第1章 风 格

人们看到最好的作家有时并不理会修辞学的规则。还好，当他们这样做虽然付出了违反常规的代价，读者还经常能从句子中发现某些具有补偿性的价值。除非作者自己也明确其做法的意思，否则最好还是按规矩做。

William Strunk和E. B. White，《风格的要素》

下面这段代码取自一个许多年前写的大程序：

```
if ( (country == SING) || (country == BRNI) ||
     (country == POL) || (country == ITALY) )
{
        /*
         * If the country is Singapore, Brunei or Poland
         * then the current time is the answer time
         * rather than the off hook time.
         * Reset answer time and set day of week.
         */
        ...
```

这段代码写得很仔细，具有很好的格式。它所在的程序也工作得很好。写这个系统的程序员会对他们的工作感到骄傲。但是这段摘录却会把细心的读者搞糊涂：新加坡、文莱、波兰和意大利之间有什么关系？为什么在注释里没有提到意大利？由于注释与代码不同，其中必然有一个有错，也可能两个都不对。这段代码经过了执行和测试，所以它可能没有问题。注释中对提到的三个国家间的关系没有讲清楚，如果你要维护这些代码，就必须知道更多的东西。

上面这几行实际代码是非常典型的：大致上写得不错，但也还存在许多应该改进的地方。

本书关心的是程序设计实践，关心怎样写出实际的程序。我们的目的是帮助读者写出这样的软件，它至少像上面的代码所在的程序那样工作得非常好，而同时又能避免那些污点和弱点。我们将讨论如何从一开始就写出更好的代码，以及如何在代码的发展过程中进一步改进它。

我们将从一个很平凡的地方入手，首先讨论程序设计的风格问题。风格的作用主要就是使代码容易读，无论是对程序员本人，还是对其他人。好的风格对于好的程序设计具有关键性作用。我们希望最先谈论风格，也是为了使读者在阅读本书其余部分时能特别注意这个问题。

写好一个程序，当然需要使它符合语法规则、修正其中的错误和使它运行得足够快，但是实际应该做的远比这多得多。程序不仅需要给计算机读，也要给程序员读。一个写得好的程序比那些写得差的程序更容易读、更容易修改。经过了如何写好程序的训练，生产的代码更可能是正确的。幸运的是，这种训练并不太困难。

程序设计风格的原则根源于由实际经验中得到的常识，它不是随意的规则或者处方。代码应该是清楚的和简单的 —— 具有直截了当的逻辑、自然的表达式、通行的语言使用方式、

有意义的名字和有帮助作用的注释等，应该避免耍小聪明的花招，不使用非正规的结构。一致性是非常重要的东西，如果大家都坚持同样的风格，其他人就会发现你的代码很容易读，你也容易读懂其他人的。风格的细节可以通过一些局部规定，或管理性的公告，或者通过程序来处理。如果没有这类东西，那么最好就是遵循大众广泛采纳的规矩。我们在这里将遵循《C程序设计语言》(The C Programming Language)一书中所使用的风格，在处理Java和C++ 程序时做一些小的调整。

我们一般将用一些好的和不好的小程序设计例子来说明与风格有关的规则，因为对处理同样事物的两种方式做比较常常很有启发性。这些例子不是人为臆造的，不好的一个都来自实际代码，由那些在太多工作负担和太少时间的压力下工作的普通程序员 (偶然就是我们自己)写出来。为了简单，这里对有些代码做了些精练，但并没有对它们做任何错误的解释。在看到这些代码之后，我们将重写它们，说明如何对它们做些改进。由于这里使用的都是真实代码，所以代码中可能存在多方面问题。要指出代码里的所有缺点，有时可能会使我们远离讨论的主题。因此，在有的好代码例子里也会遗留下一些未加指明的缺陷。

为了指明一段代码是不好的，在本书中，我们将在有问题的代码段的前面标出一些问号，就像下面这段：

```
?    #define ONE 1
?    #define TEN 10
?    #define TWENTY 20
```

为什么这些#define有问题？请想一想，如果某个具有 TWENTY个元素的数组需要修改得更大一点，情况将会怎么样。至少这里的每个名字都应该换一下，改成能说明这些特殊值在程序中所起作用的东西。

```
#define INPUT_MODE 1
#define INPUT_BUFSIZE 10
#define OUTPUT_BUFSIZE 20
```

1.1 名字

什么是名字？一个变量或函数的名字标识这个对象，带着说明其用途的一些信息。一个名字应该是非形式的、简练的、容易记忆的，如果可能的话，最好是能够拼读的。许多信息来自上下文和作用范围(作用域)。一个变量的作用域越大，它的名字所携带的信息就应该越多。

全局变量使用具有说明性的名字，局部变量用短名字。根据定义，全局变量可以出现在整个程序中的任何地方，因此它们的名字应该足够长，具有足够的说明性，以便使读者能够记得它们是干什么用的。给每个全局变量声明附一个简短注释也非常有帮助：

```
int npending = 0;  // current length of input queue
```

全局函数、类和结构也都应该有说明性的名字，以表明它们在程序里扮演的角色。

相反，对局部变量使用短名字就够了。在函数里， `n`可能就足够了， `npoints`也还可以，用`numberOfPoints`就太过分了。

按常规方式使用的局部变量可以采用极短的名字。例如用 `i`、`j`作为循环变量， `p`、`q`作为指针， `s`、`t`表示字符串等。这些东西使用得如此普遍，采用更长的名字不会有什么益处或收获，可能反而有害。比较：

```
?    for (theElementIndex = 0; theElementIndex < numberOfElements;
?            theElementIndex++)
```

```
? elementArray[theElementIndex] = theElementIndex;
```

和

```
for (i = 0; i < nelems; i++)
    elem[i] = i;
```

人们常常鼓励程序员使用长的变量名，而不管用在什么地方。这种认识完全是错误的，清晰性经常是随着简洁而来的。

现实中存在许多命名约定或者本地习惯。常见的比如：指针采用以 p结尾的变量名，例如nodep；全局变量用大写开头的变量名，例如 Global；常量用完全由大写字母拼写的变量名，如CONSTANTS等。有些程序设计工场采用的规则更加彻底，他们要求把变量的类型和用途等都编排进变量名字中。例如用 pch说明这是一个字符指针，用 strTo和strFrom表示它们分别是将要被读或者被写的字符串等。至于名字本身的拼写形式，是使用 npending或numPending还是num_pending，这些不过是个人的喜好问题，与始终如一地坚持一种切合实际的约定相比，这些特殊规矩并不那么重要。

