第16章

标准函数库

标准函数库是一个工具箱，它极大地扩展了C程序员的能力。但是，在你使用这个能力之前，你必须熟悉库函数。忽略函数库相当于你只学习怎样使用油门、方向盘和刹车来开车，却不想费神学习使用自动恒速器、收音机和空调。虽然你仍然能够驾车到达你想去的地方，但过程要艰难一些，乐趣也要少很多。

本章描述前面章节未曾覆盖的一些库函数。各小节的标题中包括了获得这些函数原型必须用#include指令包含的文件名。

16.1 整型函数

这组函数返回整型值。这些函数分为三类：算术、随机数和字符串转换。

16.1.1 算术 <stdlib.h>

标准函数库包含了4个整型算术函数。

```
int abs( int value );
long int labs( long int value );
div_t div( int numerator, int denominator );
ldiv_t ldiv( long int numer, long int denom );
```

abs函数返回它的参数的绝对值。如果其结果不能用一个整数表示，这个行为是未定义的。labs用于执行相同的任务，但它的作用对象是长整数。

div函数把它的第2个参数（分母）除以第1个参数（分子），产生商和余数，用一个div_t结构返回。这个结构包含下面两个字段，

```
int  quot;   //  商
int  rem;    //  余数
```

但这两个字段并不一定以这个顺序出现。如果不能整除，商将是所有小于代数商的整数中最靠近它的那个整数。注意/操作符的除法运算结果并未精确定义。当/操作符的任何一个操作数为负而不能整除时，到底商是最大的那个小于等于代数商的整数还是最小的那个大于等于代数商的整数，这取决于编译器。ldiv所执行的任务和div相同，但它作用于长整数，其返回值是一

个 ldiv_t 结构。

16.1.2 随机数<stdlib.h>

有些程序每次执行时不应该产生相同的结果，如游戏和模拟，此时随机数就非常有用。下面两个函数合在一起使用能够产生**伪随机数**(pseudo-random number)。之所以如此称呼是因为它们通过计算产生随机数，因此有可能重复出现，所以并不是真正的随机数。

```
int  rand( void );
void  srand( unsigned int seed );
```

rand 返回一个范围在 0 和 RAND_MAX（至少为 32,767）之间的伪随机数。当它重复调用时，函数返回这个范围内的其他数。为了得到一个更小范围的伪随机数，首先把这个函数的返回值根据所需范围的大小进行取模，然后通过加上或减去一个偏移量对它进行调整。

为了避免程序每次运行时获得相同的随机数序列，我们可以调用 srand 函数。它用它的参数值对随机数发生器进行初始化。一个常用的技巧是使用每天的时间作为随机数产生器的**种子**(seed)，如下面的程序所示：

```
srand( (unsigned int)time( 0 ) );
```

time 函数将在本章后面描述。

程序 16.1 中的函数使用整数来表示游戏用的牌并使用随机数在“牌桌”上“洗”指定数目的牌。

```
/*
** 使用随机数在牌桌上洗“牌”。第 2 个参数指定牌的数字。当这个函数第 1 次调用
** 时，调用 srand 函数初始化随机数发生器。
*/
#include <stdlib.h>
#include <time.h>
#define   TRUE      1
#define   FALSE     0

void shuffle( int *deck, int n_cards )
{
      int i;
      static   int  first_time = TRUE;

/*
** 如果尚未进行初始化，用当天的当前时间作为随机数发生器。
**
*/
if( first_time ){
         first_time = FALSE;
         srand( (unsigned int)time( NULL ) );
}

 /*
** 通过交换随机对的牌进行“洗牌”。
*/
for( i = n_cards - 1; i > 0; i -= 1 ){
          int   where;
          int   temp;

          where = rand() % i;
```

```
            temp = deck[ where ];
            deck[ where ] = deck[ i ];
            deck[ i ] = temp;
        }
    }
```

程序 16.1　用随机数洗牌　　shuffle.c

16.1.3　字符串转换 <stdlib.h>

字符串转换函数把字符串转换为数值。其中最简单的函数 atoi 和 atol，执行基数为 10 的转换。strtol 和 strtoul 函数允许你在转换时指定基数，同时它们还允许你访问字符串的剩余部分。

```
int atoi( char const *string );
long int atol( char const *string );
long int strtol( char const *string, char **unused, int base );
unsigned long int strtoul( char const *string, char **unused,
    int base );
```

如果任何一个上述函数的第 1 个参数包含了前导空白字符，它们将被跳过。然后函数把合法的字符转换为指定类型的值。如果存在任何非法缀尾字符，它们也将被忽略。

atoi 和 atol 分别把字符转换为整数和长整数值。strtol 和 atol 同样把参数字符串转换为 long。但是，strtol 保存一个指向转换值后面第 1 个字符的指针。如果函数的第 2 个参数并非 NULL，这个指针便保存在第 2 个参数所指向的位置。这个指针允许字符串的剩余部分进行处理而无需推测转换在字符串的哪个位置终止。strtoul 和 strtol 的执行方式相同，但它产生一个无符号长整数。

这两个函数的第 3 个参数是转换所执行的基数。如果基数为 0，任何在程序中用于书写整数字面值的形式都将被接受，包括指定数字基数的形式，如 0x2af4 和 0377。否则，基数值应该在 2 到 36 的范围内——然后转换根据这个给定的基数进行。对于基数 11 到 36，字母 A 到 Z 分别被解释为数值 10 到 35。在这个上下文环境中，小写字母 a-z 被解释为与对应的大写字母相同的意思。因此，

```
x = strtol("   590bear", next, 12 );
```

的返回值为 9947，并把一个指向字母 e 的指针保存在 next 所指向的变量中。转换在 b 处终止，因为在基数为 12 时 e 不是一个合法的数字。

如果这些函数的 string 参数中并不包含一个合法的数值，函数就返回 0。如果被转换的值无法表示，函数便在 errno 中存储 ERANGE 这个值，并返回表 16.1 中的一个值。

表 16.1　strtol 和 strtoul 返回的错误值

函　数	返　回　值
strtol	如果值太大且为负数，返回 LONG_MIN。如果值太大且为正数，返回 LONG_MAX
strtoul	如果值太大，返回 ULONG_MAX

16.2　浮点型函数

头文件 math.h 包含了函数库中剩余的数学函数的声明。这些函数的返回值以及绝大多数参数都是 double 类型。

警告：

一个常见的错误就是在使用这些函数时忘了包含这个头文件，如下所示：

```
double  x;
x = sqrt( 5.5 );
```

编译器在此之前未曾见到过 sqrt 函数的原型，因此错误地假定它返回一个整数，然后错误地把这个值的类型转换为 double。这个结果值是没有意义的。

如果一个函数的参数不在该函数的定义域之内，称为**定义域错误**(domain error)。例如：

```
sqrt( -5.0 );
```

就是个定义域错误，因为负值的平方根是未定义的。当出现一个定义域错误时，函数返回一个由编译器定义的错误值，并且在 errno 中存储 EDOM 这个值。如果一个函数的结果值过大或过小，无法用 double 类型表示，这称为**范围错误**(range error)。例如：

```
exp( DBL_MAX )
```

将产出一个范围错误，因为它的结果值太大。在这种情况下，函数将返回 HUGE_VAL，它是一个在 math.h 中定义的 double 类型的值。如果一个函数的结果值太小，无法用一个 double 表示，函数将返回 0。这种情况也属于范围错误，但 errno 会不会设置为 ERANGE 则取决于编译器。

16.2.1 三角函数 <math.h>

标准函数库提供了常见的三角函数。

```
double sin( double angle );
double cos( double angle );
double tan( double angle );
double asin( double value );
double acos( double value );
double atan( double value );
double atan2( double x, double y );
```

sin、cos 和 tan 函数的参数是一个用弧度表示的角度，这些函数分别返回这个角度的正弦、余弦和正切值。

asin、acos 和 atan 函数分别返回它们的参数的反正弦、反余弦和反正切值。如果 asin 和 acos 的参数并不位于-1 和 1 之间，就出现一个定义域错误。asin 和 atan 的返回值是范围在$-\pi/2$ 和 $\pi/2$ 之间的一个弧度，acos 的返回值是一个范围在 0 和 π 之间的一个弧度。

atan2 函数返回表达式 y/x 的反正切值，但它使用这两个参数的符号来决定结果值位于哪个象限。它的返回值是一个范围在$-\pi$ 和 π 之间的弧度。

16.2.2 双曲函数 <math.h>

```
double sinh( double angle );
double cosh( double angle );
double tanh( double angle );
```

