第1章 快速上手

1.1 简介

从头开始介绍一门编程语言总是显得很困难，因为有许多细节还没有介绍，很难让读者在头脑中形成一幅完整的图。在本章中，我将向大家展示一个例子程序，并逐行讲解它的工作过程，试图让大家对C语言的整体有一个大概的印象。这个例子程序同时向你展示了你所熟悉的过程在C语言中是如何实现的。这些信息再加上本章所讨论的其他主题，向你介绍了C语言的基础知识，这样你就可以自己编写有用的C程序了。

我们所要分析的这个程序从标准输入读取文本并对其进行修改，然后把它写到标准输出。程序1.1首先读取一串列标号。这些列标号成对出现，表示输入行的列范围。这串列标号以一个负值结尾，作为结束标志。剩余的输入行被程序读入并打印，然后输入行中被选中范围的字符串被提取出来并打印。注意，每行第1列的列标号为零。例如，如果输入如下：

```
4 9 12 20 -1
abcdefghijklmnopqrstuvwxyz
Hello there, how are you?
I am fine, thanks.
See you!
Bye
```

则程序的输出如下：

```
Original input : abcdefghijklmnopqrstuvwxyz
Rearranged line: efghijmnopqrstu
Original input : Hello there, how are you?
Rearranged line: o ther how are
Original input : I am fine, thanks.
Rearranged line:  fine,hanks.
Original input : See you!
Rearranged line: you!
Original input : Bye
Rearranged line:
```

这个程序的重要之处在于它展示了当你开始编写C程序时所需要知道的绝大多数基本技巧。

```
/*
** 这个程序从标准输入中读取输入行并在标准输出中打印这些输入行，
```

```
** 每个输入行的后面一行是该行内容的一部分。
**
** 输入的第 1 行是一串列标号，串的最后以一个负数结尾。
** 这些列标号成对出现，说明需要打印的输入行的列的范围。
** 例如，0 3 10 12 -1 表示第 0 列到第 3 列，第 10 列到第 12 列的内容将被打印。
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define   MAX_COLS  20              /* 所能处理的最大列号 */
#define   MAX_INPUT 1000            /* 每个输入行的最大长度 */

int   read_column_numbers( int columns[], int max );
void  rearrange( char *output, char const *input,
      int n_columns, int const columns[] );

int main( void )
{
     int    n_columns;             /* 进行处理的列标号 */
     int    columns[MAX_COLS];     /* 需要处理的列数 */
     char  input[MAX_INPUT];       /* 容纳输入行的数组 */
     char  output[MAX_INPUT];      /* 容纳输出行的数组 */

     /*
     ** 读取该串列标号
     */
     n_columns = read_column_numbers( columns, MAX_COLS );

     /*
     ** 读取、处理和打印剩余的输入行。
     */
     while( gets( input ) != NULL ){
          printf( "Original input : %s\n", input );
          rearrange( output, input, n_columns, columns );
          printf( "Rearranged line: %s\n", output );
     }

     return EXIT_SUCCESS;
}

/*
** 读取列标号，如果超出规定范围则不予理会。
*/
int read_column_numbers( int columns[], int max )
{
     int  num = 0;
     int  ch;

     /*
     ** 取得列标号，如果所读取的数小于 0 则停止。
     */
     while( num < max && scanf( "%d", &columns[num] ) == 1
          && columns[num] >= 0 )
             num += 1;
```

```
    /*
    ** 确认已经读取的标号为偶数个，因为它们是以对的形式出现的。
    */
    if( num % 2 != 0 ){
        puts( "Last column number is not paired." );
        exit( EXIT_FAILURE );
    }

    /*
    ** 丢弃该行中包含最后一个数字的那部分内容。
    */
    while( (ch = getchar()) != EOF && ch != '\n' )
        ;

    return num;
}

/*
** 处理输入行，将指定列的字符连接在一起，输出行以 NUL 结尾。
*/
void rearrange( char *output, char const *input,
    int n_columns, int const columns[] )
{
    int  col;           /* columns 数组的下标 */
    int  output_col;    /* 输出列计数器 */
    int  len;           /* 输入行的长度 */

    len = strlen( input );
    output_col = 0;

    /*
    ** 处理每对列标号。
    */
    for( col = 0; col < n_columns; col += 2 ){
        int nchars = columns[col + 1] - columns[col] + 1;

        /*
        ** 如果输入行结束或输出行数组已满，就结束任务。
        */
        if( columns[col] >= len ||
            output_col == MAX_INPUT - 1 )
            break;

        /*
        ** 如果输出行数据空间不够，只复制可以容纳的数据。
        */
        if( output_col + nchars > MAX_INPUT - 1 )
            nchars = MAX_INPUT - output_col - 1;

        /*
        ** 复制相关的数据。
        */
        strncpy( output + output_col, input + columns[col],
            nchars );
        output_col += nchars;
    }
```

```
        output[output_col] = '\0';
    }
```

