

在程序里我们始终有话要说，但是所有已知的语言都无法表述得很好。

在前章中我们用过#define与#include指令，但没有深入讨论它们。这些指令，以及我们还没有学到的指令，都是由预处理器处理的。预处理器是一个小软件，它可以在编译前编辑C程序。C语言（和C++语言）因为依赖预处理器而不同于其他的编程语言。

预处理器是一个强大的工具，但它同时也可能是许多难以发现的错误的根源。同时，预处理器也经常被错误地用来编写出一些几乎不可能读懂的程序。尽管有些C程序员十分依赖于预处理器，我依然建议适度地使用它，就像许多其他生活中的事物一样。现代的C语言编程风格呼吁减少对于处理器的依赖。在C++中，对语言的改变使得可以更进一步限制预处理器的使用。

本章首先会描述预处理器的工作过程（14.1节），并且给出一些会影响全部预处理指令的通用规则（14.2节）。14.3节和14.4节涵盖介绍预处理器最主要两种能力：宏定义和条件编译。（而处理器另外一个主要功能（即文件包含）将留到第15章再进行详细介绍。）14.5节讨论较少用到的预处理指令：#error、#line和#pragma。

14.1 预处理器的工作方式

预处理器的行为是由指令控制的。这些指令是由#字符开头的一些命令。我们已经在前几章中遇见过其中两种指令，#define和#include。

#define指令定义了一个宏——用来代表其他东西的一个名字，通常是某一类型的常量。预处理器会通过将宏的名字和它的定义存储在一起响应#define指令。当这个宏在后面的程序中使用到时，预处理器“扩展”了宏，将宏替换为它所定义的值。

#include指令告诉预处理器打开一个特定的文件，将它的内容作为正在编译的文件的一部分“包含”进来。例如，下面这行指令

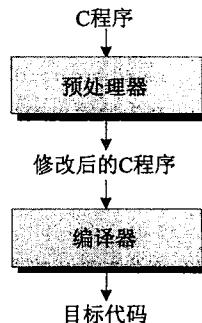
```
#include <stdio.h>
```

指示预处理器打开一个名字为stdio.h的文件，并将它的内容加到当前的程序中。（stdio.h包含了C语言标准输入/输出函数的原型。）

右图说明了预处理器在编译过程中的作用。预处理器的输入是一个C语言程序，程序可能包含指令。预处理器会执行这些指令，并在处理过程中删除这些指令。预处理器的输出是另一个程序：原程序的一个编辑后的版本，不再包含指令。预处理器的输出被直接交给编译器，编译器检查程序是否有错误，并经程序翻译为目标代码（机器指令）。

为了展现预处理器的作用，我们将它应用于2.6节的程序celsius.c。下面是原来的程序：

```
/* Converts a Fahrenheit temperature to Celsius */
```



```
#include <stdio.h>

#define FREEZING_PT 32.0
#define SCALE_FACTOR (5.0 / 9.0)

main()
{
    float fahrenheit, celsius;
    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);

    Celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
    printf("Celsius equivalent is: %.1f\n", celsius);

    return 0;
}
```

274

预处理结束后，程序是下面的样子：

空行
空行

从stdio.h中引入的行

```
空行
空行
空行
空行
空行
main()
{
    float fahrenheit, celsius;

    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);

    celsius = (fahrenheit - 32.0) * (5.0 / 9.0);

    printf("Celsius equivalent is: %.1f\n", celsius);

    return 0;
}
```

预处理器通过把stdio.h的内容加入来响应#include指令。由于长度的原因，这里没有将stdio.h的内容显示出来。预处理器也删除了#define指令，并且替换了该文件中稍后出现在任何位置上的FREEZING_PT和SCALE_FACTOR。请注意预处理器并没有删除包含指令的行，而是简单地将它们替换为空。

正如这个例子所展示的那样，预处理器不仅仅是执行了指令，还做了一些其他的事情。特别值得注意的是，它将每一处注释都替换为一个空格字符。有一些预处理器还会进一步删除不必要的空白字符，包括在每一行开始用于缩进的空格符和制表符。

在C语言较早的时期，预处理器是一个单独的程序，并将它的输出提供给编译器。如今，预处理器通常和编译器集成在一起（为了提高编译的速度）。然而，我们仍然将它们认为是不同的程序。实际上，大部分C编译器提供一些方法，使用户可以看到预处理器的输出。一些编译器在打开特定的选项时（UNIX环境下通常是-P）仅产生预处理器的输出。其他一些编译器会提供一个独立的程序，这个程序的工作与集成的预处理器一致。如果需要更多的信息，可以查看你使用的编译器的文档。

275

注意，预处理器仅知道少量C语言的规则。因此，它在执行指令时非常有可能产生非法的程序。经常是源程序看起来没问题，使错误难以查找。对于较复杂的程序，检查预处理器的输出可能是找到这类错误的有效途径。

14.2 预处理指令

大多数预处理指令属于下面3种类型：

- 宏定义。#define指令定义一个宏，#undef指令删除一个宏定义。
- 文件包含。#include指令导致一个指定文件的内容被包含到程序中。
- 条件编译。#if、#ifdef、#ifndef、#elif、#else和#endif指令可以根据编译器可以测试的条件来将一段文本块包含到程序中或排除在程序之外。

剩下的#error、#line和#pragma指令是更特殊的指令，较少用到。本章将深入研究预处理指令。唯一一个我们不会在这里详细讨论的指令是#include，因为它会在15.2节介绍。

在进一步讨论之前，先来看几条应用于所有指令规则：

- 指令都以#开始。#符号不需要在一行的行首，只要它之前只有空白字符就行。在#后是指令名，接着是指令所需要的其他信息。
 - 在指令的符号之间可以插入任意数量的空格或横向制表符。例如，下面的指令是合法的：
- ```
define N 100
```
- 指令总是在第一个换行符处结束，除非明确地指明要继续。如果想在下一行继续指令，我们必须在当前行的末尾使用\字符。例如，下面的指令定义了一个宏来表示硬盘的容量，按字节计算：

```
#define DISK_CAPACITY (SIDES *
 TRACK_PER_SIDE *\br/> SECTORS_PER_TRACK *\br/> BYTES_PER_SECTOR)
```

276

- 指令可以出现在程序中任何地方。我们通常将#define和#include指令放在文件的开始，其他指令则放在后面，甚至在函数定义的中间。
- 注释可以与指令放在同一行。实际上，在一个宏定义的后面加一个注释来解释宏的意义是一种比较好的习惯：

```
#define FREEZING_PT 32.0 /* Freezing point of water */
```

## 14.3 宏定义

我们从第2章以来使用的宏被称为简单的宏，它们没有参数。预编译器也支持带参数的宏。本节会先讨论简单的宏，然后再讨论带参数的宏。在分别讨论它们之后，我们会研究一下二者共同的特性。

### 14.3.1 简单的宏

简单的宏定义有如下格式：

[#define指令（简单的宏）] #define 标识符 替换列表

替换列表是一系列的C语言记号，包括标识符、关键字、数、字符串常量、字符串字面量、运算符和标点符号。当预处理器遇到一个宏定义时，会做一个“标识符”代表“替换列表”的记录。在文件后面的内容中，不管标识符在任何位置出现，预处理器都会用替换列表代替它。



不要在宏定义中放置任何额外的符号，否则它们会被作为替换列表的一部分。

一种常见的错误是在宏定义中使用 = :

```
#define N = 100 /*** WRONG ***/
```

```
int a[N]; /* 会成为 int a[= 100]; */
```

在上面的例子中，我们（错误地）把N定义成一对记号（= 和100）。

在宏定义的末尾使用分号结尾是另一个常见错误：

```
#define N 100; /*** WRONG ***/
int a[N]; /* become int a[100;]; */
```

这里N被定义为100和;两个记号。

277

在一个宏定义中，编译器可以检测到绝大多数由多余符号所导致的错误。但不幸的是，编译器会将每一处使用这个宏的地方标为错误，而不会直接找到错误的根源——宏定义本身，因为宏定义已经被预处理器删除了。

**Q&A** 简单的宏主要用来定义那些被Kernighan和Ritchie称为“明示常量”(manifest constant)的东西。使用宏，我们可以给数值、字符和字符串命名。

```
#define STE_LEN 80
#define TRUE 1
#define FALSE 0
#define PI 3.14159
#define CR '\r'
#define EOS '\0'
```

使用#define来为常量命名有许多显著的优点：

- 程序会更易读。一个认真选择的名字可以帮助读者理解常量的意义。否则，程序将包含大量的“魔法数”，使读者难以理解。
- 程序会更易于修改。我们仅需要改变一个宏定义，就可以改变整个程序中出现的所有该常量的值。“硬编码的”常量会更难于修改，特别是有时候当他们以稍微不同的形式出现时。（例如，如果一个程序包含一个长度为100的数组，它可能会包含一个从0到99的循环。如果我们只是试图找到所有程序中出现的100，那么就会漏掉99。）
- 可以帮助避免前后不一致或键盘输入错误。假如数值常量3.14159在程序中大量出现，它可能会被意外地写成3.1416或3.14195。

虽然简单的宏常用于定义常量名，但是它们还有其他应用。

- 可以对C语法做小的修改。实际上，我们可以通过定义宏的方式给C语言符号添加别名，从而改变C语言的语法。例如，对于习惯使用Pascal的begin和end（而不是C语言的{和}）的程序员，可以定义下面的宏：

```
#define BEGIN {
#define END }
```

我们甚至可以发明自己的语言。例如，我们可以创建一个LOOP“语句”，来实现一个无限循环：

```
#define LOOP for (;;)
```

当然，改变C语言的语法通常不是个好主意，因为它会使程序很难被其他程序员所理解。

278

- 对类型重命名。在5.2节中，我们通过重命名int创建了一个Boolean类型：

```
#define BOOL int
```

虽然有些程序员会使用宏定义的方式来实现此目的，但类型定义（7.6节）仍然是定义新类型的的最佳方法。

- 控制条件编译。如将在14.4节中看到的那样，宏在控制条件编译中起重要的作用。例如，在程序中出现的宏定义可能表明需要将程序在“调试模式”下进行编译，来使用额外的语句输出调试信息：

```
#define DEBUG
```

这里顺便提一下，如上面的例子所示，宏定义中的替换列表为空是合法的。

当宏作为常量使用时，C程序员习惯在名字中只使用大写字母。但是并没有如何将用于其他目的的宏大写的统一做法。由于宏（特别是带参数的宏）可能是程序中错误的来源，所以一些程序员更喜欢使用大写字母来引起注意。其他人则倾向于小写，即按照Kernighan和Ritchie编写的*The C Programming Language*一书中的样式。

### 14.3.2 带参数的宏

带参数的宏定义有如下格式：

**[#define指令—带参数的宏]**      **#define 标识符 ( $x_1, x_2, \dots, x_n$ ) 替换列表**

其中 $x_1, x_2, \dots, x_n$ 是标识符（宏的参数）。这些参数可以在替换列表中根据需要出现任意次。



在宏的名字和左括号之间必须没有空格。如果有空格，预处理器会认为是在定义一个简单的宏，其中 $(x_1, x_2, \dots, x_n)$ 是替换列表的一部分。

当预处理器遇到一个带参数的宏，会将定义存储起来以便后面使用。在后面的程序中，如果任何地方出现了标识符 $(y_1, y_2, \dots, y_n)$ 格式的宏调用（其中 $y_1, y_2, \dots, y_n$ 是一系列标记），预处理器会使用替换列表替代，并使用 $y_1$ 替换 $x_1$ ， $y_2$ 替换 $x_2$ ，依此类推。

例如，假定我们定义了如下的宏：

279      **#define MAX(x,y) ((x)>(y) ? (x) : (y))  
#define IS\_EVEN(n) ((n)%2==0)**

现在如果后面的程序中有如下语句：

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

预处理器会将这些行替换为

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

如这个例子所显示的，带参数的宏经常用来作为一些简单的函数使用。MAX类似一个从两个值中选取较大的值的函数。IS\_EVEN则类似于另一种函数，该函数当参数为偶数时返回1，否则返回0。

下面的例子是一个更复杂的宏：

```
#define TOUPPER(c) ('a'<=(c)&&(c)<='z'? (c)-'a'+'A':(c))
```

这个宏检测一个字符c是否在'a'与'z'之间。如果在的话，这个宏会用'c'减去'a'再加上'A'，来计算出c所对应的大写字母。如果c不在这个范围，就保留原来的c。像这样的字符处理的宏非常有用，所以C语言库在<ctype.h>（>23.4节）中提供了大量的类似的宏。其中之一就是toupper，与我们上面的TOUPPER例子作用一致（但会更高效，可移植性也更好）。

带参数的宏可以包含空的参数列表，如下例所示：

```
#define getchar() getc(stdin)
```

空的参数列表不是一定确实需要，但可以使getchar更像一个函数。（没错，这就是<stdio.h>中的getchar，getchar的确是个宏，不是函数——虽然它的功能像个函数。）

使用带参数的宏替代实际的函数有两个优点：

- 程序可能会稍微快些。一个函数调用在执行时通常会有些额外开销——存储上下文信息、复制参数的值等。而一个宏的调用则没有这些运行开销。
- 宏会更“通用”。与函数的参数不同，宏的参数没有类型。因此，只要预处理后的程序依然是合法的，宏可以接受任何类型的参数。例如，我们可以使用MAX宏从两个数中选出较大的一个，数的类型可以是int, long int, float, double等等。

但是带参数的宏也有一些缺点。

- 编译后的代码通常会变大。每一处宏调用都会导致插入宏的替换列表，由此导致程序的源代码增加（因此编译后的代码变大）。宏使用得越频繁，这种效果就越明显。当宏调用嵌套时，这个问题会相互叠加从而使程序更加复杂。思考一下，如果我们用MAX宏来找出3个数中最大的数会怎样？

```
n = MAX(i, MAX(j, k));
```

下面是预处理后的这条语句：

```
n=((i)>((j)>(k)?(j):(k))?(i):(((j)>(k)?(j):(k))));
```

- 宏参数没有类型检查。当一个函数被调用时，编译器会检查每一个参数来确认它们是否是正确的类型。如果不是，或者将参数转换成正确的类型，或者由编译器产生一个出错信息。预处理器不会检查宏参数的类型，也不会进行类型转换。
- 无法用一个指针来指向一个宏。如在17.7节中将看到的，C语言允许指针指向函数。这一概念在特定的编程条件下非常有用。宏会在预处理过程中被删除，所以不存在类似的“指向宏的指针”。因此，宏不能用于处理这些情况。
- 宏可能会不止一次地计算它的参数。函数对它的参数只会计算一次，而宏可能会计算两次甚至更多次。如果参数有副作用，多次计算参数的值可能会产生意外的结果。考虑下面的例子，其中MAX的一个参数有副作用：

```
n = MAX(i++, j);
```

下面是这条语句在预处理之后的结果：

```
n = ((i++)>(j)?(i++):(j));
```

如果i大于j，那么i可能会被（错误地）增加了两次，同时n可能被赋予了错误的值。



由于多次计算宏的参数而导致的错误可能非常难于发现，因为宏调用和函数调用看起来是一样的。更糟糕的是，这类宏可能在大多数情况下正常工作，仅在特定参数有副作用时失效。为了自保护，最好避免使用带有副作用的参数。

带参数的宏不仅适用于模拟函数调用。他们特别经常被作为模板，来处理我们经常要重复书写的代码段。如果我们已经写烦了语句

```
printf("%d\n", x);
```

因为每次要显示一个整数x都要使用它。我们可以定义下面的宏，使显示整数变得简单些：

```
#define PRINT_INT(x) printf("%d\n", x)
```

一旦定义了PRINT\_INT，预处理器会将这行

```
PRINT_INT(i/j);
```

转换为

```
printf("%d\n", i/j);
```

### 14.3.3 #运算符

宏定义可以包含两个运算符：#和##。编译器不会识别这两种运算符相反，它们会在预处理时被执行。

#运算符将一个宏的参数转换为字符串字面量。它仅允许出现在带参数的宏的替换列表中。

**Q&A**（一些C程序员将#操作理解为“stringization（字符串化）”；其他人则认为这实在是对英语的滥用。）

#运算符有大量的用途，这里只来讨论其中的一种。假设我们决定在调试过程中使用PRINT\_INT宏作为一个便捷的方法，来输出一个整型变量或表达式的值。#运算符可以使PRINT\_INT为每个输出的值添加标签。下面是改进后的PRINT\_INT：

```
#define PRINT_INT(x) printf("#x" " = %d\n", x)
```

x之前的#运算符通知预处理器根据PRINT\_INT的参数创建一个字符串字面量。因此，调用PRINT\_INT(i/j)；

会变为

```
printf("i/j" " = %d\n", i/j);
```

在C语言中相邻的字符串字面量会被合并，因此上边的语句等价于：

```
printf("i/j = %d\n", i/j);
```

当程序执行时，printf函数会同时显示表达式i/j和它的值。例如，如果i是11，j是2的话，输出为

```
i/j = 5
```

#### 14.3.4 ##运算符

##运算符可以将两个记号（例如标识符）“粘”在一起，成为一个记号。（无需惊讶，##运算符被称为“记号粘合”。）如果其中一个操作数是宏参数，“粘合”会在当形式参数被相应的实际参数替换后发生。考虑下面的宏：

```
#define MK_ID(n) i##n
```

当MK\_ID被调用时（比如MK\_ID(1)），预处理器首先使用自变量（这个例子中是1）替换参数n。接着，预处理器将i和1连接成为一个记号(i1)。下面的声明使用MK\_ID创建了3个标识符：

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```

预处理后声明变为：

```
int i1, i2, i3;
```

##运算符不属于预处理器经常使用的特性。实际上，想找到一些使用它的情况是比较困难的。为了找到一个有实际意义的##的应用，我们来重新思考前面提到过的MAX宏。如我们所见，当MAX的参数有副作用时会无法正常工作。一种解决方法是用MAX宏来写一个max函数。遗憾的是，往往一个max函数是不够的。我们可能需要一个实际参数是int值的max函数，还需要参数为float值的max函数，等等。除了实际参数的类型和返回值的类型之外，这些函数都一样。因此，这样定义每一个函数似乎是个很蠢的做法。

解决的办法是定义一个宏，并使它展开后成为max函数的定义。宏会有唯一的参数type，它表示形式参数和返回值的类型。这里还有个问题，如果我们是用宏来创建多个max函数，程序将无法编译。（C语言不允许在同一文件中出现两个同名的函数。）为了解决这个问题，我们是用##运算符为每个版本的max函数构造不同的名字。下面是宏的显示形式：

```
#define GENERIC_MAX (type) \
type type##_max(type x, type y) \
{ \
 return x > y ? x : y; \
}
```

请注意宏的定义中是如何将type和\_max相连来形成新函数名的。

现在，假如我们需要一个针对float值的max函数。下面是如何使用GENERIC\_MAX宏来定义函数：

```
GENERIC_MAX(float)
```

预处理器会将这行展开为下面的代码：

```
float float_max(float x, float y) { return x > y ? x : y; }
```

### 14.3.5 宏的通用属性

现在我们已经讨论过简单的宏和带参数的宏了，我们来看一下它们都需要遵守的规则。

283

- 宏的替换列表可以包含对另一个宏的调用。例如，我们可以用宏PI来定义宏TWO\_PI：

```
#define PI 3.14159
#define TWO_PI (2*PI)
```

当预处理器在后面的程序中遇到TWO\_PI时，会将它替换成(2\*PI)。接着，预处理器会重新检查替换列表，看它是否包含其他宏的调用（在这个例子中，调用了宏PI）。**Q&A**  
预处理器会不断重新检查替换列表，直到将所有的宏名字都替换掉为止。

- 预处理器只会替换完整的记号，而不会替换记号的片断。因此，预处理器会忽略嵌在标识符名、字符常量、字符串字面量之中的宏名。例如，假设程序含有如下代码行：

```
#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)
 puts ("Error : SIZE exceeded");
```

预处理后，这些代码行会变为：

```
int BUFFER_SIZE;

if (BUFFER_SIZE > 256)
 puts ("Error : SIZE exceeded");
```

标识符BUFFER\_SIZE和字符串"Error: SIZE exceeded"没有被预处理影响，虽然它们都包含SIZE。

- 一个宏定义的作用范围通常到出现这个宏的文件末尾。由于宏是由预处理器处理的，他们不遵从通常的范围规则。一个定义在函数中的宏并不是仅在函数内起作用，而是作用到文件末尾。
- 宏不可以被定义两遍，除非新的定义与旧的定义是一样的。小的间隔上的差异是允许的，但是宏的替换列表（和参数，如果有的话）中的记号都必须一致。
- 宏可以使用#define指令“取消定义”。#undef指令有如下形式：

[#undef指令] #undef 标识符

其中标识符是一个宏名。例如，指令

```
#undef N
```

284

会删除宏N当前的定义。（如果N没有被定义成一个宏，#undef指令没有任何作用。）

#undef指令的一个用途是取消一个宏的现有定义，以便于重新给出新的定义。

### 14.3.6 宏定义中的圆括号

在我们前面定义的宏的替换列表中有大量的圆括号。确实需要它们吗？答案是绝对需要。如果我们少用几个圆括号，宏可能有时会得到意料之外的——而且是不希望有的——结果。

对于在一个宏定义中哪里要加圆括号有两条规则要遵守。首先，如果宏的替换列表中有运算符，那么始终要将替换列表放在括号中：

```
#define TWO_PI (2*3.14159)
```

其次，如果宏有参数，每次参数在替换列表中出现时都要放在圆括号中：

```
#define SCALE(x) ((x)*10)
```

没有括号的话，我们将无法确保编译器会将替换列表和参数作为完整的表达式。编译器可能会

不按我们期望的方式应用运算符的优先级和结合性规则。

为了展示为替换列表添加圆括号的重要性，考虑下面的宏定义，其中的替换列表没有添加圆括号：

```
#define TWO_PI 2*3.14159
/* 需要给替换列表加圆括号 */
```

在预处理时，语句

```
conversion_factor = 360/TWO_PI;
```

变为

```
conversion_factor = 360/2*3.14159;
```

除法会在乘法之前执行，产生的结果并不是期望的结果。

当宏有参数时，仅给替换列表添加圆括号是不够的。参数的每一次出现都要添加圆括号。例如，假设SCALE定义如下：

```
#define SCALE(x) (x*10) /* 需要给x添加括号 */
```

在预处理过程中，语句

```
j = SCALE(i+1);
```

**285** 变为

```
j = (i+1*10);
```

由于乘法的优先级比加法高，这条语句等价于

```
j = i+10;
```

当然，我们希望的是

```
j = (i+1)*10;
```



在宏定义中缺少圆括号会导致C语言中最让人讨厌的错误。程序通常仍然可以编译通过，而且宏似乎也可以工作，仅在少数情况下会出错。

### 14.3.7 创建较长的宏

在创建较长的宏时，逗号运算符会十分有用。特别是可以使用逗号运算符来使替换列表包含一系列表达式。例如，下面的宏会读入一个字符串，再把字符串显示出来：

```
#define ECHO(s) (get(s), puts(s))
```

gets函数和puts函数的调用都是表达式，因此使用逗号运算符连接它们是合法的。我们甚至可以把ECHO宏当作一个函数来使用：

```
ECHO(str); /* 替换为 (get(str), puts(str)); */
```

除了使用逗号运算符，我们也许还可以将gets函数和puts函数的调用放在大括号中形成复合语句：

```
#define ECHO(s) { gets(s); puts(s); }
```

遗憾的是，这种方式并不奏效。假如我们将ECHO宏用于下面的if语句：

```
if (echo_flag)
 ECHO(str);
else
 gets(str);
```

将ECHO宏替换会得到下面的结果：

```
if (echo_flag)
{ gets(str); puts(str); };
```

```
else
 gets(str);
```

编译器会将头两行作为完整的if语句：

```
if (echo_flag)
{ gets(str); puts(str); }
```

编译器会将跟在后面的分号作为空语句，并且对else子句产生出错信息，因为它不属于任何if语句。我们可以通过记住永远不要在ECHO宏后面加分号来解决这个问题。但是这样做会使程序看起来有些怪异。

逗号运算符可以解决ECHO宏的问题，但并不能解决所有宏的问题。假如一个宏需要包含一系列的语句，而不仅仅是一系列的表达式，这时逗号运算符就起不到帮助的作用了。因为它只能连接表达式，不能连接语句。解决的方法是将语句放在do循环中，并将条件设置为假：

```
do { ... } while (0)
```

do循环必须始终随跟着一个分号，因此我们不会遇到在if语句中使用宏那样的问题了。为了看到这个技巧（嗯，应该说是技术）的实际作用，让我们将它用于ECHO宏中：

```
#define ECHO(s) \
 do { \
 gets (s) ; \
 puts (s) ; \
 } while (0)
```

当使用ECHO宏时，一定要加分号：

```
ECHO(str);
/* becomes do { gets(str); puts(str); } while (0); */
```

### 14.3.8 预定义宏

在C语言中预定义了一些有用的宏，见表14.1。这些宏主要是提供当前编译的信息。宏\_\_LINE\_\_和\_\_STDC\_\_是整型常量，其他3个宏是字符串字面量。我们在本章的后面会用到\_\_STDC\_\_宏，因此这里将重点放在其他的宏上。

表14-1 预定义宏

| 名    字   | 描    述                 |
|----------|------------------------|
| __LINE__ | 被编译的文件的行数              |
| __FILE__ | 被编译的文件的名字              |
| __DATE__ | 编译的日期（格式“MMM dd yyyy”） |
| __TIME__ | 编译的时间（格式“hh:mm:ss”）    |
| __STDC__ | 如果编译器接受标准C，那么值为1       |

\_\_DATE\_\_宏和\_\_TIME\_\_宏指明程序编译的时间。例如，假设程序以下面的语句开始：

```
printf("Wacky Windows (c) 1996 Wacky Software, Inc.\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

每次程序开始执行，程序都会显示下面两行：

```
Wacky Windows (c) 1996 Wacky Software, Inc.
Compiled on Dec 23 1996 at 22:18:48
```

这样的信息可以帮助区分同一个程序的不同版本。

我们可以使用\_\_LINE\_\_宏和\_\_FILE\_\_宏来找到错误。考虑下面这个检测被零除的除法的发生位置的问题。当一个C程序因为被零除而导致中止时，通常没有信息指明哪条除法运算导致错误。下面的宏可以帮助我们查明错误的根源：

```
#define CHECK_ZERO(divisor) \
 if (divisor == 0) \
```

286

287

```
printf("**** Attempt to divide by zero on line %d\n"
 "of file %s ****\n", __LINE__, __FILE__)
```

CHECK\_ZERO宏应该在除法运算前被调用：

```
CHECK_ZERO(j);
k = i / j;
```

如果j是0，会显示出如下形式的信息：

```
*** Attempt to divide by zero on line 9 of file FOO.c ***
```

类似这样的错误检测的宏非常有用。实际上，C语言库提供了一个通用的、用于错误检测的宏——assert宏（>24.1节）。

## 14.4 条件编译

C语言的预处理器可以识别大量用于支持条件编译的指令。条件编译是指根据预处理器所执行的测试结果来包含或排除程序的片断。

### 14.4.1 #if 指令和#endif 指令

假如我们正在调试一个程序。我们想要程序显示出特定变量的值，因此将printf函数调用添加到程序中重要的部分。一旦找到错误，经常需要保留这些printf函数调用，以备以后使用。条件编译允许我们保留这些调用，但是让编译器忽略它们。

下面是我们需要采取的方式。首先定义一个宏，并给它一个非0的值：

```
#define DEBUG 1
```

288

宏的名字并不重要。接下来，我们要在每组printf函数调用的前后加上#if和#endif：

```
#if DEBUG
printf("Value of i : %d\n", i);
printf("Value of j : %d\n", j);
#endif
```

在于处理过程中，#if指令会测试DEBUG的值。由于DEBUG的值非0，因此预处理器会将这两个printf函数调用保留在程序中（但#endif行会消失）。如果我们将DEBUG的值改为0并重新编译程序，预处理器则会将这4行代码都消失。编译器将不会看到这些printf函数调用，所以这些调用就不会在目标代码中占用空间，也不会在程序运行时浪费时间。我们可以将#endif保留在最终的程序中，这样如果程序在运行时出错，可以继续产生这些诊断信息（将DEBUG改为1并重新编译）。

一般来说，#if指令的格式如下：

**[#if指令]      #if 常量表达式**

#endif指令则更简单：

**[#endif指令]      #endif**

**Q&A**当预处理器遇到#if指令时，会计算常量表达式。如果表达式的值为0，那么在#if与#endif之间的行将在预处理过程中从程序中删除。否则，这些在#if和#endif之间的行会被保留在程序中，并继续被编译器处理——这时#if和#endif对程序没有任何影响。

对于没有定义过的标识符，#if指令会把它当作是值为0的宏对待。因此，如果省略DEBUG的定义，测试

```
#if DEBUG
```

会失败（但不会产生出错消息），而测试

```
#if !DEBUG
```

会成功。

#### 14.4.2 `defined` 运算符

14.3节中介绍过运算符#和##，还有另外一个运算符——defined，它仅用于预处理器。当defined应用于标识符时，如果标识符是一个定义过的宏返回1，否则返回0。#defined运算符通常与#if指令结合使用，允许写成

```
#if defined(DEBUG)
...
#endif
```

仅当DEBUG被定义成宏时，#if和#endif之间的代码会被保留在程序中。DEBUG两侧的括号不是必需的，因此可以简单写成

```
#if defined DEBUG
```

由于defined运算符仅检测DEBUG是否被定义为宏，所以不需要给DEBUG一个值：

```
#define DEBUG
```

#### 14.4.3 `#ifdef` 指令和`#ifndef` 指令

#ifdef指令测试一个标识符是否已经定义为宏；

[#ifdef指令]      #ifdef 标识符

#ifdef指令的使用与#if指令类似：

```
#ifdef 标识符
当标识符被定义为宏时需要包含代码
#endif
```

严格地说，Q&A并不需要#ifdef，因为我们可以组合#if指令和defined运算符来达到相同的效果。换而言之，指令

#ifdef 标识符

等价于

```
#if defined(标识符)
```

#ifndef指令与#ifdef指令类似，但是测试的是标识符是否没有被定义为宏：

[#ifndef指令]      #ifndef 标识符

写成

```
#ifndef 标识符
```

等价于写成

```
#if !defined(标识符)
```

#### 14.4.4 `#elif` 指令和`#else` 指令

#if指令、#ifdef指令和#ifndef指令可以像普通的if语句那样嵌套使用。当发生嵌套时，最好随着嵌套层次的增加而增加缩进。一些程序员对每一个#endif都加注释，来指明是对应于哪个条件测试的#if指令：

```
#if DEBUG
...
#endif /* DEBUG */
```

这种方法可以帮助读者更方便地找到起始的#if指令。

为了提供更多的便利，预处理器提供了#elif和#else指令：

289

290

|           |           |
|-----------|-----------|
| [#elif指令] | #elif 表达式 |
| [#else指令] | #else     |

#elif指令和#else指令可以与#if指令、#ifdef指令和#ifndef指令组合使用，来测试一系列条件：

```
#if 表达式1
当表达式1非0时需要包含的代码
#if 表达式2
当表达式1为0但表达式2非0时需要包含的代码
#else
其他情况下需要包含的代码
#endif
```

虽然上面的例子使用了#if指令，但#ifdef指令或#ifndef指令也可以替代使用。在#if指令和#endif指令之间可以有多个#elif指令，但最多只能有一个#else指令。

#### 14.4.5 使用条件编译

条件编译对于调试是非常方便的，但它并不仅限于此。下面是其他一些常见的应用：

- 编写在多台机器或多种操作系统之间可移植的程序。下面的例子中会根据WINDOWS、DOS或OS2是否被定义为宏，而将三组代码之一包含到程序中：

```
#if defined(WINDOWS)
...
#elif defined(DOS)
...
#elif defined(OS2)
...
#endif
```

一个程序中可能包含许多这样的#if指令。在程序的开头会定义这些宏之一（而且只有一个），由此选择了一个特定的操作系统。例如，定义OS2宏可以指明程序将运行在OS/2操作系统下。

- 编写可以使用不同的编译器进行编译的程序。不同的编译器经常用于不同的C语言版本，这些版本之间会有一些差异。一些会接受标准C，另外一些则不会。一些版本会包含针对特定机器的扩展，而其他的则或没有，或提供不同的扩展集。条件编译可以使程序适应于不同的编译器。思考编写这样一个程序，当使用标准C的编译器进行编译时，很可能被编译成功。\_STDC\_宏允许预处理器检测编译器是否支持标准C，如果不支持，我们可能必须修改程序的某些方面，尤其是有可能必须使用经典C的函数声明替代标准C的函数原型。对于每一处函数的声明，我们都可以使用下面的代码：

```
#if __STDC__
标准C函数原型
#else
经典C函数声明
#endif
```

- 为宏提供默认定义。条件编译使我们可以检测一个宏当前是否已经被定义了，如果没有，则提供一个默认的定义。例如，如果它还没有被定义的话，下面的代码会定义宏BUFFER\_SIZE：

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```
- 临时屏蔽包含注释的代码。由于在标准C中注释不能嵌套，因此不能直接“注释掉”包

含注释的代码。然而，可以使用`#if`指令替代：

```
#if 0
包含注释的代码行
#endif
```

**Q&A** 将代码以这种方式屏蔽掉经常称为“条件屏蔽”。

292

15.2节会讨论另外一种条件编译的常用用途：保护头文件以避免重复包含。

## 14.5 其他指令

在本章的最后，我们将简要地了解一下`#error`指令、`#line`指令和`#pragma`指令。这些指令有一个共同点：C语言的初学者不会经常使用它们。实际上，本书中的所有程序都不会使用它们，因此你可以放心地跳过本节。以后，当你准备成为一个C语言专家时，你会需要熟悉这些指令。

### 14.5.1 #error 指令

`#error`指令有如下格式：

[**#error 指令**]    [**#error 消息**]

其中，消息是一个C语言标记序列。如果预处理器遇到一个`#error`指令，它会显示一个出错消息，这个出错消息一定会包含消息。对于不同的编译器，出错消息的具体形式也可能会不一样。下面是一个典型的示例：

Error directive: 记号序列

碰到`#error`指令预示着一个严重的程序错误，大多数编译器会立即终止编译而不去找出其他错误。

`#error`指令通常与条件编译指令一起用于检测正常编译过程中不应出现的情况。例如，假定我们需要确保，一个程序在一台无法使用`int`类型来存储大于100 000的数的机器上不能编译。最大允许的`int`值用`INT_MAX`宏（>23.2节）表示，所以我们需要做的就是当`INT_MAX`宏不超过100 000时调用`#error`指令：

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

如果试图在一台整型以16位存储的机器上编译这个程序，将产生一条出错消息：

Error directive: int type is too small

`#error`指令通常会出现在`#if-#elif-#else`序列中的`#else`部分：

```
#if defined(WINDOWS)
...
#elif defined(DOS)
...
#elif defined(OS2)
...
#else
#error No operating system specified
#endif
```

293

### 14.5.2 #line 指令

`#line`指令是用来改变给程序行编号的方式的。（程序行通常是按1, 2, 3, …来编号的。）我们也可以使用这条指令使编译器认为它正在从一个有不同名字的文件中读取一个程序。

`#line`指令有两种形式。在一种形式中，指定一个行号：

**[#line 指令 (形式1)]**

#line n

*n*必须是介于1到32 767之间的整数。这条指令导致程序后面的行被编号为*n*、*n*+1、*n*+2等。

在#line的第二种形式中，需要同时指定行号和文件名：

**[#line指令 (形式2)]**

#line n "文件名"

指令后面的行会被认为是来自文件，行号由*n*开始。

#line指令的一种作用是改变\_\_LINE\_\_宏（可能还有\_\_FILE\_\_宏）的值。更重要的是，大多数编译器会使用来自#line指令的信息产生出错消息。例如，假设下列指令出现在文件foo.c的开头：

#line 10 "bar.c"

现在，假设编译器在foo.c的第5行监测到一个错误。出错消息会指向bar.c的第13行，而不是foo.c的第5行。（为什么是第13行呢？因为指令占据了foo.c的第1行，因此对foo.c的重新编号从第2行开始，并将这一行作为bar.c的第10行。）

乍一看，#line指令使人迷惑。为什么要使出错消息指向另一行，甚至是另一个文件呢？这样不是会使程序变得难以调试吗？

实际上，程序员并不经常使用#line指令。然而，它主要用于那些产生C代码作为输出的程序。**294** 程序yacc（Yet Another Compiler-Compiler）是其中著名的程序之一。yacc是一个UNIX工具，用于自动生成部分的编译器。在使用yacc之前，程序员需要准备一个包含yacc所需要的信息以及C代码段的文件。通过这个文件，yacc生成了一个C程序y.tab.c，并合并了程序员所提供的代码。程序员接着按照正常方法编译y.tab.c。通过在y.tab.c中插入#line指令，yacc会使编译器认为代码来自原始文件——也就是程序员写的那个文件。于是，任何编译y.tab.c时产生的出错消息会指向原始文件中的行，而不是y.tab.c中的行。其最终结果是：调试变得更容易，因为出错消息都指向程序员编写的文件，而不是（更复杂的）由yacc生成的文件。

### 14.5.3 #pragma 指令

#pragma指令为要求编译器执行某些特殊操作提供了一种方法。这条指令对非常大的程序或需要使用特定编译器的特殊功能的程序非常有用。

#pragma指令有如下形式：

**[#pragma指令]**

#pragma 记号

其中，记号是一般C语言的记号。#pragma指令通常只跟着一个记号，这个记号表示了一条编译器需要服从的命令。

一些编译器允许#pragma指令所包含的不仅是简单的命令。特别是有些编译器允许#pragma指令带参数：

#pragma data(heap\_size =&gt; 1000, stack\_size =&gt; 2000)

#pragma指令中出现的命令集在不同的编译器上是不一样的。你必须通过查阅你所使用的编译器的文档来了解哪些命令是可以使用的，以及这些命令的功能。顺便提一下，如果#pragma指令包含了无法识别的命令，编译器必须忽略这些#pragma指令，不允许产生出错消息。

---

## 问与答

问：我看到在有些程序中#单独占一行。这样是合法的吗？

答：是合法的。这就是所谓的空指令，它没有任何作用。一些程序员用空指令作为条件编译模块之间的间隔：

295

```
#if INT_MAX < 100000
#
#error int type is too small
#
#endif
```

当然，空行也可以。不过#可以帮助读者看出模块的范围。

问：我不清楚程序中哪些常量需要定义成宏。有没有一些可以参照的规则？(p.195)

答：一条首要的规则是每一个数字常量，如果不是0和1，就需要定义成宏。字符常量和字符串常量有一点复杂，因为使用宏来替换字符或字符串常量并不总是提高程序的可读性。我个人建议在下面的条件下使用宏来替代字符或字符串常量：（1）常量被不止一次地使用；（2）存在常量某天被修改的可能。根据第二条规则，我不会像这样使用宏：

```
#define NUL '\0'
```

虽然有些程序员会使用。

\*问：如果要被“字符串化”的参数包含" 或 \ 字符，#运算符会如何处理？(p.197)

答：它会将" 转换为 \"，\ 转换为 \\。考虑下面的宏：

```
#define STRINGIZE(x) #x
```

预处理器会将STRINGIZE("foo")替换为"\ "foo\" "。

\*问：我无法使下面的宏正常工作：

```
#define CONCAT(x,y) x##y
```

如期望那样，**CANCAT(a,b)**会给出ab，但**CONCAT(a,CONCAT(b,c))**会给出一个怪异的结果。

这是为什么？

答：感谢那些连Kernighan和Ritchie都承认“怪异”的规则，替换列表中依赖##的宏通常不能嵌套调用。

这里的问题在于CONCAT(a,CONCAT(b,c))不会按照“正常”的方式扩展——CONCAT(b,c)首先得出bc，然后CONCAT(a,bc)给出abc。C标准指明，在替换列表中，位于##运算符之前和之后的宏参数在替换时不被扩展，结果，CONCAT(a,CONCAT(b,c))扩展成aCONCAT(b,c)，而不会进一步扩展，因为没有名为aCONCAT的宏。

有一种办法可以解决这个问题，但不太好看。技巧是定义第二个宏，只是简单地调用第一个宏：

```
#define CONCAT2(x,y) CONCAT(x,y)
```

写成CONCAT2(a,CONCAT2(b,c))就会得到我们希望的结果。在扩展外面的CONCAT2调用时，预处理器将会同时扩展CONCAT2(b,c)。这里的区别在于CONCAT2的扩展列表不包含##。如果这个也不行，那也不用担心，这种问题并不是经常遇到的。

顺便提一下，#运算符也有同样的问题。如果#x出现在替换列表中，其中x是一个宏参数，其对应的实际参数也不会被扩展。因此，假设N是一个代表10的宏，且STR(x)包含替换列表#x，STR(N)扩展的结果为"N"，而不是"10"。解决的方法与我们在CONCAT例子中类似：定义第二个宏来调用STR。

\*问：如果预处理器再重新扫描时又发现了最初的宏名会如何处理呢？例如下面的例子：

```
#define N (2*M)
#define M (N+1)
```

i = N; /\* infinite loop? \*/

预处理器会将N替换为(2\*M)，接着将M替换为(N+1)。预处理器还会再次替换N，成为一个无限循环吗？(p.199)

答：一些早期的预处理器确实会进入无限循环，但符合标准C的预处理器则不会。按照C语言标准，如果在扩展宏的过程中原先的宏名重复出现的话，宏名不会再次被替换。下面是预处理后i的赋值语句的形式：

i = (2 \* (N+1));

一些大胆的程序员会通过编写其名字与保留字或标准库中的函数名匹配的宏来利用这一行为。

296

以库函数sqrt为例。sqrt函数（>23.3.5节）计算参数的平方根，如果参数为负数则返回一个由实现定义的值，我们可能希望当参数为负数时返回0。由于sqrt是标准库函数，我们无法很容易地修改它。但是我们可以定义一个宏，使它在参数为负数时返回0：

```
#define sqrt(x) ((x)>0 ? sqrt(x) : 0)
```

**297** 预处理器会截获sqrt的调用，并将它替换成上面的条件表达式。在扫描宏的过程中条件表达式中的sqrt调用不会被替换，因此会被保留由编译器处理。

问：我觉得预处理器就是一个编辑器。它如何计算常量表达式呢？（p.202）

答：预处理器比你想的要复杂。它足够“了解”C语言，所以能够计算常量表达式。虽然它不会完全按照编译器的方式去做。（例如，预处理器认为所有未定义的名字的值为0。其他的差异太深奥，就不再深入了。）在实际使用中，预处理器常量表达式中的操作数通常为常量、表示常量的宏或defined运算符的应用。

问：为什么C提供#define指令和#ifndef指令，既然我们可以使用#if指令和#define运算符达到同样效果？（p.203）

答：#ifdef指令和#ifndef指令从20世纪70年代就在C语言中存在了，而defined运算符则是在80年代的标准化过程中加到C语言中的。因此，实际的问题是：为什么将defined运算符加到C语言中？答案就是defined增加了灵活性。我们现在可以使用#if和defined运算符来测试任意数量的宏，而不再是只能使用#define和#ifndef对一个宏进行测试。例如，下面的指令检查是否FOO和BAR被定义了而BAZ没有被定义：

```
#if defined(FOO) && defined(BAR) && !defined(BAZ)
```

问：我想编译一个还没有写完的程序，因此我“条件屏蔽”未完成的部分。我在开始的地方加了一条信息来提示自己以后将程序写完：

```
#if 0
Haven't finished this part yet.
...
#endif
```

为什么我在编译时会得到一条出错消息呢？预处理器不是简单地忽略#if指令和#endif指令之间的所有行吗？（p.205）

答：在#if指令和#endif指令之间的行必须由预处理记号（>附录A）组成。预处理记号类似于C语言记号（标识符、运算符、数等）。当预处理器试图将第一行分解为记号时，会遇到Haven（一个合法的标识符），接着是't'（一个非法的字符常量）。一些预处理器会跳过#if指令和#endif指令之间所有的行而不会检查预处理记号，但这些预处理器并没有严格遵守C语言标准的规则。

## 练习

### 14.3节

1. 编写宏来计算下面的值。

- (a) x的立方。
- (b) x除以4的余数。
- (c) 如果x与y的乘积小于100值为1，否则值为0。

你写的宏始终正常工作吗？如果不是，哪些参数会失败呢？

2. 编写一个宏NELEMS(a)来计算一个一维数组a中元素的个数。提示：使用sizeof运算符。

3. 假定DOUBLE是如下宏：

```
#define DOUBLE(x) 2*x
```

- (a) DOUBLE(1+2)的值是多少？
- (b) 4/DOUBLE(2)的值是多少？

298

- (c) 改正DOUBLE的定义。  
 4. 针对下面每一个宏，举例说明宏的问题，并提出修改方法。

(a) #define AVG(x,y) (x+y)/2  
 (b) #define AREA(x,y) (x)\*(y)

- \*5. 下面的宏定义有问题：

#define ABS(a) ((a)<0?-(a):a)

举例说明为什么ABS不能正常工作，并提出修改方法。你可以假定ABS的参数没有副作用。

6. 假定TOUPPER定义成下面的宏：

#define TOUPPER(c) ('a'<=(c)&&(c)<='z'? (c)-'a'+'A':(c))

假设s是一个字符串，i是一个int型变量。给出下面每个代码段所产生的输出。

(a) strcpy(s, "abcd");  
 i = 0;  
 putchar(TOUPPER(s[+i]));  
 (b) strcpy(s, "0123");  
 i = 0;  
 putchar(TOUPPER(s[+i]));

7. (a) 编写宏DISP(f,x)，使其扩展后调用printf函数来显示函数f的参数为x时的值。例如：

DISP(sqrt, 3.0);

应该扩展为

printf("sqrt(%g) = %g\n", 3.0, sqrt(3.0));

- (b) 编写宏DISP2(f,x,y)，类似DISP但应用于有两个参数的函数。

- \*8. 假定GENERIC\_MAX是如下宏：

```
#define GENERIC_MAX(type) \
type type##_max(type x, type y) \
{ \
 return x > y ? x : y; \
}
```

- (a) 写出GENERIC\_MAX(long)被预处理器扩展后的形式。

- (b) 解释为什么GENERIC\_MAX不能应用在像unsigned long这样的基本类型上？

- (c) 如何使GENERIC\_MAX对任何基本类型都可以正常工作？提示：不要改变GENERIC\_MAX的定义。

- \*9. 如果需要一个宏，使它展开后包含当前行号和文件名。换而言之，我们会写

const char \*str = LINE\_FILE;

扩展后为

const char \*str = "Line 10 of file foo.c";

其中foo.c是包含程序的文件，10是调用LINE\_FILE行的行号。警告：这个练习仅针对高级程序员。

尝试编写前请认真阅读“问与答”小节的内容！

299

#### 14.4节

10. 假定宏M有如下定义：

#define M 10

下面哪项测试会失败？

- (a) #if M  
 (b) #ifdef M  
 (c) #ifndef M  
 (d) #if defined(M)  
 (e) #if !defined(M)

11. (a) 指出下面的程序预处理后的形式。

```
#define N 100

void f(void);

main()
{
 f();
#define N
#undef f N
#endif
 return 0;
}

void f(void)
{
#if defined(N)
 printf("N is %d\n", N);
#else
 printf("N is undefined\n");
#endif
}
```

(b) 这个程序的输出是什么？

12. 指出下面的程序预处理后的形式。其中有几行可能会导致编译错误，请找出这些错误。

```
#define N = 10
#define INC(x) x+1
#define SUB (x,y) x - y
#define SQR(x) ((x)*(x))
#define CUBE(x) (SQR(x)*(x))
#define M1(x,y) x##y
#define M2(x,y) #x #y

main ()
{
 int a[N], i, j, k, m;
#define N
 i = j;
#else
 j = i;
#endif
 i = 10 * INC(j);
 i = SUB(j, k);
 i = SQR(SQR(j++));
 i = CUBE(j);
 i = M1(j, k);
 puts(M2(i, j));
#undef SQR
 i = SQR(j);
#define SQR
 i = SQR(j);
 return 0 ;
}
```

300

301

# 编写大规模程序

很难找到正确的时间单位来衡量计算机的发展。有些大教堂用了一个世纪才建成。  
你能想象一个壮丽辉煌的大程序也能花这么长的时间吗？

虽然某些C程序小得足够放入一个单独的文件中，但是大多数程序都不是这样的。程序由多个文件构成的原则更容易让人接受。本章将会看到一个由几个源文件（source file）以及一些通常的头文件（header file）组成的典型程序。源文件包含函数的定义和外部变量，而头文件包含可以在源文件之间共享的信息。15.1节讨论源文件，15.2节详细地介绍头文件，15.3节描述把程序分割成源文件和头文件的方法，15.4节说明如何“构建”（即编译和链接）由多个文件组成的程序，以及在改变程序的部分内容后如何进行“重新构建”。

## 15.1 源文件

到现在为止一直假设C程序是由单独一个文件组成的。事实上，可以把程序分割成一定数量的源文件。根据惯例，源文件的扩展名为.c。每个源文件包含程序的部分内容，主要是函数的定义和变量。一个源文件必须包含名为main的函数，此函数作为程序的起始点。

例如，假设打算编写一个简单计算器程序，用来计算按照逆波兰符号（Reverse Polish Notation, RPN）录入的整数表达式。所谓逆波兰符号是指运算符都跟在操作数的后边。如果用户录入表达式

30 5 - 7 \*

我们希望程序可以显示出此表达式的值（此例中值为175）。如果可以使程序逐个读入操作数和运算符，那么利用栈跟踪中间结果这样的方式计算逆波兰表达式是很容易的。如果程序读取数，就把此数压入栈。如果程序读取运算符，那么将从栈顶弹出两个数进行相应的运算，然后把结果压入栈。当程序执行到用户输入的末尾时，表达式的值将在栈中。例如，程序将按照下列方式计算表达式30 5 - 7 \*的值：

- (1) 把30压入栈。
- (2) 把5压入栈。
- (3) 从栈顶弹出两个数，30减去5，结果为25，然后把此结果压回到栈中。
- (4) 把7压入栈。
- (5) 从栈顶弹出两个数，两数相乘，然后把结果压回到栈中。

在这些步骤后，栈将包含表达式的值（即175）。

把这种策略转换为程序并不困难。程序的main函数将用循环来执行下列动作：

- 读取“记号”（数或运算符）。
- 如果记号是数，那么把它压入栈。
- 如果记号是运算符，那么从栈顶弹出它的操作数进行运算，然后把结果压入栈中。

当像这样把程序分割成文件时，将相关的函数和变量放入同一文件中是很有意义的。读取

记号的函数可能和任何需要用到记号的函数一起属于某个源文件（比如说token.c文件）。比如push函数、pop函数、make\_empty函数、is\_empty函数和is\_full函数这些与栈相关的函数可能都属于一个不同的stack.c文件。表示栈的变量也属于stack.c文件，而main函数则可以在另一个calc.c文件中。

把程序分裂成多个源文件有许多显著的优点：

- 把相关的函数和变量集合在单独一个文件中可以帮助明了程序的结构。
- 可以单独对每一个源文件进行编译。如果程序规模很大而且需要频繁改变（这一点在程序开发过程中是非常普遍的）的话，这种方法可以极大地节约时间。
- 当把函数集合在单独的源文件中时，会更容易在其他程序中重新使用这些函数。在示例中，把stack.c和token.c从main函数中分离出来使得在今后更容易重新使用栈函数和记号函数。

304

## 15.2 头文件

当把程序分割为几个源文件时，问题也随之产生了：某文件中的函数如何能调用定义在其他文件中的函数呢？函数如何能访问其他文件中的外部变量呢？两个文件如何能共享同一个宏定义或类型定义呢？答案取决于#include指令，此指令使得在任意数量的源文件中共享信息成为可能，其中，这些信息可以是函数原型、宏定义、类型定义等。

#include指令告诉预处理器打开指定的文件，并且把此文件的内容插入到当前文件中。因此，如果打算几个源文件可以访问相同的信息，那么将把此信息放入文件中，然后利用#include指令把文件的内容带进每个源文件中。把按照这种方式包含的文件称为是头文件（或者有时称为包含文件）。本节后将会更详细地讨论头文件。根据惯例，头文件的扩展名为.h。

注意：C标准使用术语“源文件”来指示程序员编写的全部文件，包括.c文件和.h文件。而这里“源文件”只是指.c文件。

### 15.2.1 #include 指令

#include指令有两种书写格式。第一种格式用于属于C语言自身库的头文件：

[#include指令格式1]      `#include <文件名>`

第二种格式用于所有其他头文件，也包含任何自己编写的文件：

[#include指令格式2]      `#include "文件名"`

编译器定位头文件的方式是两种格式间的细微差异。Q&A下面是大多数编译器遵循的规则：

- `#include <文件名>`：搜寻系统头文件所在的目录（或多个目录）。例如，在UNIX系统中，通常把系统头文件保存在目录/usr/include中。
- `#include "文件名"`：搜寻当前目录，然后搜寻系统头文件所在的目录（或多个目录）。通常可以改变搜寻头文件的位置，这种改变经常利用诸如-I路径这样的命令行选项来实现。



不要在包含自行编写的头文件时用尖括号：

`#include <myheader.h> /* WRONG */`

因为预处理器将可能在保存系统头文件的地方寻找 myheader.h（显然是找不到的）。

在#include指令中的文件名可以含有帮助定位文件的信息，比如目录的路径或驱动器号：

`#include "C:\cprogs\utils.h" /* DOS path */`

```
#include "/cprogs/utils.h" /* UNIX path*/
```

虽然#include指令中记号的双引号使得文件名看起来像字符串字面量，但是预处理器不会把它们作为字符串字面量来处理。（这是幸运的，因为在DOS例子中，字符串字面量中出现的\c和\u，将被作为转义序列处理。）

**可移植性技巧** 通常最好的做法是在#include指令中不包含路径或驱动器的信息。当把程序转移到其他机器上，或者更糟的情况是转移到其他操作系统上时，这类信息会使编译程序变得很困难。

例如，下面的这些#include指令指定了驱动器或路径信息，而这些信息不可能一直是有效的：

```
#include "d-utils.h"
#include "\cprogs\utils.h"
#include "d:\cprogs\utils.h"
```

下列这些指令相对好一些。它们没有限制特殊的驱动器，而且指定的目录与当前目录相关：

```
#include <sys\stat.h>
#include "utils.h"
#include "..\include\utils.h"
```

## 15.2.2 共享宏定义和类型定义

大多数大规模的程序包含用于几个源文件（或者，最极端的情况是用于全部源文件）共享的宏定义和类型定义。这些定义应该放在头文件中。

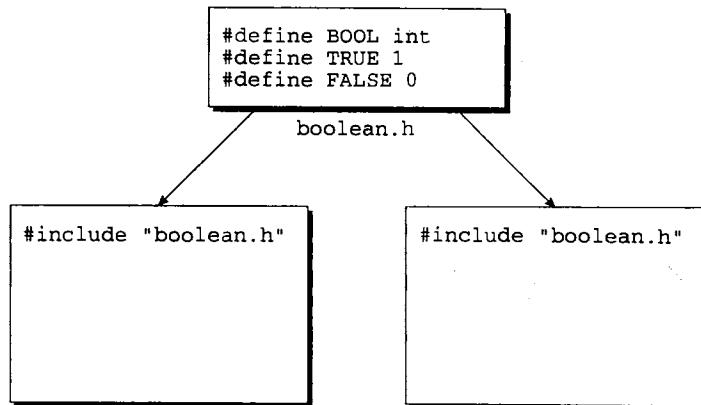
例如，假设正在编写的程序使用名为BOOL、TRUE和FALSE的宏。不用在每个需要的源文件中重复定义这些宏，而是把这些定义放在像名为boolean.h这样的头文件中，这样做是很有意义的：

```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

任何需要这些宏的源文件只需简单包含下面这一行：

```
#include "Boolean.h"
```

在下面的图中，两个文件包含了boolean.h。



类型定义在头文件中也是很普遍的。例如，不用定义BOOL宏，而是可以用typedef产生一个Bool类型。如果这样做，boolean.h文件将有下列显示：

```
#define TRUE 1
#define FALSE 0
typedef int Bool;
```

把宏定义和类型定义放在头文件中有许多明显的好处。首先，不用把定义复制给需要的源文件可以节约时间。其次，程序变得更加容易修改。改变宏定义或类型定义只需要编辑单独的头文件，而不需要修改许多使用宏或类型的源文件。最后，不需要担心由于源文件包含相同宏或类型的不同定义而导致的矛盾。

### 15.2.3 共享函数原型

假设源文件包含函数f的调用，而函数f是定义在另一个foo.c文件中的。首先，调用没有声明的函数f是非常危险的。没有函数原型可依赖，迫使编译器假定函数f的返回类型是int型的，而且假定形式参数的数量和函数f的调用中的实际参数数量是匹配的。通过默认的实际参数提升（>9.3.1节），实际参数自身自动转化为“标准格式”。编译器的假设也可能是错误的，但是，因为一次只能编译一个文件，所以是没有办法进行检查的。如果假设是错误的，那么程序大概无法工作，而且没有任何作为原因的线索。



当调用定义在其他文件中的函数f时，要始终确保编译器在调用之前看到函数f的原型。

第一个冲动可能是在调用函数f的文件中声明它。这样可以解决问题，但是可能产生持续的“噩梦”。假设50个源文件要调用函数，如何能确保函数f的原型在所有文件中都一样呢？如何能保证这些原型和foo.c文件中函数f的定义相匹配呢？如果稍后函数f发生了改变，如何能找到所有用到此函数的文件呢？

解决办法是显而易见的：把函数f的原型放进头文件中，然后在所有调用函数f的地方包含头文件。**Q&A**既然在文件foo.c中定义了函数f，那么就让我们把头文件命名为foo.h。除了在调用函数f的源文件中包含foo.h，还将需要把它包含在foo.c中，从而使编译器检查foo.h中函数f的原型是否和foo.c中的函数定义相匹配成为可能。



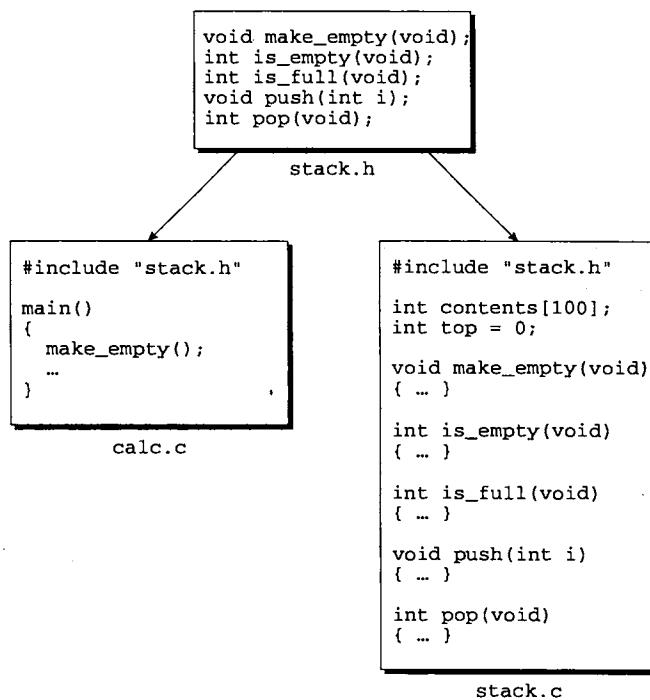
在含有函数f定义的源文件中始终包含声明函数f的头文件。如果这样做失败可能导致难以发现的错误，因为在程序别处对函数f的调用可能会和函数f的定义不匹配。

如果文件foo.c包含其他函数，那么应该像函数f一样在同一个头文件中声明大多数的函数。毕竟，文件foo.c中的其他函数大概会与函数f有关。任何含有函数f调用的文件可能会需要文件foo.c中的其他一些函数。然而，打算仅用于文件foo.c中的函数不需要在头文件中声明，如果声明了将会产生误解。

为了说明头文件中函数原型的使用，一起回到15.1节逆波兰计算器的示例。文件stack.c将包含函数make\_empty、函数is\_empty、函数is\_full、函数push和函数pop的定义。这些函数的原型应该在头文件stack.h中：

```
void make_empty(void);
int is_empty (void);
int is_full (void);
void push(int i);
int pop (void);
```

（为了避免示例复杂化，函数is\_empty和函数is\_full将不再返回Bool型值而返回int型值。）文件calc.c中将包含stack.h以便于编译器将知道每个函数的返回类型，以及形式参数的数量和类型。文件stack.c中也将包含stack.h以便于编译器可以检查stack.h中的函数原型是否和stack.c中的定义相匹配。下面这张图说明了stack.h、stack.c和calc.c。



### 15.2.4 共享变量声明

就像在函数间共享变量的方式一样，变量可以在文件中共享。为了共享函数，要把函数的定义放在一个源文件中，然后在需要调用此函数的其他文件中放置声明。共享变量的方法和此方式非常类似。

在此之前，不需要区别变量的声明和它的定义。为了声明变量i，写成如下形式：

```
int i; /* declares i and defines it as well */
```

这样不仅声明i是int型的变量，而且也对i进行了定义，从而使编译器为i留出了空间。为了声明没有定义的变量i，需要在变量声明的开始处放置关键字extern：

```
extern int i; /* declares i without defining it */
```

extern提示编译器变量i是在程序中的其他位置定义的（大多数可能是在不同的源文件中），因此不需要为i分配空间。

顺便说一句，extern可以用于所有类型的变量。在数组的声明中使用extern时，可以忽略数组的长度：

```
extern int a[];
```

**Q&A**因为此刻编译器不用为数组a分配空间，所以也就不需要知道数组a的长度了。

为了在几个源文件中共享变量i，首先把变量i的定义放置在一个文件中：

```
int i;
```

如果需要对变量i初始化，那么可以在这里放初始值。在编译这个文件时，编译器将会为变量i分配内存空间，而其他文件将包含变量i的声明：

```
extern int i;
```

通过在每个文件中声明变量i，使得在这些文件中可以访问/或修改变量i。然而，由于关键字extern，使得编译器不会在每次编译其中某个文件时为变量i分配额外的内存空间。

当在文件中共享变量时，会面临和共享函数时相似的挑战：确保变量的所有声明和变量的定义一致。



当同一个变量的声明出现在不同文件中时，编译器无法检查声明是否和变量定义相匹配。例如，一个文件可以包含如下定义：

```
int i;
```

同时另一个文件包含声明

```
extern long int i;
```

这类错误可能导致程序的行为异常。

为了避免矛盾，通常把共享变量的声明放置在头文件中。需要访问特殊变量的源文件可以稍后包含适当的头文件。此外，含有变量定义的源文件包含每一个含有变量声明的头文件，这样使编译器可以检查两者是否匹配。

虽然在文件中共享变量是C语言界中的长期惯例，但是它有重大的缺点。在19.2节中将会看到这个问题的内容，并且学习如何设计不需要共享变量的程序。

### 15.2.5 嵌套包含

**[310]** 头文件自身可以包含#include指令。虽然这种做法可能看上去有点奇怪，但实际上却是十分有用的。思考含有下列原型的stack.h文件：

```
int is_empty(void);
int is_full (void);
```

既然这些函数只能返回0或1，那么声明它们的返回类型是Bool型而不是int型是一个很好的主意：

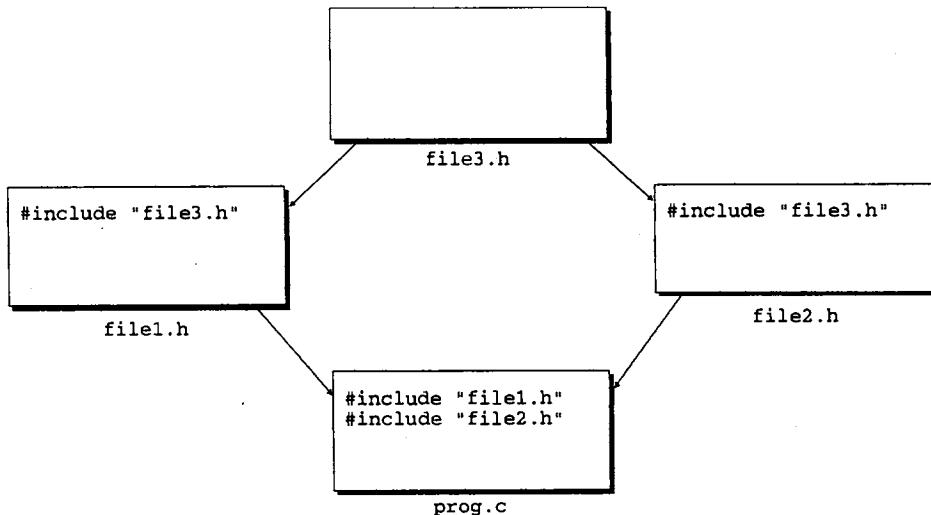
```
Bool is_empty (void);
Bool is_full (void);
```

当然，将需要在stack.h中包含文件boolean.h以便在编译stack.h时Bool的定义是有效的。

就传统而言，C程序员避免使用嵌套包含。(C语言的早期版本根本不允许嵌套包含。)但是，这种对嵌套包含的偏见正在逐渐减弱。**C++**一个原因就是嵌套包含在C++语言中的普遍应用。

### 15.2.6 保护头文件

如果源文件包含同一个头文件两次，那么可能产生编译错误的结果。当头文件包含其他头文件时，这种问题十分普遍。例如，假设file1.h包含file3.h，file2.h包含file3.h，而prog.c同时包含file1.h和file2.h：



在编译prog.c时，就将会编译两次file3.h。

包含同一个头文件两次不会总是造成编译错误。如果文件只包含宏定义、函数原型和/或变量声明，那么将不会有任何困难。然而，如果文件包含类型定义，则会带来编译错误。

就安全而言，保护全部头文件可能是个好主意。那样的话可以在稍候添加类型定义而不用冒可能因忘记保护文件而产生的风险。此外，在程序开发过程期间，避免相同头文件的不必要的重复编译可以节约许多时间。

为了防止头文件多次包含，将用#ifndef和#endif两个指令来把文件的内容闭合起来。例如，可以用如下方式保护文件boolean.h：

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;

#endif
```

在首次包含这个文件时，将不定义宏BOOLEAN\_H，所以预处理器允许保留在#ifndef和#endif之间的多行内容。但是如果再次包含此文件，那么预处理器将把#ifndef和#endif之间的多行内容删除。

宏(BOOLEAN\_H)的名字不是真正的问题。但是，给它取类似于头文件名的名字是避免和其他的宏冲突的好方法。既然不能给宏命名为BOOLEAN.H(标识符不能含有句点)，所以像BOOLEAN\_H这样的名字是个很好的选择。

### 15.2.7 头文件中的#error 指令

经常把#error指令(>14.5.1节)放置在头文件中是用来检查不应该包含头文件的条件。例如，假设头文件包含只能正常工作在DOS程序中的图形函数原型。为了保证只能在DOS程序中包含它，头文件可以包含#ifndef(或#if)指令用来检查宏，这个宏指示DOS是操作系统：

```
#ifndef DOS
#error Graphics supported only under DOS
#endif
```

如果非DOS的程序试图包含此头文件，那么编译将在#error指令处停止。

## 15.3 把程序划分成多个文件

现在应用我们知道的关于头文件和源文件的知识来开发一种把程序划分成多个文件的简单方法。这里将集中在函数上讨论，但是同样的规则也适用于外部变量。假设已经设计好程序，换句话说，已经决定程序需要什么函数以及如何把函数逻辑化地编排在相关的组中。(程序设计本身就是一个完整的主题，第19章将会讨论程序设计问题。)

下面是处理的方法。把每组函数集合放入单独的源文件中(比如用名字foo.c来表示一个这样的文件)。另外，创建和源文件同名的头文件，只是扩展名为.h(在此例中，头文件是foo.h)。还在foo.h文件中放置函数的原型，而函数的定义则是在foo.c中(在foo.h文件中不需要也不应该声明只为用于foo.c内部而设计的函数)。每个需要调用定义在foo.c文件中的函数的源文件都包含foo.h文件。而且，foo.c文件也包含foo.h文件，这是为了编译器可以检查foo.h文件中的函数原型是否与foo.c文件中的函数定义相一致。

main函数将出现在某个文件中，这个文件的名字与程序的名字相匹配。如果希望称程序为bar，那么main函数就应该在文件bar.c中。只要程序中的其他文件不调用其他函数，那么这

些函数是可以和main函数在同一个文件中的。

## 程序：文本格式化

为了说明刚刚论述的方法，现在用它来做文本格式化的小程序。既然一些操作系统已经有了名为format的程序，那么就把要编写的程序命名为fmt。作为给fmt的输入样例，将采用文件quote，假设这个文件包含下列（不完全格式化的）引用语，这些引用语来自Dennis M. Ritchie写的“*The Development of the C language*”（ACM SIGPLAN Notices（March 1993）: 207）：

```
C is quirky, flawed, and an
enormous success. While accidents of history
surely helped, it evidently satisfied a need
for a system implementation language efficient
enough to displace assembly language,
yet sufficiently abstract and fluent to describe
algorithms and interactions in a wide variety
of environments.
-- Dennis M. Ritchie
```

为了程序能在UNIX和DOS环境下运行，最好录入命令

```
fmt <quote
```

符号<提示操作系统程序fmt将用读入文件quote来代替键盘输入。把由UNIX、DOS和其他操作系统支持的这种特性称为是输入重定向（input redirection）（>22.1.2节）。当把给定的文件quote作为输入时，程序fmt将产生下列输出：

```
C is quirky, flawed, and an enormous success. While
accidents of history surely helped, it evidently satisfied a
need for a system implementation language efficient enough
to displace assembly language, yet sufficiently abstract and
fluent to describe algorithms and interactions in a wide
variety of environments. -- Dennis M. Ritchie
```

程序fmt的输出通常将显示在屏幕上，但是可以利用输出重定向（output redirection）（>22.1.2节）把结果保存在文件中：

```
313 fmt <quote >newquote
```

程序fmt的输出将放入到文件newquote中。

通常情况下，除了额外的空格和删除的空行，以及做过填充和调整的行，程序fmt的输出应该和输入一样。“填充”行意味着添加单词直到再多加一个单词就会导致行溢出时才停止。“调整”行意味着在单词间添加额外的空格以便于每行有精确的相同长度（60个字符）。必须进行调整，只有这样一行内单词间的空格才是相等的（或者几乎是相等的）。对输出的最后一行将不进行调整。

假设没有单词的长度超过20个字符。（把与单词相邻的标点符号看成是单词的一部分。）当然，这是一点点的限制，但是一旦编写和调试程序，就可以很容易的增加这种限制，事实上是从未超过这种限制的。如果程序遇到较长的单词，它需要忽略前20个字符后的所有字符，用一个单独的星号替换它们。例如，单词

```
antidisestablishmentarianism
```

将会显示成

```
antidisestablishment*
```

现在明白了程序应该完成的内容，应该考虑设计了。首先发现程序不能像读似的一个一个写单词，而必须把输入存储在一个“行缓冲区”中，直到有足够的空间填满一行。在进一步思考之后，我们决定程序的核心将是如下所示的循环：

```

for (;;) {
 读单词;
 if (不能读单词) {
 不用调整地写行缓冲区的内容;
 终止程序;
 }
 if (单词不适合在行缓冲区中) {
 调整地写行缓冲区的内容;
 清除行缓冲区;
 }
 往行缓冲区中添加单词;
}

```

314

因为我们需要函数处理单词，并且还需要函数处理行缓冲区，所以把程序划分为3个源文件。把所有和单词相关的函数放在一个文件中（word.c），而把所有和行缓冲区相关的函数放在另一个文件中（line.c）。第3个文件（fmt.c）将包含main函数。除了上述这些文件，还需要两个头文件word.h和line.h。头文件word.h将包含word.c文件中的函数原型，而头文件line.h将对line.c承担类似的工作。

通过检查主循环可以发现需要只和单词相关的函数是函数read\_word。（如果read\_word函数因为到了输入文件末尾而不读入单词，那么将通过假装读取“空”单词的方法给主循环发信号。）因此，文件word.h是一个短小的文件：

**word.h**

```

#ifndef WORD_H
#define WORD_H

 * read_word: Reads the next word from the input and *
 * stores it in word. Makes word empty if no *
 * word could be read because of end-of-file. *
 * Truncates the word if its length exceeds *
 * len. *

void read_word(char *word, int len);

#endif

```

注意宏WORD\_H是如何保护多次包含的word.h文件的。虽然word.h文件不是真的需要保护，但是按照这种方式保护所有头文件是个很好的方法。

文件line.h将不会像word.h那样短小。主循环的轮廓显示了对执行下列操作的函数的需求：

- 不调整地写行缓冲区的内容。
- 检查单词是否适合在缓冲区中。
- 调整地写行缓冲区的内容。
- 清除行缓冲区。
- 往行缓冲区中添加单词。

我们将要调用下面这些函数：flush\_line、space\_remaining、write\_line、clear\_line和add\_word。下面是头文件line.h。

315

**line.h**

```

#ifndef LINE_H
#define LINE_H

 * clear_line: Clears the current line. *

```

```

void clear_line(void);