这些函数分别返回它们的参数的双曲正弦、双曲余弦和双曲正切值。每个函数的参数都是一个以弧度表示的角度。

16.2.3 对数和指数函数 <math.h>

标准函数库存在一些直接处理对数和指数的函数。

```
double exp( double x );
double log( double x );
double log10( double x );
```

exp 函数返回 e 值的 x 次幂，也就是 e^x。

log 函数返回 x 以 e 为底的对数，也就是常说的自然对数。log10 函数返回 x 以 10 为底的对数。注意 x 以任意一个以 b 为底的对数可以通过下面的公式进行计算：

$$\log b^x = \frac{\log e^x}{\log e^b}$$

如果它们的参数为负数，两个对数函数都将出现定义域错误。

16.2.4　浮点表示形式 <math.h>

这三个函数提供了一种根据一个编译器定义的格式存储一个浮点值的方法。

```
double frexp( double value, int *exponent );
double ldexp( double fraction, int exponent );
double modf( double value, double *ipart );
```

frexp 函数计算一个**指数**(exponent)和**小数**(fraction)，这样 fraction × $2^{exponent}$ = value，其中 0.5 ≤ fraction < 1，exponent 是一个整数。exponent 存储于第 2 个参数所指向的内存位置，函数返回 fraction 的值。与它相关的函数 ldexp 的返回值是 fraction × $2^{exponent}$，也就是它原先的值。当你必须在那些浮点格式不兼容的机器之间传递浮点数时，这些函数是非常有用的。

modf 函数把一个浮点值分成整数和小数两个部分，每个部分都具有和原值一样的符号。整数部分以 double 类型存储于第 2 个参数所指向的内存位置，小数部分作为函数的返回值返回。

16.2.5　幂 <math.h>

这个家族共有两个函数。

```
double pow( double x, double y );
double sqrt( double x );
```

pow 函数返回 x^y 的值。由于在计算这个值时可能要用到对数，所以如果 x 是一个负数且 y 不是一个整数，就会出现一个定义域错误。

sqrt 函数返回其参数的平方根。如果参数为负，就会出现一个定义域错误。

16.2.6　底数、顶数、绝对值和余数 <math.h>

这些函数的原型如下所示。

```
double floor( double x );
double ceil( double x );
double fabs( double x );
double fmod( double x, double y );
```

floor 函数返回不大于其参数的最大整数值。这个值以 double 的形式返回，这是因为 double 能够表示的范围远大于 int。ceil 函数返回不小于其参数的最小整数值。

fabs 函数返回其参数的绝对值。fmod 函数返回 x 除以 y 所产生的余数，这个除法的商被限制为一个整数值。

16.2.7 字符串转换 <stdlib.h>

这些函数和整型字符串转换函数类似，只不过它们返回浮点值。

```
double atof( char const *string );
double strtod( char const *string, char **unused );
```

如果任一函数的参数包含了前导的空白字符，这些字符将被忽略。函数随后把合法的字符转换为一个 double 值，忽略任何缀尾的非法字符。这两个函数都接受程序中所有浮点数字面值的书写形式。

strtod 函数把参数字符串转换为一个 double 值，其方法和 atof 类似，但它保存一个指向字符串中被转换的值后面的第 1 个字符的指针。如果函数的第 2 个参数不是 NULL，那么这个被保存的指针就存储于第 2 个参数所指向的内存位置。这个指针允许对字符串的剩余部分进行处理，而不用猜测转换会在字符串中的什么位置结束。

如果这两个函数的字符串参数并不包含任何合法的数值字符，函数就返回零。如果转换值太大或太小，无法用 double 表示，那么函数就在 errno 中存储 ERANGE 这个值，如果值太大（无论是正数还是负数），函数返回 HUGE_VAL。如果值太小，函数返回零。

16.3 日期和时间函数

函数库提供了一组非常丰富的函数，用于简化日期和时间的处理。它们的原型位于 time.h。

16.3.1 处理器时间 <time.h>

clock 函数返回从程序开始执行起处理器所消耗的时间。

```
clock_t  clock( void );
```

注意这个值可能是个近似值。如果需要更精确的值，你可以在 main 函数刚开始执行时调用 clock，然后把以后调用 clock 时所返回的值减去前面这个值。如果机器无法提供处理器时间，或者如果时间值太大，无法用 clock_t 变量表示，函数就返回-1。

clock 函数返回一个数字，它是由编译器定义的。通常它是处理器时钟滴答的次数。为了把这个值转换为秒，你应该把它除以常量 CLOCKS_PER_SEC。

警告：

在有些编译器中，这个函数可能只返回程序所使用的处理器时间的近似值。如果宿主操作系统不能追踪处理器时间，函数可以返回已经流逝的实际时间数量。在有些一次不能运行超过一个程序的简单操作系统中，就可能出现这种情况。本章的练习之一就是探索如何判断你的系统在这方面的表现方式。

16.3.2 当天时间 <time.h>

time 函数返回当前的日期和时间。

```
time_t  time( time_t *returned_value );
```

如果参数是一个非 NULL 的指针，时间值也将通过这个指针进行存储。如果机器无法提供当前

的日期和时间，或者时间值太大，无法用 time_t 变量表示，函数就返回-1。

标准并未规定时间的编码方式，所以你不应该使用字面值常量，因为它们在不同的编译器中可能具有不同的含义。一种常见的表示形式是返回从一个任意选定的时刻开始流逝的秒数。在 MS-DOS 和 UNIX 系统中，这个时刻是 1970 年 1 月 1 日 00:00:00[1]。

警告：

调用 time 函数两次并把两个值相减，由此判断期间所流逝的时间是很有诱惑力的。但这个技巧是很危险的，因为标准并未要求函数的结果值用秒来表示。difftime 函数（下一节描述）可以用于这个目的。

日期和时间的转换 <time.h>

下面的函数用于操纵 time_t 值。

```
char *ctime( time_t const *time_value );
double difftime( time_t time1, time_t time2 );
```

ctime 函数的参数是一个指向 time_t 的指针，并返回一个指向字符串的指针，字符串的格式如下所示：

```
Sun Jul 4 04:02:48 1976\n\0
```

字符串内部的空格是固定的。一个月的每一天总是占据两个位置，即使第 1 个是空格。时间值的每部分都用两个数字表示。标准并未提及存储这个字符串的内存类型，许多编译器使用一个静态数组。因此，下一次调用 ctime 时，这个字符串将被覆盖。因此，如果你需要保存它的值，应该事先为其复制一份。注意 ctime 实际上可能以下面这种方式实现：

```
asctime( localtime( time_value ) );
```

difftime 函数计算 time1-time2 的差，并把结果值转换为秒。注意它返回的是一个 double 类型的值。

接下来的两个函数把一个 time_t 值转换为一个 tm 结构，后者允许我们很方便地访问日期和时间的各个组成部分。

```
struct  tm *gmtime( time_t  const *time_value );
struct  tm *localtime( time_t  const *time_value );
```

gmtime 函数把时间值转换为**世界协调时间**(Coordinated Universal Time, UTC)。UTC 以前被称为**格林尼治标准时间**(Greenwich Mean Time)，这也是 gmtime 这个名字的来历。正如其名字所提示的那样，localtime 函数把一个时间值转换为当地时间。标准包含了这两个函数，但它并没有描述 UTC 和当地时间的实现之间的关系。

tm 结构包含了表 16.2 所列出的字段，不过这些字段在结构中出现的顺序并不一定如此。

警告：

使用这些值最容易出现的错误就是错误地解释月份。这些值表示从 1 月开始的月份，所以 0 表示 1 月，11 表示 12 月。尽管初看上去很不符合直觉，这种编号方式被证明是一种行之有效的月份

---

[1] 在许多编译器中，time_t 被定义为一个有符号的 32 位量。2038 年应该是比较有趣的：从 1970 年开始计数的秒数将在该年溢出 time_t 变量。

编码方式，因为它允许你把这些值作为下标值使用，访问一个包含月份名称的数组。

表 16.2　　tm 结构的字段

类型 & 名称	范　围	含　义
int tm_sec;	0-61	分之后的秒数*
int tm_min;	0-59	小时之后的分数
int tm_hour;	0-23	午夜之后的小时数
int tm_mday;	1-31	当月的日期
int tm_mon;	0-11	1 月之后的月数
int tm_year;	0-??	1900 之后的年数
int tm_wday;	0-6	星期天之后的天数
int tm_yday;	0-365	1 月 1 日之后的天数
int tm_isdat;		夏令时标志