程序 1.1 重排字符 rearrang.c

1.1.1 空白和注释

现在，让我们仔细观察这个程序。首先需要注意的是程序的空白：空行将程序的不同部分分隔开来；制表符（tab）用于缩进语句，更好地显示程序的结构等等。C 是一种自由格式的语言，并没有规则要求你必须怎样书写语句。然而，如果你在编写程序时能够遵守一些约定还是非常值得的，它可以使代码更加容易阅读和修改，千万不要小看了这一点。

清晰地显示程序的结构固然重要，但告诉读者程序能做些什么以及怎样做则更为重要。**注释**(comment)就是用于实现这个功能。

```
/*
** 这个程序从标准输入中读取输入行并在标准输出中打印这些输入行，
** 每个输入行的后面一行是该行内容的一部分。
**
** 输入的第一行是一串列标号，串的最后以一个负数结尾。
** 这些列标号成对出现，说明需要被打印的输入行的列范围。
** 例如，0 3 10 12 -1 表示第 0 列到第 3 列，第 10 列到第 12 列的内容将被打印。
*/
```

这段文字就是**注释**。注释以符号/*开始，以符号*/结束。在 C 程序中，凡是可以插入空白的地方都可以插入注释。然而，注释不能嵌套，也就是说，第 1 个/*符号和第 1 个*/符号之间的内容都被看作是注释，不管里面还有多少个/*符号。

在有些语言中，注释有时用于把一段代码“注释掉”，也就是使这段代码在程序中不起作用，但并不将其真正从源文件中删除。在 C 语言中，这可不是个好主意，如果你试图在一段代码的首尾分别加上/*和*/符号来“注释掉”这段代码，你不一定能如愿。如果这段代码内部原先就有注释存在，这样做就会出问题。要从逻辑上删除一段 C 代码，更好的办法是使用#if 指令。只要像下面这样使用：

```
#if  0
      statements
#endif
```

在#if 和#endif 之间的程序段就可以有效地从程序中去除，即使这段代码之间原先存在注释也无妨，所以这是一种更为安全的方法。预处理指令的作用远比你想象的要大，我将在第 14 章详细讨论这个问题。

1.1.2 预处理指令

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define    MAX_COLS 20      /* 能够处理的最大列号 */
#define    MAX_INPUT1000    /* 每个输入行的最大长度 */
```

这 5 行称为**预处理指令**(preprocessor directives)，因为它们是由**预处理器**(preprocessor)解释的。预处理器读入源代码，根据预处理指令对其进行修改，然后把修改过的源代码递交给编译器。

在我们的例子程序中，预处理器用名叫 stdio.h 的库函数头文件的内容替换第 1 条#include 指令语句，其结果就仿佛是 stdio.h 的内容被逐字写到源文件的那个位置。第 2、3 条指令的功能类似，只是它们所替换的头文件分别是 stdlib.h 和 string.h。

stdio.h 头文件使我们可以访问**标准 I/O** 库(Standard I/O Library)中的函数，这组函数用于执行输入和输出。stdlib.h 定义了 EXIT_SUCCESS 和 EXIT_FAILURE 符号。我们需要 string.h 头文件提供的函数来操纵字符串。

提示：

如果你有一些声明需要用于几个不同的源文件，这个技巧也是一种方便的方法——你在一个单独的文件中编写这些声明，然后用#include 指令把这个文件包含到需要使用这些声明的源文件中。这样，你就只需要这些声明的一份拷贝，用不着在许多不同的地方进行复制，避免了在维护这些代码时出现错误的可能性。

提示：

另一种预处理指令是#define，它把名字 MAX_COLS 定义为 20，把名字 MAX_INPUT 定义为 1000。当这个名字以后出现在源文件的任何地方时，它就会被替换为定义的值。由于它们被定义为字面值常量，所以这些名字不能出现于有些普通变量可以出现的场合（比如赋值符的左边）。这些名字一般都大写，用于提醒它们并非普通的变量。#define 指令和其他语言中符号常量的作用类似，其出发点也相同。如果以后你觉得 20 列不够，你可以简单地修改 MAX_COLS 的定义，这样你就用不着在整个程序中到处寻找并修改所有表示列范围的 20，你有可能漏掉一个，也可能把并非用于表示列范围的 20 也修改了。

```
int  read_column_numbers( int columns[], int max );
void rearrange( char *output, char const *input,
                int n_columns, int const columns[] );
```

这些声明被称为**函数原型**(function prototype)。它们告诉编译器这些以后将在源文件中定义的函数的特征。这样，当这些函数被调用时，编译器就能对它们进行准确性检查。每个原型以一个类型名开头，表示函数返回值的类型。跟在返回类型名后面的是函数的名字，再后面是函数期望接受的参数。所以，函数 read_column_numbers 返回一个整数，接受两个类型分别是整型数组和整型标量的参数。函数原型中参数的名字并非必需，我这里给出参数名的目的是提示它们的作用。

rearrange 函数接受 4 个参数。其中第 1 个和第 2 个参数都是**指针**(pointer)。指针指定一个存储于计算机内存中的值的地址，类似于门牌号码指定某个特定的家庭位于街道的何处。指针赋予 C 语言强大的威力，我将在第 6 章详细讲解指针。第 2 个和第 4 个参数被声明为 const，这表示函数将不会修改函数调用者所传递的这两个参数。关键字 void 表示函数并不返回任何值，在其他语言里，这种无返回值的函数被称为**过程**(procedure)。

提示：

假如这个程序的源代码由几个源文件所组成，那么使用该函数的源文件都必须写明该函数的原型。把原型放在头文件中并使用#include 指令包含它们，可以避免由于同一个声明的多份拷贝而导致的维护性问题。

1.1.3　main 函数

