

第 10 章

数组和指针

本章介绍以下内容：

- 关键字：static
- 运算符：&、*（一元）
- 如何创建并初始化数组
- 指针（在已学过的基础上）、指针和数组的关系
- 编写处理数组的函数
- 二维数组

人们通常借助计算机完成统计每月的支出、日降雨量、季度销售额等任务。企业借助计算机管理薪资、库存和客户交易记录等。作为程序员，不可避免地要处理大量相关数据。通常，数组能高效便捷地处理这种数据。第 6 章简单地介绍了数组，本章将进一步地学习如何使用数组，着重分析如何编写处理数组的函数。这种函数把模块化编程的优势应用到数组。通过本章的学习，你将明白数组和指针关系密切。

10.1 数组

前面介绍过，数组由数据类型相同的一系列元素组成。需要使用数组时，通过声明数组告诉编译器数组中内含多少元素和这些元素的类型。编译器根据这些信息正确地创建数组。普通变量可以使用的类型，数组元素都可以用。考虑下面的数组声明：

```
/* 一些数组声明 */
int main(void)
{
    float candy[365];      /* 内含 365 个 float 类型元素的数组 */
    char code[12];          /* 内含 12 个 char 类型元素的数组 */
    int states[50];         /* 内含 50 个 int 类型元素的数组 */
    ...
}
```

方括号（[]）表明 candy、code 和 states 都是数组，方括号中的数字表明数组中的元素个数。

要访问数组中的元素，通过使用数组下标数（也称为索引）表示数组中的各元素。数组元素的编号从 0 开始，所以 candy[0] 表示 candy 数组的第一个元素，candy[364] 表示第 365 个元素，也就是最后一个元素。读者对这些内容应该比较熟悉，下面我们介绍一些新内容。

10.1.1 初始化数组

数组通常被用来储存程序需要的数据。例如，一个内含 12 个整数元素的数组可以储存 12 个月的天数。在这种情况下，在程序一开始就初始化数组比较好。下面介绍初始化数组的方法。

只储存单个值的变量有时也称为标量变量 (*scalar variable*)，我们已经很熟悉如何初始化这种变量：

```
int fix = 1;
float flax = PI * 2;
```

代码中的 PI 已定义为宏。C 使用新的语法来初始化数组，如下所示：

```
int main(void)
{
    int powers[8] = {1, 2, 4, 6, 8, 16, 32, 64}; /* 从 ANSI C 开始支持这种初始化 */
    ...
}
```

如上所示，用以逗号分隔的值列表（用花括号括起来）来初始化数组，各值之间用逗号分隔。在逗号和值之间可以使用空格。根据上面的初始化，把 1 赋给数组的首元素 (powers[0])，以此类推（不支持 ANSI 的编译器会把这种形式的初始化识别为语法错误，在数组声明前加上关键字 static 可解决此问题。第 12 章将详细讨论这个关键字）。

程序清单 10.1 演示了一个小程序，打印每个月的天数。

程序清单 10.1 day_mon1.c 程序

```
/* day_mon1.c -- 打印每个月的天数 */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    int index;

    for (index = 0; index < MONTHS; index++)
        printf("Month %2d has %2d days.\n", index + 1, days[index]);

    return 0;
}
```

该程序的输出如下：

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

这个程序还不够完善，每 4 年打错一个月份的天数（即，2 月份的天数）。该程序用初始化列表初始化 days[]，列表（用花括号括起来）中用逗号分隔各值。

注意该例使用了符号常量 MONTHS 表示数组大小，这是我们推荐且常用的做法。例如，如果要采用一年 13 个月的记法，只需修改#define 这行代码即可，不用在程序中查找所有使用过数组大小的地方。

注意 使用 `const` 声明数组

有时需要把数组设置为只读。这样，程序只能从数组中检索值，不能把新值写入数组。要创建只读数组，应该用 `const` 声明和初始化数组。因此，程序清单 10.1 中初始化数组应改成：

```
const int days[MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

这样修改后，程序在运行过程中就不能修改该数组中的内容。和普通变量一样，应该使用声明来初始化 `const` 数据，因为一旦声明为 `const`，便不能再给它赋值。明确了这一点，就可以在后面的例子中使用 `const` 了。

如果初始化数组失败怎么办？程序清单 10.2 演示了这种情况。

程序清单 10.2 `no_data.c` 程序

```
/* no_data.c -- 为初始化数组 */
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int no_data[SIZE]; /* 未初始化数组 */
    int i;

    printf("%2s%14s\n", "i", "no_data[i]");
    for (i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, no_data[i]);

    return 0;
}
```

该程序的输出如下（系统不同，输出的结果可能不同）：

```
i      no_data[i]
0          0
1      4204937
2      4219854
3      2147348480
```

使用数组前必须先初始化它。与普通变量类似，在使用数组元素之前，必须先给它们赋初值。编译器使用的值是内存相应位置上的现有值，因此，读者运行该程序后的输出会与该示例不同。

注意 存储类别警告

数组和其他变量类似，可以把数组创建成不同的存储类别 (*storage class*)。第 12 章将介绍存储类别的相关内容，现在只需记住：本章描述的数组属于自动存储类别，意思是这些数组在函数内部声明，并且声明时未使用关键字 `static`。到目前为止，本书所用的变量和数组都是自动存储类别。

在这里提到存储类别的原因是，不同的存储类别有不同的属性，所以不能把本章的内容推广到其他存储类别。对于一些其他存储类别的变量和数组，如果在声明时未初始化，编译器会自动把它们的值设置为 0。

初始化列表中的项数应与数组的大小一致。如果不一致会怎样？我们还是以上一个程序为例，但初始化列表中缺少两个元素，如程序清单 10.3 所示：

程序清单 10.3 somedata.c 程序

```
/* some_data.c -- 部分初始化数组 */
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int some_data[SIZE] = { 1492, 1066 };
    int i;

    printf("%2s%14s\n", "i", "some_data[i]");
    for (i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, some_data[i]);

    return 0;
}
```

下面是该程序的输出：

```
i some_data[i]
0      1492
1      1066
2      0
3      0
```

如上所示，编译器做得很好。当初始化列表中的值少于数组元素个数时，编译器会把剩余的元素都初始化为 0。也就是说，如果不初始化数组，数组元素和未初始化的普通变量一样，其中储存的都是垃圾值；但是，如果部分初始化数组，剩余的元素就会被初始化为 0。

如果初始化列表的项数多于数组元素个数，编译器可没那么仁慈，它会毫不留情地将其视为错误。但是，没必要因此嘲笑编译器。其实，可以省略方括号中的数字，让编译器自动匹配数组大小和初始化列表中的项数（见程序清单 10.4）

程序清单 10.4 day_mon2.c 程序

```
/* day_mon2.c -- 让编译器计算元素个数 */
#include <stdio.h>
int main(void)
{
    const int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31 };
    int index;

    for (index = 0; index < sizeof days / sizeof days[0]; index++)
        printf("Month %2d has %d days.\n", index + 1, days[index]);

    return 0;
}
```

在程序清单 10.4 中，要注意以下两点。

- 如果初始化数组时省略方括号中的数字，编译器会根据初始化列表中的项数来确定数组的大小。
- 注意 for 循环中的测试条件。由于人工计算容易出错，所以让计算机来计算数组的大小。sizeof 运算符给出它的运算对象的大小（以字节为单位）。所以 sizeof days 是整个数组的大小（以字节为单位），sizeof day[0] 是数组中一个元素的大小（以字节为单位）。整个数组的大小除以单个元素的大小就是数组元素的个数。

下面是该程序的输出：

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
```

我们的本意是防止初始化值的个数超过数组的大小，让程序找出数组大小。我们初始化时用了 10 个值，结果就只打印了 10 个值！这就是自动计数的弊端：无法察觉初始化列表中的项数有误。

还有一种初始化数组的方法，但这种方法仅限于初始化字符数组。我们在下一章中介绍。

10.1.2 指定初始化器 (C99)

C99 增加了一个新特性：指定初始化器 (*designated initializer*)。利用该特性可以初始化指定的数组元素。例如，只初始化数组中的最后一个元素。对于传统的 C 初始化语法，必须初始化最后一个元素之前的所有元素，才能初始化它：

```
int arr[6] = {0, 0, 0, 0, 0, 212}; // 传统的语法
```

而 C99 规定，可以在初始化列表中使用带方括号的下标指明待初始化的元素：

```
int arr[6] = {[5] = 212}; // 把 arr[5] 初始化为 212
```

对于一般的初始化，在初始化一个元素后，未初始化的元素都会被设置为 0。程序清单 10.5 中的初始化比较复杂。

程序清单 10.5 designate.c 程序

```
// designate.c -- 使用指定初始化器
#include <stdio.h>
#define MONTHS 12
int main(void)
{
    int days[MONTHS] = { 31, 28, [4] = 31, 30, 31, [1] = 29 };
    int i;

    for (i = 0; i < MONTHS; i++)
        printf("%2d %d\n", i + 1, days[i]);

    return 0;
}
```

该程序在支持 C99 的编译器中输出如下：

```
1 31
2 29
3 0
4 0
5 31
6 30
7 31
8 0
```

```
9    0
10   0
11   0
12   0
```

以上输出揭示了指定初始化器的两个重要特性。第一，如果指定初始化器后面有更多的值，如该例中的初始化列表中的片段：[4] = 31, 30, 31，那么后面这些值将被用于初始化指定元素后面的元素。也就是说，在 days[4] 被初始化为 31 后，days[5] 和 days[6] 将分别被初始化为 30 和 31。第二，如果再次初始化指定的元素，那么最后的初始化将会取代之前的初始化。例如，程序清单 10.5 中，初始化列表开始时把 days[1] 初始化为 28，但是 days[1] 又被后面的指定初始化 [1] = 29 初始化为 29。

如果未指定元素大小会怎样？

```
int stuff[] = {1, [6] = 23};           //会发生什么?
int staff[] = {1, [6] = 4, 9, 10}; //会发生什么?
```

编译器会把数组的大小设置为足够装得下初始化的值。所以，stuff 数组有 7 个元素，编号为 0~6；而 staff 数组的元素比 stuff 数组多两个（即有 9 个元素）。

