第9章

字符串、字符和字节

字符串是一种重要的数据类型，但是C语言并没有显式的字符串数据类型，因为字符串以字符串常量的形式出现或者存储于字符数组中。字符串常量很适用于那些程序不会对它们进行修改的字符串。所有其他字符串都必须存储于字符数组或动态分配的内存中（见第11章）。本章描述处理字符串和字符的库函数，以及一组相关的，具有类似能力的，既可以处理字符串也可以处理非字符串数据的函数。

9.1 字符串基础

首先，让我们回顾一下字符串的基础知识。字符串就是一串零个或多个字符，并且以一个位模式为全0的NUL字节结尾。因此，字符串所包含的字符内部不能出现NUL字节。这个限制很少会引起问题，因为NUL字节并不存在与它相关联的可打印字符，这也是它被选为终止符的原因。NUL字节是字符串的终止符，但它本身并不是字符串的一部分，所以字符串的长度并不包括NUL字节。

头文件 string.h 包含了使用字符串函数所需的原型和声明。尽管并非必需，但在程序中包含这个头文件确实是个好主意，因为有了它所包含的原型，编译器可以更好地为你的程序执行错误检查[1]。

9.2 字符串长度

字符串的长度就是它所包含的字符个数。我们很容易通过对字符进行计数来计算字符串的长度，程序9.1就是这样做的。这种实现方法说明了处理字符串所使用的处理过程的类型。但是，事实上你极少需要编写字符串函数，因为标准库所提供的函数通常能完成这些任务。不过，如果你还是希望自己编写一个字符串函数，请注意标准保留了所有以str开头的函数名，用于标准库将来的扩展。

库函数 strlen 的原型如下：

```
size_t  strlen( char const *string );
```

[1] 老的C程序常常不包含这个文件。没有函数原型，只有每个函数的返回类型才能被声明，而这些函数中的绝大多数都会忽略返回值。

警告：

注意 strlen 返回一个类型为 size_t 的值。这个类型是在头文件 stddef.h 中定义的，它是一个无符号整数类型。在表达式中使用无符号数可能导致不可预料的结果。例如，下面两个表达式看上去是相等的：

```
if( strlen( x ) >= strlen( y ) ) ...
if( strlen( x ) - strlen( y ) >= 0 ) ...
```

但事实上它们是不相等的。第 1 条语句将按照你预想的那样工作，但第 2 条语句的结果将永远是真。strlen 的结果是个无符号数，所以操作符>=左边的表达式也将是无符号数，而无符号数绝不可能是负的。

```
/*
** 计算字符串参数的长度。
*/
#include <stddef.h>

size_t
strlen( char const *string )
{
   int length;

   for( length = 0; *string++ != '\0'; )
      length += 1;

   return length;
}
```

程序 9.1 字符串长度 strlen.c

警告：

表达式中如果同时包含了有符号数和无符号数，可能会产生奇怪的结果。和前一对语句一样，下面两条语句并不相等，其原因相同。

```
if( strlen( x ) >= 10 ) ...
if( strlen( x ) - 10 >= 0 ) ...
```

如果把 strlen 的返回值强制转换为 int，就可以消除这个问题。

提示：

你很可能想自行编写 strlen 函数，灵活运用 register 声明和一些聪明的技巧使它比库函数版本效率更高。这的确是个诱惑，但事实上很少能够如愿。标准库函数有时是用汇编语言实现的，目的就是为了充分利用某些机器所提供的特殊的字符串操纵指令，从而追求最大限度的速度。即使在没有这类特殊指令的机器上，你最好还是把更多的时间花在程序其他部分的算法改进上。寻找一种更好的算法比改良一种差劲的算法更有效率，复用已经存在的软件比重新开发一个效率更高。

9.3　不受限制的字符串函数

最常用的字符串函数都是“不受限制”的，就是说它们只是通过寻找字符串参数结尾的 NUL 字节来判断它的长度。这些函数一般都指定一块内存用于存放结果字符串。在使用这些函数时，程序员必须保证结果字符串不会溢出这块内存。在本节具体讨论每个函数时，我将对这个问题作更详细的讨论。

9.3.1　复制字符串

用于复制字符串的函数是 strcpy，它的原型如下所示：

```
char    *strcpy( char *dst, char const *src );
```

这个函数把参数 src 字符串复制到 dst 参数。如果参数 src 和 dst 在内存中出现重叠，其结果是未定义的。由于 dst 参数将进行修改，所以它必须是个字符数组或者是一个指向动态分配内存的数组的指针，不能使用字符串常量。这个函数的返回值将在 9.3.3 小节描述。

目标参数的以前内容将被覆盖并丢失。即使新的字符串比 dst 原先的内存更短，由于新字符串是以 NUL 字节结尾，所以老字符串最后剩余的几个字符也会被有效地删除。

考虑下面这个例子：

```
char    message[] = "Original message";
...
if( ... )
        strcpy( message, "Different" );
```

如果条件为真并且复制顺利执行，数组将包含下面的内容：

'D'	'i'	'f'	'f'	'e'	'r'	'e'	'n'	't'	0	'e'	's'	's'	'a'	'g'	'e'	0

第 1 个 NUL 字节后面的几个字符再也无法被字符串函数访问，因此从任何现实的角度看，它们都已经是丢失的了。

警告：

程序员必须保证目标字符数组的空间足以容纳需要复制的字符串。如果字符串比数组长，多余的字符仍被复制，它们将覆盖原先存储于数组后面的内存空间的值。strcpy 无法解决这个问题，因为它无法判断目标字符数组的长度。

例如：

```
char    message[] = "Original message";
...
strcpy( message, "A different message" );
```

