuf, sizeNew) )
    使用初始化这个更大的缓冲区
```

如果 fResizeMemory 失败，pbBuf 并不会被置为 NULL。它仍会指向原来的内存块，正如我们所期待的那样。所以我们可以问：“使用 fResizeMemory，程序员有可能丢失内存块

吗？”我们还可以问：“程序员有可能会忘记处理 fResizeMemory 的错误情况吗？”

需要说明的另一个有趣问题是：自觉遵循本章给出的第一个建议（“不要在返回值中隐藏错误”）的程序员。永远不会设计出象 realloc 这样的界面。他们一开始就会做出更象 fResizeMemory 这样的设计，因而不会有 realloc 的丢失内存块问题。本书的全部论点都建筑在相互作用的基础上，它们会起到意想不到的效果。这就是一个例证。

然而，将函数的输出分开不总能使我们避免设计出隐藏陷阱的界面，我真希望对此给出一点更好的忠告，但我认为找出这些暗藏陷阱的唯一办法是停下来思考所做的设计。这样做的最佳途径是检查输入和输出的各种可能组合，寻找可能引起问题的副作用。我知道这样做有时非常乏味，但要记住：这比以后再花时间回过来考虑这一问题要划算得多。最坏的情况是略过这一步骤，那么天晓得会有多少个其他的程序员要对设计的不好的界面所引起的错误进行跟踪追击了。只要想一想为了查出由 getchar，malloc 和 realloc 这类界面暗藏陷阱的函数所引起的错误，全世界的程序员要浪费掉多少时间，我们对所有按此模式编写出其他函数简直无话可说。这真是太可怕了！其实只要在设计时多多考虑一点，就可以完全避免这种现象。

要不遗余力地寻找并消除函数界面中的缺陷

单一功能的内存管理程序

虽然在第 3 章我们花了许多时间去讨论 realloc 函数，但并没有涉及到它许多更令人奇怪的方面。如果你抽出 C 运行库手册，查出 realloc 的完整描述你就会发现一些类似于下面的叙述：

```
void* realloc( void* pv, size_t size );
```

realloc 改变先前已分配的内存块的大小，该内存块的原有内容从该块的开始位置到新块和老块长度的最小长度之间得到保留。

- 如果该内存块的新长度小于老长度，realloc 释放该块尾部不再想要的内存空间，返回的 pv 不变。
- 如果该内存块的新长度大于老长度，扩大后的内存块有可能被分配到新的地址处，该块的原有内容被拷贝到新的位置。返回的指针指向扩大后的内存块，并且该块扩大部分的内容未经初始化。
- 如果满足不了扩大内存块的请求，realloc 返回 NULL，当缩小内存块时，realloc 总会成功。
- 如果 pv 为 NULL，那么 realloc 的作用相当于调用 malloc(size)，并返回指向新分配内存块的指针，或者在该请求无法满足时返回 NULL。
- 如果 pv 不是 NULL，但新的块长为零，那么 realloc 的作用相当于调用 free(pv) 并且总是返回 NULL。
- 如果 pv 为 NULL 且当前的内存块长为零，结果无定义

哎呀！realloc 真是一个实现得“面面俱到”的最好例子，它在一个函数中完成了所有的内存管理工作。既然如此还要 malloc 干什么？还要 free 干什么？realloc 全包了。

有几个很好的理由说明我们不应该这样设计函数。首先，这样的函数怎么能指望程序员可以安全地使用呢？它包括了如此之多的细节，甚至有经验的程序员都不全知道。如果你对此有疑问，不妨调查一下，算算有多少程序员知道给 realloc 传递一个 NULL 指针相当于调用了 malloc；又有多少程序员知道给 realloc 传递一个为零的块长效果与调用 free 相同。确实，这些功能都相当隐秘，所以我们可以问他们要避免错误就必须知道的一些问题，如当调用 realloc 扩大一个内存块时会发生什么事情，或者他们是否知道此时相应的内存块可能会被移动？

realloc 的另一个问题是：我们知道传递给 realloc 的可能是无用信息，但是因为其定义如此通用使它很难防范无效的参数。如果错误地给它传递了 NULL 指针，合法；如果错误地给它传递了为零的块长也合法。更糟的是本想改变内存块的大小，却 malloc 了一个新块或 free 掉了当前的内存块。如果实际上任何参数都合法，那么我们怎样用断言检查 realloc 参数的有效性呢？不管你提供了什么样的参数，realloc 全能处理，甚至在极端的情况下也是如此。一个极端是它 free 内存块，另一个极端是它 malloc 内存块。这是截然相反的两种功能。

公平地说，程序员通常不会坐下来思考：“我打算在一个函数中设计一个完整的子系统。”象 realloc 这样的函数几乎总是产生于两个原因：一个是其多种功能是逐步演变而来的；另一个是具体的实现为其增加了多余的功能（如 free 和 malloc），为了包括这些所谓的“幸运”功能，实现该函数的程序员扩展了相应的形式描述。

不管出于什么样的理由编写了多功能的函数，都要把它分解为不同的功能。对于 realloc 来说，就是要分解出扩大内存块、缩小内存块、分配内存块和释放内存块。把 realloc 分解为四个不同的函数，我们就能使错误检查的效果更好。例如，如果要缩小内存块，我们知道相应的指针必须指向一个有效的内存块，而且新的块长必须小于（也可以等于）当前的块长。除此之外任何东西都是错误的。利用单独的 ShrinkMemory 函数我们可以通过断言来验证这些参数。

在某些情况下我们实际也许希望一个函数做多个事情。例如当调用 realloc 时，通常我们知道新的块长是大于还是小于当前的块长？这要取决于具体的程序，但我通常不知道（尽管我常常能够推算出这一信息）。对我来说，最好是有一个函数既能扩大内存块，又能缩小内存块。这样可以避免在每次需要改变内存块大小时，必须写出 if 语句。这样虽说放弃了对某些多余参数的检查，但可以得到不再需要写多个 if 语句（可能会搞乱程序）的补偿。既然我们总是知道什么时候要分配内存，什么时候要释放内存，所以应该把这些功能从 realloc 中割裂出来，使它们构成单独的函数。第 3 章介绍的 fNewMemory，FreeMemory 和 fResizeMemroy 就是这样三个定义良好的函数。

但是假如我正在编一个通常确实知道是要扩大还是缩小内存块的程序，那我一定会把 realloc 的扩大内存块和缩小内存块功能分解出来，再建立两个新的函数：

```
flag fGrowMemory(void** ppv, size_t sizeLarger);
```

```
void ShrinkMemory(void* pv, size_t sizeSmaller);
```

这样不仅可以使我能够对输入的指针和块长参数进行彻底的确认，而且调用 ShrinkMemory 的风险也小，因为它保证相应的内存块总是被缩小而且绝对不会被移动。所以不用写：

```
ASSERT( sizeNew <= sizeofBlock(pb) );    //确认 pb 和 sizeNew
(void)realloc(pb, sizeNew);              //设缩小不会失败
```

只写：

```
ShrinkMemory( pb, sizeNew );
```

就可以完成相应的确认.使用ShrinkMemory代替realloc的最简单理由是这样做会使相应的代码显得格外清晰。使用了 ShrinkMemory，就不再需要用注解说明它可能失败，不再需要用 void 的类型转换去掉返回值中无用的部分，也不再需要用验证 pb 和 sizeNew 的有效性，因为 ShrinkMemory 会为我们做这一切。但是如果使用 reallo，我甚至认为还应该使用断言检查他返回的指针是否与 pb 完全相同。

不要编写多种功能集于一身的函数
为了对参数进行更强的确认，要编写功能单一的函数

模棱两可的输入

前面我们谈过为了避免使程序员产生混淆，应该把函数的各种输出明确地分别列出。如果把这一建议也应用于函数的输入，自然就可以避免写出象 realloc 这样包罗万象的函数。realloc 输入一个内存块指针参数，但有时却可以取不可思议的 NULL 值，结果使它成了 malloc 的仿造物。realloc 还有一个块长参数，但却可以取不可思议的零值，结果使它成了 free 的仿造物。这些不可思议的参数值看起来好象没有什么害处，其实损害了程序的可理解性。我们可以看一下，下面的代码究竟是改变内存块的大小，还是分配或者释放内存块呢？

```
pbNew = realloc( pb, size );
```

我们对此一无所知，它们都有可能，这完全取决于 pb 和 size 的取值。但是假如我们知道 pb 的指向的是一个有效的内存块，size 是个合法的块长，立刻就知道它是改变内存块的大小。正象明确的输出使人容易搞清函数的结果一样，明确的输入亦使人容易理解函数要做的事情，它对必须阅读和理解别人程序的维护人员极有价值。

有时模棱两可的输入并不象在 realloc 情况下那么容易发现。让我们来看看下面的专用字符串拷贝例程。它从 strFrom 开始取 size 个字符，并把它们存储到从 strTo 开始的字符串中：

```
char* CopySubStr( char* strTo, char* strFrom, size_t size )
{
    char* strStart = strTo;
    while(size-- > 0)
```

```
        strTo++ = strFrom++;
    *strTo= '\0' ;
    return(strStart);
}
```

CopySubStr 类似于标准的函数 strcpy，所不同的是它保证起始于 strTo 的字符串确定是个以零结尾的 C 字符串。该函数的典型用法是从大字符串中抽取子串。例如从一个组合串中抽出星期几：

```
static char* strDayNames = "SunMonTueWedThuFriSat";
……
ASSERT(day>=0 && day<=6);
CopySubStr(strDay, strDayNames+day*3, 3);
```

现在我们明白了 CopySubStr 的工作方式，但你看得出该函数的输入有问题吗？只要你试着为该函数写断言去确认它的参数，就很容易发现这一问题。参数 strTo 和 strFrom 的断言可以是：

```
ASSERT( strTo != NULL && strFrom != NULL );
```

但我们怎样确认 size 参数呢？size 为零合法吗？size 大于 strFrom 的长度怎么办？如果查看该函数的实现，我们就会看到这两种情况都可以得到处理。如果在进入该函数时 size 等于零，while 循环就不会执行；如果 size 大于 strFrom，while 循环将把 strFrom 整个连同其终止符一道拷贝到 strTo 中。为了说明这点，必需在函数的注解中加以说明：

```
/* CopySubStr ——— 从字符串中抽取子串
 * 把 strFrom 的前 size 个字符转储到从 strTo
 * 开始的字符串中。如果 strFtom 中的字符数小
 * 于"size",那么 strFrom 中的所有字符都被拷
 * 贝到 strTo。如果 size 等于零，strTo 被设
 * 置成空字符串.
 */
char* CopySubStr(char* strTo, char* strFrom, size_t size)
{
    ……
```

听起来好象很熟悉，不是吗？确实如此，类似的函数就象灯泡上的灰尘一样司空见惯。但这是处理其 size 输入参数的最好方式吗？回答是"不"，至少从编写无错代码的观点来看是"不"。

例如，假定程序员在调用 CopySubStr 时错把"3"输成了"33"：