10.1.3 给数组元素赋值

声明数组后，可以借助数组下标（或索引）给数组元素赋值。例如，下面的程序段给数组的所有元素赋值：

```
/* 给数组的元素赋值 */
#include <stdio.h>
#define SIZE 50
int main(void)
{
    int counter, evens[SIZE];

    for (counter = 0; counter < SIZE; counter++)
        evens[counter] = 2 * counter;
    ...
}
```

注意这段代码中使用循环给数组的元素依次赋值。C 不允许把数组作为一个单元赋给另一个数组，除初始化以外也不允许使用花括号列表的形式赋值。下面的代码段演示了一些错误的赋值形式：

```
/* 一些无效的数组赋值 */
#define SIZE 5
int main(void)
{
    int oxen[SIZE] = {5, 3, 2, 8};           /* 初始化没问题 */
    int yaks[SIZE];

    yaks = oxen;                          /* 不允许 */
    yaks[SIZE] = oxen[SIZE];            /* 数组下标越界 */
    yaks[SIZE] = {5, 3, 2, 8};          /* 不起作用 */
```

oxen 数组的最后一个元素是 oxen[SIZE-1]，所以 oxen[SIZE] 和 yaks[SIZE] 都超出了两个数组的末尾。

10.1.4 数组边界

在使用数组时，要防止数组下标超出边界。也就是说，必须确保下标是有效的值。例如，假设有下面的声明：

```
int doofi[20];
```

那么在使用该数组时，要确保程序中使用的数组下标在 0~19 的范围内，因为编译器不会检查出这种错误（但是，一些编译器发出警告，然后继续编译程序）。

考虑程序清单 10.6 的问题。该程序创建了一个内含 4 个元素的数组，然后错误地使用了 -1~6 的下标。

程序清单 10.6 bounds.c 程序

```
// bounds.c -- 数组下标越界
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int value1 = 44;
    int arr[SIZE];
    int value2 = 88;
    int i;

    printf("value1 = %d, value2 = %d\n", value1, value2);
    for (i = -1; i <= SIZE; i++)
        arr[i] = 2 * i + 1;

    for (i = -1; i < 7; i++)
        printf("%2d %d\n", i, arr[i]);
    printf("value1 = %d, value2 = %d\n", value1, value2);
    printf("address of arr[-1]: %p\n", &arr[-1]);
    printf("address of arr[4]: %p\n", &arr[4]);
    printf("address of value1: %p\n", &value1);
    printf("address of value2: %p\n", &value2);

    return 0;
}
```

编译器不会检查数组下标是否使用得当。在 C 标准中，使用越界下标的结果是未定义的。这意味着程序看上去可以运行，但是运行结果很奇怪，或异常中止。下面是使用 GCC 的输出示例：

```
value1 = 44, value2 = 88
-1 -1
0 1
1 3
2 5
3 7
4 9
5 1624678494
6 32767
value1 = 9, value2 = -1
address of arr[-1]: 0x7fff5fbff8cc
address of arr[4]: 0x7fff5fbff8e0
address of value1: 0x7fff5fbff8e0
address of value2: 0x7fff5fbff8cc
```

注意，该编译器似乎把 value2 储存在数组的前一个位置，把 value1 储存在数组的后一个位置（其他编译器在内存中储存数据的顺序可能不同）。在上面的输出中，arr[-1] 与 value2 对应的内存地址相同，arr[4] 和 value1 对应的内存地址相同。因此，使用越界的数组下标会导致程序改变其他变量的值。不同的编译器运行该程序的结果可能不同，有些会导致程序异常中止。

C 语言为何会允许这种麻烦事发生？这要归功于 C 信任程序员的原则。不检查边界，C 程序可以运行

更快。编译器没必要捕获所有的下标错误，因为在程序运行之前，数组的下标值可能尚未确定。因此，为安全起见，编译器必须在运行时添加额外代码检查数组的每个下标值，这会降低程序的运行速度。C 相信程序员能编写正确的代码，这样的程序运行速度更快。但并不是所有的程序员都能做到这一点，所以就出现了下标越界的问题。

还要记住一点：数组元素的编号从 0 开始。最好是在声明数组时使用符号常量来表示数组的大小：

```
#define SIZE 4
int main(void)
{
    int arr[SIZE];
    for (i = 0; i < SIZE; i++)
    ...
}
```

这样做能确保整个程序中的数组大小始终一致。

10.1.5 指定数组的大小

本章前面的程序示例都使用整型常量来声明数组：

```
#define SIZE 4
int main(void)
{
    int arr[SIZE];           // 整数符号常量
    double lots[144];        // 整数字面常量
    ...
}
```

在 C99 标准之前，声明数组时只能在方括号中使用整型常量表达式。所谓整型常量表达式，是由整型常量构成的表达式。`sizeof` 表达式被视为整型常量，但是（与 C++ 不同）`const` 值不是。另外，表达式的值必须大于 0：

```
int n = 5;
int m = 8;
float a1[5];           // 可以
float a2[5*2 + 1];      // 可以
float a3[sizeof(int) + 1]; // 可以
float a4[-4];           // 不可以，数组大小必须大于 0
float a5[0];             // 不可以，数组大小必须大于 0
float a6[2.5];           // 不可以，数组大小必须是整数
float a7[(int)2.5];       // 可以，已被强制转换为整型常量
float a8[n];              // C99 之前不允许
float a9[m];              // C99 之前不允许
```

上面的注释表明，以前支持 C90 标准的编译器不允许后两种声明方式。而 C99 标准允许这样声明，这创建了一种新型数组，称为变长数组 (*variable-length array*) 或简称 VLA (C11 放弃了这一创新的举措，把 VLA 设定为可选，而不是语言必备的特性)。

C99 引入变长数组主要是为了让 C 成为更好的数值计算语言。例如，VLA 简化了把 FORTRAN 现有的数值计算例程库转换为 C 代码的过程。VLA 有一些限制，例如，声明 VLA 时不能进行初始化。在充分了解经典的 C 数组后，我们再详细介绍 VLA。

10.2 多维数组

气象研究员 Tempest Cloud 为完成她的研究项目要分析 5 年内每个月的降水量数据，她首先要解决的问

题是如何表示数据。一个方案是创建 60 个变量，每个变量储存一个数据项（我们曾经提到过这一笨拙的方案，和以前一样，这个方案并不合适）。使用一个内含 60 个元素的数组比将建 60 个变量好，但是如果能把各年的数据分开储存会更好，即创建 5 个数组，每个数组 12 个元素。然而，这样做也很麻烦，如果 Tempest 决定研究 50 年的降水量，岂不是要创建 50 个数组。是否能有更好的方案？

处理这种情况应该使用数组的数组。主数组（*master array*）有 5 个元素（每个元素表示一年），每个元素是内含 12 个元素的数组（每个元素表示一个月）。下面是该数组的声明：

```
float rain[5][12]; // 内含 5 个数组元素的数组，每个数组元素内含 12 个 float 类型的元素
```

理解该声明的一种方法是，先查看中间部分（粗体部分）：

```
float rain[5][12]; // rain 是一个内含 5 个元素的数组
```

这说明数组 rain 有 5 个元素，至于每个元素的情况，要查看声明的其余部分（粗体部分）：

```
float rain[5][12] ; // 一个内含 12 个 float 类型元素的数组
```

这说明每个元素的类型是 float[12]，也就是说，rain 的每个元素本身都是一个内含 12 个 float 类型值的数组。

根据以上分析可知，rain 的首元素 rain[0] 是一个内含 12 个 float 类型值的数组。所以，rain[1]、rain[2] 等也是如此。如果 rain[0] 是一个数组，那么它的首元素就是 rain[0][0]，第 2 个元素是 rain[0][1]，以此类推。简而言之，数组 rain 有 5 个元素，每个元素都是内含 12 个 float 类型元素的数组，rain[0] 是内含 12 个 float 值的数组，rain[0][0] 是一个 float 类型的值。假设要访问位于 2 行 3 列的值，则使用 rain[2][3]（记住，数组元素的编号从 0 开始，所以 2 行指的是第 3 行）。

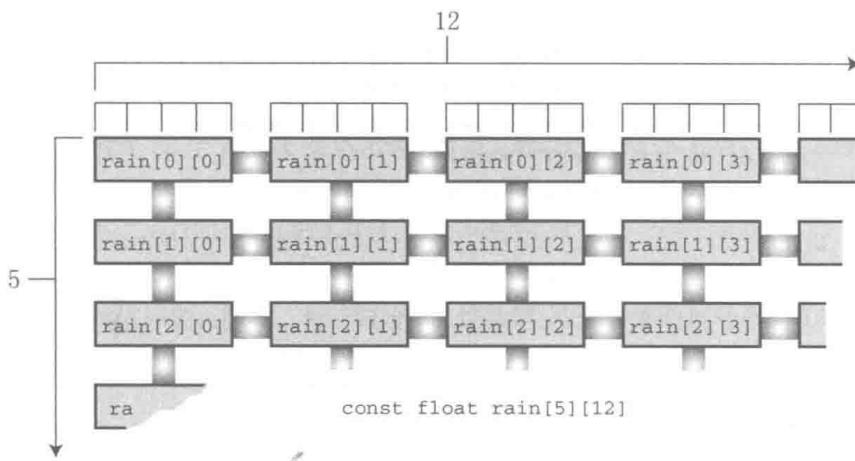


图 10.1 二维数组

该二维视图有助于帮助读者理解二维数组的两个下标。在计算机内部，这样的数组是按顺序储存的，从第 1 个内含 12 个元素的数组开始，然后是第 2 个内含 12 个元素的数组，以此类推。

我们要在气象分析程序中用到这个二维数组。该程序的目标是，计算每年的总降水量、年平均降水量和月平均降水量。要计算年总降水量，必须对一行数据求和；要计算某月份的平均降水量，必须对一列数据求和。二维数组很直观，实现这些操作也很容易。程序清单 10.7 演示了这个程序。