第 2 个字符串太长了，无法容纳于 message 字符数组中。因此，strcpy 函数将侵占数组后面的部分内存空间，改写原先恰好存储在那里的变量。如果你在使用这个函数前确保目标参数足以容纳源字符串，就可以避免大量的调试工作。

9.3.2 连接字符串

要想把一个字符串添加（连接）到另一个字符串的后面，你可以使用 strcat 函数。它的原型如下：

```
char    *strcat( char *dst, char const *src );
```

strcat 函数要求 dst 参数原先已经包含了一个字符串（可以是空字符串）。它找到这个字符串的末尾，并把 src 字符串的一份拷贝添加到这个位置。如果 src 和 dst 的位置发生重叠，其结果是未定义的。

下面这个例子显示了这个函数的一种常见用法。

```
strcpy( message, "Hello " );
strcat( message, customer_name );
strcat( message, ", how are you?" );
```

每个 strcat 函数的字符串参数都被添加到原先存在于 message 数组的字符串后面。其结果是下面这个字符串：

```
Hello Jim, how are you?
```

警告：

和前面一样，程序员必须保证目标字符数组剩余的空间足以保存整个源字符串。但这次并不是简单地把源字符串的长度和目标字符数组的长度进行比较，你必须考虑目标数组中原先存在的字符串。

9.3.3 函数的返回值

strcpy 和 strcat 都返回它们第 1 个参数的一份拷贝，就是一个指向目标字符数组的指针。由于它们返回这种类型的值，所以你可以嵌套地调用这些函数，如下面的例子所示：

```
strcat( strcpy( dst, a ), b );
```

strcpy 首先执行。它把字符串从 a 复制到 dst 并返回 dst。然后这个返回值成为 strcat 函数的第 1 个参数，strcat 函数把 b 添加到 dst 的后面。

这种嵌套调用的风格较之下面这种可读性更佳的风格在功能上并无优势。

```
strcpy( dst, a );
strcat( dst, b );
```

事实上，在这些函数的绝大多数调用中，它们的返回值只是被简单地忽略。

9.3.4 字符串比较

比较两个字符串涉及对两个字符串对应的字符逐个进行比较，直到发现不匹配为止。那个最先不匹配的字符中较“小”（也就是说，在字符集中的序数较小）的那个字符所在的字符串被认为“小于”另外一个字符串。如果其中一个字符串是另外一个字符串的前面一部分，那么它也被认为“小于”另外一个字符串，因为它的 NUL 结尾字节出现得更早。这种比较被称为“词典比较”，对于只包含大写字母或只包含小写字母的字符串比较，这种比较过程所给出的结果总是和我们日常所用的字母顺序的比较相同。

库函数 strcmp 用于比较两个字符串，它的原型如下：

```
int   strcmp( char const *s1, char const *s2 );
```

如果 s1 小于 s2，strcmp 函数返回一个小于零的值。如果 s1 大于 s2，函数返回一个大于零的值。如果两个字符串相等，函数就返回零。

警告：

初学者常常会编写下面这样的表达式

```
if( strcmp( a, b ) )
```

他以为如果两个字符串相等，它的结果将是真。但是，这个结果将正好相反，因为在两个字符串相等的情况下返回值是零（假）。然而，把这个返回值当作布尔值进行测试是一种坏风格，因为它具有三个截然不同的结果：小于、等于和大于。所以，更好的方法是把这个返回值与零进行比较。

警告：

注意标准并没有规定用于提示不相等的具体值。它只是说如果第 1 个字符串大于第 2 个字符串就返回一个大于零的值，如果第 1 个字符串小于第 2 个字符串就返回一个小于零的值。一个常见的错误是以为返回值是 1 和 -1，分别代表大于和小于。但这个假设并不总是正确的。

警告：

由于 strcmp 并不修改它的任何一个参数，所以不存在溢出字符数组的危险。但是，和其他不受限制的字符串函数一样，strcmp 函数的字符串参数也必须以一个 NUL 字节结尾。如果并非如此，strcmp 就可能对参数后面的字节进行比较，这个比较结果将不会有什么意义。

9.4　长度受限的字符串函数

标准库还包含了一些函数，它们以一种不同的方式处理字符串。这些函数接受一个显式的长度参数，用于限定进行复制或比较的字符数。这些函数提供了一种方便的机制，可以防止难以预料的长字符串从它们的目标数组溢出。

这些函数的原型如下所示。和它们的不受限制版本一样，如果源参数和目标参数发生重叠，strncpy 和 strncat 的结果就是未定义的。

```
char    *strncpy( char *dst, char const *src, size_t len );
char    *strncat( char *dst, char const *src, size_t len );
int     strncmp( char const *s1, char const *s2, size_t len );
```

和 strcpy 一样，strncpy 把源字符串的字符复制到目标数组。然而，它总是正好向 dst 写入 len 个字符。如果 strlen(src)的值小于 len，dst 数组就用额外的 NUL 字节填充到 len 长度。如果 strlen(src) 的值大于或等于 len，那么只有 len 个字符被复制到 dst 中。**注意！它的结果将不会以 NUL 字节结尾。**

警告：

strncpy 调用的结果可能不是一个字符串，因此字符串必须以 NUL 字节结尾。如果在一个需要字符串的地方（例如 strlen 函数的参数）使用了一个不是以 NUL 字节结尾的字符序列，会发生什么情况呢？strlen 函数将无法知道 NUL 字节是没有的，所以它将继续进行查找，一个字符接一个字符，直到它发现一个 NUL 字节为止。或许它找了几百个字符才找到，而 strlen 函数的这个返回值从本质上说是一个随机数。或者，如果函数试图访问系统分配给这个程序以外的内存范围，程序就会崩溃。