```
CopySubStr( strDay, strDayNames+day*3, 33 );
```

这确实是个错误，但根据 CopySubStr 的定义用 33 调用它却完全合法。是的，在交出相应的代码之前或许也可能抓住这个错误，但却没法自动地发现它，必须由人查出它。不要忘了从靠近错误的断言开始查错，要比从错误的输出开始查错速度更快。

从“无错”的观点，如果函数的参数越界或者无意义，那么即使能被智能地处理，仍然应该被视为非法的输入。因为悄悄地接受奇怪的输入值，会隐藏而不是暴露错误。在某种意义上，防错性程序设计应该允许“无拘无束”的输入。为了提高程序的健壮性，要在代码中包括相应的防错代码，而不是禁止有问题的输入：

```
/* CopySubStr ——— 从字符串中抽取子串
 * 把 strFrom 的前“size”个字符转储到从 strTo
 * 开始的字符串中，在 strFrom 中，至少必须要
 * 有“size”个字符。
 */
char* CopySubStr(char strTo, charstrFrom, size_t size)
{
    char* strStart = strTo;
    ASSERT( strTo != NULL && strFrom != NULL );
    ASSERT( size <= strlen(strFrom) );
    while( size-- > 0 )
        strTo++ = strFrom++;
    *strTo= ‘\0’ ;
    reurn( strStart );
}
```

有时允许函数接受无意义的参数 ——— 如大小为 0 的参数，是值得的，因为这样可以免除在调用时进行不必要测试。例如，因为 memset 允许其 size 参数为零，所以下面程序中的 if 语句是不必要的：

```
if( strlen != 0 )    /* 用空格填充 str */
    memset( str, chSpace, strlen(str) );
```

在允许大小为 0 的参数时要特别小心。程序员处理大小（或计数）为 0 参数通常是因为他们能够处理而不是应该处理。如果所编函数有大小参数，那么并不一定非得对大小为 0 进行处理，而要问自己：“程序员用大小为 0 的参数调用这个函数的额度是多少？”如果根本或者几乎不会这么调用，那就不要对大小为 0 进行处理，而要加上相应的断言。要记住，消除限制就是消除捕获相应错误的机会，所以一个良好的准则是，一开始就要为函数的输入选择严格的定义，并最大限度地利用断言。这样，如果过后发现某个限制过于苛刻，可以把它去掉而不至于影响到程序的其它部分。

第 3 章在 FreeMemory 中包含的 NULL 指针检查，用到的就是这一原理。因为我从来不会用 NULL 指针调用 FreeMemory，所以对我来说加强对这一错误的检查就十分重要。对此可能会有不同的看法。这里并没有对错之分，但要保证所做的是自觉的选择，而不仅仅是一种随便的习惯。

不要模棱两可，要明确地定义函数的参数

现在不要让我失败

Microsoft 公司招募雇员的政策，是在面试时就一些技术问题向候选者提问。对于程序员来说，就是给出一些编程问题。我过去常常从要求编写标准的 tolower 函数开始考核候选者。我递给候选者一个 ASCII 表，问候选者“怎样写一个函数把一个大写字母转换成对应的小写字母？”我有意对如何处理字母以外的其它符号和小写字母说得很含糊，主要是想看看他们会怎样处理这些情况。这些符号在返回时会保持不变吗？会用断言对这些符号进行检查吗？它们会不会被忽视？半数以上的程序员写出的函数会是下面这样：

```
char tolower(char ch)
{
    return( ch + ‘a’ - ‘A’ );
}
```

这种写法在 ch 是大写字母的情况下没问题，但如果 ch 是其他的符号就会出毛病。当我向候选者指出这一情况时，有时他们会说：“我假定 ch 必须是大写字母。如果它不是大写字母我可以将其不变地返回。”这种解法很合理，但其它的解法就未必。更常见的是那些未中选的候选者会说：“我没有考虑到这个问题。我可以解决这个问题，当 ch 不是大写字母时，令它返回一个错误代码。”有时他们会使 tolower 返回 NULL，有时会返回空字符。但出于某种原因，无疑-1 会占上风：

```
char tolower(char ch)
{
    if( ch >= ‘A’ && ch <= ‘Z’ )
        return( ch + ‘a’ - ‘A’ );
    else
        return(-1);
}
```

这些解法都违背了我们前面给出的建议，因为他们把出错值同真正的数据混在了一起。但真正的问题并不在于候选者没能注意到他们也许从未听说过的建议，而是他们在大可不必的情况下返回了错误代码。

这提出了另一个问题：如果函数返回错误代码，那么该函数的每个调用者都必须对该错误进行处理。如果 tolower 可能返回-1，那么就不能简单地这么写：

```
ch = tolower(ch);
```

而必须这么写：

```
int chNew;      /* 为了容纳-1，它必须是 int 类型 */
if( (chNew=tolower(ch)) != -1 )
    ch = chNew;
```

这一点与上一节有关。如果你意识到在每次调用时都必须这样使用 tolower 就会明白让

它返回一个错误代码也许并不是定义这个函数的最佳方式。

如果发现自己在设计函数时要返回一个错误代码，那么要先停下来问自己：是否还有其它的设计方法可以不用返回该错误情况，因此，不要将 tolower 定义成返回大写字母对应的小写字母，而要使其“如果 ch 是大写字母，就返回它对应的小写字母；否则，将其不改变地返回。”

如果发现无法消除错误的情况，那么可以考虑干脆不允许这些有问题的情况出现，即用断言对函数的输入进行验证。如果把这一建议应用于 tolower。就会得到：

```
char tolower(char ch)
{
    ASSERT( ch >= ‘A’ && ch <= ‘Z’ );
        return( ch + ‘a’ - ‘A’ );
}
```

这两种方法都可以使函数的调用者不必进行运行时的错误核查，这意味着产生的代码更小并且错误更少。

编写函数使其在给定有效的输入情况下不会失败

看出言外之意

站在调用者的立场上，我并没有过分强调检查所设计的函数界面有多么重要。考虑到函数只定义一次，但在程序中的许多地方都要调用它，就会明白不检查函数的调用方式是很愚蠢的。我们见过的 getchar，realloc 和蹩脚的 tolower 例子都说明了这一点，它们都导致了相应调用代码的复杂化。然而，并非只有把输出都合在一起和返回不必要的错误代码才会导致复杂的代码。有时引起代码复杂化的原因完全由于粗心而忽视了相应的函数调用“读’的效果。

例如假定在改进所编应用程序的磁盘处理部分时，碰到了一个写成下面这样的文件搜索调用：

```
if( fseek(fpDocument, offset, 1) == 0 )
```

你可以说得出它将进行某种搜索，也可以看到相应的错误情况得到了处理，但这个调用的可读程度究竟如何呢？它进行的是哪种类型的搜索（从文件开始位置、从文件的当前位置、还是从文件的结束位置开始搜索）？如果该调用返回 0 值，这究竟表明的是成功还是失败？

反过来，如果程序员使用预定义的名字来进行相应的调用；

```
#include <stdio.h>        /* 引入 SEEK_CUR 的定义 */
#define ERR_NONE 0
……
if( fseek(fpDocument, offset, SEEK_CUR) == ERR_NONE )
    ……
```

这样不是使相应的调用更清晰吗了？确实如此。但这并不是使人感到惊奇的新鲜事，程序员在几十年前就已经知道应该在程序中避免使用莫名其妙的数字。有名的常量不仅可以使代码更可读，而且使代码更可移植（考虑到在其他的系统上，SEEK_CUR 可能不是 1）。

我要指出的是，虽然许多程序员把 NULL、TRUE 和 FALSE 当作有名的常量来使用。但它们并不是有名的常量，只不过是莫明其妙数字的一种正文表示。例如，下面的调用完成什么工作？

```
UnsignedToStr(u, str, TRUE);
UnsignedToStr(u, str, FALSE);
```

你可能会猜出这些调用是用来将一个无符号的值转换成其正文表示。但上面的布尔参数对这一转换起什么作用呢？如果我把这些调用写成下面这样，是不是会更清楚一些：

```
#define BASE10 1
#define BASE16 0
.....
UnsignedToStr(u, str, BASE10);
UnsignedToStr(u, str, BASE16);
```

当程序员坐下来编写这种函数时，其布尔参数值似乎非常清楚。程序员先做函数描述，然后做函数的实现：

```
/* UnsignedToStr
 * 这一函数将一个无符号的值转换成其对应
 * 的正文表示，如果 fDecimal 为 TRUE，u 被转
 * 换成十进制表示；否则，它被转换成
 * 十六进制表示。
 */
void UnsignedToStr(unsigned u, char *strResult, flag fDecimal)
{
    .....
```

还有什么比这更清楚的吗？

但事实上，布尔参数常常表明设计者对其设计并没有深思熟虑。相应的函数可以做两种不同的事情，用布尔参数来选择想要做的事情；也可以很灵活地不只限于两种不同的功能，但程序员使用布尔值来指明唯一感兴趣的两种情况。这两种可能常常都正确。

例如，如果我们把 UnsignedToStr 看作一个只做两种不同事情的函数，就应该把它拆成下面两个函数：

```
void UnsignedToDecStr(unsigned u, char* str);
void UnsignedToHexStr(unsigned u, char* str);
```

但这种情况下，一种更好的解决办法是把它的布尔参数改成通用的参数，从而使 UnsignedToStr 更加灵活。这样可以使程序员不是传递 TRUE 或 FALSE，而是相应的转换基数：

```
void UnsignedToStr(unsigned u, char* str, unsigned base);
```

这样我们可以得到清晰的灵活设计，它使相应的调用代码容易理解，同时还增加了该函数的功能。

这一建议似乎与我们早先说过的“要严格地定义函数的参数”互相矛盾 ——— 我们把具体的 TRUE 或 FALSE 输入变成了一般的输入，函数的大部分可能取值都没有用到。但要记住，虽然参数变得一般了，但我们总是可以在函数中包括断言来检查 base 的取值永远只能是 10 或者 16。这样如果以后决定还需要进行二进制或者八进制的转换，可以放松这一断言以便程序员传递等于 2 和 8 的基数值。

比起我所见过那些参数取值是 TRUE、FALSE、2 和-1 的函数，这种做法要好得多。因为布尔参数的值域不容易扩充，所以要么你得继续忍受这些无意义的参数值，要么就得修改现有的每个调用语句。

使程序在调用点明了易懂；要避免布尔参数

向人们提示险情

作为防范错误的最后一个措施，我们可以在函数中写上相应的注解来强调它可能产生的险情，并给出函数的正确使用方式，这样可以帮助其他的程序员在使用该函数时不致出错。例如，getchar 的注解不应该这样：