/*****************
 * add_word: Adds word to the end of the current line. *
 * If this is not the first word on the line, *
 * puts one space before word. *
 *****************/
void add_word(const char *word);

/*****************
 * space_remaining: Returns the number of characters left *
 * in the current line. *
 *****************/
int space_remaining(void);

/*****************
 * write_line: Writes the current line with *
 * justification. *
 *****************/
void write_line(void);

/*****************
 * flush_line: Writes the current line without *
 * justification. If the line is empty, does *
 * nothing. *
 *****************/
void flush_line(void);

#endif

```

在编写文件word.c和文件line.c之前，可以用在头文件word.h和头文件line.h中声明的函数来编写主程序fmt.c。编写这个文件主要是把原始的循环设计翻译成C语言。

### **fmt.c**

```

/* Formats a file of text */

#include <string.h>
#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

main()
{
 char word[MAX_WORD_LEN+2];
 int word_len;

 clear_line();
 for (;;) {
 read_word(word, MAX_WORD_LEN+1);
 word_len = strlen(word);
 if (word_len == 0) {
 flush_line();
 return 0;
 }
 if (word_len > MAX_WORD_LEN)
 word[MAX_WORD_LEN] = '*';
 if (word_len + 1 > space_remaining()) {
 write_line();
 clear_line();
 }
 add_word(word);
 }
}

```

包含两个头文件line.h和word.h可以使编译器在编译fmt.c时访问到两个文件中的函数原型。

main函数利用一个技巧解决了单词超过20个字符的问题。在调用read\_word函数时，main函数告诉read\_word截短任何超过21个字符的单词。当read\_word函数返回后，main函数检查word包含的字符串长度是否超过20个字符。如果超过了，那么读入的单词必须至少是21个字符长（在截短前），所以main函数会用星号来替换第21个字符。

现在开始编写word.c程序。虽然头文件word.h只有唯一一个read\_word函数的原型，但是可以在word.c中放置额外需要的函数。在运行时，如果添加一个小的“帮助”函数read\_char，这样可以比较容易地编写函数read\_word。read\_char函数的工作就是读单独一个字符，并且把遇到的换行符或制表符转换为空格。用read\_char函数代替getchar函数进行调用，read\_word函数将自动把换行符和制表符作为空格来处理。

下面是文件word.c：

```
word.c
#include <stdio.h>
#include "word.h"

int read_char(void)
{
 int ch = getchar();

 if (ch == '\n' || ch == '\t')
 return ' ';

 return ch;
}

void read_word(char *word, int len)
{
 int ch, pos = 0;
 while ((ch = read_char()) == ' ')
 ;

 while (ch != ' ' && ch != EOF) {
 if (pos < len)
 word[pos++] = ch;
 ch = read_char();
 }

 word[pos] = '\0';
}
```

317

在讨论read\_word函数之前，有两个正好关于getchar函数的注释。第一，getchar函数实际上返回的是int型值而不是char型值，这是因为在read\_char函数中把变量ch声明为（>22.4节）int类型。第二，当不能连续读入时（通常因为读到了输入文件的末尾），getchar返回值EOF。

read\_word函数由两个循环构成。第一个循环跳过空格，在遇到第一个非空字符时停止。（EOF不是空的，所以如果到达输入文件的末尾，循环停止。）第二个循环读字符直到遇到空格或EOF时停止。循环体把字符存储到word中直到达到len的限制时停止。在这之后，循环继续读入字符，但是不再存储这些字符。read\_word函数中的最后的语句以空字符结束单词，从而构成字符串。如果read\_word在找到非空字符前遇到EOF，pos将在末尾置为0，从而使得word为空字符串。

唯一剩下的文件是line.c。这个文件提供在文件line.h中声明的函数的定义。line.c文件也将需要变量来跟踪行缓冲区的状态。一个变量line将存储当前行的字符。严格地讲，line

是我们需要的唯一变量。然而，出于对速度和便利的考虑，将用到另外两个变量：line\_len（当前行的字符数量）和num\_words（当前行的单词数量）。

下面是文件line.c：

### line.c

```
#include <stdio.h>
#include <string.h>
#include "line.h"

#define MAX_LINE_LEN 60

char line[MAX_LINE_LEN+1];
int line_len = 0;
int num_words = 0;

[318]
void clear_line(void)
{
 line[0] = '\0';
 line_len = 0;
 num_words = 0;
}

void add_word(const char *word)
{
 if (num_words > 0) {
 line[line_len] = ' ';
 line[line_len+1] = '\0';
 line_len++;
 }
 strcat(line, word);
 line_len += strlen(word);
 num_words++;
}

int space_remaining(void)
{
 return MAX_LINE_LEN - line_len;
}

void write_line(void)
{
 int extra_spaces, spaces_to_insert, i, j;

 extra_spaces = MAX_LINE_LEN - line_len;
 for (i = 0; i < line_len; i++) {
 if (line[i] != ' ')
 putchar(line[i]);
 else {
 spaces_to_insert = extra_spaces / (num_words - 1);
 for (j = 1; j <= spaces_to_insert + 1; j++)
 putchar(' ');
 extra_spaces -= spaces_to_insert;
 num_words--;
 }
 }
 putchar('\n');
}

void flush_line(void)
{
 if (line_len > 0)
 puts(line);
}
```

文件line.c中大多数函数都很容易编写。唯一需要技巧的函数是write\_line。这个函数用来调整地写一行内容。函数write\_line在line中一个一个地写字符，如果需要添加额外的空格那么就在每对单词之间停顿。额外空格的数量存储在变量spaces\_to\_insert中，这个变量的值由extra\_spaces / (num\_words -1)获得，其中extra\_spaces初始是最大行长度和当前行长度的差。因为在打印每个单词之后extra\_spaces和num\_words都发生变化，所以spaces\_to\_insert也将变化。如果extra\_spaces初始为10，并且num\_words初始为5，那么将有2个额外的空格跟着第1个单词，有2个额外空格跟着第2个单词，有3个额外的空格跟着第3个单词，以及有3个额外的空格跟着第4个单词。

319

## 15.4 构建多文件程序

在2.1节中，我们验证了编译和链接程序的过程适用于单独一个文件。现在将把这种讨论扩展到由多个文件构成的程序中。构建大规模程序需要和构建小程序相同的基本步骤：

- **编译。**必须对程序中的每个源文件单独进行编译。（不需要编译头文件。当包含头文件的源文件编译时会自动编译头文件。）编译器产生一个文件，此文件包含来自每个源文件的目标代码。这些被称为目标文件（object file）的文件在UNIX系统中的扩展名为.o，而在DOS系统中的扩展名为.obj。
- **链接。**链接器把上一步产生的目标文件和库函数的代码结合在一起生成可执行的程序。在其他责任中，链接器还有责任要解决编译器遗留的外部参考问题。（外部参考发生在一个文件中的函数调用另一个文件中定义的函数时，或者访问另一个文件中定义的变量时。）

大多数编译器允许用单独一步来构建程序。例如，对于UNIX的cc编译器来说，最好使用下列命令行来构建15.3节的fmt程序：

```
% cc -o fmt fmt.c line.c word.c
```

（字符%是UNIX提示符。）首先把三个源文件编译成目标代码，并且分别用名为fmt.o、line.o和word.c的文件来存储这些代码。然后，会自动把目标文件传递给链接器，链接器会把这些文件结合成一个单独的文件。选项-o是告诉编译器需要的可执行文件的名字是fmt。

### 15.4.1 makefile

把所有源文件的名字放在命令行中的方法很快变得枯燥乏味。更糟糕的是，如果重新编译了所有源文件，不仅仅是最近变化的源文件会受到影响，而且重新构建程序可能会浪费大量的时间。

为了更易于构建大规模的程序，UNIX系统发明了makefile的概念，这个文件包含构建程序的必要信息。makefile不仅列出了作为程序部分的文件，而且还描述了文件之间的依赖性。假设文件foo.c包含文件bar.h，那么就说foo.c“依赖于”bar.h，因为bar.h的变化将会需要重新编译foo.c。

下面是针对程序fmt而设的UNIX系统的makefile：

320

```
fmt: fmt.o word.o line.o
 cc -o fmt fmt.o word.o line.o

fmt.o: fmt.c word.h line.h
 cc -c fmt.c

word.o: word.c word.h
 cc -c word.c

line.o: line.c line.h
 cc -c line.c
```

这里有4组行。每组的第一行给出了目标文件，跟在后边的是它所依赖的文件。如果因为其中一个所依赖的文件的变化而使目标必须要重新构建，那么每组的第二行是用来执行的命令。首先来看前两组，而后两组相似。

在第一组中，fmt（可执行程序）是目标文件：

```
fmt: fmt.o word.o line.o
 cc -o fmt fmt.o word.o line.o
```

第一行说明fmt程序依赖于fmt.o文件、word.o文件和line.o文件。因为程序是最后构建，那么若3个文件中的任何一个发生改变，则都需要重新构建fmt程序。紧跟其后的一行命令说明是如何进行构建的（通过使用cc来链接3个目标文件）。

在第二组中，fmt.o是目标文件：

```
fmt.o: fmt.c word.h line.h
 cc -c fmt.c
```

第一行说明，如果文件fmt.c、word.h文件或line.h文件发生改变，那么fmt.o需要重新构建。（理由是提及fmt.c包含的word.h和line.h这两个文件，所以任意一个文件的改变都可能会对fmt.c产生影响。）下一行信息说明如何更新fmt.o（通过重新编译fmt.c）。选项-c通知编译器去编译fmt.c，但是不要试图链接它，因为它不是一个完整的程序。

用于其他操作系统的makefile是很类似的，但不是完全一样的。例如，如果使用Borland的bcc编译器，将使用略有不同的makefile：

```
fmt.exe: fmt.obj word.obj line.obj
 bcc fmt.obj word.obj line.obj
fmt.obj: fmt.c word.h line.h
 bcc -c fmt.c
word.obj: word.c word.h
 bcc -c word.c
line.obj: line.c line.h
 bcc -c line.c
```

321

编译器用bcc代替了cc，目标文件也由扩展名.obj代替了.o，同时可执行文件也由fmt.exe代替了fmt。而且，不再需要选项-o，因为第一个目标文件的名字fmt.obj就确定了可执行文件的名字。

一旦为程序创造了makefile，就能使用make工具来构建（或重新构建）程序了。通过检查与程序中每个文件相关的时间和日期，make可以确定哪个文件是过期的。然后，它因为需要重新构建程序而自动唤醒编译器和链接器。

make是如此复杂以致足够用一本书<sup>①</sup>来介绍，所以这里不会试图深入研究它的复杂性。这里只是说明真正的makefiles通常不是像例子显示的那样容易理解。这里有几种方法可以减少makefile中的冗余，并且使得它们更容易修改。但是，同时这些技术也极大地减少了它们的可读性。

顺便说一句，不是每个人都用makefile的。其他程序维护工具正在变得流行，包括一些集成开发环境支持的“工程文件”。检查所使用系统的说明文档，看看它是否支持makefile或工程文件，还是二者都支持。

### 15.4.2 链接期间的错误

一些在编译期间无法发现的错误将会在链接期间被发现。事实上，如果程序中丢失了函数定义或变量定义，那么链接器将无法解决外部引用，从而导致出现类似“Undefined symbol”或“Unresolved external reference”的信息。

<sup>①</sup> Tondo, Nathanson, and yount, *Mastering MAKE*, second Edition, Prentice-Hall 1994.

通常很容易修改链接器检查到的错误。下面是一些最常见的错误起因：

- **拼写错误。**如果变量名或函数名拼写错误，那么链接器将作为丢失来进行报告。例如，在程序中定义了函数read\_char，但是却把它写为read\_cahr，那么链接器将报告说丢失了read\_char函数。
- **丢失文件。**如果链接器不能找到文件foo.c中的函数，那么它可能不会知道此文件。这时就要检查makefile或工程文件来确保foo.c文件是列出了的。
- **丢失库。**链接器不可能找到程序中用到的全部库函数。发生在UNIX系统中的经典例子，在链接期间无法搜索到数学函数的地方，直到出现了选项-lm才办到。检查所使用的系统文档，看看可以用于链接器的选项有哪些。

### 15.4.3 重新构建程序

在开发程序期间，极少需要编译全部文件。大多数时候，将测试程序，发生变化，然后再次构建程序。为了节约时间，重新构建的过程应该只对那些可能受到最后一次变化影响的文件进行重新编译。[322]

假设按照15.3节的框架方法设计了程序，对每一个源文件都使用了头文件。为了发现在变化后需要重新编译的文件的数量，我们需要考虑两种可能性。

第一个可能性是变化影响单独一个源文件。这种情况下，只有此文件需要重新编译。(当然，在此之后整个程序将需要重新链接。)思考程序fmt。假设决定精简word.c中的函数read\_char(修改过的地方用粗体标注)：

```
int read_char (void)
{
 int ch = getchar();
 return (ch == '\n' || ch == '\t' ? ' ' : ch);
}
```

这种改变没有影响read\_char的调用方式，所以不需要修改word.h。在改变以后，只需要重新编译word.c并且重新链接程序就行了。

第二个可能性是变化影响头文件。这种情况下，应该重新编译包含此头文件的所有文件，因为它们可能潜在地受到这种变化的影响。(这些文件中的一部分可能受到影响，但是采取保守的方法。)

作为示例，思考一下程序fmt中的函数read\_word。注意，为了确定刚读入的单词的长度，main函数在调用read\_word函数后立刻调用strlen。因为read\_word函数已经知道了单词的长度(read\_word函数的变量pos负责跟踪长度)，所以使用strlen就显得有些没必要了。修改read\_word函数来返回单词的长度是很容易的。首先，改变word.h文件中的read\_word函数的原型：

```
/***
 * read_word: Reads the next word from the input and *
 * stores it in word. Makes word empty if no *
 * word could be read because of end-of-file. *
 * Truncates the word if its length exceeds *
 * len. Returns the number of characters *
 * stored. *
 ***/
int read_word(char *word, int len);
```

当然，一定要记住修改附属于read\_word函数的注释。接下来，修改word.c文件中的read\_word函数的定义：

```
int read_word(char *word, int len)
{
```

323

```

int ch, pos = 0;

while ((ch = read_char()) == ' ')
;
while (ch != ' ' & ch != EOF) {
 if (pos < len)
 word[pos++] = ch;
 ch = read_char();
}

word[pos] = '\0';
return pos;
}

```

最后，再来修改fmt.c，方法是删除对<string.h>的包含，以及按如下方式修改main函数：

```

main()
{
 char word[MAX_WORD_LEN+2];
 int word_len;

 clear_line();
 for (;;) {
 word_len = read_word (word, MAX_WORD_LEN+1);
 if (word_len == 0) {
 flush_line();
 return 0;
 }
 if (word_len > MAX_WORD_LEN)
 word[MAX WORD LEN] = '*';
 if (word_len + 1 > space_remaining()) {
 write_line();
 clear_line();
 }
 add_word(word);
 }
}

```

一旦做了上述这些修改，将需要重新构建程序fmt，方法是要重新编译word.c和fmt.c，然后再重新进行链接。不需要重新编译line.c，因为它不包含word.h，所以也就不会受到word.h改变的影响。在UNIX系统中，可以使用下列命令来重新构建程序：

```
% cc -o fmt fmt.c word.c line.o
```

注意，这里用line.o代替line.c。

使用makefile的好处之一就是可以自动进行重新构建。通过检查每个文件的日期，make实用程序可以确定从程序最后一次构建后哪些文件发生了改变。然后，它会把那些改变的文件和直接或间接依赖于它们的全部文件一起进行重新编译。

#### 15.4.4 在程序外定义宏

在编译程序时，通常C语言编译器提供一些指定宏的值的方法。这种能力使我们不需要编辑任何程序文件就对宏的值进行改变变得非常容易。当利用makefile自动构建程序时这种能力尤其有价值。

大多数UNIX编译器（和某些非UNIX编译器）支持选项-D，此选项允许宏的值用命令行来指定：

```
% cc -DDEBUG=1 foo.c
```

在这个例子中，定义宏DEBUG在程序foo.c中的值为1，就如同在foo.c的开始处出现的下面这行信息：

```
#define DEBUG 1
```

如果选项-D命名的宏是没有指定的值，那么这个值被设为1。

一些编译器也支持选项-U，这个选项“未定义”宏，就如同使用了#define一样：

```
% cc -UDEBUG foo.c
```

## 问与答

问：这里没有任何例子是使用#include指令来包含源文件。如果这样做了会发生什么？

答：这不是个好方法，但是它是合法的。这里有一个这类问题的例子可以拿来讨论。假设foo.c定义的函数f在bar.c和baz.c中会需要，所以把下列指令放在bar.c和baz.c中：

```
#include "foo.c"
```

这些文件都会很好地被编译。稍后，当链接器发现函数f的两个目标代码的副本时，问题就出现了。当然，如果只是bar.c包含此函数，而baz.c没有，那么将会从包含的foo.c中拿走。为了避免出现问题，最好只在头文件中使用#include，而不要在源文件中使用。

问：针对#include指令的精确搜索规则是什么？(p.212)

答：这与所使用的编译器有关。C标准在#include的表述中故意模糊不清。如果文件名用尖括号括起来，那么预处理器把它看成是“实现定义的地方的序列”，作为倾斜的标准来放置。如果文件名用双引号引起，那么文件就是“以实现定义的方式搜索”，而且如果没有发现，那么搜索就好像它的名字使用尖括号括起来的。原因很简单：不像DOS和UNIX系统，不是所有操作系统都有层次（像树型的）文件系统。

为了使这事更加有趣，标准根本不要求括在尖括号内的名字是文件名字，留下开放的可能，这种可能是#include指令在编译器中利用<>来完全进行操作。

问：我不理解为什么每个源文件都需要它自己的头文件。为什么没有一个大的头文件包含宏定义、类型定义和函数原型呢？通过包含这个文件，每个源文件都可以访问全部需要共享的信息。(p.214)

答：“一个大的头文件”的方法的确可以工作，许多程序员使用这种方法。而且，这种方法有一个好处：因为只有一个头文件，所以要管理的文件较少。然而，对于大规模的程序来说，这种方法的坏处大于它的好处。

使用单独一个头文件为稍后人们读程序提供无用的信息。通过多个头文件，读者可以迅速看到通过特殊源文件使用的程序的其他部分。

但是也不绝对。既然每个源文件都依赖于一个大的头文件，所以改变它会导致要对全部源文件重新编译，这是大规模程序中的一个显著缺陷。更糟的情况是，由于包含了大量信息，所以头文件可能会频繁地改变。

问：本章说到共享数组应该按照下列方式声明：

```
extern int a[];
```

既然数组和指针关系密切，那么用下列写法代替是否合法呢？(p.215)

```
extern int *a;
```

答：不合法。在用于表达式时，数组“衰退”成指针。（当数组名用作函数调用中的实际参数时我们已经注意到这种行为。）在变量声明中，数组和指针是截然不同的两种类型。

问：如果源文件包含了不是真正需要的头文件，会有损害吗？

答：不会，除非头文件的声明或定义与源文件中的冲突。否则，可能发生的最坏情况就是在编译源文件时镜像增加。

问：我需要调用文件foo.c中的函数，所以包含了匹配的头文件foo.h。但是程序编译后并不链接。为什么？

答：在C语言中编译和链接是完全独立的。头文件存在是为了给编译器而不是为链接器提供信息。如果希望调用文件foo.c中的函数，那么需要确保对foo.c进行了编译，还要确保为了找到函数使链接器意识到必须搜索到foo.c的目标文件。通常情况下，这就意味着在程序的makefile或工程文件中命

325

名foo.c。

问：如果程序调用`<stdio.h>`中的函数，这是否意味着在`<stdio.h>`中的所有函数都将和程序链接吗？

答：不是的。包含`<stdio.h>`（或者任何其他头文件）对链接没有任何影响。在任何情况下，大多数链接器都只会链接程序实际需要的函数。

326

## 练习

### 15.1节

1. 15.1节列出了把程序分割成多个源程序的几个优点。

- (a) 请描述几个其他的优点。
- (b) 请描述一些缺点。

### 15.2节

2. 下列哪个不应该放置在头文件中？为什么？

- (a) 函数原型。
- (b) 函数定义。
- (c) 宏定义。
- (d) 类型定义。

3. 如果文件是我们已经编写好的，那么已经看到用`#include <文件>`代替`#include "文件"`可能无法工作。如果文件是系统头文件，那么用`#include "文件"`代替`#include <文件>`是否有什么问题？

4. 假设文件foo.c定义了外部变量i如下：

```
int i;
```

而且文件bar.c以下列方式声明此变量：

```
extern long int i;
```

(a) 假设`sizeof(int)`和`sizeof(long int)`是完全一样的。如果文件bar.c中的一个函数给i赋值为0，请解释会发生什么？

(b) 假设`sizeof(int)`小于`sizeof(long int)`。请重复(l)的问题。

### 15.3节

5. 程序fmt通过在单词间插入额外的空格来调整行。当前编写的函数write\_line的方法是，与开始处的单词间隔相比，靠近行末尾单词的间隔略微宽一些。（例如，靠近末尾的单词彼此之间可能有3个空格，而靠近开始的单词彼此之间可能只有2个空格。）请通过替换函数write\_line来改进此程序，替换后的函数可以在靠近行末尾处的单词之间放置较大的空隙，而在行开始处的单词之间放置一般空隙。

6. 利用15.2节中的设计，编写实现逆波兰表达式计算器的程序。使计算器可以实现双目运算+、-、\*和/。假设它们的含义和在C语言中一样。

### 15.4节

7. 假设程序有3个源文件构成：`main.c`、`f1.c`和`f2.c`，此外还包括两个头文件`f1.h`和`f2.h`。全部3个源文件都包含`f1.h`，但是只有`f1.c`和`f2.c`包含`f2.h`。为此程序编写UNIX makefile。假设需要可执行文件名为`demo`。

8. 下面的问题引用了练习7描述的程序。

- (a) 当程序第一次构建时，需要对哪些文件进行编译？
- (b) 如果在程序构建后对`f1.c`进行了修改，那么需要对哪个（些）文件进行重新编译？
- (c) 如果在程序构建后对`f1.h`进行修改，那么需要对哪个（些）文件进行重新编译？
- (d) 如果在程序构建后对`f2.h`进行了修改，那么需要对哪个（些）文件进行重新编译？

9. (a) 修改程序fmt，使函数`read_word`（代替`main`函数）在被截短的单词的末尾存储\*字符。

- (b) 如果按照(a)进行了修改，那么需要对哪个（些）文件进行重新编译？

327

328

# 结构、联合和枚举

函数延迟绑定：数据结构导致绑定。

记住：在编程过程后期再结构化数据。

本章介绍3种新的类型：结构、联合和枚举。结构（structure）是可能具有不同类型的值（成员（member））的集合。联合（union）和结构很类似，不同之处在于联合的成员共享同一存储空间。这样的结果是，联合可以每次存储一个成员，但是无法同时存储全部成员。枚举（enumeration）是一种整数类型，它的值由程序员来命名。

在这3种类型中，结构是到目前为止最重要的一种类型，所以本章的大部分内容都是关于结构的。16.1节说明了如何声明结构变量，以及如何对其进行基本操作。随后，16.2节解释了定义结构类型的方法，借助结构类型，我们就可以编写函数，接受结构类型的参数或返回结构类型的值。16.3节探讨如何实现数组和结构的嵌套。本章的最后两节分别讨论了联合（16.4节）和枚举（16.5节）。

## 16.1 结构变量

到目前为止唯一介绍的数据结构就是数组。数组有两个重要特性：首先，数组的所有元素具有相同的类型；其次，为了选择数组元素需要指明元素的位置（作为整数下标）。

结构所具有的特性与数组很不相同。结构的元素（在C语言中的说法是成员）可能具有不同的类型。而且，每个结构成员都有名字，所以为了选择特殊的结构成员需要指明结构成员的名字而不是它的位置。

由于大多数编程语言都提供类似的特性，所以结构可能听起来很熟悉。在其他语言中，经常把结构称为记录（record），把结构的成员称为字段（field）。

329

### 16.1.1 结构变量的声明

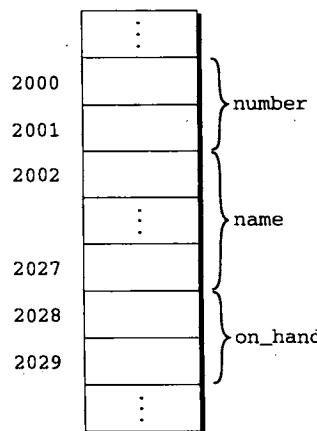
当需要存储相关数据项的集合时，结构是一种合理的选择。例如，假设需要记录存储在仓库中的零件。用来存储每种零件的信息可能包括零件的编号（整数）、零件的名称（字符串）以及现有零件的数量（整数）。为了产生一个可以存储全部3种数据项的变量，可以使用类似下面这样的声明：

```
struct {
 int number;
 char name [NAME_LEN+1] ;
 int on_hand;
} part1, part2 ;
```

每个结构变量都有3个成员：number（零件的编号）、name（零件的名称）和on\_hand（现有数量）。注意，这里的声明格式和C语言中其他变量的声明格式一样。struct{}指明了类型，而part1和part2则是具有这种类型的变量。

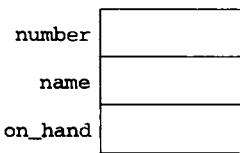
结构的成员在内存中是按照声明的顺序存储的。为了说明part1在内存中存储的形式，现在假设：（1）part1存储在地址为2000的内存单元中，（2）每个整数在内存中占两个字节，（3）

NAME\_LEN的值为25, (4) 成员之间没有间隙。根据这些假设, part1在内存中的样子如下图所示:



通常情况下不需要画出如此详细的结构。这里一般更加抽象地显示结构, 用一系列的方框表示:

330



有时还可以用水平方向的盒子来代替垂直方向的方框:



结构成员的值稍后将放入盒子中。但是现在, 这里保留为空。

每个结构表示一种新的范围。任何声明在此范围内的名字都不会和程序中的其他名字冲突。(C语言的术语表示每个结构都为它的成员设置了单独的名字空间 (name space)。) 例如, 下列声明可能出现在同一程序中:

```
struct {
 int number;
 char name [NAME_LEN+1];
 int on_hand;
} part1, part2;

struct {
 char name[NAME_LEN+1];
 int number;
 char sex;
} employee1, employee2;
```

结构part1和part2中的成员number和成员name不会和结构employee1和employee2中的成员number和成员name冲突。

### 16.1.2 结构变量的初始化

和数组一样, 结构变量也可以在声明的同时进行初始化。为了初始化结构, 要准备存储在结构中的值列表并用大括号把它括起来:

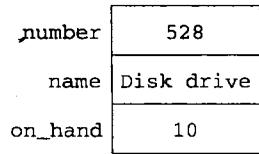
```
struct {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
```

```

} part1 = { 528, "Disk drive", 10 },
part2 = { 914, "Printer cable", 5};

```

初始化式中的值必须按照结构成员的顺序进行显示。在此例中，结构part1的成员number值为528，成员name则是"Disk driver"，以此类推。下面是结构part1初始化后的样子：



结构初始化式遵循的原则类似于数组初始化式的原则。用于结构初始化式的表达式必须是常量。例如，不能用变量来初始化结构part1的成员on\_hand。初始化式可以短于它所初始化的结构。就像对数组那样，任何“剩余的”成员都用0作为它的初始值。

### 16.1.3 对结构的操作

既然数组最常见的操作是下标操作，也就是通过位置选择数组元素的操作，那么结构最常用的操作是选择成员也就无需惊讶了。但是，结构成员的访问是通过名字而不是通过位置。

为了访问结构内的成员，首先写出结构的名字，然后写出成员的名字。例如，下列语句将显示结构part1的成员的值：

```

printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);

```

结构成员的值是左值（>4.2节），所以它们可以出现在赋值运算的左侧，或者是作为自增或自减表达式的操作数：

```

part1.number = 258; /* changes part1's part number */
part1.on_hand++; /* increments part1's quantity on hand */

```

用于访问结构成员的句点实际上就是一个C语言的运算符。参考附录B的运算符表可知，它的运算优先级和后缀++和后缀--运算符一样，所以句点运算符的优先级几乎高于所有其他运算符。考虑下面的例子：

```
scanf("%d", &part1.on_hand);
```

表达式&part1.on\_hand包含两个运算符（即&和.）。.运算符优先级高于&运算符，所以就像希望的那样，&计算的是part1.on\_hand的地址。

另一种主要的结构操作是赋值运算：

```
part2 = part1;
```

现在part2.number包含了和part1.number一样的值，part2.name也将包含和part1.name一样的值，以此类推。

因为数组不能用=运算符进行复制，所以结构可以用=运算符复制应该是一个惊喜。当复制闭合的结构时，考虑把嵌在结构内的数组进行复制甚至会带来更大的惊喜。一些程序员利用这中性质来产生“空”结构，以封装稍候将进行复制的数组：

```

struct { int a [10]; } a1, a2;

a1 = a2; /* legal, since a1 and a2 are structures */

```

运算符=仅仅用于类型一致的结构。两个同时声明的结构（比如part1和part2）是一致的。正如下一节将会看到的那样，使用同样的“结构标记”或同样的类型名声明的结构也是一致的。

除了赋值运算，C语言没有提供其他用于整个结构的操作。**Q&A**特别是不能使用运算符==和!=来判定两个结构是否相等或不等。

## 16.2 结构类型

虽然16.1节说明了声明结构变量的方法，但是它没有讨论一个重要的话题：命名结构类型。假设程序需要声明几个具有相同成员的结构变量。如果一次可以声明全部变量，那么没有什么问题。但是如果需要在程序中的不同位置声明变量，那么问题就复杂了。如果在某处编写了

```
struct {
 int number;
 char name [NAME_LEN+1];
 int on_hand;
} part1;
```

并且在另一处编写了

```
struct {
 int number;
 char name [NAME_LEN+1];
 int on_hand;
} part2;
```

那么立刻就会出现问题。重复的结构信息会使程序膨胀。既然不容易确保声明会保持一致，那么稍后改变程序会很危险。

但是这些还都不是最大的问题。根据C语言的规则，part1和part2不具有一致的类型。这样的结果是不能把part1赋值给part2，反之亦然。而且，因为part1或part2的类型没有名字，所以也就不能用它们作为函数调用的参数了。

**333** 为了克服这些困难，需要为表示结构的类型定义名字，而不是为特定结构的变量命名。正如产生的那样，**Q&A**C语言提供了两种命名结构的方法：既可以声明“结构标记”，也可以使用typedef来定义类型名（类型定义（>7.6节））。

### 16.2.1 结构标记的声明

结构标记(structure tag)是用于标识某种特定结构类型的名字。下面的例子声明了名为part的结构标记：

```
struct part {
 int number;
 char name [NAME_LEN+1];
 int on_hand;
};
```

注意，右大括号后的分号，必须用它来表示声明的结束。



如果无意间忽略了结构声明结尾处的分号，可能会导致意料之外的结果。考虑下面的例子：

```
struct part {
 int number;
 char name [NAME_LEN+1];
 int on_hand;
} *** WRONG--semicolon missing *** /
```

```
f(void)
{
 ...
 return 0; /* error detected at this line */
}
```

因为丢失了函数f的返回值类型，所以通常将默认为int类型。然而，在此例中，由于前面的结构声明没有正常终止，所以编译器会假设struct part的返回类型。编译器直到执行函数中第一条return语句时才会发现错误。结果是：含义模糊的出错信息。

334

一旦产生了标记part，就可以用它来声明变量了：

```
struct part part1, part2;
```

可惜的是，不能通过漏掉单词struct来缩写这个声明：

```
part part1, part2; /* *** WRONG *** /
```

part不是类型名。如果没有单词struct的话，它没有任何意义。

因为结构标记只有在前面放置了单词struct才会有意义，所以它们不会和程序中用到的其他名字发生冲突。程序拥有名为part的变量是非常合法的（虽然有点混淆）。

顺便说一句，结构标记的声明可以和结构变量的声明合并在一起：

```
struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} part1, part2;
```

在这里不仅声明了结构标记part（可能稍后会用part声明更多的变量），而且声明了变量part1和part2。

所有声明为struct part类型的结构彼此之间是一致的：

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;
part2 = part1; /* legal; both parts have the same type */
```

## 16.2.2 结构类型的定义

作为声明结构标记的替换，还可以用typedef来定义真实的类型名。例如，可以按照如下方式定义名为part的类型：

```
typedef struct {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} Part;
```

注意，类型Part的名字必须出现在定义的末尾，而不是在单词struct的后边。

可以像内置类型那样使用Part。例如，可以用它声明变量：

```
Part part1, part2;
```

因为类型Part是typedef的名字，所以不允许书写struct Part。无论在哪里声明，所有的Part类型的变量都是一致的。

当需要命名结构时，Q&A通常既可以选择声明结构标记也可以使用typedef。但是，正如稍后将看到的，结构用于链表（>17.5节）时，必须强制声明结构标记。

## 16.2.3 结构类型的实际参数和返回值

函数可以有结构类型的实际参数和返回值。下面来看两个例子。当把结构part用作实际参数时，第一个函数显示出结构的成员：

335

```
void print_part (struct part p)
{
 printf("part number: %d\n", p.number);
 printf("Part name: %s\n", p.name);
 printf("Quantity on hand: %d\n", p.on_hand);
}
```

下面是print\_part可能的调用方法：

```
print_part(part1);
```

第二个函数返回结构part，因为此结构构成了函数的实际参数：

```
struct part build_part(int number, const char* name,
 int on_hand)
{
 struct part p;
 p. number = number;
 strcpy (p.name, name);
 p. on_hand = on_hand;
 return p;
}
```

注意，函数build\_part的形式参数名和结构part的成员名相同是合法的，因为结构拥有自己的名字空间。下面是build\_part可能的调用方法：

```
part1 = build_part(528, "Disk driver", 10);
```

给函数传递结构和从函数返回结构都要求使用结构中所有成员的副本。这样的结果是，这些操作对程序强加了一定数量的系统开销，特别是结构很大的时候。为了避免这类系统开销，有时用传递指向结构的指针来代替传递结构本身是很明智的做法。类似地，可以使函数返回指向结构的指针来代替返回实际的结构。

有时，可能希望在函数内初始化结构变量来匹配其他结构，就可能应用结构变量作为函数的形式参数。在下面的例子中，part2的初始化式是传递给函数f的形式参数：

```
void f(struct part part1)
{
 struct part part2 = part1;
 ...
}
```

C语言允许这类初始化式，因为初始化的结构（此例中的part2）具有自动的存储期限（对函数而言它是局部的，而且没有声明为static）。初始化式可以是适当类型的任意表达式，包括返回结构的函数调用。

336

## 16.3 数组和结构的嵌套

结构和数组的组合没有限制。数组可以有结构作为元素，同时结构可以包含数组和结构作为成员。已经看过数组嵌套在结构内部的示例（结构part的成员name）。下面再来探讨其他的可能性：成员是结构的结构和元素是结构的数组。

### 16.3.1 嵌套的结构

一种结构嵌套在另一种结构中经常是非常有用的。例如，假设声明了如下的结构，此结构用来存储一个人的教名和姓：

```
struct person_name {
 char first[FIRST_NAME_LEN+1];
 char middle_initial;
 char last[LAST_NAME_LEN+1];
};
```

可以用结构person\_name作为更大结构的一部分内容：

```
struct student {
 struct person_name name;
 int_id, age;
 char sex;
} student1, student2;
```

访问student1的名或姓要求两次应用运算符。

```
strcpy(student1.name.first, "Fred");
```

使name成为结构（替换了结构student中的成员first、middle\_initial和last）的好处之一就是可以更容易地把名字作为数据单元来处理。例如，如果打算编写函数来显示名字，可以把结构person\_name只作为一个实际参数代替三个实际参数来进行传递：

```
display_name(student1.name);
```

同样地，把结构person\_name的信息复制给结构student的成员name将只需要一次而不是三次赋值操作：

```
struct person_name new_name;
...
student1.name = new_name;
```

337

### 16.3.2 结构数组

数组和结构最常见的组合之一就是具有结构元素的数组。这类数组可以用作简单的数据库。例如，下列具有结构part的数组能够存储100种零件的信息：

```
struct part inventory[100];
```

为了访问数组中的某种零件，可以使用下标方式。例如，为了显示存储在位置i的零件，可以写成

```
print_part(inventory[i]);
```

访问结构part内的成员要求把下标和成员选择组合使用。为了给inventory[i]的成员number赋值为883，可以写成

```
inventory[i].number = 883;
```

访问零件名中的单独字符要求用下标方式（选择特定的零件），跟着是成员选择（选择成员name），再跟着是下标（选择零件名中的字符）。为了使存储在inventory[i]的名字变为空字符串，可以写成

```
inventory[i].name[0] = '\0';
```

### 16.3.3 结构数组的初始化

初始化结构数组的工作和初始化多维数组的方法非常相似。每个结构都拥有自己的大括号括起来的初始符，数组的初始符会在结构初始符的外围简单括上另一对大括号。

初始化结构数组的原因之一就是计划把它作为程序执行期间不改变的信息数据库。例如，假设用到的程序在打国际长途电话时会需要访问国家（地区）代码。首先，将设置结构用来存储国家（地区）名和相应代码：

```
struct dialing_code {
 char *country;
 int code;
};
```

注意，country是个指针而不是字符数组。如果计划用结构dialing\_code作为变量可能有问题，但是这里不这样做。当初始化结构dialing\_code时，country会结束指向字符串字面量。

接下来，声明这类结构的数组并且初始化它，从而使此数组包含一些世界上人口最多的国家的代码：

```
const struct dialing_code country_codes[] =
{ {"Argentina", 54}, {"Bangladesh", 880},
 {"Brazil", 55}, {"China", 86},
 {"Colombia", 57}, {"Egypt", 20},
 {"Ethiopia", 251}, {"France", 33},
 {"Germany", 49}, {"India", 91},
```

338

```

 {"Indonesia", 62}, {"Iran", 98},
 {"Italy", 39}, {"Japan", 81},
 {"Korea, Republic of", 82}, {"Mexico", 52},
 {"Nigeria", 234}, {"Pakistan", 92},
 {"Philippines", 63}, {"Poland", 48},
 {"Russia", 7}, {"South Africa", 27},
 {"Spain", 34}, {"Thailand", 66},
 {"Turkey", 90}, {"Ukraine", 7},
 {"United Kingdom", 44}, {"Vietnam", 84},
 {"Zaire", 243}});

```

每个结构值周围的内层大括号是可选项。然而，作为书写风格最好不要忽略它们。

### 16.3.4 程序：维护零件数据库

为了说明实际应用中数组和结构是如何嵌套的，现在将开发一个相对较大的程序，此程序用来维护仓库存储的零件的信息数据库。程序围绕一个结构数组建立，且每个结构包含以下信息：零件的编号、零件的名称以及某种零件的数量。程序将支持下列操作：

- 添加新零件编号、名称和现有的初始数量。如果零件已经在数据库中，或者数据库已满，那么程序必须显示出错信息。
- 给定零件编号，显示出零件的名称和当前现有的数量。如果零件编号不在数据库中，那么程序必须显示出错信息。
- 给定零件编号，改变现有的零件数量。如果零件编号不在数据库中，那么程序必须显示出错信息。
- 显示表格列出数组库中的全部信息。零件必须按照录入的顺序显示出来。
- 终止程序的执行。

使用i（即插入）、s（即搜索）、u（即更新）、p（即显示）和q（即退出）分别表示这些操作。与程序的会话可能如下：

Enter operation code: i  
 Enter part number: 528  
 Enter part name: Disk drive  
 Enter quantity on hand: 10  
 Enter operation code: s  
 Enter part number: 528  
 Part name: Disk drive  
 Quantity on hand: 10

Enter operation code: s  
 Enter part number: 914  
 Part not found.

Enter operation code: i  
 Enter part number: 914  
 Enter part name: Printer cable  
 Enter quantity on hand: 5

Enter operation code: u  
 Enter part number: 528  
 Enter change in quantity on hand: -2

Enter operation code: s  
 Enter part number: 528  
 Part name: Disk drive  
 Quantity on hand: 8

| Part Number | Part Name  | Quantity on Hand |
|-------------|------------|------------------|
| 528         | Disk drive | 8                |

```
914 Printer cable 5
Enter operation code: q
```

程序将在结构中存储每种零件的信息。这里会限制数据库的大小为100种零件，这使用数组来存储结构成为可能，这里称此数组为inventory。（如果这个数组的范围太小，稍后可以经常改变它。）为了记录当前存储在数组中的零件的编号，将使用名为num\_parts的变量。

既然此程序是以菜单方式驱动的，那么十分容易勾划出主循环结构：

```
for (;;) {
 提示用户输入操作码
 读操作码
 Switch (操作码)
 case 'i': 执行插入操作; break;
 case 's': 执行搜索操作; break;
 case 'u': 执行更新操作; break;
 case 'p': 执行显示操作; break;
 case 'q': 终止程序;
 default: 打印出错信息;
 }
}
```

为了方便起见，将分别设置不同的函数执行插入、搜索、更新和显示操作。因为这些函数将都需要访问inventory和num\_parts，所以可以把这些变量外部化。或者把变量藏在main函数内，然后把它们作为实际参数传递给函数。从设计角度来说，通常把针对函数的变量局部化比把它们外部化更好（如果忘记了原因，请见10.2节）。然而，在此程序中，把inventory和num\_parts隐藏在main函数中将只会使程序复杂化。340

由于稍后会解释的原因，这里决定把程序分割为三个文件：invent.c文件，它包含程序的大部分内容；readline.h文件，它包含read\_line函数的原型；readline.c文件，它包含read\_line函数的定义。在本节的后半部分将讨论后两个文件。现在将集中讨论invent.c文件。

#### **invent.c**

```
/* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0; /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

/***
 * main: Prompts the user to enter an operation code,
 * then calls a function to perform the requested
 * action. Repeats until the user enters the
 * command 'q'. Prints an error message if the user
 * enters an illegal code.
******/
```

341

```
main()
{
 char code;

 for (;;) {
 printf("Enter operation code: ");
 scanf(" %c", &code);
 while (getchar() != '\n') /* skips to end of line */
 ;
 switch (code) {
 case 'i': insert();
 break;
 case 's': search();
 break;
 case 'u': update();
 break;
 case 'p': print();
 break;
 case 'q': return 0;
 default: printf("Illegal code\n");
 }
 printf("\n");
 }

/***
 * find_part: Looks up a part number in the inventory *
 * array. Returns the array index if the part *
 * number is found; otherwise, returns -1. *
 ***/
int find_part(int number)
{
 int i;

 for (i = 0; i < num_parts; i++)
 if (inventory[i].number == number)
 return i;
 return -1;
}

/***
 * insert: Prompts the user for information about a new *
 * part and then inserts the part into the *
 * database. Prints an error message and returns *
 * prematurely if the part already exists or the *
 * database is full. *
 ***/
void insert(void)
{
 int part_number;

 if (num_parts == MAX_PARTS) {
 printf("Database is full; can't add more parts.\n");
 return;
 }

 printf("Enter part number: ");
 scanf("%d", &part_number);
 if (find_part(part_number) >= 0) {
 printf("Part already exists.\n");
 return;
 }

 inventory[num_parts].number = part_number;
 printf("Enter part name: ");
```

342

```

read_line(inventory[num_parts].name, NAME_LEN);
printf("Enter quantity on hand: ");
scanf("%d", &inventory[num_parts].on_hand);
num_parts++;
}

* search: Prompts the user to enter a part number, then *
* looks up the part in the database. If the part *
* exists, prints the name and quantity on hand; *
* if not, prints an error message. *

void search(void)
{
 int i, number;

 printf("Enter part number: ");
 scanf("%d", &number);
 i = find_part(number);
 if (i >= 0) {
 printf("Part name: %s\n", inventory[i].name);
 printf("Quantity on hand: %d\n", inventory[i].on_hand);
 } else
 printf("Part not found.\n");
}

* update: Prompts the user to enter a part number. *
* Prints an error message if the part doesn't *
* exist; otherwise, prompts the user to enter *
* change in quantity on hand and updates the *
* database. *

void update(void)
{
 int i, number, change;

 printf("Enter part number: ");
 scanf("%d", &number);
 i = find_part(number);
 if (i >= 0) {
 printf("Enter change in quantity on hand: ");
 scanf("%d", &change);
 inventory[i].on_hand += change;
 } else
 printf("Part not found.\n");
}

* print: Prints a listing of all parts in the database, *
* showing the part number, part name, and *
* quantity on hand. Parts are printed in the *
* order in which they were entered into the *
* database. *

void print(void)
{
 int i;

 printf("Part Number Part Name
 "Quantity on Hand\n");
 for (i = 0; i < num_parts; i++)
 printf("%7d %-25s%11d\n", inventory[i].number,
 inventory[i].name, inventory[i].on_hand);
}

```

343

在main函数中，格式串“%c”允许scanf函数在读入操作代码之前跳过空白字符。格式串中的空格是至关重要的。如果没有它，那么scanf函数有时会读入使输入提前终止的换行符。

程序包含一个名为find\_part的函数，main函数不调用此函数。这个“辅助的”函数用来避免多余的代码和简化更重要的函数。通过调用find\_part，insert函数、search函数和update函数可以定位数据库中的零件（或者用来简单确定是否存在零件）。

这里只留了一个细节：read\_line函数。这个函数用来读零件的名字。13.3节讨论了包含书写此类函数的问题。但是13.3节开发的read\_line的写法无法正常工作在当前的程序中。请思考当用户插入零件时会发生什么：

```
Enter part number: 528
Enter part name: Disk drive
```

在录入完零件的编号后，用户按回车（或回退）键，然后再录入零件的名字后再按回车键，这样每次都给程序留下一个必须读取的无形的换行符。为了讨论的目的，现在假装这些字符都是可见的：

```
Enter part number: 528
Enter part name: Disk drive
```

当调用scanf函数来读零件编号时，函数吸收了5、2和8，但是留下了字符口未读。如果试图用原始的read\_line函数来读零件名称，那么函数将会立刻遇到字符口，并且停止读入。当数字化输入后边跟有字符输入时，这种问题非常普遍。解决办法就是编写read\_line函数，使它在开始往字符串中存储字符之前跳过空白字符。这不仅解决了换行符的问题，而且可以避免存储用户在零件名称的开始处录入的任何空格。

由于read\_line函数与invent.c文件中的其他函数无关，而且由于它在其他程序中有复用的可能，所以决定把此函数从invent.c中独立出来。read\_line函数的原型将来自头文件read.h：

```
readline.h
#ifndef READLINE_H
#define READLINE_H

/*****************
 * read_line: Skips leading white-space characters, then *
 * reads the remainder of the input line and *
 * stores it in str. Truncates the line if its *
 * length exceeds n. Returns the number of *
 * characters stored. *
 *****************/
int read_line(char str[], int n);

#endif

readline.c
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
 int ch, i = 0;

 while (isspace(ch = getchar()))
 ;

 while (ch != '\n' && ch != EOF) {
 if (i < n)
```

```

 str[i++] = ch;
 ch = getchar();
}

str[i] = '\0';
return i;
}

```

isspace函数用来判定它的实际参数是否是空白字符。15.3节解释了ch用int代替char的原因，还解释了它便于判定EOF的原因。

## 16.4 联合

像结构一样，联合（union）也是由一个或多个成员构成的，而且这些成员可能具有不同的数据类型。但是，编译器只为联合中最大的成员分配足够的内存空间。联合的成员在这个空间内彼此覆盖。这样的结果是，给一个成员赋予新值也会改变所有其他成员的值。

为了说明联合的基本性质，现在声明一个联合变量u，且这个联合变量有两个成员：

345

```

union {
 int i;
 float f;
} u;

```

注意联合的声明方式非常类似于结构的声明方式：

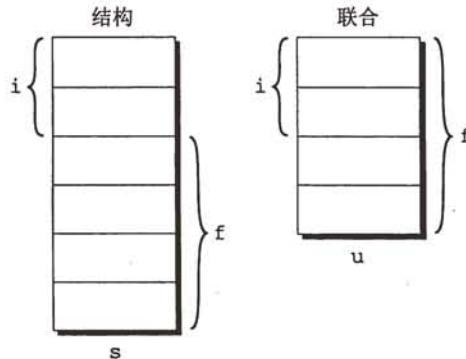
```

struct {
 int i;
 float f;
} s;

```

事实上，结构变量s和联合变量u只有一处不同：s的成员都是存储在不同的内存地址中，而u的成员都是存储在同一内存地址中。下面是s和u在内存中的存储情况（假设int类型的值要占用2个字节内存，而且float类型的值则占用4个字节内存）：

在变量s的结构中，成员i和f占有不同的内存单元。s总共占用了6个字节的内存单元。在变量u的联合中，成员i和f互相交迭（i实际上是占据了f的前两个字节内存），所以u只占用了4个字节的内存单元。如图所示，u.i和u.f具有相同的地址。



访问联合成员的方法和访问结构成员的方法相同。为了把数82存储到u的成员i中，可以写成  
u.i = 82;

为了把值74.8存储到成员f中，可以写成

u.f = 74.8;

既然编译器把联合的成员重叠存储，那么改变一个成员就会使之前存储在任何其他成员中的任

346

意值发生改变。因此，如果把一个值存储到u.f中，那么将会丢失先前存储在u.i中的任意值。（如果测试u.i的值，那么它会显示出无意义的内容。）类似地，改变u.i也会影响u.f。由于这个性质，所以可以把u想成是存储i或者存储f的地方，而不是同时存储二者的地方。（s的结构允许存储i和f。）

联合的性质和结构的性质几乎一样。所以可以用声明结构标记和类型的方法来声明联合的标记和类型。像结构一样，联合可以使用运算符=进行复制操作，也可以在函数间进行传递，还可以作为函数的返回。

联合初始化的方式甚至都和结构的初始化很类似。但是，只有联合的第一个成员可以获得初始值。例如，可以用下列方式初始化联合u的成员i为0：

```
union {
 int i;
 float f ;
} u = {0} ;
```

注意，即使初始化式是单独一个表达式，也要保证大括号的存在。大括号内的表达式必须是常量。

联合有几个重要的应用。现在将讨论其中的两种。联合的另外一个应用是关于观察存储的不同方法，由于这个应用与机器相关，所以将推迟到20.3节再介绍。

### 16.4.1 使用联合来节省空间

在结构上经常使用联合作为节省空间的一种方法。假设打算设计的结构将包含礼品册中项目的售出信息。礼品册上只有三种商品：书籍、杯子和衬衫。每个商品项目都含有库存量、价格以及和项目类型相关的其他信息：

- 书籍：书名、作者、页数。
- 杯子：设计。
- 衬衫：设计、可选颜色、可选尺寸。

首要的设计是希望结果可以具有下列结构：

```
struct catalog_item {
 int stock_number;
 float price;
 int item_type;
 char title[TITLE_LEN+1];
 char author[AUTHOR_LEN+1];
 int num_pages;
 char design[DESIGN_LEN+1];
 int colors;
 int sizes;
};
```

成员item\_type将具有值BOOK、MUG或SHIRT之一。成员colors和sizes将存储颜色和尺寸的组合代码。

347

虽然上述结构十分合用，但是它浪费了空间，因为对礼品册中的所有项目来说只有结构中的部分信息是常用到的。比如，如果项目是书籍，那么就不需要存储design、colors和sizes。通过在结构catalog\_item内部放置联合的方法，可以减少结构所要求的内存空间。联合的成员将是一些特殊的结构，每种结构都包含特定类型的商品项目所需要的数据：

```
struct catalog_item {
 int stock_number;
 float price;
 int item_type;
 union {
 struct {
```

```

char title[TITLE_LEN+1];
char author[AUTHOR_LEN+1];
int num_pages;
} book;
struct {
 char design[DESIGN_LEN+1];
} mug;
struct {
 char design[DESIGN_LEN+1];
 int colors;
 int sizes;
} shirt;
} item;
};

注意联合（名为item）是结构catalog_item的成员，而结构book、mug和shirt则是联合item的成员。如果c是表示书籍的结构catalog_item，那么可以用下列方法显示书籍的名称：

```

```
printf("%s", c.item.book.title);
```

正如上边的例子显示的那样，访问嵌套在结构内部的联合是很困难的：为了定位书籍的名称，不得不指明结构的名字（c）、结构的联合成员的名字（item）、联合的结构成员的名字（book），以及此结构的成员名（title）。C++语言通过允许匿名（anonymous）的方式使联合更容易使用。**C++**在C++语言中，在创建结构时可以忽略名字item，然后书写c.book.title来代替c.item.book.title。

## 16.4.2 使用联合来构造混合的数据结构

联合另一个重要的应用：创建含有不同数据类型的混合的数据结构。现在假设需要数组的元素是int值和float值的混合。因为数组的元素必须是相同的类型，所以好像不可能产生如此类型的数组。但是，利用联合这件事就相对容易多了。首先，定义一种联合类型，此种联合所包含的成员分别表示要存储在数组中的不同数据类型：

348

```

typedef union {
 int i;
 float f;
} Number;

```

接下来，创建一个数组，使数组的元素是具有Number类型的值：

```
Number number_array[1000];
```

数组number\_array的每个元素都是具有Number类型的联合。联合Number既可以存储int类型的值又可以存储float类型的值，所以使用此种类型值来存储数组number\_array中的int和float的混合值是可以的。例如，假设需要用数组number\_array的0号元素来存储5，而用1号元素来存储8.395。下列赋值语句可以达到期望的效果：

```
number_array[0].i = 5;
number_array[1].f = 8.395;
```

## 16.4.3 为联合添加“标记字段”

联合所面临的主要问题是：没有简便的方法可以表明联合最后改变的成员，并且因此会含有无意义的值。请思考下面这个问题。编写了一个函数，用来显示当前存储在联合Number中的值。这个函数可能有下列框架：

```

void print_number(Number n)
{
 if (n包含一个整数)
 printf("%d", n.i);
 else

```

```
 printf("%g", n.f);
}
```

可惜的是，没有办法可以帮助函数print\_number来确定n包含的是整数还是浮点数。

为了记录此信息，可以把联合嵌入一个结构中，且此结构还含有另一个成员：“标记字段”或者“判别式”，它是用来提示当前存储在联合中的内容的。在本节先前讨论的结构catalog\_item中，item\_type就是用于此目的。

下面把Number类型转换成具有嵌入联合的结构类型：

```
#define INT_KIND 0
#define FLOAT_KIND 1

typedef struct {
 int kind; /* tag field */
 union{
 int i;
 float f ;
 } u;
} Number;
```

**[349]** Number有两个成员kind和u。成员kind有两种可能的值INT\_KIND和FLOAT\_KIND。

每次给u的成员赋值时，也会改变kind，从而提示出修改的是u的哪个成员。例如，如果n是Number类型的变量，对u的成员i进行赋值操作可以采用下列形式：

```
n.kind = INT_KIND;
n.u.i = 82;
```

注意对i赋值要求首先选择n的成员u，然后才是u的成员i。

当需要找回存储在Number型变量中的数时，kind将表明联合的哪个成员是最后进行赋值的。函数print\_number可以利用这种能力：

```
void print_number(Number n)
{
 if (n.kind == INT_KIND)
 printf("%d", n.u.i);
 else
 printf("%g", n.u.f);
}
```



每次对联合的成员进行赋值时，程序负责改变标记字段的内容。如果标记字段维护数据失败，可能会导致奇怪的错误。一些编程语言采用更加安全的方法来处理标记字段，但是C语言没有做这样的尝试。

## 16.5 枚举

在一些程序中，我们会需要变量只具有少量有意义的值。例如，“布尔型”变量应该只有两种可能的值：“真值”和“假值”。用来存储扑克牌花色的变量应该只有四种隐含的值：“梅花”、“方片”、“红桃”和“黑桃”。显然可以用声明成整数的方法来处理此类变量，并且用一组代码来表示变量的可能值：

```
int s; /* s will store a suit */
s = 2; /* 2 represents "hearts" */
```

**[350]** 虽然这种方法可行，但是也遗留了许多问题。某些人读程序时可能不会意识到suit不是普通的整型变量，而且不会知道2具有特殊含义。

使用宏来定义牌的花色“类型”和不同花色的名字是一种正确的措施：

```
#define SUIT int
#define CLUBS 0
#define DIAMONDS 1
#define HEARTS 2
#define SPADES 3
```

那么前面的示例现在可以变得更加容易阅读：

```
SUIT s;
s = HEARTS;
```

这种方法是一种改进，但是它仍然不是最好的解决方案，因为这样做没有为阅读程序的人指出宏表示具有相同“类型”的值。如果可能值的数量多于这些现有的，那么为每种类型定义独立的宏是很麻烦的。而且，由于预处理器会删除已经定义的CLUBS、DIAMONDS、HEARTS和SPADES这些名字，所以在调试期间这些名字是无效的。

C语言为具有少量可能值的变量提供了特别设计的一种特殊类型。**枚举**（enumeration）是一种由程序员列出（“列举”）值的类型，而且程序员必须为每种值命名（枚举常量）。下列示例枚举的值（CLUBS、DIAMONDS、HEARTS和SPADES）可以赋值给变量s1和s2：

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

虽然枚举没有什么和结构、联合共同的地方，但是它们的声明方法很类似。然而，不像结构或联合的成员那样，枚举常量的名字必须不同于闭合作用域内声明的其他标识符。

枚举常量类似于用#define指令产生的常量，但是两者不完全一样。特别地，枚举常量是C语言的作用域规则的对象：如果枚举声明在函数体内，那么它的常量对外部函数来说是不可见的。

### 16.5.1 枚举标记和枚举类型

为了命名结构和联合的相同的原因，会常常需要创建枚举的名字。相对于结构和联合，这里有两种方法命名枚举：通过声明标记的方法，或者通过使用typedef来创建独一无二的类型名。

枚举标记类似于结构和联合的标记。例如，为了定义标记suit，可以写成

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

351

变量suit可以按照下列方法来进行声明：

```
enum suit s1, s2;
```

作为一种替换，还可以用typedef来给suit进行类型命名：

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

顺便说一下，利用typedef来创建布尔类型是非常好的一种方法：

```
typedef enum {FALSE, TRUE} Bool;
```

### 16.5.2 枚举作为整数

在系统内部，C语言会把枚举变量和常量作为整数来处理。例如，这里的枚举suit中，CLUBS、DIAMONDS、HEARTS和SPADES分别表示数0、1、2和3。

我们可以为枚举常量自由选择不同的值。现在假设希望CLUBS、DIAMONDS、HEARTS和SPADES分别表示1、2、3和4。在声明枚举时可以指明这些数：

```
enum suit {CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = 4};
```

枚举常量的值可以是任意整数，列出也可以不用按照特定的顺序：

```
enum dept {RESEARCH = 20, PRODUCTION = 10, SALES = 25};
```

两个或多个枚举常量具有相同的值甚至也是合法的。

当没有为枚举常量指定值时，它的值是一个大于前一个常量值的值。（默认第一个枚举常量的值为0。）在下列枚举中，BLACK的值为0，LT\_GRAY为7，DK\_GRAY为8，而WHITE为15：

```
enum EGA_colors {BLACK, LT_GRAY = 7, DK_GRAY, WHITE = 15};
```

因为除了稀疏地掩饰整数外，枚举的值什么也不是，所以C语言允许把它们与普通整数进行混合：

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;
i = DIAMONDS; /* i is now 1 */
s = 0; /* s is now 0 (CLUBS) */
s++; /* s is now 1 (DIAMONDS) */
i = s + 2; /* i is now 3 */
```

编译器会把s作为整型变量来处理，而名字CLUBS、DIAMONDS、HEARTS和SPADES只是数0、1、2和3的同义词。

**352** 虽然把枚举的值作为整数使用非常方便，但是把整数用作枚举的值却是非常危险**C++**的（例如，可能会在s中存储数4）。在这方面C++语言比C语言要求严格。在没有强制的情况下，C++语言不允许整数作为枚举的值来使用。

### 16.5.3 用枚举声明“标记字段”

枚举用来解决16.4节遇到的问题是非常合适的：用来确定联合中最后一个被赋值的成员。例如，在结构Number中，可以使枚举的成员kind来替代int：

```
typedef struct {
 enum {INT_KIND, FLOAT_KIND} kind;
 union {
 int i;
 float f;
 } u;
} Number;
```

这种新结构和旧结构的用法完全一样。这样做的优势是不仅远离了宏INT\_KIND和FLOAT\_KIND（他们现在是枚举常量），而且阐明了kind的含义，现在kind显然应该只有两种可能值：INT\_KIND和FLOAT\_KIND。

## 问与答

问：当试图使用`sizeof`来确定结构中的字节数量时，获得的数大于成员加在一起后的数。为什么会这样？

答：先来看看下面这个例子：

```
struct {
 char a;
 int b;
} s;
```

如果char型的值占有一个字节，而int型值占用4个字节，s会是多大呢？显然，答案是5个字节，但可能不是正确的答案。一些计算机要求数据项从某个数量字节（一般是4个字节）的倍数开始。为了满足计算机的要求，通过在邻近的成员之间留“空洞”（即无用的字节）的方法，编译器会把结构的成员“排列”起来。如果假设数据项必须从4个字节的倍数开始，那么结构s的成员a将跟着3个字节的空洞。结果是`sizeof(s)`将为8。

**353** 顺便说一句，就像在成员间有空洞一样，结构也可以在末尾有空洞。例如，

```
struct {
 int a;
```

```
char b;
} s;
```

此结构可以在成员b的后边有3个字节的空洞。

问：是否可能会在结构的开始处有“空洞”？

答：不会的。C标准说明只允许在成员之间或者最后一个成员的后边有空洞。这样的结果是结构第一个成员的地址保证和整个结构的地址完全一样。（但是，注意两个指针没有相同的类型。）

问：使用==来判定两个结构是否相等为什么是不合法的？（p.231）

答：这种操作超出了C语言的范围，因为无法实现它始终是和语言的体系相一致的。逐个比较结构成员将是极没有效率的。比较结构中的全部字节是相对较好的方法（许多计算机有特殊的指令可以用来快速执行此类比较）。然而，如果结构含有空洞，那么比较字节会产生不正确的结果。甚至是，假设对应的成员有同样的值，那么出现在空洞中的废弃值也可能会不同。这类问题可以通过下列方法解决，那就是编译器要确保空洞始终包含相同的值（比如零）。然而，初始化空洞会影响全部使用结构的程序的性能，所以它是不可行的。

问：为什么C语言提供两种命名结构类型的方法（标记命名和typedef命名）？（p.232）

答：C语言早期没有typedef，所以标记是结构类型命名的唯一有效方法。当加入typedef时，已经太晚了，以致无法删除标记了。此外，当结构的成员是指向结构自身的指针时，标记始终是非常必要的。

问：结构可否同时有标记名和类型名？（p.233）

答：可以。事实上，虽然不要求，但是标记名和类型名甚至可以是一样的：

```
typedef struct part {
 int number;
 char name [NAME_LEN+1] ;
 int on_hand;
} part ;
```

**C++** 在C++语言中，所有标记也是类型名；把part作为标记来声明也就是使part自动成为了类型名，而不再需要类型定义。

问：如何能在程序的几个文件中共享结构类型呢？

答：把结构标记（或者，如果喜欢也可以用typedef）的声明放在头文件中，然后在需要结构的地方包含此头文件就可以了。例如，为了共享结构part，可以在头文件中放入下列这行内容：

```
struct part {
 int number;
 char name [NAME_LEN+1];
 int on_hand;
};
```

注意，这里只是声明结构标记，不是声明具有这种类型的变量。

顺便提一句，含有结构标记声明或结构类型声明的头文件可能需要保护以避免多重包含（>15.2.6节）。在同一文件中声明标记或类型两次是错误的。类似的说明也适用于联合和枚举。

问：如果在两个不同的文件中包含了结构part的声明，那么一个文件中的part型变量和另一个文件中的part型变量是否一样呢？

答：技术上来说，不一样。但是，C标准提到，一个文件中的part型变量所具有的类型和另一个文件中的part型变量所具有的类型是兼容的。具有兼容类型的变量可以互相赋值，所以是“兼容的”类型和是“相同的”类型之间几乎没有差异。

## 练习

### 16.1节

1. 在下列声明中，结构x和结构y都拥有名为x和y的成员：

```
struct { int x, y; } x;
struct { int x, y; } y;
```

基于独立的基础而言，这些声明是否合法呢？两个声明是否可以同时出现在程序中呢？请证明你的想法是正确的。

2. (a) 声明名为c1、c2和c3的结构变量，每个结构变量都拥有double型的成员re和im。  
 (b) 修改(a)中的声明，使c1的成员初始值为0.0和1.0，而c2的成员初始值为1.0和0.0。(c3无初始值。)  
 (c) 编写语句用来把c2的成员复制给c1。这项操作是否可以在一条语句中完成，或者是需要两条？  
 (d) 编写语句把c1和c2的相应成员进行相加，并且把结果存储在c3中。

## 16.2节

- [355]**
3. (a) 说明如何为具有两个double型的成员re和im的结构声明名为complex的标记。  
 (b) 利用标记complex来声明名为c1、c2和c3的变量。  
 (c) 编写名为make\_complex的函数，此函数用来把两个实际参数（两个参数的类型都是double型）存储在complex型结构中，然后返回此结构。  
 (d) 编写名为add\_complex的函数，此函数用来把两个实际参数（都是complex型结构）的相应成员进行相加运算，然后返回结果（另一个complex型结构）。
  4. 重做练习3的各种操作，这次要求使用类型来命名complex。

## 16.3节

5. 下列结构用来存储图形屏幕上的对象信息。结构point用来存储屏幕上点的x轴和y轴坐标，结构rectangle用来存储矩形的左上和右下坐标点。
 

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
```

 编写函数，要求可以在rectangle型结构变量r上执行下列操作，且r作为实际参数传递。
  - (a) 计算r的面积。
  - (b) 计算r的中心，并且把此中心作为point型的值返回。
  - (c) 移动r，方法是x单元按照X轴方向移动，y单元按照y轴移动，并且返回r修改后的内容。（x和y是函数额外的实际参数。）
  - (d) 确定点p是否位于r内，返回TRUE或者FALSE。（p是具有struct point类型的额外的实际参数。）
6. 编写程序用来要求用户录入国家（地区）名称，然后可以在数组country\_codes中查找到它。如果找到对应的国家（地区）名称，程序需要显示相应的国家（地区）电话代码。如果没有找到，程序应该显示出错消息。
7. 修改程序invent.c使p（显示）操作可以根据零件数显示存储的零件。
8. 修改程序invent.c使inventory和num\_parts成为main函数的局部内容。
9. 通过为结构part添加成员price来修改程序invent.c。insert函数应该要求用户录入新的数据项的价格。serach函数和print函数应该显示价格。添加新的命令，从而允许用户改变零件的价格。

## 16.4节

10. 假设s具有下列结构：

```
struct {
 float a;
 union {
 char b[4];
 float c;
 int d;
 } e;
 char f [4];
} s;
```

如果char型值占有1个字节，int型值占有2个字节，而float型值占有4个字节，那么编译器将为s分配多大的内存空间？（假设编译器没有在成员之间留“空洞”。）

- [356]**
11. 假设s具有下列结构（point是在练习5中声明的结构标记）：

```
struct shape {
 int shape_kind; /* RECTANGLE or CIRCLE */
```

```

struct point center; /* coordinates of center */
union {
 struct {
 int length, width;
 } rectangle;
 struct {
 int radius;
 } circle;
} u;
}s;

```

请指出下列哪些语句是合法的，并且说明如何修改不合法的语句。

- (a) s.shape\_kind = RECTANGLE;
- (b) s.center.x = 10;
- (c) s.length = 25;
- (d) s.u.rectangle.width = 8;
- (e) s.u.circle = 5;
- (f) s.u.radius = 5;

12. 假设shape是练习11中声明的结构标记。编写函数用来在shape型结构变量s上完成下列操作，并且s作为实际参数在函数间传递：

- (a) 计算s的面积。
- (b) 计算s的中心，返回point型的中心值。
- (c) 移动s，方法是x单元按照x轴方向移动，y单元按照y轴移动，并且返回s修改后的内容。（x和y是函数额外的实际参数。）
- (d) 确定点p是否位于s内，返回TRUE或者FALSE。（p是具有struct point类型的额外的实际参数。）

13. 编写一个类似于invent.c的程序，利用catalog\_item型结构来存储礼品册中数据项的信息。

#### 16.5节

14. (a) 为枚举声明标记，此枚举的值表示一个星期的7天。

- (b) 用typedef定义(a)中枚举的名字。

15. 下列关于枚举常量的叙述哪些是正确的？

- (a) 枚举常量可以表示程序员指明的任何整数。
- (b) 枚举常量具有的性质和用#define产生的常量的性质完全一样。
- (c) 枚举常量的默认值为0, 1, 2, ...。
- (d) 枚举中的任何常量必须具有不同的值。
- (e) 枚举常量在表达式中可以作为整数使用。

16. 假设b和i具有如下形式的声明：

```

enum {FALSE, TRUE} b;
int i;

```

下列哪些语句是合法的？哪些是“安全的”（始终产生有意义的结果）？

357

- (a) b = FALSE;      (d) i = b;
- (b) b = i;           (e) i = 2 \* b + i;
- (c) b++;

17. (a) 棋盘的每个角上可能会有一个棋子，即兵、士、相、车、皇后或国王，也可能为空。每个棋子可能为黑色的也可能是白色的。请定义两个枚举类型Piece用来包含7种可能的值（其中一种为“空”），Color用来表示2种颜色。

- (b) 利用(a)中的类型，定义名为Square的结构类型，使此类型可以存储棋子的类型和颜色。
- (c) 利用(b)中的类型Square，声明一个名为board的 $8 \times 8$ 的数组，此数组可以用来存储棋盘上的全部内容。
- (d) 给(c)中的声明添加初始值，使board的初始值对应日常国际象棋比赛开始时的棋子布置。

358

## 第17章

## 指针的高级应用

人们只会在脑海显示复杂的信息。比如看，静景无论多么生动，都不如景色的运动、流动和改变重要。

前面的两章描述了指针的两种重要应用。第11章说明了如何利用指向变量的指针作为函数的实际参数从而允许函数修改变量。第12章说明了如何对数组元素进行指针的算术运算来处理数组。本章则通过观察两种额外的应用来完善指针的内容：动态存储分配（dynamic storage allocation）和指向函数的指针。

通过使用动态存储分配，程序可以在执行期间获得需要的内存块。17.1节解释动态存储分配的基本概念。17.2节则讨论动态分配字符串，这种方法比C语言通常的字符数组固定长度的方式更加灵活。17.3节大概地介绍数组的动态存储分配。17.4节处理存储分配的问题，即不再需要内存单元时，动态地释放已分配的内存块。

因为动态分配的结构可以链接在一起形成表、树和其他高度灵活的数据结构，所以它们在C语言编程中扮演着重要的角色。17.5节重点讲述链表（linked list），它是最基础的链式数据结构。17.6节介绍指向指针的指针，这是来自于17.5节的主题。

17.7节介绍指向函数的指针，这是非常有用的内容。C语言中一些功能最强大的库函数都期望把指向函数的指针作为实际参数。这里将考察其中一个函数qsort，它是一个对任意数组进行排序的函数。

## 17.1 动态存储分配

359 C语言的数据结构通常是固定大小的。例如，数组拥有固定数量的元素，而且每个元素具有固定的大小。因为在编写程序时强制选择了大小，所以固定大小的数据结构可能会有问题。也就是说，在没有修改程序并且再次编译程序的情况下无法改变数据结构的大小。

请思考16.3节中允许用户添加部分数据库的invent程序。数据库存储在长度为100的数组中。为了扩大数据库的容量，可以增加数组的大小并且重新编译程序。但是，无论如何增大数组，始终有可能填满数组。幸运的是没有丢失全部数据。C语言支持动态存储分配，即在程序执行期间分配内存单元的能力。利用动态存储分配，可以根据需要设计扩大（和缩小）的数据结构。

虽然可以适用于所有类型的数据，但是动态存储分配更常用于字符串、数组和结构。动态地分配结构是特别有趣的，因为可以把它们链接形成表、树或其他数据结构。

### 17.1.1 内存分配函数

为了动态地分配存储空间，将需要调用3种内存分配函数中的一种，这些函数都是声明在<stdlib.h>中的：

- malloc函数——分配内存块，但是不对内存块进行初始化。
- calloc函数——分配内存块，并且对内存块进行清除。
- realloc函数——调整先前分配的内存块。

在这3种函数中，`malloc`函数可能是最常用的一种。因为`malloc`函数不需要对分配的内存块进行清除，所以它比`calloc`函数更高效。

当为申请内存块而调用内存分配函数时，由于函数无法知道计划存储在内存块中的数据是什么类型的，所以它不能返回普通的`int`型指针或`char`型指针或其他。取而代之的，函数会返回`void*`型的值。`void*`型的值是“通用”指针，本质上它只是内存地址。

### 17.1.2 空指针

当调用内存分配函数时，无法定位满足我们需要的足够大的内存块，这种问题始终可能出现。如果真的发生了这类问题，函数会返回空指针（null pointer）。空指针是“指向为空的指针”，这是一个区别于所有有效指针的特殊值。在把返回值存储在指针变量`p`中以后，需要判断`p`是否为空指针。