程序清单 10.7 rain.c 程序

```
/* rain.c -- 计算每年的总降水量、年平均降水量和 5 年中每月的平均降水量 */
#include <stdio.h>
#define MONTHS 12      // 一年的月份数
#define YEARS   5       // 年数
```

```

int main(void)
{
    // 用 2010~2014 年的降水量数据初始化数组
    const float rain[YEARS][MONTHS] =
    {
        { 4.3, 4.3, 4.3, 3.0, 2.0, 1.2, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6 },
        { 8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.3 },
        { 9.1, 8.5, 6.7, 4.3, 2.1, 0.8, 0.2, 0.2, 1.1, 2.3, 6.1, 8.4 },
        { 7.2, 9.9, 8.4, 3.3, 1.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 6.2 },
        { 7.6, 5.6, 3.8, 2.8, 3.8, 0.2, 0.0, 0.0, 0.0, 1.3, 2.6, 5.2 }
    };
    int year, month;
    float subtotal, total;

    printf(" YEAR      RAINFALL  (inches)\n");
    for (year = 0, total = 0; year < YEARS; year++)
    {
        // 每一年，各月的降水量总和
        for (month = 0, subtotal = 0; month < MONTHS; month++)
            subtotal += rain[year][month];
        printf("%5d %15.1f\n", 2010 + year, subtotal);
        total += subtotal; // 5 年的总降水量
    }
    printf("\nThe yearly average is %.1f inches.\n\n", total / YEARS);
    printf("MONTHLY AVERAGES:\n\n");
    printf(" Jan Feb Mar Apr May Jun Jul Aug Sep Oct ");
    printf(" Nov Dec\n");

    for (month = 0; month < MONTHS; month++)
    {
        // 每个月，5 年的总降水量
        for (year = 0, subtotal = 0; year < YEARS; year++)
            subtotal += rain[year][month];
        printf("%4.1f ", subtotal / YEARS);
    }
    printf("\n");
}

return 0;
}

```

下面是该程序的输出：

YEAR	RAINFALL (inches)
2010	32.4
2011	37.9
2012	49.8
2013	44.0
2014	32.9

The yearly average is 39.4 inches.

MONTHLY AVERAGES:

Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
7.3	7.3	4.9	3.0	2.3	0.6	1.2	0.3	0.5	1.7	3.6	6.7

学习该程序的重点是数组初始化和计算方案。初始化二维数组比较复杂，我们先来看较为简单的计算部分。

程序使用了两个嵌套 `for` 循环。第 1 个嵌套 `for` 循环的内层循环，在 `year` 不变的情况下，遍历 `month` 计算某年的总降水量；而外层循环，改变 `year` 的值，重复遍历 `month`，计算 5 年的总降水量。这种嵌套循环结构常用于处理二维数组，一个循环处理数组的第 1 个下标，另一个循环处理数组的第 2 个下标：

```
for (year = 0, total = 0; year < YEARS; year++)
{ // 处理每一年的数据
    for (month = 0, subtotal = 0; month < MONTHS; month++)
        ... // 处理每月的数据
        ... // 处理每一年的数据
}
```

第 2 个嵌套 `for` 循环和第 1 个的结构相同，但是内层循环遍历 `year`，外层循环遍历 `month`。记住，每执行一次外层循环，就完整遍历一次内层循环。因此，在改变月份之前，先遍历完年，得到某月 5 年间的平均降水量，以此类推：

```
for (month = 0; month < MONTHS; month++)
{ // 处理每月的数据
    for (year = 0, subtotal = 0; year < YEARS; year++)
        ... // 处理每年的数据
        ... // 处理每月的数据
}
```

10.2.1 初始化二维数组

初始化二维数组是建立在初始化一维数组的基础上。首先，初始化一维数组如下：

```
sometype ar1[5] = {val1, val2, val3, val4, val5};
```

这里，`val1`、`val2` 等表示 `sometype` 类型的值。例如，如果 `sometype` 是 `int`，那么 `val1` 可能是 7；如果 `sometype` 是 `double`，那么 `val1` 可能是 11.34，诸如此类。但是 `rain` 是一个内含 5 个元素的数组，每个元素又是内含 12 个 `float` 类型元素的数组。所以，对 `rain` 而言，`val1` 应该包含 12 个值，用于初始化内含 12 个 `float` 类型元素的一维数组，如下所示：

```
{4.3, 4.3, 4.3, 3.0, 2.0, 1.2, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6}
```

也就是说，如果 `sometype` 是一个内含 12 个 `double` 类型元素的数组，那么 `val1` 就是一个由 12 个 `double` 类型值构成的数值列表。因此，为了初始化二维数组 `rain`，要用逗号分隔 5 个这样的数值列表：

```
const float rain[YEARS][MONTHS] =
{
    {4.3, 4.3, 4.3, 3.0, 2.0, 1.2, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6},
    {8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.3},
    {9.1, 8.5, 6.7, 4.3, 2.1, 0.8, 0.2, 0.2, 1.1, 2.3, 6.1, 8.4},
    {7.2, 9.9, 8.4, 3.3, 1.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 6.2},
    {7.6, 5.6, 3.8, 2.8, 3.8, 0.2, 0.0, 0.0, 0.0, 1.3, 2.6, 5.2}
};
```

这个初始化使用了 5 个数值列表，每个数值列表都用花括号括起来。第 1 个列表的数据用于初始化数组的第 1 行，第 2 个列表的数据用于初始化数组的第 2 行，以此类推。前面讨论的数据个数和数组大小不匹配的问题同样适用于这里的每一行。也就是说，如果第 1 个列表中只有 10 个数，则只会初始化数组第 1 行的前 10 个元素，而最后两个元素将被默认初始化为 0。如果某列表中的数值个数超出了数组每行的元素个数，则会出错，但是这并不会影响其他行的初始化。

初始化时也可省略内部的花括号，只保留最外面的一对花括号。只要保证初始化的数值个数正确，初始化的效果与上面相同。但是如果初始化的数值不够，则按照先后顺序逐行初始化，直到用完所有的值。后面没有值初始化的元素被统一初始化为 0。图 10.2 演示了这种初始化数组的方法。

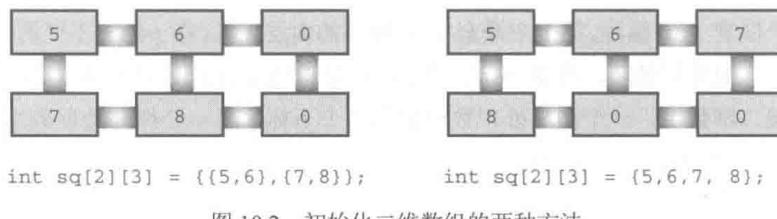


图 10.2 初始化二维数组的两种方法

因为储存在数组 rain 中的数据不能修改，所以程序使用了 `const` 关键字声明该数组。

10.2.2 其他多维数组

前面讨论的二维数组的相关内容都适用于三维数组或更多维的数组。可以这样声明一个三维数组：

```
int box[10][20][30];
```

可以把一维数组想象成一行数据，把二维数组想象成数据表，把三维数组想象成一叠数据表。例如，把上面声明的三维数组 `box` 想象成由 10 个二维数组（每个二维数组都是 20 行 30 列）堆叠起来。

还有一种理解 `box` 的方法是，把 `box` 看作数组的数组。也就是说，`box` 内含 10 个元素，每个元素是内含 20 个元素的数组，这 20 个数组元素中的每个元素是内含 30 个元素的数组。或者，可以简单地根据所需的下标值去理解数组。

通常，处理三维数组要使用 3 重嵌套循环，处理四维数组要使用 4 重嵌套循环。对于其他多维数组，以此类推。在后面的程序示例中，我们只使用二维数组。

10.3 指针和数组

第 9 章介绍过指针，指针提供一种以符号形式使用地址的方法。因为计算机的硬件指令非常依赖地址，指针在某种程度上把程序员想要传达的指令以更接近机器的方式表达。因此，使用指针的程序更有效率。尤其是，指针能有效地处理数组。我们很快就会学到，数组表示法其实是在变相地使用指针。

我们举一个变相使用指针的例子：数组名是数组首元素的地址。也就是说，如果 `flizny` 是一个数组，下面的语句成立：

```
flizny == &flizny[0]; // 数组名是该数组首元素的地址
```

`flizny` 和 `&flizny[0]` 都表示数组首元素的内存地址（`&` 是地址运算符）。两者都是常量，在程序的运行过程中，不会改变。但是，可以把它们赋值给指针变量，然后可以修改指针变量的值，如程序清单 10.8 所示。注意指针加上一个数时，它的值发生了什么变化（转换说明 `%p` 通常以十六进制显示指针的值）。

程序清单 10.8 pnt_add.c 程序

```
// pnt_add.c -- 指针地址
#include <stdio.h>
#define SIZE 4
int main(void)
{
    short dates[SIZE];
    short * pti;
    short index;
    double bills[SIZE];
    double * ptf;
    pti = dates; // 把数组地址赋给指针
    ptf = bills;
```

```

printf("%23s %15s\n", "short", "double");
for (index = 0; index < SIZE; index++)
    printf("pointers + %d: %10p %10p\n", index, pti + index, ptf + index);
return 0;
}

```

下面是该例的输出示例：

```

short           double
pointers + 0: 0xffff5fbff8dc 0xffff5fbff8a0
pointers + 1: 0xffff5fbff8de 0xffff5fbff8a8
pointers + 2: 0xffff5fbff8e0 0xffff5fbff8b0
pointers + 3: 0xffff5fbff8e2 0xffff5fbff8b8

```

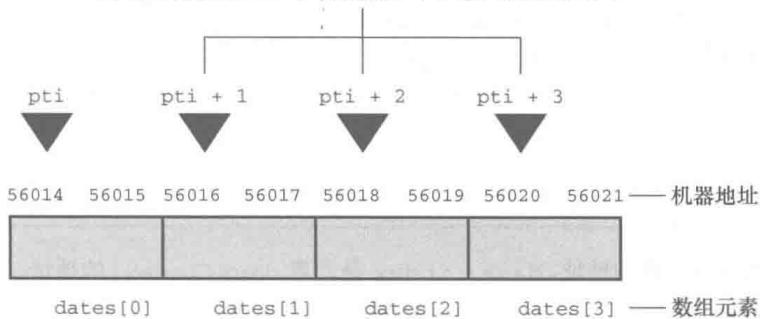
第 2 行打印的是两个数组开始的地址，下一行打印的是指针加 1 后的地址，以此类推。注意，地址是十六进制的，因此 dd 比 dc 大 1，a1 比 a0 大 1。但是，显示的地址是怎么回事？

0xffff5fbff8dc + 1 是否是 0xffff5fbff8de?

0xffff5fbff8a0 + 1 是否是 0xffff5fbff8a8?