```
/* getchar ——— 该函数与 getc(stdin)相同 */
int getchar(void)
……
```

它对程序员真的起不到什么帮助作用，我们应该把它写成：

```
/* getchar ——— 等价于 getc(stdin)
 * getchar 从 stdin 返回了一个字符，当发生了
 * 错误时，它返回“int”EOF。该函数的一种
 * 典型用法是：
 *      int ch;     // 为了容纳 EOF，ch 必须是 int 类型
 *      if( (ch=getchar()) != EOF )
 *          成功 ——— ch 是下一个字符
 *      else
 *          失败 ——— ferror(stdin)将给出错误的类型
 */
int getchar(void)
……
```

如果把这两种描述都交给初学 C 库函数的程序员，你认为对于使用 getchar 时会出现的险情哪种描述会给程序员留下比较深的印象？当程序员第一次使用 getchar 时，这两种描述会产生什么样的差别？你认为他会编写新的代码，还是会从你做的注解中复制下典型用法给

出的例子，然后再根据需要对其进行修改？

按照这种方式对函数进行注解的另一个积极作用，是它可以迫使不够谨慎的程序员停下来考虑别的程序员怎样才能使用他们编出的函数。如果程序员设计的函数界面很笨，在编写典型用法时，他就应该注意到界面的笨拙。即使他没有注意到界面的问题，只要典型用法给出的例子详尽正确，也没有什么关系。例如，倘若 realloc 被注解成如下形式，就不会引起那么多的使用问题了：

```
/* realloc( pv, size )
 * ……
 * 该函数的一种典型用法是.
 *      void* pvNew;// 用来保护 pv，以防 realloc 失败
 *      pvNew = realloc( pv, sizeNew );
 *      if( pvNew != NULL )
 *      {
 *          成功 ——— 修改 pv
 *          pv = pvNew;
 *      }
 *      else
 *          失败 ——— 不要用值为 NULL 的 pvNew 冲掉 pv
 */
void realloc( void* pv, size_t size )
```

通过复制这样的示例，即使不够谨慎的程序员也很可能会避免本章开始所讲的内存丢失问题。例子虽然并不能对所有程序员都起作用，但就象药品包装上的警告信息一样，它会对某些人产生影响。而且从任何一点看，这样做都有所帮助。

然而不要用例子来代替编写良好的界面。 getchar 和 realloc 的界面都使用户容易出错，这些害处都应该予以消除而不仅仅是给予说明。

编写注解突出可能的异常情况

小结

设计能够抵御错误的界面并不困难，但这确实需要多加考虑并且愿意放弃根深蒂固的编码习惯。这一章给出的建议只需简单地改变函数的界面，就可以使程序员编写出正确的代码，而不必过多地考虑其它部分的代码。本章贯穿始终的关键概念是“尽可能地使一切清晰明了”。如果程序员理解并记住了每个细节，也许就不会犯错误 ——— 他们之所以会犯错误，是因为他们忘记了或者从来就不知道这些重要的内容。因此要设计能够抵御错误的界面，使程序员很难无意地忽视相应的细节。

要点：

- 最容易使用和理解的函数界面，是其中每个输入和输出参数都只代表一种类型数据的界面。把错误值和其它的专用值混在函数的输入和输出参数中，只会搞乱函数的界面。
- 设计函数的界面迫使程序员考虑所有重要细节（如错误情况的处理），不要使程序员能够很容易地忽视或者忘记有关的细节。
- 老要想到程序员调用所编函数的方式，找出可能使程序员无意间引入错误的界面缺陷。尤其重要的是要争取编出永远成功的函数，使调用者不必进行相应的错误处理。
- 为了增加程序的可理解性从而减少错误，要保证所编函数的调用能够被必须阅读这些调用的程序员所理解。莫明其妙的数字和布尔参数都与这一目标背道而驰，因此应该予以消除。
- 分解多功能的函数。取更专门的函数名（如 ShrinkMemory 而不是 realloc）不仅可以增进人们对程序的理解，而且使我们可以采用更加严格的断言自动地检查出调用错误。
- 为了向程序员展示出所编函数的适当调用方法，要在函数的界面中通过注解的方式详细说明。要强调危险的方面。

练习：

1) 本章开始的函数 strdup 为一个字符串分配一个副本，但如果它失败了则返回 NULL。更能抵御错误的 strdup 界面是什么？
2) 我说过布尔输人参数的存在，常常表示可能还有更好的函数界面。但对于布尔输出参数怎样呢？例如，如果 fGetChar 失败，它返回 FALSE 并要求程序员调用 ferror(stdin)来确定出错的原因，那么更好的 getchar 界面会是什么？
3) 为什么 ANSI 的 strncpy 函数必然会使轻率的程序员犯错误？
4) 如果读者熟悉 C++的 inline 指明符，说说它对于编写能够抵御错误的函数界面的价值。
5) C++采用了类似于 pascal 中 VAR 参数的 & 引用参数。因此，不是这样写：

```
flag fGetChar(char* pch);    /* 原型 */
……
if( fGetChar(&ch) )
    ch 含有新的字符 ……
可以写:
flag fGetChar(char &ch);/* 原型 */
……
if( fGetChar(ch) )           /* 自动的传递 &ch */
    ch 含有新的字符 ……
```

从表面上看，这一加强似乎不错，因为程序员不可能“忘记”正规 C 中要求的显式&。但为什么使用这一特征会产生容易出错的界面，而不是能够抵御错误的界面？

6) 标准的 strCmp 函数取两个字符串并对它们进行逐字符的比较。如果这两个字符串相等，strcmp 返回 0；如果第一个字符串小于第二个，它返回负数；如果第一个字符串大于第二个，它返回正数。因此当调用 strcmp 时，相应的代码通常有下面的形式：

```
if( strcmp(str1,str2) rel_op 0 )
    ……
```

这里 rel_op 是 == 、 != 、> 、>= 、< 或 <= 之一，尽管这样也可以完成相应的比较，但对于不熟悉 strcmp 函数的人来说它毫无意义。为字符串比较设计至少两个其它的函数界面，所设计的界面应该更能抵御错误，更加可读。

课题：

检查一个标准的 C 库函数对相应的界面重新设计使其更能抵御错误。为了使重新设计的函数更加明了易懂，给这些函数改变名字的利弊是什么？

课题：

在大量的代码中搜寻所使用的 memset、memmove、memcpy 和 strn 系列函数（如 strncpy 等）。在所找到的调用中，有多少个要求对应的函数接受为零的计数值？所得到的这种便利足以说明允许函数接受零计数值是合理的吗？

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 标准的编泽程序可能具有 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 singned 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，则返问指向该字符的指针，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 键而设立的子窗口，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 */
```

```
strcut 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格针 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，可是在代码宝间内的代价巨大。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。常识告诉我们如果有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。如果移动对话窗口， 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，就可以说：“ 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，有一些场会需要合法使用 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 AddChlld 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 ppwndNext 指向 pwndParent -> pwndChild
     * 因为 pwndParent -> pwndChild 是链中第一个“下一个兄弟指针”
     */
    ppwndNext = &pwndParent->pwndChild;
    while( *ppwndNext != NULL )
        ppwndNext = &( *ppwndNext )->pwndSibling;
    *ppwndNext = pwndNewBorn;
}
```

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。

```
usigned 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导求替代算法上，而不是花在以某个稍微不同的方式实现同一个算法上，那么可能就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 )      /* 不要忘掉负号 */
        *pch++ = '-';
    *pch = '\0';
    ReverseStr(str);     /* 将子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措况。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，循环将永运执行下去，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~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用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( &pwnd->strWndTitle, strlen(strNewTitle)+1 ) )
    strcpy( pwnd->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( pwnd -> 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 *pwnd, char *strNewTitle )
{
    ASSERT( fValidWindow(pwnd) );
    ASSERT( strNewTitle != NULL );
    ASSERT( fValidPointer( pwnd->strWndTitle, strlen(strNewTitle) - 1 ) );
    strcpy( pwnd->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为编写在空间、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。
- 曾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，因为这样的实现对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利，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

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。

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近的时候，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(*pcd == 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

拿车钥匙的贼还是赋

很奇怪有些程序员，他们从不引用不属于地们自己的存储空间。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->pwndSilbing;
```

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教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——一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 strFrornUns 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踏螺丝刀，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( (*sdtr != '+') || 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，该函数是否更好理够呢？

```
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，说明你训练有索，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，为什么说是自找麻烦呢？