* 我们必须赞美制订 C++标准的 ANSI 标准委员会考虑之周详，它允许偶尔出现的“闰秒”加到每年的最后一分钟，对我们的时间标准进行调整，以适应地球旋转的细微变慢现象。

警告：

接下来一个常见的错误就是忘了 tm_year 这个值只是 1900 年之后的年数。为了计算实际的年份，这个值必须与 1900 相加。

当你拥有了一个 tm 结构之后，你既可以直接使用它的值，也可以把它作为参数传递给下面的函数之一。

```
char *asctime( struct tm const *tm_ptr );
size_t strftime( char *string, size_t maxsize, char const *format,
    struct tm const *tm_ptr );
```

asctime 函数把参数所表示的时间值转换为一个以下面的格式表示的字符串：

```
Sun Jul 4 04:02:48 1976\n\0
```

这个格式和 ctime 函数所使用的格式一样，后者在内部很可能调用了 asctime 来实现自己的功能。

strftime 函数把一个 tm 结构转换为一个根据某个格式字符串而定的字符串。这个函数在格式化日期方面提供了令人难以置信的灵活性。如果转换结果字符串的长度小于 maxsize 参数，那么该字符串就被复制到第 1 个参数所指向的数组中，strftime 函数返回字符串的长度。否则，函数返回-1 且数组的内容是未定义的。

格式字符串包含了普通字符和格式代码。普通字符被复制到它们原先在字符串中出现的位置。格式代码则被一个日期或时间值代替。格式代码包括一个%字符，后面跟一个表示所需值的字符。表 16.3 列出了已经实现的格式代码。如果%字符后面是一个其他任何字符，其结果是未定义的，这就允许各个编译器自由地定义额外的格式代码。你应该避免使用这种自定义的格式代码，除非你不怕牺牲代码的可移植性。特定于 locale 的值由当前的 locale 决定，它将在本章的后面讨论。%U 和%W 代码基本相同，区别在于前者把当年的第 1 个星期日作为第 1 个星期的开始而后者把当年的第 1 个星期一作为第 1 个星期的开始。如果无法判断时区，%Z 代码就由一个空字符串代替。

表 16.3　　strftime 格式代码

代　码	被...代替
%%	一个%字符
%a	一星期的某天，以当地的星期几的简写形式表示
%A	一星期的某天，以当地的星期几的全写形式表示
%b	月份，以当地月份名的简写形式表示
%B	月份，以当地月份名的全写形式表示
%c	日期和时间，使用%x %X
%d	一个月的第几天(01-31)
%H	小时，以 24 小时的格式(00-23)
%I	小时，以 12 小时的格式(00-12)
%J	一年的第几天(001-366)
%M	月数(01-12)
%M	分钟（00～59）
%P	AM 或 PM（不论哪个合适）的当地对等表示形式
%S	秒(00-61)
%U	一年的第几星期(00-53)，以星期日为第 1 天
%w	一星期的第几天，星期日为第 0 天
%W	一年的第几星期(00-53)，以星期一为第 1 天
%x	日期，使用本地的日期格式
%X	时间，使用本地的时间格式
%y	当前世纪的年份(00-99)
%Y	年份的全写形式(例如，1984)
%Z	时区的简写

最后，mktime 函数用于把一个 tm 结构转换为一个 time_t 值。

```
time_t  mktime( struct tm *tm_ptr );
```

tm 结构中 tm_wday 和 tm_yday 的值被忽略，其他字段的值也无需限制在它们的通常范围内。在转换之后，该 tm 结构会进行规格化，因此 tm_wday 和 tm_yday 的值将是正确的，其余字段的值也都位于它们通常的范围之内。这个技巧是一种简单的用于判断某个特定的日期属于星期几的方法。

16.4　非本地跳转 <setjmp.h>

setjmp 和 longjmp 函数提供了一种类似 goto 语句的机制，但它并不局限于一个函数的作用域之内。这些函数常用于深层嵌套的函数调用链。如果在某个低层的函数中检测到一个错误，你可以立即返回到顶层函数，不必向调用链中的每个中间层函数返回一个错误标志。

为了使用这些函数，你必须包含头文件 setjmp.h。这两个函数的原型如下所示：

```
int  setjmp( jmp_buf  state );
void  longjmp( jump_buf  state, int value );
```

你声明一个 jmp_buf 变量，并调用 setjmp 函数对它进行初始化，setjmp 的返回值为零。setjmp 把程序的状态信息（例如，堆栈指针的当前位置和程序的计数器）保存到跳转缓冲区[1]。你调用 setjmp 时所处的函数便成为你的“顶层”函数。

以后，在顶层函数或其他任何它所调用的函数（不论是直接调用还是间接调用）内的任何地方调用 longjmp 函数，将导致这个被保存的状态重新恢复。longjmp 的效果就是使执行流通过再次从 setjmp 函数返回，从而立即跳回到顶层函数中。

你如何区别从 setjmp 函数的两种不同返回方式呢？当 setjmp 函数第 1 次被调用时，它返回 0。当 setjmp 作为 longjmp 的执行结果再次返回时，它的返回值是 longjmp 的第 2 个参数，它必须是个非零值。通过检查它的返回值，程序可以判断是否调用了 longjmp。如果存在多个 longjmp，也可以由此判断哪个 longjmp 被调用。

16.4.1 实例

程序 16.2 使用 setjmp 来处理它所调用的函数检测到的错误，但无需使用寻常的返回和检查错误代码的逻辑。setjmp 的第 1 次调用确立了一个地点，如果调用 longjmp，程序的执行流将在这个地点恢复执行。setjmp 的返回值为 0，这样程序便进入事务处理循环。如果 get_trans、process_trans 或其他任何被这些函数调用的函数检测到一个错误，它将像下面这样调用 longjmp：

```
longjmp( restart, 1 );
```

执行流将立即在 restart 这个地点重新执行，setjmp 的返回值为 1。

这个例子可以处理两种不同类型的错误：一种是阻止程序继续执行的致命错误；另一种是只破坏正在处理的事务的小错误。这个对 longjmp 的调用属于后者。当 setjmp 返回 1 时，程序就打印一条错误信息，并再次进入事务处理循环。为了报告一个致命错误，可以用任何其他值调用 longjmp，程序将保存它的数据并退出。

```
/*
** 一个说明 setjmp 用法的程序
*/
#include "trans.h"
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

/*
** 用于存储 setjmp 的状态信息的变量。
*/
jmp_buf    restart;

int
main()
{
     int  value;
     Trans     *transaction;

     /*
```