```
int main( void )
```

```
{
```

这几行构成了 main 函数定义的起始部分。每个 C 程序都必须有一个 main 函数，因为它是程序执行的起点。关键字 int 表示函数返回一个整型值，关键字 void 表示函数不接受任何参数。main 函数的函数体包括左花括号和与之相匹配的右花括号之间的任何内容。

请观察一下缩进是如何使程序的结构显得更为清晰的。

```
    int    n_columns;              /* 进行处理的列标号 */
    int    columns[MAX_COLS];      /* 需要处理的列数 */
    char   input[MAX_INPUT];       /* 容纳输入行的数组 */
    char   output[MAX_INPUT];      /* 容纳输出行的数组 */
```

这几行声明了 4 个变量：一个整型标量，一个整型数组以及两个字符数组。所有 4 个变量都是 main 函数的局部变量，其他函数不能根据它们的名字访问它们。当然，它们可以作为参数传递给其他函数。

```
    /*
    ** 读取该串列标号
    */
    n_columns = read_column_numbers( columns, MAX_COLS );
```

这条语句调用函数 read_column_numbers。数组 columns 和 MAX_COLS 所代表的常量(20)作为参数传递给这个函数。在 C 语言中，数组参数是以**引用**(reference)形式进行传递的，也就是传址调用，而标量和常量则是按**值**(value)传递的（分别类似于 Pascal 和 Modula 中的 var 参数和值参数）。在函数中对标量参数的任何修改都会在函数返回时丢失，因此，被调用函数无法修改调用函数以传值形式传递给它的参数。然而，当被调用函数修改数组参数的其中一个元素时，调用函数所传递的数组就会被实际地修改。

事实上，关于 C 函数的参数传递规则可以表述如下：

所有传递给函数的参数都是按值传递的。

但是，当数组名作为参数时就会产生按引用传递的效果，如上所示。规则和现实行为之间似乎存在明显的矛盾之处，第 8 章会对此作出详细解释。

```
    /*
    ** 读取、处理和打印剩余的输入行。
    */
    while( gets( input ) != NULL ){
        printf( "Original input : %s\n", input );
        rearrange( output, input, n_columns, columns );
        printf( "Rearranged line: %s\n", output );
    }

    return EXIT_SUCCESS;
}
```

用于描述这段代码的注释看上去似乎有些多余。但是，如今软件开销的最大之处并非在于编写，而是在于维护。在修改一段代码时所遇到的第 1 个问题就是要搞清楚代码的功能。所以，如果你在代码中插入一些东西，能使其他人（或许就是你自己！）在以后更容易理解它，那就非常值得这样做。但是，要注意书写正确的注释，并且在你修改代码时要注意注释的更新。注释如果不正确那还不如没有！

这段代码包含了一个 while 循环。在 C 语言中，while 循环的功能和它在其他语言中一样。它首先测试表达式的值，如果是假的(0)就跳过循环体。如果表达式的值是真的（非 0），就执行循环体内的代码，然后再重新测试表达式的值。

这个循环代表了这个程序的主要逻辑。简而言之，它表示：

```
while 我们还可以读取另一行输入时
        打印输入行
        对输入行进行重新整理，把它存储于 output 数组
        打印输出结果
```

gets 函数从标准输入读取一行文本并把它存储于作为参数传递给它的数组中。一行输入由一串字符组成，以一个换行符(newline)结尾。gets 函数丢弃换行符，并在该行的末尾存储一个 NUL 字节[1]（一个 NUL 字节是指字节模式为全 0 的字节，类似'\0'这样的字符常量）。然后，gets 函数返回一个非 NULL 值，表示该行已被成功读取[2]。当 gets 函数被调用但事实上不存在输入行时，它就返回 NULL 值，表示它到达了输入的末尾（文件尾）。

在 C 程序中，处理字符串是常见的任务之一。尽管 C 语言并不存在“string”数据类型，但在整个语言中，存在一项约定：字符串就是一串以 NUL 字节结尾的字符。NUL 是作为字符串终止符，它本身并不被看作是字符串的一部分。**字符串常量**(string literal)就是源程序中被双引号括起来的一串字符。例如，字符串常量：

```
"Hello"
```

在内存中占据 6 个字节的空间，按顺序分别是 H、e、l、l、o 和 NUL。

printf 函数执行格式化的输出。C 语言的格式化输出比较简单，如果你是 Modula 或 Pascal 的用户，你肯定会对此感到愉快。printf 函数接受多个参数，其中第一个参数是一个字符串，描述输出的格式，剩余的参数就是需要打印的值。格式常常以字符串常量的形式出现。

格式字符串包含格式指定符（格式代码）以及一些普通字符。这些普通字符将按照原样逐字打印出来，但每个格式指定符将使后续参数的值按照它所指定的格式打印。表 1.1 列出了一些常用的格式指定符。如果数组 input 包含字符串 Hi friend!，那么下面这条语句

