第十一章 标准库

标准库是 ANSI C 语言标准的一个重要组成部分。在 ANSI C 标准之前，不同的 C 系统都提供了库，其中包含许多常用功能函数，以及相关的类型与变量定义。随着发展，不同 C 系统所提供的库之间的差异也逐渐显露出来。为了提高 C 程序在不同系统之间的可移植性，ANSI C 标准将库的标准化作为一项重要工作，最终结果就是这里要讨论的标准库。

目前各种 C 语言系统都提供了标准库的所有功能，并按照标准库的规范，提供了一组标准头文件。此外，大多数 C 语言系统还根据自己的需要和运行环境情况，提供了许多扩充的库功能。典型的例子如图形库、直接利用操作系统甚至计算机硬件功能的库等等。一些特殊的 C 语言系统还提供了其他的库。此外，还有一些第三方软件供应商和公开软件开发者发布了许多通用和专用的 C 语言支持库，可以用于特定的系统或者特定的应用领域。如果我们需要开发某些系统，尽可能利用已有的经过长期考验的库是一种很好的选择。

如果我们写的程序中只使用了标准库，那么这个程序就更容易移到另一 C 语言系统上，甚至移到另一种计算机上使用。如果在程序里使用了系统的扩充库，那么要将这样的程序搬到其他系统里使用，就需要做更多工作，需要修改所有使用特殊库的部分，将它们用新系统能够支持的方式重新写出来。

上述说法实际上告诉我们，在开发 C 程序时，首先应当尽可能使用 C 语言本身的功能和标准库。如果迫不得已必须使用具体 C 系统的特殊库功能，那么就应该尽量将依赖于特殊功能的程序片段封装到一些小局部中。这种做法能保证最终程序具有较好的可移植性，在将程序转到其他系统时，需要做的工作比较少。

新的 C99 标准对标准库做了一些扩充，有关情况在这里也有简单介绍。

11.1 标准库结构

一个 C 语言系统里的标准库通常包含了一组标准头文件和一个或几个库函数代码文件。有关标准头文件的情况前面已有很多讨论，在写 C 程序时，我们只需用 #include 预处理命令引入相关头文件，就可以保证程序里能够正确使用标准库功能了。

库代码文件里主要是各个标准函数的实际机器指令代码段，还有一些相关数据结构（一些实现标准库所需的变量等），可能还附带着一些为连接程序使用的信息。当然，库文件都是二进制代码文件，其具体内容和形式都不是我们需要关心的。如果在一个程序中用到某些标准函数，在程序连接时，连接程序就会从库代码文件里提取出有关函数的代码和其他相关片段，把它们拼接到结果程序里，并完成所有调用的连接。

库代码文件通常是一个或者几个很大的文件，其中包含了所有库函数的定义。而在我们的一个具体程序里，实际使用的库功能只是其中很少一部分。人们开发了这样的技术，在进行程序连接时，连接程序并不把库代码文件整个装配到可执行文件里，而是根据实际程序的需要，由库文件里提取出那些必要部分，只把这些部分装配进去。这样就保证了用户程序的紧凑性，避免程序中出现大量无用冗余代码段的情况。

标准头文件在 ANSI C 语言定义里有明确规定。这是一组正文文件，它们的作用就是为使用标准库函数的源程序提供信息。在这些头文件里列出了各个库函数的原型，定义了库函数所使用的有关类型（如表示流的 `FILE` 结构类型等）和一些符号常量（如 EOF、NULL）。通过预处理命令包含这些头文件，将使编译程序在处理我们的程序时能得到所有必要的信

息，这就可以保证程序中对标准库的使用与库文件里有关定义之间的一致性。

标准头文件通常存放在 C 语言系统的主目录下的一个子目录里，这个目录的名字一般是 h 或者 include。标准头文件包括：

```
<asset.h>    <ctype.h>     <errno.h>     <float.h>
<limits.h>   <locale.h>    <math.h>      <setjmp.h>
<signal.h>   <stdarg.h>    <stddef.h>    <stdlib.h>
<stdio.h>    <string.h>    <time.h>
```

下面几节将根据 ANSI C 标准库的头文件，分门别类介绍标准库各方面的主要功能。小节标题后面括号里给出有关头文件的名字，这些名字总用尖括号括起来。为了帮助理解，各节中还将根据需要提供一些简短的程序实例。

11.1.1 标准定义（<stddef.h>）

文件<stddef.h>里包含了标准库的一些常用定义，无论我们包含那个标准头文件，都会自动将<stddef.h>包含进来。这个文件里定义：

- 类型 size_t（sizeof 运算符的结果类型，是某个无符号整型）；
- 类型 ptrdiff_t（两个指针相减运算的结果类型，是某个有符号整型）；
- 类型 wchar_t（宽字符类型，是一个整型，其中足以存放本系统所支持的所有本地环境中的字符集的所有编码值。这里保证空字符的编码值为 0）；
- 符号常量 NULL（空指针值）；
- 宏 offsetor（这是一个带参数的宏，第一个参数应是一个结构类型，第二个参数应是结构成员名。offsetor(s,m)求出成员 m 在结构类型 t 的对象中的偏移量）。

其中有的定义也出现在其他头文件里（如 NULL）。

11.1.2 错误信息（<errno.h>）

文件<errno.h>里定义了 errno，这是一个 int 类型的表达式（可能通过宏定义）。errno 可以看作一个变量，其初始值为 0，任何标准库函数执行中出错都可能将它设置为非 0 值，但任何标准库函数的执行都不会出现将它设置为 0 的动作。

如果需要检查某些标准库函数执行时是否出现错误，可以写下面这样的程序片段：

```
int err;
... ...
err = errno; /* 保留原出错状态，如果不需要恢复也可以不保留 */
errno = 0;
... ...  /* 被检查的程序片段 */
if (errno != 0) {
    ... ... /* 出错时的处理 */
}
errno = err; /* 恢复原出错状态（如果需要） */
```

是否保留与恢复出错状态要看程序的需要。但无论如何，在被检查的程序段之前都需要将 errno 置 0，以保证检查结果的正确性。

在<errno.h>里还定义了两个宏 EDOM 和 ERANGE，它们都是非 0 整数值。如果数学函数执行中遇到参数错误（参数超出了函数的允许范围），就会将 errno 设置为 EDOM。如果数学函数计算中出现值域错误（结果溢出），就会将 errno 设置为 ERANGE。

此外，标准库还允许具体系统定义一些以 E 开头的宏，用于表示具体 C 系统的标准库可能给 errno 设置的值。

11.2 几个已经介绍过的头文件

标准库中几个头文件的主要功能已在前面章节里做过介绍，这里做一些必要的补充或说

明。没有介绍过的头文件和还有重要部分没介绍的头文件单独分节介绍。

标准输入输出函数中的原型在<stdio.h>，其中的常用函数及已在第 8 章仔细讨论过了，这里不再重复。在<stdio.h>里还有一些输入输出的辅助函数和几个以 valist（变长参数表）为参数的函数，在 11.8 节里介绍它们的情况。

11.2.2 数学函数（<math.h>）

除了第 2.4.2 节介绍了其中的许多函数，包括三角函数：

三角函数	sin	cos	tan
反三角函数	asin	acos	atan
双曲函数	sinh	cosh	tanh

指数和对数函数：

以 e 为底的指数函数	exp
自然对数函数	log
以 10 为底的对数函数	log10

其他函数包括：

平方根	sqrt
绝对值	fabs
乘幂，第一个参数作为底，第二个是指数	double pow(double, double)
实数的余数，两个参数分别是被除数和除数	double fmod(double, double)

上面所有没给出类型特征的函数都要求一个参数，其参数与返回值都是 double 类型。

除了上面列出的函数外还有几个函数，现将它们列在下表中。这些函数都返回双精度值（包括函数 ceil 和 floor，这两个函数返回的是对应于整数的双精度值）。在下表里，除其中有特别说明的参数之外，函数的其他参数都是 double 类型的。有关函数包括：

函数原型	意义解释
ceil(x)	求出不小于 x 的最小整数
floor(x)	求出不大于 x 的最大整数
atan2(y, x)	求出 $\tan^{-1}(y/x)$，其值的范围是 $[-\pi,\pi]$
ldexp(x, int n)	求出 $x\cdot 2^n$
frexp(x, int *exp)	把 x 分解为 $y\cdot 2^n$，y 是位于区间 $[1/2,1)$ 里的一个小数，作为函数值返回，整数 n 通过指针*exp 返回（应提供一个 int 变量地址）。当 x 为 0 时两个值都是 0
modf(x, double *ip)	将 x 分解为小数部分和整数部分，小数部分作为函数返回值，整数部分通过指针*ip 返回。

下面是使用 frexp 和 modf 的两个简单实例：

```
double x = 233.3574, y, z1, z2;
int n;
y = frexp(x, &n);
z1 = modf(x, &z2);
```

这两个语句将得到对 double 值 x 的两种不同分解。

11.2.2 字符处理函数（<ctype.h>）

所有字符函数都已在第 5.2.2 节介绍，见下表：

int isalpha(c)	c 是字母字符
int isdigit(c)	c 是数字字符
int isalnum(c)	c 是字母或数字字符
int isspace(c)	c 是空格、制表符、换行符
int isupper(c)	c 是大写字母
int islower(c)	c 是小写字母
int iscntrl(c)	c 是控制字符

int isprint(c)	c 是可打印字符，包括空格
int isgraph(c)	c 是可打印字符，不包括空格
int isxdigit(c)	c 是十六进制数字字符
int ispunct(c)	c 是标点符号
int tolower(int c)	当 c 是大写字母时返回对应小写字母，否则返回 c 本身
int toupper(int c)	当 c 是小写字母时返回对应大写字母，否则返回 c 本身