[1] 程序当前正在执行的指令的地址。

```
** 确立一个我们希望在 longjmp 的调用之后执行流恢复执行的地点。
*/
value = setjmp( restart );

/*
** 从 longjmp 返回后判断下一步执行什么。
*/
switch( setjmp( restart ) ){
default:
     /*
     **longjmp 被调用 -- 致命错误
     */
     fputs( "Fatal error.\n", stderr );
     break;

case 1:
     /*
     **longjmp 被调用 -- 小错误
     */
     fputs( "Invalid transaction.\n", stderr );
     /* FALL THROUGH 并继续进行处理 */

     case 0:
          /*
          ** 最初从 setjmp 返回的地点: 执行正常的处理。
          */
          while( (transaction = get_trans()) != NULL )
                  process_trans( transaction );

     }

     /*
     ** 保存数据并退出程序
     */
     write_data_to_file();

     return value == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

程序 16.2　setjmp 和 longjmp 实例　　setjmp.c

16.4.2　何时使用非本地跳转

setjmp 和 longjmp 并不是绝对必需的，因为你总是可以通过返回一个错误代码并在调用函数中对其进行检查来实现相同的效果。返回错误代码的方法有时候不是很方便，特别当函数已经返回了一些值的时候。如果存在一长串的函数调用链，即使只有最深层的那个函数发现了错误，调用链中的所有函数都必须返回并检查错误代码。在这种情况下使用 setjmp 和 longjmp 去除了中间函数的错误代码逻辑，从而对它们进行了简化。

警告：

当顶层函数（调用 setjmp 的那个）返回时，保存在跳转缓冲区的状态信息便不再有效。在此之

后调用 longjmp 很可能失败，而它的症状很难调试。这就是为什么 longjmp 只能在顶层函数或者在顶层函数所调用的函数中进行调用的原因。只有这个时候保存在跳转缓冲区的状态信息才是有效的。

提示：

由于 setjmp 和 longjmp 有效地实现了 goto 语句的功能，所以你在使用它们时必须遵循某些诫律。在程序 16.2 例子的情况下，这两个函数有助于编写更清晰、复杂度更低的代码。但是，如果 setjmp 和 longjmp 用于在一个函数内部模拟 goto 语句或者程序中存在许多执行流可能返回的跳转缓冲区时，那么程序的逻辑就会变得更加难以理解，程序将会变得更难调试和维护，另外失败的可能性也变得更大。你可以使用 setjmp 和 longjmp，但你应该合理地使用它们。

16.5 信号

程序中所发生的事件绝大多数都是由程序本身所引发的，例如执行各种语句和请求输入。但是，有些程序必须遇到的事件却不是程序本身所引发的。一个常见的例子就是用户中断了程序。如果部分计算好的结果必须进行保存以避免数据的丢失，程序必须预备对这类事件作出反应，虽然它并没有办法预测什么时候会发生这种情况。

信号就是用于这种目的。**信号**(signal)表示一种事件，它可能异步地发生，也就是并不与程序执行过程的任何事件同步。如果程序并未安排怎样处理一个特定的信号，那么当该信号出现时程序就作出一个缺省的反应。标准并未定义这个缺省反应是什么，但绝大多数编译器都选择终止程序。另外，程序可以调用 signal 函数，或者忽略这个信号，或者设置一个**信号处理函数**(signal handler)，当信号发生时程序就调用这个函数。

16.5.1 信号名 <signal.h>

表 16.4 列出了标准所定义的信号，但编译器并不需要实现所有这些信号，而且如果它觉得合适，也可以定义其他的信号。

SIGABRT 是一个由 abort 函数所引发的信号，用于终止程序。至于哪些错误将引发 SIGFPE 信号则取决于编译器。常见的有算术上溢或下溢以及除零错误。有些编译器对这个信号进行了扩展，提供了关于引发这个信号的操作的特定信息。使用这个信息可以允许程序对这个信号作出更智能的反应，但这样做将影响程序的可移植性。

表 16.4　　信　　号

信　　号	含　　义
SIGABRT	程序请求异常终止
SIGFPE	发生一个算术错误
SIGILL	检测到非法指令
SIGSEGV	检测到对内存的非法访问
SIGINT	收到一个交互性注意信号
SIGTERM	收到一个终止程序的请求

SIGILL 信号提示 CPU 试图执行一条非法的指令。这个错误可能由于不正确的编译器设置所导致。例如，用 Intel 80386 指令编译一个程序，但把这个程序运行于一台 80286 计算机上。另一个可

能的原因是程序的执行流出现了错误，例如使用一个未初始化的函数指针调用一个函数，导致 CPU 试图执行实际上是数据的东西（把数据段当成了代码段）。SIGSEGV 信号提示程序试图非法访问内存。这个信号有两个最常见的原因，其中一个是程序试图访问未安装于机器上的内存或者访问操作系统未曾分配给这个程序的内存，另一个是程序违反了内存访问的边界要求。后者可能在那些要求数据边界对齐的机器上发生。例如，如果整数要求位于偶数的边界（存储的起始位置是编号为偶数的地址），一条指定在奇数边界访问一个整数的指令将违反边界规则。未初始化的指针常常会引起这类错误。

前面几个信号是同步的，因为它们都是在程序内部发生的。尽管你无法预测一个算术错误何时将会发生，如果你使用相同的数据反复运行这个程序，每次在相同的地方将出现相同的错误。最后两个信号，SIGINT 和 SIGTERM 则是异步的。它们在程序的外部产生，通常是由程序的用户所触发，表示用户试图向程序传达一些信息。

SIGINT 信号在绝大多数机器中都是当用户试图中断程序时发生的。SIGTERM 则是另一种用于请求终止程序的信号。在实现了这两个信号的系统里，一种常用的策略是为 SIGINT 定义一个信号处理函数，目的是执行一些日常维护工作(housekeeping)并在程序退出前保存数据。但是，SIGTERM 则不配备信号处理函数，这样当程序终止时便不必执行这些日常维护工作。

16.5.2　处理信号 <signal.h>

通常，我们关心的是怎样处理那些自主发生的信号，也就是无法预测其什么时候会发生的信号。raise 函数用于显式地引发一个信号。

```
int  raise( int sig );
```

调用这个函数将引发它的参数所指定的信号。程序对这类信号的反应和那些自主发生的信号是相同的。你可以调用这个函数对信号处理函数进行测试。但如果误用，它可能会实现一种非局部的 goto 效果，因此要避免以这样方式使用它。

当一个信号发生时，程序可以使用三种方式对它作出反应。缺省的反应是由编译器定义的，通常是终止程序。程序也可以指定其他行为对信号作出反应：信号可以被忽略，或者程序可以设置一个信号处理函数，当信号发生时调用这个函数。signal 函数用于指定程序希望采取的反应。

```
void ( *signal( int sig, void ( *handler )( int ) ) )( int );
```

这个函数的原型看上去有些吓人，所以让我们对它进行分析。首先，我将省略返回类型，这样我们可以先对参数进行研究：

```
signal( int sig, void ( *handler )( int ) )
```

第 1 个参数是表 16.4 所列的信号之一，第 2 个参数是你希望为这个信号设置的信号处理函数。这个处理函数是一个函数指针，它所指向的函数接受一个整型参数且没有返回值。当信号发生时，信号的代码作为参数传递给信号处理函数。这个参数允许一个处理函数处理几种不同的信号。

现在我将从原型中去掉参数，这样函数的返回类型看上去就比较清楚。

```
void ( *signal() )( int );
```

siganl 是一个函数，它返回一个函数指针，后者所指向的函数接受一个整型参数且没有返回值。事实上，signal 函数返回一个指向该信号以前的处理函数的指针。通过保存这个值，你可以为信号设置一个处理函数并在将来恢复为先前的处理函数。如果调用 signal 失败，例如由于非法的信号代

码所致，函数将返回 SIG_ERR 值。这个值是个宏，它在 signal.h 头文件中定义。

signal.h 头文件还定义了另外两个宏，SIG_DFL 和 SIG_IGN，它们可以作为 signal 函数的第 2 个参数。SIG_DFL 恢复对该信号的缺省反应，SIG_IGN 使该信号被忽略。

16.5.3 信号处理函数

当一个已经设置了信号处理函数的信号发生时，系统首先恢复对该信号的缺省行为[1]。这样做是为了防止如果信号处理函数内部也发生这个信号可能导致的无限循环。然后，信号处理函数被调用，信号代码作为参数传递给函数。

信号处理函数可能执行的工作类型是很有限的。如果信号是异步的，也就是说不是由于调用 abort 或 raise 函数引起的，信号处理函数便不应调用除 signal 之外的任何库函数，因为在这种情况下其结果是未定义的。而且，信号处理函数除了能向一个类型为 volatile sig_atomic_t 的静态变量（volatile 在下一节描述）赋一个值以外，可能无法访问其他任何静态数据。为了保证真正的安全，信号处理函数所能做的就是对这些变量之一进行设置然后返回。程序的剩余部分必须定期检查变量的值，看看是否有信号发生。

这些严格的限制是由于信号处理的本质产生的。信号通常用于提示发生了错误。在这些情况下，CPU 的行为是精确定义的，但在程序中，错误所处的上下文环境可能很不相同，因此它们并不一定能够良好定义。例如，当 strcpy 函数正在执行时如果产生一个信号，可能当时目标字符串暂时未以 NUL 字节终结；或者当一个函数被调用时如果产生一个信号，当时堆栈可能处于不完整的状态。如果依赖这种上下文环境的库函数被调用，它们就可能以不可预料的方式失败，很可能引发另一个信号。

访问限制定义了在信号处理函数中保证能够运行的最小功能。类型 sig_atomic_t 定义了一种 CPU 可以以原子方式访问的数据类型，也就是不可分割的访问单位。例如，一台 16 位的机器可以以原子方式访问一个 16 位整数，但访问一个 32 位整数可能需要两个操作。在访问非原子数据的中间步骤时如果产生一个信号可能导致不一致的结果，在信号处理函数中把数据访问限制为原子单位可以消除这种可能性。

警告：

标准表示信号处理函数可以通过调用 exit 终止程序。用于处理除了 SIGABRT 之外所有信号的处理函数也可以通过调用 abort 终止程序。但是，由于这两个都是库函数，所以当它们被异步信号处理函数调用时可能无法正常运行。如果你必须用这种方式终止程序，注意仍然存在一种微小的可能性导致它失败。如果发生这种情况，函数的失败可能破坏数据或者表现出奇怪的症状，但程序最终将终止。

一、volatile 数据

信号可能在任何时候发生，所以由信号处理函数修改的变量的值可能会在任何时候发生改变。因此，你不能指望这些变量在两条相邻的程序语句中肯定具有相同的值。volatile 关键字告诉编译器这个事实，防止它以一种可能修改程序含义的方式“优化”程序。考虑下面的程序段：

---

[1] 编译器可以选择当信号处理函数正在执行时“阻塞”信号而不是恢复缺省行为。请参阅有关文档。

