

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);
}
```

`assert` 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，即它不应该产生其他的副作用。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交换。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“要避免”

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麼查错吗？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麼，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麼或者危险非常明显，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。一般的方法是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 plus 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，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变化”。

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么办呢？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，并让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采取的任何措施都是值得的。防错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:

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

	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					

DIVS :

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，怎样才能以在 `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：

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMPM :	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 CPM 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入口中的指令特征，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。