

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，就像泰坦尼克号邮轮沉没的灾难一样。`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` 不返回好像是内存指针的 `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
char 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 `E0F` 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險。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(*ppv, sizeNew);
    if( pbResize != NULL )
        *ppv = 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 `fResizeMemory` 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额外清晰。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 realloc，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 strFrom 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的有效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, l) == 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了吗?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,只不过是莫名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件事情，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 看作一个只做两件事情的函数，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 and 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。莫名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) 我说过布尔输入参数的存在,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 ferrror(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？