```
if( value ){
        printf( "True\n" );
}
else {
        printf( "False\n" );
}
if( value ){
        printf( "True\n" );
}
else {
        printf( "False\n" );
}
```

在普通情况下，你会认为第 2 个测试和第 1 个测试具有相同的结果。如果信号处理函数修改了这个变量，第 2 个测试的结果可能不同。除非变量被声明为 volatile，否则编译器可能会用下面的代码进行替换，从而对程序进行“优化”。这些语句在通常情况下是正确的：

```
if( value ){
        printf( "True\n" );
}
else {
        printf( "False\n" );
        printf( "False\n" );
}
```

二、从信号处理函数返回

从一个信号处理函数返回导致程序的执行流从信号发生的地点恢复执行。这个规则的例外情况是 SIGFPE。由于计算无法完成，从这个信号返回的效果是未定义的。

警告：

如果你希望捕捉将来同种类型的信号，从当前这个信号的处理函数返回之前注意要调用 signal 函数重新设置信号处理函数。否则，只有第 1 个信号才会被捕捉。接下来的信号将使用缺省反应进行处理。

提示：

由于各种计算机对不可预料的错误的反应各不相同，因此信号机制的规范也比较宽松。例如，编译器并不一定要使用标准定义的所有信号，而且在调用某个信号的处理函数之前可能会也可能不会重新设置信号的缺省行为。另一方面，对信号处理函数所施加的严重限制反映了不同的硬件和软件环境所施加的限制的交集。

这些限制和平台依赖性的结果就是使用信号处理函数的程序比不使用信号处理函数的程序可移植性弱一些。只有当需要时才使用信号以及不违反信号处理函数的规则有助于使这种类型的程序内部固有的可移植性问题降低到最低限度。

16.6　打印可变参数列表 <stdarg.h>

这组函数用于可变参数列表必须被打印的场合。注意：它们要求包含头文件 stdio.h 和 stdarg.h。

```
int vprintf( char const *format, va_list arg );
int vfprintf( FILE *stream, char const *format, va_list arg );
int vsprintf( char *buffer, char const *format, va_list arg );
```

这些函数与它们对应的标准函数基本相同，但它们使用了一个可变参数列表（请参阅第 7 章关于可变参数列表的详细内容）。在调用这些函数之前，arg 参数必须使用 va_start 进行初始化。这些函数都不需要调用 va_end。

16.7 执行环境

这些函数与程序的执行环境进行通信或者对后者施加影响。

16.7.1 终止执行 <stdlib.h>

这三个函数与正常或不正常的程序终止有关。

```
void abort( void )
void atexit( void (func)( void ) );
void exit( int status );
```

abort 函数用于不正常地终止一个正在执行的程序。由于这个函数将引发 SIGABRT 信号，你可以在程序中为这个信号设置一个信号处理函数，在程序终止（或干脆不终止）之前采取任何你想采取的动作，甚至可以不终止程序。

atexit 函数可以把一些函数注册为**退出函数**(exit function)。当程序将要正常终止时（或者由于调用 exit，或者由于 main 函数返回），退出函数将被调用。退出函数不能接受任何参数。

exit 函数在第 15 章已经作了描述，它用于正常终止程序。如果程序以 main 函数返回一个值结束，那么其效果相当于用这个值作用参数调用 exit 函数。

当 exit 函数被调用时，所有被 atexit 函数注册为退出函数的函数将按照它们所注册的顺序被反序依次调用。然后，所有用于流的缓冲区被刷新，所有打开的文件被关闭。用 tmpfile 函数创建的文件被删除。然后，退出状态返回给宿主环境，程序停止执行。

警告：

由于程序停止执行，所以 exit 函数绝不会返回到它的调用处。但是，如果任何一个用 atexit 注册为退出函数的函数再次调用了 exit，其效果是未定义的。这个错误可能导致一个无限循环，很可能只有当堆栈的内存耗尽后才会终止。

16.7.2 断言<assert.h>

断言就是声明某种东西应该为真。ANSI C 实现了一个 assert 宏，它在调试程序时很有用。它的原型如下所示[1]。

```
void assert( int expression );
```

当它被执行时，这个宏对表达式参数进行测试。如果它的值为假（零），它就向标准错误打印一条诊断信息并终止程序。这条信息的格式是由编译器定义的，但它将包含这个表示式和源文件的名

[1] 由于它是一个宏而不是函数，assert 实际上并不具有原型。但是，这个原型说明了 assert 的用法。

字以及断言所在的行号。如果表达式为真（非零），它不打印任何东西，程序继续执行。

这个宏提供了一种方便的方法，对应该是真的东西进行检查。例如，如果一个函数必须用一个不能为 NULL 的指针参数进行调用，那么函数可以用断言验证这个值：

```
assert( value != NULL );
```

如果函数错误地接受了一个 NULL 参数，程序就会打印一条类似下面形式的信息：

```
Assertion failed: value != NULL, file.c line 274
```

提示：

用这种方法使用断言使调试变得更容易，因为一旦出现错误，程序就会停止。而且，这条信息准确地提示了症状出现的地点。如果没有断言，程序可能继续运行，并在以后失败，这就很难进行调试。

注意 assert 只适用于验证必须为真的表达式。由于它会终止程序，所以你无法用它检查那些你试图进行处理的情况，例如检测非法的输入并要求用户重新输入一个值。

当程序被完整地测试完毕之后，你可以在编译时通过定义 NDEBUG 消除所有的断言[1]。你可以使用-DNDEBUG 编译器命令行选项或者在源文件中头文件 assert.h 被包含之前增加下面这个定义

```
#define NDEBUG
```

当 NDEBUG 被定义之后，预处理器将丢弃所有的断言，这样就消除了这方面的开销，而不必从源文件中把所有的断言实际删除。

16.7.3　环境 <stdlib.h>

环境(environment)就是一个由编译器定义的名字/值对的列表，它由操作系统进行维护。getenv 函数在这个列表中查找一个特定的名字，如果找到，返回一个指向其对应值的指针。程序不能修改返回的字符串。如果名字未找到，函数就返回一个 NULL 指针。

```
char *getenv( char const *name );
```

注意标准并未定义一个对应的 putenv 函数。有些编译器以某种方式提供了这个函数，不过如果你需要考虑程序的可移植性，最好还是避免使用它。

16.7.4　执行系统命令 <stdlib.h>

system 函数把它的字符串参数传递给宿主操作系统，这样它就可以作为一条命令，由系统的命令处理器执行。

```
void system( char const *command );
```

这个任务执行的准确行为因编译器而异，system 的返回值也是如此。但是，system 可以用一个 NULL 参数调用，用于询问命令处理器是否实际存在。在这种情况下，如果存在一个可用的命令处理器，system 返回一个非零值，否则它返回零。

---

[1] 可以把它定义为任何值，编译器只关心是否定义了 NDEBUG。

16.7.5 排序和查找<stdlib.h>

qsort 函数在一个数组中以升序的方式对数据进行排序。由于它是和类型无关的，所以你可以使用 qsort 排序任意类型的数据，只是数组中元素的长度是固定的。

```
void qsort( void *base, size_t n_elements, size_t el_size,
     int (*compare)(void const *, void const * ) );
```

第 1 个参数指向需要排序的数组，第 2 个参数指定数组中元素的数目，第 3 个参数指定每个元素的长度（以字符为单位）。第 4 个参数是一个函数指针，用于对需要排序的元素类型进行比较。在排序时，qsort 调用这个函数对数组中的数据进行比较。通过传递一个指向合适的比较函数的指针，你可以使用 qsort 排序任意类型值的数组。

比较函数接受两个参数，它们是指向两个需要进行比较的值的指针。函数应该返回一个整数，大于零、等于零和小于零分别表示第 1 个参数大于、等于和小于第 2 个参数。

由于这个函数与类型无关的性质，参数被声明为 void *类型。在比较函数中必须使用强制类型转换把它们转换为合适的指针类型。程序 16.3 说明了一个元素类型为一个关键字值和其他一些数据的结构的数组是如何被排序的。