程序员的责任是测试任意内存分配函数的返回值，并且在返回空指针时采取适当的操作。通过空指针试图访问内存的效果是未定义的，程序可能会崩溃或者出现不可预测的行为。

360

**Q&A** 由于用名为NULL的宏来表示空指针，所以可以用下列方式测试`malloc`函数的返回值：

```
p = malloc(10000);
if (p == NULL) {
 /* allocation failed; take appropriate action */
}
```

一些程序员把`malloc`函数的调用和NULL的测试组合在一起：

```
if ((p = malloc(10000)) == NULL) {
 /* allocation failed; take appropriate action */
}
```

名为NULL的宏在6个头文件中都有定义：`<locale.h>`、`<stddef.h>`、`<stdio.h>`、`<stdlib.h>`、`<string.h>`和`<time.h>`。只要把这些头文件中的一个包含在程序中，编译器就可以识别出NULL。当然，使用任意内存分配函数的程序都会包含`<stdlib.h>`，这使NULL必然有效。

在C语言中，指针测试真假的方法和数的测试一样。所有非空指针都为真，而只有空指针为假。因此，不用编写语句

```
if (p == NULL) ...
```

而是可以写成

```
if (!p) ...
```

而且，不用编写语句

```
if (p != NULL) ...
```

而是可以写成

```
if (p) ...
```

作为一种书写风格，这里喜欢与NULL进行明确的比较。

## 17.2 动态分配字符串

动态内存分配经常用于字符串操作。字符串始终存储在固定长度的数组中，而且可能很难预测这些数组需要的长度。通过动态地分配字符串，可以推迟到程序运行时才作决定。

### 17.2.1 使用 malloc 函数为字符串分配内存

malloc函数具有如下原型:

**[361]** `void *malloc(size_t size);`

malloc函数分配size字节的内存块，并且返回指向此内存块的指针。注意size的类型是size\_t（>21.3节），这是在C语言库中定义的无符号整数类型。除非正在分配一个非常巨大的内存块，否则可以只把size考虑成普通整数。

用malloc函数来为字符串分配内存是很容易的，因为C语言保证char型值确切需要一个字节的内存（换句话说，`sizeof(char)`的值为1。）。为给n个字符的字符串分配内存空间，可以写成

`p = malloc(n+1);`

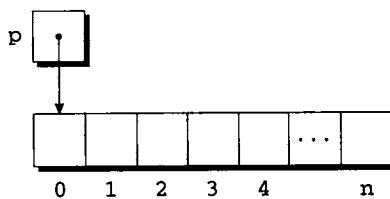
这里的p是char\*型变量。在执行复制操作时会把malloc函数返回的通用指针转化为char\*类型，而不需要强制执行。（通常情况下，可以把void\*型值赋给任何指针类型的变量。）**Q&A**然而，一些程序员喜欢强制转换malloc函数的返回值：

`p = (char *) malloc(n+1);`



当使用malloc函数为字符串分配内存空间时，不要忘记包含空字符的空间。

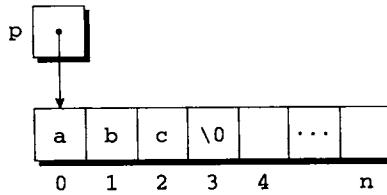
由于使用malloc函数分配内存不需要清除或者以任何方式的初始化，所以p指向带有n+1个字符的未初始化的数组：



调用strcpy函数是对上述数组进行初始化的一种方法：

`strcpy(p, "abc");`

数组中的前4个字符分别为a、b、c和\0：



### 17.2.2 在字符串函数中使用动态存储分配

**[362]** 动态存储分配使编写返回指向“新”字符串的指针的函数成为可能，所谓新字符串是指在调用此函数之前字符串并不存在。如果编写的函数不改变任何一个字符串而把两个字符串连接起来，请思考一下这样做会遇到什么问题。C标准库没有包含此类函数（因为strcat函数改变了传递过来的字符串中的一个，所以此函数并不是希望的函数），但是可以很容易的自行写出这样的函数。

自行编写的函数将测量用来连接的两个字符串的长度，然后调用malloc函数只为结果分配适当数量的内存空间。接下来函数会把第一个字符串复制到新的内存空间中，并且调用strcat函数来拼接第二个字符串。

```

char *concat(const char *s1, const char *s2)
{
 char *result;

 result = malloc(strlen(s1) + strlen(s2) + 1);
 if (result == NULL) {
 printf("Error: malloc failed in concat\n");
 exit (EXIT_FAILURE);
 }
 strcpy(result, s1);
 strcat(result, s2);
 return result;
}

```

如果malloc函数返回空指针，那么concat函数显示出错信息并且终止程序。并不总是采取正确的措施，一些程序需要从内存分配失误中恢复并且继续运行。

下面是concat函数可能的调用方式：

```
p = concat("abc", "def");
```

调用函数之后，p将指向字符串“abcdef”，此字符串是存储在动态分配的数组中的。数组包括结尾的空字符一共有7个字符长。



像concat这样动态分配存储空间的函数必须小心使用。当不再需要concat函数返回的字符串时，需要调用free函数来释放它占用的空间。如果不这样做，程序可能会过早地运行越界。

### 17.2.3 动态分配字符串的数组

13.7节解决了在数组中存储字符串的问题。我们发现按行的方式在二维字符数组中存储字符串可能会浪费空间，所以试图为字符串字面量设置为指针数组。13.7节的方法正好和数组元素是指向动态分配字符串的指针方式一样。为了说明这个观点，先来重新编写13.5节的程序remind.c，此程序显示出一个日常提示月列表。

### 17.2.4 程序：显示一个月的提示列表（改进版）

原始程序remind.c把提示字符串存储在二维字符数组中，且数组的每行包含一个字符串。在程序读入一天和相关的提示后，搜索数组从而确定这一天所处的位置，利用strcmp函数可以进行比较工作。然后，程序使用函数strcpy把该位置下面的全部字符串向下移动一个位置。最后，程序把这一天复制到数组中，并且调用strcat函数来添加这一天的提示。

在新程序中（remind2.c），数组是一维的，且数组的元素是指向动态分配的字符串的指针。在此程序中换成功能的字符串会有两个主要好处。第一，与提示存储在固定数量的字符数组中（就如同原始程序所做的那样）相比，为要存储的提示分配确切字符数量的空间，这样做可以更有效地利用空间。第二，不需要为一个新提示留空间而调用函数strcpy来把字符串“降低”，而只是移动指向字符串的指针。

下面是新程序，程序中有改动的部分用粗体进行了标注。把二维数组换成指针数组显得异常容易：只需要改变程序的8行内容。

```

remind2.c

/* Prints a one-month reminder list */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```

#define MAX_REMIND 50
#define MSG_LEN 60

int read_line(char str[], int n);

main()
{
 char *reminders[MAX_REMIND];
 char day_str[3], msg_str[MSG_LEN+1];
 int day, i, j, num_remind = 0;

 for (;;) {
 if (num_remind == MAX_REMIND) {
 printf("-- No space left --\n");
 break;
 }

 printf("Enter day and reminder: ");
 scanf("%2d", &day);
 if (day == 0)
 break;
 sprintf(day_str, "%2d", day);
 read_line(msg_str, MSG_LEN);

 for (i = 0; i < num_remind; i++)
 if (strcmp(day_str, reminders[i]) < 0)
 break;
 for (j = num_remind; j > i; j--)
 reminders[j] = reminders[j-1];

 reminders[i] = malloc(2 + strlen(msg_str) + 1);
 if (reminders[i] == NULL) {
 printf("-- No space left --\n");
 break;
 }

 strcpy(reminders[i], day_str);
 strcat(reminders[i], msg_str);

 num_remind++;
 }

 printf("\nDay Reminder\n");
 for (i = 0; i < num_remind; i++)
 printf(" %s\n", reminders[i]);
}

return 0;
}

int read_line(char str[], int n)
{
 char ch;
 int i = 0;

 while ((ch = getchar()) != '\n')
 if (i < n)
 str[i++] = ch;

 str[i] = '\0';
 return i;
}

```

## 17.3 动态分配数组

动态分配数组会获得和动态分配字符串相同的好处（不用惊讶，因为字符串就是数组）。当

编写程序时，常常很难为数组估计合适的大小。较方便的做法是等到程序运行时再来确定数组的实际大小。C语言解决了这个问题，方法是允许在程序执行期间为数组分配空间，然后通过指向数组第一个元素的指针访问数组。数组和指针之间的紧密关系已经在第12章中讨论过了，使用动态分配的数组就好像使用普通数组一样简单。

365

虽然malloc函数可以为数组分配内存空间，但calloc函数确实是最常使用的一种选择，因为calloc函数对分配的内存进行初始化。realloc函数允许根据需要对数组进行“扩展”或“缩减”。

### 17.3.1 使用 malloc 函数为数组分配存储空间

可以使用malloc函数为数组分配存储空间，这种方法和用它为字符串分配空间非常相像。主要区别就是任意数组的元素不需要像字符串那样是一个字节的长度。这样的结果是，会需要使用sizeof运算符（>7.4节）来计算出每个元素所需的空间数量。

假设正在编写的程序需要n个整数的数组，这里的n可以在程序执行期间计算出来。首先需要声明指针变量：

```
int *a;
```

一旦n的值已知了，就让程序调用malloc函数为数组分配存储空间：

```
a = malloc(n * sizeof(int));
```



当计算数组所需的空间数量时始终要使用sizeof运算符。如果分配的内存空间数量不够会产生严重的后果。思考下面的语句，此语句试图为n个整数的数组分配空间：

```
a = malloc(n * 2);
```

如果int型值大于两个字节，那么malloc函数将无法分配足够大的内存块。当稍后往数组中存储值时，程序可能会崩溃或者行为异常。

一旦a指向动态分配的内存块，就可以忽略a是指针的事实，并且把它用作数组的名字。这都要感谢C语言中数组和指针的紧密关系。例如，可以使用下列循环对a指向的数组进行初始化：

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

当然，用指针的算术运算代替下标来访问数组元素的方法是可以选择的。

### 17.3.2 calloc 函数

虽然malloc函数可以用来为数组分配内存，但是C语言还提供了另外一种选择，即calloc函数，此函数有时会更好用一些。calloc函数在<stdlib.h>中具有如下所示的原型：

```
void *calloc(size_t nmemb, size_t size);
```

calloc函数为nmemb个元素的数组分配内存空间，其中每个元素的长度都是size个字节。如果要求的空间无效，那么此函数返回空指针。**Q&A**在分配了内存之后，calloc函数会通过对所有位设置为0的方式进行初始化。例如，下列calloc函数的调用为n个整数的数组分配存储空间，并且保证全部初始为零：

```
a = calloc(n, sizeof(int));
```

既然calloc函数会清除分配的内存，而malloc函数不会，那么可能有时需要使用calloc函数为非空数组分配空间。通过调用以1作为第一个实际参数的calloc函数，可以为任何类型的数据项分配空间：

```
struct point { int x, y; } *p;
p = calloc(1, sizeof(struct point));
```

在执行此语句之后，p将指向结构，且此结构的成员x和y都会被设为零。

366

### 17.3.3 realloc 函数

一旦为数组分配完内存，稍后可能会发现数组过大或过小。realloc函数可以调整数组的大小使它更适合需要。下列realloc函数的原型出现在`<stdlib.h>`中：

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

当调用realloc函数时，ptr必须指向内存块，且此内存块一定是先前通过malloc函数、calloc函数或realloc函数的调用获得的。size表示内存块的新尺寸，新尺寸可能会大于或小于原有尺寸。虽然realloc函数不要求ptr指向正在用作数组的内存，但实际上通常是这样的。



**要确定传递给realloc函数的指针来自于先前malloc函数、calloc函数或realloc函数的调用。如果不是这样的指针，那么程序可能会行为异常。**

C标准列出几条关于realloc函数的规则：

- 当扩展内存块时，realloc函数不会对添加进内存块的字节进行初始化。
- 如果realloc函数不能按要求扩大内存块，那么它会返回空指针，并且在原有的内存块中的数据不会发生改变。
- 如果realloc函数调用时以空指针作为第一个实际参数，那么它的行为就将像malloc函数一样。
- 如果realloc函数调用时以0作为第二个实际参数，那么它会释放掉内存块。

367

C标准中明确地包含了一些关于realloc函数行为的规则。尽管如此，我们仍然希望它适当有效。在要求减少内存块大小时，realloc函数应该“在适当位置”缩减内存块，而不需要移动存储在内存块中的数据。由于同样的用意，realloc函数应该始终试图扩大内存块而不需要对其进行移动。如果无法扩大内存块（因为内存块后边的字节已经用于其他目的），realloc函数会在别处分配新的内存块，然后把旧块中的内容复制到新块中。



**一旦realloc函数返回，请一定要对指向内存块的所有指针进行更新，因为可能realloc函数移动了其他地方的内存块。**

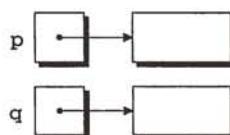
## 17.4 释放存储

malloc函数和其他内存分配函数所获得的内存块都来自一个称为堆（heap）的存储池。调用这些函数经常会耗尽堆，或者要求大的内存块也可能耗尽堆，这会导致函数返回空指针。

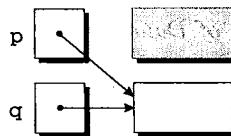
更糟的是，程序可能分配了内存块，然后又丢失了这些块的追踪路径，因而浪费了空间。请思考下面的例子：

```
p = malloc(...)
q = malloc(...)
p = q;
```

在执行完前两条语句后，p指向了一个内存块，而q指向了另一个内存块：



在把q赋值给p之后，两个指针现在都指向了第二个内存块：



368

因为没有指针指向第一个内存块（图上阴影部分），所以再也不能使用此内存块了。

对程序而言不再访问到的内存块被称为是垃圾（garbage）。在后边留有垃圾的程序有内存泄漏（memroy leak）。一些语言提供垃圾收集器（garbage collector）用于垃圾的自动定位和回收，但是C语言不提供。相反，每个C程序负责回收各自的垃圾，方法是调用free函数来释放不需要的内存。

#### 17.4.1 free 函数

free函数在<stdlib.h>中有下列原型：

```
void free(void *ptr);
```

使用free函数很容易，只是简单地把指向不再需要内存块的指针传递给free函数就可以了：

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

调用free函数来释放p所指向的内存块。然后会把这个释放的内存返回给堆，也就是使此内存块可以被后续的malloc函数或其他内存分配函数的调用可以复用。



free函数的实际参数必须是指针，而且此指针一定是被先前内存分配函数返回的。调用不带其他实际参数（比如，指向变量或数组元素的指针）的free函数可能导致不可预测的行为。

#### 17.4.2 “悬空指针”问题

虽然free函数允许收回不再需要的内存，但是使用此函数会导致一个新的问题：悬空指针（dangling pointer）。调用free(p)函数会释放p指向的内存块，但是不会改变p本身。如果忘记了p不再指向有效内存块，混乱可能随即而来：

```
char *p = malloc(4);
...
free(p);
...
strcpy(p, "abc"); /*** WRONG ***/
```

修改p指向的内存是严重的错误，因为程序不再对此内存有任何控制权了。

369



试图修改释放掉的内存块会有程序崩溃等损失惨重的后果。

悬空指针是很难发现的，因为几个指针可能指向相同的内存块。在释放内存块时，全部的指针都会留有悬空。

---

## 17.5 链表

动态存储分配对建立表、树、图和其他链接数据结构是特别有用的。本节将会介绍链表，而对其他链接数据结构的讨论超出了本书的范畴。为了获取更多的信息，可以参考Horowitz、

Sahni和Anderson-Freed的*Fundamentals of Data Structure in C*这样的书 (New York: Computer Science Press, 1993)。

链表 (Linked List) 是由一连串的结构 (称为结点) 组成的, 其中每个结点都包含指向下一个链中结点的指针:



表中的最后一个结点包含一个空指针, 图上用斜线表示出来。

在前面几章中我们已经知道, 无论何时需要存储数据项的集合时都可以使用数组, 而现在链表为我们提供了另外一种选择。链表比数组更灵活, 我们可以很容易地在链表中插入和删除结点, 也就是说允许链表根据需要扩大和缩小。另一方面, 也失去了“随机访问”数组的能力。我们可以用相同的时间访问到数组内的任何元素, 而访问链表中的结点用时不同。如果结点距离链表的开始处很近, 那么访问到它会很快。反之, 若结点靠近链表结尾处, 访问到它就很慢。

本节会描述在C语言中建立链表的方法, 还将说明如何对链表执行几个常见的操作, 即在链表开始处插入结点、搜索结点和删除结点。

### 17.5.1 声明结点类型

为了建立链表, 第一件事就是需要一个表示表中单个结点的结构。为了简化, 先假设结点只包含一个整数 (即结点的数据) 和指向表中下一个结点的指针。下面是结点结构的描述:

```

struct node {
 int value; /* data stored in the node */
 struct node *next; /* pointer to the next node */
};

```

**370** 注意, 成员next具有`struct node*`类型, 这就意味着它能存储一个指向node型结构的指针。顺便说一下, 关于名字node没有任何特殊含义, 只是一个普通的结构标记。

node型结构的一个方面受到了特别的关注。正如16.2节说明的那样, 通常可以选择使用标记或者用`typedef`来定义一种特殊的结构类型的名字。但是, 在结构有一个**Q&A**指向相同结构类型的成员时, 就像node做的那样, 要求使用结构标记。没有node标记, 就没有办法声明next的类型。

现在有了声明好的node型结构, 就需要有跟踪表开始位置的方法。换句话说, 需要变量始终指向表中的第一个结点。这里给此变量命名为first:

```
struct node *first = NULL;
```

把first初始化为NULL就说明链表初始为空。

### 17.5.2 创建结点

在构建链表时, 需要逐个创建结点, 并且把生成的每个结点加入到链表中。创建结点包括3个步骤:

- (1) 为结点分配内存单元;
- (2) 把数据存储到结点中;
- (3) 把结点插入到链表中。

这里将集中介绍前两个步骤。

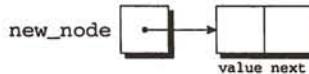
为了创建结点, 需要一个变量临时指向该结点, 并且直到该结点插入链表中为止。把此变量设为new\_node:

```
struct node *new_node;
```

可以使用malloc函数为新结点分配内存空间，并且把返回值保存在new\_node中：

```
new_node = malloc(sizeof(struct node));
```

现在new\_node指向了一个内存块，且此内存块正好放下一个node型结构：



**小心** sizeof给出的是分配的类型的名字，而不是指向此类型的指针的名字：

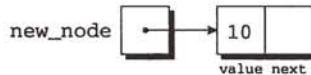
```
new_node = malloc(sizeof(new_node)); // *** WRONG ***
```

程序会始终编译，但是malloc函数将只为指向node型结构的指针分配足够的内存单元，而不是为结构本身分配。当程序试图把数据存储到new\_node可能指向的结点中时，结果是稍后可能会崩溃。

371

接下来，将把数据存储到新结点的成员value中：

```
(*new_node).value = 10;
```



为了访问到结点的成员value，可以采用间接寻址运算符\*（为了引用new\_node指向的结构），然后选择运算符.（为了选择此结构内的一个成员）。在\*new\_node周围的圆括号是强制要求的，因为运算符.的优先级高于运算符\*（运算符表）（见附录B）。

### 17.5.3 ->运算符

在继续介绍往链表中插入新结点之前，先来讨论一种有用的捷径。利用指针访问结构中的成员是如此的普遍，以致C语言针对此目的还提供了一种特殊的运算符，此运算符被称为右箭头选择（right arrow selection），它是由一个减号—跟着一个>组成的。利用运算符->可以编写语句

```
new_node->value = 10;
```

来代替语句

```
(*new_node).value = 10;
```

运算符->是运算符\*和运算符.的组合，即为了定位它指向的结构，先对new\_node间接寻址，然后再选择结构的成员value。

由于运算符->产生左值（见4.2.2节），所以可以在任何允许普通变量的地方使用它。恰好已经看到一个new\_node->value出现在赋值运算的左侧的例子。这就像出现在scanf调用中那样容易：

```
scanf("%d", &new_node->value);
```

注意，即使new\_node是一个指针，运算符&始终也是需要的。没有运算符&，就会把new\_node->value的值传递给scanf函数，而这个值是int类型。

372

### 17.5.4 在链表的开始处插入结点

现在准备往链表中插入新结点了。链表的好处之一就是在表中的任何位置添加结点：在开始处、在结尾处或者中间的任何位置。然而，链表的开始处是最容易插入结点的地方，所以这里集中讨论这种情况。

如果new\_node正指向要插入的结点，并且first正指向链表中的首结点，那么为了把结点插入链表将需要两条语句。首先，为了指向先前在链表开始处的结点，将要修改结点的成员next：

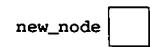
```
next_node->next = first;
```

接下来，要使first指向新结点：

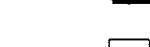
```
first = new_node;
```

如果在插入结点时链表为空，那么这些语句是否还能工作呢？幸运的是，可以。为了确信这是真的，一起来跟踪一下在空链表中插入两个结点的过程。首先，将插入的结点含有数10，跟着要插入的结点还有数20。在下图中空指针用斜线表示。

```
first = NULL;
```



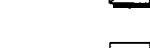
```
new_node = malloc(sizeof(struct node));
```



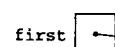
```
new_node->value = 10;
```



```
new_node->next = first;
```



```
first = new_node;
```



```
new_node = malloc(sizeof(struct node));
```



```
new_node->value = 20;
```



```
new_node->next = first;
```



```
first = new_node;
```



往链表中插入结点是经常用到的操作，所以希望为此目的编写一个函数。现在把此函数命名为add\_to\_list。此函数有两个形式参数：list（指向旧链表中首结点的指针）和n（存储在新结点中的整数）。

```
struct node *add_to_list(struct node *list, int n)
{
 struct node *new_node;
 new_node = malloc(sizeof(struct node));
 if (new_node == NULL) {
 printf("Error: malloc failed in add_to_list\n");
 exit(EXIT_FAILURE);
 }
 new_node->value = n;
 new_node->next = list;
 return new_node;
}
```

注意，add\_to\_list函数不会修改指针list，而是返回指向新产生的结点的指针（现在位于链表的开始处）。当调用add\_to\_list函数时，需要把它的返回值存储到first中：

```
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

上述语句为first指向的链表增加了含有10和20的结点。用add\_to\_list函数直接更新first，而不是为first返回新的值，这样做是个技巧。17.6节将回到这个问题。

374

下列函数用add\_to\_list来创建一个含有用户录入数的链表：

```
struct node *read_numbers(void)
{
 struct node * first = NULL;
 int n;

 printf("Enter a series of integers (0 to terminate):");
 for (;;) {
 scanf("%d", &n);
 if (n == 0)
 return first;
 first = add_to_list(first, n);
 }
}
```

链表内的数将会发生顺序倒置，因为first始终指向包含最后录入数的结点。

### 17.5.5 搜索链表

一旦创建了链表，可能就需要为某个特殊的数据段而搜索链表。虽然while循环可以用于搜索链表，但是for语句却常常是首选。在编写含有计数操作的循环时，我们习惯使用for语句，但是for语句的灵活性使它也适合其他工作，包括对链表的操作。下面是一种搜索链表的习惯方法，使用了指针变量p来跟踪“当前”结点：

**[惯用法]** `for (p = first; p != NULL; p = p->next)`

...

赋值表达式`p = p->next`使指针p从一个结点移动到下一个结点。当编写遍历链表的循环时，在C语言中总是采用这种形式的赋值表达式。

现在编写名为search\_list函数，此函数为找到整数n而搜索链表（由形式参数list指向）。如果找到n，那么search\_list函数将返回指向含有n的结点的指针；否则，它会返回空指针。search\_list函数的第一种形式依赖于惯用的“链表搜索”惯用法：

```
struct node *search_list(struct node *list, int n)
```

```

{
 struct node *p;

 for (p = list; p != NULL; p = p->next)
 if (p->value == n)
 return p;
 return NULL;
}

```

375

当然，还有许多其他方法可以编写search\_list函数。其中一种替换方式是除去变量p，而用list自身来代替进行当前结点的跟踪：

```

struct node *search_list(struct node *list, int n)
{
 for (; list != NULL; list = list->next)
 if (list->value == n)
 return list;
 return NULL;
}

```

因为list是原始链表指针的副本，所以在函数内改变它不会有任何损害。

另一种替换方法是把判定list->value == n和判定list != NULL合并起来：

```

struct node *search_list(struct node *list, int n)
{
 for (; list != NULL && list->value != n;
 list = list->next)
 ;
 return list;
}

```

因为到达链表末尾处list会为NULL，所以即使找不到n，返回的list也是正确的。如果使用while语句，那么search\_list函数的此种形式可能会更加清楚：

```

struct node *search_list(struct node *list, int n)
{
 while (list != NULL && list->value != n)
 list = list->next;
 return list;
}

```

### 17.5.6 从链表中删除结点

把数据存储到链表中一个很大的好处就是可以轻松删除不需要的结点。就像产生结点一样，删除结点也包含3个步骤：

- (1) 定位要删除的结点；
- (2) 改变前一个结点，从而使它“绕过”删除结点；
- (3) 调用free函数从而收回删除结点占用的内存空间。

第1步看起来比较困难。如果按照显而易见的方式搜索链表，那么将在指针指向要删除的结点时终止搜索。但是，这样做就不能执行第2步了，因为第2步要求改变前一个结点。

针对这个问题有各种不同的解决办法。这里将使用“追踪指针”的方法：在第1步搜索链表时，将保留一个指向前一个结点的指针（prev），还有指向当前结点的指针（cur）。如果list指向搜索的链表，并且n是要删除的整数，那么下列循环就可以实现第1步：

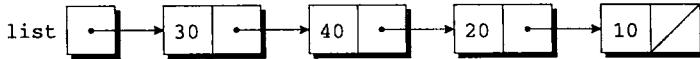
```

for (cur = list, prev = NULL;
 cur != NULL && cur->value != n;
 prev = cur, cur = cur->next)
;

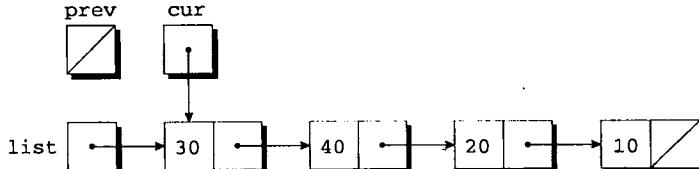
```

这里我们看到了C语言中for语句的威力。这是个很奇异的示例，它采用了空循环体和逗号运算符的丰富应用，执行的全部操作都是为了搜索到n。当循环终止时，cur指向要删除的结点，而prev指向前一个结点（如果这里有的话）。

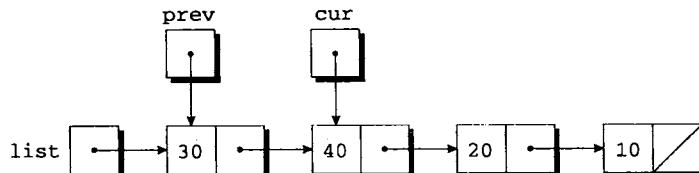
为了看清楚这个循环的工作过程，现在假设list指向依次含有30、40、20和10的链表：



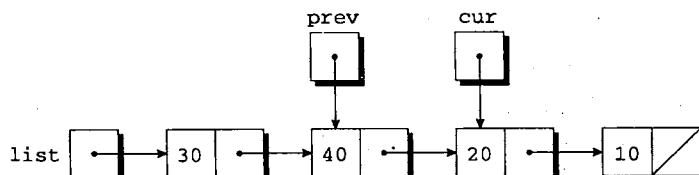
假设n为20，那么目标就是删除此链表中的第3个结点。在执行完`cur = list, prev = NULL`后，`cur`指向了链表中的第1个结点：



因为`cur`正指向一个结点，且此结点不含有20，所以判断表达式`cur != NULL && cur->value != n`为真。在执行完`prev = cur, cur = cur->next`后，开始看到指针`prev`跟踪在指针`cur`的后边：



再次判断条件表达式`cur != NULL && cur->value != n`为真，所以再次执行`prev = cur, cur = cur->next`：



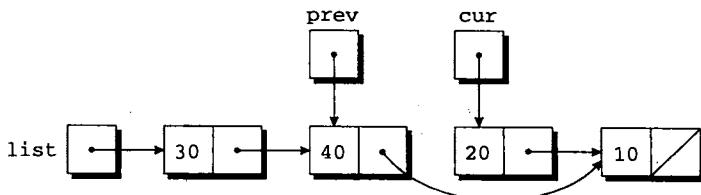
377

因为`cur`此时指向了含有20的结点，所以条件表达式`cur != NULL && cur->value != n`为假，从而循环终止。

接下来，将根据步骤2的要求执行绕过操作。语句

```
prev->next = cur->next;
```

使前一个结点中的指针指向了当前结点后面的结点：



现在准备完成步骤3，即释放当前结点占用的内存：

```
free(cur);
```

下列函数所使用的策略就是刚刚概述的操作。在给定链表和整数n时，`delete_from_list`函数就会删除含有n的第一个结点。如果没有含有n的结点，那么函数什么也不做。无论上述哪

种情况，函数都返回指向链表的指针。

```
struct node *delete_from_list(struct node *list, int n)
{
 struct node *cur, *prev;

 for (cur = list, prev = NULL;
 cur != NULL && cur->value != n;
 prev = cur, cur = cur->next)
 ;

 if (cur == NULL)
 return list; /* n was not found */
 if (prev == NULL)
 list = list->next; /* n is in the first node */
 else
 prev->next = cur->next; /* n is in some other node */
 free (cur);
 return list;
}
```

删除链表中的首结点是一种特殊情况。判定表达式`prev == NULL`会检查这种情况，这需要一种不同的绕过步骤。

### 17.5.7 链表排序

378 当链表的结点保持顺序时，我们把对结点内存储的数据进行排序称为是链表的排序。对有序链表的操作和对无序链表的操作相似。虽然往有序列表中插入结点会更困难一些（不再始终把结点放置在链表的开始处），但是搜索会更快（在到达期望结点定位的地方之后，可以停止查看操作）。下面一节中的程序既说明了插入结点增加了难度，也说明了搜索更加快速。

### 17.5.8 程序：维护零件数据库（改进版）

下面重做16.3节的零件数据库程序，这次把数据库存储在链表中。用链表代替数组有两个主要的好处：（1）不需要事先限制数据库的大小，数据库可以扩大到没有更多内存空间存储零件为止；（2）可以很容易保持用零件编号排序的数据库，当往数据库中添加新零件时，只是简单把它插入链表中的适当位置就可以了。在原来的程序中，数据库不能排序。

在新程序中，结构`part`将包含一个额外的成员（指向链表中下一个结点的指针），而且变量`inventory`是指向链表首结点的指针：

```
struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
 struct part *next;
};

struct part *inventory = NULL; /* points to first part */
```

新程序中的大多数函数非常类似于它们在原始程序中的副本。然而，函数`find_part`和函数`insert`变得更加复杂了，因为把结点保留在用零件编号排序的链表`inventory`中。

在原来的程序中，函数`find_part`返回数组`inventory`的索引。而在新程序中，函数`find_part`返回指针，此指针指向的结点含有需要的零件编号。如果没有找到零件编号，函数`find_part`会返回空指针。既然链表`inventory`是根据零件编号排序的，新版本的函数`find_part`就可以节约时间，方法是在找到含有特定零件编号的结点时停止搜索，其中此特定零件编号是大于或等于需要的零件编号的。函数`find_part`的搜索循环将有如下形式：

```
for (p = inventory;
 p != NULL && number > p->number;
 p = p->next)
;
```

当p变为NULL时（说明没有找到零件编号）或者当number > p->number为假时（说明找到的零件编号小于或等于已经存储在结点中的数），循环将终止。在后一种情况下，始终无法知道需要的数组是否真的在链表中，所以将需要另一种判断：

```
if (p != NULL && number == p->number)
 return p;
```

原始版本的函数insert把新零件存储在下一个有效的数组元素中；新版本的函数需要确定新零件在链表中所处的位置，并且把它插入到那个位置。函数insert还要检查零件编号是否已经出现在链表中了。函数insert可以通过使用类似于函数find\_part中的一个循环来完成这两项任务：

```
for (cur = inventory, prev = NULL;
 cur != NULL && new_node->number > cur->number;
 prev = cur, cur = cur->next)
;
```

此循环依赖于两个指针：指向当前结点的指针cur和指向前一个结点的指针prev。一旦终止循环，函数insert将检查是否cur不为NULL，而且new\_node->number是否等于cur->number。如果条件成立，那么零件的编号已经在链表中了。否则，函数insert将把新结点插入到prev和cur指向的结点之间；所使用的策略类似于删除结点所采用的方法。（即使新零件的编号大于链表中的任何编号，此策略仍然有效。这种情况下，cur将为NULL，而prev将指向链表中的最后一个结点。）

下面是新程序。和原始程序一样，此版本需要16.3节描述的函数read\_line。假设realine.h含有此函数的原型。

### **invent2.c**

```
/* Maintains a parts database (linked list version) */

#include <stdio.h>
#include <stdlib.h>
#include "readline.h"

#define NAME_LEN 25

struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
 struct part *next;
};

struct part *inventory = NULL; /* points to first part */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
/***
 * main: Prompts the user to enter an operation code, *
 * then calls a function to perform the requested *
 * action. Repeats until the user enters the *
 * command 'q'. Prints an error message if the user *
 * enters an illegal code. *
**/
```

379

380

```

main()
{
 char code;

 for (;;) {
 printf("Enter operation code: ");
 scanf(" %c", &code);
 while (getchar() != '\n') /* skips to end of line */
 ;
 switch (code) {
 case 'i': insert();
 break;
 case 's': search();
 break;
 case 'u': update();
 break;
 case 'p': print();
 break;
 case 'q': return 0;
 default: printf("Illegal code\n");
 }
 printf("\n");
 }

 /*****
 * find_part: Looks up a part number in the inventory
 * list. Returns a pointer to the node
 * containing the part number; if the part
 * number is not found, returns NULL.
 *****/
 struct part *find_part(int number)
 {
 struct part *p;

 for (p = inventory;
 p != NULL && number > p->number;
 p = p->next)
 ;
 if (p != NULL && number == p->number)
 return p;
 return NULL;
 }

 /*****
 * insert: Prompts the user for information about a new
 * part and then inserts the part into the
 * inventory list; the list remains sorted by
 * part number. Prints an error message and
 * returns prematurely if the part already exists
 * or space could not be allocated for the part.
 *****/
 void insert(void)
 {
 struct part *cur, *prev, *new_node;

 new_node = malloc(sizeof(struct part));
 if (new_node == NULL) {
 printf("Database is full; can't add more parts.\n");
 return;
 }

 printf("Enter part number: ");
 scanf("%d", &new_node->number);
 }
}

```

381

```

for (cur = inventory, prev = NULL;
 cur != NULL && new_node->number > cur->number;
 prev = cur, cur = cur->next)
;
if (cur != NULL && new_node->number == cur->number) {
 printf("Part already exists.\n");
 free(new_node);
 return;
}

printf("Enter part name: ");
read_line(new_node->name, NAME_LEN);
printf("Enter quantity on hand: ");
scanf("%d", &new_node->on_hand);

new_node->next = cur;
if (prev == NULL)
 inventory = new_node;
else
 prev->next = new_node;
}

/***
* search: Prompts the user to enter a part number, then *
* looks up the part in the database. If the part *
* exists, prints the name and quantity on hand; *
* if not, prints an error message. *
***/
void search(void)
{
 int number;
 struct part *p;

 printf("Enter part number: ");
 scanf("%d", &number);
 p = find_part(number);
 if (p != NULL) {
 printf("Part name: %s\n", p->name);
 printf("Quantity on hand: %d\n", p->on_hand);
 } else
 printf("Part not found.\n");
}

/***
* update: Prompts the user to enter a part number. *
* Prints an error message if the part doesn't *
* exist; otherwise, prompts the user to enter *
* change in quantity on hand and updates the *
* database. *
***/
void update(void)
{
 int number, change;
 struct part *p;

 printf("Enter part number: ");
 scanf("%d", &number);
 p = find_part(number);
 if (p != NULL) {
 printf("Enter change in quantity on hand: ");
 scanf("%d", &change);
 p->on_hand += change;
 } else
}

```

```

 printf("Part not found.\n");
}

 * print: Prints a listing of all parts in the database, *
 * showing the part number, part name, and *
 * quantity on hand. Part numbers will appear in *
 * ascending order.

void print(void)
{
 struct part *p;

 printf("Part Number Part Name
 "Quantity on Hand\n");
 for (p = inventory; p != NULL; p = p->next)
 printf("%7d %-25s%11d\n", p->number,
 p->name, p->on_hand);
}

```

注意函数insert中free的用法。函数insert在检查零件是否已经存在之前就为零件分配内存空间。如果真是这样，那么函数insert释放内存以便程序不会冒险运行过早越界。

## 17.6 指向指针的指针

在13.7节中，已经遇到过指向指针的指针。在那一节中，使用了元素类型为char \*的数组，以及一个指向数组元素的指针，此指针的类型为char\*\*。“指向指针的指针”概念也频繁出现在链式数据结构的内容中。特别是，当函数的实际参数是指针变量时，有时候会希望函数能通过指针指向别处的方式改变此变量。做这项工作就需要用到指向指针的指针。

请思考一下17.5节中的函数add\_to\_list，此函数用来在链表的开始处插入结点。当调用函数add\_to\_list时，我们会传递给它指向链表内首结点的指针，然后函数会返回指向新链表的指针：

```

struct node *add_to_list(struct node *list, int n)
{
 struct node *new_node;

 new_node = malloc(sizeof(struct node));
 if(new_node == NULL) {
 printf("Error: malloc failed in add_to_list\n");
 exit (EXIT_FAILURE);
 }
 new_node->value = n;
 new_node->next = list;
 return new_node;
}

```

假设修改了函数使它不再返回new\_node，而是把new\_node赋值给list。换句话说，把return语句从函数add\_to\_list中移走，同时用下列语句进行替换：

```
list = new_node;
```

可惜的是，这个想法无法实现。假设按照下列方式调用函数add\_to\_list：

```
add_to_list(first, 10);
```

在调用点，会把first复制给list。（像所有其他实际参数一样，指针进行了按值传递。）函数内的最后一行改变了list的值，使它指向了新的结点。但是，此赋值操作对first没有影响。

可以让函数add\_to\_list修改first，但是这就要求给函数add\_to\_list传递一个指向first的指针。下面是此函数的正确形式：

384

```

void _add_to_list(struct node **list, int n)
{
 struct node *new_node;
 new_node = malloc(sizeof(struct node));
 if (new_node == NULL) {
 printf("Error: malloc failed in add_to_list\n");
 exit(EXIT_FAILURE);
 }
 new_node->value = n;
 new_node->next : *list;
 *list = new_node;
}

```

当调用新版本的函数add\_to\_list时，第一个实际参数将会是first的地址：

```
add_to_list(&first, 10);
```

既然给list赋予了first的地址，那么可以使用\*list作为first的别名。特别是，把new\_node赋值给\*list将会修改first的内容。

## 17.7 指向函数的指针

到目前为止，已经使用指针指向过各种类型的数据，包括变量、数组元素以及动态分配的内存块。但是C语言没有要求指针只能指向数据，它还允许指针指向函数。指向函数的指针（函数指针）不像人们想象的那样奇怪。毕竟函数占用内存单元，所以每个函数都有地址，就像每个变量都有地址一样。

### 17.7.1 函数指针作为实际参数

可以以使用数据指针相同的方式使用函数指针。在C语言中把函数指针作为实际参数进行传递是十分普遍的。假设编写了名为integrate的函数用来求a点和b点之间的函数f的积分。我们希望函数integrate通过传递f作为实际参数的方式变得更为通用。为了在C语言中达到这种效果将把f声明为指向函数的指针。假设希望求积分函数具有double型形式参数，并且返回double型的结果，函数integrate的原型如下所示：

```
double integrate(double (*f)(double), double a, double b);
```

在\*f周围的圆括号说明f是个指向函数的指针，而不是函数的返回值为指针。把f声明成好像就是函数也是合法的：

```
double integrate(double f(double), double a, double b);
```

从编译器的标准来看，这种原型和前一种形式是完全一样的。

在调用函数integrate时，将把一个函数名作为第一个实际参数。例如，下列调用将计算sin函数（>23.3.2节）从0到 $\pi/2$ 之间的积分：

```
result = integrate(sin, 0.0, PI/2);
```

385

注意，在sin的后边没有圆括号。当函数名后边没跟着圆括号时，C语言编译器会产生指向函数的指针来代替产生函数调用的代码。在此例中，不是在调用函数sin，而是给函数integrate传递了一个指向函数sin的指针。如果这样看上去很混乱的话，可以想想C语言处理数组的过程。如果a是数组的名字，那么a[i]就表示数组的一个元素，而a本身则作为指向数组的指针。类似的方法，如果f是函数，那么C语言把f(x)看成是函数的调用来处理，而f本身则是指向函数的指针。

在integrate函数体内，可以调用f所指向的函数：

```
sum += (*f)(x);
```

`*f`表示`f`所指向的函数，而`x`是函数调用的实际参数。因此，在函数`integrate(sin, 0.0, PI/2)`执行期间，`*f`的每次调用实际上都是`sin`函数的调用。作为`(*f)(x)`一种替换选择，C语言允许用`f(x)`来调用`f`所指向的函数。虽然`f(x)`看上去更自然一些，但是这里将坚持把`(*f)(x)`作为`f`是指向函数的指针而不是函数名的提示。

### 17.7.2 qsort 函数

虽然指向函数的指针看似对日常编程没有什么用处，但是从事实来看这是没有远见的。实际上，C函数库中一些功能最强大的函数要求把函数指针作为实际参数。**Q&A**其中之一就是函数`qsort`，此函数的原型可以在`<stdlib.h>`中找到。函数`qsort`是给任意数组排序的通用函数。

因为数组的元素可能具有任何类型，甚至是结构或联合，所以必须告诉函数`qsort`如何确定两个数组元素哪一个“更小”。通过编写比较函数可以为函数`qsort`提供这些信息。当给定两个指向数组元素的指针`p`和`q`时，比较函数必须返回一个数。如果`*p`“小于”`*q`，那么返回的数为负数。如果`*p`“等于”`*q`，那么返回的数为零。如果`*p`“大于”`*q`，那么返回的数为正数。这里把“小于”、“等于”和“大于”放在双引号中是因为确定`*p`和`*q`如何比较是我们的责任。

函数`qsort`具有下列原型：

```
void qsort(void *base, size_t nmemb, size_t size,
 int (*compar) (const void *, const void *));
```

`base`必须指向数组中的第一个元素。（如果只是对数组的一段区域进行排序，那么要使`base`指向这段区域的第一个元素。）在一般情况下，`base`就是数组的名字。`nmemb`是要排序元素的数量（不一定是数组中元素的数量）。`size`是每个数组元素的大小，用字节来衡量。`compar`是指向比较函数的指针。当调用函数`qsort`时，它会对数组进行升序排列，并且在任何需要比较数组元素时调用比较函数。

**Q&A**为了对16.3节的数组`inventory`进行排序，可以采用函数`qsort`的下列调用方式：

```
qsort(inventory, num_parts, sizeof(struct part),
 compare_parts);
```

请注意，第二个实际参数是`num_parts`而不是`MAX_PARTS`。我们不希望对整个数组`inventory`进行排序，只是对当前存储的区域进行排序。

编写`compare_parts`函数并不像想象的那么容易，因为函数`qsort`要求它的形式参数类型为`void`型。可惜的是不能通过`void *`型的指针访问到零件编号。我们需要指向结构`part`的指针。为了解决这个问题，将用`compare_parts`把`p`和`q`赋值给`struct part *`型的变量，从而把它们转化成为希望的类型。现在`compare_parts`可以使用新指针访问到`p`和`q`指向的结构成员了。假设希望根据编号对零件排序，下面是函数`compare_parts`可能的形式：

```
int compare_parts(const void *p, const void *q)
{
 struct part *pl = p;
 struct part *ql = q;

 if (pl->number < ql->number)
 return -1;
 else if (pl->number == ql->number)
 return 0;
 else
 return 1;
}
```

虽然可以使用此版本的函数`compare_parts`，但是大多数C程序员愿意编写更加简明的函数。首先，注意到能用强制类型转换表达式替换`pl`和`ql`：

```
int compare_parts(const void *p, const void *q)
```

```

{
 if (((struct part *) p)->number <
 ((struct part *) q)->number)
 return -1;
 else if (((struct part *) p)->number ==
 ((struct part *) q)->number)
 return 0;
 else
 return 1;
}

```

在表达式`((struct part *)p)`周围的括号是必须的。如果没有这些圆括号，那么编译器会试图把`p->number`强制转换成`struct part*`型。

通过移除if语句甚至可以把函数`compare_parts`变得更短：

```

int compare_parts(const void *p, const void *q)
{
 return ((struct part *) p)->number -
 ((struct part *) q)->number;
}

```

如果`p`有较小的零件编号，那么`p`的零件编号减去`q`的零件编号会产生负值。如果两个零件编号相同，则减法结果为零。而如果`p`的零件编号较大，那么减法的结果是正数。

为了用零件的名字代替零件编号对数组`inventory`进行排序，可以使用下列写法的函数`compare_parts`：

```

int compare_parts(const void *p, const void *q)
{
 return strcmp(((struct part*) p)->name,
 ((struct part*) q)->name);
}

```

函数`compare_parts`全部需要做的就是调用函数`strcmp`，此函数会方便地返回负的、零或正的结果。

### 17.7.3 函数指针的其他用途

虽然已经强调函数指针作为实际参数对其他函数而言是无用的，但是并不是都没有好处的。C语言对待指向函数的指针就像对待指向数据的指针一样。我们可以把函数指针存储在变量中，或者用作数组的元素，再或者用作结构或联合的成员，甚至可以编写返回函数指针的函数。

下面例子中的变量就是存储指向函数的指针：

```
void (*pf)(int);
```

`pf`可以指向任何带有`int`型实际参数的函数，而且此函数返回`void`型的值。如果`f`是这样一个函数，那么可以用下列方式把`pf`指向`f`：

```
pf = f;
```

注意，在`f`的前面没有取地址符号`(&)`。因为`pf`现在指向函数`f`，所以既可以用下面这种写法调用`f`：

```
(*pf)(i);
```

也可以用下面这种写法调用：

```
pf(i);
```

元素是函数指针的数组拥有相当广泛的应用。例如，假设正在编写的程序用来显示用户选择格式的命令菜单。可以编写函数实现这些命令，然后把指向这些函数的指针存储在数组中：

```

void (*file_cmd[]) (void) = { new_cmd,
 open_cmd,
 close_cmd,

```

```

 close_all_cmd,
 save_cmd,
 save_as_cmd,
 save_all_cmd,
 print_cmd,
 exit cmd
 };
}

```

如果用户选择命令n，且n是在0到8之间的数，那么通过对数组file\_cmd进行下标操作从而找到所调用的函数：

```
(*file_cmd[n])(); /* or file_cmd[n](); */
```

当然，通过使用switch语句可以获得类似的效果。然而，在数组中存储函数指针可以有更大的灵活性，因为数组元素可以在程序运行时发生改变。

#### 17.7.4 程序：列三角函数表

下列函数用来显示含有cos函数、sin函数和tan函数（这3个函数都在`<math.h>`（>23.3节）中进行了声明）值的表格。程序围绕名为tabulate的函数构建。当给此函数传递函数指针f时，此函数会显示出函数f的值。

##### *tabulate.c*

```

/* Tabulates values of trigonometric functions */

#include <math.h>
#include <stdio.h>

void tabulate(double (*f)(double), double first,
 double last, double incr);

main()
{
 double final, increment, initial;

 printf("Enter initial value: ");
 scanf("%lf", &initial);

 printf("Enter final value: ");
 scanf("%lf", &final);

 printf("Enter increment: ");
 scanf("%lf", &increment);

 printf("\n x cos(x)\n"
 " ----- -----");
 tabulate(cos, initial, final, increment);

 printf("\n x sin(x)\n"
 " ----- -----");
 tabulate(sin, initial, final, increment);

 printf("\n x tan(x)\n"
 " ----- -----");
 tabulate(tan, initial, final, increment);

 return 0;
}

void tabulate(double (*f)(double), double first,
 double last, double incr)
{
 double x;

```

389

```

int i, num_intervals;

num_intervals = ceil((last - first) / incr);
for (i = 0; i <= num_intervals; i++) {
 x = first + i * incr;
 printf("%10.5f %10.5f\n", x, (*f)(x));
}
}

```

函数tabulate使用了函数ceil (>23.3.6节)，此函数属于标准库函数。当给定double型的实际参数x时，函数ceil会返回大于或等于x的最小整数。

下面是使用tabulate.c程序可能的结果：

```

Enter initial value: 0
Enter final value: .5
Enter increment: .1

```

| x       | cos(x)  |
|---------|---------|
| 0.00000 | 1.00000 |
| 0.10000 | 0.99500 |
| 0.20000 | 0.98007 |
| 0.30000 | 0.95534 |
| 0.40000 | 0.92106 |
| 0.50000 | 0.87758 |

| x       | sin(x)  |
|---------|---------|
| 0.00000 | 0.00000 |
| 0.10000 | 0.09983 |
| 0.20000 | 0.19867 |
| 0.30000 | 0.29552 |
| 0.40000 | 0.38942 |
| 0.50000 | 0.47943 |

| x       | tan(x)  |
|---------|---------|
| 0.00000 | 0.00000 |
| 0.10000 | 0.10033 |
| 0.20000 | 0.20271 |
| 0.30000 | 0.30934 |
| 0.40000 | 0.42279 |
| 0.50000 | 0.54630 |

390

## 问与答

问：宏NULL表示什么？(p.251)

答：NULL实际是表示0。当在要求指针的地方使用0时，会要求C语言编译器把它看成是空指针而不是整数0。提供宏NULL只是为了避免混乱。赋值表达式p = 0；既可以是给数值型变量赋值为0，也可以是给指针变量赋值为空指针。而我们无法简单地说明到底是哪一种。相反，赋值表达式p = NULL；却明确地说明p是指针。

**C++** 在C++语言中，更通用的方式是使用0而不是NULL。原因过于技术化了，很难在这里讨论清楚。一些程序员在C程序中也喜欢使用0而不是NULL。

\*问：在伴随编译器的头文件中，NULL按照如下所示进行定义：

```
#define NULL (void *) 0
```

这样把0强制转化为void\*型有什么好处吗？

答：这种技巧在标准C中是合法的。它可以帮助编译器检查到空指针的不正确使用。例如，假设试图把NULL赋值给整型变量：

i = NULL;

如果NULL定义为0，那么这个赋值绝对是合法的。但是，如果把NULL定义为(void \*)0，那么编译器将提示我们把指针赋值给整型变量违反了标准C的规则。

把NULL定义为(void \*)0还有第二点重要的好处。假设调用带有可变长度实际参数列表（>26.1节）的函数，且把传递的NULL作为了其中一个实际参数。如果NULL定义为0，那么编译器将会传递一个不正确的整数零。（在普通函数调用中，因为编译器从函数的原型可以知道它所希望的是指针，所以NULL可以正常工作。然而，当函数具有可变长度实际参数列表时，编译器不会获得这类信息，它会假设0就是表示整数。）如果NULL定义为(void \*)0，那么编译器将会传递空指针。

甚至情况更混乱的是，一些头文件把NULL定义为0L（0的long int型版本）。就像把NULL定义为0一样，这种定义是C语言早期时代的延续，那时的指针和整数彼此兼容。但是，就大多数目的而言，NULL是如何定义的真的不是问题，只是把它想成是空指针的名字就可以了。

问：既然0用来表示空指针，那么我猜想空指针就是字节中各位都为零的地址，对吗？

答：不一定。每个C语言编译器都被允许用不同的方式来表示空指针，而且不是所有编译器都使用零地址的。例如，一些编译器为空指针使用不存在的内存地址。硬件会检查出这种试图通过空指针访问内存的方式。

我们不关心如何在计算机内存储空指针。这是编译器专家关注的细节。重要的是，当在指针环境中使用0时，编译器会把它转换为适当的内部形式。

问：把NULL用作空字符，这是否可以接受？

答：绝对不行。NULL是用来表示空指针的宏，不是空字符。把NULL用作空字符对一些编译器可以适用，但不是全部都可以的（因为一些编译器把NULL定义为(void \*)0）。在任何情况下，把NULL用作非指针的内容都会导致大量的混乱，如果希望给空字符一个名字，可以使用下列定义的宏：

```
#define NUL '\0'
```

问：程序终止时得到这样一条消息“Null pointer assignment”。这是什么意思呢？

答：此消息由一些DOS程序产生，它说明程序使用坏指针（并不一定是空指针）把数据存储到内存中了。可惜的是此消息直到程序终止才显示出来，所以没有线索可以表明是哪条语句导致的错误。消息“Null pointer assignment”可能是因为在scanf函数中丢失&导致的：

```
scanf ("%d", i); /* should have been scanf ("%d", &i); */
```

另一种可能是含有指针的赋值操作对指针未进行初始化或设为空：

```
p = i; / p is uninitialized or null */
```

问：既然在获得消息“Null pointer assignment”时程序好像也可以工作，那么是否可以忽略此消息呢？

答：请重新阅读前一个问题的答案。如果得到消息“Null pointer assignment”，那么程序此时就存在错误。修正错误，或者不修正。虽然程序好像可以工作，但是却无法保证它始终会正确运行。如果使用了未初始化的指针，那么程序有时可以工作，有时则可能失败。而且如果不同的编译器对程序进行了重新编译或者把程序转到其他计算机上，那么程序正常工作的几率几乎为零。

\*问：程序如何知道发生了“Null pointer assignment”？

答：此消息依赖于这样一个事实：数据在小型或中型存储模型中是存储在单个段中，且此段的地址起始为0。编译器会在数据段的开始处留出“空洞”，即初始化为0但是未被程序使用的一小块内存。当程序终止时，它会查看在“空洞”中的任何数据是否是非零的。如果是，那么一定是通过坏指针改变的。

问：强制类型转换malloc或者其他内存分配函数的返回值，是否有什么好处呢？（p.252）

答：虽然一些程序员都这样做，但是是不是真的有什么好处。强制类型转换这些函数返回void\*型的指针并不是标准C必需的，因为void\*型的指针会在赋值操作时自动转换为任何指针类型。对返回值进行强制类型转换的习惯来自于经典C。在经典C中，内存分配函数返回char\*型的值，用强制类型转换实现是必要的。

391

392

问：函数calloc把内存块初始为“零位”，这是否意味着内存块中的全部数据项都变为零了？(p.255)

答：通常是，但不总是这样的。把整数设置成零位会始终使整数为零。把浮点数设置成零位通常会使数为零，但这是不能保证的，要依赖于浮点数的存储方式。此问题类似于指针的情况，指针各位置为零并不一定是空指针。

\*问：我已经知道了结构标记机制是如何允许结构本身包含指针的。但是，如果两个结构都含有指向对方的成员，会怎么样呢？(p.258)

答：下面是处理这种情况的方法：

```
struct s1; /* incomplete declaration of s1 */
struct s2 {
 ...
 struct s1; *p;
};
struct s1 {
 ...
 struct s2; *q;
};
```

393

因为没有指明结构s1的成员，所以第一个s1结构的声明是“不完整的”。C语言允许不完整的结构声明，因为完整的声明稍后会出现在相同的作用域内。

问：为什么不把函数qsort简单命名为sort呢？(p.270)

答：函数qsort的名字来源于1962年C.A.R. Hoare出版的快速排序算法。反过来说，即使一些qsort函数的版本采用了快速排序算法，C标准也不要求函数qsort使用快速排序算法。

问：就像下例所示那样，把函数qsort的第一个实际参数强制转换为void\*类型，不是必要的吧？(p.270)

```
qsort((void *) inventory, num_parts, sizeof(struct part),
 compare_parts);
```

答：不是必要的。任何类型的指针都可以自动转换为void \*类型的。

\*问：我打算使用函数qsort对整数数组进行排序，但是在编写比较函数时遇到了问题。编写的秘诀是什么？

答：下面是可以使用的版本：

```
int compare_ints(const void *p, const void *q)
{
 return*(int *)p-*(int *)q;
}
```

很奇怪吗？表达式(int \*)p把p强制转换为int\*类型，所以\*(int \*)p将是p所指向的整数。

\*问：我需要对字符串数组进行排序，所以计划只使用函数strcmp作为比较函数。然而，当把它传递给函数qsort时，编译器发出了出错警告。我试图通过把函数strcmp嵌入到比较函数中的方法来解决问题：

```
int compare_strings(const void *p, const void *q)
{
 return strcmp(p,q);
}
```

现在对程序进行编译，但是函数qsort好像没有对数组进行排序。我做错什么了吗？

答：首先，不能把strcmp本身传递给函数qsort，因为qsort函数要求比较函数带有两个const void \*型的形式参数。由于没有把p和q正确地假设为字符串(char \*型指针)，所以函数compare\_strings无法工作。事实上，p和q指向的数组元素含有char\*\*型指针。为了修改函数compare\_strings，将把p和q强制转换为char \*\*型的，然后用\*运算符来移走间接寻址的一层操作：

```
int compare_strings(const void *p, const void *q)
{
 return strcmp(*(char **)p, *(char **)q);
}
```

394

## 练习

### 17.2节

- 每次调用时都检查函数malloc的返回值是一件很烦人的事情。请编写一个名为my\_malloc的函数，用来自作为malloc函数的“包装器”。当调用函数my\_malloc并且要求分配n个字节时，它会转到调用malloc函数，判断malloc函数确实没有返回空指针，然后返回来自malloc的指针。如果malloc返回空指针，那么函数my\_malloc显示出错信息并且终止程序。
- 请编写名为strdup的函数，此函数使用动态存储分配来产生字符串的副本。例如，调用  

```
p = strdup(str);
```

 将为和str长度相同的字符串分配内存空间，并且把字符串str的内容复制给新字符串，然后返回指向新字符串的指针。如果分配内存失败，那么函数strdup返回空指针。
- 请编写一个程序把用户录入的一系列单词进行排序，并且显示删除的重复部分。提示：采用指针数组，且每个指针都指向动态分配的字符串。额外加分：使用qsort函数（17.7节）进行排序操作。

### 17.3节

- 请修改程序invent.c（16.3节），使其可以对数组inventory进行动态内存分配，并且稍后在填满时再次进行内存分配。初始使用malloc为拥有10个part结构的数组分配足够的内存空间。当数组没有足够的空间给新的零件时，使用realloc函数来使内存数量加倍。在每次数组变满时重复加倍操作步骤。

### 17.5节

- 假设下列声明有效：

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right };
struct rectangle *p;
```

假设希望p指向结构rectangle，其中此结构的左上角位于(0, 1)的位置上，而右下角位于(1, 0)的位置上。请编写一系列语句用来分配这样一个结构，并且像说明的那样进行初始化。

- 假设f和p的声明如下所示：

```
struct {
 union {
 char a,b;
 int c;
 }d;
 int e[5];
} f, *p = &f;
```

那么下列哪些语句是合法的？

- (a) p->b = ' ';
- (b) p->e[3] = 10;
- (c) (\*p).d.a = '\*' ;
- (d) p->d->c = 20;

- 请修改函数delete\_from\_list使它只使用一个指针变量而不是两个（即cur和prev）。
- 假设下列循环删除了链表中的全部结点，并且释放了占用的内存。但是，此循环有错误。请解释错误是什么并且说明如何修正错误。

```
for (p = first; p != NULL; p = p->next)
 free(p);
```

- 请修改程序invent2.c，方法是增加允许用户把零件从数据库中删除的e（擦除）命令。
- 15.2节描述的文件stack.c提供了在栈中进行整数排序的函数。在那一节中，栈是用数组实现的。请修改程序stack.c从而使栈现在可以作为链表来存储。使用单独一个指向链表首结点的指针变量（栈“顶”）来替换变量contents和变量top。在stack.c中编写的函数要使用此指针。删除函数

`is_full`, 用返回TRUE(如果创建的结点可以获得内存)或FALSE(如果创建的结点无法获得内存)的函数`push`来代替。

### 17.6节

11. 请修改函数`delete_from_list`(17.5节), 使函数的第一个实际参数是`struct node **`类型(即指向链表首结点的指针), 并且返回类型是`void`。在删除了期望的结点后, 函数`delete_from_list`必须修改第一个实际参数, 使其指向该链表。

### 17.7节

12. 请说明下列程序的输出结果, 并且说明理由。

```
#include <stdio.h>

int f1(int (*f)(int));
int f2(int i);

main()
{
 printf("Answer: %d\n", f1(f2));
 return 0;
}

int f1(int (*f) (int))
{
 int n = 0;
 while ((*f)(n)) n++;
 return n;
}

int f2(int i)
{
 return i * i + i - 12;
}
```

396

13. 请编写下列函数。函数`sum(g, i, j)`的调用应该返回`g(i) + ... + g(j)`。

```
int sum(int (*f)(int), int start, int end);
```

14. 设`a`是有100个整数的数组。请编写函数`qsort`的调用, 此调用只对数组`a`中的后50个元素进行排序。  
(不需要编写比较函数。)

15. 请修改函数`compare_parts`使零件根据编号进行降序排列。

16. 请修改程序`invent.c`(16.3节), 使`p`(显示)命令显示零件之前调用函数`qsort`对数组`inventory`进行排序。

17. 请编写一个函数, 要求在给定字符串作为实际参数时, 此函数搜索下列所示的结构数组寻找匹配的命令名, 然后调用和匹配名称相关的函数:

```
struct {
 char *cmd_name;
 void (*cmd_pointer)(void);
} file_cmd[] =
{ {"new", new_cmd},
 {"open", open_cmd},
 {"close", close_cmd},
 {"close all", close_all_cmd},
 {"save", save_cmd},
 {"save as", save_as_cmd},
 {"save all", save_all_cmd},
 {"print", print_cmd},
 {"exit", exit_cmd}
};
```

397

# 第18章

# 声明

让一些事情可变很容易，而掌控不变的期限则需要技巧。

声明在C语言编程中起到核心的作用。通过声明变量和函数，可以在检查程序潜在的错误以及把程序翻译成目标代码两方面为编译器提供至关重要的信息。

前几章已经提供了声明的示例，但是没有完整的描述，本章将会弥补这个缺憾。本章会探讨可以用于声明的复杂选项，并且显示变量声明和函数声明之间的几个共同点。此外，本章还为存储期限、作用域以及链接这些重要概念提供了坚实的基础。

18.1节介绍大多数声明格式的语法，这是之前我们一直回避的主题。然后，将集中讨论声明中出现的数据项：存储类型（18.2节）、类型限定符（18.3节）、声明符（18.4节）和初始化式（18.5节）。

了解声明需要一些时间，但它是需要掌握的至关重要的技能。本章可能不是全书中最重要的部分，但是在考虑成为C程序员之前需要熟练掌握它。

## 18.1 声明的语法

声明为编译器提供有关标识符含义的信息。当编写

int i;

时，是在告诉编译器：名字*i*表示当前作用域内数据类型为int的变量。声明

float f(float);

则是在告诉编译器：f是一个返回值为float型的函数，并且此函数有一个实际参数，此参数类型也为float型。

在大多数通用格式中，声明具有下列格式：

[声明的格式]

声明说明符 声明符

声明说明符（declaration specifier）描述声明的数据项的性质。声明符（declarator）给出了数据项的名字，并且可以提供关于数据项性质的额外信息。

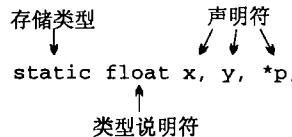
声明说明符分为以下3大类：

- **存储类型。** 存储类型一共有4种：auto、static、extern和register。在声明中最多可以出现一种存储类型。如果表示存储类型，则必须把它放置在声明中的首要位置。
- **类型限定符。** 只有两种类型限定符：const和volatile。声明可以指明一个限定符、两个都有或者一个也没有。
- **类型说明符。** 关键字void、char、short、int、long、float、double、signed和unsigned全部都是类型说明符。第7章对这些单词组合进行了描述。这些单词出现的顺序不是问题（int unsigned long和long unsigned int完全一样）。类型说明符也包括结构、联合和枚举的说明（例如，struct point{int x, y;}、struct {int x, y}或者struct point）。用typedef创建的类型名也是类型说明符。

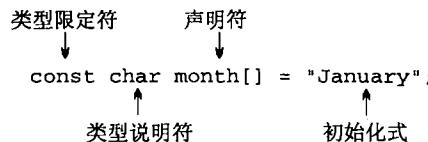
类型限定符和类型说明符必须跟随在存储类型的后边，但是两者的顺序没有严格的限制。由于书写风格，这里会将类型限定符放置在类型说明符的前面。

声明符包括标识符（简单变量的名字）、后边跟随[]的标识符（数组名）、前放置\*的标识符（指针名）和后边跟随()的标识符（函数名）。声明符之间用逗号分割。表示变量的声明符后边可以跟随初始化式。

一起看些说明这些规则的例子。下面是一个带有存储类型和三个声明符的声明：

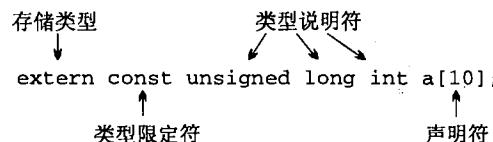


下列声明有类型限定符但是没有存储类型。此外，它还有初始化式：

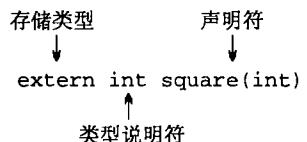


400

下列声明既有存储类型也有类型限定符。此外，它还有三种类型说明符，当然它们的顺序并不重要：



和变量声明一样，函数声明也有存储类型、类型限定符和类型说明符。下列声明具有存储类型和类型说明符：



本章余下部分将详细介绍存储类型、类型限定符、声明符和初始化式。

## 18.2 存储类型

存储类型可以用于变量、较小范围的函数和形式参数的说明。现在将集中讨论变量的存储类型。

对于本节的余下部分，术语块（block）既表示函数体（大括号闭合的部分）也表示块语句（包含声明的复合语句）。

### 18.2.1 变量的特性

C程序中的每个变量都具有3个性质：

- **存储期限。** 变量的存储期限决定了为变量预留和释放内存的时间。具有自动存储期限的变量在所属块被执行时获得内存单元，并在块终止时释放内存单元，从而会导致变量失去值。具有静态存储期限的变量在程序运行期间占有同样的内存单元，也就是可以允许变量无限期地保留它的值。

401

- **作用域。** 变量的作用域是指引用变量的那部分程序文本。变量可以有块作用域（变量从声明的地方一直到闭合块的末尾都是可见的）或者文件作用域（变量从声明的地方一直到闭合文件的末尾都是可见的）。
- **链接。** Q&A 变量的链接确定了程序的不同部分可以共享此变量的范围。具有外部链接的变量可以被程序中的几个（或许全部）文件共享。具有内部链接的变量只能属于单独一个文件，但是此文件中的函数可以共享这个变量。（如果具有相同名字的变量出现在另一个文件中，那么系统会把它作为不同的变量来处理。）无链接的变量属于单独一个函数，而且根本不能被共享。

变量的默认存储期限、作用域和链接都依赖于变量声明的位置：

- 在块内部（包括函数体）声明的变量具有自动存储期限、块作用域，并且无链接。
- 在程序的最外层，任意块外部声明的变量具有静态存储期限、文件作用域和外部链接。

下面的例子说明了变量*i*和变量*j*的默认性质：

```
int i; // 静态存储期限
 // 文件作用域
 // 外部链接

void f(void)
{
 int j; // 自动存储期限
 // 块作用域
 // 无链接
}
```

对许多变量而言，默认的存储期限、作用域和链接是可以符合要求的。当这些性质无法满足要求时，可以改变通过指定明确的存储类型来改变变量的性质：`auto`、`static`、`extern`和`register`。

## 18.2.2 auto 存储类型

**402** `auto`存储类型只对属于块的变量有效。`auto`类型的变量具有自动存储期限（无需惊讶）、块的作用域，并且无链接。`auto`存储类型几乎从来不用明确地指明，因为对于在块内部声明的变量，它是默认的。

## 18.2.3 static 存储类型

`static`存储类型可以用于全部变量，而无需考虑变量声明所在的位置。但是，块外部声明的变量和块内部声明的变量会有不同的效果。当用在块外部时，单词`static`说明变量具有内部链接。当用在块内部时，`static`把变量的存储期限从自动的变成了静态的。下面的图说明把变量*i*和变量*j*声明为`static`所产生的效果：

```
static int i; // 静态存储期限
 // 文件作用域
 // 内部链接

void f(void)
{
 static int j; // 静态存储期限
 // 块作用域
 // 无链接
}
```

在用于外部块声明时，`static`本质上隐藏了它所在声明文件内的变量。只有出现在同一文件中的函数可以看到此变量。在下面的例子中，函数*f1*和函数*f2*都可以访问到变量*i*，但是其他文件中的函数无法做到：

```
static int i;

void f1(void)
```

```
{
 /* has access to i */
}

void f2(void)
{
 /* has access to i */
}
```

`static`的此种用法可以用来实现一种称为信息隐藏（>19.2节）的技术。

块内声明的`static`型变量在程序执行期间驻留在同一存储单元内。和每次程序离开闭合块就会丢失值的自动变量不同，`static`型变量会无限期的保留值。`static`型变量具有一些有趣的性质：

- 块内的`static`型变量只在程序执行前进行一次初始化，而`auto`型变量则会在每次变成有效时进行初始化（当然，需假设它有初始化式）。
- 每次函数进行递归调用时，它都会获得一组新的`auto`型变量的集合。另一方面，如果函数含有`static`型变量，那么此递归函数的全部调用都可以共享这个`static`型变量。
- 虽然函数不应该返回指向`auto`型变量的指针，但是函数返回指向`static`型变量的指针是没有错误的。

403

声明函数中的一个变量为`static`存储类型，这样做允许函数在“隐藏”区域内的调用之间保留信息。隐藏区域是程序其他部分无法访问到的地方。然而，更经常的做法是用`static`来使程序更加有效。思考下列函数：

```
char digit_to_hex_char(int digit)
{
 const char hex_chars[16] = "0123456789ABCDEF";
 return hex_chars[digit];
}
```

每次调用`digit_to_hex_char`函数时，都会把字符0123456789ABCDEF复制给数组`hex_chars`来对其进行初始化。现在，把数组设为`static`类型的：

```
char digit_to_hex_char(int digit)
{
 static const char hex_chars[16] = "0123456789ABCDEF";
 return hex_chars[digit];
}
```

既然`static`型变量只进行一次初始化，那么这样做就改进了`digit_to_hex_char`函数的速度。

#### 18.2.4 `extern` 存储类型

`extern`存储类型使几个源文件可以共享同一个变量。15.2节介绍了使用`extern`的基本概念，所以这里的讨论不会太多。回顾过去的内容可以知道，下列声明给编译器提供的信息是，`i`是`int`型变量：

```
extern int i;
```

但是这样不会导致编译器为变量`i`分配存储单元。在C语言的术语中，上述声明不是变量`i`的定义，它只是提示编译器需要访问定义在别处的变量（可能稍后在同一文件中，或者更经常是在另一个文件中）。变量在程序中可以有多次声明，但只能有一次定义。

对规则而言，变量的`extern`声明不是定义是一个例外。初始化变量的`extern`声明可以用作变量的定义。例如，声明

```
extern int i = 0;
```

等效于声明

```
int i = 0;
```

404 这条规则防止用不同方法对初始化变量进行多次extern声明。

extern声明中的变量始终具有静态存储期限。变量的作用域依赖于声明的位置。**Q&A**如果声明在块内部，那么变量具有块作用域；否则，变量具有文件作用域：

```
extern int i; // 静态存储期限
// 文件作用域
// 什么链接?

void f(void)
{
 extern int j; // 静态存储期限
 // 块作用域
 // 什么链接?
}
```

确定extern型变量的链接有一定难度。如果变量在文件中较早的位置（任何函数定义的外部）声明为static，那么它具有内部链接；否则（通常情况下），变量具有外部连接。

### 18.2.5 register存储类型

声明变量具有register存储类型就要求编译器把变量存储在寄存器中，而不是像其他变量一样保留在内存中。（寄存器是驻留在计算机CPU中的存储单元。在传统计算机架构中，存储在寄存器中的数据会比存储在普通内存中的数据访问和更新的速度更快。）指明变量的存储类型是register是一种要求，而不是命令。如果选择，编译器可以自由的把register型变量存储在内存中。

register存储类型只对声明在块内的变量有效。register型变量具有和auto型变量一样的存储期限、作用域和链接。但是，register型变量缺乏auto型变量所具有的一种性质：由于寄存器没有地址，所以对register型变量使用取地址运算符&是非法的。这种限制甚至使得编译器选择把变量存储在内存中。

register存储类型最好用于需要频繁进行访问和/或更新的变量。例如，在for语句中的循环控制变量用register类型就是一个很好的选择：

```
int sum_array(int a[], int n)
{
 register int i;
 int sum = 0;
 for (i = 0; i < n; i++)
 sum += a[i];
 return sum;
}
```