```
        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
```

第 8 章　剩下来的就是态度问题

本书中讨论的方法可以用来检查错误和防止错误，但是这些技术并不能保证肯定可以写出无错代码，就象一个熟练的球队不可能是常胜军一样。重要的是养成好的习惯和正确的态度。

如果一个球队成天在嘴上讨论如何训练，这个球队可能有取胜的机会吗？如果这个球队的队员不断地因为工资低而牢骚满腹，或时刻耽心被换下场或裁减掉，那又会怎么样呢？虽然这些问题与球赛没有直接关系，但是却影响了球员水平的发挥。

同样读音可以使用本书的所有建议，但是，如果你持疑虑的态度或者使用错误的编码习惯，那么要写出无错的代码将是很困难的。因此，你要有必胜的信心和良好的习惯，同样，你同级的同事如果没有必胜信心和良好习惯也会遇到同样的问题。

因此在本章中将指出一些编写无错代码的主要障碍。只要能意识到这些障碍，改正就很容易了。

错误不出现，我还有一招

当向程序员询问有关他们修改错误的情况时，有多少次听到他们这样的回答：“唉呀！错误消失了。”多年以前，我就曾经向我的第一个经理说过这样的话，当时,我们正在研制 Apple Ⅱ数据库产品，经理问我若已经设法找到错误项目能否就此收尾，我说：“唉呀！错误消失了。”经理停顿了片刻，然后邀请我到他的办公室坐坐。

“你说‘错误消失了’，是什么意思？”

“哎呀！你知道，我一步步地仔细查看了错误报告。错误没再出现。”

经理在椅子上向后仰了一下问：“你认为错误到哪儿去了？”

“我不知道。”我说，“我想它已被改正了吧。”

“你并不知道谁改的，是吧？”

“是的，我是不知道。”我坦诚地回答。

“好，那你不认为你应该查明到底真正发生了什么吗？”他说。“毕竟你是在和计算机打交道，错误不会自我改正。”

然后，经理进一步解释说，错误消失有三个原因：一是错误报告不对；二是错误已被别的程序员改正了；三是这个错误依然存在但没有表现出来。也就是说，作为一个专业程序员，其职责之一就是要确定错误的消失究竟属于以上三种情况中的哪一种，从而采取相应的行动，但是决不能因为错误不出现就简单地忽略了它，就万事大吉了。

在我第一次听到这个忠告的时候，也就是在 CP/M 和 Apple Ⅱ的时代，这个忠告很有价值。实际上，在这之前的几十年里，它就很有价值，而且至今它仍然很有价值。但是，直到我自己成为项目负责人，并且发现程序员普遍乐于接受测试员搞错了或有某个程序员已经为其排除了这个错误，这时我才认识到这个忠告多么有意义。

错误消失经常是因为程序员和测试员使用了不同的程序版本。如果在程序员使用的代码中错误没有出现，就采用测试员使用的程序版本，如果错误仍未出现，就可通知测试组。但是，如果错误确实出现了，就要追踪到它早些的源程序版本，并决定如何修改它，然后再查看一下为什么在当前的源程序版本中，错误会不见了。通常错误仍然存在，只是环境有了更改从而掩盖了错误。无论什么原因，为了采取适宜的步骤来改正错误，必须弄明白为什么错误消失了。

错误几乎不会“消失”

浪费精力？

当我要求程序员在老版本源程序上寻找所报告的错误时，他们经常要发牢骚，这样做似乎象是浪费时间。要是你也这么认为的话，你要明白这样做并不是要你恢复老版本源程序，而不过要你查看一下这些源程序，为你提供查错的良机，而且这也是找到错误最有效的方法。

即使你发现错误已被改正了，那些老版本源代码中将错误分离出来也是值得的。是将错误以“改正了”终结好呢？还是给错误标以“不会再产生了”并送还给测试组好呢？测试人员将怎样做呢？他们肯定不会认为这个错误已经更正了，他们只有两种选择，一种是花时间来试图提出另一组能再产生错误的用例；另一种是丢下这个错误，将其标以“不会再产生了”

并希望错误已被改正。与在老版本源代码中找到了错误并以“ 改正了” 而终结比较起来。后两种选择都不好。

及时纠正，事半功倍

我第一次参加 Excel 小组的时候，我们把所有的错误改正工作都推迟到项目的最后。这并不是说有人用枪顶着我们的脊骨说：“直到所有的特征都实现了再去改正错误”,但总有保持进度和实现特征的压力，而在修改错误方面却一点压力也没有。我曾经说过：“除非错误使系统瘫痪了或使测试组停工，否则别急着更改它，完成进度要求的各特征之后，我们有的是时间来修改错误。”简而言之，改正错误没放在优先地位。

我相信现在的 Microsoft 程序员听到上面所讲的肯定感到很逆耳,因为项目不再以这种方式研制了。这种方法存在的问题太多，最坏的是不能预言产品什么时候能够完成。你怎样估计修改 1742 个错误的时间？当然，不仅仅有 1742 个错误需要修改，因为在程序员修改旧错误时又会引起新错误。更密切相关的是，修改一个错误可能暴露其它的潜在错误，由于第一个错误的障碍，测试组未能发现这些潜在错误。

但这还不是唯一的问题。

由于没有改正错误就完成了所要求的特征，产品看上去比它实际进展情况要提前了许多。公司的重要人物测试使用内部发行版本，发现除了一些偶然的错误之外，产品工作得很好，他们很惊奇，只用了六个月的开发时间就几乎完成了一个最终产品。他们看不到存储空间溢出错，或某些从未试用过的特征错，他们只知道代码“各特征齐全”，基本上可以工作。

到最后用几个月的时间来修改错误也往往士气不振。程序员喜欢编程序而不愿意改错，但是在每个项目的最后，有好几个月的时间，他们除了改错无事可作。由于开发组以外的每个人都明显地知道代码已接近完成，因此改错经常具有很大的压力。

这不是自找吗？

然而，自打 Macintosh Excel 1.03 开始到撤消 ——— 未宣布名字的窗口产品（由于失控的错误表造成的）为止，Microsoft 一直运行带有错误的产品，这就迫使 Microsoft 认真研究怎样开发产品。得到的结论并不使人感到惊奇：

- 不要通过把改正错误移置产品开发周期的最后阶段来节省时间。修改一年前写的代码比修改几天前写的代码更难，实际上这是浪费时间。
- “一次性”地修改错误会带来许多问题，因为发现错误越早，重复这个错误的可能性就越小。
- 错误是一种负反馈，程序开发倒是快了，却使程序员疏于检查。如果规定只有把错误全部改正之后才能增加新特征的话,那么在整个产品开发期间都可以避免程序员的疏漏，他们将忙于修改错误。反之，如果允许程序员略过错误，那就使管理失控。
- 若把错误数保持在近乎于 0 的数量上，就可以很容易地预言产品的完成时间。只需要估算一下完成 32 个特征所需的时间,而不需要估算完成 32 个特征加上改正 1742

以上这些观点并不只适用于 Microsoft 开发，而且适用于任何软件开发。如果你在发现错误时没有及时纠正，那么 Microsoft 的坏经验就是你的反面教材。你可以从自己的艰难经历中或从别人的沉痛教训中学到很多东西。到底该怎样做呢？

马上修改错误，不要推迟到最后

憨医救人

在安东尼·罗宾斯的小说《唤醒巨人》（Awaken the Giant Within）中讲了一位医生的故事。一天，有个医生走到一条汹涌的河边，她突然听到落水者的呼救声。她环顾了四周，发现没有人去救，于是，她就跳入水中，朝着落水者游去。她将落水者救上岸，做口对口的人工呼吸，这个人刚一恢复呼吸，又从河里传来了另外两个落水者的求救声。她又一次跳入水中，把这两个人救上岸，正当她安顿好这两个人时，医生又听到另外四个落水者的求救声，然后她又听到另外八个落水者的求救声 …… 问题是医生只忙于救人，抽不出时间到上游查明是谁把人们扔到水中。

象这个医生一样，程序员也经常忙于“治愈”错误而没有停下来判断一下是什么原因引起了这些错误。例如象上一章我们讲过的函数 strFromUns，由于它迫使程序员使用未受保护的数据而导致错误。但是错误总是出现在调用 strFromUns 的函数内，而不是在 strFromUns 本身。因此，你认为应该修改错误的真正根源 strFromUns 呢，还是修改出了错的调用 strFromUns 的这个函数呢？

在我把 Windows Excel 的一个特征移植到 Macintosh Excel 时（当时，它们仍是两个独立源代码段）也出现了类似的问题。在移植了一个特征之后，我开始测试代码并发现有个函数得到了未预料到的 NULL 空指针。我检查了代码，但是错误是在这个函数所调用的函数（传出 NULL）中呢，还是在这个函数本身（没有处理 NULL）呢？因此我找到原来的程序员并问他说明了情况。他马上编辑了该函数并说：“哦，这个函数无处理 NULL 指针能力。”然后，就在我站在边上看着的时候他通过插入如下代码而改正了错误，当指针为 NULL 时“快跳”出来

```
if(pb == NULL)
return(FALSE);
```

我提醒他是否这个函数不应该得到空指针 NULL，错误在调用函数中而不在这个函数中。他回答道：“我清楚这些代码，这样做就可以改正错误了。”但是我觉得这种解决方法好象只改正了错误的症状而没有改正错误的原因，于是我返回我的办公室花了 10 分钟时间来追踪 NULL 空指针的来源。发现空指针 NULL 不仅是这个错误的真正根源，而且也是另外两个已知错误的原因。

还有几次当我追踪到错误的根源时，经常这样认为：“等一下，修改这个函数可能是不对的。如果是这个函数出错的话，函数在另外的地方也应该出问题呀，可是它没有出问题呀。”

我肯定你能猜出为什么函数在另外的地方能够工作，它之所以工作是因为某个程序员已经局部性地修改了这个较为通常的错误。

修改错误要治本，不要治表

你有无事生非的代码吗？

“只要没有破坏作用，怎么改也行。”这似乎是某些程序员的口号。不管代码是否很好地工作，某些程序员总要强行在代码上留下自己的痕迹。如果你曾与那些喜欢将整个文件重新格式化以适合他们口味的程序员工作过的话，你肯定会理解我所讲的内容。尽管大多数程序员对“清理”代码非常谨慎，但是，似乎所有程序员都不同程度地做过这件事情。

清理代码的问题在于程序员总不把改进的代码作为新代码处理。例如，有些程序员在浏览文件时，看到下面所示的代码，他们就把与 0 比较的测试改为与‘\0’作比较的测试（其他程序员也可能想全部删掉测试）。

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

把 0 改为一个空字符的问题是很容易将‘\0’错误地键入为‘0’，但是有多少程序员愿意在做了这样简单的改变之后再测试一下 strcpy 呢？这提醒我们：当你做了如此简单的更改之后，你是否也象对待新编写的代码那样进行完全的测试呢？如果没有，那么这些没有必要的代码更改就会有引入错误的危险。

你可能认为只要修改后的代码仍能通过编译，那么这些更改就不算错。例如，改变局部变量的名字怎么会引出问题呢？可是它确实能引起问题。我曾经跟踪一个错误直到一个函数，这个函数具有一个局部变量名 hPrint，它与具有同样名字的全局变量冲突。由于这个函数在不久前工作还很正常，我查看了老版本的源程序，来看一下到底当前版本改变了什么并验证我的修改是否会重新引入以前有过的错误。我就发现了清理代码问题。老版本中有个局部变量 hPrint1，但是没有 hPrint2 或 hPrint3 来解释名字中‘1’的意义。删除‘1’的程序员肯定认为 hPrint1 是人为的冗余并将其清理为 hPrint，从而引起了名字冲突，导致了错误。

为了避免犯上面的错误，要经常提醒自己：与我一起工作的程序员并非一些笨蛋。当你发现一些有明显错误或显然没有必要的代码时，上面的警句将提醒你小心从事。看上去明显有问题的代码，以后你就会发现，它这样写可能有很好但又不明显的原因。我曾经见过一段荒谬的代码，它唯一的目的是当编译程序代码生成有了错误才工作（这是极罕见的 ———

译者注），如果你清理了这段代码那么就会引人错误。当然这样的代码应该有注释来解释一下它要实现的功能，但是，不是所有的程序员都想得那么周到。

因此，如果你发现了象下面这样的代码：

```
char chGetNext(void)
{
int ch;              /* ch“必须”是 int 类型 */
ch = getchar();
return(chRemapChar(ch));
}
```

不要急于删除显然“没有必要”的 ch，清理成这样的函数：