```
/*
** 使用 qsort 对一个元素为某种结构的数组进行排序
*/
#include <stdlib.h>
#include <string.h>

typedef    struct    {
           char key[ 10 ];     /* 数组的排序关键字 */
           int  other_data;    /* 与关键字关联的数据 */
} Record;

/*
**  比较函数: 只比较关键字的值。
*/
int r_compare( void const *a, void const *b ){
     return strcmp( ((Record *)a)->key, ((Record *)b)->key );
}

int
main()
{
     Record    array[ 50 ];

     /*
     ** 用 50 个元素填充数组的代码
     */
     qsort( array, 50, sizeof( Record ), r_compare );

     /*
     ** 现在，数组已经根据结构的关键字字段排序完毕
     */
```

```
    return EXIT_SUCCESS;
}
```

程序 16.3　用 qsort 排序一个数组　　qsort.c

bsearch 函数在一个已经排好序的数组中用二分法查找一个特定的元素。如果数组尚未排序，其结果是未定义的。

```
void *bsearch(void const *key, void const *base,     size_tn_elements,
    size_t el_size, int (*compare)(void const *, void const * ) );
```

第 1 个参数指向你需要查找的值，第 2 个参数指向查找所在的数组，第 3 个参数指定数组中元素的数目，第 4 个参数是每个元素的长度（以字符为单位）。最后一个参数是和 qsort 中相同的指向比较函数的指针。bsearch 函数返回一个指向查找到的数组元素的指针。如果需要查找的值不存在，函数返回一个 NULL 指针。

注意关键字参数的类型必须与数组元素的类型相同。如果数组中的结构包含了一个关键字字段和其他一些数据，你必须创建一个完整的结构并填充关键字字段。其他字段可以留空，因为比较函数只检查关键字字段。bsearch 函数的用法如程序 16.4 所示。

```
/*
** 用 bearch 在一个元素类型为结构的数组中查找
*/
#include <stdlib.h>
#include <string.h>

typedef    struct    {
           char key[ 10 ];      /* 数组的排序关键字 */
           int  other_data;     /* 与关键字关联的数据 */
} Record;

/*
**  比较函数：只比较关键字的值。
*/
int r_compare( void const *a, void const *b ){
      return strcmp( ((Record *)a)->key, ((Record *)b)->key );
}

int
main()
{
      Record   array[ 50 ];
      Record   key;
      Record   *ans;

      /*
      ** 用 50 个元素填充数组并进行排序的代码
      */

      /*
      ** 创建一个关键字结构（只用需要查找的值填充关键字字段），
      ** 并在数组中查找。
      */
      strcpy( key.key, "value" );
      ans = bsearch( &key, array, 50, sizeof( Record ),
```

```
            r_compare );

        /*
        **ans 现在指向关键字字段与值匹配的数据元素，如果无匹配，ans 为 NULL
        */

        return EXIT_SUCCESS;
    }
```

程序 16.4　用 bsearch 在数组中查找　　bsearch.c

16.8　locale

为了使 C 语言在全世界的范围内更为通用，标准定义了 locale，这是一组特定的参数，每个国家可能各不相同。在缺省情况下是“C” locale，编译器也可以定义其他的 locale。修改 locale 可能影响库函数的运行方式。修改 locale 的效果在本节的最后进行描述。

setlocale 函数的原型如下所示，它用于修改整个或部分 locale。

```
char *setlocale( int category, char const *locale );
```

category 参数指定 locale 的哪个部分需要进行修改。它所允许出现的值列于表 16.5。

如果 setlocale 的第 2 个参数为 NULL，函数将返回一个指向给定类型的当前 locale 的名字的指针。这个值可能被保存并在后续的 setlocale 函数中使用，用来恢复以前的 locale。如果第 2 个参数不是 NULL，它指定需要使用的新 locale。如果函数调用成功，它将返回新 locale 的值，否则返回一个 NULL 指针，原来的 locale 不受影响。

表 16.5　　setlocale 类型

值	修　改
LC_ALL	整个 locale
LC_COLLATE	对照序列，它将影响 strcoll 和 strxfrm 函数的行为
LC_CTYPE	定义于 ctype.h 中的函数所使用的字符类型分类信息
LC_MONETARY	在格式化货币值时使用的字符
LC_NUMERIC	在格式化非货币值时使用的字符。同时修改由格式化输入/输出函数和字符串转换函数所使用的小数点符号
LC_TIME	strftime 函数的行为

16.8.1　数值和货币格式 <locale.h>

格式数值和货币值的规则在全世界的不同地方可能并不相同。例如，在美国，一个写作 1,234.56 的数字在许多欧洲国家将被写成 1.234,56。localeconv 函数用于获得根据当前的 locale 对非货币值和货币值进行合适的格式化所需要的信息。注意这个函数并不实际执行格式化任务，它只是提供一些如何进行格式化的信息。

```
struct lconv *localeconv( void );
```

lconv 结构包含两种类型的参数：字符和字符指针。字符参数为非负值。如果一个字符参数为 CHAR_MAX，那个这个值就在当前的 locale 中不可用（或不使用）。对于字符指针参数，如果它指

向一个空字符串，它表示的意义和上面相同。

一、数值格式化

表 16.6 列出的参数用于格式化非货币的数值量。grouping 字符串按照下面的方式进行解释。该字符串的第 1 个值指定小数点左边多少个数字组成一组。第 2 个值指定再往左边一组数字的个数，以下依此类推。有两个值具有特别的意义：CHAR_MAX 表示剩余的数字并不分组，0 表示前面的值适用于数值中剩余的各组数字。

表 16.6　　格式化非货币数值的参数

字段和类型	含　义
char *decimal_point	用作小数点的字符。这个值绝不能是个空字符串
char *thousands_sep	用作分隔小数点左边各组数字的符号
char *grouping	指定小数点左边多少个数字组成一组

典型的北美格式是用下面的参数指定的：

```
decimal_point="."
thousands_sep=","
grouping="\3"
```

grouping 字符串包含一个 3[1]，后面是一个 0（也就是用于结尾的 NUL 字节）。这些值表示小数点左边的第 1 组数字将包括三个数字，其余的各组也将包括三个数字。值 1234567.89 根据这些参数进行格式化以后将以 1 234 567.89 的形式出现。

下面是另外一个例子。

```
grouping = "\4\3"
thousands_sep = "-"
```

这些值表示格式化北美地区电话号码的规则。根据这些参数，值 2125551234 将被格式化为 212-555-1234 的形式。

二、货币格式化

格式化货币值的规则要复杂得多。这是由于存在许多不同的提示正值和负值的方法、货币符号相对于值的位置等。另外，当货币值的格式化用于国际化时，规则又有所修改。首先，我们研究一些用于格式化本地（非国际）货币量的参数，见表 16.7。

表 16.7　　格式化本地货币值的参数

字段和类型	含　义
char *currency_symbol	本地货币符号
char *mon_decimal_point	小数点字符
char *mon_thousands_sep	用于分隔小数点左边各组数字的字符
char *mon_grouping	指定出现在小数点左边每组数字的数字个数
char *positive_sign	用于提示非负值的字符串
char *negative_sign	用于提示负值的字符串

[1] 注意这个数字是二进制的 3，而不是字符 3。

续表

字段和类型	含　义
char frac_digits	出现在小数点右边的数字个数
char p_cs_precedes	如果 currency_symbol 出现在一个非负值之前，其值为 1；如果出现在后面，其值为 0
char n_cs_precedes	如果 currency_symbol 出现在一个负值之前，其值为 1；如果出现在后面，其值为 0
char p_sep_by_space	如果 currency_symbol 和非负值之间用一个空格分隔，其值为 1，否则为 0
char n_sep_by_space	如果 currency_symbol 和负值之间用一个空格分隔，其值为 1，否则为 0
char p_sign_posn	提示 positive_sign 出现在一个非负值的位置。允许下列值： 0 货币符号和值两边的括号 1 正号出现在货币符号和值之前 2 正号出现在货币符号和值之后 3 正号紧邻货币符号之前 4 正号紧随货币符号之后
char n_sign_posn	提示 negative_sign 出现在一个负值中的位置。用于 p_sign_posn 的值也可用于此处

当按照国际化的用途格式化货币值时，字符串 int-curr_symbol 替代了 currency_symbol，字符 int_frac_digits 替代了 frac_digits。国际货币符号是根据 ISO 4217:1987 标准形成的。这个字符串的头三个字符是字母形式的国际货币符号，第 4 个字符用于分隔符号和值。

下面的值用一种可以被美国接受的方式对货币进行格式化。