警告：

这个问题只有当你使用 strncpy 函数创建字符串，然后或者对它们使用 str 开头的库函数，或者在 printf 中使用%s 格式码打印它们时才会发生。在使用不受限制的函数之前，你首先必须确定字符串实际上是以 NUL 字节结尾的。例如，考虑下面这个代码段：

```
char    buffer[BSIZE];
...
strncpy( buffer, name, BSIZE );
buffer[BSIZE - 1] = '\0';
```

如果 name 的内容可以容纳于 buffer 中，最后那个赋值语句没有任何效果。但是，如果 name 太长，这条赋值语句可以保证 buffer 中的字符串是以 NUL 结尾的。以后对这个数组使用 strlen 或其他不受限制的字符串函数将能够正确工作。

尽管 strncat 也是一个长度受限的函数，但它和 strncpy 存在不同之外。它从 src 中最多复制 len 个字符到目标数组的后面。但是，strncat 总是在结果字符串后面添加一个 NUL 字节，而且它不会像 strncpy 那样对目标数组用 NUL 字节进行填充。注意目标数组中原先的字符串并没有算在 strncat 的长度中。strncat 最多向目标数组复制 len 个字符（再加一个结尾的 NUL 字节），它才不管目标参数除去原先存在的字符串之后留下的空间够不够。

最后，strncmp 也用于比较两个字符串，但它最多比较 len 个字节。如果两个字符串在第 len 个字符之前存在不相等的字符，这个函数就像 strcmp 一样停止比较，返回结果。如果两个字符串的前 len 个字符相等，函数就返回零。

9.5 字符串查找基础

标准库中存在许多函数，它们用各种不同的方法查找字符串。这些各种各样的工具给了 C 程序员很大的灵活性。

9.5.1 查找一个字符

在一个字符串中查找一个特定字符最容易的方法是使用 strchr 和 strrchr 函数，它们的原型如下所示：

```
char    *strchr( char const *str, int ch );
char    *strrchr( char const *str, int ch );
```

注意它们的第 2 个参数是一个整型值。但是，它包含了一个字符值。strchr 在字符串 str 中查找字符 ch 第 1 次出现的位置，找到后函数返回一个指向该位置的指针。如果该字符并不存在于字符串中，函数就返回一个 NULL 指针。strrchr 的功能和 strchr 基本一致，只是它所返回的是一个指向字符串中该字符最后一次出现的位置（最右边那个）。

这里有个例子：

```
char    string[20] = "Hello there, honey.";
char    *ans;

ans = strchr( string, 'h' );
```

ans 所指向的位置将是 string+7，因为第 1 个'h'出现在这个位置。注意这里大小写是有区别的。

9.5.2　查找任何几个字符

strpbrk 是个更为常见的函数。它并不是查找某个特定的字符，而是查找任何一组字符第 1 次在字符串中出现的位置。它的原型如下：

```
char    *strpbrk( char const *str, char const *group );
```

这个函数返回一个指向 str 中第 1 个匹配 group 中任何一个字符的字符位置。如果未找到匹配，函数返回一个 NULL 指针。

在下面的代码段中，

```
char    string[20] = "Hello there, honey.";
char    *ans;

ans = strpbrk( string, "aeiou" );
```

ans 所指向的位置是 string+1，因为这个位置是第 2 个参数中的字符第 1 次出现的位置。和前面一样，这个函数也是区分大小写的。

9.5.3　查找一个子串

为了在字符串中查找一个子串，我们可以使用 strstr 函数，它的原型如下：

```
char *strstr( char const *s1, char const *s2 );
```

这个函数在 s1 中查找整个 s2 第 1 次出现的起始位置，并返回一个指向该位置的指针。如果 s2 并没有完整地出现在 s1 的任何地方，函数将返回一个 NULL 指针。如果第 2 个参数是一个空字符串，函数就返回 s1。

标准库中并不存在 strrstr 或 strrpbrk 函数。不过，如果你需要它们，它们是很容易实现的。程序 9.2 显示了一种实现 strrstr 的方法。这个技巧同样也可以用于实现 strrpbrk。

```
/*
** 在字符串 s1 中查找字符串 s2 最右出现的位置，并返回一个指向该位置的指针。
*/
#include <string.h>

char*
my_strrstr( char const *s1, char const *s2 )
{
      register char*last;
      register char*current;

/*
** 把指针初始化为我们已经找到的前一次匹配位置。
*/
last = NULL;

/*
**只在第 2 个字符串不为空时才进行查找，如果 S2 为空，返回 NULL。
*/
if( *s2 != '\0' ){
     /*
     ** 查找 s2 在 s1 中第 1 次出现的位置。
     */
     current = strstr( s1, s2 );

/*
** 我们每次找到字符串时，让指针指向它的起始位置。然后查找该字符串下一个匹配位置。
*/
```

```
            while( current != NULL ){
                last = current;
                current = strstr( last + 1, s2 );
            }
        }

        /* 返回指向我们找到的最后一次匹配的起始位置的指针。*/
        return last;
    }
```

程序 9.2 查找子串最右一次出现的位置 mstrrstr.c

9.6 高级字符串查找

接下来的一组函数简化了从一个字符串中查找和抽取一个子串的过程。

9.6.1 查找一个字符串前缀

strspn 和 strcspn 函数用于在字符串的起始位置对字符计数。它们的原型如下所示：

```
size_t  strspn( char const *str, char const *group );
size_t  strcspn( char cosnt *str, char const *group );
```

group 字符串指定一个或多个字符。strspn 返回 str 起始部分匹配 group 中任意字符的字符数。例如，如果 group 包含了空格、制表符等空白字符，那么这个函数将返回 str 起始部分空白字符的数目。str 的下一个字符就是它的第 1 个非空白字符。

考虑下面这个例子：

```
int     len1, len2;
char    buffer[] = "25,142,330,Smith,J,239-4123";

len1 = strspn( buffer, "0123456789" );
len2 = strspn( buffer, ",0123456789" );
```