我们的系统中，地址按字节编号，short 类型占用 2 字节，double 类型占用 8 字节。在 C 中，指针加 1 指的是增加一个存储单元。对数组而言，这意味着把加 1 后的地址是下一个元素的地址，而不是下一个字节的地址（见图 10.3）。这是为什么必须声明指针所指向对象类型的原因之一。只知道地址不够，因为计算机要知道储存对象需要多少字节（即使指针指向的是标量变量，也要知道变量的类型，否则 *pt 就无法正确地取回地址上的值）。

因为 pti 的类型是 short，所以指针 i，其值每次递增 2 字节



```

int dates[y], *pti;
pti = dates; (or pti = & dates[0];)

```

把数组dates首元素的地址
赋给指针变量pti

图 10.3 数组和指针加法

现在可以更清楚地定义指向 int 的指针、指向 float 的指针，以及指向其他数据对象的指针。

- 指针的值是它所指向对象的地址。地址的表示方式依赖于计算机内部的硬件。许多计算机（包括 PC 和 Macintosh）都是按字节编址，意思是内存中的每个字节都按顺序编号。这里，一个较大对象的地址（如 double 类型的变量）通常是该对象第一个字节的地址。
- 在指针前面使用 * 运算符可以得到该指针所指向对象的值。
- 指针加 1，指针的值递增它所指向类型的大小（以字节为单位）。

下面的等式体现了 C 语言的灵活性：

```
dates + 2 == &date[2]      // 相同的地址
*(dates + 2) == dates[2]   // 相同的值
```

以上关系表明了数组和指针的关系十分密切，可以使用指针标识数组的元素和获得元素的值。从本质上讲，同一个对象有两种表示法。实际上，C 语言标准在描述数组表示法时确实借助了指针。也就是说，定义 `ar[n]` 的意思是`* (ar + n)`。可以认为`* (ar + n)` 的意思是“到内存的 `ar` 位置，然后移动 `n` 个单元，检索储存在那里的值”。

顺带一提，不要混淆 `*(dates+2)` 和 `*dates+2`。间接运算符（`*`）的优先级高于 `+`，所以 `*dates+2` 相当于 `(*dates)+2`：

```
*(dates + 2) // dates 第 3 个元素的值
*dates + 2    // dates 第 1 个元素的值加 2
```

明白了数组和指针的关系，便可在编写程序时适时使用数组表示法或指针表示法。运行程序清单 10.9 后输出的结果和程序清单 10.1 输出的结果相同。

程序清单 10.9 day_mon3.c 程序

```
/* day_mon3.c -- uses pointer notation */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    int index;

    for (index = 0; index < MONTHS; index++)
        printf("Month %2d has %d days.\n", index + 1,
               *(days + index)); //与 days[index] 相同

    return 0;
}
```

这里，`days` 是数组首元素的地址，`days + index` 是元素 `days[index]` 的地址，而 `*(days + index)` 则是该元素的值，相当于 `days[index]`。`for` 循环依次引用数组中的每个元素，并打印各元素的内容。

这样编写程序是否有优势？不一定。编译器编译这两种写法生成的代码相同。程序清单 10.9 要注意的是，指针表示法和数组表示法是两种等效的方法。该例演示了可以用指针表示数组，反过来，也可以用数组表示指针。在使用以数组为参数的函数时要注意这点。

10.4 函数、数组和指针

假设要编写一个处理数组的函数，该函数返回数组中所有元素之和，待处理的是名为 `marbles` 的 `int` 类型数组。应该如何调用该函数？也许是下面这样：

```
total = sum(marbles); // 可能的函数调用
```

那么，该函数的原型是什么？记住，数组名是该数组首元素的地址，所以实际参数 `marbles` 是一个储存 `int` 类型值的地址，应把它赋给一个指针形式参数，即该形参是一个指向 `int` 的指针：

```
int sum(int * ar); // 对应的函数原型
```

`sum()` 从该参数获得了什么信息？它获得了该数组首元素的地址，知道要在该位置上找出一个整数。

注意，该参数并未包含数组元素个数的信息。我们有两种方法让函数获得这一信息。第一种方法是，在函数代码中写上固定的数组大小：

```
int sum(int * ar) // 相应的函数定义
{
    int i;
    int total = 0;

    for (i = 0; i < 10; i++) // 假设数组有 10 个元素
        total += ar[i]; // ar[i] 与 *(ar + i) 相同
    return total;
}
```

既然能使用指针表示数组名，也可以用数组名表示指针。另外，回忆一下，`+=` 运算符把右侧运算对象加到左侧运算对象上。因此，`total` 是当前数组元素之和。

该函数定义有限制，只能计算 10 个 `int` 类型的元素。另一个比较灵活的方法是把数组大小作为第 2 个参数：

```
int sum(int * ar, int n) // 更通用的方法
{
    int i;
    int total = 0;

    for (i = 0; i < n; i++) // 使用 n 个元素
        total += ar[i]; // ar[i] 和 *(ar + i) 相同
    return total;
}
```

这里，第 1 个形参告诉函数该数组的地址和数据类型，第 2 个形参告诉函数该数组中元素的个数。

关于函数的形参，还有一点要注意。只有在函数原型或函数定义头中，才可以用 `int ar[]` 代替 `int * ar`：

```
int sum (int ar[], int n);
```

`int *ar` 形式和 `int ar[]` 形式都表示 `ar` 是一个指向 `int` 的指针。但是，`int ar[]` 只能用于声明形式参数。第 2 种形式 (`int ar[]`) 提醒读者指针 `ar` 指向的不仅仅一个 `int` 类型值，还是一个 `int` 类型数组的元素。

注意 声明数组形参

因为数组名是该数组首元素的地址，作为实际参数的数组名要求形式参数是一个与之匹配的指针。只有在这种情况下，C 才会把 `int ar[]` 和 `int * ar` 解释成一样。也就是说，`ar` 是指向 `int` 的指针。由于函数原型可以省略参数名，所以下面 4 种原型都是等价的：

```
int sum(int *ar, int n);
int sum(int *, int);
int sum(int ar[], int n);
int sum(int [], int);
```

但是，在函数定义中不能省略参数名。下面两种形式的函数定义等价：

```
int sum(int *ar, int n)
{
    // 其他代码已省略
}

int sum(int ar[], int n);
{
```

```
// 其他代码已省略
}

可以使用以上提到的任意一种函数原型和函数定义。
```

程序清单 10.10 演示了一个程序，使用 sum() 函数。该程序打印原始数组的大小和表示该数组的函数形参的大小（如果你的编译器不支持用转换说明%zd 打印 sizeof 返回值，可以用%u 或%lu 来代替）。

程序清单 10.10 sum_arr1.c 程序

```
// sum_arr1.c -- 数组元素之和
// 如果编译器不支持 %zd, 用 %u 或 %lu 替换它
#include <stdio.h>
#define SIZE 10
int sum(int ar[], int n);
int main(void)
{
    int marbles[SIZE] = { 20, 10, 5, 39, 4, 16, 19, 26, 31, 20 };
    long answer;

    answer = sum(marbles, SIZE);
    printf("The total number of marbles is %ld.\n", answer);
    printf("The size of marbles is %zd bytes.\n",
           sizeof marbles);

    return 0;
}

int sum(int ar[], int n)      // 这个数组的大小是?
{
    int i;
    int total = 0;

    for (i = 0; i < n; i++)
        total += ar[i];
    printf("The size of ar is %zd bytes.\n", sizeof ar);

    return total;
}
```

该程序的输出如下：

```
The size of ar is 8 bytes.
The total number of marbles is 190.
The size of marbles is 40 bytes.
```

注意，marbles 的大小是 40 字节。这没问题，因为 marbles 内含 10 个 int 类型的值，每个值占 4 字节，所以整个 marbles 的大小是 40 字节。但是，ar 才 8 字节。这是因为 ar 并不是数组本身，它是一个指向 marbles 数组首元素的指针。我们的系统中用 8 字节储存地址，所以指针变量的大小是 8 字节（其他系统中地址的大小可能不是 8 字节）。简而言之，在程序清单 10.10 中，marbles 是一个数组，ar 是一个指向 marbles 数组首元素的指针，利用 C 中数组和指针的特殊关系，可以用数组表示法来表示指针 ar。

10.4.1 使用指针形参

函数要处理数组必须知道何时开始、何时结束。`sum()` 函数使用一个指针形参标识数组的开始，用一个整数形参表明待处理数组的元素个数（指针形参也表明了数组中的数据类型）。但是这并不是给函数传递必备信息的唯一方法。还有一种方法是传递两个指针，第 1 个指针指明数组的开始处（与前面用法相同），第 2 个指针指明数组的结束处。程序清单 10.11 演示了这种方法，同时该程序也表明了指针形参是变量，这意味着可以用索引表明访问数组中的哪一个元素。

程序清单 10.11 `sum_arr2.c` 程序

```
/* sum_arr2.c -- 数组元素之和 */
#include <stdio.h>
#define SIZE 10
int sump(int * start, int * end);
int main(void)
{
    int marbles[SIZE] = { 20, 10, 5, 39, 4, 16, 19, 26, 31, 20 };
    long answer;

    answer = sump(marbles, marbles + SIZE);
    printf("The total number of marbles is %ld.\n", answer);

    return 0;
}

/* 使用指针算法 */
int sump(int * start, int * end)
{
    int total = 0;

    while (*start < *end)
    {
        total += *start; // 把数组元素的值加起来
        start++; // 让指针指向下一个元素
    }

    return total;
}
```

指针 `start` 开始指向 `marbles` 数组的首元素，所以赋值表达式 `total += *start` 把首元素（20）加给 `total`。然后，表达式 `start++` 递增指针变量 `start`，使其指向数组的下一个元素。因为 `start` 是指向 `int` 的指针，`start` 递增 1 相当于其值递增 `int` 类型的大小。

注意，`sum()` 函数用另一种方法结束加法循环。`sum()` 函数把元素的个数作为第 2 个参数，并把该参数作为循环测试的一部分：

```
for( i = 0; i < n; i++)
```

而 `sum()` 函数则使用第 2 个指针来结束循环：

```
while (*start < *end)
```

因为 `while` 循环的测试条件是一个不相等的关系，所以循环最后处理的一个元素是 `end` 所指向位置的前一个元素。这意味着 `end` 指向的位置实际上在数组最后一个元素的后面。`C` 保证在给数组分配空间时，