```
printf( "Original input : %s\n", input);
```

的打印结果是：

```
Original input : Hi friends!
```

后面以一个换行符终止。

表 1.1　　常用 printf 格式代码

格　式	含　义
%d	以十进制形式打印一个整型值
%o	以八进制形式打印一个整型值
%x	以十六进制形式打印一个整型值

[1] NUL 是 ASCII 字符集中‘\0’字符的名字，它的字节模式为全 0。NULL 指一个其值为 0 的指针。它们都是整型值，其值也相同，所以它们可以互换使用。然而，你还是应该使用适当的常量，因为它能告诉阅读程序的人不仅使用 0 这个值，而且告诉他使用这个值的目的。

[2] 符号 NULL 在头文件 stdio.h 中定义。另一方面，并不存在预定义的符号 NUL，所以如果你想使用它而不是字符常量‘\0’，你必须自行定义。

续表

格　式	含　义
%g	打印一个浮点值
%c	打印一个字符
%s	打印一个字符串
\n	换行

例子程序接下来的一条语句调用 rearrange 函数。后面 3 个参数是传递给函数的值，第 1 个参数则是函数将要创建并返回给 main 函数的答案。记住，这种参数是唯一可以返回答案的方法，因为它是一个数组。最后一个 printf 函数显示输入行重新整理后的结果。

最后，当循环结束时，main 函数返回值 EXIT_SUCCESS。该值向操作系统提示程序成功执行。右花括号标志着 main 函数体的结束。

1.1.4　read_column_numbers 函数

```
/*
** 读取列标号，如果超出规定范围则不予理会。
*/
int
read_column_numbers( int columns[], int max )
{
```

这几行构成了 read_column_numbers 函数的起始部分。注意，这个声明和早先出现在程序中的该函数原型的参数个数和类型以及函数的返回值完全匹配。如果出现不匹配的情况，编译器就会报错。

在函数声明的数组参数中，并未指定数组的长度。这种格式是正确的，因为不论调用函数的程序传递给它的数组参数的长度是多少，这个函数都将照收不误。这是一个伟大的特性，它允许单个函数操纵任意长度的一维数组。这个特性不利的一面是函数没法知道该数组的长度。如果确实需要数组的长度，它的值必须作为一个单独的参数传递给函数。

当本例的 read_column_numbers 函数被调用时，传递给函数的其中一个参数的名字碰巧与上面给出的形参名字相同。但是，其余几个参数的名字与对应的形参名字并不相同。和绝大多数语言一样，C 语言中形式参数的名字和实际参数的名字并没有什么关系。你可以让两者相同，但这并非必须。

```
int   num = 0;
int   ch;
```

这里声明了两个变量，它们是该函数的局部变量。第 1 个变量在声明时被初始化为 0，但第 2 个变量并未初始化。更准确地说，它的初始值将是一个不可预料的值，也就是垃圾。在这个函数里，它没有初始值并不碍事，因为函数对这个变量所执行的第 1 个操作就是对它赋值。

```
/*
** 取得列标号，如果所读取的数小于 0 则停止。
*/
while( num < max && scanf( "%d", &columns[num] ) == 1
    && columns[num] >= 0 )
        num += 1;
```

这又是一个循环，用于读取列标号。scanf 函数从标准输入读取字符并根据格式字符串对它们进

行转换——类似于 printf 函数的逆操作。scanf 函数接受几个参数，其中第 1 个参数是一个格式字符串，用于描述期望的输入类型。剩余几个参数都是变量，用于存储函数所读取的输入数据。scanf 函数的返回值是函数成功转换并存储于参数中的值的个数。

警告：

对于这个函数，你必须小心在意，理由有二。首先，由于 scanf 函数的实现原理，所有标量参数的前面必须加上一个“&”符号。关于这点，第 8 章我会解释清楚。数组参数前面不需要加上“&”符号[1]。但是，数组参数中如果出现了下标引用，也就是说实际参数是数组的某个特定元素，那么它的前面也必须加上“&”符号。在第 15 章，我会解释在标量参数前面加上“&”符号的必要性。现在，你只要知道必须加上这个符号就行了，因为如果没有它们的话，程序就无法正确运行。

警告：

第二个需要注意的地方是格式代码，它与 printf 函数的格式代码颇为相似却又并不完全相同，所以很容易引起混淆。表 1.2 粗略列出了一些你可能会在 scanf 函数中用到的格式代码。注意，前 5 个格式代码用于读取标量值，所以变量参数的前面必须加上“&”符号。使用所有格式码（除了%c 之外）时，输入值之前的空白（空格、制表符、换行符等）会被跳过，值后面的空白表示该值的结束。因此，用%s 格式码输入字符串时，中间不能包含空白。除了表中所列之外，还存在许多格式代码，但这张表里面的这几个格式代码对于应付我们现在的需求已经足够了。

我们现在可以解释表达式：