这些判断函数在条件成立时返回非 0 值，不成立时返回 0 值。最后两个转换函数对于非字母参数返回原字符。

11.3 字符串函数（<string.h>）

几个最常用的字符串函数已在第 6.4.4 节介绍。这里要补充一点，那里介绍的串复制函数 strcpy、strncpy 和串连接函数 strcat、strncat 都返回指向结果字符串的指针值。标准库还提供了另一些字符串处理函数。

这个头文件里还提供了一批存储区函数，这些函数与字符串函数类似，都是对一系列字节做各种处理。两者的差异在于字符串函数依赖于 C 语言所规定的字符串表示，即依赖于表示字符串结束的空字符。存储区处理函数不依赖于特定表示，调用它们时必须给定被处理的字节个数。

11.3.1 一些字符串函数

现将所有的函数都列在下表里，已经介绍过的函数就不再仔细讨论了。为了描述简洁，下表里的函数说明采用了如下约定：参数表中的 s、t 表示的是(char *)类型的参数，cs、ct 表示(const char *)类型的参数（它们都应该表示字符串）。n 表示 size_t 类型的参数（前面说过，size_t 是一个无符号的整数类型），c 是整型参数（在函数里转换到 char 使用）。表中函数都在<string.h>里说明：

函数原型	意义解释
size_t strlen(cs)	求出 cs 的长度
char *strcpy(s,ct)	将 ct 复制到 s。要求 s 指定足够大的字符数组
char *strncpy(s,ct,n)	将至多 n 个字符从 ct 复制到 s。要求 s 指定足够大的字符数组。如果 ct 中字符不足，将 s 中填充空字符。
char *strcat(s,ct)	将 ct 中字符复制到 s 已有字符串之后。s 应指定一个保存着字符串，且足够大的字符数组。
char *strncat(s,ct,n)	将 ct 中至多 n 个字符复制到 s 已有字符串之后。s 应指定一个保存着字符串，且足够大的字符数组。
int strcmp(cs,ct)	比较字符串 cs 和 ct 的大小，在 cs 大于、等于、小于 ct 时分别返回正值、0、负值。
int strncmp(cs,ct,n)	比较字符串 cs 和 ct 的大小，至多比较 n 个字符。在 cs 大于、等于、小于 ct 时分别返回正值、0、负值。
char *strchr(cs,c)	在 cs 中查寻 c 并返回 c 第一个出现的位置，用指向这个位置的指针表示，cs 里没有 c 时返回值 NULL
char *strrchr(cs,c)	在 cs 中查寻 c 并返回 c 最后一个出现的位置，没有时返回 NULL
size_t strspn(cs,ct)	由 cs 起确定一段全由 ct 里的字符组成的序列，返回其长度
size_t strcspn(cs,ct)	由 cs 起确定一段全由非 ct 的字符组成的序列，返回其长度
char *strpbrk(cs,ct)	在 cs 中查寻 ct 的字符，返回第一个这种字符出现的位置，没有时返回 NULL
char *strstr(cs,ct)	在 cs 中查寻串 ct（查询子串），返回 ct 作为 cs 的子串的第一个出现的位置，ct 未出现在 cs 里时返回 NULL
char *strerror(n)	返回与错误编号 n 相关的错误信息串（指向该错误信息串的指针）
char *strtok(s,ct)	在 s 中查寻由 ct 中的字符作为分隔符而形成的单词

下面对前面没有介绍过的函数在做一些解释，其中对 strtok 的功能做了较详细的讨论。

strerror 的情况比较简单。本章开始时讲过，在程序里可以检查 C 标准库中表达式 errno 的值，当这个表达式非 0 时，可以通过 strerrno(errno)得到一个字符指针，它指向系统提供的一个有关当时错误的信息串。

字符串长度

函数 strlen 返回字符串中的实际字符个数，这一长度不包括表示结束的空字符。

```
char ss[] = "there is an island in the sea.";
int n;
n = strlen(ss);
```

执行这个语句后 n 的值将是 30。

有时我们会需要为保存得到的字符串而动态分配一块存储区。在需要这样做时，就可以用函数 strlen 求出字符串长度，但分配时一定要注意加上空字符的位置。下面函数建立参数字符串的一个副本，返回指向这个副本字符串的指针：

```
char *makecopy(char *s) {
    char *p = (char*)malloc((strlen(s)+1));
    strcpy(p, s);
    return p;
}
```

如果没有+1，调用 strcpy 复制字符串时就会越界，可能造成严重问题。在上面分配中计算大小时不必去乘 sizeof(char)，因为 C 语言规定 char 类型的大小为 1。

字符和子串查找

函数 strchr 和 strrchr 查找一个字符在一个字符串里出现的位置，它们分别从字符串得左边和右边开始查寻。函数 strpbrk 将一个串看作字符的集合，查找其中任何一个字符在另一个字符串里出现的位置，函数 strstr 查找一个串（这时称为子串）在另一个字符串里出现的位置。如果查找成功，这些函数就返回指向所确定位置的指针，失败（没有找到）时返回空指针值。例如：

```
char ss[] = "there is an island in the sea.";
char *p1, *p2, *p3;
p1 = strchr(ss, 'n');
p2 = strrchr(ss, 'n');
p3 = strstr(ss, "is");
```

执行这几个语句后，p1 将指向 ss 里单词 an 中的 n（这是 ss 里从左边起的第一个 n）；p2 将指向单词 in 中的字符 n（ss 里从右边起的第一个 n）；p3 将指向 ss 里的单词 is 的第一个字符（虽然后面还有另一个 is 出现，作为单词 island 的一部分）。

如果要确定在某个字符在一个字符串里的所有出现位置，我们可以写一个循环。例如，下面循环将输出字符串 ss 里所有 i 的位置：

```
p = ss;
while (1) {
    if ((p1 = strchr(p1, 'i')) == NULL) break;
    printf("%d ", p1 - ss);
}
```

在找到需要的字符位置后，我们当然可以做任何需要做的事情。这里将找到得位置输出只是作为一个例子。对于子串查找也可以写出类似的循环。

特定的最长前缀

字符串的一个前缀是指该串前面的一段。函数 strspn 和 strcspn 都可用于确定一个字符串中满足某种要求的最长前缀的长度。strspn 查找的是所有字符都属于某个集合（字符集合用另一个字符串表示）的最长前缀，而 strcspn 查找的是所有字符都不属于某个集合的最长前缀。下面程序片断确定输入的一行中从头开始最长数字序列的长度：

```
char s[256];
fgets(s, 256, stdin);
n = strspn(s, "0123456789");
```

strcspn 的情况与此类似，只是它以不属于指定字符集合 ct 作为判据。

单词分解

函数 strtok(s,ct)的功能比较复杂，用途是把一个字符串划分为单词的序列。这个函数把字符串看作由一集分隔符分隔开的一个个单词。函数的参数 ct 表示为完成划分所选定的分隔字符集合，s 是被处理字符串（自然，是指向字符串的指针）。

对一个非 NULL 的字符串 s 的划分通过对 strtok 的一系列调用完成。第一次调用时对字符串 s 调用本函数，将确定在 s 里的第一个完全由不属于 ct 的字符构成的段。这一调用还把紧跟在这段字符之后的一个字符修改为'\0'，并返回指向这段字符开始处的指针（这样就形成了一个由这些字符构成的字符串）。随后的所有调用都应当用 NULL 作为 strtok 的第一个参数，每次调用将返回被处理字符串里的下一个单词（作为一个字符串）。我们可以这样做下去，直到没有下一单词时 strtok 返回空指针值。

假定我们希望写一个程序，要求它找出正文文件里包含某个特定单词的所有正文行，并将这些行的内容及其顺序编号送到标准输出。作为一种设计，我们可以假定被查找文件通过函数的标准输入得到，被查找的单词通过命令行参数给出。

借助于函数 strtok，这个程序可以写成下面的样子：

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

#define LINELEN 256
char sep[] = " \t";

int main(int argc, char *argv[]) {
    int n;
    char *scan, *p;
    char ss[LINELEN], save[LINELEN];
    if (argc == 1) {
        printf("You forget to give a scan-string\n.");
        return 1;
    }
    scan = argv[1]; /* 指向要找的字符串 */

    for (n = 1; fgets(ss, 256, stdin) != NULL; ++n) {
        strcpy(save, ss);
        for (p = strtok(ss, sep); p != NULL; p = strtok(NULL, sep))
            if (strcmp(p, scan) == 0) {
                printf("%d: %s", n, save);
                break;
            }
    }