命名约定能使自己的代码更容易理解，对别人写的代码也是一样。这些约定也使人在写代码时更容易决定事物的命名。对于长的程序，选择那些好的、具有说明性的、系统化的名字就更加重要。

C++ 的名字空间和 Java的包为管理各种名字的作用域提供了方法，能帮助我们保持名字的意义清晰，又能避免过长的名字。

保持一致性。相关的东西应给以相关的名字，以说明它们的关系和差异。

除了太长之外，下面这个Java类中各成员的名字一致性也很差：

```
? class UserQueue {
?     int noOfItemsInQ, frontOfTheQueue, queueCapacity;
?     public int noOfUsersInQueue() {...}
? }
```

这里同一个词“队列 (queue)”在名字里被分别写为 Q、Queue或queue。由于只能在类型UserQueue里访问，类成员的名字中完全不必提到队列，因为存在上下文。所以：

```
? queue.queueCapacity
```

完全是多余的。下面的写法更好：

```
class UserQueue {
    int nitems, front, capacity;
    public int nusers() {...}
}
```

因为这时可以如此写：

```
queue.capacity++;
n = queue.nusers();
```

这样做在清晰性方面没有任何损失。在这里还有可做的事情。例如 items和users实际是同一种东西，同样东西应该使用一个概念。

函数采用动作性的名字。函数名应当用动作性的动词，后面可以跟着名词：

```
now = date.getTime();
putchar('\n');
```

对返回布尔类型值(真或者假)的函数命名，应该清楚地反映其返回值情况。下面这样的语句

```
? if (checkoctal(c)) ...
```

是不好的，因为它没有指明什么时候返回真，什么时候返回假。而：

```
if (isoctal(c)) ...
```

就把事情说清楚了：如果参数是八进制数字则返回真，否则为假。

要准确。名字不仅是个标记，它还携带着给读程序人的信息。误用的名字可能引起奇怪的程序错误。

本书作者之一写过一个名为isoctal的宏，并且发布使用多年，而实际上它的实现是错误的：

```
#define isoctal(c) ((c) >= '0' && (c) <= '8')
```

正确的应该是：

```
#define isoctal(c) ((c) >= '0' && (c) <= '7')
```

这是另外一种情况：名字具有正确的含义，而对应的实现却是错的，一个合情合理的名字掩盖了一个害人的实现。

下面是另一个例子，其中的名字和实现完全是矛盾的：

```
public boolean inTable(Object obj) {
    int j = this.getIndex(obj);
    return (j == nTable);
}
```

函数getIndex如果找到了有关对象，就返回 0到nTable-1之间的一个值；否则返回nTable值。而这里inTable返回的布尔值却正好与它名字所说的相反。在写这段代码时，这种写法未必会引起什么问题。但如果后来修改这个程序，很可能是由别的程序员来做，这个名字肯定会把人弄糊涂。

练习1-1 评论下面代码中名字和值的选择：

```
#define TRUE 0
#define FALSE 1

if ((ch = getchar()) == EOF)
    not_eof = FALSE;
```

练习1-2 改进下面的函数：

```
int smaller(char *s, char *t) {
    if (strcmp(s, t) < 1)
        return 1;
    else
        return 0;
}
```

练习1-3 大声读出下面的代码：

```
if ((falloc(SMRHSHSCRTCH, S_IFEXT|0644, MAXRODDHSH)) < 0)
    ...
```

1.2 表达式和语句

名字的合理选择可以帮助读者理解程序，同样，我们也应该以尽可能一目了然的形式写好表达式和语句。应该写最清晰的代码，通过给运算符两边加空格的方式说明分组情况，更一般的是通过格式化的方式来帮助阅读。这些都是很琐碎的事情，但却又是非常有价值的，就像保持书桌整洁能使你容易找到东西一样。与你的书桌不同的是，你的程序代码很可能还会被别人使用。

用缩行显示程序的结构。采用一种一致的缩行风格，是使程序呈现出结构清晰的最省力的方法。下面这个例子的格式太糟糕了：

```
    for(n++;n<100;field[n++]='\0');
    *i = '\0'; return('\n');
```

重新调整格式，可以改得好一点：

```
    for (n++; n < 100; field[n++] = '\0')
        ;
    *i = '\0';
    return('\n');
```

更好的是把赋值作为循环体，增量运算单独写。这样循环的格式更普通也更容易理解：

```
for (n++; n < 100; n++)
    field[n] = '\0';
*i = '\0';
return '\n';
```

使用表达式的自然形式。表达式应该写得你能大声念出来。含有否定运算的条件表达式比较难理解：

```
    if (!(block_id < actblks) || !(block_id >= unblocks))
        ...
```

在两个测试中都用到否定，而它们都不是必要的。应该改变关系运算符的方向，使测试变成肯定的：

```
if ((block_id >= actblks) || (block_id < unblocks))
    ...
```

现在代码读起来就自然多了。

用加括号的方式排除二义性。括号表示分组，即使有时并不必要，加了括号也可能把意图表示得更清楚。在上面的例子里，内层括号就不是必需的，但加上它们没有坏处。熟练的程序员会忽略它们，因为关系运算符(< <= == != >=)比逻辑运算符(&&和||)的优先级更高。

在混合使用互相无关的运算符时，多写几个括号是个好主意。 C语言以及与之相关的语言存在很险恶的优先级问题，在这里很容易犯错误。例如，由于逻辑运算符的约束力比赋值运算符强，在大部分混合使用它们的表达式中，括号都是必需的。

```
while ((c = getchar()) != EOF)
    ...
```

字位运算符(&和|)的优先级低于关系运算符(比如==)，不管出现在哪里：

```
    if (x&MASK == BITS)
        ...
```

实际上都意味着

```
    if (x & (MASK==BITS))
        ...
```

这个表达式所表达的肯定不会是程序员的本意。在这里混合使用了字位运算和关系运算符号，表达式里必须加上括号：

```
if ((x&MASK) == BITS)
    ...
```

如果一个表达式的分组情况不是一目了然的话，加上括号也可能有些帮助，虽然这种括

号可能不是必需的。下面的代码本来不必加括号：

```
?    leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

但加上括号，代码将变得更容易理解了：

```
leap_year = ((y%4 == 0) && (y%100 != 0)) || (y%400 == 0);
```

这里还去掉了几个空格：使优先级高的运算符与运算对象连在一起，帮助读者更快地看清表达式的结构。

分解复杂的表达式。 C、C++和Java语言都有很丰富的表达式语法结构和很丰富的运算符。因此，在这些语言里就很容易把一大堆东西塞进一个结构中。下面这样的表达式虽然很紧凑，但是它塞进一个语句里的东西确实太多了：