```
scanf("%d", &columns[num] )
```

格式码%d 表示需要读取一个整型值。字符是从标准输入读取，前导空白将被跳过。然后这些数字被转换为一个整数，结果存储于指定的数组元素中。我们需要在参数前加上一个“&”符号，因为数组下标选择的是一个单一的数组元素，它是一个标量。

while 循环的测试条件由 3 个部分组成：

```
num < max
```

这个测试条件确保函数不会读取过多的值，从而导致数组溢出。如果 scanf 函数转换了一个整数之后，它就会返回 1 这个值。最后，

```
columns[num] >= 0
```

这个表达式确保函数所读取的值是正数。如果两个测试条件之一的值为假，循环就会终止。

表 1.2　　常用 scanf 格式码

格　式	含　义	变量类型
%d	读取一个整型值	int
%ld	读取一个长整型值	long
%f	读取一个实型值(浮点数)	float
%lf	读取一个双精度实型值	double
%c	读取一个字符	char
%s	从输入中读取一个字符串	char 型数组

[1] 但是，即使你在它前面加上一个“&”也没有什么不对，所以如果你喜欢，也可以加上它。

提示：

标准并未硬性规定 C 编译器对数组下标的有效性进行检查，而且绝大多数 C 编译器确实也不进行检查。因此，如果你需要进行数组下标的有效性检查，你必须自行编写代码。如果此处不进行 num < max 这个测试，而且程序所读取的文件包含超过 20 个列标号，那么多出来的值就会存储在紧随数组之后的内存位置，这样就会破坏原先存储在这个位置的数据，可能是其他变量，也可以是函数的返回地址。这可能会导致多种结果，程序很可能不会按照你预想的那样运行。

&&是“逻辑与”操作符。要使整个表达式为真，&&操作符两边的表达式都必须为真。然而，如果左边的表达式为假，右边的表达式便不再进行求值，因为不管它是真是假，整个表达式总是假的。在这个例子中，如果 num 到达了它的最大值，循环就会终止[1]，而表达式

```
columns[num]
```

便不再被求值。

警告：

此处需要小心。当你实际上想使用&&操作符时，千万不要误用了&操作符。&操作符执行“按位与”的操作，虽然有些时候它的操作结果和&&操作符相同，但很多情况下都不一样。我将在第 5 章讨论这些操作符。

scanf 函数每次调用时都从标准输入读取一个十进制整数。如果转换失败，不管是因为文件已经读完还是因为下一次输入的字符无法转换为整数，函数都会返回 0，这样就会使整个循环终止。如果输入的字符可以合法地转换为整数，那么这个值就会转换为二进制数存储于数组元素 columns[num]中。然后，scanf 函数返回 1。

警告：

注意：用于测试两个表达式是否相等的操作符是==。如果误用了=操作符，虽然它也是合法的表达式，但其结果几乎肯定和你的本意不一样：它将执行赋值操作而不是比较操作！但由于它也是一个合法的表达式，所以编译器无法为你找出这个错误[2]。在进行比较操作时，千万要注意你所使用的是两个等号的比较操作符。如果你的程序无法运行，请检查一下所有的比较操作符，看看是不是这个地方出了问题。相信我，你肯定会犯这个错误，而且可能不止一次，我自己就曾经犯过这个错误。

接下来的一个&&操作符确保在 scanf 函数成功读取了一个数之后才对这个数进行是否赋值的测试。语句

```
num += 1;
```

使变量 num 的值增加 1，它相当于下面这个表达式

```
num = num + 1;
```

以后我将解释为什么 C 语言提供了两种不同的方式来增加一个变量的值[3]。

```
/*
```

---

1 “循环终止（the loop break）”这句话的意思是循环结束而不是它突然出现了毛病。这句话源于 break 语句，我们将在第 4 章讨论它。

2 有些较新的编译器在发现 if 和 while 表达式中使用赋值符时会发出警告信息，其理论是在这样的上下文环境中，用户需要使用比较操作的可能性要远大于赋值操作。

3 加上前缀和后缀++操作符，事实上共有 4 种方法增加一个变量的值。

```
** 确认已经读取的标号为偶数个，因为它们是以成对的形式出现的。
*/
if( num % 2 != 0 ){
      puts( "Last column number is not paired." );
      exit( EXIT_FAILURE );
}
```

这个测试检查程序所读取的整数是否为偶数个，这是程序规定的，因为这些数字要求成对出现。%操作符执行整数的除法，但它给出的结果是除法的余数而不是商。如果 num 不是一个偶数，它除以 2 之后的余数将不是 0。

puts 函数是 gets 函数的输出版本，它把指定的字符串写到标准输出并在末尾添上一个换行符。程序接着调用 exit 函数，终止程序的运行，EXIT_FAILURE 这个值被返回给操作系统，提示出现了错误。

```
/*
 ** 丢弃该行中包含最后一个数字的那部分内容。
 */
while( (ch = getchar()) != EOF && ch != '\n' )
        ;
```

当 scanf 函数对输入值进行转换时，它只读取需要读取的字符。这样，该输入行包含了最后一个值的剩余部分仍会留在那里，等待被读取。它可能只包含作为终止符的换行符，也可能包含其他字符。不论如何，while 循环将读取并丢弃这些剩余的字符，防止它们被解释为第 1 行数据。

下面这个表达式