指向数组后面第一个位置的指针仍是有效的指针。这使得 while 循环的测试条件是有效的，因为 start 在循环中最后的值是 end¹。注意，使用这种“越界”指针的函数调用更为简洁：

```
answer = sump (marbles, marbles + SIZE);
```

因为下标从 0 开始，所以 marbles + SIZE 指向数组末尾的下一个位置。如果 end 指向数组的最后一个元素而不是数组末尾的下一个位置，则必须使用下面的代码：

```
answer = sump (marbles, marbles + SIZE - 1);
```

这种写法既不简洁也不好记，很容易导致编程错误。顺带一提，虽然 C 保证了 marbles + SIZE 有效，但是对 marbles [SIZE]（即储存在该位置上的值）未作任何保证，所以程序不能访问该位置。

还可以把循环体压缩成一行代码：

```
total += *start++;
```

一元运算符*和++的优先级相同，但结合律是从右往左，所以 start++先求值，然后才是*start。也就是说，指针 start 先递增后指向。使用后缀形式（即 start++而不是++start）意味着先把指针指向位置上的值加到 total 上，然后再递增指针。如果使用*++start，顺序则反过来，先递增指针，再使用指针指向位置上的值。如果使用(*start)++，则先使用 start 指向的值，再递增该值，而不是递增指针。这样，指针将一直指向同一个位置，但是该位置上的值发生了变化。虽然*start++的写法比较常用，但是*(start++)这样写更清楚。程序清单 10.12 的程序演示了这些优先级的情况。

程序清单 10.12 order.c 程序

```
/* order.c -- 指针运算中的优先级 */
#include <stdio.h>
int data[2] = { 100, 200 };
int moredata[2] = { 300, 400 };
int main(void)
{
    int * p1, *p2, *p3;

    p1 = p2 = data;
    p3 = moredata;
    printf(" *p1 = %d,      *p2 = %d,      *p3 = %d\n", *p1, *p2, *p3);
    printf(" *p1++ = %d,   ++p2 = %d,   (*p3)++ = %d\n", *p1, ++p2, (*p3)++);
    printf(" *p1 = %d,      *p2 = %d,      *p3 = %d\n", *p1, *p2, *p3);

    return 0;
}
```

下面是该程序的输出：

```
*p1 = 100,      *p2 = 100,      *p3 = 300
*p1++ = 100,   ++p2 = 200,   (*p3)++ = 300
*p1 = 200,      *p2 = 200,      *p3 = 301
```

只有 (*p3)++ 改变了数组元素的值，其他两个操作分别把 p1 和 p2 指向数组的下一个元素。

10.4.2 指针表示法和数组表示法

从以上分析可知，处理数组的函数实际上用指针作为参数，但是在编写这样的函数时，可以选择是使用数组表示法还是指针表示法。如程序清单 10.10 所示，使用数组表示法，让函数是处理数组的这一意图更

¹ 在最后一次 while 循环中执行完 start++; 后，start 的值就是 end 的值。——译者注

加明显。另外，许多其他语言的程序员对数组表示法更熟悉，如 FORTRAN、Pascal、Modula-2 或 BASIC。其他程序员可能更习惯使用指针表示法，觉得使用指针更自然，如程序清单 10.11 所示。

至于 C 语言，`ar[i]` 和 `*(ar+1)` 这两个表达式都是等价的。无论 `ar` 是数组名还是指针变量，这两个表达式都没问题。但是，只有当 `ar` 是指针变量时，才能使用 `ar++` 这样的表达式。

指针表示法（尤其与递增运算符一起使用时）更接近机器语言，因此一些编译器在编译时能生成效率更高的代码。然而，许多程序员认为他们的主要任务是确保代码正确、逻辑清晰，而代码优化应该留给编译器去做。

10.5 指针操作

可以对指针进行哪些操作？C 提供了一些基本的指针操作，下面的程序示例中演示了 8 种不同的操作。为了显示每种操作的结果，该程序打印了指针的值（该指针指向的地址）、储存在指针指向地址上的值，以及指针自己的地址。如果编译器不支持 `%p` 转换说明，可以用 `%u` 或 `%lu` 代替 `%p`；如果编译器不支持用 `%td` 转换说明打印地址的差值，可以用 `%d` 或 `%ld` 来代替。

程序清单 10.13 演示了指针变量的 8 种基本操作。除了这些操作，还可以使用关系运算符来比较指针。

程序清单 10.13 `ptr_ops.c` 程序

```
// ptr_ops.c -- 指针操作
#include <stdio.h>
int main(void)
{
    int urn[5] = { 100, 200, 300, 400, 500 };
    int *ptr1, *ptr2, *ptr3;

    ptr1 = urn;           // 把一个地址赋给指针
    ptr2 = &urn[2];       // 把一个地址赋给指针
                        // 解引用指针，以及获得指针的地址
    printf("pointer value, dereferenced pointer, pointer address:\n");
    printf("ptr1 = %p, *ptr1 = %d, &ptr1 = %p\n", ptr1, *ptr1, &ptr1);

    // 指针加法
    ptr3 = ptr1 + 4;
    printf("\nadding an int to a pointer:\n");
    printf("ptr1 + 4 = %p, *(ptr1 + 4) = %d\n", ptr1 + 4, *(ptr1 + 4));
    ptr1++;             // 递增指针
    printf("\nvalues after ptr1++:\n");
    printf("ptr1 = %p, *ptr1 = %d, &ptr1 = %p\n", ptr1, *ptr1, &ptr1);
    ptr2--;             // 递减指针
    printf("\nvalues after --ptr2:\n");
    printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n", ptr2, *ptr2, &ptr2);
    --ptr1;              // 恢复为初始值
    ++ptr2;              // 恢复为初始值
    printf("\nPointers reset to original values:\n");
    printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);
    // 一个指针减去另一个指针
    printf("\nsubtracting one pointer from another:\n");
    printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %td\n", ptr2, ptr1, ptr2 - ptr1);
```

```

// 一个指针减去一个整数
printf("\nsubtracting an int from a pointer:\n");
printf("ptr3 = %p, ptr3 - 2 = %p\n", ptr3, ptr3 - 2);

return 0;
}

```

下面是我们的系统运行该程序后的输出：

```

pointer value, dereferenced pointer, pointer address:
ptr1 = 0x7fff5fbff8d0, *ptr1 = 100, &ptr1 = 0x7fff5fbff8c8

adding an int to a pointer:
ptr1 + 4 = 0x7fff5fbff8e0, *(ptr1 + 4) = 500

values after ptr1++:
ptr1 = 0x7fff5fbff8d4, *ptr1 = 200, &ptr1 = 0x7fff5fbff8c8

values after --ptr2:
ptr2 = 0x7fff5fbff8d4, *ptr2 = 200, &ptr2 = 0x7fff5fbff8c0

Pointers reset to original values:
ptr1 = 0x7fff5fbff8d0, ptr2 = 0x7fff5fbff8d8

subtracting one pointer from another:
ptr2 = 0x7fff5fbff8d8, ptr1 = 0x7fff5fbff8d0, ptr2 - ptr1 = 2

subtracting an int from a pointer:
ptr3 = 0x7fff5fbff8e0, ptr3 - 2 = 0x7fff5fbff8d8

```

下面分别描述了指针变量的基本操作。

- **赋值：**可以把地址赋给指针。例如，用数组名、带地址运算符（`&`）的变量名、另一个指针进行赋值。在该例中，把 `urn` 数组的首地址赋给了 `ptr1`，该地址的编号恰好是 `0x7fff5fbff8d0`。变量 `ptr2` 获得数组 `urn` 的第 3 个元素 (`urn[2]`) 的地址。注意，地址应该和指针类型兼容。也就是说，不能把 `double` 类型的地址赋给指向 `int` 的指针，至少要避免不明智的类型转换。C99/C11 已经强制不允许这样做。
- **解引用：**`*` 运算符给出指针指向地址上储存的值。因此，`*ptr1` 的初值是 100，该值储存在编号为 `0x7fff5fbff8d0` 的地址上。
- **取址：**和所有变量一样，指针变量也有自己的地址和值。对指针而言，`&` 运算符给出指针本身的地位。本例中，`ptr1` 储存在内存编号为 `0x7fff5fbff8c8` 的地址上，该存储单元储存的内容是 `0x7fff5fbff8d0`，即 `urn` 的地址。因此 `&ptr1` 是指向 `ptr1` 的指针，而 `ptr1` 是指向 `urn[0]` 的指针。
- **指针与整数相加：**可以使用 `+` 运算符把指针与整数相加，或整数与指针相加。无论哪种情况，整数都会和指针所指向类型的大小（以字节为单位）相乘，然后把结果与初始地址相加。因此 `ptr1 + 4` 与 `&urn[4]` 等价。如果相加的结果超出了初始指针指向的数组范围，计算结果则是未定义的。除非正好超过数组末尾第一个位置，C 保证该指针有效。
- **递增指针：**递增指向数组元素的指针可以让该指针移动至数组的下一个元素。因此，`ptr1++` 相当于把 `ptr1` 的值加上 4（我们的系统中 `int` 为 4 字节），`ptr1` 指向 `urn[1]`（见图 10.4，该图中使用了简化的地址）。现在 `ptr1` 的值是 `0x7fff5fbff8d4`（数组的下一个元素的地址），`*ptr` 的值为