```
?    *x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

把它分解成几个部分，意思更容易把握：

```
if (2*k < n-m)
    *xp = c[k+1];
else
    *xp = d[k--];
*x += *xp;
```

要清晰。 程序员有时把自己无穷尽的创造力用到了写最简短的代码上，或者用在寻求得到结果的最巧妙方法上。有时这种技能是用错了地方，因为我们的目标应该是写出最清晰的代码，而不是最巧妙的代码。

下面这个难懂的计算到底想做什么？

```
?    subkey = subkey >> (bitoff - ((bitoff >> 3) << 3));
```

最内层表达式把bitoff右移3位，结果又被重新移回来。这样做实际上是把变量的最低 3位设置为0。从bitoff的原值里面减掉这个结果，得到的将是 bitoff的最低3位。最后用这3位的值确定subkey的右移位数。

上面的表达式与下面这个等价：

```
subkey = subkey >> (bitoff & 0x7);
```

要弄清前一个版本的意思简直像猜谜语，而后面这个则又短又清楚。经验丰富的程序员会把它写得更短，换一个赋值运算符：

```
subkey >>= bitoff & 0x7;
```

有些结构似乎总是要引诱人们去滥用它们。运算符 ?:大概属于这一类：

```
?    child=(!LC&&!RC)?0:(!LC?RC:LC);
```

如果不仔细地追踪这个表达式的每条路径，就几乎没办法弄清它到底是在做什么。下面的形式虽然长了一点，但却更容易理解，其中的每条路径都非常明显：

```
if (LC == 0 && RC == 0)
    child = 0;
else if (LC == 0)
    child = RC;
else
    child = LC;
```

运算符?:适用于短的表达式，这时它可以把 4行的if-else程序变成1行。例如这样：

```
max = (a > b) ? a : b;
```

或者下面这样：

```
printf("The list has %d item%s\n", n, n==1 ? "" : "s");
```

但是它不应该作为条件语句的一般性替换。

清晰性并不等同于简短。短的代码常常更清楚，例如上面移字位的例子。不过有时代码长一点可能更好，如上面把条件表达式改成条件语句的例子。在这里最重要的评价标准是易于理解。

当心副作用。像 ++ 这一类运算符具有副作用，它们除了返回一个值外，还将隐含地改变变量的值。副作用有时用起来很方便，但有时也会成为问题，因为变量的取值操作和更新操作可能不是同时发生。C和C++ 对与副作用有关的执行顺序并没有明确定义，因此下面的多次赋值语句很可能将产生错误结果：

```
?     str[i++] = str[i++] = ' ';
```

这样写的意图是给str中随后的两个位置赋空格值，但实际效果却要依赖于i的更新时刻，很可能把str里的一个位置跳过去，也可能导致只对i实际更新一次。这里应该把它分成两个语句：

```
str[i++] = ' ';
str[i++] = ' ';
```

下面的赋值语句虽然只包含一个增量操作，但也可能给出不同的结果：

```
?     array[i++] = i;
```

如果初始时i的值是3，那么数组元素有可能被设置成3或者4。

不仅增量和减量操作有副作用，I/O也是一种附带地下活动的操作。下面的例子希望从标准输入读入两个互相有关的数：

```
?     scanf("%d %d", &yr, &profit[yr]);
```

这样做很有问题，因为在这个表达式里的一个地方修改了 yr，而在另一个地方又使用它。这样，除非yr的新取值与原来的值相同，否则 profit[yr]就不可能是正确的。你可能认为事情依赖于参数的求值顺序，实际情况并不是这样。这里的问题是： scanf的所有参数都在函数被真正调用前已经求好值了，所以 &profit[yr]实际使用的总是yr原来的值。这种问题可能在任何语言里发生。纠正的方法就是把语句分解为两个：

```
scanf("%d", &yr);
scanf("%d", &profit[yr]);
```

下面的练习里列举了各种具有副作用的表达式。

练习1-4 改进下面各个程序片段：

```
?     if ( !(c == 'y' || c == 'Y') )
?         return;

?     length = (length < BUFSIZE) ? length : BUFSIZE;

?     flag = flag ? 0 : 1;

?     quote = (*line == '"') ? 1 : 0;

?     if (val & 1)
?         bit = 1;
?     else
?         bit = 0;
```

练习1-5 下面的例子里有什么错？

```
    int read(int *ip) {
        scanf("%d", ip);
        return *ip;
    }
        ...
    insert(&graph[vert], read(&val), read(&ch));
```

练习1-6 列出下面代码片段在各种求值顺序下可能产生的所有不同的输出：

```
    n = 1;
    printf("%d %d\n", n++, n++);
```

在尽可能多的编译系统中试验，看看实际中会发生什么情况。

1.3 一致性和习惯用法

一致性带来的将是更好的程序。如果程序中的格式很随意，例如对数组做循环，一会儿采用下标变量从下到上的方式，一会儿又用从上到下的方式；对字符串一会儿用 strcpy做复制，一会儿又用for循环做复制；等等。这些变化就会使人很难看清实际上到底是怎么回事。而如果相同计算的每次出现总是采用同样方式，任何变化就预示着是经过了深思熟虑，要求读程序的人注意。

使用一致的缩排和加括号风格。缩排可以显示出程序的结构，那么什么样的缩排风格最好呢？是把花括号放在if的同一行，还是放在下面一行？程序员们常常就程序的这些编排形式争论不休。实际上，特定风格远没有一致地使用它们重要。不要在这里浪费时间。应该取一种风格，当然作者希望是他们所采用的风格，然后一致地使用。

应该在那些并不必须用花括号的地方都加上它们吗？与一般的圆括号一样，花括号也可以用来消除歧义，但是在使代码更清晰方面的作用却不那么大。为了保持某种一致性，许多程序员总在循环或if的体外面加花括号。当这里只有一个语句时，加花括号就不是必要的，所以作者倾向于去掉它们。如果你赞成我们的方法，那么就要注意在必需的时候不要忽略了它们，例如，在程序里需要解决“悬空的else(dangling else)”问题时。下面是这方面的一个例子：

```
    if (month == FEB) {
        if (year%4 == 0)
            if (day > 29)
                legal = FALSE;
        else
            if (day > 28)
                legal = FALSE;
    }
```

这里的缩排方式把人搞糊涂了，实际上else隶属于行：

```
            if (day > 29)
```

代码本身也是错的。如果一个if紧接在另一个之后，那么请一定加上花括号：

```
    if (month == FEB) {
        if (year%4 == 0) {
            if (day > 29)
                legal = FALSE;
        } else {
            if (day > 28)
                legal = FALSE;
```

```
        }
    }
```