当然，buffer 缓冲区在正常情况下是不会用这个方法进行初始化的。它将会包含在运行时读取的数据。但是在 buffer 中有了这个值之后，变量 len1 将被设置为 2，变量 len2 将被设置为 11。下面的代码将计算一个指向字符串中第 1 个非空白字符的指针。

```
ptr = buffer + strspn( buffer, "\n\r\f\t\v" );
```

strcspn 函数和 strspn 函数正好相反，它对 str 字符串起始部分中不与 group 中任何字符匹配的字符进行计数。strcspn 这个名字中字母 c 来源于对一组字符求补这个概念，也就是把这些字符换成原先并不存在的字符。如果你使用“ \n\r\f\t\v”作为 group 参数，这个函数将返回第 1 个参数字符串起始部分所有非空白字符的值。

9.6.2 查找标记

一个字符串常常包含几个单独的部分，它们彼此被分隔开来。每次为了处理这些部分，你首先必须把它们从字符串中抽取出来。

这个任务正是 strtok 函数所实现的功能。它从字符串中隔离各个单独的称为标记(token)的部分，并丢弃分隔符。它的原型如下：

```
char    *strtok( char *str, char const *sep );
```

sep 参数是个字符串，定义了用作分隔符的字符集合。第 1 参数指定一个字符串，它包含零个或多个由 sep 字符串中一个或多个分隔符分隔的标记。strtok 找到 str 的下一个标记，并将其用 NUL 结尾，然后返回一个指向这个标记的指针。

警告：

当 strtok 函数执行任务时，它将会修改它所处理的字符串。如果源字符串不能被修改，那就复制一份，将这份拷贝传递给 strtok 函数。

如果 strtok 函数的第 1 个参数不是 NULL，函数将找到字符串的第 1 个标记。strtok 同时将保存它在字符串中的位置。如果 strtok 函数的第 1 个参数是 NULL，函数就在同一个字符串中从这个被保存的位置开始像前面一样查找下一个标记。如果字符串内不存在更多的标记，strtok 函数就返回一个 NULL 指针。在典型情况下，在第 1 次调用 strtok 时，向它传递一个指向字符串的指针。然后，这个函数被重复调用（第 1 个参数为 NULL），直到它返回 NULL 为止。

程序 9.3 是一个简短的例子。这个函数从它的参数中提取标记并把它们打印出来（一行一个）。这些标记用空白分隔。不要被 for 语句的外观所混淆。它之所以被分成 3 行是因为它实在太长了。

```
/*
** 从一个字符数组中提取空白字符分隔的标记并把它们打印出来（每行一个）。
*/
#include <stdio.h>
#include <string.h>

void
print_tokens( char *line )
{
        static char whitespace[] = " \t\f\r\v\n";
        char  *token;

        for( token = strtok( line, whitespace );
           token != NULL;
           token = strtok( NULL, whitespace ) )
            printf( "Next token is %s\n", token );
}
```

程序 9.3 提取标记　　token.c

如果你愿意，你可以在每次调用 strtok 函数时使用不同的分隔符集合。当一个字符串的不同部分由不同的字符集合分隔的时候，这个技巧很管用。

警告：

由于 strtok 函数保存它所处理的函数的局部状态信息，所以你不能用它同时解析两个字符串。因此，如果 for 循环的循环体内调用了一个在内部调用 strtok 函数的函数，程序 9.3 将会失败。

9.7 错误信息

当你调用一些函数，请求操作系统执行一些功能如打开文件时，如果出现错误，操作系统是通过设置一个外部的整型变量 errno 进行错误代码报告的。strerror 函数把其中一个错误代码作为参数并返回一个指向用于描述错误的字符串的指针。这个函数的原型如下：

```
char  *strerror( int error_number );
```

事实上，返回值应该被声明为 const，因为你不应该修改它。

9.8 字符操作

标准库包含了两组函数，用于操作单独的字符，它们的原型位于头文件 ctype.h。第 1 组函数用于对字符分类，而第 2 组函数用于转换字符。

9.8.1 字符分类

每个分类函数接受一个包含字符值的整型参数。函数测试这个字符并返回一个整型值，表示真或假[1]。表 9.1 列出了这些分类函数以及它们每个所执行的测试。

表 9.1　　字符分类函数

函　数	如果它的参数符合下列条件就返回真
iscntrl	任何控制字符
isspace	空白字符：空格' ', 换页 '\f', 换行 '\n', 回车 '\r', 制表符 '\t'或垂直制表符'\v'
isdigit	十进制数字 0～9
isxdigit	十六进制数字，包括所有十进制数字，小写字母 a～f，大写字母 A～F
islower	小写字母 a～z
isupper	大写字母 A～Z
isalpha	字母 a～z 或 A～Z
isalnum	字母或数字，a～z，A～Z 或 0～9
ispunct	标点符号，任何不属于数字或字母的图形字符（可打印符号）
isgraph	任何图形字符
isprint	任何可打印字符，包括图形字符和空白字符

9.8.2 字符转换

转换函数把大写字母转换为小写字母或者把小写字母转换为大写字母。

```
int  tolower( int ch );
int  toupper( int ch );
```

toupper 函数返回其参数的对应大写形式，tolower 函数返回其参数的对应小写形式。如果函数的参数并不是一个处于适当大小写状态的字符（即 toupper 的参数不是小写字母或 tolower 的参数不

[1] 注意标准并没有指定任何特定值，所以有可能返回任何非零值。

是个大写字母），函数将不修改参数直接返回。

提示：

直接测试或操纵字符将会降低程序的可移植性。例如，考虑下面这条语句，它试图测试 ch 是否是一个大写字符。

```
if( ch >= 'A' && ch <= 'Z' )
```