    return 0;
}
```

这里用变量 n 记录行编号，用 save 保留读入行（因为 strtok 在分割过程中会修改它所操作的行 ss）。开始读入正文行之前用变量 scan 指向要查找的字符串，随后的循环实现处理过程：一次读入一行，而后按照字符串 sep 的定义（注意，我们的 sep 包含两个字符：空格和制表符），利用 strtok 将字符串分解为一个个单词。如果被确定的单词与 scan 相同，就输出这一行（保留在 save 里）。

在程序里把分隔字符定义成字符数组 sep，这就使修改程序变得更容易了。例如，如果要做一个程序查找 C 语言源程序里出现特定标识符的行，我们只需要修改这个字符数组的定义就可以了。有关修改也留给读者作为练习。

上面的程序在开始处定义了一个符号常量，确定输入用的行缓冲区大小。这里实际上假

设每行的字符不超过 254 个。如果这一假设不成立，那么行的序号就可能出错（例如，超长的行被作为多个行计数），也可能出现漏查的情况（例如被查字符串正好出现在超长行的断行处）。修正这些缺陷都是可能的，例如可以改用动态分配的存储区，或者另外想办法处理行的截断问题。有关修正留给读者作为练习。

当然，这个程序完全是为了演示 strtok 的使用，如果就为了查找有单词的行，也完全可以采用其他处理方式。但这里所演示的技术在处理更复杂的问题时可能有价值。

函数 strtok 的实现可以作为一个很好的练习。

11.3.2 存储区操作

<string.h>里除定义了上述字符串操作函数外，还定义了一组字符数组操作函数，或称“存储区操作函数”。这些函数的名字都以 mem 开头，通常以某种高效方式实现。在下面所有原型说明中，参数 s 和 t 的类型是(void *)，cs 和 ct 的类型是(const void *)，n 的类型是 size_t，c 的类型是 int（转换为 unsigned char）。

函数原型	意义解释
void *memcpy(s,ct,n)	从 ct 处复制 n 个字符到 s 处，返回 s
void *memmove(s,ct,n)	从 ct 处复制 n 个字符到 s 处，返回 s，这里的两个段允许重叠
int memcmp(cs,ct,n)	比较由 cs 和 ct 开始的 n 个字符，返回值定义同 strcmp
void *memchr(cs,c,n)	在 n 个字符的范围内查寻 c 在 cs 中的第一次出现，如果找到，返回该位置的指针值，否则返回 NULL
void *memset(s,c,n)	将 s 的前 n 个字符设置为 c，返回 s

函数 memcpy 简单地完成一段内存内容（长度 n 个字节）的复制工作，如果作为复制源和复制目标的两个区域有重叠，它未必能正确完成复制工作。memmove 的功能与 memcpy 相同，但是即使复制源和复制目标有重叠，它也能保证正确完成复制工作。因此这个函数的使用更安全，但另一方面，这一函数的效率可能比 memcpy 低一些。

在一般 C 系统里，标准库的这几个函数都经过了特殊的优化，它们可能比功能类似的字符串函数效率更高一些。如果程序中的效率是重要问题，那么就值得考虑采用这几个函数。还请注意，这几个函数都带有长度参数 n，表示按字符数计算的长度。

11.4 功能函数（<stdlib.h>）

头文件<stdlib.h>说明各种常用功能函数，其中随机数生成和动态存储分配的有关函数在第五章和第七章分别做了说明（第 5.2.3 节，第 7.6 节），其中的随机数生成函数包括：

函数原型	意义解释
int rand(void)	生成一个 0 到 RAND_MAX 的随机整数
void srand(unsigned seed)	用 seed 为随后的随机数生成设置种子值

与动态存储分配有关的函数包括：

函数原型	意义解释
void *calloc(size_t n, size_t size)	分配一块存储，其中足以存放 n 个大小为 size 的对象，并将所有字节用 0 字符填充。返回该存储块的地址。不能满足时返回 NULL
void *malloc(size_t size)	分配一块足以存放大小为 size 的存储，返回该存储块的地址，不能满足时返回 NULL
void *realloc(void *p, size_t size)	将 p 所指存储块调整为大小 size，返回新块的地址。如能满足要求，新块的内容与原块一致；不能满足要求时返回 NULL，此时原块不变
void free(void *p)	释放以前分配的动态存储块

下面介绍在这个头文件里说明的其他函数。

11.4.1 几个整数函数

几个简单的整数函数见下表。这里的 div_t 和 ldiv_t 是两个预定义的结构类型，用于存放整除时得到的商和余数。div_t 类型的成分是 int 类型的 quot 和 rem，ldiv_t 类型的成分是 long 类型的 quot 和 rem。

函数原型	意义解释
int abs(int n)	求整数的绝对值
long labs(long n)	求长整数的绝对值
div_t div(int n, int m)	求 n/m，商和余数分别存放到结果结构的对应成员里
ldiv_t ldiv(long n, long m)	同上，参数为长整数

11.4.2 数值转换

这组函数用于从数字字符串构造各种数值类型的值。几个常用的简单函数是：

函数原型	意义解释
double atof(const char *s)	由串 s 构造一个双精度值
int atoi(const char *s)	由串 s 构造一个整数值
long atol(const char *s)	由串 s 构造一个长整数值

与上面这三个函数对应的有三个更通用的数值构造函数，它们分别是：

1）字符串到双精度数的转换函数 strtod。原型是：

```
double strtod(const char *s, char **endp);
```

本函数忽略字符串 s 最前面的空白，由随后的一段转换生成一个双精度值返回。如果字符串 s 没用完，函数将给指针*endp 赋值，令它指向字符串剩下部分的开始位置（如果 endp 是空指针就不赋值）。如果得到的值超出 double 类型的表示范围（称为上溢），则返回预定义常量 HUGE_VAL 的值（这是一个很大的 double 值），并带上适当符号。如果得到的值小于 double 类型能表示的除了 0 之外的最小值（称为下溢）则返回值为 0。这两种情况下都将错误标志变量 errno 设置为预定义常数 ELARGE。

函数调用 atof(s)实际上等价于 strtod(s, (char **)NULL)。

2）字符串到长整数的转换函数 strtol。原型是：

```
long strtol(cnost char *s, char **endp, int base);
```

这个函数与 strtod 类似，只是转换目标为长整数。当 base 取值在 2 到 36 范围内时，假定被转换字符串 s 是按照 base 进位制书写的（数字依次用 0 到 9 和 a 到 z 表示，共计 36 个，最多能表示 36 进制的数）。如果 base 取值 0，那么就根据 s 的内容确定按八进制、十进制或十六进制方式转换（引导的 0 表示八进制，0X 或 0x 表示十六进制，否则按十进制）。如果转换结果超出长整数的表示范围，根据数的符号返回常数 LONG_MAX 或 LONG_MIN，并将错误标志变量设置为预定义常数 ELARGE。

函数调用 atoi(s)等价于(int)strtol(s, (char **)NULL, 10)，而 atol(s)等价于 strtol(s, (char **)NULL, 10)。

3）字符串到无符号长整数的转换函数 strtoul。原型是：

```
unsigned long strtoul(cnost char *s, char **endp, int base);
```

转换结果类型为 unsigned long，出错时返回 ULONG_MAX，其他与 strtol 相同。

11.4.3 执行控制

这里还提供了几个有关程序执行结束的函数，它们是：

1）非正常终止函数 abort。原型是：

```
void abort(void);
```

导致程序按照非正常方式立即终止。

2）正常终止函数 exit。原型是：

```
void exit(int status);
```

导致程序按正常方式立即终止。status 作为送给程序执行环境的出口值，用 0 表示程序执行成功结束，两个可以使用的常数为 EXIT_SUCCESS，EXIT_FAILURE。程序结束前更新完毕所有打开的文件（如果需要），并关闭所有的流。当程序以正常方式结束时，可以要求它在结束之前做一些动作，这些动作通过下面介绍的函数 atexit 登记，程序结束前按照登记的先后顺序，反向地一个个执行这些函数（abort 函数执行时不完成这些动作）。

3）正常终止注册函数 atexit。原型是：

```
int atexit(void (*fcn)(void))
```

可以用这个函数把一些函数注册为结束动作。被注册函数应当是无参数、无返回值的函数。注册正常完成时 atexit 返回值 0，否则返回非零值。

11.4.4 与执行环境交互

1）向执行环境传送命令的函数 system。原型是：

```
int system(const char *s);
```

函数 system 把串 s 传递给程序的执行环境，要求把这个串作为一个系统命令执行。如果以参数 s 为 NULL 调用函数，返回非零值表示执行环境里有命令解释器（这提供了一个检查命令解释器存在的方法）。如果 s 不是 NULL，返回值由实现确定。

2）访问执行环境的函数 getenv。原型是：

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

从执行环境中取回与字符串 s 相关联的环境串。如果找不到就返回 NULL。函数的具体结果由实现确定。在许多执行环境里，可以用这个函数去查看“环境变量”的值。

11.4.5 常用函数 bsearch 和 qsort

查找和排序是程序中经常要做的工作，bsearch 和 qsort 是标准库提供的两个通用函数。bsearch 用于完成在一批数据里查找某个数据项的工作，qsort 用于把一组数据按照某种指定方式顺序排列。这两个函数的参数都比较多，在第九章里已经介绍过函数 qsort 的使用。

1）二分法查找函数 bsearch。原型是：

```
void *bsearch(const void *key, const void *base,
              size_t n, size_t size,
              int (*cmp)(const void *keyval, const void *datum));
```

函数指针参数 cmp 的实参应当是一个与字符串比较函数 strcmp 类似的函数，它能确定一种顺序，在它的第一个参数 keyval 与第二个参数 datum 相比更大、相等或者更小时，函数应分别返回正值、零或者负值。

假设程序里有一个数组 base[0],...,base[n-1]，其中的元素是按照函数 cmp 所确定的顺序上升排列的，这一数组的元素大小是 size。当我们调用函数 bsearch，以*key 作为比较的关键码在数组里查找匹配元素（使函数 cmp 的值等于 0 的元素）时，函数将返回指向所找到的元素的指针。找不到时返回 NULL。

2）快速排序函数 qsort。原型是：

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *));
```

qsort 对比较函数 cmp 的要求与 bsearch 相同。设有数组 base[0],...,base[n-1]，元素大小为 size。利用 qsort 可以把这个数组的元素按照上升顺序重新排列。

下面是说明函数 bsearch 和 qsort 使用的一个例子：

```
int *p, a[] = {5, 6, 3, 28, 23, 34, 7, 9, 6, 14}, k = 7;
int icmp(const void *p, const void *q){
    const int *m = p, *n = q;
    return *m > *n ? 1 : (*m == *n ? 0 : -1);
}

int main () {
    ... ...
    qsort(a, sizeof(a)/sizeof(int), sizeof(int), icmp);
        /* 这样，数组a的元素已经按照上升顺序排列好了 */
    p = bsearch(&k, a, sizeof(a)/sizeof(int), sizeof(int), icmp);
        /* 指针p将指向数组a中元素7的位置 */
    ... ...
}
```