405

现在register存储类型已经不像以前那样在C程序员中流行了。今天的编译器比早期的C语言编译器更加复杂了。一些编译器可以自动决定变量保留在寄存器中是否可以获得最大的好处。

### 18.2.6 函数的存储类型

和变量的声明一样，函数的声明（和定义）可以包含存储类型，但是选项只有extern和static。在函数声明开始处的单词extern说明函数具有外部链接，也就是允许其他文件调用此函数。static说明内部链接，也就是说只能在定义函数的文件内部调用此函数。如果不指明函数的存储类型，那么会假设函数具有外部连接。

思考下面的函数声明：

```
extern int f(int i);
static int g(int i);
int h(int i);
```

函数f具有外部链接，函数g具有内部链接，而函数h（默认情况下）具有外部链接。

声明函数是extern型的就如同声明变量是auto型一样，两者都没有使用的目的。基于这个原因，本书不在函数声明中使用extern。然而，需要意识到一些程序员广泛地使用extern也是无害的。

另一方面，声明函数是static型的确是十分的有用。事实上，当声明不打算被其他文件调用的任意函数时，建议使用static存储类型。这样做的好处包括有以下两点：

- 更容易维护。把函数f声明为static存储类型保证在函数定义出现的文件之外函数f都是不可见的。这样的结果是，某些稍后修改程序的人可以知道对函数f的变化不会影响其他文件中的函数。（一个例外是：另一个文件中的函数如果传递了指向函数f的指针可能会受到函数f变化的影响。幸运的是，这种问题很容易通过检查函数f定义的文件来发现，因为传递f的函数一定也定义在此文件中。）
- 减少了“名字空间污染”。由于声明为static的函数具有内部链接，所以可以在其他文件中重新使用这些函数的名字。虽然可能不会为一些其他目的故意重新使用函数名字，但是在大规模程序中是很难避免这种现象的。带有外部链接的大量函数名可能产生的结果就是C程序员所谓的“名字空间污染”，即在不同文件中的名字意外地发生互相冲突。使用static存储类型可以有效地预防此类问题。

函数的形式参数具有和auto型变量相同的性质：自动存储期限、块作用域和无链接。唯一能用于说明形式参数存储类型的就是register。

406

### 18.2.7 小结

已经介绍了各种存储类型，现在对已知内容进行一个总结。下面的代码段说明了变量和形式参数声明中包含或者忽略存储类型的所有可能的方法。

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
 auto int g;
 int h;
 static int i;
 extern int j;
 register int k;
}
```

表18-1说明了上述例子中每个变量和形式参数的性质。

表18-1 变量和形式参数的性质

| 名 字 | 存 储 期 限 | 作 用 域 | 链 接 |
|-----|---------|-------|-----|
| a   | 静 态     | 文 件   | 外 部 |
| b   | 静 态     | 文 件   | ①   |
| c   | 静 态     | 文 件   | 内 部 |
| d   | 自 动     | 块     | 无   |
| e   | 自 动     | 块     | 无   |
| g   | 自 动     | 块     | 无   |
| h   | 自 动     | 块     | 无   |
| i   | 静 态     | 块     | 无   |
| j   | 静 态     | 块     | ①   |
| k   | 自 动     | 块     | 无   |

① 由于这里没有显示出变量b和j的定义，所以不可能确定这些变量的链接。在大多数情况下，变量会定义在另一个文件中，并且具有外部链接。

在4种存储类型之中，最重要的是extern和static。auto类型没有任何效果，且现代编译器已经使register类型变得废弃无用了。

### 18.3 类型限定符

C语言中一共有两种类型限定符：const和volatile。因为volatile只用在底层编程中，所以本书将此限定符的讨论推迟到20.3节再介绍。const用来声明一些类似于变量的对象，但这些变量是“只读”的。程序可以访问const型对象的值，但是无法改变它的值。例如，下面这个声明产生了名为n的const型对象，且此对象的值为10：

```
const int n = 10;
```

**407** 而下列声明产生了名为days\_per\_month的const型数组。

```
const int days_per_month[] =
{31, 28, 31, 30, 31, 30, 31, 31, 30, 31};
```

用const来说明对象的值不会改变具有几个好处：

- const是文档格式。声明对象是const类型就是提示任何稍候阅读程序的人，对象的值不会改变。
- 编译器可以检查到程序不会特意地试图改变对象的值。
- 当为某种可能的应用类型编写程序时（特别是嵌入式系统），编译器可以用单词const来确定数据存储到ROM（只读内存）中。

乍一看，好像const违反了前几章中用过的产生常量名的#define指令。然而，实际上#define和const之间有明显的差异：

- 可以用#define指令产生数字常量、字符常量或字符串常量的名字。const可用于产生任何类型的只读对象，这包括常量数组、常量指针、常量结构和常量联合。
- const类型的对象遵循和变量相同的作用域规则，而用#define产生的常量不遵守这些规则。特别是，不能用#define产生具有块作用域的常量。
- 和宏的值不同，可以在调试器中看到const型对象的值。
- 不同于宏**Q&A**，不可以把const型对象用于常量表达式。例如，由于数组边界必须是常量表达式，所以不能写成下列形式：

```
const int n = 10;
int a[n]; /*** WRONG ***/
```

没有绝对的原则说明何时使用#define和何时使用const。这里建议对表示数字或字符的常量使用#define。另外，还能使用常量作为数组维数，并且在switch语句或其他要求常量表达式的地方使用常量。我们使用const主要是为了保护存储在数组中的常量数据。

### 18.4 声明符

**408** 声明符是由标识符（声明的变量或函数的名字）以及可能在前边的符号\*或者跟随在后边的[]或()共同组成的。通过把\*、[]和()组合在一起，可以创建复杂声明符。

在认识较为复杂的声明符之前，先来复习一下前面了解的声明符的知识。最简单的情况，声明符就是标识符，就如同下面例子中的i：

```
int i;
```

声明符还可以包含符号\*、[]和()：

- 用\*开头的声明符表示指针：

```
int *p;
```

- 用[]结尾的声明符表示数组：

```
int a[10];
```

如果数组是形式参数，或者数组有初始化式，再或者数组的存储类型为extern，那么方括号内可以为空：

```
extern int a[];
```

因为a是在程序中的某处定义的，所以这里编译器不需要知道数组的长度。（在多维数组中，只有第一维方括号可以为空。）

- 用()结尾的声明符表示函数：

```
int abs(int i);
void swap(int *a, int *b);
int find_largest(int a[], int n);
```

C语言允许在函数声明中忽略形式参数的名字：

```
int abs();
void swap();
int find_largest();
```

甚至括号内可以为空：

```
int abs();
void swap();
int find_largest();
```

这些声明指明了abs、swap和find\_largest的返回类型，但是没有提供有关它们实际参数的信息。括号内置为空不等同于把单词void放置在圆括号内，后者说明没有实际参数。来自于经典C的这种函数声明的空括号形式正在迅速消失。这种格式比标准C的原型形式差，因为空括号形式不允许编译器检查函数调用是否有正确的实际参数。

如果全部的声明符都和前面讲的这些一样简单，那么C语言的编程将一蹴而就。可惜的是，实际程序中的声明符往往组合了符号\*、[]和()。我们已经见过这类组合的示例了。我们知道下列语句声明了一个数组，此数组的元素是10个指向整数的指针：

```
int *ap[10];
```

我们还知道下列语句声明了一个函数，此函数有float型的实际参数，并且返回指向float型值的指针。

```
float *fp(float);
```

此外，17.7节学过这样一条语句，用来声明一个指向函数的指针，此函数有int型实际参数且返回void型的值：

```
void (*pf)(int);
```

### 18.4.1 解释复杂声明

到目前为止，在声明符的理解方面还没有遇到太多的麻烦。但是，下面这个声明符是什么呢？

```
int *(*x[10])(void);
```

这个声明符组合了\*、[]和()，所以x是指针、数组还是函数并不明显。

幸运的是，无论多么费解，有两条简单的规则可以用来理解任何声明：

- 始终从内往外读声明符。换句话说，定位用来声明的标识符，并且从此处的声明开始解释。
- 在作选择时，始终先是[]和()后是\*。如果\*在标识符的前面，而标识符后边跟着[]，那么标识符表示数组而不是指针。同样地，如果\*在标识符的前面，而标识符后边跟着

409

()，那么标识符表示函数而不是指针。(当然，可以一直使用圆括号来使超过\*的[]和()优先级无效。)

首先把这些规则应用于下面这些简单的示例。在下列声明中，

```
int *ap[10];
```

ap是标识符。由于\*在ap的前面，且后边跟着[]，而[]优先级高，所以ap是指针数组。在下列声明中，

```
float *fp(float);
```

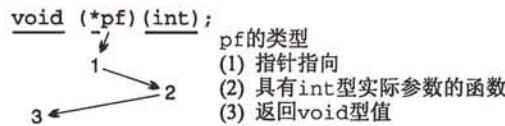
fp是标识符。由于\*在标识符的前面，且后边跟着()，而()优先级高，所以fp是返回指针的函数。

下列声明是一个小陷阱：

```
void (*pf)(int);
```

由于\*pf在闭合的括号内，所以pf应该是指针。但是(\*pf)后边跟着(int)，所以pf必须指向函数，且此函数带有int型的实际参数。单词void表明了此函数的返回类型。  
410

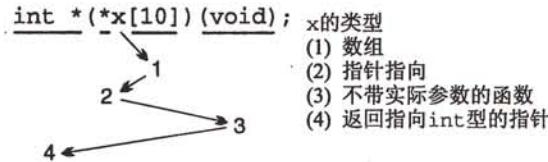
正如最后的示例说明的那样，对复杂声明符的理解经常会遇到从标识符的一边到另一边的曲折：



下面用这种曲折方法来解释早前给出的声明：

```
int *(*x[10])(void);
```

首先，定位的标识符是声明的(x)。在x前有\*，而后边又跟着[]。因为[]优先级高于\*，所以取右侧(x是数组)。接下来，从左侧找到数组中元素的类型(指针)。再接下来，到右侧找到指针所指向的数据类型(不带实际参数的函数)。最后，回到左侧看每个函数返回的内容(指向int型的指针)。图示过程如下所示：



要想熟练掌握C语言的声明需要花些时间并且要多练习。唯一的好消息是在C语言中有不能声明的特定内容。函数不能返回数组：

```
int f(int)[]; /**** WRONG ****/
```

函数不能返回函数：

```
int g(int)(int); /**** WRONG ****/
```

函数型的数组也是不可能的：

```
int a[10](int); /**** WRONG ****/
```

## 18.4.2 使用类型定义来简化声明

一些程序员利用类型定义来简化复杂的声明。考虑一下前面检查过的x的声明：

```
int *(*x[10])(void);
```

为了使x的类型更容易理解，可以使用下列一系列的类型定义：

```
typedef int *Fcn(void);
typedef Fcn *Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```

411

反向阅读，发现x具有Fcн\_ptr\_array类型，Fcн\_ptr\_array是有Fcн\_ptr值的数组，Fcн\_ptr是指向Fcн类型的指针，而且Fcн是不带实际参数的函数，且此函数返回指向int型值的指针。

## 18.5 初始话式

为了方便，C语言允许在声明变量时为它们指定初始值。为了初始化变量，可以在声明符的后边书写符号=，然后在其后再跟上初始化式。（不要把声明中的符号=和赋值运算符相混淆；初始化和赋值不一样。）

在前几章中已经见过各种各样的初始化式了。简单变量的初始化式就是一个与变量类型一样的表达式：

```
int i = 5 / 2; /* i is initially 2 */
```

如果类型不匹配，C语言会用和赋值运算相同的规则对初始化式进行类型转换（>7.5节）：

```
int j = 5.5; /* converted to 5 */
```

针对指针变量的初始化式，必须是具有和变量相同类型或void\*类型的指针表达式：

```
int *p = &i;
```

数组、结构或联合的初始化式通常是一串封闭在大括号内的值：

```
int a[5] = {1, 2, 3, 4, 5};
```

为了全面覆盖声明的范围，现在来看看一些控制初始化式的额外规则：

- 具有静态存储期限的变量的初始化式必须是常量：

```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

412

由于LAST和FIRST都是宏，所以编译器可以计算出i（ $100-1+1=100$ ）的初始值。如果LAST和FIRST是变量，那么初始化式就是非法的。

- 如果变量具有自动存储期限，那么它的初始化式不需要是常量：

```
int f(int n)
{
 int last = n - 1;
 ...
}
```

- 用大括号闭合的数组、结构或联合的初始化式必须只能包含常量表达式，不允许有变量或函数调用：

```
#define N 2
int powers[5] = {1, N, N*N, N*N*N, N*N*N*N};
```

因为N是常量，所以powers的初始化式是合法的。如果N是变量，那么程序将无法进行编译。

- 针对自动类型的结构或联合，它们的初始化式可以是另外一个结构或联合：

```
void g(struct complex c1)
{
```

```
struct complex c2 = c1;
...
}
```

虽然初始化式需要是具有适当类型的表达式，但是它们不需要一定是变量或形式参数名。例如，c2的初始化式可以是\*p，这里的p具有`struct complex*`类型，或`f(c1)`类型，f是返回`complex`结构类型的函数。

## 未初始化的变量

在前面几章中已经暗示了，未初始化变量有未定义的值。但并不总是这样的，变量的初始化值依赖于变量的存储期限：

- 具有自动存储期限的变量没有默认的初始值。不能预测自动变量的初始值，而且每次变量变为有效时可以对值进行改变。
- 具有静态存储期限的变量默认情况下的值为零。用`calloc`分配的内存是简单的给字节的位置零，而静态变量不同于此，它是基于类型的正确初始化：即整型变量初始化为0，浮点变量初始化为0.0，而指针则初始化为空指针。

作为一种书写风格，最好为静态类型的变量提供初始化式，而不是依赖事实上保证的零。

如果程序访问到没有明确初始化的变量，那么稍后阅读程序的人可能不容易确定出是否变量设为零，或者很难确定出变量是否在程序中的某处进行了赋值初始化。

413

## 问与答

问：“作用域”和“链接”之间的差异到底是什么？(p.280)

答：作用域得益于编译器，而链接得益于链接器。编译器用标识符的作用域来确定在文件定义处提到的标识符是否是合法的。当编译器把源文件翻译成目标代码时，它会注意到有外部链接的名字，并最终把这些名字存储到目标文件内的表中。因此，链接器可以访问到具有外部链接的名字，而内部链接的名字或无链接的名字对链接器而言是不可见的。

问：我无法理解具有块作用域但是外部链接的名字。可否详细解释一下？(p.282)

答：当然可以。假设某个源文件定义了变量i：

```
int i;
```

现在假设变量i的定义放在了任意函数之外，所以默认情况下它具有外部链接。在另一个文件中，有一个函数f需要访问变量i，所以f的函数体把i声明为`extern`类型：

```
void f(void)
{
 extern int i;
 ...
}
```

在第一个文件中，变量i具有文件作用域。但是，在函数f内，i是块作用域。如果除函数f以外的其他函数需要访问变量i，那么它们将需要单独进行声明。（或者简单地把变量i的声明移到函数f外，从而使其变成文件作用域。）在整个事情中会混淆的就是每次声明或定义i会建立不同的作用域。有时建立的是文件作用域，有时建立的是块作用域。

\*问：为什么不能把`const`型的对象用于常量表达式呢？`constant`不就是常量吗？(p.284)

答：不一定。`const`型对象只是保证在它的生命期内保留常量，而不是在程序的整个执行期内。假设是在函数体内声明的`const`型对象：

```
void f(int n)
{
```

```
const int m = n;
}
```

当调用函数f时，m将会被初始化为函数f的实际参数的值。然后m将在f返回之前保留常量。当再次调用函数f时，m可能会得到不同的值。这就是出现问题的地方。假设用m来制定数组的长度：

```
void f(int n)
{
 const int m = n;
 int a[m]; /*** WRONG ***/
}
```

那么直到函数f调用之前数组a的长度都是未知的，这显然违反了C语言的规则。C语言规定对编译器而言每个数组的长度都必须是已知的。

但是这还不是const唯一的问题。在块外部声明的const型对象具有外部链接，并且可以在文件之间对其进行共享。如果C语言允许在常量表达式中使用const型对象，那么很容易会自行发现下列情况：

```
extern const int n;
int a[n]; /*** WRONG ***/

```

可能在其他文件中对n进行了定义，这使编译器无法确定数组a的长度。

**C++** C语言在限制const型对象方面没有问题是令人苦恼的。C++语言通过允许const型对象出现在常量表达式中改进了这种现象，因为C++允许：(1) 它是整数，(2) 它的初始化式是常量：

```
const int n = 10;
int a[n]; /* legal in C++, but not in C */

```

默认情况下，C++语言还指定const型对象具有内部链接，这样就使此类型对象的定义可以放置到头文件中。

问：为什么声明符的语法如此古怪？

答：声明试图进行模拟使用。指针声明符的格式为\*p，这种格式和稍后将用于p的间接寻址运算符方式相匹配。数组声明符的格式为a [...]，这种格式和数组稍后的下标方式相匹配。函数声明符的格式为f(...)，这种格式和函数调用的语法相匹配。这种原因甚至可以扩展到最复杂的声明符上。请思考一下17.7节中的数组file\_cmd，此数组的元素都是指向函数的指针。数组file\_cmd的声明符格式为

```
(*file_cmd[]) (void)
```

而且调用此种函数的格式为

```
(*file_cmd[n])();
```

其中圆括号、方括号和\*都在同样的位置上。

## 练习

### 18.1节

- 请指出下列声明的存储类型、类型限定符、类型说明符、声明符和初始化式。

- (a) static char \*\*lookup(int level);
- (b) volatile unsigned long io\_flags;
- (c) extern char \*file name[MAX FILES], path[];
- (d) static const char token\_buf[] : "";

### 18.2节

- 用auto、extern、register和或static来回答下列问题。

- (a) 哪种存储类型可以用于说明能被几个文件共享的变量或函数？
- (b) 假设变量x可以被一个文件中的几个函数共享，但是对其他文件中的函数却是隐藏的。那么变量x

应该被声明为哪种存储类型呢？

(c) 哪些存储类型会影响变量的存储期限？

3. 请列出下列文件中每个变量和形式参数的存储期限、作用域和链接：

```
extern float a;
void f(register double b)
{
 static int c;
 auto char d;
}
```

4. 假设f是下列函数。如果在此之前f从来没有被调用过，那么f(10)的值是多少呢？如果在此之前f已经被调用过5次了，那么f(10)的值又是多少呢？

```
int f(int i)
{
 static int j = 0;
 return i * j++;
}
```

### 18.3节

5. 假设声明x为const型对象，那么下列关于x的语句哪条是假的呢？

- (a) 如果x的类型是int型，那么可以用它来声明数组的长度。
- (b) 编译器将查到没有对x进行赋值。
- (c) x和变量遵循同样的作用域规则。
- (d) x可以是任意类型。

### 18.4节

6. 请编写下列每个声明指定的x类型的完整描述。

- |                                 |                                          |
|---------------------------------|------------------------------------------|
| (a) char (*x[10])(int);         | (b) int (*x(int))[5];                    |
| (c) float *(*x(void))(int)[10]; | (d) void (*x(int, void (*y)(int)))(int); |

416

7. 请利用一系列的类型定义来简化练习6中的每个声明。

8. 请为下列变量和函数编写声明：

- (a) p是指向函数的指针，并且此函数带有字符型指针作为实际参数，函数返回的也是字符型指针。
- (b) f是带有两个实际参数的函数：一个参数是指向结构的指针p，且此结构标记为t；另一参数是长整数n。f返回指向函数的指针，且指向的函数没有实际参数也无返回值。
- (c) a是含有4个元素的数组，且每个元素都是指向函数的指针，而这些函数都是没有实际参数且无返回值的。a的元素初始指向的函数名分别是insert、serach、update和print。
- (d) b是含有10个元素的数组，且每个元素都是指向函数的指针，而这些函数都有两个int型实际参数且返回标记为t的结构。

9. 在18.4节看到了下列非法的声明：

```
int f(int)[]; /* Functions can't return arrays */
int g(int)(int); /* Functions can't return functions */
int a[10](int); /* Array elements can't be functions */
```

然而，可以通过使用指针获得相似的效果：函数可以返回指向数组第一个元素的指针，也可以返回指向函数的指针，而且数组的元素可以是指向函数的指针。请根据这些描述重新修订上述每个声明。

### 18.5节

10. 下列哪些声明是合法的？（假设PI是表示3.14159的宏。）

- |                       |                                              |
|-----------------------|----------------------------------------------|
| (a) char c = 65;      | (b) static int i = 5, j = i * i;             |
| (c) float f = 2 * PI; | (d) double angles[] = {0, PI/2, PI, 3*PI/2}; |

11. 下列哪些类型的变量不能进行初始化？

- (a) 数组变量。 (b) 枚举变量。 (c) 结构变量。 (d) 联合变量。 (e) 不是上述类型的变量

12. 变量的哪种性质用来确定是否具有默认的初始值？

- (a) 存储期限。 (b) 作用域。 (c) 链接。 (d) 类型。

417

# 程序设计

只要有模块化就有可能发生误解：隐藏信息意味着需要检查沟通。

实际应用的程序显然比本书中的例子要大，但你可能还没意识到会大多少。更快的CPU和更大的主存已经使我们可以编写一些几年前还完全不可行的程序。图形界面的流行同样大大增加了程序的平均长度。如今，大多数功能完整的程序至少有100 000行代码，百万行级的程序也很常见，甚至上千万行的程序都听说过。

**Q&A** 虽然C语言不是专门用来编写大规模程序的，但许多大规模程序的确是用C语言编写的。这会很复杂，需要很多的耐心和细心，但确实可以做到。本章将讨论那些有助于编写大规模程序的技术，并且会展示哪些C语言的特性（例如static存储类）特别有用。

编写一个大型程序（通常称为“大规模程序设计”）与编写小型程序有很大的不同——就如同写一篇学期论文（当然是双倍的行间距）与写一本500页的书不同一样。一个大规模程序需要更加注意编写风格，因为会有许多人一起工作。需要有仔细的文档，同时还需要对维护进行规划，因为程序可能会多次修改。

尤其是，相对于小型程序，编写一个大规模的程序需要更仔细的设计和更详细的计划。正如Smalltalk程序设计语言的设计者Alan Kay所言，“*You can build a doghouse out of anything.*”建造一个狗舍可以不需要经过任何特别设计，也可以使用任何原材料。然而对于住人的房屋就不能这么干了，因为这要复杂得多。

在第15章曾经讨论过用C语言编写大规模的程序，但更多地侧重于语言的细节。本章会再次讨论这个主题，并着重讨论好的程序设计所需的技术。当然，要全面地讨论程序设计会超出了本书的范围。但会尽量简要地涵盖一些在程序设计中的重要观念，以及如何使用它们来编写出更易读、更易于维护的C程序。

19.1节讨论如何将C程序看作是一组相互服务的模块。随后，我们会看到如何使用信息隐藏（19.2节）和抽象数据类型（19.3节）来改进程序模块。19.4节会介绍C++，一种C语言的扩展版本，更好地支持了信息隐藏、抽象数据类型以及大规模程序设计的其他方面。

419

## 19.1 模块

当设计一个C程序（或其他任何语言的程序）时，最好将它看作是一些独立的模块。模块是一组功能（服务）的集合，其中一些功能可以被程序的其他部分（称为客户）使用。每个模块都有一个接口来描述所提供的功能。模块的细节，包括这些功能自身的源代码，都包含在模块的实现中。

在C语言环境下，这些“功能”就是函数。模块的接口就是头文件，头文件中包含那些可以被程序中其他文件调用的函数的原型。模块的实现就是包含该模块中函数的定义的源文件。

为了解释这个术语，我们来看一下第15章中的计算器程序。这个程序由calc.c文件和一个栈模块组成。calc.c文件包含main函数，而栈模块则存储在stack.h和stack.c中（见后面的图）。文件calc.c是栈模块的客户，文件stack.h是栈模块的接口，stack.c文件是栈模块

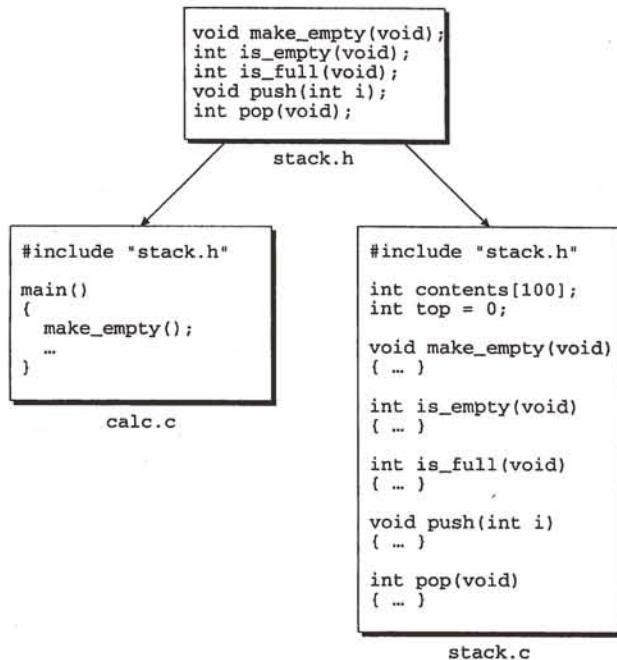
的实现，其中包括操作栈的函数的定义以及组成栈的变量的定义。

C函数库本身就是一些模块的集合。库中每个头文件都会作为一个模块的接口。以stdio.h为例，它就是包含字符串处理函数的模块的接口。

将程序分割成模块有一系列好处：

- **抽象。**如果一些模块是合理设计出来的，我们可以把它们作为抽象对待。我们知道模块会做什么，但不需要知道这些功能是如何被实现的细节。因为抽象的存在，使我们不必为了修改部分程序而了解整个程序是如何工作的。同时，通过抽象，我们可以更容易让一个团队的多个程序员共同开发一个程序。一旦对模块的接口达成一致，实现每一个模块的责任可以被分摊到各个成员身上。团队成员可以更大程度上相互独立地工作。

420



- **可复用性。**每一个提供一定功能的模块，都有可能在另一个程序中复用。例如我们的栈模块，就是可复用的。由于很难预测模块在将来是否需要，因此最好将模块设计成可复用的。
- **可维护性。**将程序模块化后，程序中的错误通常只会影响一个模块，因而更容易找到并解决错误。在解决了错误之后，重新编译程序只需要将该模块的实现进行编译即可（然后重新链接整个程序）。更广泛地说为了提高性能或将程序移植到另一个平台上，我们甚至可以替换一个完整的模块的实现。

上面这些问题都很重要，但其中可维护性是最重要的。现实中许多程序会使用许多年，在使用过程中会发现问题，并做一些改进和修改以适应需求的更新。将程序按模块进行设计会使维护更容易。维护一个程序就像维护一辆汽车一样，修理轮胎应该不需要同时检修引擎。

我们可以就近以第16章和第17章中的inventory程序为例。最初的程序（16.3节）将零件记录在一个数组中。假设在程序使用了一段时间后，客户不同意对存储的零件的数量有一个固定的上限。为了满足客户的需求，我们可能会考虑改成链表（正如在17.5节所做的）。为了做这个修改，需要仔细检查整个程序，找到所有依赖于零件存储方式的地方。如果我们一开始就采用不同的方式来设计程序——使用一个独立的模块来处理零件的存储，可能只需要重写这一个模块的实现，而不需要检查整个程序。

421

一旦我们已经认同了模块化程序设计是正确的方向，接下来的问题就是设计程序的过程中究竟应该定义哪些模块，每个模块应该提供哪些功能，各个模块之间的相互关系是什么？我们现在就来简要地看看这个问题。如果需要了解程序设计的更多信息，可以参考软件工程方面的书籍，像Ghezzi、Jazayeri、和Mandrioli的*Fundamental of Software Engineer* (Englewood Cliffs, N.J.:Prentice-Hall, 1991) 一书就是一个很好的选择。本书将采用和该书一样的术语。

### 19.1.1 内聚性与耦合性

一个好的模块接口并不是随意的一组声明。对于一个认真设计的程序，模块应该具有下面两个性质：

- **高内聚性。** 模块中的元素应该相互紧密相关。我们可以认为它们是为了同一目标而相互合作的。高内聚性会使模块更易于使用，同时使程序更容易理解。
- **低耦合性。** 模块之间应该尽可能相互独立。低耦合性可以使程序更便于修改，并方便以后复用模块。

我们的计算器的程序有这些性质吗？实现栈的模块是明显具有内聚性的，它的功能是实现与栈相关的操作。整个程序的耦合性也很低，文件calc.c依赖于stack.h（当然还有stack.c依赖于stack.h），除此之外就没有其他的明显的依赖关系了。

### 19.1.2 模块的类型

由于需要具备高内聚性、低耦合性，模块通常会属于下面几类：

- **数据池。** 数据池是一些相关的变量或常量的集合。在C语言中，这类模块通常只是一个头文件。从程序设计的角度说，通常不建议将变量放在头文件中。在C语言库中，<float.h> (►23.1节) 和<limits.h> (►23.2节) 都属于这类模块。
- **库。** 库是一组相关函数的集合。例如<string.h>就是字符串处理函数库的接口。
- **抽象对象。** 一个抽象对象是指对于隐藏的数据结构进行操作的一组函数的集合。（“对象”就是一组数据以及针对这些数据的操作的集合。如果数据是隐藏起来的，那么这个对象是“抽象的”。）
- **抽象数据类型。** 将具体数据实现方式隐藏起来的数据类型称为抽象数据类型。作为客户的模块可以使用该类型来声明变量，但不会知道这些变量的具体数据结构。如果客户模块需要对变量进行操作，则必须调用抽象数据类型所提供的函数。抽象数据类型在当今的程序设计中起着非常重要的作用。我们会在19.3节做更详细的讨论。

422

## 19.2 信息隐藏

一个设计良好的模块经常会对它的客户隐藏一些信息。例如我们的栈模块的使用者就不需要知道究竟栈是用数组实现的，还是用链表或其他方式实现的。这种谨慎地对客户隐藏信息的方法称为信息隐藏。信息隐藏有两大优点：

- **安全性。** 如果客户不知道栈是如何存储的，就不可能通过栈的内部机制擅自修改栈的数据。它们必须通过模块自身提供的函数来操作栈，而这些函数都是我们编写并测试过的。
- **灵活性。** 无论对模块的内部机制进行多大的改动，都不会很复杂。例如，我们可以首先将栈用数组实现，然后又改成是用链表或其他方式。我们当然需要重写这个模块，但是只要模块是按正确的方式设计的，就不需要改变模块的接口。

在C语言中，可以用于强行信息隐藏的主要工具是static存储类型 (►18.2.3节)。将一个函数声明成static类型可以使函数内部链接，从而阻止其他文件（包括模块的客户）调用这个

函数。将一个带文件作用域的变量声明成static类型可以达到类似的效果，使该变量只能被同一文件中的其他函数访问。

## 栈模块

为了清楚地看到信息隐藏所带来的好处，下面来看看栈模块的两种实现。一种是使用数组，另一种使用链表。我们假设模块的头文件如下所示：

```
stack.h
#ifndef STACK_H
#define STACK_H

void make_empty(void);
int is_empty(void);
void push(int i);
int pop(void);

#endif
```

423

注意，stack\_full的原型并没有放在头文件stack.h中。stack\_full函数在使用数组存储栈时是有意义的，但在使用链表来存储栈时就没有意义了。

首先，用数组实现这个栈：

```
stack1.c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

void make_empty(void)
{
 top = 0;
}

int is_empty(void)
{
 return top == 0;
}

static int is_full(void)
{
 return top == STACK_SIZE;
}

void push(int i)
{
 if (is_full()) {
 printf("Error in push: stack is full.\n");
 exit(EXIT_FAILURE);
 }
 contents[top++] = i;
}

int pop(void)
{
 if (is_empty()) {
 printf("Error in pop: stack is empty.\n");
 exit(EXIT_FAILURE);
 }
```

```
 return contents[--top];
}
```

用于实现栈的变量(`contents`和`top`)都被声明成`static`类型了,因为没有理由让程序的其他部分直接访问它们。这里在`stack.c`中包含了一个`is_full`函数,而且也被声明成`static`类型。因此对于程序的其他部分,它也被隐藏了。

在格式上,一些程序员使用宏来指明哪些函数和变量是“公有的”(即可以被程序的其他部分访问),哪些是“私有的”(即仅限该文件内访问):

```
#define PUBLIC /* empty */
#define PRIVATE static
```

将`static`写成`PRIVATE`是因为`static`在C语言中有很多的用法,使用`PRIVATE`可以更清晰地指明这里它是被用来强化信息隐藏的。下面是使用`PUBLIC`和`PRIVATE`后程序的样子:

```
PRIVATE int contents[STACK_SIZE];
PRIVATE int top : 0;

PUBLIC void make_empty(void) { ... }

PUBLIC int is_empty(void) { ... }

PRIVATE int is_full(void) { ... }

PUBLIC void push(int i) { ... }

PUBLIC int pop(void) { ... }
```

现在我们换成使用链表实现:

**stack2.c**

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct node {
 int data;
 struct node *next;
};

static struct node *top = NULL;

void make_empty(void)
{
 top = NULL;
}

int is_empty(void)
{
 return top == NULL;
}

void push(int i)
{
 struct node *new_node;

 new_node = malloc(sizeof(struct node));
 if (new_node == NULL) {
 printf("Error in push: stack is full.\n");
 exit(EXIT_FAILURE);
 }

 new_node->data = i;
 new_node->next = top;
 top = new_node;
}
```

```

int pop(void)
{
 struct node *old_top;
 int i;

 if (is_empty()) {
 printf("Error in pop: stack is empty.\n");
 exit(EXIT_FAILURE);
 }

 old_top = top;
 i = top->data;
 top = top->next;
 free(old_top);
 return i;
}

```

这里不再需要is\_full函数，因为栈已经没有固定大小了。然而，push函数需要测试malloc函数是否返回空指针。如果malloc返回空指针，则说明已经没有足够的内存来压入下一个元素了。幸运的是，is\_full函数在stack1.c中被声明成static，因此其他文件无法调用is\_full函数。由于这些文件并不知道有is\_full函数，删除这个函数也就不会影响到它们。

我们的栈示例清晰地展示了信息隐藏带来的好处：使用stack1.c还是使用stack2.c来实现模块无关紧要。两个版本具有同样的接口定义，因此可以相互替换，而不会影响程序的其他部分。

### 19.3 抽象数据类型

对于作为抽象对象的模块，像上一节中的栈模块，有一个缺点：不可能对同一对象有多个实例（在本例中，可以理解为有多个栈）。为了达到这个目的，我们需要进一步创建一个新的类型。

一旦定义了Stack类型，就可以有任意个栈了。下面的程序段显示了如何在同一个程序中有两个栈：

```

#include <stdio.h>
#include "stack.h"

main()
{
 Stack s1, s2;

 make_empty(&s1) ;
 make_empty(&s2);
 push(&s1, 1);
 push(&s2, 2);
 if (!is_empty(&s1))
 printf("%d\n", pop(&s1)) ; /* prints "1" */
 ...
}

```

我们并不知道s1和s2究竟是什么（结构？指针？），但这并不重要。对于栈模块的客户，s1和s2是抽象的对象，它只响应特定的操作（make\_empty、is\_empty、push以及pop）。

我们来将头文件stack.h改成提供Stack类型的方式。这需要给每个函数增加一个Stack类型（或Stack \*）的形式参数：

```

#define STACK_SIZE 100

typedef struct {
 int contents[STACK_SIZE];
 int top;
}

```

```

} Stack;
void make_empty(Stack *s);
int is_empty(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);

```

作为函数make\_empty、push和pop参数的栈变量需要定义为指针，因为这些函数会改变栈的内容。is\_empty函数的参数并不需要定义为指针，但这里我们仍然使用了指针。给is\_empty函数传递一个Stack指针比传递一个Stack值更有效，因为传递值会导致整个数据结构被复制。

## 封装

遗憾的是，上面的Stack不是抽象数据类型，因为stack.h暴露了Stack类型的具体实现方式，因此无法阻止客户将Stack变量作为结构直接使用：

```

Stack s1;
s1.top = 0;
s1.contents[top++] = 1;

```

427

通过提供对top和contents成员的访问，模块的客户可以破坏栈的数据。更糟糕的是，在没有检查是否需要修改客户之前，我们不能改变栈的存储方式。

我们真正需要的是一种阻止客户知道Stack类型的具体实现的方式。遗憾的是，C语言没有设计专门用于封装类型的特性。虽然确实有技巧可以达到类似目的，但使用起来相当笨拙，而且依赖性很强。**C++**实现封装的最佳方法是使用C++语言。C++语言允许我们隐藏数据类型的细节。实际上，C++语言产生的原因之一就是因为C语言不能很好地支持抽象数据类型。

---

## 19.4 C++语言

---

**C++**对于讨论程序设计的章中，如果没有谈及C++语言的话就是不完整的。C++语言是由AT&T贝尔实验室的Bjarne Stroustrup在20世纪80年代开发出来的C语言的扩展版。在现代程序设计理念上，包括抽象数据类型，C++语言比C提供了更好的支持。（当然，C语言是一种比较老的语言，所以我们并不能责怪它不支持新的程序设计技术。）C++语言中最重要的特性是支持类(class)。类使我们可以达到在19.3节中寻求的封装的效果。除此以外，C++语言提供了大量针对大规模程序设计的新特性，包括：

- 支持面对对象的程序设计，通过允许从已经存在的类“派生”出新的类，而不是从头编写新的类，从而确保了更高的代码复用率。
- 运算符重载，可以给传统的C语言的运算符赋予新的含义。运算符重载使我们可以定义新的数据类型，这些新类型甚至与基本类型毫无差别，从而扩展编程语言本身。
- 模板，可以使我们写出通用的、高度可复用的类和函数。
- 异常处理，一种统一的方式用来检测并响应错误。

C++语言的一个目标是尽可能保持与C语言的兼容。因此，C++语言中包含了标准C的全部特性。然而，这并不意味着所有C语言的程序都可以在C++的环境下编译。两种语言之间仍然存在一些小的差异，其中一些是由于C++语言增加了更多强制性限制，从而比C语言更加安全。

本节余下部分将提供C++语言的概述。注意，这里只会包含C++语言中的一些新特性，而且介绍也不会很完整。当然，你仍然可以从中了解C++语言是一种怎样的语言。

428

### 19.4.1 C语言与C++语言之间的差异

相对于C语言，C++语言增加的主要特性包括类、重载、派生、虚函数、模板以及异常处理。

但在进一步讨论这些新特性之前，我们需要讨论这两种语言之间的一些小差异。

### 1. 注释

C++语言支持单行注释。单行注释由//开始，在之后的第一个换行符处结束：

```
// This is a comment.
// So is this.
```

单行注释比C语言的注释（C语言的注释仍然是合法的）更安全，因为它们不会意外丢掉注释结束的标记。

### 2. 标记与类型名

在C++语言中，标记（用于标识特定的结构、联合或枚举的名字）会自动被认为是类型名。因此，我们将

```
typedef struct { double re, im; } Complex;
```

简单写成

```
struct Complex { double re, im; } ;
```

### 3. 不带参数的函数

在声明或定义一个不带参数的C++函数时，可以不使用void：

```
void draw(void); // no arguments
void draw(); // no arguments either
```

### 4. 默认实际参数

C++语言允许函数的实际参数有默认值。例如，下面的函数可以显示任意个数的换行符。

如果调用时没有提供任何实际参数，函数会显示一个换行符。

```
void new_line(int n = 1) // default argument
{
 while (n-- > 0)
 putchar('\n');
}
```

调用new\_line函数时，可以提供一个实际参数，也可以不提供实际参数：

```
new_line(3); // print 3 blank lines
new_line(); // prints 1 blank line by default
```

### 5. 引用参数

C语言规定实际参数是按值传递的，这使编写那些需要修改作为实际参数提供的变量（除了数组）的函数非常困难。为了回避这个限制，我们只能传递指向该变量的指针。对于一个将两个变量的内容交换的函数，其C语言的实现大致如下：

```
void swap(int *a, int *b)
{
 int temp;
 temp = *a;
 *a = *b;
 *b = temp;
}
```

当swap函数被调用时，其参数应该是指向变量的指针：

```
swap(&i, &j);
```

虽然这种方式可以正常工作，但它使用起来并不方便，代码也不易理解，还容易出错。C++语言在这方面做了一些改进，允许实际参数被声明成引用，而不是指针。下面是将a和b声明为引用后swap函数的样子：

```
void swap(int& a, int& b) // a and b are references
{
 int temp;
```

```

temp = a;
a = b;
b = temp;
}

```

当调用swap函数时，不需要在实际参数前加&运算符：

```
swap(i, j);
```

在swap的函数体中，a和b被分别理解为i和j的别名。语句temp = a确实会将i的值复制到temp中。语句a = b也确实将j的值复制到i中。同样，b = temp将temp的值复制给j。

## 6. 动态存储分配

C程序可以使用函数malloc、calloc、realloc和free来动态分配和释放内存。虽然C++程序仍然可以使用这些函数，但更好的做法是使用new和delete。new和delete是运算符，不是函数。new用来分配空间，delete用来释放分配的空间。new的操作数是一个类型说明符：

```

int *int_ptr, *int_array;
int_ptr = new int; // allocates memory for an int
int_array = new int[10]; // allocates memory for an array
 // of ten integers

```

当无法分配所要求的内存时，new会返回空指针。delete需要用一个指针作为它的操作数：

```

delete int_ptr; // releases memory pointed to by
 // int_ptr
delete [] int_array; // [] required when deallocating
 // an array

```

430

## 19.4.2 类

C语言与C++语言之间最重要的区别在于C++语言支持类（class）。（类对于C++的重要性体现在C++语言最初的名字中：“C with Classes”。）一个类根本上说就是一个抽象数据类型：一组数据以及操作这些数据的函数。通过编写一个类，可以产生一个新的数据类型。这个新数据类型的功能可以同基本数据类型同样强大。

假设我们需要以分数形式存储数，例如1/4、3/7等。如果我们编写一个类Fraction，就可以很方便地操作这些分数。可以按下面的方式声明Fraction类的变量：

```
Fraction f1, f2, f3;
```

可以使用=运算符复制分数，将分数传递给函数，或编写函数返回分数。

不仅如此，通过使用“运算符重载”的特性，我们可以将C++运算符用作对Fraction对象的操作的名字。通过重载\*运算符，可以使它实现分数间的乘法。我们将可以使用下面的代码将f1与f2相乘：

```
f3 = f1 * f2;
```

假设f1的值为1/2，f2的值为2/3，f3会被赋值为1/3。允许重载运算符使它们可以用于分数，这是使Fraction类可以与int和float一样简单易用的重要进步。

类允许我们构造任何需要的数据类型。如果我们需要一种C++语言通常不支持的数值类型（如分数、复杂的数、有无限个数的整数等），我们可以通过设计一个合适的类将这种类型添加到语言中。如果对C++语言的类型的通常行为不满意，可以构造自己的类型。例如，自己定义的Array类中，可以有一个变量对下标进行越界检查。而我们自己的String变量可以根据需要扩展或缩短。类同样适合构造没有作为C++类型提供的复杂数据结构，如队列、集合、栈等。

然而，真正使类最有意义的是它可以用来对现实世界中的对象建模，而不仅仅是作为程序通常使用的数据结构。假设我们在开发一个银行的程序，我们可能会定义诸如Account这样的类。Account类能提供类似deposit和withdraw的操作。类似这样的类会使程序更易读也更易编写，因为这样的操作更接近实际。

使用类的不足之处在于，类的设计和实现比较复杂。这是我们为易用性必须付出的代价，而这也同样是计算机领域近几年内的妥协：随着程序使用起来越来越方便，程序的内部实现也越来越复杂。

### 19.4.3 类定义

在C++语言中定义一个类非常像在C语言中定义一个结构。在最简单的情况下，类的定义几乎与结构的定义一模一样，只是将struct替换为class：

```
class Fraction {
 int numerator;
 int denominator;
};
```

numerator和denominator称为Fraction类的数据成员（data member）。顺便提一下，C++语言并不要求类名以大写字母开始，这只是许多C++程序员所遵循的规范。

一旦一个类被定义了，我们可以使用这个名字来声明变量，声明的方式与使用结构名一样：

```
Fraction f1, f2;
```

（类标记(class tag)在C++语言中可以直接作为类型名使用，因此不需要写成class Fraction。）编译器会构造两个变量f1和f2。每个变量都有自己的成员numerator和denominator。C++语言对f1和f2这样的变量有一个特殊的称呼，它们被称作Fraction类的实例（instance）。任何类的实例就是对象（object）。

结构的成员可以用运算符.`和`->访问。然而在类中，成员默认是隐藏的。因此，下面的语句是非法的：

```
f1.numerator = 0; // illegal
denom = f2.denominator; // illegal
```

我们称numerator和denominator是Fraction类的私有（private）成员。

我们可以通过将成员声明为public（公有）使这些成员可以被访问：

```
class Fraction {
public:
 int numerator;
 int denominator;
};
```

甚至可以混合使用公有的和私有的成员：

```
class Fraction {
public:
 int numerator; // accessible outside the class
private:
 int denominator; // hidden within the class
};
```

**432** 注意private的用法：指明denominator是一个私有成员。

### 19.4.4 成员函数

既然类的私有成员不能从类的外面进行访问，那么怎么修改它们或是检查它们的值呢？对这个问题的回答是十分巧妙的：那些需要访问类的数据成员的函数必须声明在类里面。属于类的函数称为成员函数（member function）。

让我们将numerator和denominator设置为Fraction类的私有成员，并给类添加两个成员函数：

```
class Fraction {
public:
 void create(int num, int denom);
```

```
void print();
private:
 int numerator;
 int denominator;
};
```

create和print是Fraction类的公有成员，因此它们可以在类以外的地方调用。

成员函数通过对对象调用，所使用的运算符“点”和访问结构的成员时所用的是一样的：

```
f1.create(1, 2); // f1 now stores 1/2
f1.print(); // prints "1/2"
```

无可否认，这看起来确实有点怪，但你会很快习惯的。下面是对create调用的理解：

“f1是Fraction类的对象，因此我们调用的是Fraction类中的create函数。

create函数会将1存到f1的numerator成员中，将2存到f1的denominator成员中。”

下面是print函数调用的含义：

“f1是Fraction类的对象，因此我们调用的是Fraction类中的print函数。

print函数会显示f1的numerator成员，接着显示一个/字符，然后显示f1的denominator成员。”

注意成员函数会知道是哪个对象在调用它，即使对象本身并不作为函数的实际参数。我们可以想象f1是一种放在函数名前面的实际参数，而不是放在实际参数列表中。

成员函数并不一定需要是公有的。在Fraction的例子中，我们可以增加reduce（用于简化分数）作为私有成员函数：

```
class Fraction {
public:
 void create(int num, int denom);
 void print();
private:
 void reduce();
 int numerator;
 int denominator;
};
```

433

在实际使用中，数据成员通常被声明成私有的。成员函数通常被声明成公有的，除非它们仅为了类的内部实现使用。

到目前为止，我们仅仅是声明了create、print和reduce这些函数，那么它们的定义在哪儿？一个可行的做法是稍后在类定义之外定义每一个成员函数。例如，函数create的定义可能会如下所示：

```
void Fraction::create(int num, int denom)
{
 numerator = num;
 denominator = denom;
 reduce();
}
```

注意，函数名前面添加了Fraction::前缀。这是必需的，否则C++编译器会将create作为一个普通的函数，而不是Fraction类的成员。还要注意create是直接访问numerator和denominator的。通常来说，成员函数可以访问类的所有成员，包括公有的和私有的。最后，请注意reduce的调用。它看起来有些奇怪，因为它似乎没有指明简化哪个对象。实际上，当一个成员函数调用另一个成员函数时，后者会默认认为是从同一个对象中调用的。换言之，下面的调用：

```
f1.create(1, 2);
```

就如同执行了下面的语句：

```
f1.numerator = num;
f1.denominator = denom;
f1.reduce();
```

除了在类之外定义成员函数以外，我们还可以选择将整个函数放在类的定义之中：

```
class Fraction {
public:
 void create(int num, int denom)
 { numerator = num; denominator = denom; reduce(); }

};
```

将成员函数的定义放在类定义之中只在函数的实现非常短小时才可以考虑。

434

现在，来给Fraction类添加乘法函数。首先，我们需要在类的定义中声明成员函数：

```
class Fraction {
public:
 void create(int num, int denom);
 void print();
 Fraction mul(Fraction f);
private:
 void reduce();
 int numerator;
 int denominator;
};
```

接下来，需要编写函数mul的定义：

```
Fraction Fraction::mul(Fraction f)
{
 Fraction result;

 result.numerator = numerator * f.numerator;
 result.denominator = denominator * f.denominator;
 result.reduce();
 return result;
}
```

乍一看，mul函数令人不解：f绝对是参与乘法的分数之一，但另一个分数在哪儿呢？这个问题的答案在于调用mul的方法：

```
f3 = f1.mul(f2);
```

下面是这条语句的含义：

“f1是一个Fraction对象，因此我们调用的是Fraction类的mul函数。mul函数会将f1的分子与f2的分子相乘，然后将产生的值存放在result的分子中。接着，mul函数会将f1的分母与f2的分母相乘，然后将产生的值存放在result的分母中。接下来，mul会调用reduce来化简result的分数。最后，mul返回result，而result将被复制到f3中。”

#### 19.4.5 构造函数

为了确保正确地初始化类的实例，类可以包含一个特殊的函数，称为构造函数（constructor）。类还可以提供一个析构函数（destructor）——当释放类的实例时进行清理。构造函数和析构函数最方便的（当然也是危险的）地方在于它们通常是被自动调用的，不需要明确的函数调用。换而言之，我们需要给我们的类编写构造函数和析构函数，编译器会安排在需要的时候自动调用它们。

Fraction类已经有一个初始化用的函数create了。我们来将create替换为构造函数。构

造函数看起来像一个与类同名的函数：

```
class Fraction {
public:
 Fraction(int num, int denom)
 { numerator = num; denominator = denom; reduce(); }
 ...
};
```

与其他的成员函数不同，构造函数没有指定的返回类型。注意构造函数被放在类的public成员部分中。

构造函数可以像其他函数一样调用，但它们通常是在声明实例时隐式调用：

```
Fraction f(3, 4); // declares and initialize f
```

在上面f的声明中，Fraction类的构造函数会被调用，调用时的实际参数是3和4。结果，f的初始值为3/4。

构造函数通常会有默认实际参数：

```
class Fraction {
public:
 Fraction(int num = 0, int denom = 1)
 { numerator = num; denominator = denom; reduce(); }
 ...
};
```

由于num和denom有默认值，Fraction构造函数在调用时可以有两个实际参数：

```
Fraction f(3, 4);
```

一个参数：

```
Fraction f(3); // same as Fraction f(3, 1);
```

或没有参数：

```
Fraction f; // same as Fraction f(0, 1);
```

## 19.4.6 构造函数和动态存储分配

构造函数和析构函数对那些需要动态分配存储空间（使用new和delete运算符）的函数特别有用。例如，假如我们已经受够了普通的C字符串的限制。创建自己的String类可以带来几大好处：

- String对象可以包含任意长度的字符串。在C语言中，字符串的大小受限于数组的长度。
- String对象的长度可以迅速地确定。要得到C语言字符串的长度，需要调用strlen函数，而strlen函数则需要遍历整个字符串来找到标志字符串末尾的空字符。
- 需要时可以给String类添加操作。在C语言中，我们不能方便地修改<string.h>来增加函数。

下面是可以用来声明String对象的方式：

```
String s1("abc"), s2("def");
```

初始化后，s1会包含"abc"，s2会包含"def"。当然，这些变量的值都可以随后修改。

由于对字符串的长度没有限制，String对象需要包含一个指针来动态地分配内存（我们将这个指针命名为text）。出于运行速度的考虑，我们还需要一个成员来保存字符串的长度：

```
class String {
 ...
private:
 char *text; // pointer to string
 int len; // length of string
};
```

接下来，需要一个构造函数来将普通的字符串转换成String对象：

```
class String {
public:
 String(const char *s); // constructor
 ...
private:
 char *text;
 int len;
};
```

下面是构造函数大概的实现：

```
String::String(const char *s)
{
 len = strlen(s);
 text = new char[len+1];
 strcpy(text, s);
}
```

在计算了s所指向的字符串的长度后，构造函数使用new运算符分配足够的内存来复制字符串。最后，构造函数将字符串复制到刚分配的内存中。

#### 19.4.7 析构函数

与动态存储分配有关类有一个很难对付的问题。思考在一个函数内部使用的String对象会发生什么：

```
void f()
{
 String s1("abc");
 ...
}
```

437

当f被调用时，s1对象开始存在。s1的构造函数分配了一个4个字符的数组，并将字符串"abc"复制到数组中。当函数f返回时，s1将不再存在，因为s1使用的是自动存储期限。不幸的是，释放String对象所占的内存时，仅释放了成员text和len使用的内存，而不会释放text所指向的内存。结果，程序会出现内存泄漏。

释放动态分配内存的问题是C++语言提供析构函数的原因之一。析构函数会在对象被释放时自动被调用。构造函数和析构函数关系密切。构造函数在对象诞生时对它进行初始化；析构函数在对象消亡时进行清理。如果一个类的构造函数动态分配了内存，析构函数很可能要释放这块内存。

与构造函数一样，析构函数也是一个成员函数。析构函数的名字与类名一致，只是在开头带了一个~（波浪）字符。析构函数没有返回类型，也没有实际参数。下面是添加了析构函数后String类的样子：

```
class String {
public:
 String(const char *s);
 ~String() { delete [] text; } // destructor
 ...
private:
 char *text;
 int len;
};
```

析构函数~String会释放text所指向的字符数组。

#### 19.4.8 重载

在C++语言中，同一作用域中的两个或更多的函数可以有同样的函数名。当函数以这种方

式重载时，编译器会根据检测函数的实际参数来决定哪个函数被调用。例如，假如在同一作用域中有两个版本的函数f：

```
void f(int);
void f(double); // overloading
```

下面显示了f调用是如何被转变的：

```
f(1); // a call of f(int)
f(1.0); // a call of f(double)
```

重载有一个主要的优点：对于执行相同操作，但操作数的类型不同的函数，可以有相同的函数名。因此，会需要记住更少的函数名。例如，下面的函数同样是计算x的y次方，但实际参数的类型不同：

```
int pow(int x, int y);
double pow(double x, double y);
```

类中的成员函数可以被重载。（实际上，这可能是函数重载在C++语言中最常见的使用方式了。）例如，通过重载我们可以给String类添加另一个构造函数：

```
class String {
public:
 String(const char *s);
 String() {text = 0; len = 0;} // overloading
 ~String() { delete [] text; }
 ...
private:
 char *text;
 int len;
};
```

如果你还在考虑为什么text被赋值为0，那么请回忆一下我们讲过0代表空指针。C++程序员通常更喜欢用0而不是NULL，其具体原因就不在这里讨论了。

这个新String的构造函数称为默认的构造函数，因为它不带实际参数。它会在声明String对象而没有指定初始值时被调用：

```
String s; // default constructor is invoked
```

除了函数的重载，C++语言还支持运算符的重载：根据操作数类型的不同，同样的运算符号可以代表不同的操作。运算符重载使我们可以重新定义C++语言的运算符，来使用在类的实例上。这样产生的程序看起来更自然，也更易读。而且类的客户可以使用运算符来执行操作，而不需要调用那些名字难以记住的函数了。

例如，如果将mul函数替换为\*运算符，Fraction类就会更好用。做起来很简单，只需要在声明函数时用operator\*替换mul就行了：

```
class Fraction {
public:
 ...
 Fraction operator*(Fraction f);
private:
 ...
};
Fraction Fraction::operator*(Fraction f)
{
 // same body as mul function
}
```

函数的内部实现是一样的，只有名字被改变了。

当一个运算符被定义为类的成员时，它其中的一个操作数始终是隐含的。因此，我们所定义的运算符\*是一个二元运算符，而不是一元运算符。当我们写如下语句时：

```
f3 = f1 * f2;
```

438

439

编译器会意识到f1是一个Fraction对象，所以会查看Fraction类，找到名字为operator\*的函数，然后将语句转换成：

```
f3 = f1.operator*(f2);
```

随后，接下来的操作与执行一般的成员函数一样。

### C++语言中的输入/输出

虽然C++程序可以使用<stdio.h>，但C++提供了额外的I/O库。<iostream.h>是新函数库中最重要的头文件。它定义了几个类，包括istream（输入流）和ostream（输出流）。I/O通过对istream和ostream操作来进行。那些从键盘获得输入和在屏幕上显示输出的简单程序可以使用cin对象来进行输入，用cout对象来进行输出。cin是istream类的一个实例，cout是ostream类的一个实例。

istream类和ostream类都与运算符重载关系密切。特别是，C运算符<<和>>（向左或向右移位）（>20.1.1节）用于绝大多数读和写操作中。istream类重载了>>，使它可以从输入流中读取数据。ostream类重载了<<，使它可以从输出流中读取数据。使用<<和>>进行交互的形式大致如下：

```
cout << "Enter a number: ";
cin >> n;
cout << "The square is ";
cout << n * n;
cout << "\n";
```

第一条语句有这样的含义：“cout是一个ostream对象，所以调用的是ostream类的operator<<函数。operator<<函数会将字符串“Enter a number”作为它的实际参数。”

使用新I/O库的一个好处是可以将它们扩展，用来读/写我们编写的类的实例。例如，我们可以使用重载运算符<<来写一个Fraction对象：

```
Fraction f(3, 4);
cout << f; // prints 3/4
```

能够使用<<来输出分数是我们又向着“使Fraction类和基本类型一样好用”的目标迈进了一步。

### 19.4.9 面向对象编程

虽然专家们还在讨论对“面向对象”的编程语言的确切要求，但对于语言必须具有下述能力的还是达成了广泛的认同：

- **封装**——具有能够定义新的类型以及一组对这个类型的操作的能力，而不会暴露类型的具体实现。（在“面向对象”的环境中，类型的值就是“对象”。）C++语言通过限制对私有数据成员的访问，从而支持了封装。
- **继承**——具有能够定义新的类型，并在新的类型中继承已经存在的类型的属性的能力。C++语言通过派生类来实现继承。
- **多态性**——对于同样的操作，对象能够根据它所属于的类采取不同的响应。在C++语言中，虚函数支持了多态性。

我们已经讨论过类，因此下面来看看类的派生和虚函数。

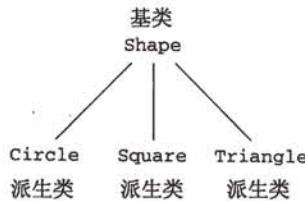
### 19.4.10 派生

需要一个新类时，C++语言允许从一个已经定义的类派生出这个新类，而不需要重新写这

个新类。例如，一个用来生成图形的程序可能需要叫作Circle、Square以及Triangle的类。这些类可以都从一个更通用的类Shape派生出来。对于3个类共同的属性，可以只在Shape类中定义一次；同时对于所有形状都适用的操作也可以定义在Shape类中。如果每个形状都有颜色和屏幕上的x-y坐标，而且每个形状都可以改变它的位置和颜色，那么Shape类可能如下所示：

```
class Shape {
public:
 void change_color(int new_color);
 void move(int x_change, int y_change);
 ...
private:
 int x, y; // coordinates of center
 int color; // current color
 ...
};
```

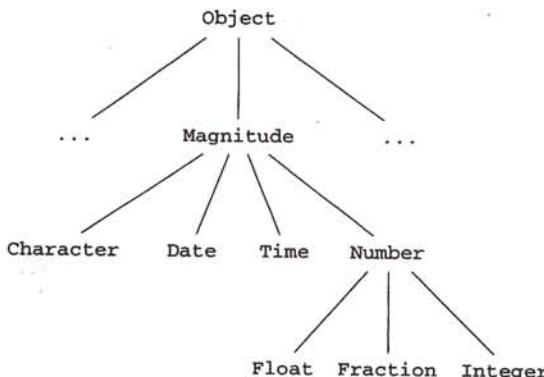
Shape类称为基类。Circle、Square和Triangle称为派生类。下面的图显示了这些类之间的关系。



派生的一大优势是可以帮助大量地复用代码。假如稍后需要添加一个Pentagon（五角形）的类，我们可以从Shape类派生出来。这比重新写Pentagon类容易得多，因为Pentagon可以从Shape中继承大多数它所需要的属性。我们只需要编写那些Pentagon与Shape不共有的属性代码就可以了。

派生也可以使维护更简单。假如Shape类有错误，我们只需要在Shape中修改就可以，不需要改动派生类。同样，如果我们要给所有形状添加一个新属性（或操作），也只要加到Shape类供派生类继承就可以了。

派生经常被用来开发可以扩展的相关类的库。在Smalltalk（一种影响了C++语言的面向对象编程语言）中，所有的类都会直接或间接地从一个单一的Object类派生。例如其中Magnitude类从Object类派生出来，然后Character、Date、Time和Number类再从Magnitude类派生，而像Float，Fraction和Integer类则都是从Number类派生出来的：



每个类都会包含它的基类中的所有属性，以及它自身所特有的属性。Magnitude类的对象有一个共同的特性，就是可以使用关系运算符（大于、小于等）进行比较。Number类的对象继

承了这一功能，同时提供额外的支持对算术运算的功能。而算术运算不会提供给其他Magnitude对象。(对日期进行比较是有意义的，但将日期相加就不合理了。)

我们已经学习了派生的概念。下面我们回到Shape的例子说明派生的具体实现。为了指明

**442** Circle类从Shape类派生的，在Circle的定义中需要包含一个派生列表：

```
class Circle: public Shape {
 ...
};
```

Circle继承了Shape类的成员（除了构造函数和析构函数）。换而言之，Shape类中的成员也是Circle类的成员。

派生类可以声明额外的在基类中不存在的数据成员和成员函数。例如，Circle类会需要一个数据成员来保存圆的半径。Shape类并没有这样一个radius成员，因此需要添加到Circle中：

```
class Circle: public Shape {
public:
 ...
private:
 int radius; // radius of circle
};
```

除了radius数据成员，Circle类的对象还会包含成员x, y和color（从Shape继承的）。

当一个类从另一个类派生时，C++语言允许基类的指针指向派生类的实例。例如，Shape\*类型的变量可以指向Circle、Square或Triangle对象：

```
Circle c;
Shape *p = &c; // Shape pointer points to a Circle
```

类似地，C++语言允许Shape\*类型的形式参数，与指向Circle、Square或Triangle对象的实际参数匹配。Shape&类型的参数也可以匹配任何Circle、Square或Triangle对象。下面的函数表面上要求一个Shape类型的实际参数，但实际上也可以使用Circle、Square或Triangle类型的对象：

```
void add_to_list(Shape& s)
{
 ...
}
```

add\_to\_list是一个可以灵活处理各种形状的函数。

### 19.4.11 虚函数

当类派生与虚函数结合使用时会更有价值。虚函数可以在基类中声明，然后在各个派生类中提供不同的实现。以Shape类为例，每个Shape对象都可以增大它的大小，因此需要一个grow函数：

```
class Shape {
public:
 void grow();
 ...
};
```

但是，grow函数对每个从Shape派生的类都不一样。为了解决这个问题，可以在Shape类中将grow声明成virtual：

```
class Shape {
public:
 virtual void grow();
 ...
};
```

现在，Circle、Square和Triangle类可以各自提供自己的grow版本了。Circle类的版本大概如下所示：

```
class Circle: public Shape {
public:
 void grow() { radius++; }
 ...
private:
 int radius; // radius of circle
};
```

大多数情况下，虚函数与普通的成员函数有相同的行为。假设c是一个Circle对象，调用c.grow()会增加c的半径。然而当通过基类的指针（或引用）调用该函数时，虚函数会有特殊的处理：根据指针当前所指向的对象的实际类型，来决定所调用的函数的版本。

假设p是一个指向Shape对象的指针。由于Circle和Square都是从Shape类派生的，p可以指向其中一个类的实例。当使用p调用grow函数时，如果p指向的是一个Circle对象，则调用Circle::grow()，而如果p指向的是一个Square对象，那么Square::grow()会被调用：

```
Shape *p;
Circle c;
Square s;

p = &c; // p points to a Circle
p->grow(); // calls Circle::grow
p = &s; // p points to a Square
p->grow(); // calls Square::grow
```

注意这里使用->来调用grow函数。通过指向对象的指针调用成员函数时，需要使用->运算符而不是.运算符。

虚函数依赖于一种称为“动态绑定”的技术，这是由于编译器并不总是能判断应该调用函数的那个版本。（对于普通函数调用，甚至对于重载函数的调用，编译器都可以判断出正确的函数版本。）考虑下面的例子：

```
if (...) {
 p = &c;
} else
 p = &s;
p->grow(); // calls either Circle::grow or Square::grow
```

编译器无法知道应该调用grow的哪个版本，这一点直到程序运行时才可以知道。

动态绑定技术使我们可以构造数据结构来包含不同类的对象（只要这些类都是从同一个基类派生出来的），然后对每个对象执行相同的操作。每一个对象会根据它自己所属的类，采取不同的响应。例如，假设我们通过把Shape对象存储在一个列表中来跟踪当前在屏幕上显示的对象。通过逐个访问每个对象，并调用它的grow函数，我们可以改变它的大小，而不需要知道它具体是什么形状：

```
while (不在列表结尾) {
 让p指向当前形状;
 p->grow(); // calls either Circle::grow, Square::grow,
 // or Triangle::grow
 向前至列表中下一项;
}
```

动态绑定可以简化我们的代码，因为我们将不需要在执行操作前使用switch语句来检测每个对象了。而且我们也可以增加新类或删除旧类，而不必改动操作数据结构的代码。例如，如果我们增加了一个Pentagon类（由Shape类派生），不需要改动扩大每个形状的循环。

444

### 19.4.12 模板

模板是构造类所使用的“模式”。(C++语言也支持函数模板，不过本书不讨论。)除了对类定义的部分内容未加具体说明以外，模板看起来很像一个普通的类。忽略类定义的部分内容可以使类更通用，也更易复用。

考虑19.3节的Stack类型。如果将它转换成一个C++的类，可能会得到如下定义：

```
class Stack {
public:
 void make_empty();
 int is_empty();
 void push(int);
 int pop();
 ...
};
```

445

遗憾的是，这个Stack对象只能存储整数。如果以后需要一个可以存储其他类型数据（例如float值）的栈时，可以将类定义复制一份，改变类名，将分散各处的int改为float。但构造一个Stack模板才是个更好的办法：

```
template <class T>
class Stack {
public :
 void make_empty ();
 int is_empty();
 void push(T);
 T pop();
 ...
};
```

除了下面这行代码，Stack模板看起来非常像Stack类：

```
template <class T>
```

这行代码指明Stack是一个模板，直到将缺少的类T补充进来才会完整。T（一个“模板实际参数”）只是一个假名，任何名字都可以。注意push现在需要参数是T类型的，同时pop会返回一个T类型的值。Stack的成员函数会如下所示：

```
template <class T>
void Stack<T>::push(T, x)
{
 ...
}
```

模板类在提供了模板参数后才会被“实例化”。虽然写成<class T>，Stack的实际参数不需要一定是个类，任何C++类型都可以。例如，下面是Stack类的3种不同的实例化方式，分别使用了int、float和char作为模板实际参数：

```
Stack<int> int_stack; // stack of int values
Stack<float> float_stack; // stack of float values
Stack<char> char_stack; // stack of char values
```

下面对push的调用解释了如何使用这三种栈：

```
int_stack.push(10); // pushes 10 onto int_stack
float_stack.push(1.2); // pushes 1.2 onto float_stack
char_stack.push('a'); // pushes 'a' onto char_stack
```

### 19.4.13 异常处理

异常是指程序运行时可能产生的情况，通常作为出错的结果。例如：使用Stack时可能产生两种错误：在栈已满时试图向栈中压入数据，或栈为空时试图弹出数据。我们可以使用名字

为StackFull和StackEmpty的异常来表示这两种错误。

当错误发生时，函数可以“抛出”一个异常。在Stack类的例子中，push函数和pop函数可以相应地抛出StackFull和StackEmpty异常：

```
void Stack::push(int i)
{
 if (无空间)
 throw StackFull();
 ...
}

int Stack::pop()
{
 if (栈空)
 throw StackEmpty();
 ...
}
```

有可能发生异常的代码会被放在“try程序块”中。异常会被“处理程序”(“catch程序块”)捕获。在下面的例子中，一个包含push和pop调用的try程序块跟着两个catch程序块。第一个catch程序块处理StackFull异常，显示Error: Stack full；第二个catch程序块处理StackEmpty异常，显示Error: Stack empty。

```
try {
 ...
 s.push(y);
 ...
 z = s.pop();
 ...
}
catch (StackFull) {
 cout << "Error: Stack full\n";
}
catch (StackEmpty) {
 cout << "Error: Stack empty\n";
}
```

在异常处理之后，程序会从最后的catch程序块后面的语句继续执行。

对于一个异常，如果在当前的try程序块末尾没有相应的catch来捕获，C++语言并不会放弃。实际上，它会检查被包围的try程序块来寻找相应的处理者。如果这次查找失败了，则会中止当前函数，并将当前的异常传播给调用函数处理。如果需要会继续传播给再上层的调用者，以次类推。最坏的情况下，异常会一直传播到main函数。如果main函数也无法处理这个异常，整个程序会被终止。异常的这种性质可以帮助避免错误被意外地忽略。

## 问与答

问：本节中提到C语言不是为开发大型程序设计的。UNIX不是大型程序吗？(p.291)

答：在C语言被设计出来时还不是。在1978年的一篇论文中，Ken Thompson估计UNIX内核大约是10 000行C代码（加上一小部分汇编代码）。UNIX的其他部分也有类似的大小。在另一篇1978年的论文中，Dennis Ritchie和他的同事将PDP-11的C编译器的大小设定为9660行。按现在的标准，这绝对只是小型程序。

问：C语言库中有什么抽象数据类型吗？

答：从技术上说，没有。但有一些很接近，包括FILE类型（定义在<stdio.h>中）。在对一个文件进行操作之前，必须声明一个FILE \*类型的变量：

```
FILE *fp;
```

这个fp变量随后会被传递给不同的文件处理函数。

程序员需要把FILE作为一个抽象的类型，在使用时不需要知道FILE具体是怎样的。假设FILE是一个结构类型，但C标准并不保证这一点。实际上，最好不要管FILE值究竟是如何存储的，因为FILE类型的定义对不同的编译器可能（也确实经常）是不一样的。

当然，我们总是可以查看stdio.h找到FILE到底是什么。如果这么做，那么就没什么可以阻止我们编写代码来访问FILE的内部机制。例如，我们可能发现FILE结构中有一个叫bsize（文件的缓冲区大小）的成员：

```
typedef struct {
 ...
 int bsize; /* buffer size */
 ...
} FILE;
```

一旦我们知道了bsize成员，就无法阻止我们直接访问特定文件的缓冲区大小：

```
printf("buffer size: %d\n", fp->bsize);
```

然而，这样做并不是个好主意，因为其他的C编译器可能将缓冲区大小存在其他名字中，或者是用其他方式跟踪这个值。试图修改bsize的值则是个更糟糕的做法：

**448**    fp->bsize = 1024;

这是一件非常危险的事，除非我们知道文件存储的全部细节。即使我们的确知道相关的细节，这样做对于不同的编译器或是同一编译器的更新版本也同样是非常危险的。

问：如果C++语言真的那么好，为什么还有人使用C语言呢？

答：以某种角度来说，这个问题是没有意义的。C++语言包含了C语言的全部特性，所以所有使用C++语言的人就在“使用”C语言。我们来换个角度问这个问题：“如果C++语言真的那么好，为什么不是所有人都使用C++语言呢？”

其一，C++比C复杂得多。由于C++实际上继承了所有C语言的特性，同时增加了大量新特性，C++显然是一个庞大的语言。C++不同功能之间的多种组合也进一步增加了语言的复杂程度。对于编写小程序，C语言更简单而且使用起来与C++不相上下。

C++所提倡的新特性需要编译器做更多的工作。因此，C++程序编译起来会比C程序慢一些。此外，使用C++的新特性会对程序的运行性能带来一些负面影响，这种影响虽然较小，但是可以察觉的。这对于一部分程序来说可能是不可接受的。

虽然C++解决了C语言的一些著名的隐患，但仍有一些没有涉及。当然，C++的新特性也会带来一些新的陷阱，而这些陷阱是C语言没有的。正如Stroustrup自己评价的那样：“C语言使你很容易击中自己的脚。C++使这变得困难了，一旦击中，它会炸飞你整条腿。”

不要忘记C语言存在的时间比C++长许多。虽然经历了几年的改动后，C++已经开始逐渐稳定下来了。但C++编译器仍需要一段时间来达到C编译器已经提供的能力。除此以外，与C++相比，C语言有更多种类的编译器，尤其在对一些不太流行的平台上。