语法驱动的编辑工具可以帮助避免这类错误。

虽然上面程序里的错误已经修正，但这个结果代码还是很难懂。如果我们用一个变量保存二月的天数，计算过程就很容易看明白了：

```
    if (month == FEB) {
        int nday;

        nday = 28;
        if (year%4 == 0)
            nday = 29;
        if (day > nday)
            legal = FALSE;
    }
```

这段代码实际上还是错的——2000年是闰年，而1900和2100都不是。要把现在这个结构改正确是非常容易的。

此外，如果你工作在一个不是自己写的程序上，请注意保留程序原有的风格。当你需要做修改时，不要使用你自己的风格，即使你特别喜欢它。程序的一致性比你本人的习惯更重要，因为这将使随你之后的其他人生活得更容易些。

为了一致性，使用习惯用法。和自然语言一样，程序设计语言也有许多惯用法，也就是那些经验丰富的程序员写常见代码片段的习惯方式。在学习一个语言的过程中，一个中心问题就是逐渐熟悉它的习惯用法。

常见习惯用法之一是循环的形式。考虑在 C、C++和Java中逐个处理n元数组中各个元素的代码，例如要对这些元素做初始化。有人可能写出下面的循环：

```
    i = 0;
    while (i <= n-1)
        array[i++] = 1.0;
```

或者是这样的：

```
    for (i = 0; i < n; )
        array[i++] = 1.0;
```

也可能是：

```
    for (i = n; --i >= 0; )
        array[i] = 1.0;
```

所有这些都正确，而习惯用法的形式却是：

```
for (i = 0; i < n; i++)
    array[i] = 1.0;
```

这并不是一种随意的选择：这段代码要求访问 n元数组里的每个元素，下标从 0到n - 1。在这里所有循环控制都被放在一个 for里，以递增顺序运行，并使用 ++的习惯形式做循环变量的更新。这样做还保证循环结束时下标变量的值是一个已知值，它刚刚超出数组里最后元素的位置。熟悉C语言的人不用琢磨就能理解它，不加思考就能正确地写出它来。

C++或Java里常见的另一种形式是把循环变量的声明也包括在内：

```
for (int i = 0; i < n; i++)
    array[i] = 1.0;
```

下面是在C语言里扫描一个链表的标准循环：

```
for (p = list; p != NULL; p = p->next)
    ...
```

同样，所有的控制都放在一个for里面。

对于无穷循环，我们喜欢用：

```
for (;;)
    ...
```

但

```
while (1)
    ...
```

也很流行。请不要使用其他形式。

缩排也应该采用习惯形式。下面这种垂直排列会妨碍人的阅读，它更像三个语句而不像一个循环：

```
?    for (
?        ap = arr;
?        ap < arr + 128;
?        *ap++ = 0
?        )
?    {
?      ;
?    }
```

写成标准的循环形式，读起来就容易多了：

```
for (ap = arr; ap < arr+128; ap++)
    *ap = 0;
```

这种故意拉长的格式还会使代码摊到更多的页或显示屏去，进一步妨碍人的阅读。

常见的另一个惯用法是把一个赋值放进循环条件里：

```
while ((c = getchar()) != EOF)
    putchar(c);
```

do-while循环远比for和while循环用得少，因为它将至少执行循环体一次，在代码的最后而不是开始执行条件测试。这种执行方式在许多情况下是不正确的，例如下面这段重写的使用getchar的循环：

```
?    do {
?        c = getchar();
?        putchar(c);
?    } while (c != EOF);
```

在这里测试被放在对putchar的调用之后，将使这个代码段无端地多写出一个字符。只有在某个循环体总是必须至少执行一次的情况下，使用 do-while循环才是正确的。后面会看到这种例子。

一致地使用习惯用法还有另一个优点，那就是使非标准的循环很容易被注意到，这种情况常常预示着有什么问题：

```
?    int i, *iArray, nmemb;
?
?    iArray = malloc(nmemb * sizeof(int));
?    for (i = 0; i <= nmemb; i++)
?        iArray[i] = i;
```

在这里分配了nmemb个项的空间，从iArray[0]到iArray[nmemb-1]。但由于采用的是 <=

做循环测试，程序执行将超出数组尾部，覆盖掉存储区中位于数组后面的内容。不幸的是，有许多像这样的错误没能及时查出来，直到造成了很大的损害。

C和C++ 中也有为字符串分配空间及操作它们的习惯写法。不采用这种做法的代码常常就隐藏着程序错误：

```
?     char *p, buf[256];
?
?     gets(buf);
?     p = malloc(strlen(buf));
?     strcpy(p, buf);
```

绝不要使用函数gets，因为你没办法限制它由输入那儿读入内容的数量。这常常会导致一个安全性问题。第6章还会再来讨论这个问题，那里要说明选择 fgets总是更好的。上面代码段里还有另一个问题：strlen求出的值没有计入串结尾的‘\0’字符，而strcpy却将复制它。所以这里分配的空间实际上是不够的，这将使 strcpy的写入超过所分配空间的界限。习惯写法是：

```
p = malloc(strlen(buf)+1);
strcpy(p, buf);
```

或在C++里：

```
p = new char[strlen(buf)+1];
strcpy(p, buf);
```

如果你在这里没有看见 +1，就要当心。

在Java里不会遇到这个特殊问题，那里的字符串不是用零结尾的数组表示，数组的下标也将受到检查。这就使Java不会出现超出数组界限访问的问题。

许多C和C++ 环境中提供了另一个库函数 strdup，它通过调用malloc和strcpy建立字符串的拷贝。有了这个函数，要避免上述错误就变得更简单了。可惜， strdup不是ANSI C标准中的内容。

还有一点，无论是上面的原始代码，还是其修正版本里都没有检查 malloc的返回值。我们忽略这个改进，是为了集中精力处理这里的主要问题。在实际程序中，对于 malloc、realloc、strdup及任何牵涉到存储分配的函数，它们的返回值都必须做检查。

用else-if表达多路选择。多路选择的习惯表示法是采用一系列的if ... else if ... else，形式如下：

```
if (condition1)
    statement1
else if (condition2)
    statement2
...
else if (conditionn)
    statementn
else
    default-statement
```

这里的条件(condition)从上向下读，遇到第一个能够满足的条件，就执行对应的语句，而随后的结构都跳过去。在这里，各个语句部分可以只是单个语句，也可以是由花括号括起的一组语句。最后一个else处理默认情况，或说是处理没有选中其他部分时的情况。如果不存在默认动作，尾随的else部分就可以没有。另一个更好的办法是利用它给出一个错误信息，