```
currency_symbol="$"      p_cs_precedes='\1'
mon_decimal_point="."    n_cs_precedes='\1'
mon_thousands_sep=","    p_sep_by_space='\0'
mon_grouping="\3"        n_sep_by_space='\0'
positive_sign=""         p_sign_posn='\1'
negative_sign="CR"       n_sign_posn='\2'
frac_digits='\2'
```

使用上面这些参数，值 1234567890 和 -1234567890 将分别以 $1 234 567 890.00 和 $1 234 567 890.00CR 的形式出现。

设置 n_sign_posn='\0'可以使上面的负值以($1 234 567 890.00)的形式出现。

16.8.2　字符串和 locale <string.h>

一台机器的字符集的对照序列是固定的，但 locale 提供了一种方法指定不同的序列。当你必须使用一个并非缺省的对照序列时，可以使用下列两个函数。

```
int  strcoll( char const *s1, char const *s2 );
size_t  strxfrm( char *s1, char const *s2, size_t size );
```

strcoll 函数对两个根据当前 locale 的 LC_COLLATE 类型参数指定的字符串进行比较。它返回一个大于、等于或小于零的值，分别表示第 1 个参数大于、等于或小于第 2 个参数。

注意这个比较可能比 strcmp 需要多得多的计算量，因为它需要遵循一个并非是本地机器的对照序列。当字符串必须以这种方式反复进行比较时，我们可以使用 strxfrm 函数减少计算量。它把根据当前的 locale 解释的第 2 个参数转换为另一个不依赖于 locale 的字符串。尽管转换后的字符串的内容是未确定的，但使用 strcmp 函数对这种字符串进行比较和使用 strcoll 函数对原先的字符串进行比较的结果是相同的。

16.8.3　改变 locale 的效果

除了前面描述的那些效果之外，改变 locale 还会产生一些另外的效果。

1．locale 可能向正在执行的程序所使用的字符集增加字符（但可能不会改变现存字符的含义）。例如，许多欧洲语言使用了能够提示重音、货币符号和其他特殊符号的扩展字符集。

2．打印的方向可能会改变。尤其是，locale 决定一个字符应该根据前面一个被打印的字符的哪个方向进行打印。

3．printf 和 scanf 函数家族使用当前 locale 定义的小数点符号。

4．如果 locale 扩展了正在使用的字符集，isalpha、islower、isspace 和 isupper 函数可能比以前包括更多的字符。

5．正在使用的字符集的对照序列可能会改变。这个序列由 strcoll 函数使用，用于字符串之间的相互比较。

6．strftime 函数所产生的日期和时间格式的许多方面都是特定于 locale 的，前面已有所描述。

16.9　总结

标准函数库包含了许多有用的函数。第 1 组函数返回整型结果。abs 和 labs 函数返回它们的参数的绝对值。div 和 ldiv 函数用于执行整数除法。和/操作符不同，当其中一个参数为负时，商的值是精确定义的。rand 函数返回一个伪随机数。调用 srand 允许你从一串伪随机值中的任意一个位置开始产生随机数。atoi 和 atol 函数把一个字符串转换为整型值。strtol 和 strtoul 执行相同的转换，但它们可以给你更多的控制。

下一组函数中的绝大部分接受一个 double 参数并返回 double 结果。标准库提供了常用的三角函数 sin、cos、tan、asin、acos、atan 和 atan2。头三个函数接受一个以弧度表示的角度参数，分别返回该角度对应的正弦、余弦、正切值。接下来的三个函数分别返回与它们的参数对应的反正弦、反余弦和反正切值。最后一个函数根据 x 和 y 参数计算反正切值。双曲正弦、双曲余弦和双曲正切分别由 sinh、cosh 和 tanh 函数进行计算。exp 函数返回以 e 值为底，其参数为幂的指数值。log 函数返回其参数的自然对数，log10 函数返回以 10 为底的对数。

frexp 和 ldexp 函数在创建与机器无关的浮点数表示形式方面是很有用的。frexp 函数用于计算一个给定值的表示形式。ldexp 函数用于解释一个表示形式，恢复它的原先值。modf 函数用于把一个浮点值分割成整数和小数部分。pow 函数计算以第 1 个参数为底，第 2 个参数为幂的指数值。sqrt 函数返回其参数的平方根。floor 函数返回不大于其参数的最大整数，ceil 函数返回不小于其参数的最小整数。fabs 函数返回其参数的绝对值。fmod 函数接受两个参数，返回第 2 个参数除以第 1 个参数的余数。最后，atof 和 strtod 函数把字符串转换为浮点值。后者能够在转换时提供更多的控制。

接下来的一组函数用于处理日期和时间。clock 函数返回从程序执行开始到调用这个函数之间所花费的处理器时间。time 函数用一个 time_t 值返回当前的日期和时间。ctime 函数把一个 time_t 值转换为人眼可读的日期和时间表示形式。difftime 函数计算两个 time_t 值之间以秒为单位的时间差。gmtime 和 localtime 函数把一个 time_t 值转换为一个 tm 结构，tm 结构包含了日期和时间的所有组成部分。gmtime 函数使用世界协调时间，localtime 函数使用本地时间。asctime 和 strftime 函数把一个

tm 结构值转换为人眼可读的日期和时间的表示形式。strftime 函数对转换结果的格式提供了强大的控制。最后，mktime 把存储于 tm 结构中的值进行规格化，并把它们转换为一个 time_t 值。

非本地跳转由 setjmp 和 longjmp 函数提供。调用 setjmp 在一个 jmp_buf 变量中保存处理器的状态信息。接着，后续的 longjmp 调用将恢复这个被保存的处理器状态。在调用 setjmp 的函数返回之后，可能无法再调用 longjmp 函数。

信号表示在一个程序的执行期间可能发生的不可预料的事件，诸如用户中断程序或者发生一个算术错误。当一个信号发生时系统所采取的缺省反应是由编译器定义的，但一般都是终止程序。你可以通过定义一个信号处理函数并使用 signal 函数对其进行设置，从而改变信号的缺省行为。你可以在信号处理函数中执行的工作类型是受到严格限制的，因为程序在信号出现之后可能处于不一致的状态。volatile 数据的值可能会改变，而且很可能是由于自身所致。例如，一个在信号处理函数中修改的变量应该声明为 volatile。raise 函数产生一个由它的参数指定的信号。

vprintf、vfprintf 和 vsprintf 函数和 printf 函数家族执行相同的任务，但需要打印的值以可变参数列表的形式传递给函数。abort 函数通过产生 SIGABRT 信号终止程序。atexit 函数用于注册退出函数，它们在程序退出前被调用。assert 宏用于断言，当一个应该为真的表达式实际为假时，它就会终止程序。当调试完成之后，你可以通过定义 NDEBUG 符号去除程序中的所有断言，而不必把它们物理性地从源代码中删除。getenv 从操作系统环境中提取值。system 接受一个字符串参数，把它作为命令用本地命令处理器执行。

qsort 函数把一个数组中的值按照升序进行排序，bsearch 函数用于在一个已经排好序的数组中用二分法查找一个特定的值。由于这两个函数都是与类型无关的，所以它们可以用于任何数据类型的数组。

locale 就是一组参数，根据世界各国的约定差异对 C 程序的行为进行调整。setlocale 函数用于修改整个或部分 locale。locale 包括了一些用于定义数值如何进行格式化的参数。它们描述的值包括非货币值、本地货币值和国际货币值。locale 本身并不执行任何形式的格式化，它只是简单地提供格式化的规范。locale 可以指定一个和机器的缺省序列不同的对照序列。在这种情况下，strxcoll 用于根据当前的对照序列对字符串进行比较。它所返回的值类型类似 strcmp 函数的返回值。strxfrm 函数把一个当前对照序列的字符串转换为一个位于缺省对照序列的字符串。用这种方式转换的字符串可以用 strcmp 函数进行比较，比较的结果和用 strxcoll 比较原先的字符串的结果相同。

16.10 警告的总结

1. 忘了包含 math.h 头文件可能导致数学函数产生不正确的结果。
2. clock 函数可能只产生处理器时间的近似值。
3. time 函数的返回值并不一定是以秒为单位的。
4. tm 结构中月份的范围并不是从 1 到 12。
5. tm 结构中的年是从 1900 年开始计数的年数。
6. longjmp 不能返回到一个已经不再处于活动状态的函数。
7. 从异步信号的处理函数中调用 exit 或 abort 函数是不安全的。
8. 当每次信号发生时，你必须重新设置信号处理函数。
9. 避免 exit 函数的多重调用。

16.11　编程提示的总结

1. 滥用 setjmp 和 longjmp 可能导致晦涩难懂的代码。
2. 对信号进行处理将导致程序的可移植性变差。
3. 使用断言可以简化程序的调试。

16.12　问题

1. 下面的函数调用返回什么？