在写函数调用时需要提供数组元素的大小，所用的比较函数通常也需要自己定义。这样，这两个函数的使用是非常灵活的。例如，我们也可以用 qsort 对以结构作为成分的数组进行排序，在比较“大小”时可以只使用相应结构的某一个成分或几个成分，而且可以根据程序的需要，定义任何比较方式。参看第九章的有关实例。

11.5 日期和时间（<time.h>）

头文件<time.h>定义了几个与时间有关的类型，说明了许多与时间处理有关的函数。time_t 和 clock_t 是两个表示时间的算术类型：类型 clock_t 用于表示计时时间；类型 time_t 用于表示某种日历时间，这种时间从历史上的某个时刻开始计算。

此外，这里还定义了一种表示本地时间的结构 struct tm，它包含下述成分：

int tm_sec;	剩余秒数（0～61）	int tm_year;	1900 后的年序数
int tm_min;	剩余分钟数（0～59）	int tm_wday;	星期中日序数（0～6）
int tm_hour;	剩余小时数（0～23）	int tm_yday;	年中日序数（0～365）
int tm_mday;	月中日数（1～31）	int tm_isday;	夏季时标志
int tm_mon;	年中月数（1～12）		

结构成分 tm_isday 取正值表示采用的是夏季时，取零值表示不用夏季时，取负值表示不存在有关信息。程序里可以使用这种结构，进而访问其中的各个成分，下面介绍的一些时间函数也使用或者生成这种结构。请注意，这里的 tm 只是一个结构标志（不是类型），使用时必须写 struct tm。

下面介绍头文件<time.h>说明的函数，其中函数 clock()已经在第四章里介绍过：

1）计时函数 clock，原型是：

```
clock_t clock(void)
```

求出从程序开始执行到本函数调用时刻的处理器时间，用 clock()/CLOCKS_PER_SEC 得到的是以秒数计的时间。如果不能得到有关时间信息，函数返回-1。CLOCKS_PER_SEC 是本文件中定义的一个符号常量，有些老系统用的是 CLK_TCK。

2）时间函数 time，原型是：

```
time_t time(time_t *tp)
```

求出当时的日历时间（从某个确定时刻开始计时的时间）。如果 tp 不是 NULL，那么同时也通过这个指针进行赋值。日历时间通常作为相对时间来使用（参看 difftime）。在无法得到有关时间信息时返回-1。

3）时间差函数 difftime，原型是：

```
double difftime(time_t t1, time_t t1)
```

求出两个日历时间的差，以秒计算。

4）时间转换函数 mktime，原型是：

```
time_t mktime(struct tm *tp)
```

从由结构*tp 表示的局部时间出发，通过转换得到与之对应的日历时间（结果采用与函数 time 一样的表示方式）。实参结构*tp 的各个成分应满足前面提出的值限制。mktime 返回日历时间；如果不能得到结果就返回-1。

下面四个函数的返回值都是指向某个静态变量的指针。这种说法意味着在标准库模块里的些地方定义了有关的静态变量，这些函数在完成计算时返回有关变量的地址。由于静态变量的生存期是程序的整个执行期间，这种做法不会产生悬空引用的问题。但也有一点必须注意：这些静态变量始终存在，它们也可能被有关函数的再次调用所修改。如果将调用某函数得到的指针记录在一个指针变量 p 里，再次调用该函数，p 所指的结构（就是相应的静态变量）内容通常会被改变。使用这些函数的计算结果时必须注意这种问题。

5）本地时间转换函数 localtime，原型是：

```
struct tm *localtime(const time_t *tp)
```

从*tp 表示的日历时间出发，通过转换得到对应的本地时间，将其存放在一个静态的 tm 结构变量里，返回这一结构的地址。

6）字符串时间转换函数 asctime，原型是：

```
char *asctime(const struct tm *tp)
```

由结构*tp 表示的本地时间出发进行转换，得到下面形式的字符串（最后有一个换行字符和表示字符串结束的空字符，这一字符串也保存在一个静态变量里）：

```
Sun Oct 4 18:00:00 1998\n\0
```

7）字符串时间转换函数 ctime，原型是：

```
char *ctime(const time_t *tp)
```

它等价于 asctime(localtime(tp))，从日历时间出发进行转换。

8）国际标准时转换函数 gmtime，原型是：

```
struct tm *gmtime(const time_t *tp)
```

由*tp 表示的日历时间出发，转换得到与之对应的国际标准时间（格林威治时间），将转换结果存入一个静态的 tm 结构变量里，返回这个结构的地址。如果无法表示就返回 NULL。

最后一个函数是为输出时间信息而设置的，它按照要求构造一个输出用的字符串。

9）时间格式化函数 strftime，原型是：

```
size_t strftime(char *s, size_t smax, const char *fmt,
                const struct tm *tp)
```

对*tp 所表示的时间做格式化处理，并把由这个处理生成的字符串存入由 s 指向的字符数组里。smax 给定输出字符数的限制。fmt 是一个与 printf 类似的格式描述串，其中的一

般字符直接复制到 s 里。由字符%开始加上另一个字符形成了一个转换描述。这些描述按照下面方式进行替换，替换结果存入 s 中的相应位置：

%a	简化的星期 X 的名字	%I	12 时记法的时数	%w	一周里日的序数（0～6）
%A	完整的星期 X 的名字	%j	日的序数（001～366）	%W	年中周的序数（00～53）
%b	简化月名	%m	月的序数（01～12）	%x	本地日期
%B	完整月名	%M	分钟数（00～59）	%X	本地时间
%c	本地日期和时间	%p	AM 或 PM，表示上下午	%y	去掉世纪部分的年序数
%d	月中日序数（1～31）	%S	秒数（00～61）	%Y	包括世纪部分的年序数
%H	24 时记法的时数	%U	年中周的序数（00～53）	%Z	时区编号

其中%U 要求把星期日作为每周里的第一天，而%W 要求把星期一作为每周的第一天。此外，转换描述%%表示百分号符号本身。

11.6 实现特征（<limit.h>和<float.h>）

ANSI C 标准留下了一些具体问题，要求每个具体 C 语言系统自己确定。例如，具体的 C 系统要规定各种整型的具体表示方式，由此决定它们的表示范围。还要给出浮点数类型的各方面规定等等。C 语言标准库包含两个特殊的头文件（<limit.h>和<float.h>），其中定义了一些符号常量，它们描述了由具体实现确定的各种特殊值。对于每个 C 语言系统，这些常量的值根据系统的实际情况给出来。这样，人们写程序时就可以使用这些符号常量，有助于写出不依赖于具体实现的程序。

11.6.1 整数类型特征

头文件<limit.h>里定义了一批与各种整数类型有关的符号常量。下表里列出了这些常量。其中给出了 ANSI C 标准所要求的最小值（考虑其绝对值的大小）。一个具体 C 语言系统所定义的值可能与下面列出的不同：

符号常量名	允许最小值（具体值）	意义解释
CHAR_BIT	8	char 类型的字位数
CHAR_MAX	SCHAR_MAX 或 UCHAR_MAX	char 类型的最大值
CHAR_MIN	0 或 SCHAR_MIN	char 类型的最小值
INT_MAX	+32767	int 类型的最大值
INT_MIN	-32767	int 类型的最小值
LONG_MAX	+2147483647	long 类型的最大值
LONG_MIN	-2147483647	long 类型的最小值
SCHAR_MAX	+127	signed char 类型的最大值
SCHAR_MIN	-127	signer char 类型的最小值
SHRT_MAX	+32767	short 类型的最大值
SHRT_MIN	-32767	short 类型的最小值
UCHAR_MAX	255	unsigner char 类型的最大值
UINT_MAX	65535	unsigner int 类型的最大值
ULONG_MAX	4294967295	unsigner long 类型的最大值
USHRT_MAX	65535	unsigner short 类型的最大值

11.6.2 浮点数类型特征

头文件<float.h>里定义了各种与浮点数类型和浮点数算术计算有关的常量。每个具体 C 语言系统都为这些常量指定了具体值。下表列出了的是与 float 类型有关的常量：

符号常量名	允许最小值	意义解释
FLT_RADIX	2	指数表示的基数（2，16 等）
FLT_ROUNDS		浮点数加运算时的舍入方式
FLT_DIG	6	十进制精度的位数

FLT_EPSILON	1E-5	与 0 不同的最小的数
FLT_MANT_DIG		尾数数字个数（以基 FLT_RADIX 计）
FLT_MAX	1E+37	float 类型的最大值
FLT_MAX_EXP		能表示的以 FLT_RADIX 为基的位数
FLT_MAX10_EXP		能表示的以 10 为基的位数
FLT_MIN	1E-37	float 类型的最小值
FLT_MIN_EXP		能表示的以 FLT_RADIX 为基的位数
FLT_MIN10_EXP		能表示的以 10 为基的位数

其中的 FLT_RADIX 和 FLT_ROUNDS 对所有浮点数类型都有效。

类型 double、long double 也各有一组类似的常量。double 对应的常量的名字分别是：DBL_DIG，DBL_EPSILON，DBL_MANT_DIG，DBL_MAX，DBL_MAX_EXP，DBL_MAX10_EXP，DBL_MIN，DBL_MIN_EXP，DBL_MIN10_EXP。

long double 类型对应的常量的名字分别是：LDBL_DIG，LDBL_EPSILON，LDBL_MANT_DIG，LDBL_MAX，LDBL_MAX_EXP，LDBL_MAX10_EXP，LDBL_MIN，LDBL_MIN_EXP，LDBL_MIN10_EXP。