```
char chGetNext(void)
{
return( chRemapChar(getchar()) );
}
```

这样整理后，如果 chRemapChar 是宏，它会多次求参数的值，从而可能引入了错误。因此，保持“没有必要的”局部变量，避免没有必要的错误。

除非关系产品的成败，否则不要整理代码

把“冷门”特征打入冷宫

避免清理代码只是编写无错代码普遍原则的特例，这个普遍原则就是：如果没有必要就不要编写（或修改）代码。这个建议看上去似乎很奇怪，但是如果你经常提出疑问：“这个特征对产品的成败有什么重要作用？”从而删掉这些特征，那么你就不会陷入困境。

某些特征加到产品中并没有价值，但是它之所以存在仅仅是为了填满特征集；另外一些特征的存在是因为大公司买主要求这些特征；还有一些特征能够存在是因为竞争者的产品具有这些特征，评审人就决定将这些特征纳入特征表中。如果有很好的市场和产品规划小组，那就不应该加入这些没有必要的特征。但是，作为一名程序员，不仅会随大流采用这些特征，甚至还可能是某些没有必要特征的发源人。

你曾经听过程序员说这样的话吗？“如果 WordSmasher 可以做 …… ，那将是个大‘冷门’。”这个所谓“冷门”特征是因为它能提高产品的质量呢，还是因为它的实现在技术上具有挑战性？如果这个特征能提高产品的质量，那么应将该特征推迟到程序的下个版本实现，到那时将对其进行合理的评价并制定相应的进度表。如果这个特征仅仅是一种技术上的挑战，那么否决它。我的建议并不是要抑制创造力，而是要遏制那些不必要的特征以及相关错误的发展。

有时，技术上具有挑战性的特征能提高产品的质量，有时就不能。请小心选择。

不要实现没有战略意义的特征

不存在免费午餐

“自由”特征是那些多余性错误的另一个来源。表面上，自由特征似乎是值得的，因为这只需要很少甚至不需要做任何努力就能跳过已有的设计。怎样才能比这更好呢？具有自由特征会带来很大的问题，尽管它们对产品的成败几乎从未起过任何关键的作用。正如我在上一节讲的，你应该把任何非关键特征看成是错误的来源。程序员向程序内增加自由特征是因为他们可以增加而不是因为他们必须增加。如果它不需要你付出任何代价，那为什么不增加一个特征呢？

啊！但是这是谬论。对于程序员来说，增加自由特征可能不费事，但是对于特征来讲，它不仅仅增多了代码，还必须有人为该特征写又档，还必须有人来测试它。不要忘记还必须有人来修改该特征可能出现的错误。

当我听到某个程序员说某特征是自由的，我就知道他没有花时间来考虑纳入该特征的真正代价。

不设自由特征

灵活性滋生错误

避免错误的另一条策略是排除设计中没有必要的灵活性。这个原则贯穿本书的始终。例如，在第一章，我使用了选择编译警告以避免出现冗余的和有风险的 C 语言惯用语。在第 2 章，我把 ASSERT 定义为一条语句来防止在表达式中错误地使用宏。在第 3 章，我使用了断言来捕获传递给 FreeMemory 的 NULL 指针，即使使用 NULL 指针调用 free 函数是合法的，我也这么做了。…… 我可以列出每一章的例子。

灵活设计的问题在于，设计越灵活，就越难发觉错误。还记得我在第 5 章中针对 realloc 所强调的那几点吗？你几乎可以扔掉 realloc 的任何输入集，可它仍将继续执行，但是它可能并没按你所希望的去执行。更糟糕的是，由于函数很灵活，因此不能插入有意义的断言验证输入的有效性。但是，如果把 realloc 分成为扩展、收缩、分配、释放存储块四个专门函数，则确认函数变元就要容易得多了。

除了过度灵活的函数之外，还应该时刻警惕着过度灵活的特征。由于灵活的特征可能产生一些没有预料到的“合法”情况，你可能会认为这些情况不需要测试甚至认为这就是合法的，因此，灵活特征同样很棘手。

例如，当我为 Apple 的 Excel 和新的 Macintosh Ⅱ机器的 Excel 增加彩色支持程序时，我要从 Windows Excel 上移植一段代码，该代码允许用户指定显示在电子表格格子内的正文颜色。例如，向一个格子内增加颜色，你应该选择已有的格子形式，如下所示（将 1234.5678 打印为$1,234.57）：