这条语句在使用 ASCII 字符集的机器上能够运行，但在使用 EBCDIC 字符集的机器上将会失败。另一方面，下面这条语句

```
if( isupper( ch ) )
```

无论机器使用哪个字符集，它都能顺利运行。

9.9 内存操作

根据定义，字符串由一个 NUL 字节结尾，所以字符串内部不能包含任何 NUL 字符。但是，非字符串数据内部包含零值的情况并不罕见。你无法使用字符串函数来处理这种类型的数据，因为当它们遇到第 1 个 NUL 字节时将停止工作。

不过，我们可以使用另外一组相关的函数，它们的操作与字符串函数类似，但这些函数能够处理任意的字节序列。下面是它们的原型。

```
void    *memcpy( void *dst, void const *src, size_t length );
void    *memmove( void *dst, void const *src, size_t length );
void    *memcmp( void const *a, void const *b, size_t length );
void    *memchr( void const *a, int ch, size_t length );
void    *memset( void *a, int ch, size_t length );
```

每个原型都包含一个显式的参数说明需要处理的字节数。但和 strn 带头的函数不同，它们在遇到 NUL 字节时并不会停止操作。

memcpy 从 src 的起始位置复制 length 个字节到 dst 的内存起始位置。你可以用这种方法复制任何类型的值，第 3 个参数指定复制值的长度（以字节计）。如果 src 和 dst 以任何形式出现了重叠，它的结果是未定义的。

例如：

```
char    temp[SIZE], values[SIZE];
...
memcpy( temp, values, SIZE );
```

它从数组 values 复制 SIZE 个字节到数组 temp。

但是，如果两个数组都是整型数组该怎么办呢？下面的语句可以完成这项任务：

```
memcpy( temp, values, sizeof( values ) );
```

前两个参数并不需要使用强制类型转换，因为在函数的原型中，参数的类型是 void*型指针，而任何类型的指针都可以转换为 void*型指针。

如果数组只有部分内容需要被复制，那么需要复制的数量必须在第 3 个参数中指明。对于长度大于一个字节的数据，要确保把数量和数据类型的长度相乘，例如：

```
memcpy( saved_answers, answers, count * sizeof( answers[0] ) );
```

你也可以使用这种技巧复制结构或结构数组。

memmove 函数的行为和 memcpy 差不多，只是它的源和目标操作数可以重叠。虽然它并不需要以下面这种方式实现，不过 memmove 的结果和这种方法的结果相同：把源操作数复制到一个临时位置，这个临时位置不会与源或目标操作数重叠，然后再把它从这个临时位置复制到目标操作数。memmove 通常无法使用某些机器所提供的特殊的字节-字符串处理指令来实现，所以它可能比 memcpy 慢一些。但是，如果源和目标参数真的可能存在重叠，就应该使用 memmove，如下例所示：

```
/*
** Shift the values in the x array left one position.
*/
memmove( x, x + 1, ( count - 1 ) * sizeof( x[ 0 ] ) );
```

memcmp 对两段内存的内容进行比较，这两段内存分别起始于 a 和 b，共比较 length 个字节。这些值按照无符号字符逐字节进行比较，函数的返回类型和 strcmp 函数一样——负值表示 a 小于 b，正值表示a大于b，零表示a等于b。由于这些值是根据一串无符号字节进行比较的，所以如果memcmp函数用于比较不是单字节的数据如整数或浮点数时就可能给出不可预料的结果。

memchr 从 a 的起始位置开始查找字符 ch 第 1 次出现的位置，并返回一个指向该位置的指针，它共查找 length 个字节。如果在这 length 个字节中未找到该字符，函数就返回一个 NULL 指针。

最后，memset 函数把从 a 开始的 length 个字节都设置为字符值 ch。例如：

```
memset( buffer, 0, SIZE );
```

把 buffer 的前 SIZE 个字节都初始化为 0。

9.10 总结

字符串就是零个或多个字符的序列，该序列以一个 NUL 字节结尾。字符串的长度就是它所包含的字符的数目。标准库提供了一些函数用于处理字符串，它们的原型位于头文件 string.h 中。

strlen 函数用于计算一个字符串的长度，它的返回值是一个无符号整数，所以把它用于表达式时应该小心。strcpy 函数把一个字符串从一个位置复制到另一个位置，而 strcat 函数把一个字符串的一份拷贝添加到另一个字符串的后面。这两个函数都假定它们的参数是有效的字符串，而且如果源字符串和目标字符串出现重叠，函数的结果是未定义的。strcmp 对两个字符串进行词典序的比较。它的返回值提示第 1 个字符串是大于、小于还是等于第 2 个字符串。

长度受限的函数 strncpy、strncat 和 strncmp 都类似它们对应的不受限制版本。区别在于这些函数还接受一个长度参数。在 strncpy 中，长度指定了多少个字符将被写入到目标字符数组中。如果源字符串比指定长度更长，结果字符串将不会以 NUL 字节结尾。strncat 函数的长度参数指定从源字符串复制过来的字符的最大数目，但它的结果始终以一个 NUL 字节结尾。strcmp 函数的长度参数用于限定字符比较的数目。如果两个字符串在指定的数目里不存在区别，它们便被认为是相等的。

用于查找字符串的函数有好几个。strchr 函数查找一个字符串中某个字符第 1 次出现的位置。strrchr 函数查找一个字符串中某个字符最后一次出现的位置。strpbrk 在一个字符串中查找一个指定字符集中任意字符第 1 次出现的位置。strstr 函数在一个字符串中查找另一个字符串第 1 次出现的位置。