以帮助捕捉“不可能发生”的情况。

在这里应该把所有的else垂直对齐，而不是分别让每个else与对应的if对齐。采用垂直对齐能够强调所有测试都是顺序进行的，而且能防止语句不断退向页的右边缘。

一系列嵌套的if语句通常是说明了一段粗劣笨拙的代码，或许就是真正的错误：

```
?     if (argc == 3)
?         if ((fin = fopen(argv[1], "r")) != NULL)
?             if ((fout = fopen(argv[2], "w")) != NULL) {
?                 while ((c = getc(fin)) != EOF)
?                     putc(c, fout);
?                 fclose(fin); fclose(fout);
?             } else
?                 printf("Can't open output file %s\n", argv[2]);
?         else
?             printf("Can't open input file %s\n", argv[1]);
?     else
?         printf("Usage: cp inputfile outputfile\n");
```

这些if要求读它的人在头脑里维持一个下推堆栈，不断记住前面做了什么测试，读到某个地方能把记住的内容弹出来，直到确定了对应动作(如果还记得的话)。由于这里最多就是做一个动作，改变测试顺序完全可以得到一个更清晰的版本。这里我们还纠正了原版本中的资源流失问题[⊖]。

```
if (argc != 3)
    printf("Usage: cp inputfile outputfile\n");
else if ((fin = fopen(argv[1], "r")) == NULL)
    printf("Can't open input file %s\n", argv[1]);
else if ((fout = fopen(argv[2], "w")) == NULL) {
    printf("Can't open output file %s\n", argv[2]);
    fclose(fin);
} else {
    while ((c = getc(fin)) != EOF)
        putc(c, fout);
    fclose(fin);
    fclose(fout);
}
```

从开始向下读，直到遇到第一个值为真的测试，转而去做对应的动作，然后从最后的 else后面继续下去。这里要遵守的规则是：一个判断应该尽可能接近它所对应的动作。也就是说，一旦做过了一个测试，马上就应该去做某些事情。

人们往往企图重复使用某段代码，结果常写出一段紧紧纠缠在一起的程序：

```
?     switch (c) {
?     case '-':  sign = -1;
?     case '+':  c = getchar();
?     case '.':  break;
?     default:   if (!isdigit(c))
?                    return 0;
?     }
```

在这个开关语句里，写了一个狡猾的从上面掉下[⊜]的语句序列，目的不过是避免重写那仅有一行的代码。这样做也不符合习惯写法：case语句最后都应该写一个break，如果偶尔有例外，

⊖ 在源代码中，某些情况下程序执行中将不会释放文件缓冲区，从而造成未释放资源的流失，使它们可能无法重新投入使用。作者的话就是指出原来存在这个错误。——译者

⊜ 指在前一个case语句序列最后不写break，使执行直接“掉”到下面的case语句序列中去。——译者

在那里一定要加上注释。按传统编排方式和结构写出来的内容很容易读，虽然稍微长了一点：

```
? switch (c) {
? case '-':
?     sign = -1;
?     /* fall through */
? case '+':
?     c = getchar();
?     break;
? case '.':
?     break;
? default:
?     if (!isdigit(c))
?         return 0;
?     break;
? }
```

增加长度也不符合提高清晰性的要求。还好，对于这种不常见的结构，用一系列 else-if语句写可能更清楚些：

```
if (c == '-') {
    sign = -1;
    c = getchar();
} else if (c == '+') {
    c = getchar();
} else if (c != '.' && !isdigit(c)) {
    return 0;
}
```

围在每个单行语句块外面的花括号强调了它们是平行结构。

“从上面掉下”的方式在一种情况下是可以接受的，那就是几个 case使用共同的代码段。常规的编排形式是：

```
case '0':
case '1':
case '2':
    ...
    break;
```

这里不需要任何注释。

练习1-7 把下面的C/C++程序例子改得更清晰些：

```
? if (istty(stdin))  ;
? else if (istty(stdout))  ;
?      else if (istty(stderr))  ;
?           else return(0);


? if (retval != SUCCESS)
? {
?     return (retval);
? }
? /* All went well! */
? return SUCCESS;


? for (k = 0; k++ < 5; x += dx)
?     scanf("%lf", &dx);
```

练习1-8 确定下面的Java程序段中的错误，并把它改写为一个符合习惯的循环。

```
int count = 0;
while (count < total) {
    count++;
    if (this.getName(count) == nametable.userName()) {
        return (true);
    }
}
```

1.4 函数宏

老的C语言程序员中有一种倾向，就是把很短的执行频繁的计算写成宏，而不是定义为函数。完成I/O的getchar，做字符测试的isdigit都是得到官方认可的例子。人们这样做最根本的理由就是执行效率：宏可以避免函数调用的开销。实际上，即使是在 C语言刚诞生时(那时的机器非常慢，函数调用的开销也特别大)，这个论据也是很脆弱的，到今天它就更无足轻重了。有了新型的机器和编译程序，函数宏的缺点就远远超过它能带来的好处。

避免函数宏。在C++ 里，在线函数更削减了函数宏的用武之地，在 Java里根本就没有宏这种东西。即使是在C语言里，它们带来的麻烦也比解决的问题更多。

函数宏最常见的一个严重问题是：如果一个参数在定义中出现多次，它就可能被多次求值。如果调用时的实际参数带有副作用，结果就会产生一个难以捉摸的错误。下面的代码段来自某个<ctype.h>，其意图是实现一种字符测试：

```
#define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

请注意，参数c在宏的体里出现了两次。如果isupper在下面的上下文中调用：

```
while (isupper(c = getchar()))
    ...
```

那么，每当遇到一个大于等于A的字符，程序就会将它丢掉，而下一个字符将被读入并去与 Z做比较。C语言标准是仔细写出的，它允许将 isupper及类似函数定义为宏，但要求保证它们的参数只求值一次。因此，上面的实现是错误的。

直接使用ctype提供的函数总比自己实现它们更好。如果希望更安全些，那么就一定不要嵌套地使用像getchar这种带有副作用的函数。我们重写上面的测试，把一个表达式改成两个，这里还为捕捉文件结束留下机会：

```
while ((c = getchar()) != EOF && isupper(c))
    ...
```

有时多次求值带来的是执行效率问题，而不是真正的错误。考虑下面这个例子：