总之，对于“简而达意”的程序，以及需要更广泛的移植性的程序，C语言更适合。对于大型的、功能齐全的程序（包括那些有复杂的图形用户界面的程序）来说，C++更强一些。

## 练习

### 19.1节

1. 队列类似于栈，两者的差异是队列的数据从一端添加，而从另一端按FIFO（先进先出）的方式删除。

对于队列的操作可以包括：

- 向队列的末端加入一个数据。

- 从队列的开始删除一个数据。
- 返回队列第一个数据（不改变对列）。
- 返回队列的末尾数据（不改变对列）。
- 检查队列是否为空。

以头文件queue.h的形式给队列定义一个接口。

#### 19.2节

2. 修改文件stack2.c，以使用PUBLIC宏和PRIVATE宏。
3. (a) 按照练习1中的描述用数组实现一个队列模块。  
(b) 按照练习1中的描述用链表实现一个队列模块。

449

#### 19.3节

4. (a) 编写一个基于数组的stack类型的实现。  
(b) 使用链表替换数组，重写上面的stack类型。（给出stack.h和stack.c。）
5. (a) 将练习1中的queue.h头文件加以修改，使之定义一个Queue类型。同时修改queue.h头文件中的函数，用Queue（或Queue\*）作为形式参数。  
(b) 使用数组实现Queue类型。  
(c) 使用链表实现Queue类型。

#### 19.4节

6. Fraction类需要一个析构函数吗？验证你的答案。
7. 给Fraction类添加重载运算符+、-和/。写出这些运算符的定义以及print函数和reduce函数的实现。
8. 与C语言的printf和scanf相比，C++语言的<<和>>运算符有什么优点？
9. 将练习5的Queue类型改成Queue类。
10. 将练习9的Queue类改成Queue模板。

450

当程序要求关注不相干的内容时，所用的编程语言就是低级的。

前面几章中讨论的是C语言中高级的、与机器无关的特性。虽然这些特性对不少程序都够用了，但仍有一些程序需要进行位级别的操作。位操作和其他一些低级运算在编写系统程序（包括编译器和操作系统）、加密程序、图形程序以及其他一些需要高执行速度或高效地使用空间的程序时非常有用。

20.1节介绍C语言的按位运算符。按位运算符提供了对单个位或位域的访问。20.2节介绍如何声明包含位域的结构。最后，20.3节描述如何使用一些普通的C语言特性（类型定义、联合和指针）来帮助编写低级程序。为了方便说明，本节中的所有例子都是16位的，而且可以方便地将这些示例扩展到32位。这里的讨论也不会依赖于特定的操作系统，20.3节的部分内容针对DOS编程除外。

本章中描述的一些技术需要用到数据在内存中如何存储的知识，这对不同的机器和编译器可能会不同。依赖于这些技术很可能使程序丧失可移植性，因此除非必要，否则最好尽量避免使用它们。如果确实需要，尽量将使用限制在特定的模块中，不要分散在各处。同时，最重要的是确保使用文档记录所做的事！

## 20.1 按位运算符

C语言提供了6个按位运算符。这些运算符可以用于在整数和字符上进行按位运算。这里先讨论移位运算符。

### 20.1.1 移位运算符

移位运算符可以改变数的二进制形式，将它的位向左或向右移动。C语言提供了两个移位运算符。参见表20-1。

表20-1 移位运算符

| 符    号 | 含    义 |
|--------|--------|
| <<     | 左移位    |
| >>     | 右移位    |

运算符<<和运算符>>的操作数可以是任意整型或字符型的。对两个操作数都会进行整型提升，返回值的类型是左边操作数提升后的类型。

$i << j$ 的值是将*i*中的位左移*j*位后的结果。每次从*i*的最左端溢出一位，在*i*的最右端补一个0位。 $i >> j$ 的值是将*i*中的位右移*j*位后的结果。如果*i*是无符号数或非负值，则需要在*i*的左端补0。如果*i*是负值，其结果是由实现定义的。一些实现会在左端补0，其他一些实现会保留符号位而补1。

**可移植性技巧** 为了更好地保留可移植性，最好仅对无符号数进行移位运算。

下面的例子展示了对数13应用移位运算的效果：

```
unsigned int i, j;
i = 13; /* i is now 13 (binary 0000000000001101) */
j = i << 2; /* j is now 52 (binary 0000000000110100) */
j = i >> 2; /* j is now 3 (binary 0000000000000011) */
```

如上面的例子所示，两个运算符都不会改变它的操作数。如果要对一个变量进行移位，需要使用复合赋值运算符`<<=`和`>>=`：

```
i = 13; /* i is now 13 (binary 0000000000001101) */
i <=> 2; /* i is now 52 (binary 0000000000110100) */
i >=> 2; /* i is now 13 (binary 0000000000001101) */
```



移位运算符的优先级比算术运算符的优先级低，因此可能产生意料之外的结果。  
例如，`i<<2+1`等同于`i<<(2+1)`，而不是`(i<<2)+1`。

452

## 20.1.2 按位求反运算符、按位与运算符、按位异或运算符和按位或运算符

表20-2列出了余下的按位运算符。

表20-2 其他按位运算符

| 符 号                | 含 义  |
|--------------------|------|
| <code>~</code>     | 按位求反 |
| <code>&amp;</code> | 按位与  |
| <code>^</code>     | 按位异或 |
| <code> </code>     | 按位或  |

运算符`~`是一元运算符，对其操作数会进行整型提升。其他运算符都是二元运算符，对其操作数进行常用的算术转换。

运算符`~`、`&`、`^`和`|`对操作数的每一位执行布尔运算。`~`运算符会产生对操作数求反的结果，即将每一个0替换成1，将每一个1替换成0。`运算符&`对两个操作数相应的位执行逻辑与运算。`运算符^`和`|`相似（都是对两个操作数执行逻辑或运算），差异是当两个操作数的位都是1时，`^`产生0而`|`产生1。



不要将按位运算符`&`和`|`与逻辑运算符`&&`和`||`相混淆。**Q&A**有时候按位运算会得到与逻辑运算相同的结果，但它们绝不等同。

下面的例子说明了运算符`~`、`&`、`^`、`|`的作用：

```
i = 21; /* i is now 21 (binary 0000000000010101) */
j = 56; /* j is now 56 (binary 0000000000111000) */
k = ~i; /* k is now 65514 (binary 111111111101010) */
k = i & j; /* k is now 16 (binary 0000000000010000) */
k = i ^ j; /* k is now 45 (binary 0000000000101101) */
k = i | j; /* k is now 61 (binary 0000000000111101) */
```

其中对`~i`所显示的值是基于`unsigned int`类型的值占有16位的假设。

对运算符`~`需要一些额外的说明，因为它可以帮助我们使低级程序可移植性更好。假设我们需要一个整数，它的所有位都为1。最好的方法是使用`~0`，因为它不会依赖于整数所包含的位的个数。类似地，如果我们需要一个整数，除了最后5位其他的位全都为1，我们可以写成`~0x001f`。

运算符`~`、`&`、`^`和`|`有不同的优先级：

最高级: ~

&

^

453

最低级: |

因此,可以在表达式中组合使用这些运算符,而不必添加括号。例如,可以写`i & ~j | k`而不需要写成`(i & (~j)) | k`,同样,可以写`i ^ j & ~k`而不需要写成`i ^ (j & (~k))`。当然,仍然可以使用括号来避免混淆。



运算符&、^和|的优先级比关系运算符和判等运算符低。因此,下面的语句不会得到期望的结果:

```
if (status & 0x4000 != 0) ...
```

语句会先计算`0x4000 != 0`(结果是1),接着判断`status & 1`是否非0,而不是判断`status & 0x4000`是否非0。

组合赋值运算符`&=`、`^=`和`|=`分别对应于按位运算符`&`、`^`和`|`:

```
i = 21; /* i is now 21 (binary 0000000000010101) */
j = 56; /* j is now 56 (binary 0000000000111000) */
i &= j; /* i is now 16 (binary 0000000000010000) */
i ^= j; /* i is now 40 (binary 0000000000101000) */
i |= j; /* i is now 56 (binary 0000000000111000) */
```

### 20.1.3 用按位运算符访问位

在进行低级编程时,经常会需要将信息存储为单个位或一组位。例如,在编写图形程序时,我们可能会需要将两个或更多的像素挤在一个字节中。通过按位运算,我们可以提取或修改存储在少数几个位中的数据。

假设`i`是一个16位整型变量。以`i`为例,我们来看看如何使用最常用的单个位运算:

- **设置位。**假设我们需要将设置`i`的第4位。(我们假定最高位为第15位,最低位为第0位。)设置第4位的最简单方法是将`i`的值与常量`0x0010`(一个在第4位上为1的“掩码”)进行或运算:

**[惯用法]** `i = 0x0000; /* i is now 0000000000000000 */`  
`i |= 0x0010; /* i is now 000000000000010000 */`

更通用的做法是,如果需要设置的位的位置存储在变量`j`中,可以使用移位运算符来构造掩码:

```
i |= 1 << j; /* set bit j */
```

例如,如果`j`的值为3,`1<<j`是`0x0008`。

454

- **将位清0。**要清除`i`的第4位,可以使用第4位为0、其他位为1的掩码:

**[惯用法]** `i = 0x0fff; /* i is now 0000000011111111 */`  
`i &= ~0x0010; /* i is now 0000000011101111 */`

按照类似的思路,我们可以很容易编写语句来清除一个特定的位,这个位的位置存储在一个变量中:

```
i &= ~(1 << j); /* clears bit j */
```

- **测试位。**下面的`if`语句测试`i`的第4位是否被设置:

**[惯用法]** `if (i & 0x0010) ... /* test bit 4 */`

如果要测试第`j`位是否被设置,可以使用下面的语句:

```
if (i & 1 << j)... /* test bit j */
```

为了使对于位的操作更容易，经常会给它们起名字。例如，如果我们想要使用一个数的第0、1和2位对应于相应的颜色蓝、绿和红。首先，定义名字分别代表三个位：

```
enum {BLUE = 1, GREEN = 2, RED = 4};
```

当然，可以将BLUE、GREEN和RED定义成宏。设置、清除或测试BLUE位可以如下进行：

```
i |= BLUE; /* sets BLUE bit */
i &= ~BLUE; /* clears BLUE bit */
if (i & BLUE)... /* tests BLUE bit */
```

同时设置、清除或测试几个位也一样简单：

```
i |= BLUE | GREEN; /* sets BLUE and GREEN bits */
i &= ~ (BLUE | GREEN); /* clears BLUE and GREEN bits */
if (i & (BLUE | GREEN))... /* tests BLUE and GREEN bits */
```

其中if语句测试BLUE位和GREEN位中是否其中一位被设置了。

#### 20.1.4 用按位运算符访问位域

处理一组连续的位（位域）比处理单个位要复杂一点。下面是两种最常见的位域操作的例子：

- 修改位域。修改位域需要使用按位与（用来清除位域），接着使用按位或（用来将新的位存入位域）。下面的语句显示了如何将二进制的值101存入变量i的第4位—第6位：

```
i = i & ~0x0070 | 0x0050; /* stores 101 in bits 4-6 */
```

运算符&清除了i的第4位至第6位，接着运算符|设置了第6位和第4位。注意，使用*i |= 0x0050*并不总是可行，这只会设置第6位和第4位，但不会改变第5位。为了使上面的例子更通用，我们假设变量j包含了需要存储到i的第4位—第6位的值。我们需要在执行按位或操作之前将j移位至相应的位置：

```
i = (i & ~0x0070) | (j << 4); /* store j in bits 4-6 */
```

运算符<<的优先级比运算符&和|的优先级高，所以可以去掉圆括号：

```
i = i & ~0x0070 | j << 4;
```

455

- 获取位域。当位域处在数的末尾时（在低序号的位时），获得它的值非常方便。例如，下面的语句获取了变量i的第0位—第2位：

```
j = i & 0x0007; /* retrieves bits 0-2 */
```

掩码0x0007在每个需要的位置都为1。如果位域位于i的中间，那首先需要将位域移位至数的最右端，再使用运算符&来获取位域。例如要获取i的第4位—第6位，可以使用下面的语句：

```
j = (i >> 4) & 0x0007; /* retrieves bits 4-6 */
```

#### 20.1.5 程序：XOR 加密

将数据加密的一种最简单的方法就是，将每一个字符与一个密匙进行异或（XOR）运算。假设密匙是一个&字符。（见附录E）如果将它与字符z异或，我们会得到字符\（假定使用ASCII字符集）。具体计算如下：

|          |                 |            |
|----------|-----------------|------------|
| 00100110 | (&的ASCII码)      |            |
| XOR      | <u>01111010</u> | (z的ASCII码) |
|          | 01011100        | (\的ASCII码) |

要将信息解码，只需采用相同的算法。换而言之，只需将加密后的信息再次加密，即可得到原始的信息。例如，如果我们将&字符与\字符异或，就可以得到原来的字符z：

```

00100110 (&的ASCII码)
XOR 01011100 (\的ASCII码)
 01111010 (z的ASCII码)

```

下面的程序xor.c通过将每个字符与&字符进行异或来加密信息。原始信息可以由用户输入，或者使用输入重定向从文件中读入。加密后的信息可以在屏幕上显示，也可以通过输出重定向来存入文件中（>22.1节）。例如，假设文件msg包含下面的内容：

```

Trust not him with your secrets, who, when left
alone in your room, turns over your papers.
Johann Kaspar Lavater (1741-1801)

```

为了将文件msg加密，并将加密后的信息存储在文件newmsg中，需要使用下面的命令：

```
xor <msg> newmsg
```

文件newmsg将包含下面的内容：

```

rTSUR HIR NOK QORN _IST UCETCRU, QNI, QNCH JC@R
GJIHC OH _IST TIIK, RSTHU IPCT_IST VGVCTU.
--IINGHH mGUVT jGPGRCT (1741-1801)

```

要获取原始的信息，需要使用命令

```
xor <newmsg>
```

将原始信息显示在屏幕上。

正如例子中看到的，程序不会改变一些字符，包括数字。将这些字符与&异或会产生不可见的控制字符，这在一些操作系统中会引发错误。在第22章中，我们会看到在读和写包含控制字符的文件时，如何避免问题的发生。而这里，为了安全我们将使用iscntrl函数（>23.4.1节）来检查原始字符或新字符（加密后的字符）是否为控制字符。如果是的话，让程序使用原始的字符，而不用新的字符。

下面是完成后的程序，相当短小：

```

xor.c
/* Performs XOR encryption */

#include <ctype.h>
#include <stdio.h>

#define KEY '&'

main()
{
 int orig_char, new_char;

 while ((orig_char = getchar()) != EOF) {
 new_char = orig_char ^ KEY;
 if (iscntrl(orig_char) || iscntrl(new_char))
 putchar(orig_char);
 else
 putchar(new_char);
 }
 return 0;
}

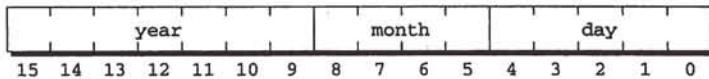
```

## 20.2 结构中的位域

虽然在20.1节中的方法可以操作位域，但这些方法不易使用，而且可能会引起一些误会。幸运的是，C语言提供了另一种选择——声明其成员表示位域的结构。

例如，我们来看看DOS是如何存储在文件创建和最后修改的日期的。由于日期、月和年都

是很小的数，将它们按整数存储会很浪费空间。然而，DOS只使用了16位来存储日期。其中5位用于日、4位用于月、7位用于年：



利用位域，我们可以定义相同形式的C结构：

```
struct file_date {
 unsigned int day: 5;
 unsigned int month: 4;
 unsigned int year: 7;
};
```

在每个成员后面的数指定了它所占用位的长度。由于所有的成员的类型都一样，如果需要，我们可以简化声明：

```
struct file_date {
 unsigned int day: 5, month: 4, year: 7;
};
```

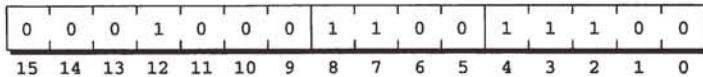
位域的类型必须是int、unsigned int或signed int。使用int会引起二义性，因为一些编译器将位域的最高位作为符号位，而其他一些编译器则不会。

**可移植性技巧** 将所有的位域声明为unsigned int或signed int。

我们可以将位域如同结构的其他成员一样使用。如下面的例子：

```
struct file_date fd;
fd.day = 28;
fd.month = 12;
fd.year = 8; /* represents 1988 */
```

在这些赋值语句之后，变量fd的形式如下图所示。



458

使用按位运算符可以达到同样效果，而且使用按位运算符甚至可能使程序更快些。然而，使程序更易读通常比获得几个微妙更重要一些。

使用位域有一个限制，这个限制对结构的其他成员不起作用。由于通常意义上讲位域没有地址，C语言不允许将&运算符用于位域。由于这条规则，像scanf这样的函数无法直接向位域中存储数据：

```
scanf("%d", &fd.day); /* *** WRONG *** /
```

当然，我们可以用scanf函数将输入读入到一个普通的整型变量中，然后再赋值给fd.day。

### 位域是如何存储的

C语言标准在如何存储位域方面给编译器保留了一定的自由度。我们再仔细看一下编译器是如何处理包含位域成员的结构声明的。

有关编译器处理位域的规则基于“存储单元”的概念。一个存储单元的大小是由实现定义的，通常为8位、16位或32位。当编译器处理结构的声明时，会将位域逐个放入存储单元，位域之间没有间隙，直到剩下的空间不够用来放下一个位域了。这时，一些编译器会跳到下一个存储单元继续存放位域，而另一些则会将位域拆开跨存储单元存放。（具体哪种情况会发生是由实现定义的。）位域存放的顺序（从左至右，还是从右至左）也是由实现定义的。

前面file\_date例子假设是基于16位长的存储单元的（8位的存储单元也可以，只要编译器

将month字段拆开跨两个存储单元存放。) 我们也可以假设位域是从右至左存储的(第一个位域会占据低序号的位), 这是DOS系统上编译器常用的方式。

为了提供对位域存储的更多控制, C语言允许忽略位域的名字。未命名的位域经常用来作为字段间的“填充”, 以保证其他位域存储在适当的位置。例如, 考虑与DOS文件关联的时间以下列方式存储:

```
struct file_time {
 unsigned int seconds: 5;
 unsigned int minutes: 6;
 unsigned int hours: 5;
};
```

(可能你在奇怪怎么可能将一个0~59的数只用5位存储呢。实际上, DOS将秒数除以2, 因此seconds成员实际存储的是0~29的数。) 如果我们并不关心seconds字段, 可以不给它命名:

```
struct file_time {
 unsigned int: 5; /* not used */
 unsigned int minute: 6;
 unsigned int hours: 5;
};
```

其他的位域仍会正常放置, 如同second字段存在时一样。

另一个用来控制位域存储的技巧是指定未命名的字段长度为0:

```
struct s {
 unsigned int a: 4;
 unsigned int : 0; /* 0-length bit-field */
 unsigned int b: 8;
};
```

长度为0的位域是给编译器的一个信号, 告诉编译器将下一个位域放在一个存储单元的起始位置。假设存储单元是8位长的, 编译器会给成员a分配4位, 接着跳过余下的4位到下一个存储单元, 然后给成员b分配8位。如果存储单元是16位, 编译器会给a分配4位, 接着跳过12位, 然后给成员b分配8位。

## 20.3 其他低级技术

前面几章中讲过的一些C语言的特性也同样经常用于编写低级程序。作为本章的结尾, 我们来看几个重要的例子: 定义代表存储单元的类型, 使用联合来回避通常的类型检查, 以及将指针作为地址使用。我们还将介绍18.3节中没有讨论的volatile类型限定符。

### 20.3.1 定义依赖机器的类型

依据定义, char类型占据一个字节, 所以我们有时将字符当作是字节来存储一些并不一定是字符形式的数据。但这样做时, 最好定义一个BYTE类型:

```
typedef unsigned char BYTE;
```

对于不同的机器, 我们还可能需要定义其他类型, 例如:

```
typedef unsigned int WORD;
```

**460** 在稍后的例子中, 我们会使用到BYTE和WORD类型。

### 20.3.2 用联合从多个视角看待数据

虽然在16.4节的例子中已经介绍了有关联合的便捷的使用方式, 但是在C语言中, 联合经常是被用于一个完全不同的目的: 将一块内存看成是两种或更多不同的数据形式。

这里根据20.2节中描述的file\_date结构给出一个简单的例子。由于一个file\_date结构正好放入两个字节中, 我们可以将任何两个字节的数据当作是一个file\_date结构。特别是可

以将一个unsigned int值看作是一个file\_date结构（假设整数是16位长）。下面定义的联合可以使我们方便地将一个整数与文件日期相互转换：

```
union int_date {
 unsigned int i;
 struct file_date fd;
};
```

通过这个联合，我们可以以两个字节的形式获取磁盘中文件的日期，然后得到其中的month、day和year字段的值。相反地，我们也可以按照file\_date结构构造一个日期，然后作为两个字节写入磁盘中。

作为一个使用int\_date联合的例子，下面的函数将整型参数以文件日期的形式显示出来：

```
void print_date(unsigned int n)
{
 union int_date u;

 u.i = n;
 printf("%d/%d/%d\n", u.fd.month, u.fd.day,
 (u.fd.year+1980)%100;
}
```

为了得到年的最后两位，我们将u.fd.year加上1980（因为年是以相对于1980年计算的——根据微软的说法，1980年就是世界的开始），然后计算其结果与100相除的余数。

在使用寄存器时，这种使用联合从多种角度看待数据的方法会非常有用，因为寄存器通常划分为较小的单元。以Intel 80x86处理器为例，它包含16位的寄存器——AX、BX、CX和DX。每一个寄存器都可以看作是两个8位的寄存器。例如AX可以被划分为AH和AL两个寄存器。

当针对使用Intel处理器的计算机编程时，可能会需要用到表示寄存器AX、BX、CX和DX中的值的变量。我们需要对16位寄存器和8位寄存器都进行访问，同时保留它们之间的关系（改变AX的值会同时改变AH和AL的值，改变AH也会同时改变AX）。为了解决这一问题，可以构造两个结构，一个包含对应于16位寄存器的成员，另一个包含对应于8位寄存器的成员。然后构造一个包含这两个结构的联合：

```
union {
 struct {
 WORD ax, bx, cx, dx;
 } word;
 struct {
 BYTE al, ah, bl, bh, cl, ch, dl, dh;
 } byte;
} regs;
```

461

word结构的成员会和byte结构的成员相互重叠。例如，ax会使用与al和ah同样的内存空间。当然，这恰恰就是我们所需要的。下面是一个使用regs联合的例子：

```
regs.byte.ah = 0x12;
regs.byte.al = 0x34;
printf("AX: %x\n", regs.word.ax);
```

对ah和al的改变也会影响ax，其输出是：

AX: 1234

### 20.3.3 将指针作为地址使用

在11.1节中我们已经看到了，指针实际上就是一种内存地址。虽然我们通常不需要知道其细节内容，但是编写低级程序时，这些细节内容就很重要了。

在一些计算机中，地址所包含的位的个数与整型或长整型一致。因此，构造一个指针来表示某个特定的地址是十分方便的：只需要将一个整数强制转换成指针就行。例如，下面的例子将地址1000（十六进制）存入一个指针变量：

```
BYTE *p;
p = (BYTE *) 0x1000; /* p contains address 0x1000 */
```

对于其他一些计算机则比较麻烦。当一台使用Intel CPU的计算机运行在“实时模式”下时(DOS使用的模式)，地址由两个16位数组成：段地址和偏移量。构造一个包含特定地址的指针通常需要调用由非标准头提供的宏。例如，MK\_FP宏(make far pointer)可以根据一对段地址/偏移量来构造指针，这个宏通常可以在<dos.h>中找到：

```
BYTE far *p;
p = MK_FP(segment, offset);
```

其中，far(不属于标准C)指明p是一个“远指针”。换而言之，它由段地址和偏移量构成。(“近指针”只包含偏移量。)

462

### 20.3.4 程序：设置 Num Lock 键

在IBM PC机及其兼容机上，Num Lock切换用来确定数字键盘上的按键是作为数字键使用，还是作为移动光标的方向键使用。用户可以通过按Num Lock键，打开或关闭Num Lock功能。

下面的两个程序nlockon.c和nlockoff.c可以在不按Num Lock键的条件下设置Num Lock状态。这种功能在批处理文件(一种包含一系列DOS命令的文件)中十分有用。例如，我们可以将nlockoff命令放在DOS的autoexec.bat文件中，使Num Lock在机器启动时被关闭。

这些程序其实很容易编写，因为Num Lock的状态就保存在内存中，而且在每一台计算机中保存的地址都一样。在第40(十六进制)地址段、偏移量为17(十六进制)的字节的第5位(右数第6位)用来控制Num Lock状态。设置这一位会打开Num Lock，而清除这一位则关闭Num Lock。nlockon.c和nlockoff.c只需要将这个字节的地址存在一个指针中，然后使用按位运算符修改该字节中的Num Lock位即可。

程序nlockon.c和nlockoff.c是专门针对支持far关键字，并且提供MK\_FP宏的DOS编译器编写的程序。程序nlockon.c使用|=运算符来设置Num Lock位：

```
nlockon.c
/* Turns Num Lock on */

#include <dos.h>

typedef unsigned char BYTE;

main()
{
 BYTE far *p = MK_FP(0x0040, 0x0017);

 p |= 0x20; / sets Num Lock bit */
 return 0;
}
```

nlockoff.c使用&=运算符来清除Num lock位：

```
nlockoff.c
/* Turns Num Lock off */

#include <dos.h>

typedef unsigned char BYTE;

main()
{
 BYTE far *p = MK_FP(0x0040, 0x0017);

 p &= ~0x20; / clears Num Lock bit */
 return 0;
}
```

463

### 20.3.5 volatile 类型限定符

在一些计算机中，一部分内存空间是“易变”的，保存在这种内存空间的数据可能会在程序运行期间发生改变，即使程序自身并未试图存放新值。例如，一些内存空间可能被用于保存直接来自输入设备的数据。

使用 volatile 类型限定符，我们可以通知编译器程序中使用了这类易变的数据。volatile 限定符通常使用在用于指向易变内存空间的指针的声明中：

```
volatile BYTE *p; /* p will point to a volatile byte */
```

为了了解为什么要使用 volatile，我们假设指针 p 指向的内存空间用于存放用户通过键盘输入的最近一个字符。这个内存空间是易变的：每次用户输入一个新字符，这里的值都会发生改变。我们可能使用下面的循环获取键盘输入的字符，并将它们存入一个缓冲数组中：

```
while (缓冲区未满) {
 等待输入;
 buffer[i] = *p;
 if (buffer[i++] == '\n')
 break;
}
```

一个完整的编译器可能会注意到这个循环既没有改变 p，也没有改变 \*p。因此可能会对程序进行优化，使 \*p 只被读取一次：

```
在寄存器中存储*p;
while (缓冲区未满) {
 等待输入;
 buffer[i] = 存储在寄存器中的值;
 if (buffer[i++] == '\n')
 break;
}
```

优化后的程序会不断复制同一个字符来填满缓冲区，这并不是我们想要的程序。将 p 声明成指向不稳定的数据可以避免这一问题的发生，因为 volatile 限定符会通知编译器 \*p 每一次都必须从内存中重新读取。

对于 volatile 的其他用法，请见第 24 章的“问与答”小节。

## 问与答

问：为什么说 & 和 | 运算符产生的结果有时会跟 && 和 || 一样，但又不总是如此呢？(p.315)

答：我们来比较一下 i&j 与 i&&j (对 | 与 || 是类似的)，只要 i 和 j 的值是 0 或 1 (任何组合都可以)，两个表达式的值是一样的。然而，一旦 i 和 j 是其他的值，两个表达式的值不会始终一致。例如，如果 i 的值是 1，j 的值是 2，那么 i&j 的值是 0 (i 和 j 之间没有哪一位同为 1)，而 i&&j 的值是 1。如果 i 的值是 3，而 j 的值是 2，那么 i&j 的值是 2，i&&j 的值则是 1。

464

另一个问题是副作用。计算 i&j++ 总会先增加 j，而计算 i&&j++ 有时会先增加 j。

## 练习

### 20.1 节

\*1. 指出下面每一个代码段的输出。假定 i、j 和 k 都是 unsigned int 类型的变量。

- (a) i = 8; j = 9;  
printf("%d", i >> 1 + j >> 1);
- (b) i = 1;

```

printf("%d", i & ~i);
(c) i = 2; j = 1; k = 0;
 printf("%d", ~i & j ^ k);
(d) i = 7; j = 8; k = 9;
 printf("%d", i ^ j & k);

```

2. 请说出如何“切换”一个位（从0改为1或从1改为0）。通过编写一条语句切换变量i的第4位来说明这种方法。

- \*3. 请解释下面的宏对它的实际参数起什么作用。假设参数具有相同类型。

```
#define M(x,y) ((x)^=(y), (y)^=(x), (x)^=(y))
```

4. 在计算机图形处理中，颜色通常用3个数存储的，分别代表红、绿和蓝3种颜色。假定每个颜色需要8位来存储，而且我们希望将三种颜色一起存放在一个长整型数据中。请编写一个名为MK\_COLOR的宏，包含3个参数（红、绿、蓝的强度）。MK\_COLOR宏需要返回一个long int值，其中后3个字节分别包含红、绿和蓝，红作为最后一个字节。

5. 编写名字为GET\_RED、GET\_GREEN和GET\_BLUE3个宏，并以一个给定的颜色值作为参数（见练习4）。宏会返回一个8位的值表示给定颜色中红、绿、或蓝。

6. (a) 使用按位运算符编写如下函数：

```
unsigned short int swap_byte(unsigned short int i);
```

函数swap\_byte的返回值是将i的两个字节调换后产生的结果。（在大多数计算机中，短整型数据占两个字节。）例如，假设i的值是0x1234（二进制形式为00010010 00110100），那么swap\_byte的返回值应该为0x3412（二进制形式00110100 00010010）。编写一个程序来测试你的函数。程序以十六进制读入数，然后交换两个字节并显示出来：

```
Enter a hexadecimal number: 1234
Number with byte swapped: 3412
```

提示：使用%hx转换来读入和输出十六进制数。

465

- (b) 将swap\_byte函数的函数体化简为一条语句。

7. 编写如下函数：

```
unsigned int rotate_left(unsigned int i, int n);
unsigned int rotate_right(unsigned int i, int n);
```

函数rotate\_left(i,n)的值应是将i左移n位并将从左侧移出的位移入i右端而产生的结果。（例如，假定整型占16位，rotate\_left(0x1234, 4)将返回0x2341。）类似地，函数rotate\_right也类似，只是将数字中的位向右循环移位。

8. 假定函数f的实现如下：

```
unsigned int f(unsigned int i, int m, int n)
{
 return (i >> (m+1-n) & ~(~0 << n));
}
```

- (a) ~(~0 << n)的结果是什么？

- (b) 函数f的作用是什么？

## 20.2节

9. 当按照IEEE浮点标准存储浮点数时，一个float类型的值由1个符号位（最左边的位或最重要的位），8个指数位以及23个小数位依次组成。请设计一个32位的结构类型，包含与符号位、指数位和小数位相对应的位域成员。声明的位域类型为unsigned int。请参考你的用户手册来决定位域的顺序。警告：一些编译器会限制位域在16位以内，因此当你编译这个结构时可能会有出错信息。

## 20.3节

10. 请设计一个联合类型，使一个32位的值既可以看作是一个float型的值，也可以看作是练习9中定义的结构。写一个程序将1存储在结构的符号位，将128存储在指数位，0存储在小数位。然后按float值的形式显示存储在联合中的值。（如果你的位域设置正确的话，结果应该是-2.0。）

466

# 标 准 库

每个程序都是某些其他程序的一部分，但很少是正合适的。

前面几章中零碎地介绍了一些C语言标准库的相关知识。在本章中，我们将完整地讨论标准库。在21.1节中，会列举使用库的一些通用的指导原则，还会介绍在一些库的头中发现的技巧：使用宏来“隐藏”函数。21.2节会对标准库的15个头分别做概述性的介绍。

在随后几章中，将深入讨论标准库的头，并将相关联的头放在一起讨论。其中`<stddef.h>`明显不同于其他的，因此会在21.3节中介绍。

## 21.1 标准库的使用

C语言的标准库总共划分成15个部分，每个部分用一个头描述。许多编译器都会使用扩展后的库，因此包含的头通常会多于15个。额外添加的头当然不属于标准库的范畴，所以我们不能假设其他的编译器也可以支持这些头。这类头通常提供一些针对特定机型或特定操作系统的函数（这也解释了为什么它们不属于标准库），它们可能会提供对屏幕或键盘更多的操作的函数。用于支持图形或窗口界面的头也是很常见的。

标准头主要由函数原型、类型定义、以及宏定义组成。如果我们的文件中调用了头中的函数，或是使用了头中定义的类型或宏，那么我们就需要在文件开头将相应的头包含进来。当一个文件包含了多个头时，`#include`指令的顺序无关紧要。

467

### 21.1.1 对标准库中使用的名字一些限制

任何包含了标准头的文件都必须遵守两条规则。第一，该文件不能以任何目的再使用在头文件中定义过的宏的名字。例如，如果文件包含了`<stdio.h>`，就不能再使用`NULL`了，因为使用这个名字的宏已经在`<stdio.h>`中定义过了；第二，具有文件作用域的库名（尤其是类型名）也不可以在文件层次重定义。因此，一旦文件包含了`<stdio.h>`，由于`<stdio.h>`中已经将`size_t`定义为类型名，那么就不允许在文件作用域内将`size_t`重定义为任何标识符。

虽然上述这些限制似乎是显而易见的，但C语言还有一些其他的限制，可能是你想不到的：

- 由一个下划线和一个大写字母开头或由两个下划线开头的标识符，属于标准库中保留的标识符。程序不允许因任何目的使用这种形式的标识符。
- 由一个下划线开头的标识符被保留，用于文件作用域内的标识符和标记。除非仅声明在函数内部，否则不应该使用这类标识符。
- 在标准库中所有外部链接的标识符被保留，用于作为需要外部链接的标识符。特别是所有标准库函数的名字都被保留。因此，即使文件不需要包含`<stdio.h>`，也不应该声明一个外部函数叫`printf`，因为在标准库中已经有一个同名的函数了。

这些规则对程序的所有文件都起作用，不论文件包含了哪个头。虽然这些规则并不总是强制性的，但不遵守这些规则可能会导致程序的可移植性下降。

### 21.1.2 使用宏隐藏函数

C程序员经常会用宏来替代小的函数，这在标准库中同样很常见。C语言标准允许在头中定

义与库函数同名的宏，为了起到保护作用，还要求有实际的函数存在。因此，对于库的头，声明一个函数并同时定义一个有相同名字的宏的情况并不少见。

在`<ctype.h>`(>23.4节)中有大量这样成对的函数或宏定义的例子，例如用来检测一个其字符是否可以显示的函数`isprint`。通常的做法是在`<ctype.h>`中定义`isprint`作为一个函数：

```
int isprint(int c);
```

并同时把它定义为一个宏：

```
#define isprint(c) ((c) >= 0x20 && (c) <= 0x7e)
```

**468** 在默认情况下，对`isprint`的调用会被作为宏调用（因为宏名会在预处理时被替换）。

在大多数情况下，我们对于使用宏来替代实际的函数还是满意的，因为这样可能会提高程序的运行速度。然而在某些情况下，我们可能需要的是一个真实的函数。这可能是由于需要尽量缩小可执行代码的大小，也可能是因为需要一个指向这个函数的指针(>17.7节)。

如果确实存在这种需求，我们可以使用`#undef`指令(>14.3.5节)来删除宏定义，从而可以访问到真实的函数。例如，下面的代码通过取消了`isprint`的宏定义来使用`isprint`函数：

```
#include <ctype.h>
#undef isprint
```

即使`isprint`不是宏，这样的做法也不会带来任何负面影响，因为当所提供的名字没有被定义成宏时，`#undef`指令不会起任何作用。

此外，我们也可以通过给名字加圆括号来屏蔽个别宏调用：

```
(isprint) (c)
```

预处理器无法分辨出带圆括号的宏，除非宏名后跟着一个左圆括号；而编译器则不会这么容易被欺骗，它仍可以认出`isprint`函数。

## 21.2 标准库概述

现在简单讨论一下标准库中的15个头。本节可以帮助你分辨出你所需要的是C标准库的哪部分。在本章及随后几章中会对每个头有更详细的介绍。如果需要了解某个库函数的具体信息，请参考附录D。

### 1. `<assert.h>`: 调试

`<assert.h>`(>24.1节)仅包含`assert`宏。我们可以在程序中插入该宏，从而检查程序状态。一旦任何检查失败，程序会被终止。

### 2. `<ctype.h>`: 字符处理

`<ctype.h>`(>23.4节)包括用于字符分类及大小写转换的函数。

### 3. `<errno.h>`: 错误

`<errno.h>`(>24.2节)提供了`errno` (“error number”)。`errno`是一个左值(lvalue)，可以在调用特定库函数后进行检测，来判断调用过程中是否有错误发生。

### 4. `<float.h>`: 浮点型的特性

`<float.h>`(>23.1节)提供了用于描述浮点类型特性的宏，包括值的范围及精度。

### 5. `<limits.h>`: 整型的大小

`<limits.h>`(>23.2节)提供了用于描述整数类型和字符类型特性的宏，包括它们的最大值和最小值。

### 6. `<locale.h>`: 本地化

`<locale.h>`(>25.1节)提供一些函数来帮助程序适应针对一个国家或地区的特定行为方式。这些与本地化相关的功能包括数显示的方式(包括用于小数点的字符)、货币的格式(例

如货币符号)、字符集以及日期和时间的表示形式。

#### 7. <math.h>: 数学计算

<math.h> (>23.3节) 提供了大量用于数学计算的函数, 包括三角函数, 双曲函数、指数函数、对数函数、幂函数、相近取整(四舍五入)函数及绝对值运算函数。其中大部分函数使用double类型的实际参数, 并返回一个double类型的值。

#### 8. <setjmp.h>: 非本地跳转

<setjmp.h> (>24.4节) 提供了setjmp函数和longjmp函数。setjmp函数会“标记”程序中的一个位置, 随后可以用longjmp返回被标记的位置。这些函数可以用来从一个函数跳转到另一个(仍然活动中的)函数中, 绕过正常的函数返回机制。setjmp函数和longjmp函数主要用来处理程序执行过程中的重大问题。

#### 9. <signal.h>: 信号处理

<signal.h> (>24.3节) 提供了用于异常情况(信号)处理的函数, 包括中断和运行时错误。signal函数可以设置一个函数, 使系统会在给定信号发生后自动调用该函数; raise函数用来产生一个信号。

#### 10. <stdarg.h>: 可变实际参数

<stdarg.h> (>26.1节) 提供给函数可以处理不定个数个参数的工具, 就像printf和scanf函数。

470

#### 11. <stddef.h>: 常用定义

<stddef.h> (>21.3节) 提供了经常使用的类型和宏的定义。

#### 12. <stdio.h>: 输入/输出

<stdio.h> (22.1节至22.8节) 提供了大量用于输入/输出的函数。包括对顺序读写和随机读写文件的操作。

#### 13. <stdlib.h>: 常用实用程序

<stdlib.h> (>26.2节) 包含了大量无法划归于其他头的函数。包含在<stdlib.h>中的函数可以将字符串转换成数、产生伪随机值、执行内存管理任务、与操作系统通信、执行搜索与排序以及对多字节字符及字符串进行操作。

#### 14. <string.h>: 字符串处理

<string.h> (>23.5节) 提供了用于进行字符串操作的函数, 包括复制、拼接、比较及搜索。

#### 15. <time.h>: 日期和时间

<time.h> (>26.3节) 提供相应的函数来获取日期和时间、操纵时间和以多种方式显示时间等。

471

## 21.3 <stddef.h>: 常用定义

<stddef.h> 提供了常用的类型和宏的定义, 但没有声明任何函数。定义的类型包括:

- ptrdiff\_t。当进行指针相减运算时, 其结果的类型。
- size\_t。运算符sizeof的返回值类型。
- wchar\_t。一种足够大的、可以用于表示所有支持的地区的所有字符的类型。

所有这3种类型都是整数类型。其中ptrdiff\_t必须是带符号的类型, 而size\_t则必须是无符号的类型。关于wchar\_t得更多细节, 见25.2节。

<stddef.h> 中还定义了两个宏。一个是NULL, 用来表示空指针。另一个宏offsetof需要两个参数: 类型(一个结构类型)和指定成员(结构的一个成员)。offsetof宏会计算结构的起点到指定成员间的字节数。

考虑下面的结构：

```
struct s {
 char a;
 int b[2];
 float c;
};
```

`offsetof(struct s, a)` 的值一定是0，C语言确保结构的第一个成员的地址与结构自身地址相同。我们无法确定地说出b和c的偏移量是多少。一种可能是`offsetof(struct s, b)`是1（因为a的长度是一个字节），`offsetof(struct s, c)`是5（假设整数是16位）。然而，一些编译器会在结构中留下一些空洞（无效字节）（见在第16章结尾的“问与答”小节），从而会影响到`offsetof`产生的值。例如，对于在a后面留下一个无效字节的编译器，b和c的偏移量相应会是2和6。这就是`offsetof`宏的优点：对任意编译器，它都会返回正确的偏移量，使我们可以编写移植性更好的程序。

`offsetof`有很多用途。例如，假如我们需要将结构s的前两个成员写入文件，但忽略成员c。我们无法使用`fwrite`函数（>22.6节）来写`sizeof(struct s)`个字节，因为这样会将整个结构写入文件。然而，我们可以只写`offsetof(struct s, c)`个字节。

最后一点：一些在`<stddef.h>`中定义的类型和宏在其他头中也会出现。（例如，`NULL`宏在`<locale.h>`、`<stdio.h>`、`<stdlib.h>`、`<string.h>`和`<time.h>`中也有定义。）因此，只有少数程序真的需要包含`<stddef.h>`。

## 问与答

问：我注意到书中使用“标准头”，而不是“标准头文件”。不使用“文件”有什么具体原因吗？

答：是的。依据C标准，一个“标准头”不需要一定是文件。虽然绝大部分编译器确实将标准头以文件形式存储，但标准实际上允许将标准头直接内置在编译器自身中。

472

## 练习

### 21.1节

1. 在你的系统中找到存放头文件的位置。找出那些非标准头，并指明每一个的用途。
2. 在存放头文件的目录中（见练习1），找到一个使用宏来隐藏函数的标准头。
3. 当使用宏隐藏函数时，在头文件中哪一个必须放在前面：宏定义还是函数原型？验证你的结论。

### 21.2节

4. 你期望在哪个标准头中可以分别找到下面描述的函数或宏？
  - (a) 可以得到当前是星期几的函数。
  - (b) 判断一个字符是否是数字的函数。
  - (c) 给出最大的`unsigned int`的值的宏定义。
  - (d) 对一个浮点数，得到比它大的最近的整数的函数。
  - (e) 指定一个字符包含多少位的宏。
  - (f) 指定在一个`double`类型的值中，有效位个数的宏。
  - (g) 在字符串中查找特定字符的函数。
  - (h) 用来以读的方式打开一个文件的函数。

### 21.3节

5. 编写一个程序，声明书中的结构s，并显示出成员a、b和c的大小和偏移量。（使用`sizeof`来得到大小，使用`offsetof`来得到偏移量。）同时使程序显示出整个结构的大小。根据这些信息，判断结构中是否包含空洞（无效字节）。如果包含，指出每一个的位置和大小。

473

# 输入/输出

在人机共生的世界中，必须调整的是人：机器是无法调整的。

C语言的输入/输出库是用`<stdio.h>`进行表示的。它是标准库中最大且最重要的部分。由于输入/输出是C语言的高级应用，因此这里将用一整章（在本书中，本章也许不是最重要的一章，但却是最大的一章）来讨论`<stdio.h>`。

从第2章开始，我们已经在使用`<stdio.h>`了，而且已经对`printf`函数、`scanf`函数、`putchar`函数、`getchar`函数、`puts`函数以及`gets`函数的使用有了一定的了解。本章会提供更多有关上述这些函数的信息，还会介绍一些新的用于文件处理的函数。而且值得高兴的是许多新函数和我们已经熟知的函数有着紧密的联系。例如，`fprintf`函数就是`printf`函数的“文件版”。

本章的开始将会讨论一些基本问题：流（stream）的概念、`FILE`类型、输入和输出重定向以及文本文件和二进制文件的差异（22.1节）。随后将转入讨论特别为文件使用而设计的函数，其中包括有打开和关闭文件的函数（22.2节）。

在讨论完`printf`函数、`scanf`函数以及与“格式化”输入/输出相关的函数以后（22.3节），将会看到读/写非格式化数据的函数：

- `getc`函数、`putc`函数以及相关的函数，它们每次读写一个字符（22.4节）。
- `gets`函数、`puts`函数以及相关的函数，它们每次读写一行字符（22.5节）。
- 读/写数据块的`fread`函数和`fwrite`函数（22.5节）。

然后，22.7节会说明如何在文件上执行随机的访问操作。最后，22.8节会描述`sprintf`函数和`sscanf`函数，这两个函数不同于读和写一个字符串的`printf`函数和`scanf`函数。

475

本章涵盖了`<stdio.h>`中的绝大部分函数，只是忽略了其中4个函数。它们分别是`perror`函数（>24.2节）、`vfprintf`函数、`vprintf`函数和`vsprintf`函数（>26.1.2节）。这些函数和C语言库中的其他内容关系紧密。

## 22.1 流

在C语言中，术语流意味着任意输入的源或任意输出的目的地。许多小型程序，就像前面介绍的那些，它们都是通过一个流（通常和键盘相关）获得全部的输入，并且通过另一个流（通常和屏幕相关）写出全部的输出。

而较大规模的程序可能会需要额外的流。这些流常常表示为磁盘上的文件，但却可以和其他类型的设备相关联：调制解调器、网络端口、打印机、光盘驱动器等。这里将集中讨论磁盘上的文件，因为这类文件通用且容易理解。（在应该说流的时候，这里可能偶尔会使用术语文件。）但是，请千万记住一点，`<stdio.h>`中的许多函数不仅可以处理表示成文件的流，还可以处理所有其他形式的流。

### 22.1.1 文件指针

C程序中流的访问是通过文件指针（file pointer）实现的。此指针拥有的类型为FILE \*（在<stdio.h>中定义了FILE的类型）。用文件指针表示的特定流具有标准化的名字。如果需要，则可以声明一些额外的文件指针。例如，除了标准流，如果程序还需要两个流，那么可以在程序中包含下列形式的声明：

```
FILE *fp1, *fp2;
```

虽然操作系统通常会限制在任意某时刻可以打开的流的数量，但是一个程序可以声明任意数量的FILE \*型变量。

### 22.1.2 标准流和重定向

**Q&A**<stdio.h>提供了3种标准流（表22-1）。这3个标准流是备用的，也就是说不能声明它们，也无法打开或关闭它们。

表22-1 标准流

| 文件指针   | 流    | 默认的含义 |
|--------|------|-------|
| stdin  | 标准输入 | 键盘    |
| stdout | 标准输出 | 屏幕    |
| stderr | 标准错误 | 屏幕    |

前面使用过的printf、scanf、putchar、getchar、puts和gets这些函数都是通过476 stdin获得输入，并且用stdout进行输出。默认情况下，stdin表示键盘，而stdout和stderr则表示屏幕。然而，某些操作系统允许通过所谓的重定向（redirection）机制来改变这些默认的含义。

例如，在UNIX和DOS操作系统中可以迫使程序从文件中而不是键盘上获得输入。方法是在命令行中在字符<的后边放上文件的名字：

```
demo <in.dat
```

这种方法被称为是输入重定向（input redirection），它本质上是使stdin流表示为文件（此例中为文件in.dat）而非键盘。重定向的绝妙之处在于demo程序不会意识到正在从文件in.dat中读取数据，而只是知道从stdin获得的任何数据是从键盘上录入的。

输出重定向（output redirection）和此很类似。在UNIX和DOS系统中对stdout流进行重定向是通过在命令行中在字符>的后边放置文件名实现的：

```
demo >out.dat
```

现在所有写入stdout的数据将进入out.dat文件中，而不是出现在屏幕上。顺便说一下，我们还可以把输出重定向和输入重定向进行合并：

```
demo <in.dat >out.dat
```

输出重定向的一个问题是会把写给stdout的每样内容都放入到文件中。如果程序运行失常并且开始发出错误信息，那么我们只能在看到文件的那一刻才会知道。而这些应该是出现在stderr中的。把错误信息写到stderr而不是stdout中，这样做可以保证在stdout发生重定向时这些错误信息将仍会出现在屏幕上。

### 22.1.3 文本文件与二进制文件

<stdio.h>支持两种类型的文件：文本文件和二进制文件。在文本文件（text file）中，字节表示字符，这使人们可以检查或编辑文件。例如，C程序的源代码是存储在文本文件中的。

另一方面，在二进制文件（binary file）中，字节不一定就表示字符，字节组还可以表示其他类型的数据，比如整数和浮点数。如果看看某个二进制文件，就会立刻意识到可执行的C程序是存储在二进制文件中的。

为了明白文本文件和二进制文件之间的区别，可以思考一下在文件中存储数32 767的方法。一种选择是以文本的形式把3、2、7、6、7作为字符存储起来。假设字符集为ASCII，那么就可以得到下列5个字节：

|          |          |          |          |          |
|----------|----------|----------|----------|----------|
| 00110011 | 00110010 | 00110111 | 00110110 | 00110111 |
| '3'      | '2'      | '7'      | '6'      | '7'      |

477

另一种选择是以二进制的形式存储此数，这种方法只会占用两个字节：

|          |          |
|----------|----------|
| 01111111 | 11111111 |
|----------|----------|

就像上述示例显示的那样，用二进制的形式存储数可以节省相当大的空间。

为什么一定要区分文本文件和二进制文件呢？毕竟，无论用哪种形式，一个文件就是一个字节的序列。文本文件按行进行划分，所以必须用一些方法来标记每行的末尾，比如采用特殊的字符。而且，操作系统还可能用特殊的字符来说明文本文件的结束。另一方面，二进制文件不是按行进行划分的，而且由于二进制文件可以合法地包含任何字符，所以不可能留出文件结束字符。

在DOS操作系统中，文本文件和二进制文件之间存在两方面的差异：

- **行的结尾。**当文本文件中写入换行符时，此换行符会扩展成一对字符，即回行符和跟随的回车符。与之对应的转换发生在输入过程中。然而，把换行符写入二进制文件时，它就是单独一个字符（换行符）。
- **文件末尾。**在文本文件中把字符Ctrl+Z（\x1a）设定为文件的结束标记。（不一定要在文本文件末尾有字符Ctrl+Z，但是某些编辑器会把它放上。）而在二进制文件中字符Ctrl+Z没有特别的含义，处理它就像其他任何字符一样。

与此相反的，UNIX操作系统对文本文件和二进制文件不进行区分。两者会以相同的方式进行存储。一个UNIX文本文件在每行的结尾只有单独一个换行符，而且没有特殊字符用来标记文件末尾。

当编写用来读或写文件的程序时，需要考虑是文本文件还是二进制文件。要在屏幕上显示文件的内容，程序就可能要把文件设定为文本文件。而另一方面，一个文件复制程序就不能把要复制的文件设定为文本文件。如果那样做，那么就不能完全复制含有文件结束字符的二进制文件了。在无法确定文件是文本形式还是二进制形式时，安全的做法是把文件设定为二进制文件。

## 22.2 文件操作

简单正是输入和输出重定向吸引人的地方之一。它不需要打开文件、关闭文件或者执行任何其他明确的文件操作。可惜的是，重定向在许多应用中受到限制。当程序依赖重定向时，它无法控制自己的文件，甚至无法知道这些文件的名字。更糟糕的是，如果程序需要在同一时间读入两个文件或者写出两个文件，重定向都无法做到。

478

当重定向无法满足需要时，将终止使用<stdio.h>提供的文件操作。本节将探讨这些文件操作，包括打开文件、关闭文件、改变缓冲文件的方式、删除文件以及重命名文件。

### 22.2.1 打开文件

```
FILE *fopen(const char *filename, const char *mode);
```

用流的方式打开文件，这要求调用fopen函数。fopen函数的第一个实际参数是含有要打开文件名的字符串。（“文件名”可能包含关于文件位置的信息，例如驱动器号或路径。）第二个实际参数是“模式字符串”，它用来说明打算对文件执行的操作内容。例如，字符串“r”说明将从文件读入数据，但是不能写数据。



**DOS程序员：**在fopen函数调用的文件名中含有字符\时，请一定小心。因为C语言会把字符\看成是转义序列（>7.3.1节）的开始标志。

```
fopen("c:\project\test1.dat", "r");
```

这个调用将始终无法执行，因为编译器会把\t看成是转义字符（\p的含义是未定义的）。为了避免这类问题，可以用\\代替\：

```
fopen("c:\\project\\\\test1.dat", "r");
```

fopen函数返回一个文件指针。程序可以（且通常将）把此指针存储在一个变量中，稍后在需要对文件进行操作时使用它。fopen函数的典型调用形式如下所示：

```
fp = fopen("in.dat", "r"); /* opens in.dat for reading */
```

当程序为了稍后从文件in.dat中读数据而调用输入函数时，将会把fp作为一个实际参数。

当无法打开文件时，fopen函数会返回空指针。也许文件不存在，或者文件在错误的地方，再或者是未获得打开文件的许可。



**永远不能假设可以打开文件。**为了确保不会返回空指针，需要始终测试fopen函数的返回值。

### 22.2.2 模式

打算传递给fopen函数的模式字符串的内容不仅依赖于稍后将要对文件采取的操作内容，还取决于文件上的数据是文本形式还是二进制形式。为了打开一个文本文件，可以采用表22-2中的一种模式字符串。

479

表22-2 用于文本文件的模式字符串

| 字符串  | 含义                   |
|------|----------------------|
| "r"  | 打开文件用于读              |
| "w"  | 打开文件用于写（文件不需要存在）     |
| "a"  | 打开文件用于追加（文件不需要存在）    |
| "r+" | 打开文件用于读和写，从文件头开始     |
| "w+" | 打开文件用于读和写（如果文件存在就截去） |
| "a+" | 打开文件用于读和写（如果文件存在就追加） |

**Q&A**当使用fopen打开二进制文件时，会需要在模式字符串中包含字母b。表22-3列出了用于二进制文件的模式字符串。

表22-3 用于二进制文件的模式字符串

| 字 符 串        | 含 义                  |
|--------------|----------------------|
| "rb"         | 打开文件用于读              |
| "wb"         | 打开文件用于写（文件不需要存在）     |
| "ab"         | 打开文件用于追加（文件不需要存在）    |
| "r+b"或者"rb+" | 打开文件用于读和写，从文件头开始     |
| "w+b"或者"wb+" | 打开文件用于读和写（如果文件存在就截去） |
| "a+b"或者"ab+" | 打开文件用于读和写（如果文件存在就追加） |

从表22-2和表22-3可以看出<stdio.h>对写数据和追加数据进行了区分。当给文件写数据时，通常会对先前的内容覆盖写。然而，当为追加打开文件时，试图给文件写数据实际是在文件末尾进行添加，因而在会留存文件的原始内容。

顺便说一下，应用于既读又写一个已打开的文件时的特殊规则（模式字符串包含字符+）。如果不是先调用一个文件定位函数（见22.7节），那么就不能把读转换成写。而且，如果既没有调用fflush函数（本节稍后会有介绍）也没有调用文件定位函数，那么就不能把写转化为读。

### 22.2.3 关闭文件

```
int fclose(FILE *stream);
```

fclose函数允许程序关闭不再使用的文件。fclose函数的实际参数必须是文件指针，此指针来自fopen函数或freopen函数（本节稍后会有介绍）的调用。如果成功关闭了文件，那么fclose函数会返回零。否则，它将会返回错误代码EOF（在<stdio.h>中定义的宏）。

为了说明如何在实际中使用fopen函数和fclose函数，这里给出了一个程序的框架。此程序打开文件example.dat进行读数据操作，并要检查打开是否成功，然后在程序终止前再把文件关闭：

```
#include <stdio.h>
#include <stdlib.h>

#define FILE_NAME "example.dat"
main()
{
 FILE *fp

 fp = fopen(FILE_NAME, "r")
 if (fp == NULL) {
 printf("Can't open %s\n", FILE_NAME);
 exit(EXIT_FAILURE);
 }
 ...
 fclose(fp);
 return 0
}
```

480

当然，按照C程序员的编写习惯，通常不会把fopen函数的调用和fp的声明组合在一起使用：

```
FILE *fp = fopen(FILE_NAME, "r");
```

或判定是否为NULL：

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) ...
```

### 22.2.4 为流附加文件

```
FILE *fopen(const char *filename, const char *mode);
```

`freopen`函数为已经打开的流附加一个不同的文件。`freopen`函数最常见的应用是把文件和其中一个标准流相关联，这些标准流包括：`stdin`、`stdout`或`stderr`。例如，为了使程序开始往文件`foo`中写数据，可以使用下列形式的`freopen`函数调用：

```
if (fopen("foo", "w", stdout) == NULL) {
 /* error; foo can't be opened */
}
```

在关闭了任何先前与`stdout`相关联的文件之后（通过命令行重定向或者前一个`freopen`函数调用），`freopen`函数将打开文件`foo`，并且使此文件和`stdout`相关联。

`freopen`函数通常返回的值是它的第三个实际参数（一个文件指针）。如果无法打开新的文件，那么`freopen`函数会返回空指针。（如果无法关闭旧的文件，那么`freopen`函数会忽略掉错误。）

## 22.2.5 从命令行获取文件名

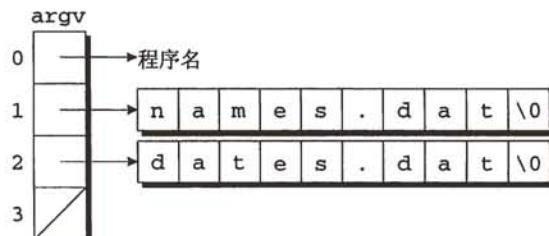
当正在编写的程序需要打开文件时，一个问题立刻变得清晰起来：如何为程序提供文件名呢？把文件名嵌入程序自身的做法不会提供更多的灵活性，而且提示用户输入文件名可能也是笨拙的做法。最好的解决方案常常是通过用户在程序运行时录入的命令行来获取文件的名字。例如，当执行命名为`demo`的程序时，可以通过把文件名放入命令行的方法为程序提供文件名：

```
demo name.dat dates.dat
```

在13.7节中，看到了通过定义带有两个形式参数`main`函数的方法访问到命令行实际参数的过程：

```
main(int argc, char *argv[])
{
 ...
}
```

`argc`是命令行实际参数的数量，而`argv`是一个指针数组，数组中的指针都指向实际参数字符串。`argv[0]`指向程序的名字，从`argv[1]`到`argv[argc-1]`都指向剩余的实际参数，而`argv[argc]`是空指针。在上述例子中，`argc`是3，`argv[0]`指向含有程序名的字符串，`argv[1]`指向字符串`"name.dat"`，而`argv[2]`则指向字符串`"dates.dat"`：



## 22.2.6 程序：检查文件是否可以打开

下面的程序用来确定，若文件存在就可以打开进行读入。在运行程序时，用户将给出要检测的文件的名字：

```
canopen f1.dat
```

然后程序将显示出`f1.dat can be opened`或者显示出`f1.dat can't be opened`。如果在命令行中录入了错误的实数参数数量，那么程序将显示出信息`usage: canopen filename`来提醒用户`canopen`需要单独一个文件名。

**canopen.c**

```
/* Checks whether a file can be opened for reading */
```

```
#include <stdio.h>
main(int argc, char *argv[])
{
 FILE *fp
 if (argc != 2) {
 printf("usage: canopen filename\n")
 Return 2
 }
 if ((fp = fopen(argv[1], "r")) == NULL) {
 printf("%s can't be opened\n", argv[1]);
 return 1;
 }
 printf("%s can be opened\n", argv[1]);
 fclose(fp);
 return 0;
}
```

482

注意，为了丢弃canopen的输出，也为了简单地测试返回的状态值（如果可以打开文件，则值为0，否则为1），可以使用重定向。

## 22.2.7 临时文件

```
FILE *tmpfile(void);
char *tmpnam(char *s);
```

现实世界中的程序经常需要产生临时文件，即只在程序运行时存在的文件。例如，C语言编译器就常常产生临时文件。编译器可能先把C程序翻译成一些存储在文件中的中间形式。然后，在稍后把程序翻译成目标代码时编译器会读取这些文件。一旦完全编译通过了程序，就不再需要保留那些含有程序中间形式的文件了。`<stdio.h>`提供了两个函数用来处理临时文件，即`tmpfile`函数和`tmpnam`函数。

`tmpfile`函数产生临时文件，这些临时文件将存到文件关闭时或程序终止时。`tmpfile`函数的调用会返回文件指针，此指针可以用于稍后访问文件：

```
FILE *temppt;
Temppt = tmpfile(); /* creates a temporary file */
```

如果产生文件失败，`tmpfile`函数就会返回空指针。

虽然`tmpfile`函数很易于使用，但是它还是有两个缺点：(1) 无法知道`tmpfile`函数产生的文件名是什么；(2) 无法决定稍后是否要使文件成为永久性的。如果这些限制生成了问题，可以替换的解决方案就是用`fopen`函数产生临时文件。显然不会希望此文件拥有和前面已经存在的文件相同的名字，所以就需要一些方法产生新的文件名。这也就是`tmpnam`函数出现的原因。

`tmpnam`函数为临时文件产生名字。如果它的实际参数是空指针，那么`tmpnam`函数会把文件名存储到静态变量中，并且返回指向此变量的指针：

```
char *filename;
filename = tmpnam(NULL); /* creates a temporary file name */
```

483

否则的话，`tmpnam`函数会把文件名复制到程序员提供的字符数组中：

```
char filename[L_tmpnam];
tmpnam(filename); /* creates a temporary file name */
```

在后一种情况下，`tmpnam`函数也会返回指向临时文件名的指针。`L_tmpnam`在`<stdio.h>`中是一个宏，它说明了保存临时文件名的字符数组有多长。



当传递指向tmpnam函数的指针时，一定要确保指针指向至少有L\_tmpnam个字符的数组。而且，还要当心不能过于频繁地调用tmpnam函数。宏TMP\_MAX（在<stdio.h>中定义的）说明程序执行期间由tmpnam函数产生的临时文件名的最大数量。

## 22.2.8 文件缓冲



从磁盘驱动器传出或者传入信息是相对较慢的操作。这样的结果是每次程序想读或写字符时无法直接访问磁盘文件来进行。获得有效性能的诀窍就是缓冲（buffering）：写入流的数据实际是存储在内存的缓冲区域内。当缓冲区满了（或者关闭流）时，缓冲区会“清洗”（写入实际的输出设备）。输入流可以用类似的方法进行缓冲：缓冲区包含来自输入设备的数据。从缓冲区读数据代替了从设备本身读数据。缓冲在效率上可以取得巨大的收益，因为从缓冲区读字符或者在缓冲区内存储字符几乎不花什么时间。当然，需要花时间把缓冲区的内容传递给磁盘，或者从磁盘传递给缓冲区，但是一个大的“块移动”比任何微小的字符移动要快很多。

当看似有用时，<stdio.h>中的函数会自动完成缓冲操作。缓冲发生在屏幕的后台，而且通常不用担心它的操作。然而，极少的情况下可能需要我们承担更主动的作用。如果真是如此，可以使用fflush函数、setbuf函数和setvbuf函数。

**484** 当程序向文件中写输出时，数据通常是放在缓冲区中而不是文件内。当缓冲区满了或者关闭文件时，缓冲区会自动清洗。然而，通过调用fflush函数，程序可以像希望的那样频繁地清洗文件的缓冲区。调用

```
fflush(fp); /* flushes buffer for fp */
```

为和文件相关联的fp清洗了缓冲区。调用

```
fflush(NULL); /* flushes all buffers */
```

清洗了全部输出流。如果调用成功，那么fflush函数会返回零，而如果发生错误，则返回EOF。

setvbuf函数允许改变缓冲流的方法，并且允许控制缓冲区的大小和位置。函数的第三个实际参数说明了期望缓冲区的类型：

- \_IOFBF（满缓冲）。当缓冲区为空时，从流读入数据。或者当缓冲区满时，向流写入数据。
- \_IOLBF（行缓冲）。每次从流读入一行数据或者向流写入一行数据。
- \_IONBF（无缓冲）。直接从流读入数据或者直接向流写入数据，而没有缓冲区。

（所有这3种宏都在<stdio.h>中进行了定义。）

setvbuf函数的第二个实际参数（如果它不是空指针的话）是期望缓冲区的地址。缓冲区可以有静态存储期限、自动存储期限或是可以动态分配的。使缓冲区自动化将允许它的空间在块退出时可以被自动的重声明。动态分配缓冲区使在不需要时可以释放缓冲区。setvbuf函数的最后一个实际参数是缓冲区内字节的数量。较大的缓冲区可以提供更好的性能，而较小的缓冲区可以节约空间。

例如，下列这个setvbuf函数的调用利用buffer数组中的N个字节作为缓冲区，而把stringstream的缓冲变成了满缓冲：

```
char buffer[N];
setvbuf(stream, buffer, _IOFBF, N);
```



必须在打开stream之后，而且执行任何其他在stream上的操作之前调用setvbuf函数。

如果调用成功，setvbuf函数返回零。如果要求的缓冲模式是无效的或者无法提供，那么setvbuf函数会返回非零值。

setbuf函数是一个较早期的函数，它用来设定缓冲模式的默认值和缓冲区的大小。如果buf是空指针，那么setbuf(stream, buf)函数的调用就等价于

```
(void) setvbuf(stream, NULL, _IONBF, 0);
```

否则的话，它就等价于

```
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

这里的BUFSIZ是在<stdio.h>中定义的宏。我们把setbuf函数看成是陈旧的内容，不建议大家在新程序中使用。



当使用setvbuf函数或者setbuf函数时，一定要确保在释放缓冲区之前已经关闭了流。

## 22.2.9 其他文件操作

```
int remove(const char *filename);
int rename(const char *old, const char *new);
```

remove函数和rename函数允许程序执行基本的文件管理操作。不同于本节中大多数其他函数，remove函数和rename函数对文件名而不是文件指针进行处理。如果调用成功，两个函数都返回零。否则，都返回非零值。

remove函数删除文件：

```
remove("foo"); /* deletes the file named "foo" */
```

如果程序使用fopen函数来产生临时文件，那么它可以使用remove函数在程序终止前删除此文件。要确信已经关闭了要移除的文件。移除文件的效果就是当前打开的是由实现定义的。

rename函数改变文件的名字：

```
rename("foo", "bar"); /* renames "foo" to "bar" */
```

如果程序需要决定使文件变为永久的，那么rename函数是很便于对用fopen函数产生的临时文件进行换名的。如果具有新的名字的文件已经存在了，那么效果会是由实现定义的。



如果打开了要换名的文件，那么一定要确保在调用rename函数之前此文件是关闭的。如果文件是打开的，则无法对文件进行换名。

## 22.3 格式化的输入/输出

在本节中，我们将介绍用来控制格式串读/写的库函数。这些库函数包括已经知道的printf函数和scanf函数。这类库函数可以在输入时把字符格式的数据转化为数字格式的数据，并且可以在输出时把数字格式的数据再转化成字符格式的数据。其他的输入/输出函数都不能完成这样的转化。

### 22.3.1 ...printf类函数

```
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
```

fprintf函数和printf函数为输出流写可变的数据项，并且利用格式串来控制输出的形式。这两个函数的原型都是以省略号(>26.1节)结尾的，省略号说明额外的实际参数的可变数量，这两个函数的返回值是写入的字符数。若出错则返回一个负值。

fprintf函数和printf函数唯一的不同就是printf函数始终向标准输出流stdout中写输出，而fprintf函数则向它自己的第一个实际参数说明的流中写输出：

```
printf("Total: %d\n", total); /* writes to stdout */
fprintf(fp, "Total: %d\n", total); /* writes to fp */
```

printf函数的调用等价于fprintf函数把stdout作为第一个实际参数而进行的调用。

但是，不要以为fprintf函数只是把数据写入磁盘文件的函数。和<stdio.h>中的许多函数一样，fprintf函数用于任何输出流都是非常好的。事实上，fprintf函数最普通的应用之一就是向标准错误stderr写出错信息，而标准错误流和磁盘文件是没有任何关系的。下面就是这类调用的一个示例：

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

即使用户重定向stdout，向stderr写入的信息也保证会出现在屏幕上。

在<stdio.h>中还有其他两种函数也可以向流写入格式化的输出。(>26.1节)这两个不常见的函数一个是vfprintf函数，另一个是vprintf函数。这两个函数都依赖于<stdarg.h>中定义的va\_list类型，因此这两个函数将和<stdarg.h>一起进行讨论。

### 22.3.2 ...printf类函数的转换说明

fprintf函数和printf函数都要求格式串包含普通字符或转换说明。普通字符将会原样输出，而转换说明则描述了如何把剩余的实参转换为字符格式显示出来。3.1节简要介绍了转换说明，而且在其后续章中还添加了不少这方面的细节。现在，我们将对已知的转换说明内容进行回顾，并且把剩余的内容补充完整。

487

...printf这类函数的转换说明由字符%和跟随其后的最多5个不同的选项构成：



下面对上述这些选项进行了详细的描述，并且显示的顺序是按照要求的次序排列的：

- 标志。(可选项，允许多于一个)。标志-使转换参数在其字段内左对齐。而其他标志则会影响显示数的形式。表22-4给出了标志的一个完整列表。

表22-4 用于...printf类函数的标志

| 标    志 | 含    义                                                  |
|--------|---------------------------------------------------------|
| -      | 在字段域内左对齐                                                |
| +      | 以+开头的正符号数                                               |
| 空格     | 用空格作为正符号数的前缀(+标志取代空格标志)                                 |
| #      | 以0开头的八进制数，以0x或0X开头的十六进制数。浮点数始终是十进制形式。不能删除由g或G转换的输出数的尾部零 |
| 0(零)   | 用前导零在数的字段宽度内进行填充。如果转换是d、i、o、u、x或X，而且指定了精度，那么可以忽略标志0     |

- **最小字段宽度**(可选项)。如果字符的数量太少以致于无法达到字段宽度的最小值时，就会对字符数量进行扩充。(默认情况下会在数据项的左侧添加空格，因此要在其字段宽度内进行右对齐。)如果字符数量过多超过了字段宽度的最小值，那么会始终完整地显示出来。字

段宽度既可以是整数也可以是字符\*。如果是字符\*，那么字段宽度由下一个参数决定。

- 精度（可选项）。精度的含义依赖于转换说明符：

如果转换说明符是d、i、o、u、x、X，那么精度表示最少数位数（如果数位数少于精度值，则添加前导零）；如果转换说明符是e、E、f，那么精度表示小数点后的数字位数；如果转换说明符是g、G，那么精度表示最大有效数位数；如果转换说明符是s，那么精度表示最大字符串数。

精度是由一个小数点（.）跟随一个整数或字符串构成的。如果出现字符\*，那么精度由下一个参数决定。如果只有小数点，那么精度就为零。

488

- h、l或L这些字母中的一个（可选项）。当把这些字母用于显示整数时，字母h说明整数是short型的；字母l说明整数是long型的。当和e、E、f、g或G一起使用时，字母l说明的是long double型的参数。
- 转换说明符。转换说明符必须是表22-5列出的某一种字符。注意f、e、E、g和G全部设计用来描述double型的值，但是把它们用于float型的值效果一样很好。这都是因为有了默认的实际参数提升（>9.3.1节），所以当对带有可变实参的函数进行传递时，float型实参会自动转化为double型数据。与此类似的，把字符串传递给...printf类函数时也会自动转换为int型，所以也可以正常使用转换说明符c。

表22-5 ...printf类函数的转换说明符

| 转换说明符   | 含义                                                                                                              |
|---------|-----------------------------------------------------------------------------------------------------------------|
| d、i     | 有符号整数转换为十进制形式                                                                                                   |
| o、u、x、X | 无符号整数转换为8进制(o)、10进制(u)或16进制(x、X)形式。x表示用小写字母a-f来显示十六进制数；而X表示用大写字母A-F来显示十六进制数                                     |
| f       | double型值转换为十进制形式，并且把小数点放置在正确的位置上。如果没有说明精度，那么在小数点后面显示6个数字                                                        |
| e、E     | 转换为科学计数法形式表示的double型值。如果没有说明精度，那么在小数点后面显示6个数字。如果选择e，那么要把字母e放在指数前面。如果选择E，那么要把字母E放在指数前面                           |
| g、G     | g会把double型值转化为f形式或者e形式。仅当数值的指数部分小于-4，或者指数部分大于或等于精度值时，会选择e形式显示。不显示尾部零，且小数点仅在后边跟有数字时才显示出来。而G会在f形式和E形式之间进行选择       |
| c       | 显示无符号字符的int型值                                                                                                   |
| s       | 写出由实参指向的字符串。当达到精度值（如果存在）或者遇到空字符串时，才停止写操作                                                                        |
| p       | 转化为可显示格式的void *型值                                                                                               |
| n       | 匹配的实参必须是指向int型（如果h放在n前面，表示为short int型数；如果l放在n前面，则表示long int型数）数的指针。到目前为止，...printf类函数调用输出的字符的数量会存储到指向的整数中，不产生输出 |
| %       | 写字符%                                                                                                            |