标准库还提供了一些更加高级的字符串查找函数。strspn 函数计算一个字符串的起始部分匹配

一个指定字符集中任意字符的字符数量。strcspn 函数计算一个字符串的起始部分不匹配一个指定字符集中任意字符的字符数量。strtok 函数把一个字符串分割成几个标记。每次当它调用时，都返回一个指向字符串中下一个标记位置的指针。这些标记由一个指定字符集的一个或多个字符分隔。

strerror 把一个错误代码作为它的参数。它返回一个指向字符串的指针，该字符串用于描述这个错误。

标准库还提供了各种用于测试和转换字符的函数。使用这些函数的程序比那些自己执行字符测试和转换的程序更具移植性。toupper 函数把一个小写字母字符转换为大写形式，tolower 函数则执行相反的任务。iscntrl 函数检查它的参数是不是一个控制字符，isspace 函数测试它的参数是否为空白字符。isdigit 函数用于测试它的参数是否为一个十进制数字字符，isxdigit 函数则检查它的参数是否为一个十六进制数字字符。islower 和 isupper 函数分别检查它们的参数是否为大写和小写字母。isalpha 函数检查它的参数是否为字母字符，isalnum 函数检查它的参数是否为字母或数字字符，ispunct 函数检查它的参数是否为标点符号字符。最后，isgraph 函数检查它的参数是否为图形字符，isprint 函数检查它的参数是否为图形字符或空白字符。

memxxx 函数提供了类似字符串函数的能力，但它们可以处理包括 NUL 字节在内的任意字节。这些函数都接受一个长度参数。memcpy 从源参数向目标参数复制由长度参数指定的字节数。memmove 函数执行相同的功能，但它能够正确处理源参数和目标参数出现重叠的情况。memcmp 函数比较两个序列的字节，memchr 函数在一个字节序列中查找一个特定的值。最后，memset 函数把一序列字节初始化为一个特定的值。

9.11　警告的总结

1. 应该使用有符号数的表达式中使用 strlen 函数。
2. 在表达式中混用有符号数和无符号数。
3. 使用 strcpy 函数把一个长字符串复制到一个较短的数组中，导致溢出。
4. 使用 strcat 函数把一个字符串添加到一个数组中，导致数组溢出。
5. 把 strcmp 函数的返回值当作布尔值进行测试。
6. 把 strcmp 函数的返回值与 1 和-1 进行比较。
7. 使用并非以 NUL 字节结尾的字符序列。
8. 使用 strncpy 函数产生不以 NUL 字节结尾的字符串。
9. 把 strncpy 函数和 strxxx 族函数混用。
10. 忘了 strtok 函数将会修改它所处理的字符串。
11. strtok 函数是不可再入的[1]。

9.12　编程提示的总结

1. 不要试图自己编写功能相同的函数来取代库函数。
2. 使用字符分类和转换函数可以提高函数的移植性。

---

[1] 译注:不可再入是指函数在连续几次调用中，即使它们的参数相同，其结果也可能不同。

9.13 问题

1. C 语言缺少显式的字符串数据类型，这是一个优点还是一个缺点？
2. strlen 函数返回一个无符号量(size_t)，为什么这里无符号值比有符号值更合适？但返回无符号值其实也有缺点，为什么？
3. 如果 strcat 和 strcpy 函数返回一个指向目标字符串末尾的指针，和事实上返回一个指向目标字符串起始位置的指针相比，有没有什么优点？
4. 如果从数组 x 复制 50 个字节到数组 y，最简单的方法是什么？
5. 假定你有一个名叫 buffer 的数组，它的长度为 BSIZE 个字节，你用下面这条语句把一个字符串复制到这个数组：

```
strncpy( buffer, some_other_string, BSIZE - 1 );
```

它能不能保证 buffer 中的内容是一个有效的字符串？

6. 用下面这种方法

```
if( isalpha( ch ) ){
```

取代下面这种显式的测试有什么优点？

```
if( ch >= 'A' && ch <= 'Z' ||
    ch >= 'a' && ch <= 'z' ){
```

7. 下面的代码怎样进行简化？

```
for( p_str = message; *p_str != '\0'; p_str++ ){
    if( islower( *p_str ) )
        *p_str = toupper( *p_str );
}
```

8. 下面的表达式有何不同？

```
memchr( buffer, 0, SIZE ) - buffer
strlen( buffer )
```

9.14 编程练习

★ 1. 编写一个程序，从标准输入读取一些字符，并统计下列各类字符所占的百分比。

控制字符
空白字符
数字
小写字母
大写字母
标点符号
不可打印的字符

请使用在 ctype.h 头文件中定义的字符分类函数。

★ 2. 编写一个名叫 my_strlen 的函数。它类似于 strlen 函数，但它能够处理由于使用 strn---

函数而创建的未以 NUL 字节结尾的字符串。你需要向函数传递一个参数，它的值就是保存了需要进行长度测试的字符串的数组的长度。

★ 3. 编写一个名叫 my_strcpy 的函数。它类似于 strcpy 函数，但它不会溢出目标数组。复制的结果必须是一个真正的字符串。

★ 4. 编写一个名叫 my_strcat 的函数。它类似于 strcat 函数，但它不会溢出目标数组。它的结果必须是一个真正的字符串。

★ 5. 编写函数

```
void my_strncat( char *dest, char *src, int dest_len );
```

它用于把 src 中的字符串连接到 dest 中原有字符串的末尾，但它保证不会溢出长度为 dest_len 的 dest 数组。和 strncat 函数不同，这个函数也考虑原先存在于 dest 数组的字符串长度，因此能够保证不会超越数组边界。

★ 6. 编写一个名叫 my_strcpy_end 的函数取代 strcpy 函数，它返回一个指向目标字符串末尾的指针（也就是说，指向 NUL 字节的指针），而不是返回一个指向目标字符串起始位置的指针。

★ 7. 编写一个名叫 my_strrchr 的函数，它的原型如下：

```
char *my_strrchr( char const *str, int ch );
```