```
(ch = getchar() ) != EOF && ch != '\n'
```

值得花点时间讨论。首先，getchar 函数从标准输入读取一个字符并返回它的值。如果输入中不再存在任何字符，函数就会返回常量 EOF(在 stdio.h 中定义)，用于提示文件的结尾。

从 getchar 函数返回的值被赋给变量 ch，然后把它与 EOF 进行比较。在赋值表达式两端加上括号用于确保赋值操作先于比较操作进行。如果 ch 等于 EOF，整个表达式的值就为假，循环将终止。若非如此，再把 ch 与换行符进行比较，如果两者相等，循环也将终止。因此，只有当输入尚未到达文件尾并且输入的字符并非换行符时，表达式的值才是真的（循环将继续执行）。这样，这个循环就能剔除当前输入行最后的剩余字符。

现在让我们进入有趣的部分。在大多数其他语言中，我们将像下面这个样子编写循环：

```
ch = getchar();
while( ch != EOF && CH != '\n' )
        ch = getchar();
```

它将读取一个字符，接下来如果我们尚未到达文件的末尾或读取的字符并不是换行符，它将继续读取下一个字符。注意这里两次出现了下面这条语句

```
ch = getchar();
```

C 可以把赋值操作蕴含于 while 语句内部，这样就允许程序员消除冗余语句。

提示：

例子程序中的那个循环的功能和上面这个循环相同，但它包含的语句要少一些。无可争议，这种形式可读性差一点。仅仅根据这个理由，你就可以理直气壮地声称这种编码技巧应该避免使用。但是，你之所以会觉得这种形式的代码可读性较差，只是因为你对 C 语言及其编程的习惯用法不熟

悉之故。经验丰富的 C 程序员在阅读（和编写）这类语句时根本不会出现困难。在没有明显的好处时，你应该避免使用影响代码可读性的方法。但在这种编程习惯用法中，同样的语句少写一次带来的维护方面的好处要更大一些。

一个经常问到的问题是：为什么 ch 被声明为整型，而我们事实上需要它来读取字符？答案是 EOF 是一个整型值，它的位数比字符类型要多，把 ch 声明为整型可以防止从输入读取的字符意外地被解释为 EOF。但同时，这也意味着接收字符的 ch 必须足够大，足以容纳 EOF，这就是 ch 使用整型值的原因。正如第 3 章所讨论的那样，字符只是小整型数而已，所以用一个整型变量容纳字符值并不会引起任何问题。

提示：

对这段程序最后还有一点说明：这个 while 循环的循环体没有任何语句。仅仅完成 while 表达式的测试部分就足以达到我们的目的，所以循环体就无事可干。你偶尔也会遇到这类循环，处理它们应该没问题。while 语句之后的单独一个分号称为**空语句**(empty statement)，它就是应用于目前这个场合，也就是语法要求这个地方出现一条语句但又无需执行任何任务的时候。这个分号独占一行，这是为了防止读者错误地以为接下来的语句也是循环体的一部分。

```
    return num;
}
```

return 语句就是函数向调用它的表达式返回一个值。在这个例子里，变量 num 的值被返回给调用该函数的程序，后者把这个返回值赋值给主程序的 n_columns 变量。

1.1.5 rearrange 函数

```
/*
** 处理输入行，将指定列的字符连接在一起，输出行以 NUL 结尾。
*/
void
rearrange( char *output, char const *input,
     int n_columns, int const columns[] )
{
     int  col;           /* columns 数组的下标 */
     int  output_col;    /* 输出列计数器 */
     int  len;           /* 输入行的长度 */
```

这些语句定义了 rearrange 函数并声明了一些局部变量。此处最有趣的一点是：前两个参数被声明为指针，但在函数实际调用时，传给它们的参数却是数组名。当数组名作为实参时，传给函数的实际上是一个指向数组起始位置的指针，也就是数组在内存中的地址。正因为实际传递的是一个指针而不是一份数组的拷贝，才使数组名作为参数时具备了传址调用的语义。函数可以按照操纵指针的方式来操纵实参，也可以像使用数组名一样用下标来引用数组的元素。第 8 章将对这些技巧进行更详细的说明。

但是，由于它的传址调用语义，如果函数修改了形参数组的元素，它实际上将修改实参数组的对应元素。因此，例子程序把 columns 声明为 const 就有两方面的作用。首先，它声明该函数的作者的意图是这个参数不能被修改。其次，它导致编译器去验证是否违背该意图。因此，这个函数的调用者不必担心例子程序中作为第 4 个参数传递给函数的数组中的元素会被修改。

```
len = strlen( input );
```

```
output_col = 0;

/*
** 处理每对列标号。
*/
for( col = 0; col < n_columns; col += 2 ){
```

这个函数的真正工作是从这里开始的。我们首先获得输入字符串的长度，这样如果列标号超出了输入行的范围，我们就忽略它们。C 语言的 for 语句跟它在其他语言中不太像，它更像是 while 语句的一种常用风格的简写法。for 语句包含 3 个表达式（顺便说一下，这 3 个表达式都是可选的）。第一个表达式是**初始部分**，它只在循环开始前执行一次。第二个表达式是**测试部分**，它在循环每执行一次后都要执行一次。第三个表达式是**调整部分**，它在每次循环执行完毕后都要执行一次，但它在测试部分之前执行。为了清楚起见，上面这个 for 循环可以改写为如下所示的 while 循环：