200（即 `urn[1]` 的值）。注意，`ptr1` 本身的地址仍是 `0x7fff5fbff8c8`。毕竟，变量不会因为值发生变化就移动位置。

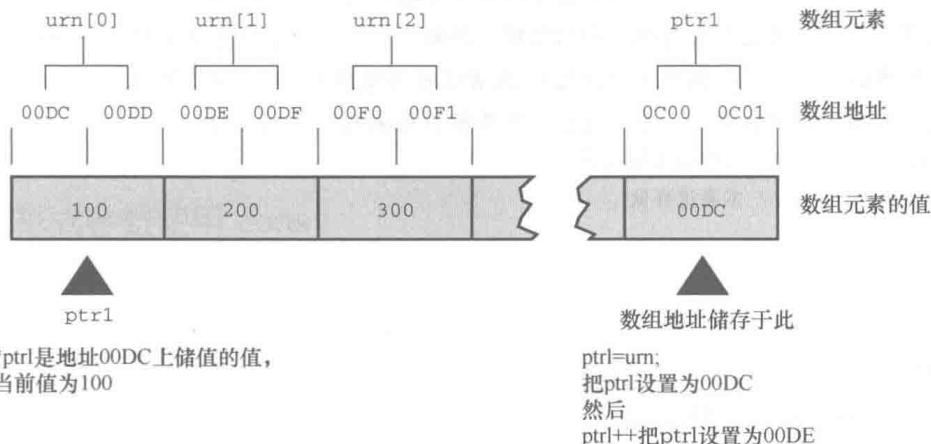


图 10.4 递增指向 int 的指针

- **指针减去一个整数：**可以使用 - 运算符从一个指针中减去一个整数。指针必须是第 1 个运算对象，整数是第 2 个运算对象。该整数将乘以指针指向类型的大小（以字节为单位），然后用初始地址减去乘积。所以 `ptr3 - 2` 与 `&urn[2]` 等价，因为 `ptr3` 指向的是 `&urn[4]`。如果相减的结果超出了初始指针所指向数组的范围，计算结果则是未定义的。除非正好超过数组末尾第一个位置，C 保证该指针有效。
- **递减指针：**当然，除了递增指针还可以递减指针。在本例中，递减 `ptr3` 使其指向数组的第 2 个元素而不是第 3 个元素。前缀或后缀的递增和递减运算符都可以使用。注意，在重置 `ptr1` 和 `ptr2` 前，它们都指向相同的元素 `urn[1]`。
- **指针求差：**可以计算两个指针的差值。通常，求差的两个指针分别指向同一个数组的不同元素，通过计算求出两元素之间的距离。差值的单位与数组类型的单位相同。例如，程序清单 10.13 的输出中，`ptr2 - ptr1` 得 2，意思是这两个指针所指向的两个元素相隔两个 int，而不是 2 字节。只要两个指针都指向相同的数组（或者其中一个指针指向数组后面的第 1 个地址），C 都能保证相减运算有效。如果指向两个不同数组的指针进行求差运算可能会得出一个值，或者导致运行时错误。
- **比较：**使用关系运算符可以比较两个指针的值，前提是两个指针都指向相同类型的对象。

注意，这里的减法有两种。可以用一个指针减去另一个指针得到一个整数，或者用一个指针减去一个整数得到另一个指针。

在递增或递减指针时还要注意一些问题。编译器不会检查指针是否仍指向数组元素。C 只能保证指向数组任意元素的指针和指向数组后面第 1 个位置的指针有效。但是，如果递增或递减一个指针后超出了这个范围，则是未定义的。另外，可以解引用指向数组任意元素的指针。但是，即使指针指向数组后面一个位置是有效的，也能解引用这样的越界指针。

解引用未初始化的指针

说到注意事项，一定要牢记一点：千万不要解引用未初始化的指针。例如，考虑下面的例子：

```
int * pt; // 未初始化的指针
*pt = 5;      // 严重的错误
```

为何不行？第 2 行的意思是把 5 储存在 pt 指向的位置。但是 pt 未被初始化，其值是一个随机值，所以不知道 5 将储存在何处。这可能不会出什么错，也可能会擦写数据或代码，或者导致程序崩溃。切记：创建一个指针时，系统只分配了储存指针本身的内存，并未分配储存数据的内存。因此，在使用指针之前，必须先用已分配的地址初始化它。例如，可以用一个现有变量的地址初始化该指针（使用带指针形参的函数时，就属于这种情况）。或者还可以使用第 12 章将介绍的 malloc() 函数先分配内存。无论如何，使用指针时一定要注意，不要解引用未初始化的指针！

```
double * pd; // 未初始化的指针
*pd = 2.4; // 不要这样做
```

假设

```
int urn[3];
int * ptr1, * ptr2;
```

下面是一些有效和无效的语句：

有效语句	无效语句
ptr1++;	urn++;
ptr2 = ptr1 + 2;	ptr2 = ptr2 + ptr1;
ptr2 = urn + 1;	ptr2 = urn * ptr1;

基于这些有效的操作，C 程序员创建了指针数组、函数指针、指向指针的指针数组、指向函数的指针数组等。别紧张，接下来我们将根据已学的内容介绍指针的一些基本用法。指针的第一个基本用法是在函数间传递信息。前面学过，如果希望在被调函数中改变主调函数的变量，必须使用指针。指针的第二个基本用法是用在处理数组的函数中。下面我们再来看一个使用函数和数组的编程示例。

10.6 保护数组中的数据

编写一个处理基本类型（如，int）的函数时，要选择是传递 int 类型的值还是传递指向 int 的指针。通常都是直接传递数值，只有程序需要在函数中改变该数值时，才会传递指针。对于数组别无选择，必须传递指针，因为这样做效率高。如果一个函数按值传递数组，则必须分配足够的空间来储存原数组的副本，然后把原数组所有的数据拷贝至新的数组中。如果把数组的地址传递给函数，让函数直接处理原数组则效率要高。

传递地址会导致一些问题。C 通常都按值传递数据，因为这样做可以保证数据的完整性。如果函数使用的是原始数据的副本，就不会意外修改原始数据。但是，处理数组的函数通常都需要使用原始数据，因此这样的函数可以修改原数组。有时，这正是我们需要的。例如，下面的函数给数组的每个元素都加上一个相同的值：

```
void add_to(double ar[], int n, double val)
{
    int i;
    for (i = 0; i < n; i++)
        ar[i] += val;
}
```

因此，调用该函数后，prices 数组中的每个元素的值都增加了 2.5：

```
add_to(prices, 100, 2.50);
```

该函数修改了数组中的数据。之所以可以这样做，是因为函数通过指针直接使用了原始数据。

然而，其他函数并不需要修改数据。例如，下面的函数计算数组中所有元素之和，它不用改变数组的数据。但是，由于 ar 实际上是一个指针，所以编程错误可能会破坏原始数据。例如，下面示例中的 ar[i]++ 会导致数组中每个元素的值都加 1：

```

int sum(int ar[], int n) // 错误的代码
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i]++; // 错误递增了每个元素的值
    return total;
}

```

10.6.1 对形式参数使用 const

在 K&R C 的年代，避免类似错误的唯一方法是提高警惕。ANSI C 提供了一种预防手段。如果函数的意图不是修改数组中的数据内容，那么在函数原型和函数定义中声明形式参数时应使用关键字 `const`。例如，`sum()` 函数的原型和定义如下：

```

int sum(const int ar[], int n); /* 函数原型 */

int sum(const int ar[], int n) /* 函数定义 */
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i];
    return total;
}

```

以上代码中的 `const` 告诉编译器，该函数不能修改 `ar` 指向的数组中的内容。如果在函数中不小心使用类似 `ar[i]++` 的表达式，编译器会捕获这个错误，并生成一条错误信息。

这里一定要理解，这样使用 `const` 并不是要求原数组是常量，而是该函数在处理数组时将其视为常量，不可更改。这样使用 `const` 可以保护数组的数据不被修改，就像按值传递可以保护基本数据类型的原始值不被改变一样。一般而言，如果编写的函数需要修改数组，在声明数组形参时则不使用 `const`；如果编写的函数不用修改数组，那么在声明数组形参时最好使用 `const`。

程序清单 10.14 的程序中，一个函数显示数组的内容，另一个函数给数组每个元素都乘以一个给定值。因为第 1 个函数不用改变数组，所以在声明数组形参时使用了 `const`；而第 2 个函数需要修改数组元素的值，所以不使用 `const`。

程序清单 10.14 arf.c 程序

```

/* arf.c -- 处理数组的函数 */
#include <stdio.h>
#define SIZE 5
void show_array(const double ar[], int n);
void mult_array(double ar[], int n, double mult);
int main(void)
{
    double dip[SIZE] = { 20.0, 17.66, 8.2, 15.3, 22.22 };

    printf("The original dip array:\n");
    show_array(dip, SIZE);
    mult_array(dip, SIZE, 2.5);
    printf("The dip array after calling mult_array():\n");
    show_array(dip, SIZE);
}

```

```

        return 0;
    }

/* 显示数组的内容 */
void show_array(const double ar[], int n)
{
    int i;

    for (i = 0; i < n; i++)
        printf("%8.3f ", ar[i]);
    putchar('\n');
}

/* 把数组的每个元素都乘以相同的值 */
void mult_array(double ar[], int n, double mult)
{
    int i;

    for (i = 0; i < n; i++)
        ar[i] *= mult;
}

```

下面是该程序的输出:

```

The original dip array:
 20.000 17.660 8.200 15.300 22.220
The dip array after calling mult_array():
 50.000 44.150 20.500 38.250 55.550

```

注意该程序中两个函数的返回类型都是 void。虽然 `mult_array()` 函数更新了 `dip` 数组的值，但是并未使用 `return` 机制。

10.6.2 const 的其他内容

我们在前面使用 `const` 创建过变量:

```
const double PI = 3.14159;
```

虽然用 `#define` 指令可以创建类似功能的符号常量，但是 `const` 的用法更加灵活。可以创建 `const` 数组、`const` 指针和指向 `const` 的指针。

程序清单 10.4 演示了如何使用 `const` 关键字保护数组:

```
#define MONTHS 12
...
const int days[MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

如果程序稍后尝试改变数组元素的值，编译器将生成一个编译期错误消息:

```
days[9] = 44; /* 编译错误 */
```

指向 `const` 的指针不能用于改变值。考虑下面的代码:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * pd = rates; // pd 指向数组的首元素
```

第 2 行代码把 `pd` 指向的 `double` 类型的值声明为 `const`，这表明不能使用 `pd` 来更改它所指向的值:

```
*pd = 29.89; // 不允许
pd[2] = 222.22; // 不允许
rates[0] = 99.99; // 允许，因为 rates 未被 const 限定
```

无论是使用指针表示法还是数组表示法，都不允许使用 `pd` 修改它所指向数据的值。但是要注意，因为 `rates` 并未被声明为 `const`，所以仍然可以通过 `rates` 修改元素的值。另外，可以让 `pd` 指向别处：

```
pd++; /* 让 pd 指向 rates[1] -- 没问题 */
```

指向 `const` 的指针通常用于函数形参中，表明该函数不会使用指针改变数据。例如，程序清单 10.14 中的 `show_array()` 函数原型如下：

```
void show_array(const double *ar, int n);
```