这个函数类似于 strchr 函数，只是它返回的是一个指向 ch 字符在 str 字符串中最后一次出现（最右边）的位置的指针。

★ 8. 编写一个名叫 my_strnchr 的函数，它的原型如下：

```
char *my_strnchr( char const *str, int ch, int which );
```

这个函数类似于 strchr 函数，但它的第 3 个参数指定 ch 字符在 str 字符串中第几次出现。例如，如果第 3 个参数为 1，这个函数的功能就和 strchr 完全一样。如果第 3 个参数为 2，这个函数就返回一个指向 ch 字符在 str 字符串中第 2 次出现的位置的指针。

★★ 9. 编写一个函数，它的原型如下：

```
int count_chars( char const *str,
char const *chars );
```

函数应该在第 1 个参数中进行查找，并返回匹配第 2 个参数所包含的字符的数量。

★★★ 10. 编写函数

```
int palindrome( char *string );
```

如果参数字符串是个回文，函数就返回真，否则就返回假。回文就是指一个字符串从左向右读和从右向左读是一样的[1]。函数应该忽略所有的非字母字符，而且在进行字符比较时不用区分大小写。

★★★ 11. 编写一个程序，对标准输入进行扫描，并对单词“the”出现的次数进行计数。进行比较时应该区分大小写，所以“The”和“THE”并不计算在内。你可以认为

[1] 前提是空白字符、标点符号和大小写状态被忽略。当 Adam（亚当）第 1 次遇到 Eve（夏娃）时他可能会说的一句话：“Madam, I'm Adam”就是回文一例。

各单词由一个或多个空格字符分隔，而且输入行在长度上不会超过 100 个字符。计数结果应该写到标准输出上。

★★★ 12. 有一种技巧可以对数据进行加密，并使用一个单词作为它的密匙。下面是它的工作原理：首先，选择一个单词作为密匙，如 TRAILBLAZERS。如果单词中包含有重复的字母，只保留第 1 个，其余几个丢弃。现在，修改过的那个单词列于字母表的下面，如下所示：

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
T R A I L B Z E S
```

最后，底下那行用字母表中剩余的字母填充完整：

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
T R A I L B Z E S C D F G H J K M N O P Q U V W X Y
```

在对信息进行加密时，信息中的每个字母被固定于顶上那行，并用下面那行的对应字母一一取代原文的字母。因此，使用这个密匙，ATTACK AT DAWN（黎明时攻击）就会被加密为 TPPTAD TP ITVH。

这个题材共有三个程序（包括下面两个练习），在第 1 个程序中，你需要编写函数

```
int prepare_key( char *key );
```

它接受一个字符串参数，它的内容就是需要使用的密匙单词。函数根据上面描述的方法把它转换成一个包含编好码的字符数组。假定 key 参数是个字符数组，其长度至少可以容纳 27 个字符。函数必须把密匙中的所有字符要么转换为大写字母，要么转换为小写字母（随你选择），并从单词中去除重复的字母，然后再用字母表中剩余的字母按照你原先所选择的大小写形式填充到 key 数组中。如果处理成功，函数返回一个真值。如果 key 参数为空或者包含任何非字母字符，函数将返回一个假值。

★★ 13. 编写函数

```
void encrypt( char *data, char const *key );
```

它使用前题 prepare_key 函数所产生的密匙对 data 中的字符进行加密。data 中的非字母字符不作修改，但字母字符则用密匙所提供的编过码的字符一一取代源字符。字母字符的大小写状态应该保留。

★★ 14. 这个问题的最后部分就是编写函数

```
void decrypt( char *data, char const *key );
```

它接受一个加过密的字符串为参数，它的任务是重现原来的信息。除了它是用于解密之外，它的工作原理应该与 encrypt 相同。

★★★ 15. 标准 I/O 库并没有提供一种机制，在打印大整数时用逗号进行分隔。在这个练习中，你需要编写一个程序，为美元数额的打印提供这个功能。函数将把一个数字字符串（代表以美分为单位的金额）转换为美元形式，如下面的例子所示：

输入	输出	输入	输出
空	$0.00	12345	$123.45
1	$0.01	123456	$1,234.56
12	$0.12	1234567	$12,345.67
123	$1.23	12345678	$123,456.78
1234	$12.34	123456789	$1,234,567.89

下面是函数的原型：

```
void dollars( char *dest, char const *src );
```

src 将指向需要被格式化的字符（你可以假定它们都是数字）。函数应该像上面例子所示的那样对字符进行格式化，并把结果字符串保存到 dest 中。你应该保证你所创建的字符串以一个 NUL 字节结尾。src 的值不应被修改。你应该使用指针而不是下标。

提示：首先找到第 2 个参数字符串的长度。这个值有助于判断逗号应插入到什么位置。同时，小数点和最后两位数字应该是唯一的需要你进行处理的特殊情况。

★★★ 16. 这个程序与前一个练习的程序相似，但它更为通用。它按照一个指定的格式字符串对一个数字字符串进行格式化，类似许多 BASIC 编译器所提供的“print using”语句。函数的原型应该如下：

```
int format( char *format_string,
char const *digit_string );
```

digit_string 中的数字根据一开始在 format_string 中找到的字符从右到左逐个复制到 format_string 中。注意被修改后的 format_string 就是这个处理过程的结果。当你完成时，确定 format_string 依然是以 NUL 字节结尾的。根据格式化过程中是否出现错误，函数返回真或假。