```
$#,##0.00
```

并且在前面加上颜色声明。为了显示蓝色，用户就需要将上面的形式改为：

[blue]$#,##0.00

如果用户写了[red]，那么数据以红色显示，如此等等。

Excel 的产品说明非常清楚，颜色说明应放在数据形式的开始处，但是当我移植了这个特征打开始测试代码时，我发现下面的所有形式都工作

$#,##0.00[blue]

$#,##[blue]0.00

$[blue]#,##0.00

你可以将[blue]放在任何地方。当我向原来的程序员询问这是个错误还是个特征时，他说颜色声明可以放在任意位置“仅仅是脱离了语法分析循环。”他不认为允许一点点额外的灵活性是个错误，当时我也那么认为，于是代码就那样保留下来了。然而，回顾一下，我们不应该允许这个额外的灵活性。

不久测试组发现了六个微妙的错误，最终所有这些错误都起因于格式的语法分析程序，因为它没有料想到会发现彩色说明处于格式中间的情况。但是我们没有通过删除没有必要的灵活性来改正这个错误（这需要增加一个简单的 if 语句），而只是改正了这些特定的错误，即改正了错误的症状，从而保留了任何人已不再需要的灵活性。时至今日，Excel 仍允许将彩色说明置于你所希望的任何位置。

因此在你实现特征时要记住：不要使它们具有没有必要的灵活性，但是要容易使用。这两者是有差别的。

不允许没有必要的灵活性

移植的代码也是新代码

在把 Windows Excel 代码移植到 Maxintosh Excel 的过程中，我得到了这样一条教训，人们对这种移植过来的代码，总想少做些检查。毕竟这些代码是在原来的产品中测试过的。我在把移植代码交给测试组之前就应捕获 Excel 数字格式代码中的全部错误，但是我没有这么做。我只是把代码拷贝到 Macintosh Excel，做了一些为把这些代码连接到项目中所必须的修改，然后临时测试了一下代码来验证它已被正确地连接起来了。我没有全面测试特征本身，因为我认为这已经测试过了。这是失策的，特别是在当时的情况下，Windows Excel 本身也正处于开发阶段，这就更是失策。那正是 Microsoft 小组把修改错误推迟到产品周期的最后阶段那个时期。

实际上，不管你是怎样实现特征的，是从头开始设计实现，还是依据某个已有代码来设计实现的，你都有责任排除要加入到项目的那些代码中所存在的错误。如果 Macintosh Excel 只具有与 Windows Excel 相同的错识这可以吗？当然不可以，因为这并不能减轻这些错误的严重性。我一犯懒它就出现了。

“试一试”是个忌讳词

你也许说过多次类似这样的话：“我不知道该怎样来 …… ”，而别的程序员回答你：“你是否试过 …… ？”几乎可以在每个新成立的小组中听到类似这样的对话。某程序员邮出一条消息问：“我怎样才能把光标隐藏起来？”第一个人说：“试着把光标移到屏幕之外去”，另一个人建议：“把光标标志置为 0，光标象素就不可见了”，第三个人或许会说：“光标只是个位映象，因此想办法把它的宽度和高度都置为零”。试、试、试 ……

我承认这是个荒唐的例子，但肯定你听到过类似的对话。通常在被建议“试一试”的所有方案中，可能都不是可以采纳的合适方案。当别人告诉你试一试某件事情时，只是告诉你一个考虑过的猜测并非问题的答案。

试一试各种方案有什么错？假如试验的东西已被系统明确定义的话，那么没有任何错误。但事情常常不是这样，当程序员开始尝试某方案时，他往往会远离他们所了解的系统，进人到饥不择食地寻求解答的境界，这种解很可能有无意识的副作用，将来还要更改。程序员还有个坏习惯，就是有意识地从自由存储区读取，你此对此有何看法呢？free 肯定没有定义自由存储区中的内容是什么，但有些程序员由于某种原因感到他们需要引用自由存储区，他们一试，偏巧成功了，于是他们只好依赖 free 来实施这种行为。

因此注意听取程序员向你提出的建议，如：“你可以试一试 …… ”等，你就会发现大多数建议利用了未定义的或病态定义的副作用。如果程序员提建议时知道怎样求解，他们就不会向你说“试一试”。例如，他们肯定会告诉你“使用 SetCursorState(INVISIBLE)系统调用。”

在找到正确的解法之前，不要一味地“试”，要花时间寻求正确的解

少试多读

几年来，在 Microsoft 的 Macintosh 程序员都能接收到 Macintosh 新闻小组在其内部网络上的一些只读编辑物。这些编辑物很有趣，但它并不十分有用，常常不能回答所提出的问题。总有一些程序员提出那些答案已清楚写在“ 苹果公司内用 Macintosh 手册” 中的问题，

但是，程序员得到的回答除了在手册中清楚地给出解的以外，往往是笼统的解决方法。幸运的是，总有几个 Macintosh 的内部专家能给出明确的答案，如：“参看 Macintosh 内部手册第 4 章，第 32 页，它上面说你应 ……”。

我的观点是：如果你发现你自己正在测试问题的可能解时，停下来，拿出你的手册仔细阅读。这可没有玩代码那么有趣，也没有向别人询问怎么试那么容易，但你将学到许多有关操作系统的知识和如何在它上面编程的知识。

“神圣的”进度表

当要实现相当大的特征时，某些程序员不得不花上两个星期趴在键盘上编写代码，从不着急测试他的程序。另一些程序员则在实现了十来个小特征之后才停下来检查他的程序。如果这种方法能使程序员彻底全面地测试他们的代码，那么这种方法就没有什么错误。但是，这可能吗？

请考虑一下这种情况：一个程序员要用 5 天实现 5 个特征。这个程序员有两种选择：一种是实现一个特征就测试一个特征，一个一个地进行；另一种是五个五个地进行。你认为实际上哪一种方法能产生强健的代码呢？几年来，我考察了这两种编码风格。绝大多数情况下，边编写代码边测试代码的程序员较少出错。我甚至可以告诉你为什么会是这样的。

假设程序员把 5 天时间全部用来实现 5 个特征，但是随后他意识到在进度表中他没有剩下太多的时间来全面测试这些代码了。你认为程序员会用额外的一天或两天来全面测试这些代码吗？或者玩一玩代码，验证一下代码似乎是工作正常的，然后就转到下个特征呢？当然，答案要取决于程序员和工作环境。但是，带来的问题却是是放弃进度计划，还是减少测试，如果放弃进度计划，大多数公司都会表示不满，而减少测试，则会失去负反馈，程序员可能更赞成保持进度计划。

即使程序员单个而不是成批地编写和测试特征，也常常由于进度原因，程序员仍要减少测试。但是当程序员成批实现特征时效果更加明显。在一批特征中只要有一个困准特征就会占用所有特征的测试时间。

使用进度表的缺点是程序员会给速度比测试还高的优先级，本质上就是进度获得了比写正确代码还高的优先级。我的经验是，如果程序员按照进度的时间来编写某个特征的代码，那么即使减少测试，他也要按进度“完成”该特征。他会想到：“如果在代码中有某些未知的错误，测试组会通知我的。”

为了避免这一陷阱，尽量编写和测试小块代码，不要用进度作为借口跳过测试这一步。

尽量编写和测试小块代码。
即使测试代码会影响进度，也要坚持测试代码

名实难符

第 5 章曾解释过，getchar 的名字经常使得程序员认为，该函数返回一个字符，它实际上返回一个 int。同样，程序员经常认为，测试组会测试他们的代码，这是他们的工作，除此之外，他们还干什么呢？其实这种看法是错误的。无论程序员们怎么认为，测试组的存在并非是为了测试程序员写的代码，他们是为了保护公司，最终使用户不受低劣产品的损害。

如果和房屋建筑过程比较一下，就很容易理解测试的作用。在房屋建筑中，建设者建房，检查员检查它。但是检查员并不“测试”这些房屋：电气工程师决不会亲自去安装房屋的电线，也决不会在不接通电源，不测试保险盒，不用万用表检查每个出线口之前就交付线路。这个电气工程师决不会认为：“我不必做这些测试，如果有问题，检查员会通知我的。”有这种想法的电气工程师会很快发现他们难以找到工作。

就象上述房屋检查员一样，程序测试员不负责测试程序的主要理由是，他们不具备必要的工具和技巧，虽然有例外，但这是一条原则。尽管和计算机界的说法不一样，但是测试员测试用户的代码不可能要比用户自己测试的更好。测试员能够加入断言来捕获有问题的数据流吗？测试员能够对存储管理程序进行像第 3 章中那样的子系统测试吗？测试员能够使用调试程序来逐次逐条指令地通过代码，以检查每条代码路径是否都按照设计的要求工作吗？而现状是，尽管程序员测试他们的代码要比测试员有效得多，但是他们却不做，这就是因为计算机界有这些说法。

但是不要误解我的意思，测试组在开发过程中起着重要的作用，但决不是程序员所想象的那种作用。他们在检验产品时，寻找使程序失败的缺陷，证实产品是否与以前推出的产品不兼容，提醒发展部门改进产品性能，利用产品在实际使用中的情况来证实产品的这些特征是非常有用的。所有上述的都跟测试无关，仅仅是在产品中注入质量。

因此，请记住第 2 章中所说的，如果要持续不断地写出无错代码，就必须抓住要害并不受其控制，不要依靠测试组来发现错误，因为这不是他们的工作。

测试代码的责任不在测试员身上，而是程序员自己的责任

重复劳动

如果程序员负有测试代码的责任，那么就自然出现了这个问题：“程序员和测试员在做重复的努力吗？”可能是。但是再问一遍，当然不是。程序员测试代码，是从里向外测试，而测试员则是从外向里测试。

例如，程序员测试代码时，总是从测试每个函数开始，逐次逐条指令（或行）地通过各条代码路径，验证代码和数据流，逐步向外移动来证实函数能够在子系统中与其它函数一道正常操作，最后程序员利用单元测试来验证各个独立的子系统之间能够正确地相互配合。通过单元测试，还能检测内部数据结构的状态。

另一方面，测试员却把代码作为一个黑盒子，从程序的各个输入处进行测试以寻找错误，测试员也可能利用回归测试来证实所有报告的错误都已排除。然后，测试员逐步向里推进，利用代码覆盖工具，来检查在全局测试中执行了多少内部代码，随之获得的信息产生新的测试，来执行未接触到的代码。

这是使用两个不同“算法”测试程序的例子。之所以这样，是因为程序员强调的是代码而测试员强调的是特征，两者从不同的方位考虑问题，这就增加了发现未知错误的机会。

遭白眼的测试员

读者是否注意到，当测试组发现一个错误后，有多少程序员发出宽慰的叹息，他们会说：“唷！我很高兴程序在交付之前测试出了这个错误。”然而另有一些程序员，在测试员报告他们程序中的错误特别是指出一段代码中的多个错误时，他们却对此忿恨不满。我曾经见到过这种程序员怒发冲冠，也曾听到有些项目负责人说为什么测试员让我不得安宁，这是测试错误，因为我们已经删掉了这个数据。”有一次，我还阻止过一位项目负责人和一位测试负责人之间的拳打脚踢，原因是项目负责人已经处于推迟交付产品的巨大压力之下，而测试组还在继续报告错误，这使他很不安。

这听起来是否是很愚蠢？的确是很愚蠢。在我们没有注意到这个产品是在非难和压力下交付之前，容易觉得这是多么荒唐可笑。但是设身处地想一想，如果你被错误包围着，交付期已过数月，便很容易认为这些测试员的确是坏家伙。

每当我看见程序员向测试人员发火时，我总是把他们拉到一旁并问他们：你们为什么要测试人员为程序员所犯的错误负责呢，和测试员发火毫无道理，他们仅仅是报信者。

每当测试员向你报告你的代码中有某个错误时，你最先的反应是震惊和不相信，你本来就没想到测试员会在你的代码中发现错误；你的第二个反应应该是表示感谢，因为测试员帮助你避免交付错误。

不要责怪测试员发现了你的错误

不存在荒谬的错误

有时你会听到程序员抱怨某个错误太荒谬 ,或者抱怨某个测试员经常报告一些愚蠢的错误。如果你听到这样的抱怨时 , 制止并提醒他 , 测试员并不判断错误的严重性 , 也不说这些错误是否值得排除。测试员必须报告所有的错误 , 不管是愚蠢的还是不愚蠢的 , 尽管测试员知道 , 有些愚蠢的错误可能是某个严重问题的副作用。

但是真正的问题是 ,程序员在测试这个代码时为什么没有捕获这些错误呢？即使这些错误很轻微并且不值得排除 ,但找出错误的根源也是非常重要的 ,以避免将来出现类似的错误。

一个错误可能很轻微。但是它的存在本身就很严重。

建立自己的优先顺序

如果往前翻几页重温一下本书的主要观点，你就会惊奇地发现，其中有些观点似乎是相互矛盾的。然而当你仔细思考以后，你可能又不那么认为了。总之，程序员要经常和编写快速代码和紧凑代码这样相互矛盾的目标打交道。

因此问题是，当面临两个可能的实现时究竟选哪一个？可以肯定要在快速算法和小巧算法之间做出选择是比较困难的，但是，要在快速算法与可维护算法之间、或者在小巧但有风险的算法与较大但易于测试的算法之间作出选择时，你会做出怎样的选择呢？肯定有些程序员会不假思索地回答这些问题，但也有一些程序员不能确定到底选择哪一种。如果几星期之后再问他们同样的问题，他们将会给出不同的答案。

程序员之所以不能确定这种互易的理由是，因为他们除了知道象大小或速度这些非常普通的优先次序之外，不知道他们自己的优先顺序是什么。但是在程序设计中如果没有明确的优先顺序，就象是盲人骑瞎马一样，在每个转弯处都要停下来并问问自己：“现在我该怎么办？”这时你往往做出错误的选择。

然而有些程序员，他们清楚地知道自己的优先顺序，但是由于他们的优先顺序不正确或相抵触，在关键问题上他们没有认真思考因此不断地作出错误的选择。例如，许多有经验的程序员仍受 70 年代末期提倡的优先顺序的影响，那时存储空间很少，微机运行很慢，为了写有用的程序，必须要用可维护性来换取空间和速度。但是现在，程序越来越复杂，而 RAM 的容量却越来越大，计算机运行速度也不断加快，以至于即使用很差的算法，大多数任务也都能按时完成。因此现在的优先顺序不同了，不再用可维护性来换取空间和速度，否则就会得到在速度上并不明显地快但又不可维护的程序。仍还有一些程序员把程序大小和速度奉为神灵，把它们看作是产品成败的关键。由于这些程序员的有限顺序已经过时，因此他们一直在做着错误的实现选择。

因此，只要你还没有考虑过你的优先顺序，那么你就要坐下来为你自己（如果你是项目

负责人，就为你的小组）认真地建立优先级列表，从而使你能够在完成项目目标的过程中不断地作出最佳选择。注意我说的是“项目目标”。你的优先级列表不应该反映你想要做的，而应反映你应该做的。例如，某些程序员可能把“个人表达方式”列为最高优先级，这样对程序员或者对产品有利吗？这些程序员接不接受命名标准？同不同意使用{}的定位风格，还是自搞一套呢？

应当指出，没有“正确”的方法来确定你的优先级序列，但是所选定的优先顺序将决定代码的风格和质量。让我们看一看约克和吉尔两个程序员的优先级列表：

约克的优先级列表	吉尔的优先级列表
正确性	正确性
全局效率	可测试性
大小	全局效率
局部效率	可维护性 / 明晰性
个人方便性	一致性
可维护性 / 明晰性	大小
个人表达方式	局部效率
可测试性	个人表达方式
一致性	个人方便性

这些优先顺序将怎样影响约克和吉尔的代码呢？两人都首先集中在写出正确代码上，这仅仅是他们在优先级排列上唯一的相同之处。可以看出，约克非常重视大小和速度，对编写清晰代码关心一般，几乎不怎么考虑测试代码是否容易。

而吉尔则把更多的注意力放在编写正确的代码上，只是在大小和速度危及到产品是否成功时，才把它们作为考虑对象。吉尔认为，除非能够很容易地测试代码，否则就无法验证代码是否正确，因此吉尔把可测试性放在优先级排列顺序列表中很高的位置。

你认为这两个程序员哪个更可能：

- 使得所选择的编译程序都能自动捕获错误并报警？虽然为了使用安全环境可能需要额外做点工作。
- 使用断言和子系统作调试检查？
- 走查每一条代码路径从微观赏验证所有刚写的新代码？
- 用安全函数界面取代有风险的函数界面？虽然在每个调用点可能要额外声称 1～2 条以上的指令。
- 使用可移植类型，以及当用到移位的情况下而用除法（例如使用/4 代替>>2）？
- 避免使用第 7 章中的效率技巧？

这是个充满问题的表吗？我不这样认为。我问你：“你认为吉尔和约克谁会读这本书并按照书中的建议取做？” 吉尔和约克谁会去阅读《程序设计风格要素》或者其他指导性书籍，并按照书中的建议去做呢？

读者应该注意到，由于约克的优先顺序安排，他会把注意力集中在对产品不利的代码上，他回在如何使每一行代码尽量快和小上浪费时间，而对产品长期健全却很少考虑。吉尔

却相反，根据她的优先顺序，她把注意力集中在产品上，而不是代码上，除非证明（或显然）确实需要考虑大小和速度，否则她不考虑大小和速度。

现在想一想，代码和产品哪个对你的公司更重要？因此你的优先顺序应该是怎样的？

建立自己优先级列表并坚持之

说出道道

你是否看过别人写的代码，并奇怪他们为什么这样写呢？你是否就此代码问过他们，而后他们说：“哎呀，我不知道我为什么这样写，我猜我当时感觉到这样写正确吧。”

我经常评审代码，寻找帮助程序员改进技术的方法。我发现“哎呀，我不知道”这样的回答相当普遍，我还发现作出这种回答的程序员没有建立明确的优先顺序，他们的决定似乎具有随意性。相反地，具有明确优先顺序的程序员，精确地知道他们为什么选择这个实现，并且当问及他为什么这样实现时，他能够说出道道。

小结

本章还没有提到一个很重要的观点，这就是：你必须养成经常询问怎样编写代码的习惯。本书就是长期坚持询问一些简单问题所得的结果。

- 我怎样才能自动检测出错误？
- 我怎样才能防止错误？
- 这种想法和习惯是帮助我编写无错代码呢还是妨碍了我编写无错代码？

本章的所有观点都是询问最后一个问题所产生的结果。审视一下自己的观念很重要，这些观念就反映了个人考虑问题的优先次序。如果你认为测试组的存在是为了测试你的代码，那么在编写无错代码方面你就会继续有麻烦，因为你的观念在某种程度上告诉你，在测试方面马虎点是可以的。如果你没有在观念上想着要编写无错代码，那你怎么可能会试着编写无错代码呢？

如果你想编写无错代码，就应该清除妨碍你达到这一目标的观念，清除的方法就是反问一下自己，自己的观念对达到目标是有益的还是有害的。

要点：

- 错误既不会自己产生，也不会自己改正。如果你得到了一个错误报告，但这个错误不再出现了。不要假设测试员发生了幻觉，而要努力查找错误，甚至要恢复程序的

老版本。

- 不能“以后”再修改错误。这是许多产品被取消的共同教训。如果在你发现错误的时候就及时地更正了错误，那你的项目就不会遭受毁灭性的命运。当你的项目总是保持近似于 0 个错误时，怎么可能会有一系列的错误呢？
- 当你跟踪查到一个错误时，总要问一下自己，这个错误是否会是一个大错误的症状。当然，修改一个刚刚追踪到的症状很容易，但是要努力找到真正的起因。
- 不要编写没有必要的代码。让你的竞争者去清理代码，去实现“冷门”但无价值的特征，去实现自由特征。让他们花大量的时间去修改由于这些无用代码所引起的所有没有必要的错误。
- 记住灵活与容易使用并不是一回事。在你设计函数和特征时，重点是使之容易使用；如果它们仅仅是灵活的，象 realloc 函数和 Excel 中的彩色格式特征那样，那么就没法使得代码更加有用；相反地，使得发现错误变得更困难了。
- 不要受“试一试”某个方案以达到预期结果的影响。相反，应把花在尝试方案上的时间用来寻找正确的解决方法。如果必要，与负责你操作系统的公司联系，这比提出一个在将来可能会出问题的古怪实现要好。
- 代码写得尽量小以便于全面测试。在测试中不要马虎。记住，如果你不测试你的代码，就没有人会测试你的代码了。无论怎样，你也不要期望测试组为你测试代码。
- 最后，确定你们小组的优先级顺序，并且遵循这个顺序。如果你是约克，而项目需要吉尔，那么至少在工作方面你必须改变习惯。

课题：

说服你们程序设计组建立或采纳一个优先级列表。如果你们公司具有不同层次的人才（例如初级程序设计员，程序设计员，高级程序设计员，程序设计分析员），你可能要考虑不同的层次使用不同的优先级列表，为什么？

附录A　编码检查表

本附录给出的问题列表，总结了本书的所有观点。使用本表的最好办法是花两周时间评审一下你的设计和编码实现。先花几分钟时间看一看列表，一旦熟悉了这些问题，就可以灵活自如地按它写代码了。此时，就可以把表放在一边了。

一般问题

—— 你是否为程序建立了 DEBUG 版本？
—— 你是否将发现的错误及时改正了？
—一 你是否坚持彻底测试代码．即使耽误了进度也在所不惜？
—— 你是否依靠测试组为你测试代码？
—一 你是否知道编码的优先顺序？
—一 你的编译程序是否有可选的各种警告？

关于将更改归并到主程序

—一 你是否将编译程序的警告（包括可选的）都处理了？
—— 你的代码是否未用 Lint
—一 你的代码进行了单元测试吗？
—一 你是否逐步通过了每一条编码路径以观察数据流？
—一 你是否逐步通过了汇编语言层次上的所有关键代码？
—— 是否清理过了任何代码？如果是，修改处经过彻底测试了吗？
—一 文档是否指出了使用你的代码有危险之处？
—— 程序维护人员是否能够理解你的代码？

每当实现了一个函数或子系统之时

—一 是否用断言证实了函数参数的有效性？

—— 代码中是否有未定义的或者无意义的代码？
—— 代码能否创建未定义的数据？
—— 有没有难以理解的断言？对它们作解释了没有？
—— 你在代码中是否作过任何假设？
—— 是否使用断言警告可能出现的非常情况？
—— 是否作过防御性程序设计？代码是否隐藏了错误？
—— 是否用第二个算法来验证第一个算法？
—— 是否有可用于确认代码或数据的启动（startup）检查?
—— 代码是否包含了随机行为？能消除这些行为吗？
—— 你的代码若产生了无用信息，你是否在 DEBUG 代码中也把它们置为无用信息？
—— 代码中是否有稀奇古怪的行为？
—— 若代码是子系统的一部分，那么你是否建立了一个子系统测试？
—— 在你的设计和代码中是否有任意情况？
—— 即使程序员不感到需要，你也作完整性检查吗？
—— 你是否因为排错程序太大或太慢，而将有价值的 DEBUG 测试抛置一边？
—— 是否使用了不可移植的数据类型？
—— 代码中是否有变量或表达式产生上溢或下溢？
—— 是否准确地实现了你的设计？还是非常近似地实现了你的设计？
—— 代码是否不止一次地解同一个问题？
—— 是否企图消除代码中的每一个 if 语句？
—— 是否用过嵌套？：运算符？
—— 是否已将专用代码孤立出来？
—— 是否用到了有风险的语言惯用语？
—— 是否不必要地将不同类型的运算符混用？
—— 是否调用了返回错误的函数？你能消除这种调用吗？
—— 是否引用了尚未分配的存储空间？
—— 是否引用已经释放了的存储空间？
—— 是否不必要地多用了输出缓冲存储？
—— 是否向静态或全局缓冲区传送了数据？
—— 你的函数是否依赖于另一个函数的内部细节？
—— 是否使用了怪异的或有疑问的 C 惯用语？
—— 在代码中是否有挤在一行的毛病？
—— 代码有不必要的灵活性吗？你能消除它们吗？
—— 你的代码是经过多次“试着”求解的结果吗？
—— 函数是否小并容易测试？

每当设计了一个函数或子系统后

—一 此特征是否符合产品的市场策略？

—一 错误代码是否作为正常返回值的特殊情况而隐藏起来？

—一 是否评审了你的界面，它能保证难于出现误操作吗？

—一 是否具有多用途且面面俱到的函数？

—一 你是否有太灵活的（空空洞洞的）函数参数？

—一 当你的函数不再需要时，它是否返回一个错误条件？

—一 在调用点你的函数是出易读？

—一 你的函数是否有布尔量输入？

修改错误之时

—— 错误无法消失，是否能找到错误的根源？

—一 是修改了错误的真正根源，还是仅仅修改了错误的症状？

附录B　内存登录例程

本附录中的代码实现了第 3 章中讨论的内存登录例程的一个简单链表版本。这个代码有意作了简化使之便于理解，但这并不意味着它不可以用在那些大量地使用内存管理程序的应用之中。但在你花时间重写代码使其使用 AVL 树、B 树或其它可以提供快速查找的数据结构之前，试一下这个代码验证它对于实际应用是否太慢了。你也许会发现这个代码很合用，特别是在没有分配许多全局共享的存储模块之时，更是如此。

该文件中给出的实现是很直观的：每当分配一个内存块时，该例程就额外地分配一小块内存以存放 blockinfo(块信息)结构，块信息中有登录信息(定义见下文)。当一个新的 blockinfo 结构创建时，就填充登录信息并置于链表结构的头部。该链表没有特意的顺序。再次说明，该实现是精选的，因为它既简单又容易理解。

block.h：