11.7 定义变长度参数表（<stdarg.h>）

从这里开始的几节介绍标准库提供的一些特殊机制，每种机制实现某一种特殊功能。这些机制在复杂的程序里可能用到。要理解这些机制，我们不但要看到表面情况，还需要理解在程序中为什么需要它们，这实际上要求读者有更多的程序设计实践。本书对有关机制都做了较详细的解释，并提供一些小例子，以使本书对 C 语言的介绍比较完全，也供读者查询。初学者可以只对这些东西做一点初步了解，或者暂时不学习这部分的内容。

标准库提供的一些参数的数目可以有变化的函数。例如我们很熟悉的 printf，它需要有一个格式串，还应根据需要为它提供任意多个“其他参数”。这种函数被称作“具有变长度参数表的函数”，或简称为“变参数函数”。我们写程序中有时也可能需要定义这种函数。要定义这类函数，就必须使用标准头文件<stdarg.h>，使用该文件提供的一套机制，并需要按照规定的定义方式工作。本节介绍这个头文件提供的有关功能，它们的意义和使用，并用例子说明这类函数的定义方法。

一个变参数函数至少需要有一个普通参数，其普通参数可以具有任何类型。在函数定义中，这种函数的最后一个普通参数除了一般的用途之外，还有其他特殊用途。下面从一个例子开始说明有关的问题。

假设我们想定义一个函数 sum，它可以用任意多个整数类型的表达式作为参数进行调用，希望 sum 能求出这些参数的和。这时我们应该将 sum 定义为一个只有一个普通参数，并具有变长度参数表的函数，这个函数的头部应该是（函数原型与此类似）：

```
int sum(int n, ...)
```

我们实际上要求在函数调用时，从第一个参数 n 得到被求和的表达式个数，从其余参数得到被求和的表达式。在参数表最后连续写三个圆点符号，说明这个函数具有可变数目的参数。凡参数表具有这种形式（最后写三个圆点），就表示定义的是一个变参数函数。注意，这样的三个圆点只能放在参数表最后，在所有普通参数之后。

为了能在变参数函数里取得并处理不定个数的“其他参数”，头文件<stdarg.h>提供了一套机制。这里提供了一个特殊类型 va_list。在每个变参数函数的函数体里必须定义一个 va_list 类型的局部变量，它将成为访问由三个圆点所代表的实际参数的媒介。下面假设函数 sum 里所用的 va_list 类型的变量的名字是 vap。在能够用 vap 访问实际参数之前，必须首先用“函数”a_start 做这个变量初始化。函数 va_start 的类型特征可以

大致描述为：

```
va_start(va_list vap, 最后一个普通参数)
```

实际上 va_start 通常并不是函数，而是用宏定义实现的一种功能。在函数 sum 里对 vap 初始化的语句应当写为：

```
va_start(vap, n);
```

在完成这个初始化之后，我们就可以通过另一个宏 va_arg 访问函数调用的各个实际参数了。宏 va_arg 的类型特征可以大致地描述为：

```
类型 va_arg(va_list vap, 类型名)
```

在调用宏 va_arg 时必须提供有关实参的实际类型，这一类型也将成为这个宏调用的返回值类型。对 va_arg 的调用不仅返回了一个实际参数的值（“当前”实际参数的值），同时还完成了某种更新操作，使对这个宏 va_arg 的下次调用能得到下一个实际参数。对于我们的例子，其中对宏 va_arg 的一次调用应当写为：

```
v = va_arg(vap, int);
```

这里假定 v 是一个有定义的 int 类型变量。

在变参数函数的定义里，函数退出之前必须做一次结束动作。这个动作通过对局部的 va_list 变量调用宏 va_end 完成。这个宏的类型特征大致是：

```
void va_end(va_list vap);
```

下面是函数 sum 的完整定义，从中可以看到各有关部分的写法：

```
int sum(int n, ...) {
    va_list vap;
    int i, s = 0;
    va_start(vap, n);

    for (i = 0; i < n; i++) s += va_arg(vap, int);

    va_end(vap);
    return s;
}
```

这里首先定义了 va_list 变量 vap，而后对它初始化。循环中通过 va_arg 取得顺序的各个实参的值，并将它们加入总和。最后调用 va_end 结束。

下面是调用这个函数的几个例子：

```
k = sum(3, 5+8, 7, 26*4);
m = sum(4, k, k*(k-15), 27, (k*k)/30);
```

在编写和使用具有可变数目参数的函数时，有几个问题值得注意。首先，虽然在上面描述了头文件所提供的几个宏的“类型特征”，实际上这仅仅是为了说明问题。因为实际上我们没办法写出来有关的类型，系统在预处理时进行宏展开，编译时即使发现错误，也无法提供关于这些宏调用的错误信息。所以，在使用这些宏的时候必须特别注意类型的正确性，系统通常无法自动识别和处理其中的类型转换问题。

第二：调用 va_arg 将更新被操作的 va_list 变量（如在上例的 vap），使下次调用可以得到下一个参数。在执行这个操作时，va_arg 并不知道实际有几个参数，也不知道参数的实际类型，它只是按给定的类型完成工作。因此，写程序的人应在变参数函数的定义里注意控制对实际参数的处理过程。上例通过参数 n 提供了参数个数的信息，就是为了控制循环。标准库函数 printf 根据格式串中的转换描述的数目确定实际参数的个数。如果这方面信息有误，函数执行中就可能出现严重问题。编译程序无法检查这里的数据一致性问题，需要写程序的人自己负责。在前面章节里，我们一直强调对 printf 等函数调用时，要注意格式串与其他参数个数之间一致性，其原因就在这里。

第三：编译系统无法对变参数函数中由三个圆点代表的那些实际参数做类型检查，因为函数的头部没有给出这些参数的类型信息。因此编译处理中既不会生成必要的类型转换，也

不会提供类型错误信息。考虑标准库函数 printf，在调用这个函数时，不但实际参数个数可能变化，各参数的类型也可能不同，因此不可能有统一方式来描述它们的类型。对于这种参数，C 语言的处理方式就是不做类型检查，要求写程序的人保证函数调用的正确性。

假设我们写出下面的函数调用：

```
k = sum(6, 2.4, 4, 5.72, 6, 2);
```

编译程序不会发现这里参数类型不对，需要做类型转换，所有实参都将直接传给函数。函数里也会按照内部定义的方式把参数都当作整数使用。编译程序也不会发现参数个数与 6 不符。这一调用的结果完全由编译程序和执行环境决定，得到的结果肯定不会是正确的。

可以定义以 va_list 作为参数的函数，这里就不举例子了。下一节里的几个标准库输出函数可以看作这方面的实例。

11.8 其他与输入输出有关的函数（<stdio.h>）

<stdio.h>是我们最早接触的头文件，前面有关文件和输入输出的一章里又详细介绍了其中的与输入输出和文件处理有关的功能。本节介绍前面没有介绍的一些情况，包括一些与输入输出辅助函数，还要介绍几个以 va_list 为参数的输入输出函数。

11.8.1 符号常量和类型

前面已经介绍了<stdio.h>定义的一些符号常量，包括 EOF 和 NULL；用于文件定位函数的符号常量 SEEK_CUR，SEEK_END，SEEK_SET；最大临时文件数常量 TMP_MAX 和临时文件名最大长度 L_tmpname；用于控制文件缓冲方式的符号常量_IOFBF，_IOLBF 和_IONBF。也介绍了在这个文件里定义的与输入输出有关的类型 FILE（文件记录类型，是一个结构类型）和 fpos_t（文件位置类型，常常也是一个结构类型）。

这里还定义了下述符号常量：

- BUFSIZ，表示一个正整数，它是 setbuf 所用的缓冲区大小，可以用作 setvbuf 的表示缓冲区大小的参数。
- FOPEN_MAX，表示一个正整数，是允许同时打开的最大文件数。
- FILENAME_MAX，表示一个正整数，其值表示字符数组长度，这种字符数组足以保存本系统开发的 C 程序可以打开的最长的文件名。

11.8.2 文件操作函数

1）文件重新打开函数 freopen。其原型是：

```
freopen(const char *filename, const char *mode, FILE *stream);
```

这个函数按照 mode 描述的模式重新打开指定文件，并将该文件与流 stream 相关联。这一函数通常被用于改变标准流 stdio、stdout 或者 stderr 的文件关联。例如，我们希望将程序执行中通过 stderr 送出的错误都记录到文件 err.log 里，那么就只需在程序执行的准备阶段加入下述语句：

```
freopen("err.log", "w", stderr);
```

在工作正常完成时本函数返回 stream；执行中出错返回 NULL。如果函数执行完成，那么原先与 stream 关联的流将被关闭。

此外，在程序的执行过程中，我们也可能需要换一种方式使用原先已经打开的文件。例如，在某个程序的前面阶段创建了一个输出文件，并把程序中产生的许多数据存入其中了。在程序的后面又希望重头开始读入文件内容在程序里使用，一个可能的办法就是以读的方式重新打开这个流。

2）缓冲区冲刷函数 fflush。其原型是：

```
int fflush(FILE *stream);
```

这个函数只对输出流有效，对输入流的作用没有定义。fflush 要求从立刻执行一次从 stream 到与之关联的文件的实际写操作，把流 stream 缓冲区里现存的数据立即写到相应文件里去。操作正常完成时返回值 0，出错时返回 EOF。

3）文件删除函数 remove。其原型是：

```
int remove(const char *filename);
```

这个函数删除名字为 filename 的文件。工作正常时返回 0，无法正常完成工作时返回非 0 值。被删除的文件当时不应该是打开的，否则效果没有定义，由实现确定。