```
strtol("12345", NULL, -5 );
```

2. 如果说 rand 函数产生的“随机”数并不是真正的随机数，那么事实上它们能不能满足我们的需要呢？

3. 在你的系统上，下面的程序是什么结果？

```
#include <stdlib.h>
int
main()
{
     int  i;
     for( i = 0; i < 100; i += 1 )
          printf( "%d\n", rand() % 2 );
}
```

4. 你怎样编写一个程序，判断在你的系统中 clock 函数衡量 CPU 时间用的是 CPU 使用时间还是总流逝时间？

5. 下面的代码段试图用军事格式(military format)打印当前时间。它有什么错误？

```
#include <time.h>
struct tm *tm;
time_t now;
...
now = time();
tm = localtime( now );
printf( "%d:%02d:%02d %d/%02d/%02d\n",
    tm->tm_hour, tm->tm_min, tm->tm_sec,
    tm->tm_mon, tm->tm_mday, tm->tm_year );
```

6. 下面的程序有什么错误？当它在你的系统上执行时会发生什么？

```
#include <stdlib.h>
#include <setjmp.h>

jmp_buf jbuf;

void
set_buffer()
{
        setjmp( jbuf );
}

int
main( int ac, char **av )
{
```

```
    int     a = atoi( av[ 1 ] );
    int     b = atoi( av[ 2 ] );

    set_buffer();
    printf( "%d plus %d equals %d\n",
        a, b, a + b );
    longjmp( jbuf, 1 );
    printf( "After longjmp\n" );
    return EXIT_SUCCESS;
}
```

7. 编写一个程序，判断一个整数除以零或者一个浮点数除以零会不会产生 SIGFPE 信号。你如何解释这个结果？
8. qsort 函数所使用的比较函数在第 1 个参数小于第 2 个参数的情况下应该返回一个负值，在第 1 个参数大于第 2 个参数的情况下应该返回一个正值。如果比较函数返回相反的值，对 qsort 的行为有没有什么影响？

16.13 编程练习

★ 1. 计算机人群中颇为流行的一个笑话是“我 29 岁，但我不告诉你这个数字的基数！”如果基数是 16，这个人实际上是 41 岁。编写一个程序，接受一个年龄作为命令行参数，并在 2～36 的范围中计算那个字面值小于等于 29 的最小基数。例如，如果用户输入 41，程序应该计算出这个最小基数为 16。因为在 16 进制中，十进制 41 的值是 29。

★★ 2. 编写一个函数，通过返回一个范围为 1 至 6 的随机整数来模拟掷骰子。注意这 6 个值出现的概率应该相同。当这个函数第 1 次调用时，它应该用当天的当前时间作为种子来产生随机数。

★★ 3. 编写一个程序，以一种三岁小孩的方式来说明当前的时间（例如，时针在 6 上面，分针在 12 上面）。

★★ 4. 编写一个程序，接受三个整数为命令行参数，把它们分别解释为月（1～12）、日（1～31）和年（0～？）。然后，它应该打印出这个日子是星期几（或将是星期几）。对于哪个范围的年份，这个程序的结果才是正确的？

★★ 5. 冬天的天气预报常常会给出“风寒(wind chill)”这个词，它的意思是一个特定的温度或风速所感觉到的寒冷度。例如，如果气温为摄氏-5 度（华氏 23 度），并且风速每秒 10 米（22.37mph，即每小时 22.37 英里），那么风寒度便是摄氏-22.3 度（华氏-8.2 度）。

编写一个函数，使用下面的原型，计算风寒度。

```
double  wind_chill( double temp, double velocity );
```

temp 是摄氏气温的度数，velocity 是风速（米/秒）。函数返回摄氏形式的风寒度。风寒度是用下面的公式计算的：

$$Windchill = \frac{(A + B\sqrt{V} + CV)\Delta t}{A + B\sqrt{X} + CX}$$

对于一个给定的气温和风速。这个公式给出在风速为 4mph（风寒度标准）的情况下产生相同寒冷感的温度。V 是以米/秒计的风速，Δt 是 33-temp，也就是中性皮肤

温度（摄氏 33 度）和气温之间的温度差。常量 A=10.45，B=10，C=-1。X=1.78816，它是 4mph 转换为米/秒的值。

★★ 6. 用于计算抵押的月付金额的公式是：

$$P=\frac{AI}{1-(1+I)^{-N}}$$

A 是贷款的数量，I 是每个时段的利率（小数形式，而不是百分数形式），N 是贷款需要支付的时段数。例如，一笔$100 000 的 20 年期利率 8%的贷款每月需要支付$836.44（20 年共有 240 个支付时段，每个支付时段的利率为 0.66667）。

编写一个函数，它的原型如下所示，计算每月支付的贷款。

```
double payment( double amount, double interest,  int years );
```

years 指定货款的时期，amount 是贷款的数量，interest 是用百分数形式（例如，12%）表示的年利率。函数应该计算并返回贷款的月付金额，四舍五入至美分。

★★★ 7. 设计良好的随机数生成函数所生成的值看上去很像随机数，但随着时间的延长，其结果会显示出一致性。从随机值派生而来的数字也具有这些属性。例如，一个设计欠佳的随机数生成函数的返回值看上去像是随机数，但实际上却是奇数和偶数交替出现。如果对这些看似的随机数对 2 取模（例如，用于模拟抛硬币的结果），其结果将是一个 0 和 1 交叉的序列。另一种较差的随机数生成函数只返回奇数值。把这些值对 2 取模的结果将是一个连续的 0 序列。这两类值都无法作为随机数使用，因为它们不够“随机”。

编写一个程序，在你的系统中测试随机数生成函数。你应该生成 10 000 个随机数并执行两种类型的测试。首先是频率测试，把每个随机数对 2 取模，看看结果 0 和 1 的次数各有多少。然后对 3 到 10 做同样的测试。这些结果将不会具有精确的一致性，但各个余数在频率上的峰谷差异不应该太大。

其次是周期性频率测试，取每个随机数和它之前的那个随机数，将它们对 2 取模。使用这两个余数作为一个二维数组的下标并增加指定位置的值。对 3～10 重复进行上面的取模测试。同样，这些结果将不会具有很严格的规律，但应该具有近似的一致性。

修改你的程序，这样你可以为随机数生成函数提供不同的种子，并对使用几个不同的种子所产生的随机值进行测试。你的随机数生成函数是不是足够优秀？

★★★ 8. 某个文件包含了家庭成员的年龄。同一个家庭成员的年龄位于同一行，中间由一个空格分隔。例如，下面的数据

```
45  42  22
36  35  7  3  1
22  20
```

描述了三个分别具有 3 个、5 个和 2 个成员的家庭的年龄。

编写一个程序，计算用这种文件形式表示的每个家庭的平均年龄。它应该使用%5.2f 格式打印平均年龄，后面跟一个冒号和输入数据。这个问题和前一章的编程练习类似，但它没有家庭成员的数量限制！但是，你可以假定每个输入行的长度不超过 512 个字符。

★★★ 9. 在一个有 30 名学生的班级里，两个学生的生日是同一天的概率有多大？如果一群

人中两个成员的生日是同一天的概率为 50%，那么这个人群应该有多少人？

编写一个程序，回答这些问题。取 30 个随机数，并把它们对 365 取模，分别表示一年内的各天（忽略闰年）。然后对这些值进行检查，看看有没有相同的。重复这个测试 10 000 次，对这个频率作一个估计。

为了回答第 2 个问题，对程序进行修改，它把人数作为一个命令行参数，把当天的时间作为随机数生成函数的种子，数次运行这个程序，以获得这个概率较为精确的估计值。

★★★★ 10. 插入排序(insertion sort)就是逐个把值插入到一个数组中。第 1 个值存储于数据的起始位置。每个后续的值在数组中寻找合适的插入位置，如果需要的话，对数组中原有的值进行移动以留出空间，然后再插入该值。

编写一个名叫 insertion_sort 的函数执行这个任务。它的原型应该和 qsort 函数一样。

提示：考虑把数组的左边作为已排序的部分，右边作为未排序的部分。最初已排序部分为空。当你的函数插入每个值时，已排序部分和未排序部分的边界向右移动，以便插入。当所有的元素都被插入时，未排序部分便为空，数组排序完毕。