关于指针赋值和 `const` 需要注意一些规则。首先，把 `const` 数据或非 `const` 数据的地址初始化为指向 `const` 的指针或为其赋值是合法的：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
const double * pc = rates; // 有效
pc = locked;           // 有效
pc = &rates[3];         // 有效
```

然而，只能把非 `const` 数据的地址赋给普通指针：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
double * pnc = rates; // 有效
pnc = locked;         // 无效
pnc = &rates[3];       // 有效
```

这个规则非常合理。否则，通过指针就能改变 `const` 数组中的数据。

应用以上规则的例子，如 `show_array()` 函数可以接受普通数组名和 `const` 数组名作为参数，因为这两种参数都可以用来初始化指向 `const` 的指针：

```
show_array(rates, 5);      // 有效
show_array(locked, 4);     // 有效
```

因此，对函数的形参使用 `const` 不仅能保护数据，还能让函数处理 `const` 数组。

另外，不应该把 `const` 数组名作为实参传递给 `mult_array()` 这样的函数：

```
mult_array(rates, 5, 1.2); // 有效
mult_array(locked, 4, 1.2); // 不要这样做
```

C 标准规定，使用非 `const` 标识符（如，`mult_array()` 的形参 `ar`）修改 `const` 数据（如，`locked`）导致的结果是未定义的。

`const` 还有其他的用法。例如，可以声明并初始化一个不能指向别处的指针，关键是 `const` 的位置：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
double * const pc = rates; // pc 指向数组的开始
pc = &rates[2];           // 不允许，因为该指针不能指向别处
*pc = 92.99;              // 没问题 -- 更改 rates[0] 的值
```

可以用这种指针修改它所指向的值，但是它只能指向初始化时设置的地址。

最后，在创建指针时还可以使用 `const` 两次，该指针既不能更改它所指向的地址，也不能修改指向地址上的值：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * const pc = rates;
pc = &rates[2];    // 不允许
*pc = 92.99;      // 不允许
```

10.7 指针和多维数组

指针和多维数组有什么关系？为什么要了解它们的关系？处理多维数组的函数要用到指针，所以在使用这种函数之前，先要更深入地学习指针。至于第 1 个问题，我们通过几个示例来回答。为简化讨论，我们使用较小的数组。假设有下面的声明：

```
int zippo[4][2]; /* 内含 int 数组的数组 */
```

然后数组名 zippo 是该数组首元素的地址。在本例中，zippo 的首元素是一个内含两个 int 值的数组，所以 zippo 是这个内含两个 int 值的数组的地址。下面，我们从指针的属性进一步分析。

- 因为 zippo 是数组首元素的地址，所以 zippo 的值和 `&zippo[0]` 的值相同。而 zippo[0] 本身是一个内含两个整数的数组，所以 zippo[0] 的值和它首元素（一个整数）的地址（即 `&zippo[0][0]` 的值）相同。简而言之，zippo[0] 是一个占用一个 int 大小对象的地址，而 zippo 是一个占用两个 int 大小对象的地址。由于这个整数和内含两个整数的数组都开始于同一个地址，所以 zippo 和 zippo[0] 的值相同。
- 给指针或地址加 1，其值会增加对应类型大小的数值。在这方面，zippo 和 zippo[0] 不同，因为 zippo 指向的对象占用了两个 int 大小，而 zippo[0] 指向的对象只占用一个 int 大小。因此，`zippo + 1` 和 `zippo[0] + 1` 的值不同。
- 解引用一个指针（在指针前使用 * 运算符）或在数组名后使用带下标的[]运算符，得到引用对象代表的值。因为 zippo[0] 是该数组首元素（zippo[0][0]）的地址，所以 `*zippo[0]` 表示储存在 zippo[0][0] 上的值（即一个 int 类型的值）。与此类似，`*zippo` 代表该数组首元素（zippo[0]）的值，但是 zippo[0] 本身是一个 int 类型值的地址。该值的地址是 `&zippo[0][0]`，所以 `*zippo` 就是 `&zippo[0][0]`。对两个表达式应用解引用运算符表明，`**zippo` 与 `*&zippo[0][0]` 等价，这相当于 `zippo[0][0]`，即一个 int 类型的值。简而言之，zippo 是地址的地址，必须解引用两次才能获得原始值。地址的地址或指针的指针就是双重间接（double indirection）的例子。

显然，增加数组维数会增加指针的复杂度。现在，大部分初学者都开始意识到指针为什么是 C 语言中最难的部分。认真思考上述内容，看看是否能用所学的知识解释程序清单 10.15 中的程序。该程序显示了一些地址值和数组的内容。

程序清单 10.15 zippol.c 程序

```
/* zippol.c -- zippo 的相关信息 */
#include <stdio.h>
int main(void)
{
    int zippo[4][2] = { { 2, 4 }, { 6, 8 }, { 1, 3 }, { 5, 7 } };

    printf(" zippo = %p,      zippo + 1 = %p\n", zippo, zippo + 1);
    printf("zippo[0] = %p, zippo[0] + 1 = %p\n", zippo[0], zippo[0] + 1);
    printf(" *zippo = %p,   *zippo + 1 = %p\n", *zippo, *zippo + 1);
    printf("zippo[0][0] = %d\n", zippo[0][0]);
    printf(" *zippo[0] = %d\n", *zippo[0]);
    printf(" **zippo = %d\n", **zippo);
    printf("      zippo[2][1] = %d\n", zippo[2][1]);
    printf("*(zippo+2) + 1 = %d\n", (*(zippo + 2) + 1));

    return 0;
}
```

下面是我们的系统运行该程序后的输出：

```

zippo = 0x0064fd38,      zippo + 1 = 0x0064fd40
zippo[0]= 0x0064fd38,    zippo[0] + 1 = 0x0064fd3c
*zippo = 0x0064fd38,     *zippo + 1 = 0x0064fd3c
zippo[0][0] = 2
*zippo[0] = 2
**zippo = 2
zippo[2][1] = 3
*(*(zippo+2) + 1) = 3

```

其他系统显示的地址值和地址形式可能不同，但是地址之间的关系与以上输出相同。该输出显示了二维数组 zippo 的地址和一维数组 zippo[0] 的地址相同。它们的地址都是各自数组首元素的地址，因而与 &zippo[0][0] 的值也相同。

尽管如此，它们也有差别。在我们的系统中，int 是 4 字节。前面讨论过，zippo[0] 指向一个 4 字节的数据对象。zippo[0] 加 1，其值加 4（十六进制中，38+4 得 3c）。数组名 zippo 是一个内含 2 个 int 类型值的数组的地址，所以 zippo 指向一个 8 字节的数据对象。因此，zippo 加 1，它所指向的地址加 8 字节（十六进制中，38+8 得 40）。

该程序演示了 zippo[0] 和 *zippo 完全相同，实际上确实如此。然后，对二维数组名解引用两次，得到储存在数组中的值。使用两个间接运算符 (*) 或者使用两对方括号 ([]) 都能获得该值（还可以使用一个*和一对 []，但是我们暂不讨论这么多情况）。

要特别注意，与 zippo[2][1] 等价的指针表示法是 *(*zippo+2) + 1。看上去比较复杂，应最好能理解。下面列出了理解该表达式的思路：

zippo	← 二维数组首元素的地址（每个元素都是内含两个 int 类型元素的一维数组）
zippo+2	← 二维数组的第 3 个元素（即一维数组）的地址
*(zippo+2)	← 二维数组的第 3 个元素（即一维数组）的首元素（一个 int 类型的值）地址
*(zippo+2) + 1	← 二维数组的第 3 个元素（即一维数组）的第 2 个元素（也是一个 int 类型的值）地址
*(*zippo+2) + 1	← 二维数组的第 3 个一维数组元素的第 2 个 int 类型元素的值，即数组的第 3 行第 2 列的值 (zippo[2][1])

以上分析并不是为了说明用指针表示法 (*(*zippo+2) + 1) 代替数组表示法 (zippo[2][1])，而是提示读者，如果程序恰巧使用一个指向二维数组的指针，而且要通过该指针获取值时，最好用简单的数组表示法，而不是指针表示法。

图 10.5 以另一种视图演示了数组地址、数组内容和指针之间的关系。

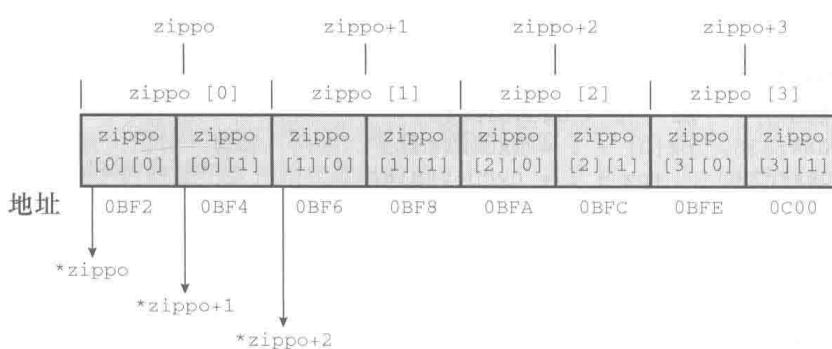


图 10.5 数组的数组

10.7.1 指向多维数组的指针

如何声明一个指针变量 pz 指向一个二维数组（如，zippo）？在编写处理类似 zippo 这样的二维数组时会用到这样的指针。把指针声明为指向 int 的类型还不够。因为指向 int 只能与 zippo[0] 的类型匹配，说明该指针指向一个 int 类型的值。但是 zippo 是它首元素的地址，该元素是一个内含两个 int 类型值的一维数组。因此，pz 必须指向一个内含两个 int 类型值的数组，而不是指向一个 int 类型值，其声明如下：

```
int (* pz)[2]; // pz 指向一个内含两个 int 类型值的数组
```

以上代码把 pz 声明为指向一个数组的指针，该数组内含两个 int 类型值。为什么要在声明中使用圆括号？因为[]的优先级高于*。考虑下面的声明：

```
int * pax[2]; // pax 是一个内含两个指针元素的数组，每个元素都指向 int 的指针
```

由于[]优先级高，先与 pax 结合，所以 pax 成为一个内含两个元素的数组。然后*表示 pax 数组内含两个指针。最后，int 表示 pax 数组中的指针都指向 int 类型的值。因此，这行代码声明了两个指向 int 的指针。而前面有圆括号的版本，*先与 pz 结合，因此声明的是一个指向数组（内含两个 int 类型的值）的指针。程序清单 10.16 演示了如何使用指向二维数组的指针。