4）文件更名函数 rename。其原型是：

```
int rename(const char *oldname, const char *newname);
```

这一函数将原来名字为 oldname 的文件更名为 newname。操作失败就返回非 0 值。

5）临时文件创建函数 tmpfile。其原型是：

```
FILE *tmpfile(void);
```

本函数没有参数。它的执行将创建一个临时文件，并以模式 "wb+" 打开它。如果在程序里关闭本函数所创建的流，或者程序的执行结束，程序都会自动删除这个临时文件。在正常情况下本函数返回关联于该文件的流（文件指针），在函数无法创建并打开文件时返回 NULL 指针值。

我们在写程序时，有时需要保存一些程序运行的中间信息。如果这种信息的量很大，或者保存在内存不太方便，就可以借助于这个函数创建临时文件，将信息保存到文件里，供程序后面使用。程序结束时将自动删除这种临时创建的文件。

6）临时名字生成函数 tmpnam。其原型是：

```
char * tmpnam(char s[]);
```

这个函数生成一个可以用作文件名字的字符串，它保证所生成的名字不会是环境中现有的文件名。函数返回指向所生成名字字符串的指针。如果给 tmpnam 的实参是空指针 NULL，则函数将在一个内部的静态数组变量里保存所生成的名字，并返回这个字符串的地址。每次调用这个函数时将生成一个与以往不同的名字，而且保证至少可以生成 TMP_MAX 个不同的名字。如果参数 s 不是 NULL，那么就要求它表示一个至少可以保存 L_tmpnam 个字符的数组。此时函数将生成的名字存入这个数组里，返回数组的起始地址。

注意，标准库提供这一函数的意图是为程序生成内部使用的文件名。它只是生成一个可以用作文件名的字符串（其形式依赖于有关的操作系统环境），并没有创建文件。如果需要，就应该以这样得到的名字去创建文件。

11.8.3 流缓冲区操作函数

标准库允许在程序中设置输入输出流的缓冲区状态，为此提供了两个函数：

1）控制流缓冲的函数 setvbuf。函数原型是：

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

这个函数控制流 stream 的缓冲状态和方式。对它的调用必须是在流被创建之后，在执行任何其他操作（包括读写操作）之前。mode 的值用符号常量表示：_IOFBF 表示采用完全缓冲方式；_IOLBF 表示对正文文件采用行缓冲方式；_IONBF 表示采用非缓冲方式。如果 buf 不是 NULL，就用 buf 表示的字符数组做为流缓冲区。如果 buf 是 NULL 就另行分配缓冲区。参数 size 规定缓冲区的大小。函数执行出错时返回非 0 值。

现在解释一下各种缓冲方式的意义。1）完全缓冲方式。对于输入而言，在填充时用来自文件的信息填满缓冲区，直至缓冲区信息用完再做下一次填充。对输出而言，直到缓冲区装满才执行一次实际向文件的输出操作。2）行式缓冲。对于输入，在填充时一次装入一行

字符；输出时，一旦缓冲区里有了一个完整的行就送入文件。

2）缓冲设置函数 setbuf。函数原型为：

```
void setbuf(FILE *stream, char *buf);
```

本函数可以看着 setvbuf 的简化版本。如果 buf 为 NULL 就关闭流 stream 的缓冲区，在这个流中采用非缓冲方式执行输入输出。如果 buf 不是 NULL，函数相当于：

```
setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

在一些程序里需要控制程序的输出输入方式，此时就可能用到这两个函数。如果要做更仔细的控制，就应该用 setvbuf。

11.8.4 文件定位及定位函数

文件可以看作是字节的序列（无论每个字节具体是什么），这些字节从文件头开始顺序排列，每个字节在序列中都有一个特定的位置。在操作一个文件的过程中，我们可以认为，存在着一个表明该文件的当前处理位置的指示器。当文件以读或写的方式打开时，这个指示器被设于文件的起始位置；当文件以附加方式打开时，指示器一开始就被设置在文件末尾。在程序处理文件的过程中，随着读写操作的不断进行，文件指示器也顺序向后移动，当然，无论何时，它总指着随后的读写操作应该进行的位置。在一般情况下，这种指示器移动是随着读写操作的进行而自动发生的。

为使文件的使用更加灵活方便，标准库里还提供了几个直接对文件指示器进行操作的函数，这就是现在要介绍的文件定位函数。这些函数包括：

1）文件指示器复位函数 rewind。函数原型为：

```
void rewind(FILE *stream)
```

其作用是将流 stream 的文件指示器重新设置到文件的起始位置。

2）文件指示器位置检查函数 ftell。函数原型为：

```
long ftell(FILE *stream)
```

返回流 stream 当时的指示器值（一个长整数值）。出错时返回值为-1L。

3）文件指示器位置设置函数 ftell。函数原型为：

```
int fseek(FILE *stream, long offset, int origin)
```

设置 stream 的文件指示器值，使此后的输入或输出操作由这个位置开始进行。函数的参数 origin 指定指示器定位的基准点，其可能取值如下：

SEEK_SET	以文件头作为定位的基准点
SEEK_CUR	以文件指示器的当前位置作为基准点
SEEK_END	以文件尾作为基准点

对于正文流使用操作 fseek 有很多限制。标准库要求，这时参数 offset 的值或者为 0（用于将文件指示器移动文件的头或尾）；或者是以前由函数 ftell 调用得到并保存下来的值。还要求 origin 的值必须为 SEEK_SET，也就是说，采用 fseek 定位必须以文件头为基准点。从这些规定可以看出，要在程序里执行对于正文流的定位操作，只能用于把指示器移到文件头或者文件尾，或者移到前面通过 ftell 确定的某个位置。

对于二进制文件而言，函数 fseek 的参数值就没有任何限制。移动目标就是文件里以 origin 为基准点，具有偏移量 offset 的那个位置。

我们在前面介绍函数打开模式时曾经提到，对那些可读可写的流，在读操作和写操作切换之间应当重新做一次文件定位，所说的就是需要执行本小节介绍的这些操作。

标准库还允许我们把当前文件位置保存在一种特殊数据结构 fpos_t 里，并允许用保存在这种结构里的信息重新设置文件的当前访问位置。标准库为此提供了两个函数：

```
int fgetpos(FILE *stream, fpos_t *pos);
```

```
int fsetpos(FILE *stream, fpos_t *pos);
```

函数 fgetpos 将当前文件访问位置的信息记入由指针 pos 所指的变量（应是 fpos_t 类型的变量），以便将来通过 fgetpos 函数使用。当执行中出错时返回非 0 值。fsetpos 由指针 pos 所指变量取出有关文件位置记录，并用它设置流 stream。

类型 fpos_t 是标准库定义的，专门为了支持上述操作的功能。其具体定义我们不必关心，更不要自己去修改这种类型的变量的内容，只应该通过上面两个函数使用它们。

11.8.5 其他有关函数

标准库还提供了一个文件结束判断函数 feof。其原型是：

```
int feof(FILE *stream)
```

本函数判断流 stream 当时是否已到达文件末尾。这个函数主要用于二进制文件，在遇到文件结束时函数返回非 0 值，否则返回 0 值。

程序执行中出现错误也是常见的现象。对于这类情况，标准库提供了一些机制供人们编程中使用。系统中有可能存在着一集状态指示变量。如果输入输出函数在执行中出了错，它们就会去设置系统内部的出错状态。在标准头文件<errno.h>中定义了一个可以求出错误编号的 errno 表达式，该表达式的值为 0 表示没有错误，其余的值表示了程序能检查的各种错误情况。每个 C 语言系统都根据需要定义了自己的一组错误编号。

除了输入输出函数外，还有一些标准库函数的执行也可能设置错误编号值。例如许多数学函数在参数不符合要求时也会出错并设置错误编号。究竟一个具体系统检查哪些错误，如何设置错误编号等，需要查阅该系统的手册或联机帮助信息。此外，系统通常还在有关头文件里定义了相应的错误信息串，可以在需要时输出，供人参考。

标准库定义的与错误处理有关的函数包括：

1）错误标志复位（清除）函数，其原型是

```
void clearerr(FILE *stream)
```

这个函数清除与流 stream 相关联的错误标志和文件结束标志。

正文文件与二进制文件

一个文件的内容是就一个字节序列，每个字节中保存的都是一个二进制编码。就此而言，并没有什么“正文文件”和“二进制文件”之分。

但是有一类常见文件，其中字节都是下述字符的编码：可打印字符（字母、数字、标点符号），空白字符（空格、制表符、换行符）。文件的最后一个字符是表示文件结束的控制字符。这类文件就被称为“正文文件”，所有针对正文文件的操作（如 getc，fprintf，fgets 等）都是针对这种文件定义的，它们或者从这种文件读入，或者生成这种文件的内容。标准输入和标准输出文件也正好属于这类“文件”。对这类文件的使用都应该按照字符方式进行，应该通过正文流连接它们（打开文件）。

人们一般将除“正文文件”之外的文件称为“二进制文件”。一个一般二进制文件也是一个有确定长度（即文件长度或者文件大小）的字节序列，但其中的每个字节都可以保存任何编码值。这里有一个特别值得提出的情况：二进制文件里的任何一个字节中都可能保存正好被用于作为“（正文）文件结束”标志的那个字符编码值。如果我们以正文流的方式打开和处理一般二进制文件，例如做前面最简单的文件复制，就可能出现实际上并没有处理完整个文件而处理却结束了的情况。此时一定是输入函数遇到了一个表示“文件结束”的字符，因此它就认为文件结束了。

为了在处理二进制文件时正确判断文件结束，通过输入函数的返回值就不行了。标准库为此特别提供了检查文件指示器是否达到文件末尾的函数 feof，供程序里检查文件结束使用，特别是用于检查通过二进制关联的一般二进制文件的结束。

2）文件操作错误检查函数