```
col = 0;
while( col < n_columns ) {
        循环体
        col += 2;
}

int  nchars = columns[col + 1] - columns[col] + 1;

     /*
     ** 如果输入行结束或输出行数组已满，就结束任务。
     */
     if( columns[col] >= len ||
          output_col == MAX_INPUT - 1 )
            break;

     /*
     ** 如果输出行数据空间不够，只复制可以容纳的数据。
     */
     if( output_col + nchars > MAX_INPUT - 1 )
          nchars = MAX_INPUT - output_col - 1;

     /*
     ** 复制相关的数据。
     */
     strncpy( output + output_col, input + columns[col],
          nchars );
     output_col += nchars;
```

这是 for 循环的循环体，它一开始计算当前列范围内字符的个数，然后决定是否继续进行循环。如果输入行比起始列短，或者输出行已满，它便不再执行任务，使用 break 语句立即退出循环。

接下来的一个测试检查这个范围内的所有字符是否都能放入输出行中，如果不行，它就把 nchars 调整为数组能够容纳的大小。

提示：

在这种只使用一次的“一次性”程序中，不执行数组边界检查之类的任务，只是简单地让数组“足够大”从而使其不溢出的做法是很常见的。不幸的是，这种方法有时也应用于实际产品代码中。这种做法在绝大多数情况下将导致大部分数组空间被浪费，而且即使这样有时仍会出现溢出，从而

导致程序失败[1]。

最后，strncpy 函数把选中的字符从输入行复制到输出行中可用的下一个位置。strncpy 函数的前两个参数分别是目标字符串和源字符串的地址。在这个调用中，目标字符串的位置是输出数组的起始地址向后偏移 output_col 列的地址，源字符串的位置则是输入数组起始地址向后偏移 columns[col] 个位置的地址。第 3 个参数指定需要复制的字符数[2]。输出列计数器随后向后移动 nchars 个位置。

```
    }
    output[output_col] = '\0';
}
```

循环结束之后，输出字符串将以一个 NUL 字符作为终止符。注意，在循环体中，函数经过精心设计，确保数组仍有空间容纳这个终止符。然后，程序执行流便到达了函数的末尾，于是执行一条隐式的 return 语句。由于不存在显式的 return 语句，所以没有任何值返回给调用这个函数的表达式。在这里，不存在返回值并不会有问题，因为这个函数被声明为 void（也就是说，不返回任何值），并且当它被调用时，并不对它的返回值进行比较操作或把它赋值给其他变量。

1.2 补充说明

本章的例子程序描述了许多 C 语言的基础知识。但在你亲自动手编写程序之前，你还应该知道一些东西。首先是 putchar 函数，它与 getchar 函数相对应，它接受一个整型参数，并在标准输出中打印该字符(如前所述，字符在本质上也是整型)。

同时，在函数库里存在许多操纵字符串的函数。这里我将简单地介绍几个最有用的。除非特别说明，这些函数的参数既可以是字符串常量，也可以是字符型数组名，还可以是一个指向字符的指针。

strcpy 函数与 strncpy 函数类似，但它并没有限制需要复制的字符数量。它接受两个参数：第 2 个字符串参数将被复制到第 1 个字符串参数，第 1 个字符串原有的字符将被覆盖。strcat 函数也接受两个参数，但它把第 2 个字符串参数添加到第 1 个字符串参数的末尾。在这两个函数中，它们的第 1 个字符串参数不能是字符串常量。而且，确保目标字符串有足够的空间是程序员的责任，函数并不对其进行检查。

在字符串内进行搜索的函数是 strchr，它接受两个参数，第 1 个参数是字符串，第 2 个参数是一个字符。这个函数在字符串参数内搜索字符参数第 1 次出现的位置，如果搜索成功就返回指向这个位置的指针，如果搜索失败就返回一个 NULL 指针。strstr 函数的功能类似，但它的第 2 个参数也是一个字符串，它搜索第 2 个字符串在第 1 个字符串中第 1 次出现的位置。

1.3 编译

你编译和运行 C 程序的方法取决于你所使用的系统类型。在 UNIX 系统中，要编译一个存储于文件 testing.c 的程序，要使用以下命令：

---

1 精明的读者会注意到，如果遇到特别长的输入行，我们并没有办法防止 gets 函数溢出。这个漏洞确实是 gets 函数的缺陷，所以应该换用 fgets(将在第 15 章描述)。

2 如果源字符串的字符数少于第 3 个参数指定的复制数量，目标字符串中剩余的字节将用 NUL 字节填充。

```
cc testing.c
a.out
```

在 PC 中，你需要知道你所使用的是哪一种编译器。如果是 Borland C++，在 MS-DOS 窗口中，可以使用下面的命令：