```
#define ROUND_TO_INT(x) ((int) ((x)+(((x)>0)?0.5:-0.5)))
    ...
size = ROUND_TO_INT(sqrt(dx*dx + dy*dy));
```

这种写法使平方根函数的计算次数比实际需要多了一倍。甚至对于很简单的实际参数，像ROUND_TO_INT体这样的复杂表达式也会转换成许多指令。这里确实应该把它改成一个函数，在需要时调用。宏将在它每次被调用的地方进行实例化，结果会导致被编译的程序变大 (C++的在线函数也存在这个缺点)。

给宏的体和参数都加上括号。如果你真的要使用函数宏，那么请特别小心。宏是通过文本替换方式实现的：定义体里的参数被调用的实际参数替换，得到的结果再作为文本去替换原来

的调用段。这种做法与函数不同，常给人带来一些麻烦。假如 square是个函数，表达式：

```
1 / square(x)
```

的工作将很正常。而如果它的定义如下：

```
? #define square(x)  (x) * (x)
```

上面表达式将被展开成一个错误的内容：

```
? 1 / (x) * (x)
```

这个宏应该定义为：

```
#define square(x)  ((x) * (x))
```

这里所有的括号都是必需的。即使是在宏定义里完全加上括号，也不可能解决前面所说的多次求值问题。所以，如果一个操作比较复杂，或者它很具一般性，值得包装起来，那么还是应该使用函数。

C++ 提供的在线函数既避免了语法方面的麻烦，而且又可得到宏能够提供的执行效率，很适合用来定义那些设置或者提取一个值的短小函数。

练习1-9 确定下面的宏定义中的问题：

```
? #define ISDIGIT(c) ((c >= '0') && (c <= '9')) ? 1 : 0
```

1.5 神秘的数

神秘的数包括各种常数、数组的大小、字符位置、变换因子以及程序中出现的其他以文字形式写出的数值。

(1) 给神秘的数起个名字。作为一个指导原则，除了 0和1之外，程序里出现的任何数大概都可以算是神秘的数，它们应该有自己的名字。在程序源代码里，一个具有原本形式的数对其本身的重要性或作用没提供任何指示性信息，它们也导致程序难以理解和修改。下面的片段摘自一个在24×80的终端屏幕上打印字母频率的直方图程序，由于其中存在一些神秘的数，程序的意义变得很不清楚：

```
?       fac = lim / 20;     /* set scale factor */
?       if (fac < 1)
?           fac = 1;
?                           /* generate histogram */
?       for (i = 0, col = 0; i < 27; i++, j++) {
?           col += 3;
?           k = 21 - (let[i] / fac);
?           star = (let[i] == 0) ? ' ' : '*';
?           for (j = k; j < 22; j++)
?               draw(j, col, star);
?       }
?       draw(23, 2, ' '); /* label x axis */
?       for (i = 'A'; i <= 'Z'; i++)
?           printf("%c  ", i);
```

在这段代码里包含许多数值，如 20、21、22、23、27等等。它们应该是互相有关系的……但是……它们确实有关系吗？实际上，在这个程序里应该只有三个数是重要的： 24是屏幕的行数；80是列数；还有26，它是字母表中的字母个数。但这些数在代码中都没出现，这就使上面那些数显得更神秘了。

通过给上面的计算中出现的各个数命名，我们可以把代码弄得更清楚些。我们发现，例如数字3是由(80-1)/26得到的，而let应该有26个项，而不是27个(这个超一(off-by-one)错误可能是由于写程序的人把屏幕坐标当作从1开始而引起的)。通过一些简化后，我们得到的结果代码是：

```
enum {
    MINROW   = 1,                   /* top edge */
    MINCOL   = 1,                   /* left edge */
    MAXROW   = 24,                  /* bottom edge (<=) */
    MAXCOL   = 80,                  /* right edge (<=) */
    LABELROW = 1,                   /* position of labels */
    NLET     = 26,                  /* size of alphabet */
    HEIGHT   = MAXROW - 4,          /* height of bars */
    WIDTH    = (MAXCOL-1)/NLET      /* width of bars */
};
    ...
    fac = (lim + HEIGHT-1) / HEIGHT;    /* set scale factor */
    if (fac < 1)
        fac = 1;
    for (i = 0; i < NLET; i++) {    /* generate histogram */
        if (let[i] == 0)
            continue;
        for (j = HEIGHT - let[i]/fac; j < HEIGHT; j++)
            draw(j+1 + LABELROW, (i+1)*WIDTH, '*');
    }
    draw(MAXROW-1, MINCOL+1, ' ');  /* label x axis */
    for (i = 'A'; i <= 'Z'; i++)
        printf("%c  ", i);
```

现在，主循环到底做什么已经很清楚了：它是一个熟悉的从 0到NLET的循环，是一个对数据(数组)元素操作的循环。程序里对draw的调用也同样容易理解，因为像MAXROW和MINCOL这样的词提醒我们实际参数的顺序。更重要的是，现在我们已经很容易把这个程序修改为能够对付其他的屏幕大小或不同的数据了。数被揭掉了神秘的面纱，代码的意义也随之一目了然了。

把数定义为常数，不要定义为宏。 C程序员的传统方式是用#define行来对付神秘的数值。C语言预处理程序是一个强有力的工具，但是它又有些鲁莽。使用宏进行编程是一种很危险的方式，因为宏会在背地里改变程序的词法结构。我们应该让语言去做正确的工作[⊖]。在C和C++ 里，整数常数可以用枚举语句声明，就像上面的例子里所做的那样。在 C++里任何类型都可使用const声明的常数：

```
const int MAXROW = 24, MAXCOL = 80;
```

在Java中可以用final声明：

```
static final int MAXROW = 24, MAXCOL = 80;
```

C语言里也有const值，但它们不能用作数组的界。这样， enum就是C中惟一可用的选择了。

使用字符形式的常量，不要用整数。人们常用在<ctype.h>里的函数，或者用与它们等价的内容检测字符的性质。有一个测试写成这样：

```
?     if (c >= 65 && c <= 90)
?         ...
```

⊖ 注意，C预处理命令不是C语言本身的组成部分，而是一组辅助成分。这里说“让语言……”，也就是说不要用预处理命令做。——译者

这种写法将完全依赖于特殊的字符表示方式。写成下面这样更好一些：

```
if (c >= 'A' && c <= 'Z')
    ...
```

但是，如果在某个编码字符集里的字母编码不是连续的，或者其中还夹有其他字母，那么这种描述的效果就是错的。最好是直接使用库函数，在C和C++里写：

```
if (isupper(c))
    ...
```

或在Java里面：

```
if (Character.isUpperCase(c))
    ...
```

与此类似的还有另一个问题，那就是程序里许多上下文中经常出现的0。虽然编译系统会把它转换为适当类型，但是，如果我们把每个0的类型写得更明确更清楚，对读程序的人理解其作用是很有帮助的。例如，用(void*)0或NULL表示C里的空指针值，用'\0'而不是0表示字符串结尾的空字节。也就是说，不要写：