```
int ferror(FILE *stream)
```

在输入输出操作出错时，与操作的流相关的内部状态变量（出错标志变量）被自动设置。函数 ferror 可以检查与某个流相关联的出错标志是否被设置。在流 stream 出错而设置了相关状态变量时 ferror 函数返回非 0 值。

3）打印当时错误信息的函数

```
void perror(char *s)
```

这个函数打印错误信息。它检查当时记录的错误编号（该编号反映的是这次 perror 调用之前最近发生的错误），把相应的信息送到流 stderr。信息输出的形式是：首先输出字符串 s，然后是一个冒号，随后是错误信息字符串和一个换行符。

11.8.6 采用 va_list 参数的输出函数

标准库还提供了三个函数，它们分别具有与 printf、fprintf 和 sprintf 相同的功能，但它们都不是变参数函数，每个函数的参数数目都是固定的，其中都有一个 va_list 参数。下面分别介绍这几个函数。

1）函数 vprintf，其原型是：

```
int vprintf(const char* format, va_list args);
```

这个函数的功能等价于 printf，但用 va_list 参数代替了那里的变长实参表。给它的对应于 args 的实参应该已经用 va_start 初始化；在 vprintf 里也不做 va_end。这意味着调用它的函数需要定义一个 va_list 类型的变量，经过初始化后才能送给 vprintf，在 vprintf 结束后还需要自己做结束处理（用 va_end）。

函数正常完成时返回输出的字符数，输出出错时返回 EOF 值。

2）函数 vfprintf，其原型是：

```
int vfprintf(FILE *f, const char* format, va_list args);
```

这个函数的功能等价于 fprintf，但用 va_list 参数代替了那里的变长实参表。函数正常完成时返回输出的字符数，输出出错时返回 EOF 值。

3）函数 vsprintf，其原型是：

```
int vsprintf(char *s, const char* format, va_list args);
```

这个函数的功能等价于 sprintf，但用 va_list 参数代替了那里的变长实参表。函数正常完成时返回输出的字符数，输出出错时返回 EOF 值。

程序实例

假定程序里希望定义一个出错报告函数，在运行中发现错误时，该函数首先向 stderr 流输出一个字符串，而后以格式化方式输出一系列“参数”的值。此时希望能采用 printf 格式串的形式描述输出信息的转换方式，这样，这个函数的原型可以设计为：

```
void eprintf(char *s, char *format, ...);
```

这种设计使得 eprintf 可以格式化输出任意多的“参数”。下面是这一函数的可能使用：

```
eprintf("Can't open file", "%s", fname); /* 假定fname是文件名字符串 */
eprintf("Func abc has args", "a: %f, b: %d, c: %s", a, b, c);
```

问题是 eprintf 的实现，我们当然不希望去重新实现一次 printf 的格式化功能。利用 vfprintf 就很容易地完成这一函数，关键是将一个 va_list 参数传递给 vfprintf。下面是函数 eprintf 的实现：

```
void eprintf(char *s, char *format, ...) {
    va_list as;
    va_start(as, format);
```

```
        fprintf(stderr, "Error Report, %s: ", s);
        vfprintf(stderr, format, as);
        fputc('\n', stderr);
        va_end(as);
    }
```

11.9 非局部控制转移（<setjmp.h>）

C 程序执行中的正常控制流程很容易理解：程序执行总从函数 main 开始；当一个函数被调用时，它的体开始执行；当执行达到函数体里的某个 return 语句时，该函数的执行结束，控制随之返回调用它的地方，从那里继续执行下去。函数体里可以有各种控制结构和控制语句，它们的执行将导致控制流在函数体内的不同部分之间转移，但无论怎样转移，控制都不会离开这个函数，直到遇到一个 return 语句。在整个过程中，函数的调用和退出是非常有规律的：被调用的函数将以相反的顺序退出。各个函数的退出完全由它自己的语句确定，只有执行达到了 return 语句，才出现函数执行结束的情况。

当然这里也有例外。例如，当程序中执行标准库函数 exit 或 abort 时，这一程序就直接结束，无论当时有多少层函数调用，这些函数也都立刻结束。但这毕竟只是在程序结束时才会出现，在正常情况下函数总是顺序地一个个退出。

实际中也确实存在一些情况，其中我们可能希望程序能突破正常流程的执行方式，一个常见问题就是从许多层的函数调用里直接退出来。

假设在程序中有对某函数 f 的调用，而 f 执行中引起复杂的一层层的函数调用。在一般情况下，所有这些调用都按正常规则执行，直到那些调用都结束后，函数 f 的执行才可能结束。但是也可能出现一种情况：在某个深层调用的函数里遇到特殊的问题，这时继续执行 f 以及由它调用的功能已经没有意义了，也就是说，此时实际上应该立刻结束函数 f 的执行，返回到对函数 f 的调用点。这也是在实际中常有的情况。

没有特殊机制也可以解决这个问题。最容易想到的方法是增加一个全局的标志变量，在遇到特殊情况时就给这个变量设一个特殊值。我们让函数 f 和被 f 调用的函数都不断检查这个变量，一旦发现它具有这个特殊值时就退出。这样做虽然可以解决问题，但是非常麻烦。反复检查也使程序变得更繁琐，使程序的意义更难理解，也破坏程序的可读性和易维护性。

利用标准库提供的功能可以比较方便地解决这类问题。在标准头文件<setjmp.h>里定义了类型 jmp_buf 和两个函数（或者宏）setjmp 和 longjmp。下面简单介绍它们的功能，并用小例子说明其使用方式。

类型 jmp_buf 的变量用来保存程序执行现场的有关信息，这种变量可以称作“环境变量”或者“现场变量”。函数（宏）setjmp 的执行将把当时的执行现场信息保存到作为参数的那个 jmp_buf 类型的变量里。在此之后，如果我们用这个变量调用函数 longjmp，就会导致原来保存在这个变量里的执行现场被恢复，使程序可以返回到这一原来保存的现场，换一种方式，从这里重新执行下去。setjmp 的“类型特征”是：

```
int setjmp(jmp_buf env)
```

在调用 setjmp 时，应该用一个全局的 jmp_buf 变量，这样才能方便地在另一个函数里访问这个变量，恢复保存在这个变量里的现场。longjmp 的“类型特征”是：

```
void longjmp(jmp_buf env, int val)
```

其作用就是恢复由参数 env 保存的现场。

对 setjmp 的调用通常出现在一个使用“逻辑值”的上下文里，例如在条件语句或者循环语句里面。下面是一种典型的调用方式：

```
if (setjmp(env) == 0)
    ... /*这里写执行setjmp之后需要做的正常工作，包括进一步的函数调用*/
else
```

```
        ... /*这里写由于longjmp调用而返回后应当做的事情*/
```

在这个程序片段被执行时，对函数 setjmp 的调用除了将执行环境的现场保存在 env 里之外，还立刻返回 0 值，这就使随后的有关程序片段被执行。如果在这个执行阶段中出现了对 longjmp（用同一个 jmp_buf 变量 env）的调用，就会导致上面保存的执行现场得以恢复，程序将再次从 setjmp 处返回，不过这一次函数的返回值不是 0 值。从而就使上面程序片段中 else 后面的程序段被执行。

程序里调用 longjmp 时，处理需要指定一个环境变量（jmp_buf 类型的变量）参数外，还应给定一个整型的 val 参数值。这个调用将导致程序的执行转移到最近的那个（对同一个环境变量的）setjmp 调用所保存的现场处，并使该函数以非 0 值返回。根据这个值，程序就能进入另一条执行路径。如果在调用 longjmp 时提供的 val 值不是 0，这个值将被直接作为 setjmp 的返回值。如果提供的 val 值是 0，那么系统自动产生一个非 0 值（许多系统里采用 1 作为这个值）。

在执行 longjmp 时，一个必要条件是相应的 setjmp 调用所在的函数还没有结束。对于上面例子，如果这个 setjmp 调用出现在函数 f 体里，在 f 的执行中调用了 setjmp，那么与之相对应的 longjmp 调用只能出现在函数 f 的内部，或者出现在那些被 f 直接或间接调用的函数里，否则就是错误的。当程序的执行通过调用 longjmp 回到函数 f 后，在 f 里可以用的变量的值情况分为两类：对于函数 f 里那些在执行 setjmp 后修改过的局部自动变量，它们的值现在已经没有定义了（因此就不该再用了）；其他变量（如外部变量等）保持为 longjmp 调用时的状态，它们的值仍然可以使用。

要想准确理解这种非正常执行控制转移的意义，需要有对程序中复杂控制转移实际需要的了解，并需要对 C 语言的基本实现方式和有关操作的实现方式有一些了解。这方面的问题已经超出了本书的讨论范围。

11.10 调试断言和异常处理（<assert.h>和<signal.h>）

头文件<assert.h>里定义了一个函数（通常定义为一个宏）：

```
void assert(int n)
```

对这个函数的调用可以出现在程序里任何地方。在程序执行遇到对 assert 的调用，当实际参数表达式求出的值是 0 时，程序将输出下面形式的一行信息：