```
bcc testing.c
testing
```

1.4　总结

本章的目的是描述足够的 C 语言的基础知识，使你对 C 语言有一个整体的印象。有了这方面的基础，在接下来章节的学习中，你会更加容易理解。

本章的例子程序说明了许多要点。注释以/*开始，以*/结束，用于在程序中添加一些描述性的说明。#include 预处理指令可以使一个函数库头文件的内容由编译器进行处理，#define 指令允许你给字面值常量取个符号名。

所有的 C 程序必须有一个 main 函数，它是程序执行的起点。函数的标量参数通过传值的方式进行传递，而数组名参数则具有传址调用的语义。字符串是一串由 NUL 字节结尾的字符，并且有一组库函数以不同的方式专门用于操纵字符串。printf 函数执行格式化输出，scanf 函数用于格式化输入，getchar 和 putchar 分别执行非格式化字符的输入和输出。if 和 while 语句在 C 语言中的用途跟它们在其他语言中的用途差不太多。

通过观察例子程序的运行之后，你或许想亲自编写一些程序。你可能觉得 C 语言所包含的内容应该远远不止这些，确实如此。但是，这个例子程序应该足以让你上手了。

1.5　警告的总结

1. 在 scanf 函数的标量参数前未添加&字符。
2. 机械地把 printf 函数的格式代码照搬于 scanf 函数。
3. 在应该使用&&操作符的地方误用了&操作符。
4. 误用=操作符而不是==操作符来测试相等性。

1.6　编程提示的总结

1. 使用#include 指令避免重复声明。
2. 使用#define 指令给常量值取名。
3. 在#include 文件中放置函数原型。
4. 在使用下标前先检查它们的值。
5. 在 while 或 if 表达式中蕴含赋值操作。
6. 如何编写一个空循环体。
7. 始终要进行检查，确保数组不越界。

1.7 问题

1. C 是一种自由形式的语言，也就是说并没有规则规定它的外观究竟应该怎样[1]。但本章的例子程序遵循了一定的空白使用规则。你对此有何想法？

✍ 2. 把声明（如函数原型的声明）放在头文件中，并在需要时用#include 指令把它们包含于源文件中，这种做法有什么好处？

3. 使用#define 指令给字面值常量取名有什么好处？

4. 依次打印一个十进制整数、字符串和浮点值，你应该在 printf 函数中分别使用什么格式代码？试编一例，让这些打印值以空格分隔，并在输出行的末尾添加一个换行符。

✍ 5. 编写一条 scanf 语句，它需要读取两个整数，分别保存于 quantity 和 price 变量，然后再读取一个字符串，保存在一个名叫 department 的字符数组中。

6. C 语言并不执行数组下标的有效性检查。你觉得为什么这个明显的安全手段会从语言中省略？

7. 本章描述的 rearrange 程序包含下面的语句

```
strncpy( output + output_col,
     input + columns[col], nchars );
```

strcpy 函数只接受两个参数，所以它实际上所复制的字符数由第 2 个参数指定。在本程序中，如果用 strcpy 函数取代 strncpy 函数会出现什么结果？

✍ 8. rearrange 程序包含下面的语句

```
while( gets( input ) != NULL ) {
```

你认为这段代码可能会出现什么问题？

1.8 编程练习

★ 1. “Hello world!”程序常常是 C 编程新手所编写的第 1 个程序。它在标准输出中打印 Hello world!，并在后面添加一个换行符。当你希望摸索出如何在自己的系统中运行 C 编译器时，这个小程序往往是一个很好的测试例。

✍★★ 2. 编写一个程序，从标准输入读取几行输入。每行输入都要打印到标准输出上，前面要加上行号。在编写这个程序时要试图让程序能够处理的输入行的长度没有限制。

★★ 3. 编写一个程序，从标准输入读取一些字符，并把它们写到标准输出上。它同时应该计算 checksum 值，并写在字符的后面。

checksum(检验和)用一个 singed char 类型的变量进行计算，它初始为-1。当每个字符从标准输入读取时，它的值就被加到 checksum 中。如果 checksum 变量产出了溢出，那么这些溢出就会被忽略。当所有的字符均被写入后，程序以十进制整数的形式打印出 checksum 的值，它有可能是负值。注意在 checksum 后面要添加一个换行符。

---

[1] 但预处理指令则有较严格的规则。

在使用 ASCII 码的计算机中，在包含“Hello world!”这几个词并以换行符结尾的文件上运行这个程序应该产生下列输出：

```
Hello world!
102
```

★★ 4. 编写一个程序，一行行地读取输入行，直至到达文件尾。算出每行输入行的长度，然后把最长的那行打印出来。为了简单起见，你可以假定所有的输入行均不超过 1000 个字符。

★★★ 5. rearrange 程序中的下列语句

```
if( columns[col] >= len ... )
        break;
```

当字符的列范围超出输入行的末尾时就停止复制。这条语句只有当列范围以递增顺序出现时才是正确的，但事实上并不一定如此。请修改这条语句，即使列范围不是按顺序读取时也能正确完成任务。

★★★ 6. 修改 rearrange 程序，去除输入中列标号的个数必须是偶数的限制。如果读入的列标号为奇数个，函数就会把最后一个列范围设置为最后一个列标号所指定的列到行尾之间的范围。从最后一个列标号直至行尾的所有字符都将被复制到输出字符串。