```
# ifdef DEBUG
    /* ---------------------------------------------------------------------
     * blockinfo 是个数据结构. 它记录一个已分配内存块的存储登录信息。
     * 每个已分配的内存块在内存登录中都有一个相应的 blockinfo 结构
     */
    typedef struct BLOCKINFO
    {
    struct BLOCKINFO * pbiNext;
    byte* pb;                       /* 存储块的开始位置 */
    size_t size;                /* 存储块的长度 */
    flag fReferenced;               /* 曾经引用过吗？*/
    }blockinfo;                         /* 命名：bi、*pbi */

    flag fCreateBlockInfo(byte* pbNew, size_t sizeNew);
    void FreeBlockInfo(byte* pbToFree);
    void UpdateBlockInfobyte(byte* pbOld, byte* pbNew, size_t sizeNew);
    size_t sizeofBlock(byte* pb);

    void ClearMemoryRefs(void);
    void NoteMemoryRef(void* pv);
    void CheckMemoryRefs(void);
    flag fValidPointer(void* pv, size_t size);
```

```
#endif
```

block.c:

```
#ifdef DEBUG
    /* ---------------------------------------------------------------
     * 该文件中的函数必须要对指针进行比较，而 ANSI 标准不能确保该操作是
     * 可移植的。
     *
     * 下面的宏将该文件所需的指针比较独立出来。该实现采用了总能进行直接
     * 比较的“直截了当”的指针，下面的定义对某些通用 80x86 内存模型不适用。
     */
    #define fPtrLess(pLeft,pRight)              ((pLeft) < (pRight))
    #define fPtrGrtr(pLeft,pRight)              ((pLeft) < (pRight))
    #define fPtrEqual(pLeft, pRight)         ((pLeft) = = (pRight))
    #define fPtrLEssEq(pLeft, pRight)           ((pLEft) < = (pRight))
    #define fPtrGrtrEq(pLeft, pRight)           ((pLeft) > = (pRright))

    /* ---------------------------------------------------------------- */
    /*                 * * * * 私有数据/函数 * * * *                    */
    /* ---------------------------------------------------------------- */

    /* ---------------------------------------------------------------
     * pbiHead 指向内存管理程序调试的单向链接列表。
     */

    static blockinfo* pbiHead = NULL;
    /* ---------------------------------------------------------------
     * pbiGetBlockInfo(pb)
     *
     * pbiGetBlockInfo 查询内存登录找到 pb 所指的存储块，并返回指向内
     * 存登录中相应 blockinfo 结构的指针。注意：pb 必须指向一个已分配的
     * 存储块，否则将得到一个断言失败；该函数或者引发断言或者成功，它从
     * 不返回错误。
     *
     * blockinfo * pbi;
     * ……
```