格式字符串可以包含下列字符：

#　在两个字符串中都是从右向左进行操作。格式字符串中的每个#字符都被数字字符串中的下一个数字取代。如果数字字符串用完，格式字符串中所有剩余的#字符由空白代替（但存在例外，请参见下面对小数点的讨论）。

,　如果逗号左边至少有一位数字，那么它就不作修改。否则它由空白代替。

.　小数点始终作为小数点存在。如果小数点左边没有一位数字，那么小数点左边的那个位置以及右边直到有效数字为止的所有位置都由 0 填充。

下面的例子说明了对这个函数的一些调用的结果。符号□用于表示空白。

为了简化这个项目，你可以假定格式字符串所提供的格式总是正确的。最左边至少有一个#符号，小数点和逗号的右边也至少有一个#符号。而且逗号绝不会出现在小数点的右边。你需要进行检查的错误只有：

a）数字字符串中的数字多于格式字符串中的#符号。

b）数字字符串为空。

发生这两种错误时，函数返回假，否则返回真。如果数字字符串为空，格式字符串在返回时应未作修改。如果你使用指针而不是下标来解决问题，你将会学到更多的东西。

格式字符串	数字字符串	结果格式字符串
#####	12345	12345
#####	123	¤¤123
##,###	1234	¤1,234
##,###	123	¤¤¤123
##,###	1234567	34,567
#,###,###.##	123456789	1,234,567.89
#,###,###.##	1234567	¤¤¤12,345.67
#,###,###.##	123	¤¤¤¤¤¤¤¤1.23
#,###,###.##	1	¤¤¤¤¤¤¤¤0.01
#####.#####	1	¤¤¤¤0.00001

提示：开始时让两个指针分别指向格式字符串和数字字符串的末尾，然后从右向左进行处理。对于作为参数传递给函数的指针，你必须保留它的值，这样你就可以判断是否到达了这些字符串的左端。

★★★★ 17. 这个程序与前两个练习类似，但更加一般化了。它允许调用程序把逗号放在大数的内部，去除多余的前导零以及提供一个浮动美元符号等。

这个函数的操作类似于 IBM 370 机器上的 Edit 和 Mark 指令。它的原型如下：

```
char *edit( char *pattern, char const *digits );
```

它的基本思路很简单。模式(pattern)就是一个图样，处理结果看上去应该像它的样子。数字字符串中的字符根据这个图样所提供的方式**从左向右**复制到模式字符串。数字字符串的第 1 位有效数字很重要。结果字符串中所有在第 1 位有效数字之前的字符都由一个“填充”字符代替，函数将返回一个指针，它所指向的位置正是第 1 位有效数字存储在结果字符串中的位置（调用程序可以根据这个返回指针，把一个浮动美元符号放在这个值左边的毗邻位置）。这个函数的输出结果就像支票上打印的结果一样——这个值左边所有的空白由星号或其他字符填充。

在描述这个函数的详细处理过程之前，看一些这个操作的例子是有很帮助的。为了清晰起见，符号¤用于表示空格。结果字符串中带下划线的那个数字就是返回值指针所指向的字符（也就是第 1 位有效数字），如果结果字符串中不存在带下划线的字符，说明函数的返回值是个 NULL 指针。

模式字符串	数字字符串	结果字符串
*#,###	1234	*1,234
*#,###	123456	*1,234
*#,###	12	*1,2
*#,###	0012	****12
*#,###	¤¤12	****12
*#,###	¤1¤¤	***100
*X#Y#Z	空	**
¤#,##!.##	¤23456	¤¤¤234.56
¤#,##!.##	023456	¤¤¤234.56
$#,##!.##	¤¤¤456	$$$$4.56
$#,##!.##	0¤¤¤¤6	$$$$0.06
$#,##!.##	0	$$$
$#,##!.##	1	$1,
$#,##!.##	Hi¤there	$H,i0t.he

现在，让我们讨论这个函数的细节。函数的第 1 个参数就是模式，模式字符串的第 1 个字符就是“填充字符”。函数使数字字符串修改模式字符串中剩余的字符来产生结果字符串。在处理过程中，模式字符串将被修改。输出字符串不可能比原先的模式字符串更长，所以不存在溢出第 1 个参数的危险（因此不需要对此进行检查）。

模式是从左向右逐个字符进行处理的。每个位于填充字符后面的字符的处理结果将是三中选一：(a)原样保留，不作修改；(b)被一个数字字符串中的字符代替；(c)被填充字符代替。

数字字符串也是从左向右进行处理的，但它本身在处理过程中绝不会被修改。虽然它被称为“数字字符串”，但是它也可以包含任何其他字符，如上面的例子之一所示。但是，数字字符串中的空格应该和数字 0 一样对待（它们的处理结果相同）。函数必须保持一个“有效”标志，用于标志是否有任何有效数字从数字字符串复制到模式字符串。数字字符串中的前导空格和前导 0 并非有效数字，其余的字符都是有效数字。

如果模式字符串或数字字符串有一个是 NULL，那就是个错误。在这种情况下，函数应该立即返回 NULL。

下面这个表列出了所有需要的处理过程。列标题“signif”就是有效标志。“模式”和“数字”分别表示模式字符串和数字字符串的下一个字符。表的左边列出了所有可能出现的不同情况，表的右边描述了每种情况需要的处理过程。例如，如果下一个模式字符是'#'，有效标志就设为假。数字字符串的下一个字符是'0'，所以用一个填充字符代替模式字符串中的#字符，对有效标志不作修改。

如果你找到这个…			你应该这样处理...		
模式	signif	数字	模式	signif	说明
'\0'	无关紧要	不使用	不作修改	不作修改	返回保存的指针
'#'	无关紧要	'\0'	'\0'	不作修改	返回保存的指针
	假	'0'或' '	填充字符	不作修改	
		其他任何字符	数字	真	保存指向该字符的指针
	真	任何字符	数字	不作修改	
'!'	无关紧要	'\0'	'\0'	不作修改	返回保存的指针
	假	任何字符	数字	真	保存指向该字符的指针
	真	任何字符	数字	不作修改	
其他任何符号	假	不使用	填充字符	不作修改	
	真	不使用	不作修改	不作修改	