```
?   str = 0;
?   name[i] = 0;
?   x = 0;
```

应该写成：

```
str = NULL;
name[i] = '\0';
x = 0.0;
```

我们赞成使用不同形式的显式常数，而把0仅留做整数常量。采用这些形式实际上指明了有关值的用途，能起一点文档作用。可惜的是，在C++里人们都已接受了用0(而不是NULL)表示空指针。Java为解决这个问题采用了一种更好的方法，它定义了一个关键字null，用来表示一个对象引用实际上并没有引用任何东西。

利用语言去计算对象的大小。不要对任何数据类型使用显式写出来的大小。例如，我们应该用sizeof(int)而不是2或者4。基于同样原因，写sizeof(array[0])可能比sizeof(int)更好，因为即使是数组的类型改变了，也没有什么东西需要改变。

利用运算符sizeof常常可以很方便地避免为数组大小引进新名字。例如，写：

```
char buf[1024];

fgets(buf, sizeof(buf), stdin);
```

这里的缓冲区大小仍然是个神秘数。但是它只在这个声明中出现了一次。为局部数组的大小引进一个新名字价值并不大，而写出的代码能在数据大小或类型改变的情况下不需要任何改动，这一点肯定是有价值的。

Java语言中的数组有一个length域，它给出数组的元素个数：

```
char buf[] = new char[1024];

for (int i = 0; i < buf.length; i++)
    ...
```

在C和C++里没有与.length对应的内容。但是，对于那些可以看清楚的数组(不是指针)，下面的宏定义能计算出数组的元素个数：

```
#define NELEMS(array) (sizeof(array) / sizeof(array[0]))
```

```
double dbuf[100];

for (i = 0; i < NELEMS(dbuf); i++)
    ...
```

在这里，数组大小只在一个地方设置。如果数组的大小改变，其余代码都不必改动。对函数参数的多次求值在这里也不会出问题，因为它不会出现任何副作用，事实上，这个计算在程序编译时就已经做完了。这是宏的一个恰当使用，因为它做了某种函数无法完成的工作，从数组声明计算出它的大小。

练习1-10 如何重写下面定义，使出错的可能性降到最小？

```
#define FT2METER    0.3048
#define METER2FT    3.28084
#define MI2FT       5280.0
#define MI2KM       1.609344
#define SQMI2SQKM   2.589988
```

1.6 注释

注释是帮助程序读者的一种手段。但是，如果在注释中只说明代码本身已经讲明的事情，或者与代码矛盾，或是以精心编排的形式干扰读者，那么它们就是帮了倒忙。最好的注释是简洁地点明程序的突出特征，或是提供一种概观，帮助别人理解程序。

不要大谈明显的东西。注释不要去说明明白白的事，比如 i++能够将i值加1等等。下面是我们认为最没有价值的一些注释：

```
/*
 * default
 */
default:
    break;


/* return SUCCESS */
return SUCCESS;


zerocount++;    /* Increment zero entry counter */


/* Initialize "total" to "number_received" */
node->total = node->number_received;
```

所有这些都该删掉，它们不过是一些无谓的喧嚣。

注释应该提供那些不能一下子从代码中看到的东西，或者把那些散布在许多代码里的信息收集到一起。当某些难以捉摸的事情出现时，注释可以帮助澄清情况。如果操作本身非常明了，重复谈论它们就是画蛇添足了：

```
while ((c = getchar()) != EOF && isspace(c))
    ;                           /* skip white space */
if (c == EOF)                   /* end of file */
    type = endoffile;
else if (c == '(')              /* left paren */
    type = leftparen;
else if (c == ')')              /* right paren */
    type = rightparen;
else if (c == ';')              /* semicolon */
    type = semicolon;
```

```
?     else if (is_op(c))              /* operator */
?         type = operator;
?     else if (isdigit(c))            /* number */
?         ...
```

这些注释也都应该删除，因为仔细选择的名字已经携带着有关信息。

给函数和全局数据加注释。注释当然可以有价值。对于函数、全局变量、常数定义、结构和类的域等，以及任何其他加上简短说明就能够帮助理解的内容，我们都应该为之提供注释。

全局变量常被分散使用在整个程序中的各个地方，写一个注释可以帮人记住它的意义，也可以作为参考。下面是从本书第3章取来的一个例子：

```
struct State {  /* prefix + suffix list */
    char    *pref[NPREF];   /* prefix words */
    Suffix  *suf;           /* list of suffixes */
    State   *next;          /* next in hash table */
};
```

放在每个函数前面的注释可以成为帮人读懂程序的台阶。如果函数代码不太长，在这里写一行注释就足够了。

```
// random: return an integer in the range [0..r-1].
int random(int r)
{
    return (int)(Math.floor(Math.random()*r));
}
```

有些代码原本非常复杂，可能是因为算法本身很复杂，或者是因为数据结构非常复杂。在这些情况下，用一段注释指明有关文献对读者也很有帮助。此外，说明做出某种决定的理由也很有价值。下面程序的注释介绍了逆离散余弦变换 (inverse discrete cosine transform, DCT)的一个特别高效的实现，它用在一个JPEG图像解码器里：

```
/*
 * idct: Scaled integer implementation of
 * Inverse two dimensional 8x8 Discrete Cosine Transform,
 * Chen-Wang algorithm (IEEE ASSP-32, pp 803-816, Aug 1984)
 *
 * 32-bit integer arithmetic (8-bit coefficients)
 * 11 multiplies, 29 adds per DCT
 *
 * Coefficients extended to 12 bits for
 * IEEE 1180-1990 compliance
 */

static void idct(int b[8*8])
{
    ...
}
```

这个很有帮助的注释点明了参考文献，简短地描述了所使用的数据，说明了算法的执行情况，还说明为什么原来的代码应该修改，以及做了怎样的修改等等。

不要注释差的代码，重写它。应该注释所有不寻常的或者可能迷惑人的内容。但是如果注释的长度超过了代码本身，可能就说明这个代码应该修改了。下面的例子是一个长而混乱的注释和一个条件编译的查错打印语句，它们都是为了解释一个语句：

```
    /* If "result" is 0 a match was found so return true (non-zero).
       Otherwise, "result" is non-zero so return false (zero). */

    #ifdef DEBUG
    printf("*** isword returns !result = %d\n", !result);
    fflush(stdout);
    #endif

    return(!result);
```

否定性的东西很不好理解，应该尽量避免。在这里，部分问题来自一个毫无信息的变量名字result。改用另一个更具说明性的名字matchfound之后，注释就再没有存在的必要，打印语句也变得清楚了：