请认真遵守这里描述的规则。使用无效转换说明的结果是无法定义的。



许多看似可能的转换说明（比如说%le、%lf和%lg）实际是无效的。

489

### 22.3.3 ...printf类函数的转换说明示例

现在来看一些示例。在前面我们已经看过大量日常转换说明的例子了，所以下面将集中说明一些更高级的应用示例。正像前几章那样，这里将用·表示空格字符。

我们首先来看看标志在转换%d上的效果（它们在其他转换上具有类似的效果）。表22-6的第一行显示了%8d没有任何标志的效果。接下来的四行分别显示了带有标志-、+、空格以及0所产

生的效果（标志#永远不能用于%d中）。剩下的几行显示了标志组合所产生的效果。

表22-6 标志在转换%d上所产生的效果

| 转换说明  | 对123应用转换说明的结果 | 对-123应用转换说明的结果 |
|-------|---------------|----------------|
| %8d   | .....123      | .....-123      |
| %-8d  | 123.....      | -123.....      |
| %+8d  | ....+123      | ....-123       |
| % 8d  | .....123      | .....-123      |
| %08d  | 00000123      | -0000123       |
| %-+8d | +123.....     | -123.....      |
| %- 8d | ·123.....     | -123.....      |
| %+08d | +0000123      | -0000123       |
| % 08d | ·0000123      | -0000123       |

表22-7说明了在转换o、x、X、g和G上标志#所产生的效果。（标志#也可用于e、E和f，但是这种用法非常少见。）

表22-7 标志#的效果

| 转换说明 | 对123应用转换说明的结果 | 对123.0应用转换说明的结果 |
|------|---------------|-----------------|
| %8o  | .....173      |                 |
| %#8o | .....0173     |                 |
| %8x  | .....7b       |                 |
| %#8x | ....0x7b      |                 |
| %8X  | .....7B       |                 |
| %#8X | ....0X7B      |                 |
| %8g  |               | .....123        |
| %#8g |               | ·123.000        |
| %8G  |               | .....123        |
| %#8G |               | ·123.000        |

在前面的章中已经在数表示上使用过最小字段宽度和精度了，所以这里不再给出更多的示例了。而表22-8显示了在转换%s上最小字段宽度和精度所产生的效果。

表22-8 最小字段宽度和精度在转换%s上所产生的效果

| 转换说明   | 对"bogus"应用转换说明的结果 | 对"buzzword"应用转换说明的结果 |
|--------|-------------------|----------------------|
| %6s    | ·bogus            | buzzword             |
| %-6s   | bogus·            | buzzword             |
| .4s    | bogu              | buzz                 |
| %6.4s  | ..bogu            | ..buzz               |
| %-6.4s | bogu..            | buzz..               |

表22-9说明了转换%g如何以%e格式显示一些数而以%f格式显示其他数。表中的全部数都用转换说明%.4g进行了书写。前两个数所具有的指数至少为4，因此它们是按照%e的格式显示的。接下来的8个数是按照%f的格式显示的。最后两个数所具有的指数少于-4，所以也用%e的格式进行显示。

表22-9 转换%g的示例

| 数       | 对数应用转换%.4g所产生的结果 |
|---------|------------------|
| 123456. | 1.235e+05        |
| 12345.6 | 1.235e+04        |
| 1234.56 | 1235             |
| 123.456 | 123.5            |

(续)

| 数             | 对数应用转换%.4g所产生的结果 |
|---------------|------------------|
| 12.3456       | 12.35            |
| 1.23456       | 1.235            |
| 0.123456      | 0.1235           |
| 0.0123456     | 0.01235          |
| 0.00123456    | 0.001235         |
| 0.000123456   | 0.0001235        |
| 0.0000123456  | 1.235e-05        |
| 0.00000123456 | 1.235e-06        |

过去，我们假设最小字段宽度和精度都是嵌在格式串中的常量。在数中放置字符\*通常会允许把此字符说明是在格式串后的实际参数。例如，下列printf函数的调用都产生相同的输出：

```
printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```

注意，为字符\*而填充的值只出现在显示值之前。顺便说一句，字符\*的主要优势就是它允许使用宏来说明字段宽度或精度：

```
printf("%*d*", WIDTH, I);
```

在程序执行期间我们甚至可以计算出字段宽度或精度：

```
printf("%*d", page_width/num_cols, I);
```

最不常见的转换说明是%p和%n。转换%p允许显示出指针的值：

```
printf("%p\n", (void*)ptr); /* displays value of ptr */
```

虽然在调试时%p偶尔有用，但它不是大多数程序员在日常基本会用到的特性。当用%p进行显示时，C标准不会指定指针显示的形式，但它可能会以八进制或十六进制的形式显示出来。[491]

转换%n用来找出到目前为止由...printf函数调用所显示出的字符数量。例如，

```
printf("%d%n\n", 123, &len);
```

在下列调用后len的值将为3，因为在执行转换%n的时候printf函数已经显示出3个字符(123)。注意在len前面必须要有&，这样才不会显示出len自身的值。

### 22.3.4 ...scanf类函数

```
int fscanf(FILE *stream, const char *format, ...);
int scanf(const char *format, ...);
```

fscanf函数和scanf函数从输入流读入数据，并且使用格式串来说明输入的格式。任意数量的指针跟在格式串的后边作为额外的实际参数。把输入的数据项进行转换（根据格式串中的转换说明）并且存储在指针指定的位置上。

Scanf函数始终从标准输入流stdin中读入内容，而fscanf函数则从它自己的第一个实参所指定的流中读入内容：

```
scanf("%d%d", &I, &j); /* reads from stdin */
fscanf(fp, "%d%d", &I, &j); /* reads from fp */
```

scanf函数的调用等价于fscanf函数把stdin作为第一个实际参数而进行的调用。

如果发生输入失败（即没有可以读入的字符），或者发生匹配失败（即输入字符无法和格式串相匹配），那么...scanf类函数都会提前返回。这两个函数都返回读入并且赋值给实参的数据

项数量。如果在能读取任何数据项之前发生输入失败，那么会返回EOF。

在C程序中测试scanf函数的返回值的循环很普遍。例如，下列循环逐个读取一串整数，在首个问题符号处停止：

**[惯用法]**

```
while (scanf ("%d", &I) == 1) {
 ...
}
```

### 22.3.5 ...scanf类函数的格式化字符串

...scanf类函数的调用类似于...printf类函数的调用。然而，这样的相似可能会产生误解，实际上...scanf类函数的工作是不同于...printf类函数的。把scanf函数和fscanf函数看成是“模式匹配”函数是很合适的。格式串表示的模式就是...scanf类函数试图匹配的输入方式。如果输入和格式串不匹配，那么一旦发现不匹配函数就会返回。将来为了读取不匹配的输入字符将对其进行“回退”操作。

...scanf类函数的格式串可能含有三种信息类型：

- **转换说明。**在...scanf类函数格式串中的转换说明类似于...printf类函数格式串中的转换说明。大多数转换说明会在输入项的开始处跳过空白字符（%[、%c和%n例外]（>3.2.2节）。但是，转换说明从来不会跳过尾部的空白字符。如果输入含有·123¤，那么转换说明%d会把·、1、2和3吸收进来，但是留下¤不读取。（这里使用·表示空格符，而用¤表示换行符。）
- **空白字符。**在...scanf类函数格式串中的一个或多个连续的空白字符匹配零个或多个输入流中的空白字符。
- **非空白字符。**除了%，其他非空白字符和输入流中的相同字符进行匹配。

例如，格式串"ISBN %d-%d-%ld-%d"说明输入由下列这些内容构成：多个字母ISBN，一些可能的空白字符，一个整数，字符-，一个整数（可能前面放置空白字符），字符-，一个长整数（可能前面放置空白字符），字符-和一个整数（可能前面放置空白字符）。

### 22.3.6 ...scanf类函数的转换说明

实际上用于...scanf类函数的转换说明比用于...printf类函数的转换说明简单了许多。...scanf类函数的转换说明由字符%和跟随其后的选项构成。下面按照出现的顺序列出了可能的选项。

- **字符\*（可选项）。**字符\*的出现意味着赋值屏蔽（assignment suppression）：读入此数据项，但是不会把它赋值给变量。用\*匹配的数据项不会包含在...scanf类函数返回的计数中。
- **最大字段宽度（可选项）。**最大字段宽度限制了输入项的字符数量。如果达到了这个最大值，那么此数据项的转换将结束。对转换开始处跳过的空白字符不会进行统计。
- **h、l或L这些字母中的一个（可选项）。**当用于读取整数时，字母h说明相匹配的实参是指向short型整数的指针；而字母l则说明是指向long型整数的指针。当和e、E、f、g或者G一起使用时，字母l说明实参是指向double型值的指针；而字母L则说明实参是指向long double型值的指针。
- **转换说明符。**转换说明符必须是表22-10中列出的某一种字符。

表22-10 用于...scanf类函数的转换说明符

| 转换说明符 | 含 义                                                      |
|-------|----------------------------------------------------------|
| d     | 匹配十进制整数                                                  |
| i     | 匹配整数。假定数是十进制形式的，除非它以0（说明是八进制形式）开头，或者是以0x或0X（说明是十六进制形式）开头 |

| 转换说明符         | 含 义                                                                                                                         |
|---------------|-----------------------------------------------------------------------------------------------------------------------------|
| o             | 匹配八进制整数。设定相应的实参是指向unsigned int型值的指针                                                                                         |
| u             | 匹配十进制整数。设定相应的实参是指向unsigned int型值的指针                                                                                         |
| x, X          | 匹配十六进制整数。设定相应的实参是指向unsigned int型值的指针                                                                                        |
| e, E, f, g, G | 匹配float型值                                                                                                                   |
| s             | 匹配一序列非空白字符，然后在末尾添加空字符                                                                                                       |
| [             | 匹配来自扫描集合（稍后解释）的非空的字符序列，然后在末尾添加空字符                                                                                           |
| c             | 匹配n个字符，这里的n是最大字段宽度值。如果没有指定字段宽度，那么就匹配一个字符。不在末尾添加空字符                                                                          |
| p             | 匹配以...printf类函数可以写出的格式的指针值                                                                                                  |
| n             | 相应的实参必须指向int型的变量（如果在n前有h，则需要是short int型变量。如果在n前有l，则需要是long int型变量）。把到目前为止读入的字符数量存储到此变量中。没有输入会被吸收进去，而且...scanf类函数的返回值也不会受到影响 |
| %             | 匹配字符%                                                                                                                       |

数值型数据项可能始终用符号（+或-）作为开头。然而，说明符o、u、x和x把数据项转换成无符号的形式，所以通常不用这些说明符来读取负数。

说明符[是说明符s更加复杂（且更加灵活）的版本。使用[的完整转换说明格式是%[集合]或者%^[集合]，这里的集合可以是任意字符集。（但是，如果]是集合中的一个字符，那么它必须要首先出现。）%[集合]匹配集合（即扫描集合）中的任意字符序列。%^[集合]匹配非集合（换句话说，构成扫描集合的全部字符不在集合中）中的任意字符序列。例如，%[abc]匹配的是只含有字母a、b和c的任何字符串，而%^[abc]匹配的是不含有字母a、b或c的任何字符串。

许多...scanf类函数的转换说明符和<stdlib.h>中的字符串转换函数（>26.2.1节）有着紧密的联系。这些函数把字符串（比如"-297"）转换成与其等价的数值型值（-297）。例如，说明符d寻找可选择的符号+或-，后边跟着一串十进制的数字。这样就与要把字符串转换成十进制数时strtol函数所要求的格式完全一样了。表22-11说明了转换说明符和字符串转换函数之间的对应关系。

494

表22-11 ...scanf类函数转换说明符和字符串转换函数之间的对应关系

| 转换说明符         | 字符串转换函数         |
|---------------|-----------------|
| d             | 10作为基数的strtol函数 |
| i             | 0作为基数的strtol函数  |
| o             | 8作为基数的strtol函数  |
| u             | 10作为基数的strtol函数 |
| x, X          | 16作为基数的strtol函数 |
| e, E, f, g, G | strtod函数        |

当编写scanf函数的调用时是需要十分小心的。在scanf函数格式串中无效的转换说明就像在printf函数格式串中的一样糟糕。这两种情况都会导致未定义的行为出现。

### 22.3.7 scanf 函数的示例

下面出现的每个示例都将把scanf函数应用在它右侧显示的输入字符上。调用会吸收用删除线显示出的字符。调用后变量的值会出现在输入的右侧。

表22-12中的示例说明了把转换说明、空白字符以及非空白字符组合在一起的效果。表22-13中的示例显示了赋值屏蔽和指定字段宽度的效果。表22-14中的示例描述了更加深奥的转换说明符（即i、[和n）。

表22-12 scanf函数示例（第一组）

| scanf函数的调用                   | 输入        | 变量                  |
|------------------------------|-----------|---------------------|
| n = scanf("%d%d", &i, &j);   | 12, 34#   | n:1<br>i:12<br>j:?  |
| n = scanf("%d,%d", &i, &j);  | 12, , 34# | n:1<br>i:12<br>j:?  |
| n = scanf("%d ,%d", &i, &j); | 12, , 34# | n:2<br>i:12<br>j:34 |
| n = scanf("%d, %d", &i, &j); | 12, , 34# | n:1<br>i:12<br>j:?  |

495

表22-13 scanf函数示例（第二组）

| scanf函数的调用                           | 输入            | 变量                              |
|--------------------------------------|---------------|---------------------------------|
| n = scanf("%*d%d", &i);              | 12-34#        | n:1<br>i:34                     |
| n = scanf("%*s%s", str);             | My-Fair-Lady# | n:1<br>str: "Fair"              |
| n = scanf("%1d%2d%3d", &i, &j, &k);  | 12345#        | n:3<br>i:1<br>j:23<br>k:45      |
| n = scanf("%2d%2s%2d", &i, str, &j); | 123456#       | n:3<br>i:12<br>str:"34"<br>j:56 |

表22-14 scanf函数示例（第三组）

| scanf函数的调用                        | 输入           | 变量                          |
|-----------------------------------|--------------|-----------------------------|
| n = scanf("%i%i%i", &i, &j, &k);  | 12-012-0x12# | n:3<br>i:12<br>j:10<br>k:18 |
| n = scanf("%[0123456789]", str);  | 123abc#      | n:1<br>Str: "123"           |
| n = scanf("%[0123456789]", str);  | abc123#      | n:0<br>Str: ?               |
| n = scanf("%[^0123456789]", str); | ab@123#      | n:1<br>Str: "abc"           |
| n = scanf("%*d%d%n", &i, &j);     | 10-20-30#    | n:1<br>i:20<br>j:5          |

### 22.3.8 检测文件末尾和错误条件

如果要求...`scanf`类函数读入并存储n个数据项，那么希望它的返回值就是n。如果返回值小于n，那么一定是出错了。一共有三种可能情况：

- **文件末尾。** 函数在完全匹配格式串之前遇到了文件末尾。
- **错误。** 错误的发生超出了函数控制的范围。
- **匹配失败。** 数据项的格式是错误的。例如，函数可能在搜索整数的第一个数字期间遇到了一个字母。

496

但是如何可以告知发生的是哪类问题呢？在许多情况下，这都不是问题；程序出问题了，可以把它舍弃掉。然而，当需要明确失败的原因时就可能会花费很多时间。

每个流都有与之相关的两个指示器：**错误指示器**（error indicator）和**文件末尾指示器**（end-of-file indicator）。当打开流时会清除这些指示器，而当流上的操作失败时就会设置某个指示器。遇到文件末尾就设置文件末尾指示器，而遇到错误就设置错误指示器。但是，匹配失败不会改变任何一个指示器。

一旦设置了错误指示器或者文件末尾指示器，它就会保持这种状态，直到可能由`clearerr`函数的调用引发的明确清除操作为止。`clearerr`函数可以清除文件末尾指示器和错误指示器：

```
clearerr(fp); /* clears eof and error indicators for fp */
```

**Q&A**某些其他的库函数因为副作用可以清除某种指示器或两种都可以清除，所以不会需要经常使用`clearerr`函数。

虽然没有直接访问错误指示器和文件末尾指示器，但是我们可以调用`feof`函数和`ferror`函数来判断一个流的指示器，从而检测出先前在流上的操作失败的原因。如果为和流相关的fp设置了文件末尾指示器，那么`feof(fp)`函数调用就会返回非零值。如果设置了错误指示器，那么`ferror(fp)`函数的调用也会返回非零值。而其他情况下，这两个函数都会返回零。

当`scanf`函数返回小于预期的值时，可以使用`feof`函数和`ferror`函数来检测问题。如果`feof`函数返回了非零的值，那么就说明已经到达了输入文件的末尾。如果`ferror`函数返回了非零的值，那么就表示在输入过程中产生了错误。如果两个函数都没有返回非零值，那么一定是发生了匹配失败。不管问题是什么，`scanf`函数的返回值都会告诉我们在问题产生前所读入的数据项的数量。

为了明白`feof`函数和`ferror`函数可能的使用方法，现在来编写一个函数。此函数用来搜索文件中以某个整数起始的行。下面是预计的函数调用方式：

```
n = find_int("foo", &i);
```

其中，“foo”是要搜索的文件的名字，i用来存放要搜索的整数的值，而给n赋的值就是找到的整数所在行的序号。如果出现问题（文件无法打开或者输入错误，再或者没有此整数起始的行），`find_int`函数将返回一个错误代码（分别是-1、-2或-3）。

```
int find_int(const char *filename, int *ptr)
{
 FILE *fp = fopen(filename, "r");
 int line = 1;

 if (fp == NULL)
 return -1; /* can't open file */
 while (fscanf(fp, "%d", ptr) != 1) {
 if (ferror(fp)) {
 fclose(fp);
 return -2; /* input error */
 }
 if (feof(fp)) {
 fclose(fp)
```

497

```

 return -3 /* integer not found */
 }
 fscanf(fp, "%*[^\n]") /* skips rest of line */
 line++;
}

fclose(fp);
return line;
}

```

在while表达式中, find\_int函数调用fscanf函数是打算从文件中读取整数。如果尝试失败了(fscanf函数返回了不为1的值),那么find\_int函数就会调用ferror函数和feof函数来了解问题是输入错误还是遇到了文件末尾。如果都不是,那么fscanf函数一定是由于匹配错误产生的问题。由此, find\_int函数会跳过当前行剩余部分的字符,并且行计数器进行自增,然后继续在下一行寻找。请注意转换说明%\*[^\n]的用法是跳过全部字符直到下一个换行符为止。

## 22.4 字符的输入/输出

在本节中,我们将讨论用于读和写单一字符的库函数。这些函数用于文本流和二进制流是等效的。

请注意,本节中的函数把字符作为int型而非char型的值来处理。这样做的原因之一就是由于输入函数是通过返回EOF来说明一个文件末尾(或错误)情况的,而EOF又是一个负的整型常量。

### 22.4.1 输出函数

```

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);

```

putchar函数向标准输出流stdout写一个字符:

```
putchar(ch); /* writes ch to stdout */
```

**498** fputc函数和putc函数是putchar函数向任意流写字符的更通用的版本:

```

fputc(ch, fp); /* writes ch to fp */
putc(ch, fp); /* writes ch to fp */

```

虽然putc函数和fputc函数做的工作相同,但是putc函数经常作为宏来实现,而fputc函数则作为函数使用。putchar函数通常也作为宏来使用:

```
#define putchar(c) putc*((c), stdout)
```

既提供putc函数又提供fputc函数的库看起来很奇怪。但是,正如14.3节看到的那样,宏本身有几个潜在的问题。**Q&A**虽然程序员通常偏好使用putc函数,因为此函数可以提高程序的运行速度,但是fputc函数作为备选也是可用的。

如果出现了错误,那么上述这三种函数都会为流设置错误指示器并且返回EOF。否则,它们都会返回写入的字符。

### 22.4.2 输入函数

```

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);

```

getchar函数从标准输入流stdin中读入一个字符:

```
ch = getchar(); /* reads a character from stdin */
```

fgetc函数和getc函数从任意流中读入一个字符：

```
ch = fgetc(fp); /* reads a character from fp */
ch = getc(fp); /* reads a character from fp */
```

虽然getc函数和fgetc函数做的工作相同，但是getc函数通常作为宏来实现，而fgetc函数则作为函数来使用。getchar函数本身常常是按下列方式定义的宏：

```
#define getchar() getc(stdin)
```

对于从文件中读取字符来说，程序员通常喜欢用getc函数胜过用fgetc函数。因为getc函数是宏，所以它执行起来的速快较快。如果getc函数不合适，那么fgetc函数作为备选是可以使用的。

如果出现问题，那么这三个函数的行为是一样的。如果是遇到了文件末尾的问题，那么这三个函数都会设置流的文件末尾指示器，并且返回EOF。如果产生了错误，它们则都会设置流的错误指示器，并且返回EOF。为了区分这两种情况，可以调用feof函数或者ferror函数。

fgetc函数、getc函数和getchar函数最常见的用法之一就是从文件读入字符直到遇到文件末尾。一般习惯使用下列while循环来实现此目的：

**[惯用法]** while((ch = getc(fp)) != EOF){  
    ...  
}

在从与fp相关的文件中读入字符并且把它存储到变量ch（它必须是int类型的）之中后，判定条件会把ch与EOF进行比较。如果ch不等于EOF，这表示还未到达文件末尾，那么就可以执行循环体。如果ch等于EOF，则循环终止。



当对文件进行读取时，始终要把fgetc函数、getc函数或getchar函数的返回值存储在int型的变量中，而不是char型的变量中。**Q&A**把char型变量与EOF进行比较可能会产生错误的结果。

还有另外一种字符输入函数，即ungetc函数。此函数把从流中读入的字符进行“回退”，并且清除掉流的文件末尾指示器。如果在输入过程中需要“回看”字符，那么这种能力可能会非常有效。比如，为了读入一系列数字，并且在遇到首个非数字时停止操作，可以写成

```
while (isdigit(ch = getc(fp))) {
 ...
}
ungetc(ch, fp); /* puts back last value of ch */
```

通过持续调用ungetc函数而无需干涉读入操作就可以回退字符的数量，此数量依赖于实现和所含的流类型。这里只会确保第一次的ungetc函数调用是成功的。调用文件定位函数（即fseek函数、fsetpos函数或rewind函数）（>22.7节）会导致回退的字符丢失了。

如果要求ungetc函数回退，那么它会返回字符。如果试图在另一次读入操作或者文件定位操作之前回退过多的字符，那么ungetc函数会返回EOF。

### 22.4.3 程序：复制文件

下列程序用来进行文件的复制操作。当程序执行时，会在命令行上指定原始文件名和新文件名。例如，为了把文件f1.c复制给文件f2.c，可以使用命令行

```
fcopy f1.c f2.c
```

如果命令行上没有两个正确的文件名，或者至少有一个文件无法打开，那么程序fcopy都将产生出错信息。

499

```
500 fcopy.c
/* Copies a file */

#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
 FILE *source_fp, *dest_fp;
 int ch;

 if (argc != 3) {
 fprintf(stderr, "usage: fcopy source dest\n");
 exit(EXIT_FAILURE);
 }

 if ((source_fp = fopen(argv[1], "rb")) == NULL) {
 fprintf(stderr, "Can't open %s\n", argv[1]);
 exit(EXIT_FAILURE)
 }

 if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
 fprintf(stderr, "Can't open %s\n", argv[2]);
 fclose(source_fp);
 exit(EXIT_FAILURE)
 }

 while ((ch = getc(source_fp)) != EOF)
 putc(ch, dest_fp)

 fclose(source_fp);
 fclose(dest_fp);
 return 0;
}
```

采用“rb”和“wb”作为文件的模式使fcopy程序既可以复制文本文件也可以复制二进制文件。如果用“r”和“w”来代替，那么程序将无法复制二进制文件。

## 22.5 行的输入/输出

下面将要介绍读和写行的库函数。虽然这些函数也可有效的用于二进制文本流，但是它们多数用于文本流。

### 22.5.1 输出函数

```
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

在13.3节已经见过puts函数，它是用来向标准输出流stdout写入一串字符的：

**501** puts("Hi, there!"); /\* writes to stdout \*/

在写入字符串中的字符以后，puts函数总会添加一个换行符。

fputs函数是puts函数的更通用版。此函数的第二个实参指明了输出要写入的流：

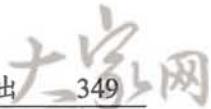
```
fputs("Hi, there!", fp); /* writes to fp */
```

不同于puts函数，fputs函数不会自己写入换行符，除非字符串中本身含有换行符。

当出现错误时，上面这两种函数都会返回EOF。否则，它们都会返回一个非负的数。

### 22.5.2 输入函数

```
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```



在13.3节中已经见过gets函数了，它是用来从标准输入流stdin中读取一串字符的：

```
gets(str); /* reads a line from stdin */
```

gets函数逐个读取字符，并且把它们存储在字符串中，直到它读到换行符时停止，且会丢弃此换行符。

fgets函数是gets函数的更通用版。它可以从任意流中读取信息。fgets函数也比gets函数更安全，因为它会限制将要存储的字符的数量。下面是使用fgets函数的方法，假设str是字符串数组的名字：

```
fgets(str, sizeof(str), fp); /* read a line from fp */
```

在响应此调用时，fgets函数将逐个读入字符，在遇到首个换行符时停止操作，或者当已经读入了sizeof(str)-1个字符时结束操作，且无论这两种情况哪种先发生都可以。如果fgets函数读入了换行符，那么它会把换行符和其他字符一起存储。（因此，gets函数从来不存储换行符，而fgets函数有时会存储换行符。）

如果出现了错误，或者是在存储任何字符之前达到了输入流的末尾，那么gets函数和fgets函数都会返回空指针。（通常，可以使用feof函数或ferror函数来检测错误的类型。）否则的话，两个函数都会返回指向读入字符串的指针。就像希望的那样，两个函数都会在字符串的末尾存储空字符。

现在既然已经学习了fgets函数，那么建议大家在大多数情况下用fgets函数来代替gets函数来使用。对于gets函数而言，始终会有跟踪超出接收数组的范畴的可能，所以只有在确保读入字符正好适合数组大小时使用gets函数才是安全的。在没有保证的时候（而且通常是没有的），更安全的做法是使用fgets函数。注意如果把stdin作为第三个实参进行传递，那么fgets函数就会从标准输入流中进行读入：

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

502

## 22.6 块的输入/输出

```
size_t fread(void *ptr, size_t size, size_t nmemb,
 FILE *stream);
size_t fwrite(const void *ptr, size_t size,
 size_t nmemb, FILE *stream);
```

fread函数和fwrite函数允许程序在单步中读和写大的数据块。**Q&A**虽然小心使用fread函数和fwrite函数可以用于文本流，但是它们主要还是用于二进制的流。

fwrite函数设计用来把内存中的数组复制给流。fwrite函数调用中首个实参就是数组的地址，第二个实参是每个数组元素的大小（按字节衡量），而第三个实参则是要写的元素数量。第四个实参是文件指针，此指针说明了要写入的数据位置。例如，为了写入整个数组a的内容，就可以使用下列fwrite函数调用：

```
fwrite(a, sizeof(a[0]), sizeof(a)/sizeof(a[0]), fp);
```

没有规定必须写入整个数组，可以很容易的写入数组任何区间的内。fwrite函数返回实际写入的元素（不是字节）的数量。如果出现写入错误，那么此数就会小于第三个实参。

fread函数将从流读入数组的元素。fread函数的实参类似于fwrite函数的实参：数组的地址、每个元素的大小（按字节衡量）、读入的元素数量以及文件指针。为了把文件的内容读入数组a，可以使用下列fread函数调用：

```
n = fread(a, sizeof(a[0]), sizeof(a)/sizeof(a[0]), fp);
```

检查fread函数的返回值是非常重要的。此返回值说明了实际读入的元素（不是字节）的数量。

此数应该等于第三个实参，除非达到了输入文件末尾或者出现了错误。`feof`函数和`ferror`函数可以用于检测任何缺陷的缘由。



请注意不要把 `fread`函数的第二个实参和第三个实参搞混了。思考下面这个 `fread`函数的调用：

```
 fread(a, 1, 100, fp);
```

这里要求 `fread`函数读入100个元素，且每个元素占有一个字节，所以它返回0~100的值。而下面的调用则要求 `fread`函数读入一个有100个字节的块：

```
 fread(a, 100, 1, fp);
```

此情况下 `fread`函数的返回值不是0就是1。

当程序需要在终止之前把数据存储到文件中时使用 `fwrite`函数是非常方便的。稍后，程序（或者由于其他原因是另外的程序）可以使用 `fread`函数从内存中把数据读回来。不考虑显示，数据不一定要是数组格式的。 `fread`函数和 `fwrite`函数都可以用于全部类型的变量。特别是可以用 `fread`函数读入结构，或者用 `fwrite`函数写出结构。例如，为了把结构变量`s`写入文件，可以使用下列形式的 `fwrite`函数调用：

```
 fwrite(&s, sizeof(s), 1, fp);
```

## 22.7 文件的定位

`fseek`、 `fgetpos`和 `fsetpos`三个函数与文件位置有关。 `fseek`的第3个实参是“文件位置”， `fgetpos`的第3个实参是“文件位置指针”， `fsetpos`的第3个实参也是“文件位置”。 `fseek`的第3个实参是“文件位置”， `fgetpos`的第3个实参是“文件位置指针”， `fsetpos`的第3个实参也是“文件位置”。

每个流都有相关联的文件位置（file position）。在打开文件时，根据模式可以在文件的起始处或者末尾处设置文件位置。然后，在执行读或写操作时，文件位置会自动推进，并且允许按照顺序贯穿整个文件。

虽然对许多应用来说顺序访问是很好的，但是某些程序需要具有在文件中跳跃的能力，即可以在这里访问一些数据又可以到那里访问其他数据。例如，如果文件包含一系列记录，我们可能希望直接跳到特殊的记录处，并对其进行读入或更新。`<stdio.h>`通过提供5个函数来支持这种形式的访问，这些函数允许程序确定当前的文件位置或者允许改变文件的位置。

`fseek`函数改变与首个实参（即文件指针）相关的文件位置。第三个实参说明计算出的新位置是否和文件的起始处、当前位置或文件末尾有关。`<stdio.h>`为此定义了3种宏：

- `SEEK_SET`: 文件的起始处。
- `SEEK_CUR`: 文件的当前位置。
- `SEEK_END`: 文件的末尾处。

第二个实参是个（可能为负的）字节计数器。例如，为了移动到文件的开始处，搜索的方向将为`SEEK_SET`，而且字节计数器为零：

```
 fseek(fp, 0L, SEEK_SET); /* moves to beginning of file */
```

为了移动到文件的末尾处，搜索的方向则应该是`SEEK_END`：

```
 fseek(fp, 0L, SEEK_END); /* moves to end of file */
```

为了向后移动10个字节，搜索的方向应该为`SEEK_CUR`，并且字节计数器要为-10：

```
 fseek(fp, -10L, SEEK_CUR); /* moves back 10 bytes */
```

注意，字节计数器的类型是long int型的，所以这里用0L和-10L作为实参。（当然，用0和-10也可以工作，因为实参会自动转化为正确的类型。）

通常情况下，fseek函数返回零。如果产生错误（例如，要求的位置不存在），那么fseek函数就会返回非零值。

顺便提一句，文件定位函数最适合用于二进制的流。C语言不禁止程序对文本流使用这些定位函数，但是对于操作系统的差异要小心。由于这些差异，fseek函数对流是文本型的还是二进制型的很敏感。对于文本流而言，或者（1）offset（fseek的第二个实参）必须为零；或者（2）whence（fseek的第三个实参）必须是SEEK\_SET，且通过前面的ftell函数调用获得offset的值。（换句话说，我们只可以利用fseek函数移动到文件的起始处或者文件的末尾处，在或者返回前一次访问到的位置。）对于二进制流而言，fseek函数不要求支持whence是SEEK\_END的调用。

ftell函数以长整型返回当前文件位置。（如果发生错误，ftell函数会返回-1L，并且把错误码存储到errno中。）ftell可能会存储返回的值并且稍后将其提供给fseek函数的调用，这也使返回前一个文件位置成为可能：

```
long int file_pos;
...
file_pos = ftell (fp); /* saves current position */
...
fseek (fp, file_pos, SEEK_SET); /* returns to old position */
```

如果fp是二进制流，那么ftell(fp)调用会以字节计数来返回当前文件位置，其中零表示文件开始。（但是，如果fp是文本流，ftell(fp)返回的值不一定是字节计数，结果，最好不要对ftell函数返回的值进行算术运算。例如，为了查看两个文件位置的差距而把ftell返回的值相减不是个好做法。）

rewind函数会把文件位置设置在起始处。调用rewind(fp)函数几乎等价于fseek(fp, 0L, SEEK\_SET)，两者的差异是rewind函数不返回值，但是会为fp清除掉错误指示器。

fseek函数和ftell函数都有一个问题：它们都限制文件的位置必须是存储在长整型数中。**Q&A**为了用于大型文件，标准C提供了两种额外函数，即fgetpos函数和fsetpos函数。这两个函数都可以用于大型文件，因为它们都用fpos\_t型值来表示文件位置。fpos\_t型值不一定就是整数，比如，它可以是结构。

调用fgetpos(fp, &file\_pos)会把与fp相关的文件位置存储到file\_pos型变量中。调用fsetpos(fp, &file\_pos)会为fp设置文件的位置，且此位置是存储在file\_pos中的值。（此值必须是通过前一个fgetpos调用获得的。）如果fgetpos函数调用失败或者fsetpos函数调用失败，那么都会把错误代码存储到errno中。当调用成功时，这两个函数都会返回零；否则，都会返回非零值。

下面是使用fgetpos函数和fsetpos函数保存文件位置并且稍后返回的方法：

```
fpos_t file_pos;
...
fgetpos (fp, &file_pos); /* saves current position */
...
fsetpos (fp, &file_pos); /* returns to old position */
```

## 程序：修改零件记录文件

下面这个程序把part结构的二进制文件读入到数组中，且给每个结构的成员on\_hand置为0，然后再把此结构写回到文件中。注意，打开的文件是既可读又可写的（"rb+"）。

```
invclear.c
/* Modifies a file of part records by setting the quantity
```

```

on hand to zero */

#include <stdio.h>
#include <stdlib.h>

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
 int number;
 char name [NAME_LEN+1];
 int on_hand;
} inventory [MAX_PARTS];

int num_parts;

main()
{
 FILE *fp;
 int i;

 if ((fp = fopen ("invent.dat", "rb+")) == NULL) {
 fprintf (stderr,"Can't open inventory file\n");
 exit(EXIT_FAILURE);
 }

 num_parts = fread (inventory, sizeof (struct part),MAX_PARTS, fp);
 for (i = 0; i < num_parts; i++)
 inventory[i].on_hand = 0;

 rewind (fp);
 fwrite (inventory, sizeof(struct part), num_parts, fp);
 fclose (fp);

 return 0;
}

```

顺便说一下，这里调用rewind函数是很关键的。在调用完fread函数之后，文件位置是在文件的末尾。如果打算不先调用rewind函数就调用fwrite函数，那么fwrite函数将会在文件末尾添加新的数据，而不是在旧的数据上覆写。

## 22.8 字符串的输入/输出

```

int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);

```

sprintf函数和sscanf函数允许用字符串作为流读/写数据。

sprintf函数与printf函数和fprintf函数都很类似，唯一的不同就是sprintf函数把输出写入字符数组（利用sprintf函数的第一个实参指向的数组）而不是流中。sprintf函数的第二个实参是格式串，这与printf函数和fprintf函数所用的一样。例如，

```
 sprintf(str, "%d/%d/%d", 9, 20, 94);
```

此调用将会把9/20/94复制到str中。当完成向字符串写入的时候，sprintf函数会添加一个空字符，并且返回所存储字符的数量（不计空字符）。

sprintf函数有着广泛的应用。例如，有些时候可能希望为输出而对数据进行格式化，但不是真的要把数据写出。这时就可以使用sprintf函数来实现格式化，然后把结果存储在字符串中直到需要产生输出的时候再写出。sprintf函数还可以用于数的格式向字符格式的转化。

sscanf函数与scanf函数和fscanf函数都很类似，唯一的不同就是sscanf函数是从字符串（利用sscanf函数的第一个实参指向的字符串）而不是流中读取数据。sscanf函数的第二

个实参是格式串，这与scanf函数和fscanf函数所用的一样。

sscanf函数对于从由其他输入函数读入的字符串中提取数据非常方便。例如，可以使用fgets函数来获取一行输入，然后把此行数据传递给sscanf函数进一步处理：

```
fgets(str, sizeof(str), stdin); /* reads a line of input */
sscanf(str, "%d%d", &i, &j); /* extracts two integers */
```

507

用sscanf函数代替scanf函数或者fscanf函数的好处之一就是，可以按需要多次检测输入行，而不再只是一次，这样使识别替换的输入格式和从错误中恢复都变得更加容易了。下面思考一下读取日期的问题。读取的日期既可以是月/日/年的格式，也可以是月-日-年的格式。假设str含有一行输入，那么可以强制月、日和年按照如下形式显示：

```
if (sscanf (str, "%d /%d/%d", &month,&day, &year) == 3)
 printf ("Month: %d, day: %d, year: %d\n", month, day, year);
else if (sscanf (str, "%d -%d -%d", &month, &day, &year) == 3)
 printf ("Month: %d, day: %d, year: %d\n", month, day, year);
else
 printf ("Date not in the proper form\n");
```

像scanf函数和fscanf函数一样，sscanf函数也返回成功读入并存储的数据项的数量。如果在找到第一个数据项之前到达了字符串的末尾，那么sscanf函数会返回EOF。

还有另一个字符串输入/输出函数，即vsprintf函数（>26.1.2节）。因为vsprintf函数依赖于在<stdarg.h>中定义的va\_list类型，所以把它和<stdarg.h>一起讨论。

## 问与答

问：你这里只列出了3种标准流，即stdin、stdout和stderr，但是我的编译器还提供了stdaux和stdprn。这些是什么呢？(p.330)

答：虽然大多数DOS编译器都支持stdaux和stdprn，但是它们不是标准C的内容。stdaux表示COM（串行）端口，而stdprn表示PRN（并行）端口。通过在stdaux上执行输入/输出操作，程序可以通过串行端口连接的设备（比如调制解调器）进行交流。通过向stdprn进行写操作，程序可以直接向打印机发送输出。

问：如果我使用输入重定向或输出重定向，那么重定向的文件名会作为命令行参数显示出来吗？

答：不会。操作系统会把这些文件名从命令行中移走。假设用下列录入运行程序：

```
demo foo <in_file bar >out_file baz
```

argc的值将为4，argv[0]将会指向程序名，argv[1]会指向"foo"，argv[2]会指向"bar"，而argv[3]则会指向"baz"。

问：我正打算编写一个需要在文件中存储数据的程序，以便稍候其他程序可以读取某些数据。对数据的存储格式而言，文本格式和二进制格式哪种更好呢？

答：这是有依赖关系的。如果数据全部以文本开始，那么用哪种格式存储没有太大的差异。然而，如果数据包含数，那么决定就非常困难。

508

通常二进制格式比较受欢迎，因为此种格式的读和写都非常快。当存储到内存中时，数已经是二进制格式了，所以将它们复制给文件是非常容易的。用文本格式写数据就会相对慢许多，因为每个数必须要转化成（通常用fprintf函数）字符格式。而稍候读取文件也将花费更多的时间，因为必须要把数从文本格式转换回二进制格式。此外，就像在22.1节看到的那样，以二进制格式存储数据常常会节省空间。

然而，二进制文件有两个缺点。很难阅读，这也就妨碍了调试过程。而且，二进制文件通常无法从一个系统移植到另一个系统，因为不同类型的计算机存储数据的方式是不同的。比如，某些机器按照2个字节的方式存储整数，而其他一些机器则可能用4个字节进行存储。一些机器期望首先存储数

的高字节部分，而其他机器则可能希望先存储低字节部分。

问：用于UNIX系统的C程序好像从不在模式字符串中使用字母**b**，即使在打开的文件是二进制的时候也是如此。这意味着什么？(p.332)

答：在UNIX系统中，文本文件和二进制文件具有完全相同的格式，所以不需要使用字母**b**。但是，UNIX的程序员也应该包含字母**b**，这样他们的程序将会更加适合移植到其他操作系统上。

问：我已经看过调用**fopen**函数并且把字母**t**放在模式字符串中的程序了。字母**t**意味着什么呢？

答：C标准允许额外的字符在模式字符串中出现，但是它们要跟在**r**、**w**、**a**、**b**或+的后边。DOS编译器经常允许使用**t**来说明打开的文件是文本模式而不是二进制模式的。当然，无论如何文本模式都是默认的，所以字母**t**不增加任何内容。任何可能的情况下，最好避免使用字母**t**和其他不可移植的特性。

问：为什么调用**fclose**函数来关闭文件呢？当程序终止时所有打开的文件都会自动关闭难道不是真的吗？

答：通常情况下是真的，但如果调用**abort**函数(►26.2.5节)来终止程序就不是了。即使在不用**abort**函数的时候，调用**fclose**函数始终还是有许多好的理由。首先，这样会减少打开文件的数量。操作系统对程序每次可以打开的文件数量都有限制，而大规模的程序可能会与此种限制相冲突。(定义在<stdio.h>中的宏**FOPEN\_MAX**指定了实现可以同时打开的文件的最少量。)其次，这样做程序会变得易于阅读和修改。通过寻找**fclose**函数，读者很容易确定不再使用此文件的位置。最后，这样做很安全。关闭文件确保可以正确地更新文件的内容和目录。如果稍候程序崩溃了，至少文件不会受到影响。

问：我正在编写的程序会提示用户录入文件的名字。我要设置多长的字符数组才可以存储这个文件名字呢？(p.334)

答：这与使用的操作系统有关。幸运的是，你可以使用宏**FILENAME\_MAX**(定义在<stdio.h>中的)来指定数组的大小。**FILENAME\_MAX**是字符串的长度，这个字符串将会保存实现保证可以打开的最长的文件名。

问：**fflush**可以清除为读和写而打开的流吗？

答：根据C标准，调用**fflush**函数的结果是把流定义为(1)为输出打开，或者(2)为更新打开并且流的最后操作不是读。在其他全部情况下，调用**fflush**函数的结果是未定义的。当传递给**fflush**函数空指针时，它会清除所有满足(1)或(2)的流。

问：在...printf类函数或...scanf类函数调用中，格式串可以是变量吗？

答：当然。它可以是char \*类型的任意表达式。这个特性使...printf类函数和...scanf类函数甚至比猜想的缘由更加多样。请看下面这个来自Kernighan和Ritchie所著的*The C Programming Language*一书的经典案例。此案例显示出程序的命令行参数，以空格分隔：

```
while (--argc > 0)
 printf((argc > 1) ? "%s" : "%s", ++argv);
```

这里的格式串是表达式(argc > 1) ? "%s" : "%s"，其结果是除了最后一个参数以外，其他所有命令行参数都会使用"%s"。

问：除了**clearerr**函数，哪个库函数可以清除流的错误指示器和文件末尾指示器？(p.345)

答：调用**rewind**函数可以清除这两种指示器，就好像打开或重新打开流一样；而调用**ungetc**函数、**fseek**函数或者**fsetpos**函数仅可以清除文件末尾指示器。

问：我无法使**feof**函数工作。因为即使到了文件末尾，它好像还是返回0。我做错了什么吗？

答：当前面的读操作失败时，**feof**函数只会返回1。在尝试读之前，你无法使用**feof**函数来检查文件末尾。相反，你应该首先尝试读，然后检查来自输入函数的返回值。如果返回的值表明操作不成功，那么你可以随后使用**feof**函数来确定失败是否是因为到达了文件末尾。换句话说，最好不要认为调用**feof**函数是检测文件末尾的方法；相反，把它想成是确认文件尾方式的想法正是读取操作失败的原因的方法。

问：我始终不明白为什么输入/输出库除了提供名为**fputc**和**fgetc**的函数还提供名为**putc**和**getc**的宏呢？依据21.2节的介绍，**putc**和**getc**已经有两种版本了(宏和函数)。如果需要真正的函数来代

替宏，我们可以通过未定义的宏来显示putc函数或getc函数。所以，为什么要有fputc和fgetc存在呢？(p.346)

答：这是历史原因。早在标准化以前，C语言没有规则要求真正的函数在库中备份每种参数化的宏。putc函数和getc函数传统上只作为宏来实现，而fputc函数和fgetc函数则只作为函数来实现。

\*问：把fgetc函数、getc函数或者getchar函数的返回值存储到char型变量中会有什么问题？我不明白为什么判断char型变量的值是否为EOF会得到错误的结果呢？(p.347)

答：在这个判定中有两种情况可能导致错误的结果。为了使下面的讨论更具体，这里假设使用二进制补码存储方式。

首先，假定char型是无符号类型。（回想到某些编译器把char型变量作为有符号类型来处理，而其他编译器则会把它看成是无符号类型的。）现在假设getc函数返回EOF，这里把用来存储EOF的char型变量命名为ch。既然EOF是-1的代名词，所以ch将用值255作为结束。ch无符号字符与EOF有符号整数进行比较就要求把ch转化成有符号整数（在这个例子中是255）。因为255不等于-1，所以与EOF的比较失败了。

反之，现在假设char是有符号类型。如果getc函数从二进制流中读取了含有值255的字节，请想想这样会产生什么情况呢？把255存储在char型变量中将会为它带来值-1，因为ch是有符号字符。如果判断ch是否等于EOF，将会（错误地）产生真的结果。

问：为什么22.4节（字符的输入/输出）不介绍任何关于getch函数和getche函数的内容呢？

答：简单说就是因为getch函数和getche函数都不是标准输入/输出函数库的内容。这些允许程序捕捉单个按键的函数通常是由DOS编译器在非标准<conio.h>（控制台的输入输出）中提供的。

标准输入函数getc、fgetc和getchar都是分配缓冲区的。也就是说，这些函数是在用户按下回车（返回）键时才开始读取输入的。而另一方面，getch函数和getche函数在按回车时返回字符。这两个函数的差异在于getch不回送输入的字符而getche回送。换句话说，如果使用函数getch，用户不会看见自己正在敲击的字符是什么。

当用户按下了功能键、光标键或者计算机键盘上任何其他的特殊键时，getch函数和getche函数都可以检测到。当用户按下这类键时，这两个函数都会返回0。在下一次调用函数时，它们会返回“扫描码”用以说明按下的是哪个键。如果你使用的是DOS编译器，那么编译器手册应该会提供一个扫描码的列表，或者可以参考有关PC系列编程的书籍。

像getch和getche这类函数对于编写某些类型的程序是非常有用的。首先，它们允许交互式程序的构造，比如编辑器。这样做应该能即刻响应用户的输入。其次，它们允许程序告知什么时候用户按下特殊键。最后，函数getch允许程序读取输入而不用回送，这在某些情况下确实是很有趣处的（比如说，读入密码的情况）。

当然，getch函数和getche函数也有它们的问题。这些函数不会给用户回退和纠正错误的机会。而且，由于这两种函数都是非标准的，所以调用它们的程序可能无法移植到UNIX或其他操作系统。

问：当正在读取用户输入时，如何能跳过当前输入行左侧的全部字符呢？

答：一种可能是编写一个小函数来读入并且忽略掉（并且包含）第一个换行符之前的所有字符：

```
void skip_line(void)
{
 while (getchar() != '\n')
 ;
}
```

另外一种可能是要求scanf函数跳过第一个换行符前的所有字符：

```
scanf("%*[^\n]"); /* skips characters up to new-line */
```

scanf函数将读取第一个换行符之前的所有字符，但是不会把它们存储下来（\*说明会抑制赋值操作）。使用scanf函数的唯一问题是它会留下换行符不读，所以可能需要单独丢弃换行符。

无论做什么，都不要调用fflush函数：

`fflush(stdin); /* effect is undefined */`

虽然某些实现允许使用`fflush`函数来“清洗”未读取的输入，但是这样做并不是一个好主意。

`fflush`函数是设计用来清洗输出流的。C标准规定`fflush`函数对输入流的效果是未定义的。

问：为什么把`fread`函数和`fwrite`函数用于文本流不是一个好主意呢？(p.349)

答：困难之一是，在某些操作系统中当对文本文档写操作时会把换行符变成一对字符（详细内容见22.1节）。这就需要考虑这种扩展，否则就很有可能丢失数据的跟踪。例如，如果使用`fwrite`函数来写含有80个字符的块，所以有些块可能在文件结束时会占用多于80个字节因为换行符可能被扩展。

问：为什么有两套文件定位函数（即`fseek/ftell`和`fsetpos/fgetpos`）呢？一套函数难道不够吗？(p.351)

答：`fseek`函数和`ftell`函数作为C库的一部分已有些年头了，所以它们必须包含在C标准里。可惜的是这两个函数无法用于超大规模的文件（这类文件在设计C语言的年代并不普遍），因此在C语言标准化期间又加入了`fsetpos`函数和`fgetpos`函数。

512

问：为什么本章不讨论屏幕控制，即移动光标、改变屏幕上字符颜色等呢？

答：标准C没有提供用于屏幕控制的函数。标准只发布那些通过广泛的计算机和操作系统可以合理标准化的问题，而屏幕控制超出了这个范畴。如果在DOS系统中工作，对于屏幕控制可以有几种选择，其中包括可以调用`<conio.h>`中的函数，大多数DOS编译器都提供`<conio.h>`。UNIX系统程序员面临的问题则是程序需要在多种不同的终端上工作。解决这个问题的传统做法是使用UNIX的`curses`库，这个库支持不依赖终端方式的屏幕控制。

问：图形的标准函数是怎样的呢？

答：没有图形的标准函数，可参见前一个问题的答案。如果你的程序需要图形功能，那么有几种选择。你所在的编译器可能带有图形库，或者也可以获得由第三方编写的图形库，再或者是你编写自己的库。

## 练习

### 22.1节

1. 指出下列每个文件可能是包含文本数据还是二进制数据。
  - (a) 由C语言编译器产生的目标代码文件。
  - (b) 由C语言编译器产生的程序列表。
  - (c) 从一台计算机发送到另一台计算机的电子邮件。
  - (d) 含有图形映象的文件。

### 22.2节

2. 指出在下列每种情况下会把哪种模式字符串传递给`fopen`函数：
  - (a) 数据库管理系统打开含有将被更新的记录的文件。
  - (b) 邮件程序打开存有消息的文件以便可以在文件末尾添加额外的信息。
  - (c) 图形程序打开含有将被显示在屏幕上的图片的文件。
  - (d) 操作系统命令解释器打开含有将被执行的命令的“批文件”（或者“壳脚本”）。
3. 扩充`canopen`程序，以便用户可以把任意数量的文件名放置在命令行中：

```
canopen foo bar baz
```

这个程序应该为每个文件分别显示出`can be opened`消息或者`can't be opened`消息。如果命令行中没有参数，那么程序应该返回2；如果无法打开任何文件，那么程序返回1；如果可以打开所有文件，程序应返回0。

### 22.3节

513

4. 请指出如果`printf`函数用`%#012.5g`作为转换说明来执行显示操作，下列数据显示的形式：

- (a) 83.7361
- (b) 29748.6607

- (c) 1054932234.0  
 (d) 0.0000235218

5. `printf`函数的转换说明`%4d`和`%04d`有区别吗？如果有，请说明区别是什么。  
 \*6. 编写`printf`函数的调用，要求如果变量`widget`（类型为`int`型）的值为1，则显示`1 widget`；如果值为`n`，则显示出`n widgets`。不允许使用`if`语句或任何其他语句；答案必须是单独的一个`printf`调用。  
 \*7. 假设按照下列形式调用`scanf`函数：

```
n = scanf("%d%f%d", &i, &x, &j);
```

（其中，`i`、`j`和`n`都是`int`型变量，而`x`是`float`型变量。）假设输入流含有下面所示的字符，请指出这个调用后`i`、`j`、`n`和`x`的值。此外，请说明一下调用会消耗掉哪些字符。

- (a) 10·20·30#  
 (b) 1.0·2.0·3.0#  
 (c) 0.1·0.2·0.3#  
 (d) .1·.2·.3#

8. 在前面几章中，当希望跳过空白字符而读取非空字符时，已经使用过`scanf`函数的“`%c`”格式串。而一些程序员用“`%1s`”来代替。这两种方法等效吗？如果不等效，区别是什么？

#### 22.4节

9. 要想从标准输入流中读取一个字符，下列调用方式哪种是无效的？

- (a) `getch()`  
 (b) `getchar()`  
 (c) `getc(stdin)`  
 (d) `fgetc(stdin)`

10. 程序`fcopy`有一个小缺陷：当它向目的文件写时无法检查错误。虽然在写操作过程中错误是极少的，但是偶尔会发生（比如，磁盘可能会变满）。假设希望一旦发生错误，程序可以写出消息并且立刻终止，请说明如何为`fcopy.c`添加遗漏的错误检查。

11. 在程序`fcopy`中出现了下列循环：

```
while ((ch = getc (source_fp)) != EOF)
putc(ch, dest_fp);
```

假设忽略表达式`ch = getc (source_fp)`两边的圆括号：

```
while (ch = getc (source_fp) != EOF)
putc(ch, dest_fp);
```

程序可以无错通过编译？如果可以，那么运行时程序会做些什么呢？

514

12. 编写一个名为`toupper`的程序，用来把文件中的所有字母转化成大写形式。（其他非字母字符不改变。）用户将采用命令行上的输入文件名：

```
toupper test.doc
```

让程序`toupper`把输出写到`stdout`中。

13. 编写一个名为`fact`的程序，通过把任意数量的文件写到标准输出中而把这些文件一个接一个的“拼接”起来，而且文件之间没有间隙。例如，下列命令将在屏幕上显示文件`f1.c`、`f2.c`和`f3.c`：

```
fcat f1.c f2.c f3.c
```

如果任何文件都无法打开，那么程序`fcat`应该发出错误信息。提示：因为每次只可以打开一个文件，所以程序`fcat`只需要一个文件指针变量。一旦对一个文件完成操作，程序`fcat`在打开下一个文件时可以使用同一个文件指针变量。

14. 让下列每一个程序都通过命令行获得文件名，并都把输出写到`stdout`中。

(a) 编写一个名为`cntchar`的程序，用来统计文本文件中字符的数量。

(b) 编写一个名为`cntword`的程序，用来统计文本文件中单词的数量（所谓“单词”指的是不含空白字符的任意序列）。

- (c) 编写一个名为cntline的程序，用来统计文本文件中行的数量。
15. 20.1节中的程序xor拒绝对原始格式或加密格式中是控制字符的字节进行加密。现在可以摆脱这种限制了。修改此程序使输入文件名和输出文件名都是命令行的参数。以二进制形式打开这两个文件，并且把用来检查原始字符或加密字符是否是控制字符的判断删除。
16. 编写一个名为hexdump的程序，以十六进制代码序列的形式显示文件中的字节，且每行显示20个代码（如下所示）。

```
43 68 61 69 72 6d 61 6e 20 42 69 6c 6c 20 6c 65 61 64 73 20
74 68 65 20 68 61 70 70 79 20 77 6f 72 6b 65 72 73 20 69 6e
20 73 6f 6e 67 21 0d 0a
```

用户需要在命令行中指定文件名。请确保文件以"rb"模式打开。

17. 在进行文件内容压缩的众多方法中，最简单快捷的方法之一是行程长度编码方式。这种方法通过一对字节替换相同的字节序列来进行文件的压缩：重复计数后面跟着重复的字节。例如，假设文件以下列字节序列开始进行压缩（以十六进制形式显示）：

```
46 6f 6f 20 62 61 72 21 21 21 20 20 20 20 20
```

压缩后的文件将包含下列字节：

```
01 46 02 6f 01 20 01 62 01 61 01 72 03 21 05 20
```

如果原始文件包含许多相同字节的长序列，那么行程长度编码的方法非常适用。最差的情况是行程长度编码可能实际上是文件的长度的两倍。

- (a) 请编写名为comp的程序，此程序使用行程长度编码方法来压缩文件。为了运行程序comp，将使用下列格式的命令行：

**515** comp 原始文件 压缩后的文件

如果压缩后的文件没有扩充，那么程序comp将添加扩展名.rle。例如，命令

comp foo bar

将会使程序comp创建名为bar.rle的文件，并且把文件foo的压缩版写到此文件中。（程序comp将在文件bar.rle开始处保存文件foo的名字。）提示：练习16中的程序hexdump可以用来调试。

- (b) 请编写名为uncomp的程序，此程序是程序comp的反向操作。程序uncomp的命令行格式为：

uncomp 压缩后的文件

如果压缩后的文件没有扩充，那么程序uncomp将添加扩展名.rle。例如，命令

uncomp bar

将会使程序uncomp打开文件bar.rle，并且写出文件内容的未压缩版，同时把文件的名字保存在bar.rle的开始处。

## 22.5节

### 18.

- (a) 请编写自己版本的fgets函数，使此函数的操作尽可能和实际的fgets函数相同。特别是一定要确保函数具有正确的返回值。为了避免和标准库发生冲突，请不要把自己编写的函数也命名为fgets。
- (b) 请编写自己版本的fputs函数，规则和(a)要求的一样。

## 22.6节

### 19. 通过添加两个新的操作的方式修改16.3节中的invent程序：

- 在指定文件中保存数据库。
- 从指定文件中装载数据库。

分别使用代码d（转储）和r（恢复）来表示这两种操作。与用户的交互应该按照下列显示进行：

```
Enter operation code: d
Enter name of output file: invent.dat
```

```
Enter operation code: r
```

Enter name of input file: invent.bat

20. 编写程序对由invent程序存储的含有零件记录的两个文件进行合并（见练习19）。假设每个文件中的记录都是根据零件编号进行排序的，而且希望结果文件也应是排好序的。如果两个文件都拥有相同编号的零件，那么要对记录中存储的数量进行合并。（作为连贯的检查，程序要比较零件的名称，并且在不匹配时显示出错信息。）程序在命令行上要包含输入文件名以及合并后的文件名。
- \*21. 修改17.5节中的程序invent2，方法是添加练习19中描述的d（转储）操作和r（恢复）操作。因为零件的结构不存储在数组中，所以d操作无法通过单独一个fwrite调用来保存所有内容。因而，它需要访问链表中的每个节点，保存零件的编号、名称以及文件中现有零件的数量。（不保存指针next，因为一旦程序终止，此指针将不会有效。）当程序从文件中读取零件时，r操作将每次一个节点地重新构建列表。

## 22.7节

22. 编写fseek函数的调用来在二进制文件中执行下列文件定位操作，其中，二进制文件的数据以64字节“记录”的形式进行排列。采用fp作为下列每种情况中的文件指针。
- (a) 移动到记录n的开始处（假设文件中的首记录记为0）。
  - (b) 移动到文件中最后一条记录的开始处。
  - (c) 向前移动一条记录。
  - (d) 向后移动两条记录。

516

## 22.8节

23. 编写一个名为disptime的程序，用来从命令行读取数据，并且按照下列格式显示出来：

September 13, 1995

允许用户以9-13-95或者9/13/95的形式录入日期，并假设日期中没有空格。如果没有按照指定格式录入日期，那么程序显示出错信息。提示：使用sscanf函数从命令行参数中截取出年、月、日。

517

# 第23章

## 库对数值和字符数据的支持

与计算机过长时间的接触把数学家变成了书记员，反之亦然。

本章会介绍5个函数库的头，这5个头提供了对数值、字符和字符串的支持。23.1节和23.2节介绍了`<float.h>`和`<limits.h>`，它们包含了用于描述数值和字符类型特性的宏。23.3节~23.5节讨论余下的标准头：`<math.h>`（数学函数）、`<ctype.h>`（字符函数）以及`<string.h>`（字符串函数）。

### 23.1 `<float.h>`：浮点型的特性

`<float.h>`中提供了用来定义浮点型的范围及精度的宏。在`<float.h>`中没有类型和函数的定义。

有两个宏对所有浮点型适用。`FLT_ROUND`说明了浮点加法的舍入模式。表23-1列出了`FLT_ROUND`的可能值。`FLT_RADIX`指定了指数基数的形式，最小值是2（二进制）。

表23-1 舍入模式

| 取 值 | 含 义     |
|-----|---------|
| -1  | 不确定     |
| 0   | 趋于零     |
| 1   | 趋于最近有效值 |
| 2   | 趋于正无穷   |
| 3   | 趋于负无穷   |

519

其他宏用来描述特定类型的特性，这里会用一系列的表格来描述。根据宏是针对`float`、`double`还是`long double`类型，每个宏都会以`FLT`、`DBL`或`LDBL`开头。C标准对这些宏给出了相当具体的定义，因此这里的介绍会更注重易于理解，而不会十分精确。依据C标准，表中列出了部分宏的最大值和最小值。

表23-2列出了与定义有效数字个数相关的宏。

表23-2 `<float.h>`中的有效数字宏

| 宏 名                        | 取 值       | 宏的描述                                |
|----------------------------|-----------|-------------------------------------|
| <code>FLT_MANT_DIG</code>  |           | 有效数字的个数（基数 <code>FLT_RADIX</code> ） |
| <code>DBL_MANT_DIG</code>  |           |                                     |
| <code>LDBL_MANT_DIG</code> |           |                                     |
| <code>FLT_DIG</code>       | $\geq 6$  | 有效数字的个数（十进制）                        |
| <code>DBL_DIG</code>       | $\geq 10$ |                                     |
| <code>LDBL_DIG</code>      | $\geq 10$ |                                     |

表23-3列出了与指数相关的宏。

表23-3 &lt;float.h&gt;中的指数宏

| 宏名              | 取值         | 宏的描述                  |
|-----------------|------------|-----------------------|
| FLT_MIN_EXP     |            | FLT_RADIX能表示的最小(负的次幂) |
| DBL_MIN_EXP     |            |                       |
| LDBL_MIN_EXP    |            |                       |
| FLT_MIN_10_EXP  | $\leq -37$ | 10能表示的最小(负的次幂)        |
| DBL_MIN_10_EXP  | $\leq -37$ |                       |
| LDBL_MIN_10_EXP | $\leq -37$ |                       |
| FLT_MAX_EXP     |            | FLT_RADIX能表示的最大次幂     |
| DBL_MAX_EXP     |            |                       |
| LDBL_MAX_EXP    |            |                       |
| FLT_MAX_10_EXP  | $\geq +37$ | 10能表示的最大次幂            |
| DBL_MAX_10_EXP  | $\geq +37$ |                       |
| LDBL_MAX_10_EXP | $\geq +37$ |                       |

表23-4列出了其他宏，描述了最大值、最接近0的值（最小正数），两个数之间的最小差值。

表23-4 &lt;float.h&gt;中的最大值、最小值和差值宏

| 宏名           | 取值              | 宏的描述                                                    |
|--------------|-----------------|---------------------------------------------------------|
| FLT_MAX      | $\geq 10^{+37}$ | 最大的浮点数                                                  |
| DBL_MAX      | $\geq 10^{+37}$ |                                                         |
| LDBL_MAX     | $\geq 10^{+37}$ |                                                         |
| FLT_MIN      | $\leq 10^{-37}$ | 最小的规格化浮点数                                               |
| DBL_MIN      | $\leq 10^{-37}$ |                                                         |
| LDBL_MIN     | $\leq 10^{-37}$ |                                                         |
| FLT_EPSILON  | $\leq 10^{-5}$  | 最小的正数 $\epsilon$ , $\epsilon$ 满足: $1.0+\epsilon$ 不等于1.0 |
| DBL_EPSILON  | $\leq 10^{-9}$  |                                                         |
| LDBL_EPSILON | $\leq 10^{-9}$  |                                                         |

由于只有进行数值分析的专家才会对上述<float.h>中定义的宏感兴趣，这可能是标准库中最不常用的头。

520

## 23.2 <limits.h>: 整值类型的大小

<limits.h>中提供了用于定义每种整型和字符型取值范围的宏。在<limits.h>中没有定义类型或函数。

在<limits.h>中，一组宏用于字符型: char、signed char和unsigned char。表23-5列举了这些宏以及它们的最大值或最小值。

表23-5 &lt;limits.h&gt;中的字符型宏

| 宏名         | 取值          | 宏的描述          |
|------------|-------------|---------------|
| CHAR_BIT   | $\geq 8$    | 每个字符包含位的个数    |
| SCHAR_MIN  | $\leq -127$ | 最小带符号字符       |
| SCHAR_MAX  | $\geq +127$ | 最大带符号字符       |
| UCHAR_MAX  | $\geq 255$  | 最大无符号字符       |
| CHAR_MIN   | ①           | 最小字符          |
| CHAR_MAX   | ②           | 最大字符          |
| MB_LEN_MAX | $\geq 1$    | 多字节字符最多包含的字节数 |

① 如果char类型被当作signed char类型，CHAR\_MIN与SCHAR\_MIN相等；否则CHAR\_MIN为0。

② 根据char类型被作为signed char或unsigned char，CHAR\_MAX分别与SCHAR\_MAX或UCHAR\_MAX相等。

其他在<limits.h>中定义的宏针对整型：short int、unsigned short int、int、unsigned int、long int以及unsigned long int。表23-6列举了这些宏以及它们的最大值或最小值。

表23-6 &lt;limits.h&gt;中整型的宏

| 宏 名       | 取 值                | 宏的描述      |
|-----------|--------------------|-----------|
| SHRT_MIN  | $\leq -32767$      | 最小短整型数    |
| SHRT_MAX  | $\geq +32767$      | 最大短整型数    |
| USHRT_MAX | $\geq 65535$       | 最大无符号短整型数 |
| INT_MIN   | $\leq -32767$      | 最小整型数     |
| INT_MAX   | $\geq +32767$      | 最大整型数     |
| UINT_MAX  | $\geq 65535$       | 最大无符号整型数  |
| LONG_MIN  | $\leq -2147483647$ | 最小长整型数    |
| LONG_MAX  | $\geq +2147483647$ | 最大长整型数    |
| ULONG_MAX | $\geq 4292967295$  | 最大无符号长整型数 |

<limits.h>中定义的宏在查看编译器是否支持特定大小的整数时十分方便。例如，如果要判断int类型是否可以用来存储像100 000一样大的数，可以使用下面的预处理指令：

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

如果int类型不适用，#error指令（>14.5.1节）会中止编译。

进一步讲，可以使用<limits.h>中的宏来帮助程序选择正确的类型定义。假设Quantity类型的变量必须可以存储像100 000一样大的整数。那么如果INT\_MAX大于100 000，我们就可以将Quantity定义为int；否则，则需要定义为long int：

```
#if INT_MAX >= 100000
typedef int Quantity;
#else
typedef long int Quantity;
#endif
```

### 23.3 <math.h>：数学计算

<math.h>中定义的函数包含下面5种类型：

- 三角函数。
- 双曲函数。
- 指数和对数函数。
- 幂函数。
- 就近取整函数、绝对值函数和取余函数

在深入讨论这些类型之前，先来简单了解一下<math.h>中的这些函数是如何处理错误的。

#### 23.3.1 错误

<math.h>中的函数对错误的处理方式与其他库函数不同。当发生错误时，<math.h>中的大多数函数会将一个错误代码存储到一个名字为errno的特定变量中（在<errno.h>（>24.2节）中）。此外，一旦函数的返回值大于double类型的最大取值，<math.h>中的函数会返回一个特殊的值，这个值由HUGE\_VAL宏定义（这个宏在<math.h>中定义）。HUGE\_VAL是double类型，

但不一定是一个普通的数。(IEEE浮点运算标准(>7.2节)定义了一个值叫“无穷”——这个值是HUGE\_VAL的一个合理的选择。)

- 定义域错误:** 函数的实参超出了函数的定义域。当定义域错误发生时, 函数的返回值是由实现定义的, 同时EDOM(“定义域错误”)会被存储到errno中。在一些<math.h>的实现中, 当定义域错误发生时, 函数会返回值NAN(“非数”)。NAN是在IEEE标准中定义的另一个特殊的值(与“无穷”类似)。
- 取值范围错误:** 函数的返回值超出了double类型的取值范围。如果返回值的绝对值过大(溢出), 函数会根据结果的符号返回正的或负的HUGE\_VAL。此外, 值ERANGE(“取值范围错误”)会被存储到errno中。如果返回值的绝对值太小(下溢出), 函数返回零; 一些实现可能也会将ERANGE存到errno中。

522

本节不讨论取余时可能发生的错误。在附录D中描述的函数会解释导致每种错误的情况。

### 23.3.2 三角函数

```
double cos(double x);
double sin(double x);
double atan(double x);
double atan2(double y, double x);
double cos(double x);
double sin(double x);
double atan(double x);
double tan(double x);
```

cos函数、sin函数和tan函数分别用来计算余弦、正弦和正切。假定PI被定义为3.14159265, 那么以PI/4为参数调用cos函数、sin函数和tan函数会产生如下的结果:

```
cos(PI/4) => 0.707107
sin(PI/4) => 0.707107
tan(PI/4) => 1.0
```

注意, 传递给cos函数、sin函数和tan函数的参数是以弧度表示的, 而不是以角度表示的。

acos函数、asin函数和atan函数分别用来计算反余弦、反正弦和反正切:

```
acos(1.0) => 0.0
asin(1.0) => 1.5708
atan(1.0) => 0.785398
```

对cos函数、的计算结果直接调用acos函数不一定会得到最初传递给cos函数的值, 因为acos函数始终返回一个0~π的值。asin函数与atan函数会返回-π/2~π/2的值。

atan2函数用来计算y/x的反正切值, 其中y是函数的第一个参数, x是第二个参数。atan2函数的返回值在-π~π。调用atan(x)与调用atan2(x, 1.0)等价。

### 23.3.3 双曲函数

```
double cosh(double x);
double sinh(double x);
double tanh(double x);
```

cosh函数、sinh函数和tanh函数分别用来计算双曲余弦、双曲正弦和双曲正切:

523

```
cosh(0.5) => 1.12763
sinh(0.5) => 0.521095
tanh(0.5) => 0.462117
```

传递给cosh函数、sinh函数和tanh函数的实参必须以弧度表示, 而不能以角度表示。

### 23.3.4 指数函数和对数函数

```
double exp(double x);
```

```
double frexp(double value, int *exp);
double ldexp(double x, int exp);
double log(double x);
double log10(double x);
double modf(double value, double *iptr);
```

`exp`函数返回e的 $x$ 次幂。

`exp(3.0) ⇒ 20.0855`

`log`函数与`exp`函数相反，它计算参数以e为底取对数的结果。`log10`计算“常用”对数（以10为底）的结果：

`log(20.0855) ⇒ 3.0`  
`log10(1000) ⇒ 3.0`

对于不以e为底或不以10为底的对数，计算起来也不复杂。例如，下面的函数对任意的 $x$ 和 $b$ ，计算以 $b$ 为底 $x$ 的对数：

```
double logb(double x, double b)
{
 return log(x) / log(b);
}
```

`modf`函数和`frexp`函数将一个`double`类型的值拆解为两部分。`modf`将它的第一个参数分为整数和小数部分，返回其中的小数部分，并将整数部分存入第二个参数所指向的变量中：

`modf(3.14159, &int_part) ⇒ 0.14159 (int_part被赋值为3.0)`

虽然`int_part`的类型必须为`double`，但我们始终都可以随后将它强制转换成`int`或`long int`。

`frexp`函数将浮点数拆成小数部分 $f$ 和指数部分 $n$ ，使得原始值等于 $f \times 2^n$ ，其中 $0.5 \leq f \leq 1$ 或 $f=0$ 。函数返回 $f$ ，并将 $n$ 存入第二个参数所指向的（整数）变量中：

`frexp(12.0, &exp) ⇒ .75 (exp被赋值为4)`  
`frexp(0.25, &exp) ⇒ 0.5 (exp被赋值为-1)`

`ldexp`函数会复原`frexp`产生的结果，将小数部分和指数部分组合成一个数：

`ldexp(.75, 4) ⇒ 12.0`  
`ldexp(0.5, -1) ⇒ 0.25`

一般而言，调用`ldexp(x, exp)`将返回 $x \times 2^{\exp}$ 。

### 23.3.5 幂函数

```
double pow(double x, double y);
double sqrt(double x);
```

`pow`函数计算第一个参数的幂，幂的次数由第二个参数指定：

`pow(3.0, 2.0) ⇒ 9.0`  
`pow(3.0, 0.5) ⇒ 1.73205`  
`pow(3.0, -3.0) ⇒ 0.037037`

`sqrt`函数计算平方根：

`sqrt(3.0) ⇒ 1.73205`

顺便提一下，由于通常`sqrt`函数的运行速度非常快，因此使用`sqrt`计算平方根比使用`pow`更好。

### 23.3.6 就近取整函数、绝对值函数和取余函数

```
double ceil(double x);
double fabs(double x);
double floor(double x);
double fmod(double x, double y);
```

`ceil`函数(`ceiling`)返回一个`double`类型的值，这个值是大于或等于其参数的最小整数。  
`floor`函数则返回小于或等于其参数的最大整数：

```
ceil(7.1) ⇒ 8.0
ceil(7.9) ⇒ 8.0
ceil(-7.1) ⇒ -7.0
ceil(-7.9) ⇒ -7.0
floor(7.1) ⇒ 7.0
floor(7.9) ⇒ 7.0
floor(-7.1) ⇒ -8.0
floor(-7.9) ⇒ -8.0
```

525

换言之，`ceil`“向上舍入”到最近的整数，`floor`“向下舍入”到最近的整数。没有一个标准库函数用来就近舍入到最近的整数，但我们可以简单地使用`ceil`函数和`floor`函数来实现：