```
 * pbi = pbiGetBlockInfo(pb);
 * // pbi->pb 指向 pb 所指存储块的开始位置
 * // pbi->size 是 pb 所指存储块的大小
 */

static blockinfo* pbiGetBlockInfo(byte* pb)
{
blockinfo* pbi;
for( pbi = pbiHead; pbi !=  NULL; pbi = pbi->pbiNext )
{
    byte* pbStart = pbi->pb;            /* 为了可读性 */
    byte* pbEnd = pbi->pb + pbi->size  -  1;
    if( fPtrGrtrEq( pb, pbStart ) && fPtrLessEq( pb, pbEnd ) )
        break;
}
/* 没能找到指针？它是（a）垃圾？(b) 指向一个已经释放了的存储块？
 * 或（c）指向一个在由 fResizeMemory 重置大小时而移动了的存储块？
 */
ASSERT( pbi != NULL );
return( pbi );
}

/* ------------------------------------------------------------ */
/*                    * * * * 公共函数 * * * *                  */
/* ------------------------------------------------------------ */

/* ------------------------------------------------------------ */
 * fCreateBlockInfo(pbNew, sizeNew)
 *
 * 该函数为由 pbNew : sizeNew 定义的存储块建立一个登录项。如果成功地
 * 建立了登录信息则该函数返回 TRUE , 否则返回 FALSE 。
 *
 *              if( fCreateBlockInfo( pbNew, sizeNew ) )
 *                  成功 ——— 该内存登录具有 pbNew : sizeNew 项
 *              else
 *                  失败 ——— 由于没有该项则应释放 pbNew
 */
```

```
flag fCreateBlockInfo( byte* pbNew, size_t sizeNew )
{
blockinfo* pbi;
ASSERT( pbNew != NULL && sizeNew != 0 );
pbi = ( blockinfo* )malloc( sizeof( blockinfo ) );
if( pbi != NULL )
{
    pbi->pb = pbNew;
    pbi->size = sizeNew;
    pbi->pbiNext = pbiHead;
    pbiHead = pbi ;
}
return(flag)( pbi != NULL );
}

/* ------------------------------------------------------------------
 * FreeBlockInfo( pbToFree )
 *
 * 该函数清除由 pbToFree 所指存储块的登录项。pbToFree 必须指向一
 * 个已分配存储块的开始位置，否则将得到一个断言失败。
 */

void FreeBlockInfo( byte* pbToFree )
{
blocinfo *pbi, *pbiPrev;
for( pbi = pbiHead; pbi != NULL; pbi = pbi->pbiNext )
{
    if( fPtrEqual( pbi->pb, pbToFree ) )
    {
        if( pbiPrev == NULL )
            pbiHead = pbi->pbiHead;
        else
            pbiPrev->pbiNext = pbi->pbiNext;
        break;
    }
    pbiPrev = pbi;
```

```
}

/* 如果是 pbi 是 NULL 则 pbToFree 无效 */
ASSERT( pbi != NULL );
/* 在释放之前破坏*pbi 的内容 */
memset(pbi, bGarbage, sizeof(blockinfo));
free(pbi);
}

/* ------------------------------------------------------------------
 * UpdateBlockInfo ( pbOld , pbNew , sizeNew )
 *
 * UpdateBlockInfo 查出 pbOld 所指存储块的登录信息，然后该函数修
 * 改登录信息已反映该存储块现在所处的新位置（pbNew）和新的字节长
 * 度（sizeNew)。pbOld 必须指向一个已分配存储块的开始位置，否则
 * 将得到一个断言失败。
 */
void UpdateBlockInfo( byte* pbOld, byte* pbNew, size_t sizeNew )
{
blockinfo* pbi;
ASSERT( pbNew != NULL && sizeNew != 0 );
pbi = pbiGetBlockInfo( pbOld );
ASSERT( pbOld == pbi->pb );          /* 必须指向一个存储块的开始位置 */
pbi->pb = pbNew;
pbi->size = sizeNew;
}

/* ------------------------------------------------------------------
 * sizeofBlock (pb )
 * sizeofBlock 返回 pb 所指存储块的大小。pb 必须指向一个已分配存储块
 * 的开始位置，则将得到一个断言失败。
 */
size_t sizeofBlock( byte* pb )
{
blockinfo* pbi;
pbi = pbiGetBlockInfo(pb);
ASSERT(pb == pbi->pb );          /* 必须指向存储块的开始位置 */
```

```
return( pbi->size );
}


/* ---------------------------------------------------------------- */
/*        下面例程用来寻找丢失的存储块和悬挂的指针。有关这些例程的     */
/*                                说明参见第三章                    */
/* ---------------------------------------------------------------- */

/* ----------------------------------------------------------------
 * ClearMemoryRefs( void )
 *
 * ClearMemoryRefs 将内存登录中所有存储块标志为未引用。
 */
void ClearMemoryRefs( void )
{
blockinfo* pbi;
for( pbi = pbiHead; pbi != NULL; pbi = pbi->pbiNext )
    pbi->fReferenced = FALSE;
}

/* ----------------------------------------------------------------
 * NoteMemoryRef( pv )
 *
 * NoteMemoryRefs 将 pv 所指的存储块标志为已引用。注意：pv 不必指向一
 * 个存储块的开始位置；它可以指向一个已分配存储块的任何位置。
 */

void NoteMemoryRef ( void * pv )
(
blockinfo* pbi;
pbi = pbiGetBlockInfo( (byte*)pv );
pbi->fReferenced = TRUE;
}

/* ----------------------------------------------------------------
 * CheckMemoryRefs( void )
```

```
 * CheckMemoryRefs 扫描内存登录以寻找未通过调用 NoteMemoryRef 进行标志
 * 的存储块。如果该函数发现了一个未被标志的存储块，它就引发断言。
 */
void CheckMemoryRefs( void )
{
blockinfo * pbi ;
for( pbi = pbiHead; pbi != NULL; pbi = pbi->pbiNext )
{
    /* 简单检查存储块的完整性。如果引发该断言，就说明管理 blockinfo
     * 的调试代码有某些错误，或者说明紊乱的内存已经破坏了数据结构。
     * 无论哪种情况，都存在错误。
     */
    ASSERT ( pbi->pb != NULL && pbi->size != 0 );
    /* 检查丢失/遗漏的存储空间。如果引发了该断言，就说明 app 或者丢
     * 失了该存储块的轨道或者没有用 NoteMemoryRef 解释所有的全局指针。
     */
    ASSERT( pbi->fReferenced );
}
}

/* ------------------------------------------------------------------
 * fValidPointer( pv, size )
 *
 * fValidPointer 验证 pv 指向一个已分配的存储块并且从 pv 所指处到
 * 块的结尾至少有“size”个已分配字节。如果有任一个条件没有满足，
 * fValidPointer 将引发断言；该函数将从不返回 FALSE，fValidPointer
 * 之所以返回一个（总为 TRUE）标记是为了允许在断言宏内调用该函
 * 数。当这不是最有效的方法时，不采用#ifdef DEBUG 或者不引入其它象
 * 断言的宏，而单纯地使用断言来处理调试/交付版本控制。
 *
 * ASSERT( fValidPointer( pb, size ) );
 */
flag fValidPointer( void* pv, size_t size )
{
blockinfo* pbi;
byte* pb = ( byte* )pv;
pbi = pbiGetBlockInfo( pb );            /* 使 pv 有效 */
```

```
    ASSERT( pv != NULL && size != 0 );
    /* pv 是有效的，但 size 呢？（如果 pb + size 上溢出了该存储块，则
     * size 是无效的）
     */
    ASSERT( fPtrLessEq( pb + size, pbi->pb + pbi->size ) );
    return( TRUE );
    }
#endif
```

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(expression2)
    ……
}
else
……
if(expression1)
{
if(expression2)
    ……
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( stdeer );
    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(*ppb, sizeNew + sizeofDebugByte); /* 变更了 */
if( pbResize != NULL )
{
    #ifdef DEBUG
    {
        *( pbResize + sizeNew ) = bDebugByte;            /* 加 */
        UpdateBlockInfo( *ppb, 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？

解决这个问题的一个方法是：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去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 );
#enfif
……

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 SetFarilures 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 是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, strRgiht) < 0 )
#define fStrGreater(strLeft, strRight)  ( strcmp(strLeft, strRight) > 0 )
#define fStrEqual(strLeft, strRight)( strcmp(strLeft, strRgiht) == 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( … ) == FASLE )
```

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市完全正确的，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：

```
                mover0, #PRINTER
                callprintDevice
                ……
printDisplay:   mover0, #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。

后记　走向何方

我们的讨论就要结束了，读者或许还会带着疑虑问我：“你本人相信有可能写出无错程序吗？”当然不能绝对地、百分之百地保证这一点。但是可以相信，只要你坚持照做就能写出非常接近于无错的程序。这就象粉刷房间一样，在粉刷房间时可以不弄脏地毯，但必须在地上铺上布，围上挡板，并小心地粉刷，还要下决心坚持到底。同样，读者必须努力剔除代码中的错误，要想做到这一点的唯一途径就是按照正确的方向坚持下去。

尽管你把写无错代码置于首位，但是只利用本书中给出的技术还不能完全达到这一目标。事实上，不存在一个能保证编码不出任何错误的准则表。因此，最重要的是，读者自己要坚持建立一个查错表列出你查出的错误，以避免重犯以前犯过的错误。这个表中的某些项也许会使你大吃一惊。

例如，我曾经将一个令人讨厌的微小错误引入了 Excel：在浏览一个文件时，不小心删了一行。当时我没有检测到这一错误，把这一改动了的文件连同其它文件一起合并到主程序。以后别人发现了这个错误并追踪到我这。我就想，怎样才能检测并防止这类错误呢，答案很显然：在把更改了的代码合并到主程序之前，利用源代码控制管理程序列出所做的改变。这一额外的步骤付诸实施并不占用多少时间，在其后的五年中，它帮助我发现了三个重大的错误以及一些不太恰当的小更改。这三个错误是否值五年的努力？对我来说是值得的，因为正如我前面所说，这样做费不了多大的事，我就可以知道不会把不希望的更改带入主程序。再说一遍，改正错误要放在首要地位。

读者或许会发现，评审代码就是为了解决问题，提供较好的文档就是为了帮助开发产品的内部工作人员。如果不采用单元测试，或许你早就这么做了。读者甚至会发现，特意加入第 3 章练习 6 中提到的 DEBUG 代码对帮助测试员是十分有用的。有时候这种解决问题的方法不太切合实际，这时最好的办法就是避开导致错误的算法和实现。

事实上，无法做到完全消除错误，但是通过不懈的努力，可以加大犯同样错误的时间间隔。为帮助大家做到这一点，在附录 A 中给出了程序员使用的检查清单，这个检查清单综合了本书的所有观点。

综上所述，成功地书写无错代码的关键可以总结为一个总的原则

决不允许同样错误出现两次