```
#ifdef DEBUG
printf("*** isword returns matchfound = %d\n", matchfound);
fflush(stdout);
#endif

return matchfound;
```

不要与代码矛盾。许多注释在写的时候与代码是一致的。但是后来由于修正错误，程序改变了，可是注释常常还保持着原来的样子，从而导致注释与代码的脱节。这很可能是本章开始的那个例子的合理解释。

无论产生脱节的原因何在，注释与代码矛盾总会使人感到困惑。由于误把错误注释当真，常常使许多实际查错工作耽误了大量时间。所以，当你改变代码时，一定要注意保证其中的注释是准确的。

注释不仅需要与代码保持一致，更应该能够支持它。下面的例子里的注释是正确的，它正确地解释了后面两行的用途。但细看又会发现它与代码矛盾，注释中谈的是换行，而代码中说的则是空格：

```
    time(&now);
    strcpy(date, ctime(&now));
    /* get rid of trailing newline character copied from ctime */
    i = 0;
    while(date[i] >= ' ') i++;
    date[i] = 0;
```

一个可能的改进是采用惯用法重写代码：

```
    time(&now);
    strcpy(date, ctime(&now));
    /* get rid of trailing newline character copied from ctime */
    for (i = 0; date[i] != '\n'; i++)
        ;
    date[i] = '\0';
```

现在注释和代码一致了。但是这两者都还可以进一步改进，应该写得更直截了当些。这里要解决的问题就是删除函数ctime返回时放在字符串最后的换行字符。注释里应该说明这个情况，代码也应该是做这件事：

```
time(&now);
strcpy(date, ctime(&now));
/* ctime() puts newline at end of string; delete it */
date[strlen(date)-1] = '\0';
```

最后这个表达式是在C语言里截去字符串最后字符的习惯写法。现在代码变短了，注释也支持它，解释了为什么这个语句需要做。

澄清情况，不要添乱。注释应该在困难的地方尽量帮助读者，而不是给他们设置障碍。下面的例子中遵循了我们为函数写注释的建议，解释函数不寻常的特征。但是这里的函数是 strcmp，这是个标准的东西，具有人们熟悉的界面，它的不寻常特性对手头工作来说根本不重要：

```
?    int strcmp(char *s1, char *s2)
?    /* string comparison routine returns -1 if s1 is */
?    /* above s2 in an ascending order list, 0 if equal */
?    /* 1 if s1 below s2 */
?    {
?        while(*s1==*s2)  {
?            if(*s1=='\0')  return(0);
?            s1++;
?            s2++;
?        }
?        if(*s1>*s2) return(1);
?        return(-1);
?    }
```

如果真的需要用许多话来解释出了什么事，这可能就说明了有关代码应该重写。上面的代码可以改进，但其中真正的问题在注释，它几乎有代码本身那么长，而且也非常含糊（例如，什么是这里的above？）。我们很难说这段代码难以理解，但是，由于它实现的是一个标准函数，它的注释就应该在这方面有所帮助，应该概述其行为，告诉我们哪里有它原来的定义。下面的注释是合适的：

```
/* strcmp: return < 0 if s1<s2, > 0 if s1>s2, 0 if equal */
/*         ANSI C, section 4.11.4.2 */
int strcmp(const char *s1, const char *s2)
{
    ...
}
```

学生常被告之应该注释所有的内容。职业程序员也常被要求注释他们的所有代码。但是，应该看到，盲目遵守这些规则的结果却可能是丢掉了注释的真谛。注释是一种工具，它的作用就是帮助读者理解程序中的某些部分，而这些部分的意义不容易通过代码本身直接看到。我们应该尽可能地把代码写得容易理解。在这方面你做得越好，需要写的注释就越少。好的代码需要的注释远远少于差的代码。

练习1-11 评论下面的注释：

```
?    void dict::insert(string& w)
?    // returns 1 if w in dictionary, otherwise returns 0


?    if (n > MAX || n % 2 > 0) // test for even number


?    // Write a message
?    // Add to line counter for each line written
?
?    void write_message()
?    {
?        // increment line counter
?        line_number = line_number + 1;
?        fprintf(fout, "%d  %s\n%d  %s\n%d  %s\n",
?            line_number, HEADER,
```

```
?            line_number + 1, BODY,
?            line_number + 2, TRAILER);
?        // increment line counter
?        line_number = line_number + 2;
?    }
```

1.7 为何对此费心

在这一章里，我们谈论的主要问题是程序设计的风格：具有说明性的名字、清晰的表达式、直截了当的控制流、可读的代码和注释，以及在追求这些内容时一致地使用某些规则和惯用法的重要性。没人会争辩说这些是不好的。

但是，为什么要为风格而煞费苦心？只要程序能运行，谁管它看起来是什么样子？把它弄得漂亮点是不是花费了太多时间？这些规则难道没有随意性吗？

我们的回答是：书写良好的代码更容易阅读和理解，几乎可以保证其中的错误更少。进一步说，它们通常比那些马马虎虎地堆起来的、没有仔细推敲过的代码更短小。在这个拼命要把代码送出门、去赶上最后期限的时代，人们很容易把风格丢在一旁，让将来去管它们吧。但是，这很可能是一个代价非常昂贵的决定。本章的一些例子说明了，如果对好风格问题重视不够，程序中哪些方面可能出毛病。草率的代码是很坏的代码，它不仅难看、难读，而且经常崩溃。

这里最关键的结论是：好风格应该成为一种习惯。如果你在开始写代码时就关心风格问题，如果你花时间去审视和改进它，你将会逐渐养成一种好的编程习惯。一旦这种习惯变成自动的东西，你的潜意识就会帮你照料许多细节问题，甚至你在工作压力下写出的代码也会更好。

补充阅读

就像我们在本章开始时说的，写出好的代码与书写好的英文有许多共同之处。 Strunk和White的《风格的要素》(The Elements of Style，Allyn & Bacon)仍然是关于如何写好英文的最好的简短的书。

本章采用了Brian Kernighan和P. J. Plauger的《程序设计风格的要素》(The Elements of Programming Style，McGraw-Hill，1978)中的方式。 Steve Maguire的《写可靠的代码》(Writing Solid Code，Microsoft Press，1993)是有关程序设计各方面的忠告的一本佳作。 Steve McConnell的《完整编程》(Code Complete，Microsoft Press，1993)和Peter van der Linden的《熟练的C程序设计：深入C的奥密》(Expert C Programming: Deep C Secret，Prentice Hall，1994)中都有一些关于程序风格的有益讨论。