```
double round(double x)
{
 return x < 0.0 ? ceil(x-0.5) : floor(x+0.5);
}
```

`fabs`函数计算参数的绝对值：

```
fabs(7.1) ⇒ 7.1
fabs(-7.1) ⇒ 7.1
```

`fmod`函数返回第一个参数除以第二个参数所得的余数：

```
fmod(5.5, 2.2) ⇒ 1.1
```

C语言不允许对%运算符使用浮点操作数，不过`fmod`函数足以用来替代%运算符。

## 23.4 <ctype.h>: 字符处理

<ctype.h>提供了两类函数：字符测试函数（如`isdigit`函数，用来检测一个字符是否是数字）和字符大小写转换函数（如`toupper`函数，用来将一个小写字母转换成大写字母）。

虽然C语言并不要求我们使用<ctype.h>中的函数来测试字符或进行大小写转换，但我们仍建议使用<ctype.h>中定义的函数来进行这类操作。第一，这些函数已经针对运行速度进行过优化（实际上，大多数都是用宏实现的）；第二，使用这些函数会使程序的可移植性更好，因为这些函数可以在任何字符集上运行；第三，当地点改变时（见25.1节），<ctype.h>中的函数会相应地调整其行为，使我们编写的程序可以正确地运行在世界上不同的地点。

<ctype.h>中定义的函数都以`int`类型作为参数，并返回一个`int`类型的值。当然通常都可以忽略这种细节，因为需要时C语言可以自动将`char`类型的参数转换为`int`类型，或将`int`类型的返回值转换成`char`类型。

526

### 23.4.1 字符测试函数

| 函数名                   | 功能             |
|-----------------------|----------------|
| <code>isalnum</code>  | 判断字符是否是字母或数字   |
| <code>isalpha</code>  | 判断字符是否是字母      |
| <code>isblank</code>  | 判断字符是否是空格或制表符  |
| <code>isdigit</code>  | 判断字符是否是数字      |
| <code>islower</code>  | 判断字符是否是小写字母    |
| <code>isupper</code>  | 判断字符是否是大写字母    |
| <code>isxdigit</code> | 判断字符是否是十六进制数字  |
| <code>isprint</code>  | 判断字符是否是可打印字符   |
| <code>iscntrl</code>  | 判断字符是否是控制字符    |
| <code>isgraph</code>  | 判断字符是否是图形字符    |
| <code>ispunct</code>  | 判断字符是否是标点符号    |
| <code>isascii</code>  | 判断字符是否是ASCII字符 |

根据参数是否符合某种特性，每个字符测试函数都会返回0或1。表23-7列出了每个函数所测试的属性：

表23-7 舍入模式

| 取 值         | 取值对应的舍入模式             |
|-------------|-----------------------|
| isalnum(c)  | c是否是字母或数字             |
| isalpha(c)  | c是否是字母                |
| iscntrl(c)  | c是否是控制字符 <sup>①</sup> |
| isdigit(c)  | c是否是十进制数字             |
| isgraph(c)  | c是否是可显示字符(除空格外)       |
| islower(c)  | c是否是小写字母              |
| isprint(c)  | c是否是可显示字符(包括空格)       |
| ispunct(c)  | c是否是标点符号 <sup>②</sup> |
| isspace(c)  | c是否是空白字符 <sup>③</sup> |
| isupper(c)  | c是否是大写字母              |
| isxdigit(c) | c是否是十六进制数字            |

① 在ASCII字符集中, 控制字符包括\0x00至\0x1f, 以及\0x7f。

② 标点符号包括所有可显示字符中除掉空格, 字母数字以外的其他字符。

③ 空白字符包括空格、换页符(\f)、换行符(\n)、回车符(\r)、横向制表符(\t)和纵向制表符(\v)。

### 23.4.2 程序: 测试字符测试函数

下面的程序通过将字符测试函数应用于字符串"azAZ0 !\t"中的字符, 来展示这些函数的作用。

#### tchrtest.c

```
/* Tests the character-testing functions */

#include <ctype.h>
#include <stdio.h>

#define TEST (f) printf (" %c ", f(*p) ? 'x' : ' ');

527 main ()
{
 char *p ;
 printf (" alnum cntrl graph print"
 " space xdigit\n"
 " alpha digit lower punct"
 " upper \ n") ;

 for (p = "azAZ0 !\t"; *p != '\0' ; p++){
 if (iscntrl(*p))
 printf("\x%02x:", *p) ;
 else
 printf(" %c:", *p) ;
 TEST (isalnum) ;
 TEST (isalpha) ;
 TEST (iscntrl) ;
 TEST (isdigit) ;
 TEST (isgraph) ;
 TEST (islower) ;
 TEST (isprint) ;
 TEST (ispunct) ;
 TEST (isspace) ;
 TEST (isupper) ;
 TEST (isxdigit)
 printf("in") ;
 }
 return 0 ;
}
```

程序产生的输出如下:

|       |       |       |       |       |        |
|-------|-------|-------|-------|-------|--------|
| alnum | cntrl | graph | print | space | xdigit |
| alpha | digit | lower | punct | upper |        |
| x     | x     | x     | x     | x     | x      |

```

z: x x x x x
A: x x x x x
Z: x x x x x
0: x x x x
: x x x
!: x x x
\x09: x

```

### 23.4.3 字符大小写转换函数

```

int tolower(int c);
int toupper(int c);

```

tolower函数返回与作为参数的字母相对应的小写字母，而toupper函数返回与作为参数的字母相对应的大写字母。对于这两个函数，如果所传参数不是字母，那么将返回原始字符，不加任何改变。

528

### 23.4.4 程序：测试大小写转换函数

下面的程序对字符串 "aA0!" 中的字符进行大小写转换。

```

tcasemap.c
/* Tests the case-mapping functions */

#include <ctype.h>
#include <stdio.h>

main()
{
 char *p;

 for (p = "aA0!"; *p != '\0'; p++) {
 printf("tolower('c') is '%c'; ", *p, tolower(*p));
 printf("toupper('c') is '%c'\n", *p, toupper(*p));
 }
 return 0;
}

```

程序产生的输出如下：

```

tolower('a') is 'a'; toupper('a') is 'A'
tolower('A') is 'a'; toupper('A') is 'A'
tolower('0') is '0'; toupper('0') is '0'
tolower('!') is '!'; toupper('!') is '!'

```

## 23.5 <string.h>: 字符串处理

第一次见到<string.h>是在13.5节，那一节讨论了最基本的字符串操作：strcpy（字符串复制）、strcat（字符串拼接）、strcmp（字符串比较）以及strlen（字符串长度计算）。接下来我们将看到，在<string.h>中还有许多其他字符串处理函数，以及一些对字符数组进行操作的函数。这些针对字符数组的函数不要求它们以空字符结尾。

<string.h>提供了5类函数：

- 复制函数，将字符从内存中的一处复制到另一处。
- 拼接函数，向字符串末尾追加字符。
- 比较函数，用于比较字符数组的函数。
- 搜索函数，在字符数组中搜索特定字符、字符组或字符串。
- 其他函数，初始化字符数组或计算字符串的长度。

下面我们来逐一讨论每一类函数。

529

### 23.5.1 复制函数

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char *s1, const char *s2, size_t n);
char *strncpy(char *s1, const char *s2, size_t n);
```

**Q&A**这4个复制函数将字符（字节）从内存的一处（源）移动到另一处（目的）。每个函数都要求第一个参数指向目的，第二个参数指向源。所有的复制函数都会返回第一个参数（即指向目的的指针）。

`memcpy`函数从源向目的复制n个字符，其中n是函数的第三个参数。如果源和目的之间有重叠，`memcpy`函数的行为是未定义的。`memmove`函数与`memcpy`函数类似，只是当源和目的重叠时也可以正常工作。

`strcpy`函数将一个以空字符结尾的字符串从源复制到目的。`strncpy`与`strcpy`类似，只是它不会复制多于n个字符，其中n是函数的第三参数。（如果n太小，`strncpy`可能无法复制结尾的空字符。）如果`strncpy`遇到源字符串中的空字符，`strncpy`会向目的字符串不断追加空字符，直到写满n个字符为止。与`memcpy`类似，`strcpy`和`strncpy`不保证当源和目的相重叠时可以正常工作。

下面的例子展示了所有的复制函数。注释中给出了哪些字符会被复制。

```
char source[] = {'h', 'o', 't', '\0', 't', 'e', 'a'};
char dest[7];

memcpy(dest, source, 3); /* h, o, t
 */
memcpy(dest, source, 4); /* h, o, t, \0
 */
memcpy(dest, source, 7); /* h, o, t, \0, t, e, a
 */

memmove(dest, source, 3); /* h, o, t
 */
memmove(dest, source, 4); /* h, o, t, \0
 */
memmove(dest, source, 7); /* h, o, t, \0, t, e, a
 */

strcpy(dest, source); /* h, o, t, \0
 */
strncpy(dest, source, 3); /* h, o, t
 */
strncpy(dest, source, 4); /* h, o, t, \0
 */
strncpy(dest, source, 7); /* h, o, t, \0, \0, \0, \0
 */
```

注意，`memcpy`、`memmove`和`strncpy`都不要求使用空字符结尾的字符串，它们对任意内存块都可以正常工作；然而`strcpy`函数则会持续复制字符，直到遇到一个空字符为止，因此`strcpy`仅用于以空字符结尾的字符串。

530

### 23.5.2 拼接函数

```
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
```

`strcat`函数将它的第二个参数追加到第一个参数的末尾。两个参数都必须是以空字符结尾的字符串。`strcat`函数会在拼接后的字符串末尾添加空字符。考虑下面的例子：

```
char str[7] = "tea";
strcat(str, "bag"); /* adds b, a, g, \0 to end of str */
```

字母b会覆盖字符a后面的空字符，因此现在`str`包含字符串“teabag”。`strcat`函数会返回它的第一个参数（指针）。

`strncat`函数与`strcat`函数基本一致，只是它的第三个参数会限制复制的字符的个数：

```
char str[7] = "tea";
strncat(str, "bag", 2); /* adds b, a, \0 to str */
```

```
strncat(str, "bag", 3); /* adds b, a, g, \0 to str */
strncat(str, "bag", 4); /* adds b, a, g, \0 to str */
```

正如上面例子所示, strnact函数会保证其结果字符串始终以空字符结尾。

### 23.5.3 比较函数

```
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

我们会将比较函数分为两组讨论。第一组中的函数 (memcmp函数、strcmp和strncmp函数) 比较两个字符数组。比较是按照计算机自身的排序顺序 (通常为ASCII) 对每个字符逐一进行的。第二组中的函数 (strcoll函数和strxfrm函数) 在需要考虑本地化 (>25.1节) 时使用。

memcmp函数、strcmp函数和strncmp函数有许多共同的特性。这三个函数都需要指向字符数组的指针作为参数, 然后用第一个字符数组中的字符逐一地与第二个字符数组中的字符进行比较。这三个函数都是在遇到第一个不匹配的字符时返回。另外, 这三个函数都是根据比较结束时第一个字符数组中的字符是小于、等于或大于第二个字符数组中的字符, 而相应地返回一个负整数、0或正整数。

这三个函数之间的差异在于如果数组相同, 何时停止比较。memcmp函数包含第三个参数n, n会用来限制参与比较的字符个数, 但memcmp函数不会关心空字符。strcmp函数没有对字符数设定限制, 因此会在其中任意一个字符数组中遇到空字符时停止比较。(因此, strcmp函数只能用于以空字符结尾的字符串。) strncmp结合了memcmp和strcmp, 当比较的字符数达到n个或在其中任意一个字符数组中遇到空字符时停止比较。

531

下面的例子解释了memcmp函数、strcmp函数和strncmp函数的上述特性:

```
char s1[] == {'b', 'i', 'g', '\0', 'c', 'a', 'r'};
char s2[] == {'b', 'i', 'g', '\0', 'c', 'a', 't'};

if (memcmp(s1, s2, 3) == 0) ... /* true */
if (memcmp(s1, s2, 4) == 0) ... /* true */
if (memcmp(s1, s2, 7) == 0) ... /* false */

if (strcmp(s1, s2) == 0) ... /* true */

if (strncmp(s1, s2, 3) == 0) ... /* true */
if (strncmp(s1, s2, 4) == 0) ... /* true */
if (strncmp(s1, s2, 7) == 0) ... /* true */
```

strcoll函数与strcmp函数类似, 但比较的结果依赖于当前的本地化设置 (通过调用setlocale函数 (>25.1.2节) 设定)。对于那些根据程序运行的地点不同而可能按不同方式比较的程序, strcoll函数会比较有用。

大多数情况下, strcoll都足够用来处理依赖本地化设置情况下的字符串比较。但有些时候, 我们可能需要多次进行比较 (strcoll的一个潜在问题是, 它不是很快), 或者需要改变本地化设置但不希望影响比较的结果。在这些情况下, strxfrm函数 (“字符串转换”) 可以用来代替strcoll使用。

strxfrm函数会对它的第二个参数 (一个字符串) 进行转换, 将转换的结果放在第一个参数所指向的字符串中。第三个参数用来限制向数组输出结果的字符个数。对带有两个转换后的字符串调用strcmp函数所产生的结果 (负、0或正) 与使用原始字符串调用strcoll函数的结果相同。

strxfrm函数返回转换后字符串的长度。因此strxfrm函数通常会被调用两次: 一次用于判断转换后字符串的长度, 一次来进行转换。下面是一个例子:

```
size_t len;
char *transformed;

len = strxfrm(NULL, original, 0);
transformed = malloc(len+1);
strxfrm(transformed, original, len);
```

532

### 23.5.4 搜索函数

```
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strupr(const char *s1, const char *s2);
char *strchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
```

strchr函数在字符串中搜索指定字符。下面的例子说明了如何使用strchr函数在字符串中搜索字母f：

```
char *p, str[] = "Form follows function.";

p = strchr(str, 'f'); /* finds first 'f' */
```

strchr函数会返回一个指针，这个指针指向str中出现的第一个f（单词follows中的f）。如果需要多次搜索字符也很简单，例如，可以使用下面的调用搜索第二个f（即单词function中的f）：

```
p = strchr(p+1, 'f'); /* find next 'f' */
```

memchr函数与strchr函数类似，但memchr函数会在搜索了指定数量的字符后停止搜索，而不是当遇到首个空字符时才停止。memchr函数的第三个参数用来限制搜索时需要检测的字符总数。当不希望对整个字符串进行搜索或搜索的内存块不是以空字符结尾时，memchr函数会十分有用。下面的例子用memchr函数在一个没有以空字符结尾的字符数组中进行搜索：

```
char *p, str[22] = "Form follows function.";

p = memchr(str, 'f', sizeof(str));
```

与strchr函数类似，memchr函数也会返回一个指针，指向该字符第一次出现的位置。如果没有找到所查找的字符，两个函数都会返回空指针。

strrchr函数与strchr类似，但会反向搜索字符：

```
char *p, str[] = "Form follows function.";

p = strrchr(str, 'f'); /* finds last 'f' */
```

在此例中，strrchr函数会首先找到字符串末尾的空字符，然后反向查找字母f（单词function中的f）。如果找不到指定的字符，strrchr函数也会返回空指针。

strupr函数比strchr函数更通用。它返回一个指针，该指针指向第一个实际参数中与第二个实参中任意一个字符匹配的最左边一个字符：

```
char *p, str[] = "Form follows function.";

p =strupr(str, "mn"); /* finds first 'm' or 'n' */
```

在此例中，p最终会指向单词Form中的字母m。与其他搜索函数一样，当找不到指定的字符时，函数会返回空指针。

strspn函数和strcspn函数与其他的搜索函数不同，它们会返回一个表示字符串中特定位置的整数(size\_t类型)。**Q&A**当给定一个需要搜索的字符串以及需要搜索的字符集时，strspn函数返回字符串中第一个不属于给定字符集中的字符的下标。对于同样的参数，strcspn函数返回第一个属于给定字符集中的字符的下标。下面是使用两个函数的例子：

```

size_t len;
char str[] = "Form follows function.";

len = strspn(str, "morF"); /* len = 4 */
len = strspn(str, "\t\n"); /* len = 0 */
len = strcspn(str, "morF"); /* len = 0 */
len = strcspn(str, "\t\n"); /* len = 4 */

```

strstr函数在第一个参数（字符串）中搜索第二个参数（也是字符串）。在下面的例子中，strstr函数搜索单词fun：

```

char *p, str[] = "Form follows function.";
p = strstr(str, "fun"); /* locates "fun" in str */

```

strstr函数返回指向在字符串中找到的第一处匹配子串的指针。如果找不到，则返回空指针。在上例的调用后，p会指向function中的字母f。

strtok函数是最复杂的搜索函数。它的目的是在一个字符串中搜索一个“记号”——就是一系列不包含特定分割符的字符。调用strtok(s1, s2)会在s1中搜索，从而查找一系列非空字符，这些字符中不包含s2中所指定的字符。在找到的记号后面的那个字符的位置，strtok函数会写入一个空字符来标记这个记号的末尾，然后返回一个指向这个记号的首字符的指针。这个过程可以持续进行，直到strtok函数返回一个空指针，表明找不到符合要求的记号。

对于strtok函数最有用的特点是可以稍后调用strtok函数在同一字符串中搜索其他的记号。调用strtok(NULL, s2)就可以继续上一次的strtok函数调用。和上一次调用一样，strtok函数会用一个空字符来标记新的记号的末尾，然后返回一个指向新记号的首字符的指针。这个过程可以持续进行，直到strtok函数返回一个空指针，表明找不到符合要求的记号。

为了更清楚地解释strtok函数工作的原理，我们来使用它从如下书写的日期中：

*month day year*

534

抽取出月、日和年。其中月与日之间，及日与年之间以空格或制表符分割。此外，空格或制表符前可能有逗号字符。假定开始时，字符串str有如下形式：

|     |  |   |   |   |   |   |  |  |   |   |   |   |   |   |   |    |
|-----|--|---|---|---|---|---|--|--|---|---|---|---|---|---|---|----|
| str |  | A | p | r | i | l |  |  | 2 | 8 | , | 1 | 9 | 9 | 8 | \0 |
|-----|--|---|---|---|---|---|--|--|---|---|---|---|---|---|---|----|

在调用后

`p = strtok(str, " \t";`

字符串str的形式如下：

|     |  |   |   |   |   |   |    |  |   |   |   |   |   |   |   |    |
|-----|--|---|---|---|---|---|----|--|---|---|---|---|---|---|---|----|
| str |  | A | p | r | i | l | \0 |  | 2 | 8 | , | 1 | 9 | 9 | 8 | \0 |
|-----|--|---|---|---|---|---|----|--|---|---|---|---|---|---|---|----|

p指向月的第一个字符，同时一个空字符标志着月的末尾。使用空指针作为strtok函数的第一个参数调用，会从上次结束的位置继续查找：

`p = strtok(NULL, " \t");`

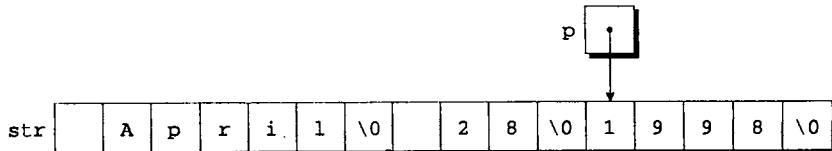
这个调用后，p指向日的第一个字符：

|     |  |   |   |   |   |   |    |  |   |   |    |   |   |   |   |    |
|-----|--|---|---|---|---|---|----|--|---|---|----|---|---|---|---|----|
| str |  | A | p | r | i | l | \0 |  | 2 | 8 | \0 | 1 | 9 | 9 | 8 | \0 |
|-----|--|---|---|---|---|---|----|--|---|---|----|---|---|---|---|----|

strtok函数的最后一个调用用来找到年：

```
p = strtok(NULL, " \t,");
```

这次调用后, str的形式如下所示:



当重复调用strtok函数将一个字符串分割成记号时, 对每次调用第二个参数并不需要保持一致。在我们的例子中, strtok函数的最后一次调用使用" \t,"替代了" \t"。

### 23.5.5 其他函数

```
void *memset(void *s, int c, size_t n);
size_t strlen(const char *s);
```

memset函数会将一个字符的多个副本存储到指定的内存区域。假设p指向一块N个字节的内存, 例如, 调用

```
memset(p, ' ', N);
```

会在区块内存的每个字节中存储空格。memset函数的一个用途是将数组全部初始化为0:

```
memset(a, 0, sizeof(a));
```

memset函数会返回它的第一个参数(指针)。

strlen函数返回字符串的长度, 字符串末尾的空字符不计算在内。请见13.5节中使用strlen函数调用的例子。

此外还有一个与字符串相关的函数——strerror函数(►24.2节), 我们会和<errno.h>一起讨论。

---

## 问与答

---

问: 为什么<string.h>中提供了那么多方法来做同一件事呢? 我们真的需要4个复制函数(memcpy、memmove、strcpy和strncpy)吗? (p.368)

答: 我们先看memcpy函数和strcpy函数, 使用这两个函数的目的是不同的: strcpy函数只会复制一个以空字符结尾的字符数组(也就是字符串); memcpy函数可以复制任意内存区域, 而不需要这样的终止符(例如整数数组)。

另外两个函数可以使我们在运行速度和安全性中做出选择。strncpy函数比strcpy函数更安全, 因为它限制了复制字符的个数。当然安全也是有代价的, 因此strncpy函数也会比strcpy函数稍慢一点。使用memmove函数也需要做出类似的抉择。memmove函数可以将字符从一块内存区域复制到另一块可能会重叠的内存区域中。在同样情况下, memcpy函数无法保证能够正常工作。然而如果我们确保没有重叠, memcpy函数很可能会比memmove函数要快一些。

问: 为什么strspn函数有这么一个奇怪的名字? (p.70)

答: 不将strspn函数的返回值理解为第一个不属于指定字符集合中的字符的下标, 而是将它的返回值理解为是其中所有属于指定字符集合的字符的最长“跨度”长度。

---

## 练习

---

### 23.3节

- 编写一个程序, 使用下面的公式求方程 $ax^2+bx+c=0$ 的根:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

程序需要提示a、b、和c的值，然后显示出x的两个解。（如果 $b^2 - 4ac$ 的值小于0，那么程序需要显示一条信息，指出根是虚数。）

2. 扩展round函数，使它可以将x舍入成小数点后n位。例如，调用round(3.14159, 3)会返回3.142。  
提示：将x乘以 $10^n$ ，舍入成最接近的整数，再除以 $10^n$ 。确保你的函数对正数x和负数x都可以正常工作。

### 23.4节

3. 使用isalpha函数和isalnum函数编写一个函数，用来检查一个字符串是否符合C语言标识符的语法（也就是说，它由字母、数字和下划线组成，并以字母或下划线开始）。
4. 编写一个程序，将文件从标准输入复制到标准输出，删除所有空行（仅包含空白字符的行）。
5. 编写一个程序，将文件从标准输入复制到标准输出，将每个单词的首字母大写。

### 23.5节

6. 对于下面列举的每种情况，指出使用memcpy, memmove, strcpy和strncpy中哪一个函数最好。  
假定所列举的行为都是由一个函数调用完成的。
  - (a) 将数组中的每个元素都“下移”一个位置，以便将第0个位置空出给新的元素。
  - (b) 通过将后面的所有字符都前移一个字节来删除以空字符结尾的字符串中的第一个字符。
  - (c) 将一个字符串复制到一个字符数组中，这个字符数组的大小可能不够存放整个字符串。如果数组太小，就将字符串截断，而且在字符数组的末尾不需要空字符作为结尾。
  - (d) 将一个数组变量的内容复制到另一个数组变量中。

7. 在23.5节中阐述了如何反复调用strchr函数在字符串中找到所有出现的指定字符。那么能否通过反复调用 strrchr函数反向找到所有出现的指定字符呢？

8. 使用strchr函数编写如下函数：

```
int numchar(const char *s, char ch);
```

函数numchar返回字符ch在字符串s中出现的次数。

9. 使用一个strchr函数调用来替换下面if语句中的测试条件：

```
if (ch == 'a' || ch == 'b' || ch == 'c') ...
```

537

10. 使用一个strstr函数调用来替换下面if语句中的测试条件：

```
if (strcmp(str, "foo") == 0 || strcmp(str, "bar") == 0 ||
 strcmp(str, "baz") == 0) ...
```

提示：将字符串字面量合并到一个字符串中，并使用一个特殊的字符分割它们。你的解决方案是否对str的内容有所依赖呢？

11. 编写一个程序，提示用户输入一系列单词，然后按相反的顺序显示出来。将输入按字符串的形式读入，然后使用 strtok函数将它们重新分割成单词。

12. 编写一个memset函数的调用，将一个以空字符结尾的字符串s的最后n个字符替换为'!'字符。

13. 许多<string.h>的版本提供了额外的（非标准的）函数，例如下面列出的一些函数。使用标准C的特性给出每一个函数的实现。

- (a) strdup(s) —— 返回一个指针，该指针指向通过调用malloc函数获得的内存中保存的s的一个副本。如果没有足够的内存可分配，则返回空指针。
- (b) stricmp(s1, s2) —— 与strcmp函数类似，但不考虑字母的大小写。
- (c) strlwr(s) —— 将s中的大写字母转换为小写字母，其他字符不变；返回s。
- (d) strrev(s) —— 反转字符串s中的字符顺序（空字符除外）；返回s。
- (e) strset(s, ch) —— 将s用ch的副本填充；返回s。

538

编写无错程序的方法有两种，但只有用第三种方法写的程序才行得通。

虽然学习C语言的学生所编写的程序在遇到异常输入时经常无法正常运行，但真正商业用途的程序却必须“非常强壮”——能够从错误中恢复正常而不至于崩溃。为了使程序非常强壮，需要我们能够预见程序执行时可能遇到的错误，包括对每个错误进行检测，并为错误一旦发生时提供一种合适的行为。

本章讲述了两种在程序中检测错误的方法：通过调用assert函数（24.1节）以及通过查询errno变量（24.2节）。24.3节讲解如何使程序检测并处理称为信号（signal）的条件，一些信号用于表示错误。最后，24.4节探讨了setjmp和longjmp机制，它们在相应处理错误时经常用到。

错误的检测和处理并不是C语言的强项。C语言对运行时错误以多种形式表示，而没有提供一种统一的方式。而且，在C语言程序中，必须由程序员将检测错误的代码编写在程序代码中。因此，很容易忽略一些可能发生的错误。一旦这些被略掉的错误中有某个错误发生，程序经常可以继续运行，虽然不是很好。C++C++语言对C语言的这一弱点进行了改进，提供了一种新的处理错误的方式——异常处理（exception handling）。

## 24.1 <assert.h>：诊断

```
void assert(int expression);
```

assert函数声明在<assert.h>中。它使程序可以监控自己的行为，并提早检测可能会发生的错误。539

虽然assert函数实际上是一个宏，但它是按照函数的使用方式设计的。assert函数有一个参数，这个参数必须是一种“断言”——一个我们认为在正常情况下一定为真的表达式。每次执行assert函数时，都会检查其参数的值。如果参数的值不为0，assert函数会显示一条信息（显示到stderr（标准错误流）（>22.1.1节）），并调用abort函数（>26.2.5节）终止程序执行。

例如，假定文件demo.c声明了一个长度为N的数组a。而我们担心demo.c程序中的语句

```
a[i] = 0;
```

可能会由于i不在0~N-1而导致程序失败。我们可以使用assert宏在给a[i]赋值前检查这种情况：

```
a[i]:
assert(0 <= i && i < N) ; /* checks subscript first */
a[i] = 0; /* now does the assignment */
```

如果i的值小于0或者大于等于N，程序在输出类似下面的消息后会终止：

```
Assertion failed: 0 <= i && i < N, file DEMO.C, line 109
```

标准C并不要求显示的消息和上面的格式完全一样。但是，标准C要求在显示的消息中指明传递给assert函数的参数（以文本格式）、包含assert调用的文件名以及assert调用所在的行号。

assert有一个缺点：因为它引入了额外的检查，因此会增加程序的运行时间。偶尔使用一次assert可能对程序的运行速度没有很大影响；但在实时程序中，这种对运行时间的增加可能是无法接受的。因此，许多程序在测试过程中会使用assert调用，但当程序最终完成时就会禁止assert调用。要禁止assert调用很容易，只需要在包含<assert.h>之前定义宏NDEBUG即可：

```
#define NDEBUG
#include <assert.h>
```

NDEBUG宏的值不重要，只要定义了NDEBUG宏即可。一旦之后程序又有错误发生，可以去掉NDEBUG宏的定义来重新起用assert调用。



不要在assert调用中使用有副作用的表达式，或有副作用的函数调用。一旦禁止了assert调用，这些表达式将不再会被计算。考虑下面的例子：

```
assert((p = malloc(n+1) != NULL);
```

一旦定义了NDEBUG，assert调用会被忽略并且malloc不会被调用。

540

## 24.2 <errno.h>: 错误

标准库中的一些函数通过向<errno.h>中声明的errno变量存储一个错误代码（一个正整数）来表示有错误发生。（error可能实际上是个宏。如果确实是宏，C语言标准要求它表示左值（>4.2.2节），以便和变量一样使用。）大部分使用errno变量的函数集中在<math.h>，但也有一些在标准库的其他部分。

假设我们需要使用一个库函数，该库函数通过给errno赋值来产生程序运行出错的信号。在调用这个函数之后，我们可以检查errno的值是否为零。如果不为零，则表示在函数调用过程中有错误发生。举例来说，假如需要检查sqrt函数（求平方根）（>23.3.5节）的调用是否出错，可以使用类似下面的代码：

```
errno = 0;
y = sqrt(x);
if (errno != 0) {
 fprintf(stderr, "sqrt error; program terminated. \n");
 exit(EXIT_FAILURE);
}
```

对于像sqrt这类可能会改变errno值的函数，在调用前将errno置零非常重要。虽然在程序刚开始运行时errno的值为零，但有可能在随后的函数调用中已经被改动了。库函数不会将errno清零，这是程序责任。

**Q&A** 当错误发生时，向errno中存储的值通常是EDOM或ERANGE。（这两个宏都定义在<errno.h>中。）这两个值分别代表两种在一数学函数调用时可能发生的错误：

- **定义域错误(EDOM)**: 传递给函数的一个参数不属于函数的定义域。例如用负数作为sqrt的参数就会导致一个定义域错误。
- **取值范围错误(ERANGE)**: 函数的返回值太大，无法用double类型的值表示。例如，用1000作为exp函数（>23.3.4节）的参数就经常会导致一个取值范围错误，因为 $e^{1000}$ 太大以致无法在大多数计算机上用double类型表示。

一些函数可能会导致这两种错误，我们可以用errno分别与EDOM和ERANGE比较后，确定究竟发生了哪种错误。

### perror函数和strerror函数

```
void perror(const char *s);
char *strerror(int errnum);
```

541

当库函数向`errno`存储了一个非零值时，我们可能会希望显示一条描述这种错误的信息。一种实现方式是调用`perror`函数（声明在`<stdio.h>`中），它会按如下顺序显示以下信息：(1) 调用`perror`的参数，(2) 一个分号，(3) 一个空格，(4) 一条出错消息，消息的内容根据`errno`的值决定，(5) 一个换行符。`perror`函数会输出到`stderr`（▶22.1.1节），而不是标准输出。

下面的代码是一个使用`perror`的例子：

```
errno = 0;
y = sqrt(x);
if (errno != 0) {
 perror("sqrt error");
 exit(EXIT_FAILURE);
}
```

`perror`函数在`sqrt error`后所显示的出错消息是由实现定义的。下面是一种可能的形式：

```
sqrt error: Math argument
```

这里我们假定`Math argument`是与`EDOM`错误相对应的消息。而`ERANGE`错误则通常会对应于另一条消息，例如`Result too large`。

`strerror`函数定义在`<string.h>`中，它与`perror`关系紧密。当以错误代码调用`strerror`时，函数会返回一个指针，它指向一个描述这种错误的字符串。例如，调用

```
puts(strerror(EDOM));
```

可能会显示

```
Math argument
```

如果给`strerror`函数传递`errno`作为它的参数，那么函数`perror`所显示的出错消息与`strerror`所返回的信息是相同的。

## 24.3 <signal.h>: 信号处理

`<signal.h>`提供了处理异常情况工具，即信号（signal）。信号有两种类型：运行时错误（例如除以0）和程序以外导致的事件。例如，许多操作系统都允许用户中断或终止运行的程序，在C语言中这些事件作为信号。当有错误或外部事件发生时，我们称产生了一个信号。大多数信号是异步的：它们可以在程序执行过程中的任意时刻发生，而不仅是在程序员所知道的特定时刻发生。

542

由于信号可能会在任何意想不到的时刻发生，因此必须用一种唯一的方式来统一处理它们。

### 24.3.1 信号宏

`<signal.h>`定义了一系列的宏，用于表示不同的信号。**Q&A**表24-1列出这些宏以及它们的含义。C语言的实现可以提供更多的信号宏，只要宏的名字以`SIG`开头并随其后使用大写字母组成。

表24-1 信号

| 宏名                   | 含义                                 |
|----------------------|------------------------------------|
| <code>SIGABRT</code> | 异常终止（可能由于调用 <code>abort</code> 导致） |
| <code>SIGFPE</code>  | 在数学运算中发生错误（可能是除以0或溢出）              |
| <code>SIGILL</code>  | 非法指令                               |
| <code>SIGINT</code>  | 中断                                 |
| <code>SIGSEGV</code> | 非法存储访问                             |
| <code>SIGTERM</code> | 终止请求                               |

C标准并不要求表24-1中列出的信号都自动发生，因为对于某个特定的计算机或操作系统，不是所有的信号都有意义。大多数C语言的实现都至少支持其中的一部分。

### 24.3.2 signal 函数

```
void (*signal(int sig, void (*func)(int)) (int));
```

在<signal.h>中最重要的函数就是signal函数，它会安装一个信号处理函数，以便将来给定的信号发生时使用。signal函数的使用比它的原型看起来要简单得多。第一个参数是特定信号的代码，第二个参数是一个指向函数的指针，这个函数就是当信号发生时用来处理信号的函数。例如，下面的signal函数调用对SIGINT信号安装了一个处理函数：

```
signal(SIGINT, handler);
```

handler就是信号处理函数的函数名。一旦随后在程序执行过程中出现了SIGINT信号，handler函数就会自动被调用。

每个信号处理函数都必须有一个int类型的参数。当一个特定的信号出现并调用相应的处理函数时，信号的代码会作为参数传递给处理函数。知道是哪种信号导致了处理函数被调用是十分有用的，尤其是，它允许我们对多个信号使用同一个处理函数。

信号处理函数几乎可以做所有它想做的事。这可能包含忽略该信号、执行一些错误修复或终止程序。然而，除非信号是由调用abort函数或raise函数引发的，否则信号处理函数不应该调用任何库函数，或试图使用一个静态存储期限的变量。

一旦信号处理函数返回，程序会从信号发生点恢复并继续执行。但是，有一些特殊情况。如果信号是SIGABRT，当处理函数返回时程序会终止（异常地）。如果信号是SIGFPE，那么处理函数返回的结果是未定义的。（也就是说，不要用它。）

虽然signal函数有返回值，但经常被忽略。返回值是指向对于指定信号的前一个处理函数的指针。如果需要，可以将它保存在变量中。特别是，如果我们打算恢复原来的处理函数，那么就需要保留signal函数的返回值：

```
void (*orig_handler)(int); /* function pointer */
orig_handler = signal(SIGINT, handler);
```

上面的调用将handler函数设置为SIGINT的处理函数，并将指向原始的处理函数的指针保存在变量orig\_handler中。如果要恢复原来的处理函数，我们需要使用下面的代码：

```
signal(SIGINT, orig_handler); /* restores original handler */
```

### 24.3.3 预定义的信号处理函数

除了编写我们自己的信号处理函数，我们还可以选择使用<signal.h>提供的预定义的处理函数。有两个这样的函数，每个都是用宏表示的：

- **SIG\_DFL**。函数SIG\_DFL按“默认”方式处理信号。可以使用这样的调用安装SIG\_DEF

```
signal(SIGINT, SIG_DFL); /* use default handler */
```

调用SIG\_DFL的结果是由实现定义的，但大多数情况下会导致程序终止。

- **SIG\_IGN**。调用

```
signal(SIGINT, SIG_IGN); /* ignore SIGINT signal */
```

指明随后当信号SIGINT发生时，忽略该信号。

除了SIG\_DFL和SIG\_IGN，<signal.h>可能还会提供其他的信号处理函数其函数名必须是以SIG\_开头并随其后使用大写字母组成。当程序刚开始执行时，根据不同的实现，每个信号的处理函数都会被初始化为SIG\_DFL或SIG\_IGN。

<signal.h>还定义了另一个宏——SIG\_ERR它看起来像是个信号处理函数。实际上，

SIG\_ERR根本不是处理函数，它是用来在安装处理函数时检测是否发生错误的宏。如果一个signal调用失败（即能不对所指定的信号安装处理函数），就会返回SIG\_ERR并在errno中存入一个正值。因此，为了测试signal调用是否失败，可以使用如下代码：

```
544 if (signal(SIGINT, handler) == SIG_ERR) {
 /* error; can't install handler for SIGINT */
}
```

在整个信号处理机制中，有一个特殊的技巧：如果信号是由处理这个信号的函数引发的会怎样呢？为了避免无限递归，C语言要求——除了SIGILL以外，当一个信号的处理函数被调用时，该信号对应的处理函数要被重置为SIG\_DFL（默认处理函数）或以其他方式加以封锁。（我们无法控制这一过程，因为一切都是在后台执行的。）



信号处理完之后，除非处理函数被重新安装，否则该信号不会被同一个函数处理两遍。Q&A当然，一种实现方法是在处理函数返回前调用signal函数。

### 24.3.4 raise 函数

```
int raise(int sig);
```

虽然通常信号都是自然产生的，但有时候如果程序可以触发信号，就会非常方便。raise函数就可以实现这一目的，而且它也是包含在<signal.h>中的函数。raise函数的参数指定所描述信号的代码：

```
raise(SIGABRT); /* raises the SIGABRT signal */
```

raise函数的返回值可以用来测试调用是否成功：0代表成功，非0则代表失败。

### 24.3.5 程序：测试信号

下面的程序说明了如何使用信号。首先，给SIGILL信号安装了一个惯用的处理函数（并小心地保存了原先的处理函数），然后调用raise\_sig产生一个信号；其次，程序将SIG\_IGN设置为SIGILL的处理函数并再次调用raise\_sig；最后，它将信号SIGILL原先的处理函数重新安装，并最后调用一次raise\_sig。

```
545 signal.c
/* Tests signals */

#include <signal.h>
#include <stdio.h>

void handler(int sig);
void raise_sig(void);

main()
{
 void (*orig_handler)(int);

 printf("Installing handler for signal %d\n", SIGILL);
 orig_handler = signal(SIGILL, handler);
 raise_sig();

 printf("Changing handler to SIG_IGN\n");
 signal(SIGILL, SIG_IGN);
 raise_sig();

 printf("Restoring original handler\n");
 signal(SIGILL, orig_handler);
 raise_sig();
}
```

```

printf("Program terminates normally\n");
return 0;
}

void handler(int sig)
{
 printf("Handler called for signal %d\n", sig);
}

void raise_sig(void)
{
 raise(SIGILL);
}

```

当然，调用raise并不需要在单独的函数中。这里定义raise\_sig函数只是为了说明一点：无论信号是从哪里发出的（无论是在main函数中还是在其他函数中）它都会被最近安装的处理函数捕获。

由于在C标准中有一大部分信号处理机制是未定义的，因此上述程序的输出可能会有所不同。下面是一种可能的输出形式：

```

Installing handler for signal 4
Handler called for signal 4
Changing handler to SIG_IGN
Restoring original handler

```

从这个输出的结果中，我们看到SIGILL的值为4，而且最初SIGILL的处理函数一定是SIG\_DFL。（如果是SIG\_IGN，我们应该会看到信息Program terminates normally）最后，那么可以注意到SIG\_DFL会导致程序终止，但不会显示出错误消息。

## 24.4 <setjmp.h>: 非局部跳转

```

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);

```

546

通常情况下，函数会返回到它被调用的位置。我们无法使用goto语句是它转到其他地方，因为goto只能跳转到同一函数内的标号处。但是<setjmp.h>可以使一个函数直接跳转到另一个函数，而不需要返回。

在<setjmp.h>中最重要的内容就是setjmp宏和longjmp函数。setjmp宏“标记”程序中的一个位置；随后可以使用longjmp跳转到该位置。虽然这一强大的机制可以有多种潜在的用途，它主要被用于错误处理。

如果要为将来的跳转标记一个位置，可以调用setjmp宏，调用的参数是一个jmp\_buf类型的变量（同样定义在<setjmp.h>中）。Q&A setjmp宏会将当前“环境”（包括一个，指向setjmp宏自身被调用的位置的指针）保存到变量中以便随后可以在调用longjmp函数时使用，然后返回0。

如果要返回setjmp宏所标记的位置可以使用longjmp函数，调用的参数是我们在调用setjmp宏时使用的同一个jmp\_buf类型的变量。longjmp函数会首先根据jmp\_buf变量的内容恢复当前环境，然后从setjmp宏调用中返回——这是最难以理解的。这次setjmp宏的返回值是val，就是调用longjmp函数时的第二个参数。（但是如果val的值为0，那么setjmp宏会返回1。）



一定要确保作为longjmp函数的参数已经被setjmp初始化了。否则，调用longjmp会导致未定义的行为。（程序很可能崩溃。）

总而言之，setjmp会在第一次调用时返回0；随后，longjmp将控制权重新转给最初的setjmp

宏调用，而setjmp在这次调用时会返回一个非零值。明白了吗？看来我们可能需要个例子...

### 程序：测试 setjmp 和 longjmp

下面的程序使用setjmp宏在main函数中标记一个位置；然后函数f2通过调用longjmp函数返回到这个位置。

```
tsetjmp.c
/* Tests setjmp/longjmp */

#include <setjmp.h>
#include <stdio.h>

static jmp_buf env;

void f1(void);
void f2(void);

main()
{
 int ret;

 ret = setjmp(env);
 printf("setjmp returned %d\n", ret);
 if (ret != 0) {
 printf("Program terminates: longjmp called\n");
 return 0;
 }
 f1();
 printf("Program terminates normally\n");
 return 0;
}

void f1(void)
{
 printf("f1 begins\n");
 f2();
 printf("f1 returns\n");
}

void f2(void)
{
 printf("f2 begins\n");
 longjmp(env, 1);
 printf("f2 returns\n");
}
```

程序的输出如下：

```
setjmp returned 0
f1 begins
f2 begins
setjmp returned 1
Program terminates:longjmp called
```

setjmp宏的最初调用返回0，因此main函数会调用f1。接着，f1调用f2，f2使用longjmp函数将控制权重新转移给main函数，而不是返回到f1。当longjmp函数被执行时，控制权重新回到setjmp宏调用。这时，setjmp宏返回1（就是在longjmp函数调用时所指定的值）。

## 问与答

问：我使用的<errno.h>版本中除了EDOM和ERANGE以外，还定义了其他的宏。这是合法的吗？(p.375)

答：是合法的。C标准允许使用宏表示其他错误条件，只要宏的名字以字母E开头并随其后使用数字或大

写字母组成。

问：一些表示信号的宏的名字含义比较模糊，比如SIGFPE和SIGSEGV。这些名字是如何得来的呢？(p.376)

答：信号的名字可以追溯到早期的C语言编译器，它们运行在DEC PDP-11计算机上。PDP-11的硬件可以检测一些错误，诸如“Floating Point Exception”和“Segmentation Violation”。 548

\*问：我注意到在<signal.h>中有一个叫sig\_atomic\_t的类型。它的作用是什么？

答：sig\_atomic\_t是一个整数类型。按照C标准，它可以“作为一个基本元素”使用。换言之，CPU可以使用一条机器指令从内存中获取它的值或存储它的值，而不用两条甚至更多的机器指令。sig\_atomic\_t通常被定义为int，因为大多数CPU都可以只用一条指令装载或存储一个整数。

通常，一个信号处理函数不应该访问有静态存储期限的变量。但是C标准允许一种例外的情况：信号处理函数可以向sig\_atomic\_t类型的变量存入一个值，只要该变量被声明为volatile（类型限定符）。(►20.3.5节)想要理解这条奇特规则的原因，可以考虑一下如果信号处理函数试图修改一个比sig\_atomic\_t类型大的变量会发生什么情况？如果程序在信号发生前从内存中获取了这个变量的一部分，然后在信号处理后获取余下的部分，那么程序可能以一个无用的值终止。sig\_atomic\_t类型的变量只需要用一条语句获取，而且volatile类型的变量每次使用时必须重新获取，因此上述问题就不会发生了。

问：如果信号处理函数不支持调用库函数，那么信号处理函数又怎么能调用signal函数来重新安装它自己呢？(p.378)

答：C标准允许这一例外发生。信号处理函数可以合法地调用singal函数，只要第一个参数是当前正在被处理的信号就可以。

问：程序tsignal在信号处理函数内调用了printf函数。这不是非法的吗？

答：这是C标准所允许的另一个例外：如果信号处理函数是由raise或abort调用的，那么就可以调用库函数。

问：setjmp会如何修改传递给它的参数呢？C语言不是始终以值的形式传递参数吗？(p.379)

答：C标准要求jmp\_buf必须是一个数组类型，因此传递给setjmp的实际上是一个指针。

问：我在使用setjmp时，程序有时无法通过编译。这是什么问题？

答：按照标准C，只有两种使用setjmp的方式是合法的：

- 作为表达式语句（可能会强制转换成void）。
- 作为if、switch、while、do或for语句中控制表达式的一部分。整个控制表达式必须符合下面的形式之一。（constexp是一个计算结果为整数的常量表达式，并且op是关系或判等运算符。）

```
setjmp(...)
!setjmp(...)
constexp op setjmp(...)
setjmp(...) op constexpop
```

549

一些编译器允许不符合这些规则的setjmp调用。但是如果遵守这些规则，程序就不是可移植的。

问：调用longjmp函数后，程序中变量的值是什么？

答：大部分变量的值保留了longjmp函数被调用时的值。然而，包含setjmp宏的函数中的自动变量的值是不确定的，除非该变量被声明为volatile或者在执行setjmp后没有被修改过。

问：在信号处理函数里调用longjmp函数合法吗？

答：是合法的，只要信号处理函数的调用不是由于在信号处理函数执行过程中触发的信号。

## 练习

### 24.1节

1. (a) 断言可以用来检测两种问题：(1) 如果程序正确执行就不应该发生的问题；(2) 超出程序控制范围之外的问题。请解释为什么assert更适用于第一类问题？

(b) 请举出3个超出程序控制范围之外的问题的例子。

### 24.2节

2. (a) 编写一个名为try\_math\_fcn的“包装”函数，用来调用数学函数（假定有一个double类型的参数，并返回一个double类型的值），然后检测调用是否成功。下面是使用try\_math\_fnc函数的例子：

```
y = try_math_fnc(sqrt, x, "Error in call of sqrt");
```

如果调用sqrt(x)成功，try\_math\_fnc返回sqrt函数计算的结果。如果调用失败，try\_math\_fnc需要调用 perror 显示消息 Error in call of sqrt，然后调用 exit 函数终止程序。

(b) 编写一个与try\_math\_fnc具有相同的效果的宏，但是要求使用函数的名字来构造出错消息：

```
y = TRY_MATH_FNC(sqrt, x);
```

如果调用sqrt失败，显示的出错消息应该是“Error in call of sqrt”。提示：让TRY\_MATH\_FNC 调用try\_math\_fnc。

### 24.3节

3. 给SIGINT编写一个信号处理函数，用来记录它被调用了多少次。要求处理函数必须忽略前两次发生的信号，并在第三次发生时终止程序（通过调用exit）。

### 24.4节

4. 在invent程序中（16.3节），main函数中用一个for循环来提示用户输入一个操作代码，读入代码，然后根据代码调用insert、search、update或print。以这种方法在main函数中加入一个setjmp 调用，要求使随后的longjmp调用会返回到for循环。（在调用longjmp函数后，用户会被提示输入一个操作码，随后程序正常执行。）setjmp宏需要一个jmp\_buf类型的变量，这个变量应该在哪儿声明呢？

# 国际化特性

如果您的计算机说英语，那么它可能产自日本。

在最初设计时，C语言并不十分适合在多个国家使用。经典C假定字符都是单字节的，并且所有计算机都识别字符#、[、\、]、^、{、}和~，因为这些字符都需要在C程序中用到。遗憾的是这些假定并不是在世界的任何地方都适用。在20世纪80年代创造标准C的专家意识到了将C语言国际化的重要性。本章描述他们给C语言添加的特性和函数库，而这些给全世界的程序员带来了帮助。

<locale.h>（25.1节）提供了允许程序员针对特定的“地区”（可能是国家、洲或省或者一种特定的文化）删减程序行为的函数。多字节字符和宽字符（25.2节）使程序可以工作在更大的字符集上，例如亚洲国家的字符集。三字符序列（25.3节）使我们可以在一些不支持某些C语言编程中常用字符的机器上编写程序。

在1994年C语言对整个国际社会的重要性得到了强调，在这一年针对ISO C标准的修正草案1批准通过。这一提案提出了为编写国际化程序增加的额外库，包括<iso646.h>、<wctype.h>以及<wchar.h>。由于这些内容还没有被广泛使用，所以本书不准备再讨论修正草案1的细节了。如果想了解更多细节，可以参考Harbison和Steele的*C: A Reference Manual*（第4版）（Englewood Cliffs, N.J.: Prentice-Hall, 1995）。

## 25.1 <locale.h>: 本地化

<locale.h>提供的函数用于控制标准库中对于不同的地点会不一样的部分。地区通常是一个国家，但并不需要一定如此。例如，一个国家的不同区域也可能被作为单独的地区来对待。地区甚至可以代表同一区域的不同文化。

[551]

在标准库中，依赖地区的部分包括：

- **数值的格式。**例如在一些地区，小数点是一个圆点（297.48），而在另一些地方则是逗号（297,48）。
- **货币的格式。**例如，不同国家的货币符号不同。
- **字符集。**字符集通常依赖于特定地区的语言。亚洲国家通常比西方国家需要更大的字符集。
- **日期和时间的表示形式。**例如，一些地方习惯在写日期时先写月（8/24/97），而另一些地方习惯先写日（24/8/97）。

### 25.1.1 类别

通过修改地区，程序可以改变它的行为来适应世界的不同区域。但地区改动可能会影响库的许多方面，其中一部分可能是我们不希望改变的。幸好，我们不需要同时对库的所有部分进行改变。实际上，可以使用下列宏中的一种来指定一个类型：

- LC\_COLLATE。影响两个字符串比较函数（`strcoll`和`strxfrm`）的行为。（两个函数都

声明在<string.h> (>23.5节) 中。)

- LC\_CTYPE。影响<ctype.h> (>23.4节) 中函数 (除了isdigit和isxdigit) 的行为。同时还影响<stdlib.h>中的多字节函数 (>25.2.1节)。
- LC\_MONETARY。影响由localeconv函数返回的货币格式信息。不影响任何库函数的行为。
- LC\_NUMERIC。影响格式化输入/输出函数 (例如printf和scanf) 使用的小数点字符以及<stdlib.h>中的字符串转换函数 (atof和strtod) (>26.2.1节)，还会影响localeconv函数返回的非货币格式信息。
- LC\_TIME。影响strftime函数 (在<time.h>中声明) (>26.3.2节) 的行为，该函数将时间转换成字符串。

C语言的实现提供了其他类型并且定义了上面未列出的以LC\_开头的宏。

### 25.1.2 setlocale 函数

552

```
char *setlocale(int category, const char *locale);
```

setlocale函数修改当前的地点，可以是针对一个类型的，也可以是针对所有类型的。如果setlocale调用的第一个参数是LC\_COLLATE、LC\_CTYPE、LC\_MONETARY、LC\_NUMERIC或LC\_TIME之一，那么改变就只影响一个类型。如果第一个参数是LC\_ALL，调用就会影响所有类型。C语言标准对第二个参数仅定义了两种可能值：“C”和“”。其他的地区可以针对不同的C实现定义。

在任意程序执行开始时，都会隐含执行调用

```
setlocale(LC_ALL, "C");
```

当地点设置为“C”时，库函数按正常方式执行，小数点是一个句点。

如果在程序运行起来后想改变地点，就需要显式调用setlocale函数。用“”作为第二个参数调用setlocale函数可以切换到本地模式 (native locale)。这种模式下程序会适应本地的环境。C语言标准并没有定义切换到本地模式的具体影响。一些setlocale函数实现的会检查当前的运行环境 (与getenv函数 (>26.2.5节) 的方式一样)，查找特定名字 (可能是与表示类型的宏同名) 的环境变量。而另一些实现根本什么都不做。(C语言标准并没有要求setlocale有什么特定的作用。当然，如果库中的setlocale什么都不做，那么这个库在世界的一些地区它可能不会卖得很好。)

对于除“C”和“”以外的其他地点我们可能无法提供更多介绍，因为它们在不同的编译器之间可能有很大的差异。一些编译器可能不会提供任何其他地点。另一些编译器则可能会提供名为“Germany”的设置。一种常用的编译器使用类似“en\_GB.WIN1252”的复杂字符串作为地点。其中en指定语言 (English)，GB是国家 (Great Britain)，WIN1252是字符集 (Windows多语言字符集)。

当setlocale函数调用成功时，它会返回一个指向字符串的指针，这个字符串与新地点的类型相关联。(例如，这个字符串可能就是地点名字自身。) 如果调用失败，setlocale函数返回空指针。

setlocale函数也可以当作搜索函数使用。如果第二个参数是空指针，setlocale函数会返回一个指向字符串的指针，这个字符串与当前地区类型的设置相关联。这一特性在将第一个参数设为LC\_ALL时特别有用，因为这时可以获取对应于所有类型的当前设置。**Q&A** setlocale函数返回的字符串可以 (通过复制到变量中) 被保存起来以便以后调用setlocale函数时使用。

### 25.1.3 localeconv 函数

```
struct lconv *localeconv(void);
```

虽然可以通过调用setlocale函数来获取当前地区的信息，但是setlocale函数可能不是以最有效的形式返回信息的。为了找到关于当前地区的有效说明信息 (小数点字符是什么？货

553

币符号是什么？），就需要声明在<locale.h>中的另一个唯一的函数——`localeconv`函数。

`localeconv`函数返回指向`struct lconv`类型结构的指针，且指向的结构包含当前地区的详细信息。此结构具有静态存储期限，而且稍后通过`localeconv`函数或者`setlocale`函数调用还可以对此结构进行修改。请一定要确信在上述函数之一擦除结构信息之前，已经从`lconv`结构中摘取了需要的信息。

`lconv`结构中的一些成员具有`char*`类型，而另一些成员则具有`char`类型。表25-1列出了结构中所有`char*`类型的成员，其中前3个成员用来处理非货币型数值的格式，而其他成员则处理货币型数值。此表还说明了每个成员在“C”地区中（默认情况下）的值，其中“”意味着“无效的”。

表25-1 `lconv`结构的`char*`类型的成员

|       | 名 称                            | 在“C”地区中的值 | 描 述                 |
|-------|--------------------------------|-----------|---------------------|
| 非货币类的 | <code>decimal_point</code>     | “.”       | 十进制小数点字符            |
|       | <code>thousands_sep</code>     | “”        | 在十进制小数点前，用来分隔数字组的字符 |
|       | <code>grouping</code>          | “”        | 数字组的大小尺寸            |
| 货币类的  | <code>int_curr_symbol</code>   | “”        | 国际货币符号 <sup>①</sup> |
|       | <code>currency_symbol</code>   | “”        | 区域货币符号              |
|       | <code>mon_decimal_point</code> | “.”       | 十进制小数点字符            |
|       | <code>mon_thousands_sep</code> | “”        | 在十进制小数点前，用来分隔数字组的字符 |
|       | <code>mon_grouping</code>      | “”        | 数字组的大小尺寸            |
|       | <code>positive_sign</code>     | “”        | 用来说明非负值的字符串         |
|       | <code>negative_sign</code>     | “”        | 用来说明负值的字符串          |

① 分隔符（常常是空格或者句点）后边跟着3个字母的缩写。例如，意大利、荷兰、挪威以及瑞士的国际货币符号分别是“ITL.”、“NLG.”、“NOK”和“CHF”。

这里需要特别说明一下成员`grouping`和成员`mon_grouping`。在这两个字符串中的每个字符都说明了每组数字的大小。（分组工作是从十进制小数点开始自右向左进行的。）`CHAR_MAX`的值说明没有进一步要执行的分组操作了；0说明前面的元素应该用于其余的数字。例如，字符串“\3”（\3的后边跟着\0）说明第一组应该有3个数字，然后所有其他数字也应该以3分组中。

表25-2列出了`lconv`结构中的`char`类型成员，并且说明了在“C”地区中每个成员的值；其中`CHAR_MAX`的值意味着“无效”。表25-2中的所有成员都必须处理的是货币型值的格式。表25-3说明了如何解释成员`p_sign_posn`和成员`n_sign_posn`的值。

554

表25-2 `lconv`结构的`char`类型的成员

| 名 称                          | 在“C”地区中的值             | 描 述                                                                                 |
|------------------------------|-----------------------|-------------------------------------------------------------------------------------|
| <code>int_frac_digits</code> | <code>CHAR_MAX</code> | 十进制小数点后的数字个数（国际格式）                                                                  |
| <code>frac_digits</code>     | <code>CHAR_MAX</code> | 十进制小数点后的数字个数（区域格式）                                                                  |
| <code>p_cs_precedes</code>   | <code>CHAR_MAX</code> | 如果 <code>currency_symbol</code> 先于非负值，则为1；如果 <code>currency_symbol</code> 继数值之后，则为0 |
| <code>p_sep_by_space</code>  | <code>CHAR_MAX</code> | 如果 <code>currency_symbol</code> 是用空格来分隔非负值，则为1；否则为0                                 |
| <code>n_cs_precedes</code>   | <code>CHAR_MAX</code> | 如果 <code>currency_symbol</code> 先于负值，则为1；如果 <code>currency_symbol</code> 继数值之后，则为0  |
| <code>n_sep_by_space</code>  | <code>CHAR_MAX</code> | 如果 <code>currency_symbol</code> 是用空格来分隔负值，则为1；否则为0                                  |
| <code>p_sign_posn</code>     | <code>CHAR_MAX</code> | <code>positive_sign</code> 的位置表明非负值（见表25-3）                                         |
| <code>n_sign_posn</code>     | <code>CHAR_MAX</code> | <code>negative_sign</code> 的位置表明负值（见表25-3）                                          |

表25-3 p\_sign\_posn和n\_sign\_posn的值

| 值 | 含    义                      |
|---|-----------------------------|
| 0 | 围绕在数量和currency_symbol周围的圆括号 |
| 1 | 在数量和currency_symbol之前的符号    |
| 2 | 继数量和currency_symbol之后的符号    |
| 3 | 直接在currency_symbol之前的符号     |
| 4 | 直接继currency_symbol之后的符号     |

为了说明lconv结构的成员如何随着地区的不同而不同，下面来比较两个假想的示例。表25-4显示了用于美国和意大利两国的lconv结构成员的货币型常用值（稍后的示例是来自C标准自身的）。

表25-4 用于美国和意大利两国的lconv结构成员的货币型常用值

| 成    员            | 美    国 | 意  大  利 |
|-------------------|--------|---------|
| int_curr_symbol   | "USD " | "ITL."  |
| currency_symbol   | "\$"   | "L."    |
| mon_decimal_point | "."    | "."     |
| mon_thousands_sep | "\,"   | "\,"    |
| mon_grouping      | "\3"   | "\3"    |
| positive_sign     | "+"    | "+"     |
| negative_sign     | "_+"   | "_+"    |
| int_frac_digits   | 2      | 0       |
| frac_digits       | 2      | 0       |
| p_cs_precedes     | 1      | 1       |
| p_sep_by_space    | 0      | 0       |
| n_cs_precedes     | 1      | 1       |
| n_sep_by_space    | 0      | 0       |
| p_sign_posn       | 4      | 1       |
| n_sign_posn       | 4      | 1       |

555

下面是7593格式化成上述两个区域货币型值的情况：

| 美    国 | 意  大  利      |
|--------|--------------|
| 正数格式   | \$7,593.00   |
| 负数格式   | \$-7,593.00  |
| 国际化格式  | USD 7,593.00 |

请记住C语言的库函数不能自动格式化货币型值，直到每个程序都使用lconv结构中的信息才可以完成格式化。

## 25.2 多字节字符和宽字符

程序在适应不同地区的过程中最大的难题之一就是字符集的问题。在美国，主流计算机使用ASCII字符集，而其他的多数使用EBCDIC。在美国以外的地方，情况变得更加复杂。在一些国家，计算机采用类似于ASCII的字符集，但是缺少了某些字符。25.3节将会进一步讨论这个问题。在亚洲的其他国家则面临不同的问题：书写的语言要求巨大的字符集，通常是以千计的。

因为定义已经把char型值的大小限制为一个字节，所以通过改变char类型的含义来处理更

大的字符集显然是不可能的。取而代之的是，C语言允许编译器提供一种可扩展的字符集。这种字符集可以用于编写C程序（例如，在注释和字符串中），也可以用于程序运行的环境中，或者两种地方都有。**Q&A**C语言提供了两种用于可扩展字符集的编码：**多字节字符**（*multibyte character*）和**宽字符**（*wide character*）。C语言还提供了把一种编码转换成另外一种编码的函数。

### 25.2.1 多字节字符

在**多字节字符**编码中，一个或多个字节表示一个可扩展的字符。任何可扩展的字符集必须包含C语言要求的基本字符（即字母、数字、运算符、标点符号和空白字符）。这些字符都要求是单字节的。可以把其他字节解释为多字节字符的开始。

#### 日文字符集

日文采用不同的写入系统。最复杂的是日文中的汉字（kanji），它由上千个符号组成，因为符号实在是太多了，以至于不能用单个字节编码表示，（日文中的汉字符号实际上源自中国的汉字，汉字也有一个和大字符集类似的问题。）没有统一的方法对日文中的汉字编码，常用的编码包括JIS（日本工业标准）、Shift-JIS和EUC（可扩展的UNIX编码）。

556

一些多字节字符集依靠**依赖状态编码**（state-dependent encoding）。在这类编码中，每个多字节字符序列都以**初始移位状态**（initial shift state）开始。序列中稍后遇到的一些多字节字符会改变移位状态，并且会影响后续字节的含义。例如，日本的JIS编码把单字节码与双字节码进行混合，而嵌入在字符串中的“转义序列”则说明了单字节模式和双字节模式互相切换的时间。（反之，Shift-JIS编码不是依赖状态的。每个字符要求一个或者两个字节，但是双字节字符的第一个字节可以始终区别于单字节字符。）

在任何编码中，无论移位状态如何，C标准都要求零字节始终用来表示空字符。而且，零字节不能是多字节字符的第二个（或者后一个）字节。

C语言库提供了两种与多字节字符相关的宏MB\_LEN\_MAX和MB\_CUR\_MAX，这两种宏说明了多字节字符中字节的最大数量。宏MB\_LEN\_MAX（定义在<limits.h>中）给出了任意支持区域的最大值，而宏MB\_CUR\_MAX（定义在<stdlib.h>中）则给出了当前区域的最大值。（改变地区可能会影响多字节字符的解释。）显然，宏MB\_CUR\_MAX不可能大于宏MB\_LEN\_MAX。

### 25.2.2 宽字符

另外一种对可扩展字符集进行编码的方法是使用**宽字符**（*wide characters*）。宽字符是一种其值表示字符的整数。不同于长度不同的多字节字符，采用特殊实现支持的所有宽字符都要求相同的字节数。

宽字符具有wchar\_t类型（定义在<stddef.h>和<stdlib.h>中），且它必须是整数类型才可以表示任何支持地区的可扩展字符集。例如，如果两个字节足够表示任何可扩展字符集，那么将会把wchar\_t定义成unsigned short int类型。

使用宽字符的一个好处是C语言支持宽字符常量和宽字符串字面量。宽字符常量类似于普通的字符常量，只是前者需要有字母L作为前缀：

L'a'

而宽字符串字面量也需要用字母L作为前缀：

L"abc"



557 此字符串表示一个含有宽字符L'a'、L'b'和L'c'并且后边跟着代码为零的宽字符的数组。

### Unicode

宽字符非常适合于固定长度编码的字符集。一个重要的示例就是Unicode。作为一种通用字符集的尝试，Unicode是一种每个国家都可以采用的编码。Windows NT操作系统当前支持Unicode，而且它很可能会及时成为其他操作系统的特性。每个Unicode字符占用两个字节，所以Unicode可以表示65 536个字符，并且为所有现代语言以及一些古老语言（例如，梵语<sup>①</sup>）要求的字母表预留了足够的空间。Unicode还包含了一定数量的特殊符号，比如用于数学运算的符号等。

#### 25.2.3 多字节字符函数

```
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
```

mblen函数检测第一个参数是否指向形成有效多字节字符的字节序列。如果是，函数返回字符中的字节数。如果不是，函数则返回-1。作为一种特殊情况，如果函数的第一个参数指向空字符，则mblen函数返回0。函数的第二个参数限制了mblen函数将检测的字节的数量。通常情况下会传递MB\_CUR\_MAX。

下面的函数使用mblen函数来确定字符串是否由有效的多字节字符构成。（此示例和稍后本节中的wccheck示例都来自于P. J. Plauger写的*The Standard C Library*）如果s指向有效字符串，则函数返回零。

```
int mbcheck(const char *s)
{
 int n;

 for(mblen(NULL, 0) ; ; s += n)
 if ((n = mblen(s, MB_CUR_MAX)) <= 0)
 return n;
}
```

mbcheck函数的两个概念需要特别说明一下。首先是mblen(NULL, 0)的神秘调用。此调用使mblen函数跟踪静态变量中的移位状态。mblen(NULL, 0)的调用设置了此变量的初始状态，以便于可以正确解释字符串中稍后的字符。（把空指针传递给mbtowc函数或者wctomb函数具有相似的效果。顺便说一下，每个函数都有自己的移位状态。）mblen函数的调用可以改变移位状态。其次是有关终止的问题。请记住s指向的是以空字符结尾的普通字符串。当mblen函数遇到这个空字符时将返回零，这样的结果会导致mbcheck函数返回。

mbtowc函数把（由函数的第二个参数指向的）多字节字符转换为宽字符。第一个参数指向函数将存储结果的变量，第三个参数限制了mbtowc函数将检测的字节的数量。mbtowc函数返回和mblen函数一样的值：如果有效，则返回字符中字节的数量；如果无效，则返回-1；如果第二个参数指向空字符，则返回零。

wctomb函数把宽字符（第二个参数）转换为多字节字符，并且把多字节字符存储到第一个参数指向的数组中。wctomb函数可以存储和MB\_LEN\_MAX一样多的字符到数组中，但是不附加空字符。转换会考虑当前的移位状态，如果需要还会更新移位状态。如果有效，wctomb

<sup>①</sup> 一种古印度语，为印度及吠陀经所用文字，也是印度的古典文学语言。——编者注

函数会返回字符中字节的数量；如果无效，则返回-1。（注意，如果要求转换空的宽字符，则返回1。）

下面这个函数使用wctomb函数来确定是否可以把宽字符的字符串转换为有效的多字节字符：

```
int wccheck(wchar_t *wcs)
{
 char buf [MB_LEN_MAX];
 int n;

 for (wctomb (NULL, 0); ; ++wcs)
 if ((n = wctomb (buf, *wcs)) <= 0)
 return -1; /* invalid character */
 else if (buf [n-1] == '\0')
 return 0; /* all characters are valid */
}
```

顺便说一下，mblen函数、mbtowc函数和wctomb函数都可以用来监测多字节编码是否是依赖状态的。当传递空指针作为char\*类型的参数时，如果多字节字符是依赖状态的，那么上述每种函数都会返回非零值；否则返回零。

#### 25.2.4 多字节字符串函数



mbstowcs函数把多字节字符串转换为宽字符串序列。函数的第二个参数指向多字节字符串，而第一个参数则指向宽字符的数组，第三个参数限制了可以存储在数组中的宽字符数量。当达到上限或者遇到空字符（存储在宽字符数组中）时，mbstowcs函数就停止。函数会返回修改的数组元素的数量，但是无论如何不会包括用来终止的零代码。如果遇到无效的多字节字符，mbstowcs函数则返回-1。

wcstombs函数和mbstowcs函数正好相反：它把宽字符串转换为多字节字符。函数的第二个参数指向宽字符串，第一个参数指向存储多字节字符的数组，第三个参数限制了存储在数组中的字节的数量。当达到上限或者遇到空字符（函数存储的）时，wcstombs函数就停止。函数会返回存储的字节的数量，但是无论如何不会包含用来终止的空字符。如果遇到一个宽字符无法对应任何多字节字符，则wcstombs函数返回-1。

mbstowcs函数假设要转换的字符串以初始移位状态开始，而由wcstombs函数产生的字符串则始终是以初始移位状态开始。

### 25.3 三字符序列

三字符序列（trigraph sequence）（或者简称为“三字符”）是一种三个字符码，它可以用作ASCII字符的替代品。三字符寻址的问题很简单：C程序需要字符#、[、\、]、^、{、|、}和~。许多欧洲国家使用缺少这样一些字符的ASCII的替换形式。例如，在德国，把#、[、\、]、^、{、|、}和~分别替換成了Ä、Ö、Ü、ä、ö、ü和ß。三字符提供了一种编写有效C程序而不使用任何缺少字符的方法。

表25-5给出了三字符序列的完整列表。所有三字符都以??开始，这样做虽然不能十分吸引人，但至少可以便于发现三字符。

表25-5 三字符序列

| 三字符序列 | 等价的ASCII码 |
|-------|-----------|
| ??=   | #         |
| ??(   | [         |
| ??/   | \         |
| ??)   | ]         |
| ??'   | ^         |
| ??<   | {         |
| ??!   |           |
| ??>   | }         |
| ??-   | ~         |

三字符可以自由地替换成等价的ASCII码。例如，程序

```
#include <stdio.h>

main()
{
 printf("hello, world\n");
 return 0;
}
```

560

可以写成

```
??=include <stdio.h>

main()
??<
 printf("hello, word??/n")
 return 0;
??>
```

尽管不是一直需要，但是要求所有的标准C编译器都接受三字符序列。偶尔，这个特性可能会导致问题。



在字符串中请小心放置??，因为编译器可能会把它作为三字符序列的开始标志。

如果发生这种情况，那么通过在第二个?字符的前面放置字符\来把第二个字符?变成转义序列。?\?这样组合的结果就不会被看作是三字符的开始了。

## 问与答

问：**setlocale**函数可以返回多长的地区信息字符串？(p.384)

答：不存在最大长度。这就引发了一个问题：如果不知道字符串的长度，如何为字符串设置空间呢？当然，答案就是动态存储分配。下面这个例子（基于Harbison和Steele写的*C: A Reference Manual*一书的类似示例）说明了如何确定需要的空间数量，然后再把地区信息复制到此空间中：

```
char *temp,*old_locale;

temp = setlocale(LC_ALL, NULL);
if (temp == NULL) {
 /* locale information not available */
}
old_locale = malloc (strlen (temp)+1);
if (old_locale == NULL) {
 /* memory allocation failed */
}
strcpy (old_locale, temp);
```

为了恢复旧的地区信息，最好首先切换成本地模式，然后再恢复旧的地区：

```
setlocale(LC_ALL, "") /* switch to native locale */
setlocale(LU_ALL, old_locale); /* restore old locale */
```

561

问：为什么C语言提供多字节字符和宽字符呢？两者选其一难道不够吗？(p.387)

答：两种编码用于不同的目的。多字节字符用于输入/输出目的很方便，因为输入/输出设备经常是面向字节的。但是宽字符更适用于程序内部，因为每个宽字符占有相同的空间。因此，程序可以读入多字节字符输入，把它转换为便于程序内部操作的宽字符格式，然后再把宽字符转换回用于输出的多字节格式。

## 练习

### 25.1节

1. 请确定你用的编译器支持哪种地区。
2. 编写一个程序，用来测试你用的编译器的""（本地）地区是否和"C"地区一样。

### 25.2节

3. 用于kanji（日文中的汉字）的Shift-JIS编码要求每个字符是单字节或者是双字节的。如果字符的第一个字节位于0x81和0x9f之间，或者位于0xe0和0xef之间，那么就需要第二个字节。（把任何其他字节看成是整个字符。）第二个字节必须在0x40和0x7e之间，或者在0x80和0xfc之间。（所有的范围都包含边界值。）对于下面的每个字符串，当传递其作为参数时，请指出25.2节的mbcheck函数将会返回的值。

- (a) "\x05\x87\x80\x36\xed\xaa"
- (b) "\x20\xe4\x50\x88\x3f"
- (c) "\xde\xad\xbe\xef"
- (d) "\x8a\x60\x92\x74\x41"

### 25.3节

4. 请通过尽可能多地用三字符替换字符的方法来修改下面的程序段。