```
Assertion failed: 表达式, file 文件名, line 行号
```

输出之后调用标准函数 abort，结束程序的执行。这里的文件名和行号自动根据系统预定义的宏名字得到。

函数（宏）assert 通常在程序调试时使用。我们在写程序时，可以在源程序里的一些关键位置插入对函数 assert 的调用。例如，可以用这种方式检查程序运行中某个变量的值是否满足要求，某些变量的值之间是否满足某种关系，等等。合理地使用这个函数可以帮助我们发现程序在调试运行中出现的错误。

在源程序里，如果在对头文件<assert.h>的包含命令之前定义符号常量 NDEBUG，就可以关闭程序中所有由 assert 引起的检查和信息输出。这一功能可以使我们在需要关闭所有检查时不必去修改程序的内容。

在一个程序的执行过程中，有时可能出现一些动态发生的异常情况，例如出现数值计算的结果越界的情况，或者出现除数为零等。在出现某些问题时，计算机硬件可能自动产生有关的中断信号，要求程序响应；另一方面，执行的程序本身也可能主动发出有关异常情况的信号，要求某种特殊处理。

标准头文件<signal.h>提供了引发异常和处理异常的有关机制，其中主要是提供了两

个函数。函数 raise 的执行将发出一个异常信息，利用函数 signal 可以定义与各个异常相关联的处理程序（一般把它们称作“异常处理器”）。

函数 raise 的类型特征是：

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

这个函数的作用是产生一个异常信号，送给程序要求处理。如果该异常被成功地处理，函数返回 0 值，否则返回非 0 的值。通过检查这个值，就可以了解有关处理的情况。

系统提供了两个内部定义的异常处理器，它们分别用符号常量 SIG_DFL 和 SIG_IGN 表示。如果与某个异常信号相关联的处理器是 SIG_DFL，在发生这个异常时，程序将采用某种由具体系统实现确定的方式进行处理。如果与某个异常关联的处理器是 SIG_IGN，执行中发生的这种异常将被忽略。如果与某个异常相关联的处理器不是这两种情况，发生异常时有关处理器（函数）将被调用。

函数 signal 的类型很复杂，要读懂它，需要对 C 语言复杂类型描述有准确的理解。下面是函数 signal 的类型：

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

这个原型说明了 signal 是个函数，它有两个参数：一个是整型参数 sig，另一个是函数指针参数 handler。指针参数 handler 指向的是有一个整型参数且没有返回值的函数。函数 signal 的返回值是函数指针，其类型与参数 handler 的类型一样，指向具有一个整型参数，没有返回值的函数。由此可知，异常处理器函数的类型特征应当是：

```
void 函数名(int)
```

在调用函数 signal 时，应当给它提供一个整型参数（表示一个异常类）和一个函数指针参数（表示一个异常处理器函数）。signal 的执行效果就是将这个处理器函数与指定的异常类关联起来，它的返回值是指向原来与这个异常关联的那个处理器（函数）的指针。对各种异常信号，有关处理器的初始设置由具体 C 系统确定。

合法的异常信号包括：

SIGABRT	程序非正常结束，例如执行了 abort
SIGFPE	浮点数算术出错，例如溢出或者除零
SIGILL	非法的函数映象，例如遇到了非法指令
SIGINT	交互请求，例如出现中断信号
SIGSEGV	非法的存储区访问，例如访问超出了规定范围
SIGTERM	给程序送一个终止信号

当某个异常信号发生时，signal 恢复到初始状态，相应的异常处理器随之被用发生的异常信号调用，就像是执行了函数调用(*handler)(sig)。如果该处理器函数返回，程序将从发生异常的位置继续执行下去。

11.11 标准库的其他功能

这里简单介绍与本地化和多字节字符有关的概念，以及标准库为此提供的功能。并不对这些功能及其应用做更多讨论，有关细节请查阅 C 语言标准手册和其他相关书籍。

11.11.1 本地化

一个具体的程序可能在某个国家使用，而本地的文化传统和习惯就可能对程序的某些行为提出特殊要求。如果一个程序希望能在不同地域使用，我们就可能希望它的行为方式尽可能符合本地的文化习惯。标准库头文件<locale.h>就是为此目的设计的，这个文件里定义了一个类型和两个函数，还定义了一些宏。

这里定义的类型是 struct lconv（locale conventions，本地文化习惯），其中至少应包含下述成员：

类型和成员名	默认值	解释
char *decimal_point;	"."	格式化非金融量时所用的小数点
char *thousands_sep;	""	非金融量中的数位间隔符（默认为空串，下同）
char *grouping;	""	说明非金融量中数位间隔方式
char *int_curr_symbol;	""	本地所用的国际标准货币符号
char *current_symbol;	""	本地货币符号
char *mon_decimal_point;	""	金融量所用的小数点
char *mon_thousants_sep;	""	金融量所用的数位间隔符
char *mon_grouping;	""	说明金融量所用的数位间隔方式
char *positive_sign;	""	说明非负金融量格式化方式的串
char *negative_sign;	""	说明负金融量格式化方式的串
char int_frac_digits;	CHAR_MAX	按国际标准显示金融量时的小数位数
char frac_digits;	CHAR_MAX	按本地方式显示金融量时的小数位数
char p_cs_precedes;	CHAR_MAX	说明货币符号放在非负数值之前（1）或后（0）
char p_sep_by_space;	CHAR_MAX	货币符号与非负数值之前间有（1）无（0）空格
char n_cs_precedes;	CHAR_MAX	说明货币符号放在负数值之前（1）或后（0）
char n_sep_by_space;	CHAR_MAX	货币符号与负数值之前间有（1）无（0）空格
char p_sign_posn;	CHAR_MAX	说明非负金融量中 positive_sign 的位置
char n_sugn_posn;	CHAR_MAX	说明负金融量中 negative_sign 的位置

其中默认值用空串或者 CHAR_MAX 表示没有相关信息。

用函数 localeconv 填充一个 struct lconv 结构，其中填入当前本地化的信息：

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

从这个结构中可以找到上述成员的信息。

这里定义了宏：LC_ALL，LC_COLLATE，LC_CTYPE，LC_MONETARY，LC_NUMERIC，LC_TIME。这些宏展开后的值可用作 setlocale 函数的第一个参数。该函数原型为：

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

用于选择程序中本地化描述 locale 中由 category 确定的部分。可用于提取本地化信息，或者修改本地化信息的指定属性。以 NULL 作为 locale 参数时函数 setlocale 返回当时本地化描述的指定属性。非 NULL 的 locale 参数指定新的本地化设置。

上述各个宏分别表示本函数所涉及的范围。LC_ALL 表示函数操作针对整个本地化描述，LC_COLLATE 涉及与本地化有关的字符串函数 strcoll 和 strxrfm，LC_CTYPE 影响所有字符函数（<ctype.h>里定义）的行为方式，LC_MONETARY 影响各种金融格式化形式，LC_NUMERIC 影响数值格式化形式（包括小数点和数位间隔），LC_TIME 影响有关时间显示的字符串函数 strftime 和 strfxtime。

11.11.2 多字节字符

C 语言基本的 char 类型是针对类似英语的拼音文字设计的，这些语言的基本字符集很小，用一个字节就可以区分不同字符。然而，由于文化差异，一些国家民族所用的基本字符集很多，典型的就是中文。将每个中文字看作一个字符，中文字符集将包含数千或数万字符。用一个字节无法区分这些字符。为此，C 语言通过标准库提供了多字节处理能力。

标准库为处理多字节字符多字节字符类型 wchar_t，还提供了几个特殊函数，包括在表示多字节字符的字符串与 wchar_t 类型的值之间的转换，在表示一系列多字节字符的字节串与 wchar_t 串形式之间的转换函数等。

wchar_t 类型在<stddef.h>里定义，这是一种整型，其值的范围足以表示相应本地环境（参看下面有关本地化的讨论）所规定的扩展字符集中的所有成员（“字符”）。

标准库还定义了几个多字节字符处理函数和多字节字符串处理函数。这些函数的行为都受本地化环境宏 LC_CTYPE 的影响。多字节字符处理函数包括 mblen，mbtowc 和 wctomb，多字节字符串处理函数包括 mbstowcs 和 wcstombs。几个包含 to 的函数都是字符表示或者字符串转换函数，完成多字节字符（串）与普通字符（串）之间的转换。

有关细节这里不介绍了。

本章讨论的重要概念

文件，流，打开文件，关闭文件，正文流（字符流），二进制流，FILE 类型，文件指针，缓冲式输入输出，缓冲区，透明性，字符输入输出，格式化输入输出，行式输入输出，缓冲区刷新，文件定位，文件指示器，直接输入输出，输入输出格式控制，标准库，执行控制，语言实现的规定，调试断言，变参数函数，执行现场，从深层调用中退出，程序异常及处理

练习

1. 请写出两个函数，它们实现类似 strspn 和 strcspn 的功能，确定字符串里满足一定条件的最长前缀的长度，但判断的依据通过一个函数指针参数得到。该指针的实参应是以 char 为参数返回 int 值的函数（谓词），在 char 满足要求时返回非 0 值，不满足要求是返回 0 值。这种函数与标准库 ctype.h 里定义的字符分类函数类似。（注意，标准库 ctype.h 里的“字符分类函数”通常是用宏定义实现的，因此不能用于作为函数指针参数的实参。）
2. 自己实现标准库函数 strtok 的功能。请给函数另起一个名字，以免与库函数冲突。（提示：请考虑使用静态局部变量）。
3. 写一个函数，它有一个双精度数组参数和不定个数的整数参数，它能够根据所给的一组整参数值（作为数组元素下标）由数组里取出对应的元素，求出这些元素的和。
4. 请设法只使用标准库函数 getchar 实现 scanf 的功能。
5. 请设法只使用标准库函数 putchar 实现 printf 的功能。
6. 试写一个 C 源程序的预处理程序。先完成处理#include 命令（如果考虑系统头文件，那么就需要其他信息。可以只考虑当前目录下的文件包含问题）的工作，然后考虑简单宏命令的处理和条件编译的处理，最后考虑带参数宏定义的处理。