```
While ((orig_char = getchar()) != EOF) {
 new_char = orig_char ^ KEY;
 if (iscntrl(orig_char) || iscntrl(new_char))
 putchar(orig_char);
 else
 putchar(new_char);
}
```

562

确定程序参数的应该是用户，而不应该是它们的创造者。

本章讨论标准库中剩下的三个头：`<stdarg.h>`、`<stdlib.h>`和`<time.h>`。这三个头不同于库函数中的其他头文件，所以把它们留到最后来介绍。`<stdarg.h>`（26.1节）可使编写的函数带有可变数量的实参；`<stdlib.h>`（26.2节）是函数的分类，但是此分类不适合其他库函数头的某一种。`<time.h>`允许程序处理日期和时间。

### 26.1 `<stdarg.h>`：可变长度实参

我们已经见过类似`printf`函数和`scanf`函数这样的函数，它们在接收的参数数量上没有固定的限制。然而，这种能力是对库函数而言不限制处理可变数量的参数。现在`<stdarg.h>`将提供一种工具使我们自行编写的函数也具有可变长度的参数列表。`<stdarg.h>`定义了一种`va_list`类型和三种宏，这三种宏名为`va_start`、`va_arg`和`va_end`。可以把这些宏看成是带有上述原型的函数。

563

为了了解这些宏的工作过程，这里将用它们来编写一个名为`max_int`的函数。此函数用来在任意数量的整型参数中找出最大数。下面是此函数的调用过程：

```
max_int(3, 10, 30, 20)
```

函数的第一个实参说明了跟随其后的其他参数的数量。这里的`max_int`函数调用将会返回30（即10、30和20中的最大数）。

下面是`max_int`函数的定义：

```
int max_int(int n, ...) /* n must be at least 1 */
{
 va_list ap;
 int i, current, largest;

 va_start (ap, n) ;
 largest = va_arg(ap, int) ;

 for (i : i; i < n; i++) {
 current = va_arg (ap, int) ;
 if (current > largest)
 largest = current;
 }

 va_end (ap) ;
 return largest;
}
```

在形式参数列表中的...符号（省略号）表示参数n后边跟随着其他可变数量的参数。

max\_int函数体从声明va\_list类型的变量开始：

```
va_list ap;
```

声明这样的变量是为了强制max\_int函数可以访问到跟在n后边的实参。

语句va\_start(ap, n);指出了实参列表中可变长度部分开始的位置（这里的情况是从n后边开始）。带有可变数量参数的函数必须至少有一个“正常的”形式参数，在最后一个正常参数的后边始终会有省略号出现在参数列表的末尾。564

语句largest = va\_arg(ap, int);把获取的max\_int函数第2个参数（n后面的一个）赋值给变量largest，并且自动前进到下一个参数处。语句中的单词int指明希望的max\_int函数的第2个实参是int类型的。当程序执行内部循环时，语句current = va\_arg(ap, int);会逐个获取max\_int函数余下的参数。



不要忘记在获取当前参数后，宏va\_arg始终会前进到下一个参数的位置上。正是由于这个特点，所以这里不能用如下方式编写max\_int函数的循环：

```
for (I = 1; I < n; I++)
 if (va_arg (ap, int) > largest) /*** WRONG ***
 largest = va_arg(ap, int);
```

在函数返回之前，要求用语句va\_end(ap);进行“清扫”。（如果不返回，函数可能会再次调用va\_start并且遍历参数列表。）

当调用带有可变实参列表的函数时，编译器会在匹配省略号的全部参数上执行默认的实参提升（>9.3.1节）：把字符型值提升为整数，并且把float型值提升为double型值。这样的结果是va\_arg不会感觉到传递过来的是字符类型或float类型，因为提升后的参数将永远不会具有这些类型。

### 26.1.1 调用带有可变实参列表的函数

调用带有可变实参列表的函数是一个固有的风险提议。回溯到第3章就会发现给printf函数和scanf函数传递错误参数是多么危险。其他带有可变实参列表的函数也同样容易出问题。主要的难点就是带有可变实参列表的函数很难确定传递过来的参数的数量或类型。所以必须要把这个信息传递给函数，并且/或者函数假设知道了这个信息。示例中的max\_int函数依靠第一个实参来指明跟随其后的其他参数的数量，并且它还假定参数是int类型的。而像printf函数和scanf函数这样的函数则是依靠格式化字符串来描述其他的参数的数量和每种参数的类型。

另外一个问题是不得不处理NULL作为参数传递的情况。通常都把NULL定义成表示0。但是，当把0传递给带有可变实参列表的函数时，编译器会假定它表示的是一个整数，因为这里没有办法可以说明希望它表示的是一个空指针。解决这种问题的方法就是添加一个强制类型转换，用(void\*) NULL来代替NULL，这样就可以表明传递的是一个空指针了。（请见第17章的“问与答”小节关于此点的更多详细讨论。）565

### 26.1.2 v...printf类函数



vfprintf函数、vprintf函数和vsprintf函数（即v...printf类函数）都属于<stdio.h>。本节讨论这些函数是因为它们总是和宏联合用于<stdarg.h>中。

v...printf类函数和fprintf函数、printf函数以及sprintf函数有着紧密的联系。但是，不同于这些函数的是v...printf类函数具有固定数量的实参。每个v...printf类函数的最后一个



实参都是一个va\_list型值。这个类型的值意味着此函数将可以由带有可变实参列表的函数调用。实际上，v...printf类函数主要用于编写“包装”函数。包装函数接收可变数量的实参，并且稍后把这些参数传递给v...printf类函数。

举一个例子，假设正在工作的程序需要不时地显示出错消息，而且我们希望每条消息都以下列格式的前缀开始：

```
** Error n:
```

这里的n在显示第一条出错消息时是1，并且n会随着下一条错误信息而逐次加一。为了使产生出错信息更加容易，我们将编写一个名为errorf的函数，此函数类似于printf函数，但是它在输出的开始处添加了\*\* Error n:，而且此函数不向stdout输出而是向stderr写输出。errorf函数将调用vfprintf函数来完成实际输出的大部分内容。下面是errorf函数可能的写法：

```
int errorf(const char *format, ...)
{
 static int num_errors = 0;
 int n;
 va_list ap;

 num_errors++;
 fprintf(stderr, "** Error %d: ", num_errors);
 va_start(ap, format);
 n = vfprintf(stderr, format, ap);
 va_end(ap);
 fprintf(stderr, "\n");
 return n;
}
```

566

## 26.2 <stdlib.h>: 通用的实用工具

<stdlib.h>涵盖了全部不适合于任何其他头的函数。<stdlib.h>中的函数可以大致分为7种各不相关的组：

- 字符串转换函数。
- 伪随机序列生成函数。
- 内存管理函数。
- 与外部环境的通信。
- 搜索和排序实用工具。
- 整数算术运算函数。
- 多字节字符和字符串函数。

这里将逐个介绍每组函数，但是有两组例外：内存管理函数以及多字节字符和字符串函数。

内存管理函数（即malloc函数、calloc函数、realloc函数和free函数）允许程序分配内存块，而且稍后它们还允许程序释放或者改变内存块的大小。第17章已经详细描述了这4种内存管理函数。

多字节字符和字符串函数允许程序对多于一个字节长度的字符进行操作。25.2节已经介绍了多字节字符，并且解释说明了多字节函数的工作原理。

### 26.2.1 字符串转换函数

```
double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
double strtod(const char *nptr, char **endptr);
```

```
long lval;
long lmax;
long lmin;
long lneg;
long lzero;
```

这一组的函数会把含有字符格式的数字符串转换成它的等价数值。这些函数中有3个函数是非常旧的，而其他3个函数则是在C语言标准化过程中添加进来的。

旧函数（`atof`函数、`atoi`函数和`atol`函数）把字符串分别转换成`double`、`int`或者`long int`型值。每个函数都会在字符串的开始处跳过空白字符，并且把后续字符作为数的一部分，同时每个函数还会在第一个不是数的部分的字符处停止。

新函数（`strtod`函数、`strtol`函数和`strtoul`函数）更加复杂精妙。举个例子来说，这三个函数会通过修改`endptr`指向的变量来指出转换停止的位置。（如果不在乎转换结束的位置，那么函数的第二个参数可以为空指针。）为了检测函数是否可以使用到整个字符串，只需检测此变量是否可以指向空字符。更厉害的是`strtol`函数和`strtoul`函数还有一个`base`参数用来说明要转换数的基数。函数支持的基数是2到36之间的所有数（包括2和36）。

除了比原来的旧函数更通用以外，新函数还更善于处理错误。旧函数没有办法可以指明转换期间用到的字符串的数量。此外，如果旧函数无法定位要转换的数，它会返回零。如果数过大，那么旧函数则返回未定义的值。这样的结果是，上述这些都无法通过检测函数的返回值来发现任何问题。而如果转换产生的用来表示的值过大（或者过小），那么新函数就会把`ERANGE`存储在`errno`中（`<errno.h>`▶24.2节）。

由于`strtod`函数、`strtol`函数和`strtoul`函数的添加，`atof`函数、`atoi`函数和`atol`函数就显得多余了。为了早期的C程序方便使用，这些旧函数仍保留在函数库中，但是这里推荐新程序使用`strtod`函数、`strtol`函数和`strtoul`函数。

## 26.2.2 程序：测试字符串转换函数

下面这个程序通过应用6种字符串转换函数中的每一种来把字符串转换为数值格式。在调用了`strtod`函数、`strtol`函数和`strtoul`函数之后，程序还会显示出是否每种版本都产生了有效的结果，以及是否每种版本可以用到整个字符串。程序将从命令行中获得输入字符串。

```
tstrconv.c
/* Tests string conversion functions */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

#define CHK_VALID printf(" %s %s\n",
 errno != ERANGE ? "Yes" : "No ", \
 *ptr == '\0' ? "Yes" : "No");

main(int argc, char *argv[])
{
 char *ptr;

 if (argc != 2) {
 printf("usage: tstrconv string\n");
 exit(EXIT_FAILURE);
 }

 printf("Function Return Value\n");
 printf("-----\n");
 printf("atof %g\n", atof(argv[1]));
 printf("atoi %d\n", atoi(argv[1]));
 printf("atol %ld\n\n", atol(argv[1]));
```

567

568

```

printf("Function Return Value Valid? "
 "String Consumed?\n"
 "----- ----- ----- "
 "\n");

errno = 0;
printf("strtod %-12g", strtod(argv[1], &ptr));
CHK_VALID;

errno = 0;
printf("strtol %-12ld", strtol(argv[1], &ptr, 10));
CHK_VALID;

errno = 0;
printf("strtoul %-12lu", strtoul(argv[1], &ptr, 10));
CHK_VALID;

return 0;
}

```

如果3000000000是命令行参数，那么tstrconv的输出可能不会有如下显示：

| Function | Return Value | Valid? | String Consumed? |
|----------|--------------|--------|------------------|
| atof     | 3e+09        | Yes    | Yes              |
| atoi     | 24064        | No     | Yes              |
| atol     | -1294967296  | Yes    | Yes              |

| Function | Return Value | Valid? | String Consumed? |
|----------|--------------|--------|------------------|
| strtod   | 3e+09        | Yes    | Yes              |
| strtol   | 2147483647   | No     | Yes              |
| strtoul  | 30000000000  | Yes    | Yes              |

虽然3000000000这个数作为无符号长整数是有效的，但是因为它太长了对许多机器而言都很难表示为长整数。atoi函数和atol函数就无法发现这个问题，而且它们会以返回奇怪的值结束。strtol函数会执行正确地转换，而strtoul函数则会返回2147483647（最大的长整数）并且把ERANGE存储到errno中。

如果123.456是命令行参数，那么输出将是：

| Function | Return Value | Valid? | String Consumed? |
|----------|--------------|--------|------------------|
| atof     | 123.456      | Yes    | Yes              |
| atoi     | 123          | Yes    | No               |
| atol     | 123          | Yes    | No               |

| Function | Return Value | Valid? | String Consumed? |
|----------|--------------|--------|------------------|
| strtod   | 123.456      | Yes    | Yes              |
| strtol   | 123          | Yes    | No               |
| strtoul  | 123          | Yes    | No               |

569

所有的函数都会把这个输入作为有效的数来处理，但是整型函数会在小数点处停止。strtol函数和strtoul函数无法完全用到整个输入，就是因为这个问题。

如果foo是命令行参数，那么输出将是：

| Function | Return Value | Valid? | String Consumed? |
|----------|--------------|--------|------------------|
| atof     | 0            | Yes    | Yes              |
| atoi     | 0            | Yes    | No               |
| atol     | 0            | Yes    | No               |

| Function | Return Value | Valid? | String Consumed? |
|----------|--------------|--------|------------------|
| strtod   | 0            | Yes    | No               |

|         |   |     |    |
|---------|---|-----|----|
| strtol  | 0 | Yes | No |
| strtoul | 0 | Yes | No |

全部函数看到字母f立刻返回零。str...类函数不会改变errno，但是我们会明白出现这问题实际是函数没有使用到字符串。

### 26.2.3 伪随机序列生成函数

```
int rand(void);
void srand(unsigned int seed);
```

rand函数和srand函数都可以用来生成伪随机数。这两个函数用于模拟程序和玩游戏程序（例如，在纸牌游戏中用来模拟骰子滚动或者发牌。）。

每次调用rand函数时，它都会返回一个0~RAND\_MAX（定义在<stdlib.h>中的宏）的数。rand函数返回的数事实上不是随机的。这些数是由“种子”值产生的。但是，对于偶然的观察者而言，rand函数表现出来的是产生了不相关的数序列。

调用srand函数可以提供用于rand函数的种子值。如果在srand函数之前调用rand函数，那么会把种子值设定为1。每个种子值确定了一个特殊的“随机”数序列。srand函数允许用户选择他们自己想要的序列。

始终使用同一个种子值的程序将总会从rand函数得到相同的数序列。这个特点有时是非常有用的：程序在每次运行时按照相同的方式运行，这样会使得测试更加容易。但是，用户通常是希望每次程序运行时rand函数能产生不同的序列的。（玩纸牌的程序如果总是发同样的牌是不会受欢迎的。）使种子值“随机化”的最简单方法就是调用time函数（time函数>26.3.1节），它会返回一个对当前日期和时间进行编码的数。把time函数的返回值传递给srand函数，这样可以使rand函数在每次运行时的行为都不相同。这种方法可以见10.2节中的示例guess.c程序和guess2.c程序。

570

### 26.2.4 程序：测试伪随机序列生成函数

下面这个程序首先显示由rand函数返回的前10个值，然后允许用户选择新的种子值。此过程会反复执行直到用户输入零作为种子值为止。

```
trand.c
/* Tests the pseudo-random sequence generation functions */

#include <stdio.h>
#include <stdlib.h>

main()
{
 int i, seed;

 printf("This program displays the first ten values of "
 "rand.\n");

 for (;;) {
 for (i = 0; i < 10; i++)
 printf("%d ", rand());
 printf("\n\n");
 printf("Enter new seed value (0 to terminate): ");
 scanf("%d", &seed);
 if (seed == 0)
 break;
 srand(seed);
 }

 return 0;
}
```

下面是程序运行时可能的交互情况：

```
This program displays the first ten values of rand.
346 130 10982 1090 11656 7117 17595 6415 22948 31126

Enter new seed value (0 to terminate): 100
1862 11548 3973 4846 9095 16503 6335 13684 21357 21505

Enter new seed value (0 to terminate): 1
346 130 10982 1090 11656 7117 17595 6415 22948 31126

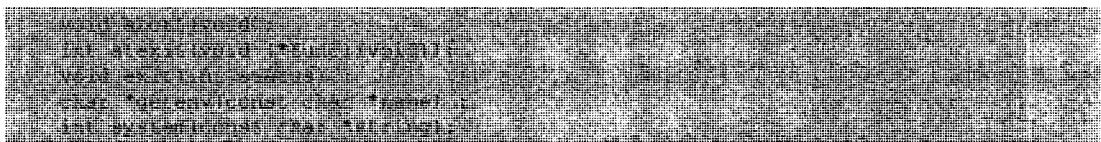
Enter new seed value (0 to terminate): 0
```

编写rand函数的方法有很多，所以这里不保证每种rand函数的版本都可以生成上述这样的数。

571

请注意，选择1作为种子值与根本没有指明种子值会得到相同的数列。

## 26.2.5 与环境的通信



这一组函数为操作系统提供了简单的界面。它们允许程序：正常的或不正常的终止，并且为操作系统返回一个状态码；从用户的外部环境获取信息；执行操作系统的命令。

在程序中的任何位置执行exit(*n*)调用等价于在main函数中执行return *n*；语句：即程序终止，并且把*n*作为状态码返回给操作系统。`<stdlib.h>`定义了宏EXIT\_FAILURE和宏EXIT\_SUCCESS，这些宏可以用作exit函数的参数。exit函数另一个唯一的可移植参数就是0，它和宏EXIT\_SUCCESS意义相同。返回其他不是上述这些的状态码也是合法的，但是它们对所有操作系统而言都是不可移植的。

当终止程序时，(atexit函数)通常还会在屏幕后台执行一些最后的动作，包括清洗输出缓冲区，关闭打开的流，以及删除临时文件。当然可能还有其他希望程序终止时执行的“清扫”操作。atexit函数允许用户“注册”一个临近程序终止时要调用的函数。例如，为了注册名为cleanup的函数，可以用如下方式调用atexit函数：

```
atexit(cleanup);
```

当把函数指针传递给atexit函数时，它会为将来的引用而把指针保存起来。稍后，程序终止时都将自动调用任何由atexit函数注册的函数。(如果注册了几个函数，那么将首先调用最新注册的函数)。

abort函数类似于exit函数，但是前者会导致异常的程序终止。还不能调用由atexit函数注册的函数。依靠实现，abort函数可以是这样一种情况：不能清洗文件缓冲区、不能关闭流和不能删除临时文件。**Q&A** abort函数返回一个由实现定义的状态码来说明“不成功的终止”。

许多操作系统都会提供“外部环境”：即一套描述用户特征的字符串。这些字符串通常包含用户运行程序时要搜索的路径、用户终端的类型（比如多用户系统的情况）等等。例如，一条UNIX系统的搜索路径可能如下所示：

```
PATH=~/bin:/bin:/usr/bin:.
```

572

而DOS系统的路径可能具有类似表示：

```
PATH=C:\;C:DOS;C:\WINDOWS
```

getenv函数提供了在用户的外部环境中访问任意字符串的功能。例如，为了找到PATH字符串的当前值，可以写成

```
p = getenv("PATH");
```

在执行了此条语句之后, p会指向诸如“`~/bin:/bin:/usr/bin:.`”或者“`C:\;C:DOS;C:\WINDOWS`”这样的字符串。由getenv函数返回的字符串会被静态分配, 并且由稍后的函数调用进行改变。

system函数允许C程序运行另一个程序(可能是一个操作系统命令)。system函数的参数是命令行, 这类似于一个在操作系统提示下的录入内容。例如, 假设正在编写的程序需要当前目录中的文件列表。UNIX程序将按照下列方式调用system函数:

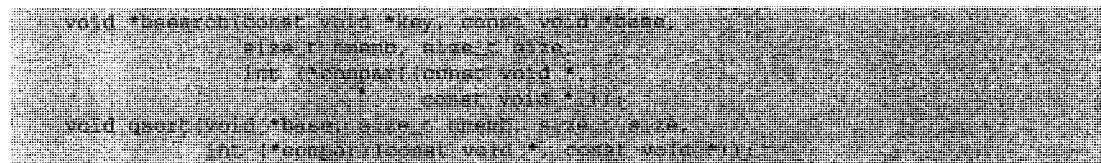
```
system("ls >myfiles");
```

而DOS程序会使用略有不同的调用方式:

```
system("dir >myfiles");
```

无论哪种调用之后, myfiles都将包含目录列表。system函数返回的值是由实现定义的。通常情况下, system函数会返回来自程序的终止状态码, 此程序是system函数要求运行的。测试这个返回值可以检测程序是否正常工作。调用带有空指针的system函数有特别的含义: 如果命令处理程序是有效的, 那么函数会返回非零值。

## 26.2.6 搜索和排序实用工具



A screenshot of a terminal window showing the output of the 'ls' command. The output lists several files and directories, including 'a.out', 'answ', 'answ.c', 'answ.h', 'answ.o', 'answ1', 'answ1.c', 'answ1.h', 'answ1.o', 'answ2', 'answ2.c', 'answ2.h', 'answ2.o', 'answ3', 'answ3.c', 'answ3.h', 'answ3.o', 'answ4', 'answ4.c', 'answ4.h', 'answ4.o', 'answ5', 'answ5.c', 'answ5.h', 'answ5.o', 'answ6', 'answ6.c', 'answ6.h', 'answ6.o', 'answ7', 'answ7.c', 'answ7.h', 'answ7.o', 'answ8', 'answ8.c', 'answ8.h', 'answ8.o', 'answ9', 'answ9.c', 'answ9.h', 'answ9.o', 'answ10', 'answ10.c', 'answ10.h', 'answ10.o', 'answ11', 'answ11.c', 'answ11.h', 'answ11.o', 'answ12', 'answ12.c', 'answ12.h', 'answ12.o', 'answ13', 'answ13.c', 'answ13.h', 'answ13.o', 'answ14', 'answ14.c', 'answ14.h', 'answ14.o', 'answ15', 'answ15.c', 'answ15.h', 'answ15.o', 'answ16', 'answ16.c', 'answ16.h', 'answ16.o', 'answ17', 'answ17.c', 'answ17.h', 'answ17.o', 'answ18', 'answ18.c', 'answ18.h', 'answ18.o', 'answ19', 'answ19.c', 'answ19.h', 'answ19.o', 'answ20', 'answ20.c', 'answ20.h', 'answ20.o', 'answ21', 'answ21.c', 'answ21.h', 'answ21.o', 'answ22', 'answ22.c', 'answ22.h', 'answ22.o', 'answ23', 'answ23.c', 'answ23.h', 'answ23.o', 'answ24', 'answ24.c', 'answ24.h', 'answ24.o', 'answ25', 'answ25.c', 'answ25.h', 'answ25.o', 'answ26', 'answ26.c', 'answ26.h', 'answ26.o', 'answ27', 'answ27.c', 'answ27.h', 'answ27.o', 'answ28', 'answ28.c', 'answ28.h', 'answ28.o', 'answ29', 'answ29.c', 'answ29.h', 'answ29.o', 'answ30', 'answ30.c', 'answ30.h', 'answ30.o', 'answ31', 'answ31.c', 'answ31.h', 'answ31.o', 'answ32', 'answ32.c', 'answ32.h', 'answ32.o', 'answ33', 'answ33.c', 'answ33.h', 'answ33.o', 'answ34', 'answ34.c', 'answ34.h', 'answ34.o', 'answ35', 'answ35.c', 'answ35.h', 'answ35.o', 'answ36', 'answ36.c', 'answ36.h', 'answ36.o', 'answ37', 'answ37.c', 'answ37.h', 'answ37.o', 'answ38', 'answ38.c', 'answ38.h', 'answ38.o', 'answ39', 'answ39.c', 'answ39.h', 'answ39.o', 'answ40', 'answ40.c', 'answ40.h', 'answ40.o', 'answ41', 'answ41.c', 'answ41.h', 'answ41.o', 'answ42', 'answ42.c', 'answ42.h', 'answ42.o', 'answ43', 'answ43.c', 'answ43.h', 'answ43.o', 'answ44', 'answ44.c', 'answ44.h', 'answ44.o', 'answ45', 'answ45.c', 'answ45.h', 'answ45.o', 'answ46', 'answ46.c', 'answ46.h', 'answ46.o', 'answ47', 'answ47.c', 'answ47.h', 'answ47.o', 'answ48', 'answ48.c', 'answ48.h', 'answ48.o', 'answ49', 'answ49.c', 'answ49.h', 'answ49.o', 'answ50', 'answ50.c', 'answ50.h', 'answ50.o', 'answ51', 'answ51.c', 'answ51.h', 'answ51.o', 'answ52', 'answ52.c', 'answ52.h', 'answ52.o', 'answ53', 'answ53.c', 'answ53.h', 'answ53.o', 'answ54', 'answ54.c', 'answ54.h', 'answ54.o', 'answ55', 'answ55.c', 'answ55.h', 'answ55.o', 'answ56', 'answ56.c', 'answ56.h', 'answ56.o', 'answ57', 'answ57.c', 'answ57.h', 'answ57.o', 'answ58', 'answ58.c', 'answ58.h', 'answ58.o', 'answ59', 'answ59.c', 'answ59.h', 'answ59.o', 'answ60', 'answ60.c', 'answ60.h', 'answ60.o', 'answ61', 'answ61.c', 'answ61.h', 'answ61.o', 'answ62', 'answ62.c', 'answ62.h', 'answ62.o', 'answ63', 'answ63.c', 'answ63.h', 'answ63.o', 'answ64', 'answ64.c', 'answ64.h', 'answ64.o', 'answ65', 'answ65.c', 'answ65.h', 'answ65.o', 'answ66', 'answ66.c', 'answ66.h', 'answ66.o', 'answ67', 'answ67.c', 'answ67.h', 'answ67.o', 'answ68', 'answ68.c', 'answ68.h', 'answ68.o', 'answ69', 'answ69.c', 'answ69.h', 'answ69.o', 'answ70', 'answ70.c', 'answ70.h', 'answ70.o', 'answ71', 'answ71.c', 'answ71.h', 'answ71.o', 'answ72', 'answ72.c', 'answ72.h', 'answ72.o', 'answ73', 'answ73.c', 'answ73.h', 'answ73.o', 'answ74', 'answ74.c', 'answ74.h', 'answ74.o', 'answ75', 'answ75.c', 'answ75.h', 'answ75.o', 'answ76', 'answ76.c', 'answ76.h', 'answ76.o', 'answ77', 'answ77.c', 'answ77.h', 'answ77.o', 'answ78', 'answ78.c', 'answ78.h', 'answ78.o', 'answ79', 'answ79.c', 'answ79.h', 'answ79.o', 'answ80', 'answ80.c', 'answ80.h', 'answ80.o', 'answ81', 'answ81.c', 'answ81.h', 'answ81.o', 'answ82', 'answ82.c', 'answ82.h', 'answ82.o', 'answ83', 'answ83.c', 'answ83.h', 'answ83.o', 'answ84', 'answ84.c', 'answ84.h', 'answ84.o', 'answ85', 'answ85.c', 'answ85.h', 'answ85.o', 'answ86', 'answ86.c', 'answ86.h', 'answ86.o', 'answ87', 'answ87.c', 'answ87.h', 'answ87.o', 'answ88', 'answ88.c', 'answ88.h', 'answ88.o', 'answ89', 'answ89.c', 'answ89.h', 'answ89.o', 'answ90', 'answ90.c', 'answ90.h', 'answ90.o', 'answ91', 'answ91.c', 'answ91.h', 'answ91.o', 'answ92', 'answ92.c', 'answ92.h', 'answ92.o', 'answ93', 'answ93.c', 'answ93.h', 'answ93.o', 'answ94', 'answ94.c', 'answ94.h', 'answ94.o', 'answ95', 'answ95.c', 'answ95.h', 'answ95.o', 'answ96', 'answ96.c', 'answ96.h', 'answ96.o', 'answ97', 'answ97.c', 'answ97.h', 'answ97.o', 'answ98', 'answ98.c', 'answ98.h', 'answ98.o', 'answ99', 'answ99.c', 'answ99.h', 'answ99.o', 'answ100', 'answ100.c', 'answ100.h', 'answ100.o', 'answ101', 'answ101.c', 'answ101.h', 'answ101.o', 'answ102', 'answ102.c', 'answ102.h', 'answ102.o', 'answ103', 'answ103.c', 'answ103.h', 'answ103.o', 'answ104', 'answ104.c', 'answ104.h', 'answ104.o', 'answ105', 'answ105.c', 'answ105.h', 'answ105.o', 'answ106', 'answ106.c', 'answ106.h', 'answ106.o', 'answ107', 'answ107.c', 'answ107.h', 'answ107.o', 'answ108', 'answ108.c', 'answ108.h', 'answ108.o', 'answ109', 'answ109.c', 'answ109.h', 'answ109.o', 'answ110', 'answ110.c', 'answ110.h', 'answ110.o', 'answ111', 'answ111.c', 'answ111.h', 'answ111.o', 'answ112', 'answ112.c', 'answ112.h', 'answ112.o', 'answ113', 'answ113.c', 'answ113.h', 'answ113.o', 'answ114', 'answ114.c', 'answ114.h', 'answ114.o', 'answ115', 'answ115.c', 'answ115.h', 'answ115.o', 'answ116', 'answ116.c', 'answ116.h', 'answ116.o', 'answ117', 'answ117.c', 'answ117.h', 'answ117.o', 'answ118', 'answ118.c', 'answ118.h', 'answ118.o', 'answ119', 'answ119.c', 'answ119.h', 'answ119.o', 'answ120', 'answ120.c', 'answ120.h', 'answ120.o', 'answ121', 'answ121.c', 'answ121.h', 'answ121.o', 'answ122', 'answ122.c', 'answ122.h', 'answ122.o', 'answ123', 'answ123.c', 'answ123.h', 'answ123.o', 'answ124', 'answ124.c', 'answ124.h', 'answ124.o', 'answ125', 'answ125.c', 'answ125.h', 'answ125.o', 'answ126', 'answ126.c', 'answ126.h', 'answ126.o', 'answ127', 'answ127.c', 'answ127.h', 'answ127.o', 'answ128', 'answ128.c', 'answ128.h', 'answ128.o', 'answ129', 'answ129.c', 'answ129.h', 'answ129.o', 'answ130', 'answ130.c', 'answ130.h', 'answ130.o', 'answ131', 'answ131.c', 'answ131.h', 'answ131.o', 'answ132', 'answ132.c', 'answ132.h', 'answ132.o', 'answ133', 'answ133.c', 'answ133.h', 'answ133.o', 'answ134', 'answ134.c', 'answ134.h', 'answ134.o', 'answ135', 'answ135.c', 'answ135.h', 'answ135.o', 'answ136', 'answ136.c', 'answ136.h', 'answ136.o', 'answ137', 'answ137.c', 'answ137.h', 'answ137.o', 'answ138', 'answ138.c', 'answ138.h', 'answ138.o', 'answ139', 'answ139.c', 'answ139.h', 'answ139.o', 'answ140', 'answ140.c', 'answ140.h', 'answ140.o', 'answ141', 'answ141.c', 'answ141.h', 'answ141.o', 'answ142', 'answ142.c', 'answ142.h', 'answ142.o', 'answ143', 'answ143.c', 'answ143.h', 'answ143.o', 'answ144', 'answ144.c', 'answ144.h', 'answ144.o', 'answ145', 'answ145.c', 'answ145.h', 'answ145.o', 'answ146', 'answ146.c', 'answ146.h', 'answ146.o', 'answ147', 'answ147.c', 'answ147.h', 'answ147.o', 'answ148', 'answ148.c', 'answ148.h', 'answ148.o', 'answ149', 'answ149.c', 'answ149.h', 'answ149.o', 'answ150', 'answ150.c', 'answ150.h', 'answ150.o', 'answ151', 'answ151.c', 'answ151.h', 'answ151.o', 'answ152', 'answ152.c', 'answ152.h', 'answ152.o', 'answ153', 'answ153.c', 'answ153.h', 'answ153.o', 'answ154', 'answ154.c', 'answ154.h', 'answ154.o', 'answ155', 'answ155.c', 'answ155.h', 'answ155.o', 'answ156', 'answ156.c', 'answ156.h', 'answ156.o', 'answ157', 'answ157.c', 'answ157.h', 'answ157.o', 'answ158', 'answ158.c', 'answ158.h', 'answ158.o', 'answ159', 'answ159.c', 'answ159.h', 'answ159.o', 'answ160', 'answ160.c', 'answ160.h', 'answ160.o', 'answ161', 'answ161.c', 'answ161.h', 'answ161.o', 'answ162', 'answ162.c', 'answ162.h', 'answ162.o', 'answ163', 'answ163.c', 'answ163.h', 'answ163.o', 'answ164', 'answ164.c', 'answ164.h', 'answ164.o', 'answ165', 'answ165.c', 'answ165.h', 'answ165.o', 'answ166', 'answ166.c', 'answ166.h', 'answ166.o', 'answ167', 'answ167.c', 'answ167.h', 'answ167.o', 'answ168', 'answ168.c', 'answ168.h', 'answ168.o', 'answ169', 'answ169.c', 'answ169.h', 'answ169.o', 'answ170', 'answ170.c', 'answ170.h', 'answ170.o', 'answ171', 'answ171.c', 'answ171.h', 'answ171.o', 'answ172', 'answ172.c', 'answ172.h', 'answ172.o', 'answ173', 'answ173.c', 'answ173.h', 'answ173.o', 'answ174', 'answ174.c', 'answ174.h', 'answ174.o', 'answ175', 'answ175.c', 'answ175.h', 'answ175.o', 'answ176', 'answ176.c', 'answ176.h', 'answ176.o', 'answ177', 'answ177.c', 'answ177.h', 'answ177.o', 'answ178', 'answ178.c', 'answ178.h', 'answ178.o', 'answ179', 'answ179.c', 'answ179.h', 'answ179.o', 'answ180', 'answ180.c', 'answ180.h', 'answ180.o', 'answ181', 'answ181.c', 'answ181.h', 'answ181.o', 'answ182', 'answ182.c', 'answ182.h', 'answ182.o', 'answ183', 'answ183.c', 'answ183.h', 'answ183.o', 'answ184', 'answ184.c', 'answ184.h', 'answ184.o', 'answ185', 'answ185.c', 'answ185.h', 'answ185.o', 'answ186', 'answ186.c', 'answ186.h', 'answ186.o', 'answ187', 'answ187.c', 'answ187.h', 'answ187.o', 'answ188', 'answ188.c', 'answ188.h', 'answ188.o', 'answ189', 'answ189.c', 'answ189.h', 'answ189.o', 'answ190', 'answ190.c', 'answ190.h', 'answ190.o', 'answ191', 'answ191.c', 'answ191.h', 'answ191.o', 'answ192', 'answ192.c', 'answ192.h', 'answ192.o', 'answ193', 'answ193.c', 'answ193.h', 'answ193.o', 'answ194', 'answ194.c', 'answ194.h', 'answ194.o', 'answ195', 'answ195.c', 'answ195.h', 'answ195.o', 'answ196', 'answ196.c', 'answ196.h', 'answ196.o', 'answ197', 'answ197.c', 'answ197.h', 'answ197.o', 'answ198', 'answ198.c', 'answ198.h', 'answ198.o', 'answ199', 'answ199.c', 'answ199.h', 'answ199.o', 'answ200', 'answ200.c', 'answ200.h', 'answ200.o', 'answ201', 'answ201.c', 'answ201.h', 'answ201.o', 'answ202', 'answ202.c', 'answ202.h', 'answ202.o', 'answ203', 'answ203.c', 'answ203.h', 'answ203.o', 'answ204', 'answ204.c', 'answ204.h', 'answ204.o', 'answ205', 'answ205.c', 'answ205.h', 'answ205.o', 'answ206', 'answ206.c', 'answ206.h', 'answ206.o', 'answ207', 'answ207.c', 'answ207.h', 'answ207.o', 'answ208', 'answ208.c', 'answ208.h', 'answ208.o', 'answ209', 'answ209.c', 'answ209.h', 'answ209.o', 'answ210', 'answ210.c', 'answ210.h', 'answ210.o', 'answ211', 'answ211.c', 'answ211.h', 'answ211.o', 'answ212', 'answ212.c', 'answ212.h', 'answ212.o', 'answ213', 'answ213.c', 'answ213.h', 'answ213.o', 'answ214', 'answ214.c', 'answ214.h', 'answ214.o', 'answ215', 'answ215.c', 'answ215.h', 'answ215.o', 'answ216', 'answ216.c', 'answ216.h', 'answ216.o', 'answ217', 'answ217.c', 'answ217.h', 'answ217.o', 'answ218', 'answ218.c', 'answ218.h', 'answ218.o', 'answ219', 'answ219.c', 'answ219.h', 'answ219.o', 'answ220', 'answ220.c', 'answ220.h', 'answ220.o', 'answ221', 'answ221.c', 'answ221.h', 'answ221.o', 'answ222', 'answ222.c', 'answ222.h', 'answ222.o', 'answ223', 'answ223.c', 'answ223.h', 'answ223.o', 'answ224', 'answ224.c', 'answ224.h', 'answ224.o', 'answ225', 'answ225.c', 'answ225.h', 'answ225.o', 'answ226', 'answ226.c', 'answ226.h', 'answ226.o', 'answ227', 'answ227.c', 'answ227.h', 'answ227.o', 'answ228', 'answ228.c', 'answ228.h', 'answ228.o', 'answ229', 'answ229.c', 'answ229.h', 'answ229.o', 'answ230', 'answ230.c', 'answ230.h', 'answ230.o', 'answ231', 'answ231.c', 'answ231.h', 'answ231.o', 'answ232', 'answ232.c', 'answ232.h', 'answ232.o', 'answ233', 'answ233.c', 'answ233.h', 'answ233.o', 'answ234', 'answ234.c', 'answ234.h', 'answ234.o', 'answ235', 'answ235.c', 'answ235.h', 'answ235.o', 'answ236', 'answ236.c', 'answ236.h', 'answ236.o', 'answ237', 'answ237.c', 'answ237.h', 'answ237.o', 'answ238', 'answ238.c', 'answ238.h', 'answ238.o', 'answ239', 'answ239.c', 'answ239.h', 'answ239.o', 'answ240', 'answ240.c', 'answ240.h', 'answ240.o', 'answ241', 'answ241.c', 'answ241.h', 'answ241.o', 'answ242', 'answ242.c', 'answ242.h', 'answ242.o', 'answ243', 'answ243.c', 'answ243.h', 'answ243.o', 'answ244', 'answ244.c', 'answ244.h', 'answ244.o', 'answ245', 'answ245.c', 'answ245.h', 'answ245.o', 'answ246', 'answ246.c', 'answ246.h', 'answ246.o', 'answ247', 'answ247.c', 'answ247.h', 'answ247.o', 'answ248', 'answ248.c', 'answ248.h', 'answ248.o', 'answ249', 'answ249.c', 'answ249.h', 'answ249.o', 'answ250', 'answ250.c', 'answ250.h', 'answ250.o', 'answ251', 'answ251.c', 'answ251.h', 'answ251.o', 'answ252', 'answ252.c', 'answ252.h', 'answ252.o', 'answ253', 'answ253.c', 'answ253.h', 'answ253.o', 'answ254', 'answ254.c', 'answ254.h', 'answ254.o', 'answ255', 'answ255.c', 'answ255.h', 'answ255.o', 'answ256', 'answ256.c', 'answ256.h', 'answ256.o', 'answ257', 'answ257.c', 'answ257.h', 'answ257.o', 'answ258', 'answ258.c', 'answ258.h', 'answ258.o', 'answ259', 'answ259.c', 'answ259.h', 'answ259.o', 'answ260', 'answ260.c', 'answ260.h', 'answ260.o', 'answ261', 'answ261.c', 'answ261.h', 'answ261.o', 'answ262', 'answ262.c', 'answ262.h', 'answ262.o', 'answ263', 'answ263.c', 'answ263.h', 'answ263.o', 'answ264', 'answ264.c', 'answ264.h', 'answ264.o', 'answ265', 'answ265.c', 'answ265.h', 'answ265.o', 'answ266', 'answ266.c', 'answ266.h', 'answ266.o', 'answ267', 'answ267.c', 'answ267.h', 'answ267.o', 'answ268', 'answ268.c', 'answ268.h', 'answ268.o', 'answ269', 'answ269.c', 'answ269.h', 'answ269.o', 'answ270', 'answ270.c', 'answ270.h', 'answ270.o', 'answ271', 'answ271.c', 'answ271.h', 'answ271.o', 'answ272', 'answ272.c', 'answ272.h', 'answ272.o', 'answ273', 'answ273.c', 'answ273.h', 'answ273.o', 'answ274', 'answ274.c', 'answ274.h', 'answ274.o', 'answ275', 'answ275.c', 'answ275.h', 'answ275.o', 'answ276', 'answ276.c', 'answ276.h', 'answ276.o', 'answ277', 'answ277.c', 'answ277.h', 'answ277.o', 'answ278', 'answ278.c', 'answ278.h', 'answ278.o', 'answ279', 'answ279.c', 'answ279.h', 'answ279.o', 'answ280', 'answ280.c', 'answ280.h', 'answ280.o', 'answ281', 'answ281.c', 'answ281.h', 'answ281.o', 'answ282', 'answ282.c', 'answ282.h', 'answ282.o', 'answ283', 'answ283.c', 'answ283.h', 'answ283.o', 'answ284', 'answ284.c', 'answ284.h', 'answ284.o', 'answ285', 'answ285.c', 'answ285.h', 'answ285.o', 'answ286', 'answ286.c', 'answ286.h', 'answ286.o', 'answ287', 'answ287.c', 'answ287.h', 'answ287.o', 'answ288', 'answ288.c', 'answ288.h', 'answ288.o', 'answ289', 'answ289.c', 'answ289.h', 'answ289.o', 'answ290', 'answ290.c', 'answ290.h', 'answ290.o', 'answ291', 'answ291.c', 'answ291.h', 'answ291.o', 'answ292', 'answ292.c', 'answ292.h', 'answ292.o', 'answ293', 'answ293.c', 'answ293.h', 'answ293.o', 'answ294', 'answ294.c', 'answ294.h', 'answ294.o', 'answ295', 'answ295.c', 'answ295.h', 'answ295.o', 'answ296', 'answ296.c', 'answ296.h', 'answ296.o', 'answ297', 'answ297.c', 'answ297.h', 'answ297.o', 'answ298', 'answ298.c', 'answ298.h', 'answ298.o', 'answ299', 'answ299.c', 'answ299.h', 'answ299.o', 'answ300', 'answ300.c', 'answ300.h', 'answ300.o', 'answ301', 'answ301.c', 'answ301.h', 'answ301.o', 'answ302', 'answ302.c', 'answ302.h', 'answ302.o', 'answ303', 'answ303.c', 'answ303.h', 'answ303.o', 'answ304', 'answ304.c', 'answ304.h', 'answ304.o', 'answ305', 'answ305.c', 'answ305.h', 'answ305.o', 'answ306', 'answ306.c', 'answ306.h', 'answ306.o', 'answ307', 'answ307.c', 'answ307.h', 'answ307.o', 'answ308', 'answ308.c', 'answ308.h', 'answ308.o', 'answ309', 'answ309.c', 'answ309.h', 'answ309.o', 'answ310', 'answ310.c', 'answ310.h', 'answ310.o', 'answ311', 'answ311.c', 'answ311.h', 'answ311.o', 'answ312', 'answ312.c', 'answ312.h', 'answ312.o', 'answ313', 'answ313.c', 'answ313.h', 'answ313.o', 'answ314', 'answ314.c', 'answ314.h', 'answ314.o', 'answ315', 'answ315.c', 'answ315.h', 'answ315.o', 'answ316', 'answ316.c', 'answ316.h', 'answ316.o', 'answ317', 'answ317.c', 'answ317.h', 'answ317.o', 'answ318', 'answ318.c', 'answ318.h', 'answ318.o', 'answ319', 'answ319.c', 'answ319.h', 'answ319.o', 'answ320', 'answ320.c', 'answ320.h', 'answ320.o', 'answ321', 'answ321.c', 'answ321.h', 'answ321.o', 'answ322', 'answ322.c', 'answ322.h', 'answ322.o', 'answ323', 'answ323.c', 'answ323.h', 'answ323.o', 'answ324', 'answ324.c', 'answ324.h', 'answ324.o', 'answ325', 'answ325.c', 'answ325.h', 'answ325.o', 'answ326', 'answ326.c', 'answ326.h', 'answ326.o', 'answ327', 'answ327.c', 'answ327.h', 'answ327.o', 'answ328', 'answ328.c', 'answ328.h', 'answ328.o', 'answ329', 'answ329.c', 'answ329.h', 'answ329.o', 'answ330', 'answ330.c', 'answ330.h', 'answ330.o', 'answ331', 'answ331.c', 'answ331.h', 'answ331.o', 'answ332', 'answ332.c', 'answ332.h', 'answ332.o', 'answ333', 'answ333.c', 'answ333.h', 'answ333.o', 'answ334', 'answ334.c', 'answ334.h', 'answ334.o', 'answ335', 'answ335.c', 'answ335.h', 'answ335.o', 'answ336', 'answ336.c', 'answ336.h', 'answ336.o', 'answ337', 'answ337.c', 'answ337.h', 'answ337.o', 'answ338', 'answ338.c', 'answ338.h', 'answ338.o', 'answ339', 'answ339.c', 'answ339.h', 'answ339.o', 'answ340', 'answ340.c', 'answ340.h', 'answ340.o', 'answ341', 'answ341.c', 'answ341.h', 'answ341.o', 'answ342', 'answ342.c', 'answ342.h', 'answ342.o', 'answ343', 'answ343.c', 'answ343.h', 'answ343.o', 'answ344', 'answ344.c', 'answ344.h', 'answ344.o', 'answ345', 'answ345.c', 'answ345.h', 'answ345.o', 'answ346', 'answ346.c', 'answ346.h', 'answ346.o', 'answ347', 'answ347.c', 'answ347.h', 'answ347.o', 'answ348', 'answ348.c', 'answ348.h', 'answ348.o', 'answ349', 'answ349.c', 'answ349.h', 'answ349.o', 'answ350', 'answ350.c', 'answ350.h', 'answ350.o', 'answ351', 'answ351.c', 'answ351.h', 'answ351.o', 'answ352', 'answ352.c', 'answ352.h', 'answ352.o', 'answ353', 'answ353.c', 'answ353.h', 'answ353.o', 'answ354', 'answ354.c', 'answ354.h', 'answ354.o', 'answ355', 'answ355.c', 'answ355.h', 'answ355.o', 'answ356', 'answ356.c', 'answ356.h', 'answ356.o', 'answ357', 'answ357.c', 'answ357.h', 'answ357.o', 'answ358', 'answ358.c', 'answ358.h', 'answ358.o', 'answ359', 'answ359.c', 'answ359.h', 'answ359.o', 'answ360', 'answ360.c', 'answ360.h', 'answ360.o', 'answ361', 'answ361.c', 'answ361.h', 'answ361.o', 'answ362', 'answ362.c', 'answ362.h', 'answ362.o', 'answ363', 'answ363.c', 'answ363.h', 'answ363.o', 'answ364', 'answ364.c', 'answ364.h', 'answ364.o', 'answ365', 'answ365.c', 'answ365.h', 'answ365.o', 'answ366', 'answ366.c', 'answ366.h', 'answ366.o', 'answ367', 'answ367.c', 'answ367.h', 'answ367.o', 'answ368', 'answ368.c', 'answ368.h', 'answ368.o', 'answ369', 'answ369.c', 'answ369.h', 'answ369.o', 'answ370', 'answ370.c', 'answ370.h', 'answ370.o', 'answ371', 'answ371.c', 'answ371.h', 'answ371.o', 'answ372', 'answ372.c', 'answ372.h', 'answ372.o', 'answ373', 'answ373.c', 'answ373.h', 'answ373.o', 'answ374', 'answ374.c', 'answ374.h', 'answ374.o', 'answ375', 'answ375.c', 'answ375.h', 'answ375.o', 'answ376', 'answ376.c', 'answ376.h', 'answ376.o', 'answ377', 'answ377.c', 'answ377.h', 'answ377.o', 'answ378', 'answ378.c', 'answ378.h', 'answ378.o', 'answ379', 'answ379.c', 'answ379.h', 'answ379.o', 'answ380', 'answ380.c', 'answ380.h', 'answ380.o', 'answ381', 'answ381.c', 'answ381.h', 'answ381.o', 'answ382', 'answ382.c', 'answ382.h', 'answ382.o', 'answ383', 'answ383.c', 'answ383.h', 'answ383.o', 'answ384', 'answ384.c', 'answ384.h', 'answ384.o', 'answ385', 'answ385.c', 'answ385.h', 'answ385.o', 'answ386', 'answ386.c', 'answ386.h', 'answ386.o', 'answ387', 'answ387.c', 'answ387.h', 'answ387.o', 'answ388', 'answ388.c', 'answ388.h', 'answ388.o', 'answ389', 'answ389.c', 'answ389.h', 'answ389.o', 'answ390', 'answ390.c', 'answ390.h', 'answ390.o', 'answ391', 'answ391.c', 'answ391.h', 'answ391.o', 'answ392', 'answ392.c', 'answ392.h', 'answ392.o', 'answ393', 'answ39

**airmiles.c**

```

/* Determines air mileage from New York to other cities */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct city_info {
 char *city;
 int miles;
};

int compare_cities(const void *key_ptr,
 const void *element_ptr);

main()
{
 # char city_name[81];
 struct city_info *ptr;
 const struct city_info mileage[] =
 {{"Acapulco", 2260}, {"Amsterdam", 3639},
 {"Antigua", 1783}, {"Aruba", 1963},
 {"Athens", 4927}, {"Barbados", 2100},
 {"Bermuda", 771}, {"Bogota", 2487},
 {"Brussels", 3662}, {"Buenos Aires", 5302},
 {"Caracas", 2123}, {"Copenhagen", 3849},
 {"Curacao", 1993}, {"Frankfurt", 3851},
 {"Geneva", 3859}, {"Glasgow", 3211},
 {"Hamburg", 3806}, {"Kingston", 1583},
 {"Lima", 3651}, {"Lisbon", 3366},
 {"London", 3456}, {"Madrid", 3588},
 {"Manchester", 3336}, {"Mexico City", 2086},
 {"Milan", 4004}, {"Nassau", 1101},
 {"Oslo", 3671}, {"Paris", 3628},
 {"Reykjavik", 2600}, {"Rio de Janeiro", 4816},
 {"Rome", 4280}, {"San Juan", 1609},
 {"Santo Domingo", 1560}, {"St. Croix", 1680},
 {"Tel Aviv", 5672}, {"Zurich", 3926});

printf("Enter city name: ");
scanf("%80[^n]", city_name);
ptr = bsearch(city_name, mileage,
 sizeof(mileage)/sizeof(mileage[0]),
 sizeof(mileage[0]), compare_cities);
if (ptr != NULL)
 printf("%s is %d miles from New York City.\n",
 city_name, ptr->miles);
else
 printf("%s wasn't found.\n", city_name);

return 0;
}

int compare_cities(const void *key_ptr,
 const void *element_ptr)
{
 return strcmp((char *) key_ptr,
 ((struct city_info *) element_ptr)->city);
}

```

### 26.2.8 整数算术运算函数

```

int abs(int j);
div_t div(int numer, int denom);
long int labs(long int j);
ldiv_t ldiv(long int numer, long int denom);

```

**abs**函数返回int型值的绝对值, **labs**函数返回long int型值的绝对值。

**div**函数用第一个实参数以第二个实参, 并且返回一个**div\_t**型值。**div\_t**是一个含有商成

员（命名为quot）和余数成员（命名为rem）的结构。例如，如果ans是一个div\_t型变量，那么可以写成

```
ans = div(5,2);
printf("Quotient: %d Remainder: %d\n", ans.quot, ans.rem);
```

**Q&A** ldiv函数和div函数很类似，不同之处在于ldiv函数用于处理长整数。ldiv函数返回的ldiv\_t型值也包含quot和rem两个成员。（在<stdlib.h>中定义了div\_t类型和ldiv\_t类型。）

## 26.3 <time.h>: 日期和时间

<time.h>提供的函数可以确定时间（和日期），可以在时间值上进行算术操作，还可以显示格式化的时间。然而，在介绍这些函数之前，需要讨论一下时间是如何存储的。<time.h>提供了三种类型，每种类型表示一种不同的存储时间的方法：

- clock\_t: 按照“时钟嘀嗒”进行测量的时间值。
- time\_t: 紧凑的编码时间和日期（日历时间）。
- struct tm: 把时间分解成秒、分、时等。经常把struct tm类型值称为分解时间。表26-1说明了tm结构的成员。全部成员都是int类型的。

表26-1 tm结构的成员

| 名 称      | 描 述        | 最 小 值 | 最 大 值           |
|----------|------------|-------|-----------------|
| tm_sec   | 分后边的秒      | 0     | 61 <sup>①</sup> |
| tm_min   | 时后边的分      | 0     | 59              |
| tm_hour  | 从午夜以后的时    | 0     | 23              |
| tm_mday  | 每月的天       | 1     | 31              |
| tm_mon   | 从一月以后的月份   | 0     | 11              |
| tm_year  | 从1900以后的年份 | 0     | —               |
| tm_wday  | 从星期日以后的天   | 0     | 6               |
| tm_yday  | 从一月一日以后的天  | 0     | 365             |
| tm_isdst | 白天省时标记     | 0     | 0               |

① 如果白天省时有效，就为正数；如果无效，就为零；如果信息是未知的，就为负数。

② 允许两个额外的“闰秒”。

上述这些类型有不同的用途。clock\_t值只善于表示时间区间。而time\_t值和struct tm值则可以存储完整的日期和时间。time\_t值是紧密编码，所以他们占用很少的空间。而struct tm值却要求较大的空间，但是这类值常常易于使用。C标准规定了clock\_t和time\_t必须是“算术运算类型”，姑且这样理解。结果是无法知道clock\_t值和time\_t值是要作为整数存储还是浮点数存储。

现在来看看<time.h>中的函数。这些函数分为两组：时间处理函数和时间转换函数。

### 26.3.1 时间处理函数



clock函数返回表示处理器时间的clock\_t值，程序从执行开始后就使用这个处理器时间了。为了把这个值转换为秒，将把值除以定义在<time.h>中的宏CLOCKS\_PER\_SEC。

当用clock函数来确定程序运行多长时间时，习惯做法是调用两次clock函数：一次在main函数开始处，一次在程序就要终止之前：

```
#include <time.h>

main()
{
 clock_t start_clock = clock();
 ...
 printf("Processor time used: %g sec.\n",
 (clock() - start_clock) / (double) CLOCKS_PER_SEC);
 return 0;
}
```

初始调用clock函数的理由是，由于隐藏的“启动”代码，程序会在到达main函数之前使用一些处理器时间。在main函数开始处调用clock函数可以确定启动代码需要多长时间，以便于稍候可以减去这部分时间。

C标准没有说明clock\_t是整数类型还是浮点类型，也不知道宏CLOCKS\_PER\_SEC的类型。结果是我们无法知道下列表达式的类型：

```
(clock() - start_time) / CLOCKS_PER_SEC
```

这样就很难用printf函数来显示内容。为了解决这个问题，这里的示例把宏CLOCKS\_PER\_SEC强制成double型，从而迫使整个表达式具有了double类型。

time函数返回当前的日历时间。如果实参不是空指针，那么time函数也会把日历时间存储在实参指向的对象中。time函数返回两种不同方式的时间的能力是历史遗留问题，但是它为用户提供了两种书写的选项，既可以是

```
curtime = time(NULL);
```

也可以是

```
time(&cur_time);
```

**577** 这里的cur\_time是具有time\_t类型的变量。

difftime函数返回time0（较早的时间）和time1之间按秒衡量的差值。因此，为了计算程序的实际运行时间（不是处理器时间），可以采用下列代码：

```
#include <time.h>

main()
{
 time_t start_time = time(NULL);
 ...
 printf("Running time: %g sec.\n",
 difftime(time(NULL), start_time));
 return 0;
}
```

mktime函数把分解时间（存储在函数参数指向的结构中）转换为日历时间，然后返回。作为副作用，mktime函数会根据下列规则调整结构的成员：

- mktime函数会改变其值不在合法范围内的任何成员（表26-1），这样的改变可能会依次要求改变其他成员。例如，如果tm\_sec过大，那么mktime函数会把它减少到合适的范围内（0~59），并且会为tm\_min增加额外的分钟数。如果现在tm\_min过大，那么mktime函数会减少tm\_min，同时为tm\_hour增加额外的小时数。如果必要，此过程还将继续对成员tm\_mday、成员tm\_mon和成员tm\_year进行操作。
- 在调整完结构的其他成员后，（如果必要）mktime函数会给tm\_wday（按星期算的天）和tm\_yday（按年算的天）设置正确的值。在调用mktime函数之前，从来不需要对tm\_wday

和`tm_yday`的值进行任何初始化，因为`mktme`函数会忽略这些成员的初始值。

`mktme`函数调整`tm`结构成员的能力对于和时间相关的算术计算非常有用。例如，现在用`mktme`函数来回答下列这个问题：如果1996年的奥林匹克运动会从7月19日开始，并且历时16天，那么请说出结束的日期是哪天？这里将把开始日期1996年7月19日存储在`tm`结构中：

```
struct tm t;
t.tm_mday = 19;
t.tm_mon = 6; /* July */
t.tm_year = 96; /* 1996 */
```

为了确保结构的其他成员不包含可能影响结果的垃圾值，这里还要对这些成员也进行初始化（成员`tm_wday`和`tm_yday`除外）：

```
t.tm_sec = 0;
t.tm_min = 0;
t.tm_hour = 0;
t.tm_isdst = -1;
```

578

接下来，将把16添加给成员`tm_mday`：

```
t.tm_mday += 16;
```

这样操作的结果使在成员`tm_mday`中产生了35，它超出了成员的取值范围。调用`mktme`函数将会使结构的成员恢复到正确的取值范围内：

```
mktme(&t);
```

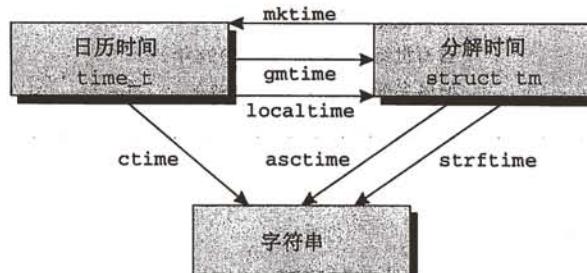
这里将舍弃`mktme`函数的返回值，因为我们只对在`t`上的函数效果感兴趣。现在，和`t`相关的成员具有下列值：

| 成 员                  | 值   | 含 义                   |
|----------------------|-----|-----------------------|
| <code>tm_mday</code> | 4   | 4                     |
| <code>tm_mon</code>  | 7   | August                |
| <code>tm_year</code> | 96  | 1996                  |
| <code>tm_wday</code> | 0   | Sunday                |
| <code>tm_yday</code> | 216 | 217th day of the year |

## 26.3.2 时间转换函数

```
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char *s, size_t maxsize,
 const char *format,
 const struct tm *timeptr);
```

时间转换函数可以把日历时间转换成分解时间，还可以把时间（日历时间或分解时间）转换成字符串格式。下图说明了这些函数之间的关联关系。



579

此图包含了mktime函数。C标准把此函数划分为“处理”函数而不是“转换”函数。

gmtime函数和localtime函数很类似。当程序把指针传递给日历时间时，这两种函数都会返回一个指向结构的指针，且指向的结构含有等价的分解时间。**Q&A** localtime函数会产生本地时间，而gmtime函数的返回值则是用UTC（协调世界时间）表示的。

asctime（ASCII时间）函数返回指向类似Tue Aug 30 17:07:12 1994\n格式串的指针，且此字符串的构造来源于函数参数所指向的分解时间。字符串存储在静态变量中，每次调用asctime函数都可以修改此静态变量。

ctime函数返回指向字符串的指针，且该指针所指向的字符串是描述本地时间的。调用ctime(&t)就等价于调用asctime(localtime(&t))。

strftime函数和asctime函数一样，也把分解时间转换成字符串格式。然而，不同于asctime函数的是，strftime函数可以提供大量对时间格式化方法的控制。事实上，strftime函数类似于sprintf函数（>22.8节），因为strftime函数会根据格式串（函数的第三个参数）把字符“写入”到字符串s（函数的第一个参数）中。格式串可能含有普通字符和表26-2显示的转换说明符。其中普通字符不会改变原样地复制给字符串s。函数的最后一个参数指向tm结构，此结构用来作为数据和时间信息源。函数的第二个参数是对存储在字符串s中的字符数量的限制。

表26-2 用于strftime函数的转换说明符

| 转换说明符 | 替换的内容                             |
|-------|-----------------------------------|
| %a    | 缩写的星期名（比如，Tue）                    |
| %A    | 完整的星期名（比如，Tuesday）                |
| %b    | 缩写的月份名（比如，Aug）                    |
| %B    | 完整的月份名（比如，August）                 |
| %c    | 完全的日期和时间（比如，Aug 30 17:07:12 1994） |
| %d    | 月内的天（01~31）                       |
| %H    | 24小时制的小时（00~23）                   |
| %I    | 12小时制的小时（01~12）                   |
| %j    | 年内的天（001~366）                     |
| %m    | 月份（1~12）                          |
| %M    | 分钟（00~59）                         |
| %p    | AM/PM指示符（AM或PM）                   |
| %S    | 秒（00~61） <sup>①</sup>             |
| %U    | 星期数（00-53） <sup>②</sup>           |
| %w    | 星期几（0-6）                          |
| %W    | 星期数量（00-53） <sup>③</sup>          |
| %x    | 完全的日期（比如，Aug 30 1994）             |
| %X    | 完全的时间（比如，17:07:12）                |
| %y    | 不含世纪的年（00~99）                     |
| %Y    | 含有世纪的年（比如，1994年）                  |
| %Z    | 时区名或缩写（比如，EST）                    |
| %%    | %                                 |

① 允许两个额外的“闰秒”。

② 把第一个星期日看作是第一个星期的开始。

③ 把第一个星期看作是第一个星期的开始。

strftime函数不同于<time.h>中的其他函数，它对当前地区（地区>25.1节）是很敏感的。

改变LC\_TIME类型可能会影响转换说明符的行为。表26-2中的例子严格地设为"C"地区。在德国地区内，%A可能会产生Dienstag而不是Tuesday。

### 26.3.3 程序：显示日期和时间

现在需要一个显示当前日期和时间的程序。当然，程序的第一步是要调用time函数来获得日历时间。第二步是把时间转换成字符串格式并显示出来。最简单的做法就是第二步调用ctime函数，它会返回一个指向含有日期和时间的字符串的指针，然后把此指针传递给puts函数或printf函数。

到目前为止，一切方法都没问题。可是，我们希望程序按照特定方式显示的日期和时间究竟是什么呢？假设这里需要如下形式的显示格式：

08-30-94 5:07p

ctime函数一直采用相同的日期和时间格式，所以对此无能为力。strftime函数相对好一些。使用strftime函数可以基本达到希望的显示要求，但是strftime函数无法显示没有零开头的单数字小时数。而且，strftime函数使用AM和PM而不是a和p来表示时间。[580]

既然strftime函数不够好，那么这里还有另外一种选择：把日历时间转换为分解时间，然后从结构中抽取相关的信息，同时自己用printf函数或类似的函数把信息进行格式化。我们甚至可以使用strftime函数来实现某些格式化，然后用其他函数来完成整个工作。

下面这个程序举例说明了选择方案。程序用三种方法显示了当前日期和时间：一种格式是由ctime函数格式化的，一种格式是接近于我们需求的（由strftime函数产生的），还有一种则是正确的格式（由printf函数产生的）。采用ctime函数版本容易实现，采用strftime函数版本则有点困难，而采用printf函数版本更困难。

#### *datetime.c*

```
/* Displays the current date and time in three formats */
#include <stdio.h>
#include <time.h>

main()
{
 time_t current = time(NULL);
 struct tm *ptr;
 char date_time[19];
 int hour;
 char am_or_pm;
 /* print date and time in default format */
 puts(ctime(¤t));

 /* print date and time using strftime to format */
 strftime(date_time, sizeof(date_time),
 "%m-%d-%y %I:%M%p\n", localtime(¤t));
 puts(date_time);

 /* print date and time using custom formatting */
 ptr = localtime(¤t);
 hour = ptr->tm_hour;
 if (hour <= 11)
 am_or_pm = 'a';
 else {
 hour -= 12;
 am_or_pm = 'p';
 }
 if (hour == 0)
 hour = 12;
 printf("%2d-%2d-%2d %2d:%2d%c\n", ptr->tm_mon+1,
```

[580]

[581]

```

ptr->tm_mday, ptr->tm_year, hour, ptr->tm_min,
am_or_pm);

return 0;
}

```

此程序的输出如下所示：

```

Tue Aug 30 17:07:12 1994
08-30-94 05:07PM
08-30-94 5:07p

```

## 问与答

**问：**虽然`<stdlib.h>`提供了6种把字符串转换成数的函数，但是它没有出现任何把数转换成字符串的函数。提供什么了呢？

**答：**某些C的库提供名为`itoa`的函数可以把数转换成为字符串。但是，使用这类函数不是一个好主意，因为它们不是C标准的内容且无法移植。把数转换成为字符串的最好做法就是调用`sprintf`函数（>22.8节）：

```

char s[10];
int i;

sprintf(s, "%d", i); /* stores i in the string s */

```

582

`sprintf`函数不但可以移植，而且可以对数的显示提供大量的控制。

**\*问：**`abort`函数和SIGABRT信号之间是否存在联系呢？(p.398)

**答：**存在。调用时，`abort`函数实际产生SIGABRT信号。如果没有SIGABRT的处理函数，那么程序会像26.2节描述的那样异常终止。如果为SIGABRT安装了处理函数（通过调用`signal`函数>24.3.2节），那么就调用处理函数。如果处理函数返回，那么随后程序会异常终止。但是，如果处理函数不返回（比如它调用了`longjmp`函数（>24.4节）），那么程序就无法终止。

**问：**难道就没有办法为`bsearch`函数或`qsort`函数避免比较函数中所有讨厌的强制类型转换吗？

**答：**有，虽然在程序的其他地方会包含强制类型转换。一起来看看用于`airmiles.c`程序中的比较函数：

```

int compare_cities(const void *key_ptr,
 const void *element_ptr
{
 return strcmp((char *) key_ptr,
 ((struct city_info *)element_ptr)->city);
}

```

我们可以用更自然的方式编写此函数，且不会有强制类型转换：

```

int compare_cities(const void *key_ptr,
 const struct city_info *element_ptr)
{
 return strcmp(key_ptr, element_ptr->city);
}

```

但是，我们无法把新版的`compare_cities`函数传递给`bsearch`函数，后者需要指向函数的指针，且此函数必须带有两个`void*`类型的参数。解决方案是为`bsearch`函数调用添加一个强制类型转换：

```

ptr = bsearch(city_name, mileage,
 sizeof(mileage)/sizeof(mileage[0]),
 sizeof(mileage[0]),
 (int (*) (const void *, const void *))
 compare_cities);

```

这种方法是否可以使程序更易读？这需要你自己来评判。

**问：**为什么存在`div`函数和`ldiv`函数呢？难道只用/和%运算符不行吗？(p.401)

答: `div`函数和`ldiv`函数同`/`运算符和`%`运算符不完全一样。回顾4.1节就会知道把`/`运算符和`%`运算用于负的运算数无法得到可移植的结果。如果*i*和*j*为负数, 那么*i/j*的值是向上舍入还是向下舍入是由实现定义的, 就像*i%j*结果的符号一样。但是, 由`div`函数和`ldiv`函数计算的答案是不依赖于实现的。商向零舍入, 余数则根据公式 $n=q \times d + r$ 计算得出。公式里的*n*是原始数, *q*是商, *d*是除数, 而*r*是余数。下面是几个例子:

| <i>n</i> | <i>d</i> | <i>q</i> | <i>r</i> |
|----------|----------|----------|----------|
| 7        | 3        | 2        | 1        |
| -7       | 3        | 2        | 1        |
| 7        | -3       | -2       | -1       |
| -7       | -3       | 2        | -1       |

效率则是`div`函数和`ldiv`函数存在的另一个原因。许多机器可以在一条指令里计算出商和余数, 所以调用`div`函数或`ldiv`函数比分别使用`/`运算符和`%`运算符要快得多。

问: `gmtime`函数名字的出处在哪?

(p.404)  
答: 编程语言不是唯一被标准化的内容。国际时间在1883年标准化时诞生了24个时区。因为许多人(特别是航海家和天文学家)需要一种方法来规定绝对时间而不是对某个时区而言的相对时间, 所以建立格林威治标准时间(Greenwich Mean Time)就是基于子午线横穿英国的格林威治。最近, 虽然格林威治标准时间又被重新命名为协调世界时(Goordinated Universal Time), 但是人们早已经广泛使用`gmtime`函数了。

## 练习

### 26.1节

- 重新编写`max_int`函数, 要求不再把传递整数的个数作为第一个参数, 我们必须采用0作为最后一个参数。**提示:** `max_int`函数必须至少有一个“正常”参数, 所以不能把参数*n*移走, 相反假设它是要比较的数之一。
- 编写`printf`函数的简写版, 要求新函数只有一种转换说明`%d`, 并且在第一个参数后边的所有参数都必须是`int`类型的。
- 编写下列函数:

```
char *vstrcat(const char *first, ...);
```

假设`vstrcat`函数除最后一个参数必须是个空指针(强制成`char *`类型)外, 全部参数都是字符串。函数返回的指针指向动态分配的且含有参数拼接的字符串。如果没有足够的内存, 那么`vstrcat`函数应该返回空指针。**提示:** `vstrcat`函数必须遍历参数两次: 一次用来确定返回字符串需要的内存数量, 另一次用来把参数复制到字符串中。

### 26.2节

- 解释说明下列语句的含义。假设`value`是`long int`型的变量, `p`是`char*`型的变量。  
**Value = strtol(p, &p, 10);**
- 编写一条可以随机从7、11、15或19中取一个数分配给变量*n*的语句。
- 编写一个可以随机返回`double`型值*d* ( $0.0 \leq d < 1.0$ ) 的函数。
- 编写一个程序, 用来模拟称为“掷双骰”游戏。程序要通过随机选择1到6之间的两个数来“滚动”一对模拟的骰子。如果两个数的和是7或11, 那么程序显示信息Player wins。如果和为2、3或12, 则显示Player loses。否则, 程序要重复滚动骰子直到再一次达到原始和(Player wins)或者骰子合计为7(Player loses)为止。程序需要在每次模拟滚动后显示一下骰子的值。
- (a) 编写一个程序, 使它可以调用`rand`函数1000次并且显示函数返回的每个值的最低位(如果返回值是偶数, 则为0; 如果返回值为奇数, 则为1。) 你看到过什么模式吗? (`rand`的返回值的最后几位往往不是特别随机的。)

- (b) 如何改进rand函数的随机性，使它可以在一个小范围内产生数？
9. 编写两个函数来测试atexit函数。一个函数显示That's all,, 另一个显示folks!。在程序终止时用atexit函数来注册这两个要调用的函数。请一定确保这两个函数按照正确顺序进行调用，只有这样才可以在屏幕上看到That's all, folks!。

### 26.3节

10. 编写一个程序，用clock函数来测算qsort函数对有100个整数的数组进行排序所用的时间，其中此数组初始时是反序排列元素的。确保该程序还可以用于有1000个整数的数组或10 000个整数的数组。
11. 编写一个函数，要求当向此函数传递年（比如，1996）时，函数返回一个表示在年份开始处的time\_t值（即……第1小时的第1分钟的第1秒）。
12. 编写一个程序，提示用户录入一个日期（月、日和年）和一个整数n，然后显示n天后的日期。
13. 编写一个程序，提示用户录入两个日期，然后显示两个日期之间相差的天数。提示：请使用mktime函数和difftime函数。
14. 编写一个可以按照下列每种格式显示当前日期和时间的程序。请使用strftime函数来完成全部或大部分格式化的工作。

- (a) Tuesday, August 30, 1994                    05:07p  
(b) Tue, 30 Aug 94                    17:07  
(c) 08/30/94                    5:07:12 PM

# 计算机精品学习资料大放送

[软考官方指定教材及同步辅导书下载](#) | [软考历年真题解析与答案](#)

[软考视频](#) | [考试机构](#) | [考试时间安排](#)

[Java 一览无余：Java 视频教程](#) | [Java SE](#) | [Java EE](#)

[.Net 技术精品资料下载汇总：ASP.NET 篇](#)

[.Net 技术精品资料下载汇总：C#语言篇](#)

[.Net 技术精品资料下载汇总：VB.NET 篇](#)

[撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程](#)

[Visual C++\(VC/MFC\)学习电子书及开发工具下载](#)

[Perl/CGI 脚本语言编程学习资源下载地址大全](#)

[Python 语言编程学习资料\(电子书+视频教程\)下载汇总](#)

[最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载](#)

[数据库管理系统\(DBMS\)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇](#)

[平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载](#)

[最强 HTML/xHTML、CSS 精品学习资料下载汇总](#)

[最新 JavaScript、Ajax 典藏级学习资料下载分类汇总](#)

[网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总](#)

[UML 学习电子资下载汇总 软件设计与开发人员必备](#)

[经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通](#)

[天罗地网：精品 Linux 学习资料大收集\(电子书+视频教程\) Linux 参考资源大系](#)

[Linux 系统管理员必备参考资料下载汇总](#)

[Linux shell、内核及系统编程精品资料下载汇总](#)

[UNIX 操作系统精品学习资料<电子书+视频>分类总汇](#)

[FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频](#)

[Solaris/OpenSolaris 电子书、视频等精华资料下载索引](#)

[>> 更多精品资料请访问大家论坛计算机区...](#)