程序清单 10.16 zippo2.c 程序

```
/* zippo2.c -- 通过指针获取 zippo 的信息 */
#include <stdio.h>
int main(void)
{
    int zippo[4][2] = { { 2, 4 }, { 6, 8 }, { 1, 3 }, { 5, 7 } };
    int(*pz)[2];
    pz = zippo;

    printf("  pz = %p,      pz + 1 = %p\n",      pz, pz + 1);
    printf("pz[0] = %p,  pz[0] + 1 = %p\n",  pz[0], pz[0] + 1);
    printf("  *pz = %p,    *pz + 1 = %p\n",    *pz, *pz + 1);
    printf("pz[0][0] = %d\n",  pz[0][0]);
    printf("  *pz[0] = %d\n", *pz[0]);
    printf("  **pz = %d\n",   **pz);
    printf("      pz[2][1] = %d\n", pz[2][1]);
    printf("*(*(pz+2) + 1) = %d\n", *(*(pz + 2) + 1));

    return 0;
}
```

下面是该程序的输出：

```
pz = 0x0064fd38,      pz + 1 = 0x0064fd40
pz[0] = 0x0064fd38,  pz[0] + 1 = 0x0064fd3c
  *pz = 0x0064fd38,    *pz + 1 = 0x0064fd3c
pz[0][0] = 2
  *pz[0] = 2
  **pz = 2
      pz[2][1] = 3
  *(*(pz+2) + 1) = 3
```

系统不同，输出的地址可能不同，但是地址之间的关系相同。如前所述，虽然 pz 是一个指针，不是数组名，但是也可以使用 pz[2][1] 这样的写法。可以用数组表示法或指针表示法来表示一个数组元素，既

可以使用数组名，也可以使用指针名：

```
zippo[m][n] == *(*(zippo + m) + n)
pz[m][n] == *(*(pz + m) + n)
```

10.7.2 指针的兼容性

指针之间的赋值比数值类型之间的赋值要严格。例如，不用类型转换就可以把 `int` 类型的值赋给 `double` 类型的变量，但是两个类型的指针不能这样做。

```
int n = 5;
double x;
int * p1 = &n;
double * pd = &x;
x = n;           // 隐式类型转换
pd = p1;         // 编译时错误
```

更复杂的类型也是如此。假设有如下声明：

```
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2; // 一个指向指针的指针
```

有如下的语句：

```
pt = &ar1[0][0]; // 都是指向 int 的指针
pt = ar1[0];     // 都是指向 int 的指针
pt = ar1;         // 无效
pa = ar1;         // 都是指向内含 3 个 int 类型元素数组的指针
pa = ar2;         // 无效
p2 = &pt;          // both pointer-to-int *
*p2 = ar2[0];    // 都是指向 int 的指针
p2 = ar2;         // 无效
```

注意，以上无效的赋值表达式语句中涉及的两个指针都是指向不同的类型。例如，`pt` 指向一个 `int` 类型值，而 `ar1` 指向一个内含 3 个 `int` 类型元素的数组。类似地，`pa` 指向一个内含 2 个 `int` 类型元素的数组，所以它与 `ar1` 的类型兼容，但是 `ar2` 指向一个内含 2 个 `int` 类型元素的数组，所以 `pa` 与 `ar2` 不兼容。

上面的最后两个例子有些棘手。变量 `p2` 是指向指针的指针，它指向的指针指向 `int`，而 `ar2` 是指向数组的指针，该数组内含 2 个 `int` 类型的元素。所以，`p2` 和 `ar2` 的类型不同，不能把 `ar2` 赋给 `p2`。但是，`*p2` 是指向 `int` 的指针，与 `ar2[0]` 兼容。因为 `ar2[0]` 是指向该数组首元素 (`ar2[0][0]`) 的指针，所以 `ar2[0]` 也是指向 `int` 的指针。

一般而言，多重解引用让人费解。例如，考虑下面的代码：

```
int x = 20;
const int y = 23;
int * p1 = &x;
const int * p2 = &y;
const int ** pp2;

p1 = p2;      // 不安全 -- 把 const 指针赋给非 const 指针
p2 = p1;      // 有效 -- 把非 const 指针赋给 const 指针
pp2 = &p1;    // 不安全 -- 嵌套指针类型赋值
```

前面提到过，把 `const` 指针赋给非 `const` 指针不安全，因为这样可以使用新的指针改变 `const` 指针指向的数据。编译器在编译代码时，可能会给出警告，执行这样的代码是未定义的。但是把非 `const` 指针赋给 `const` 指针没问题，前提是只进行一级解引用：

```
p2 = p1; // 有效 -- 把非 const 指针赋给 const 指针
```

但是进行两级解引用时，这样的赋值也不安全，例如，考虑下面的代码：

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1;    // 允许，但是这导致 const 限定符失效（根据第 1 行代码，不能通过*pp2 修改它所指向的内容）
*pp2 = &n;    // 有效，两者都声明为 const，但是这将导致 p1 指向 n (*pp2 已被修改)
*p1 = 10; // 有效，但是这将改变 n 的值（但是根据第 3 行代码，不能修改 n 的值）
```

发生了什么？如前所示，标准规定了通过非 `const` 指针更改 `const` 数据是未定义的。例如，在 Terminal 中（OS X 对底层 UNIX 系统的访问）使用 `gcc` 编译包含以上代码的小程序，导致 `n` 最终的值是 13，但是在相同系统下使用 `clang` 来编译，`n` 最终的值是 10。两个编译器都给出指针类型不兼容的警告。当然，可以忽略这些警告，但是最好不要相信该程序运行的结果，这些结果都是未定义的。

C `const` 和 C++ `const`

C 和 C++ 中 `const` 的用法很相似，但是并不完全相同。区别之一是，C++ 允许在声明数组大小时使用 `const` 整数，而 C 却不允许。区别之二是，C++ 的指针赋值检查更严格：

```
const int y;
const int * p2 = &y;
int * p1;
p1 = p2; // C++ 中不允许这样做，但是C可能只给出警告
```

C++ 不允许把 `const` 指针赋给非 `const` 指针。而 C 则允许这样做，但是如果通过 `p1` 更改 `y`，其行为是未定义的。

10.7.3 函数和多维数组

如果要编写处理二维数组的函数，首先要能正确地理解指针才能写出声明函数的形参。在函数体中，通常使用数组表示法进行相关操作。

下面，我们编写一个处理二维数组的函数。一种方法是，利用 `for` 循环把处理一维数组的函数应用到二维数组的每一行。如下所示：

```
int junk[3][4] = { {2,4,5,8}, {3,5,6,9}, {12,10,8,6} };
int i, j;
int total = 0;
for (i = 0; i < 3 ; i++)
    total += sum(junk[i], 4); // junk[i] 是一维数组
```

记住，如果 `junk` 是二维数组，`junk[i]` 就是一维数组，可将其视为二维数组的一行。这里，`sum()` 函数计算二维数组的每行的总和，然后 `for` 循环再把每行的总和加起来。

然而，这种方法无法记录行和列的信息。用这种方法计算总和，行和列的信息并不重要。但如果每行代表一年，每列代表一个月，就还需要一个函数计算某列的总和。该函数要知道行和列的信息，可以通过声明正确类型的形参变量来完成，以便函数能正确地传递数组。在这种情况下，数组 `junk` 是一个内含 3

个数组元素的数组，每个元素是内含 4 个 int 类型值的数组（即 junk 是一个 3 行 4 列的二维数组）。通过前面的讨论可知，这表明 junk 是一个指向数组（内含 4 个 int 类型值）的指针。可以这样声明函数的形式参数：

```
void somefunction( int (* pt)[4] );
```

另外，如果当且仅当 pt 是一个函数的形式参数时，可以这样声明：

```
void somefunction( int pt[][4] );
```

注意，第 1 个方括号是空的。空的方括号表明 pt 是一个指针。这样的变量稍后可以用作相同方法作为 junk。下面的程序示例中就是这样做的，如程序清单 10.17 所示。注意该程序清单演示了 3 种等价的原型语法。

程序清单 10.17 array2d.c 程序

```
// array2d.c -- 处理二维数组的函数
#include <stdio.h>
#define ROWS 3
#define COLS 4
void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [] [COLS], int);           // 省略形参名，没问题
int sum2d(int(*ar)[COLS], int rows);        // 另一种语法
int main(void)
{
    int junk[ROWS][COLS] = {
        { 2, 4, 6, 8 },
        { 3, 5, 7, 9 },
        { 12, 10, 8, 6 }
    };

    sum_rows(junk, ROWS);
    sum_cols(junk, ROWS);
    printf("Sum of all elements = %d\n", sum2d(junk, ROWS));

    return 0;
}

void sum_rows(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;

    for (r = 0; r < rows; r++)
    {
        tot = 0;
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
        printf("row %d: sum = %d\n", r, tot);
    }
}

void sum_cols(int ar[] [COLS], int rows)
{
    int r;
    int c;
```

```

int tot;

for (c = 0; c < COLS; c++)
{
    tot = 0;
    for (r = 0; r < rows; r++)
        tot += ar[r][c];
    printf("col %d: sum = %d\n", c, tot);
}
}

int sum2d(int ar[][][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];

    return tot;
}

```

该程序的输出如下：

```

row 0: sum = 20
row 1: sum = 24
row 2: sum = 36
col 0: sum = 17
col 1: sum = 19
col 2: sum = 21
col 3: sum = 23
Sum of all elements = 80

```

程序清单 10.17 中的程序把数组名 `junk`（即，指向数组首元素的指针，首元素是子数组）和符号常量 `ROWS`（代表行数 3）作为参数传递给函数。每个函数都把 `ar` 视为内含数组元素（每个元素是内含 4 个 `int` 类型值的数组）的数组。列数内置在函数体中，但是行数靠函数传递得到。如果传入函数的行数是 12，那么函数要处理的是 12×4 的数组。因为 `rows` 是元素的个数，然而，因为每个元素都是数组，或者视为一行，`rows` 也可以看成是行数。

注意，`ar` 和 `main()` 中的 `junk` 都使用数组表示法。因为 `ar` 和 `junk` 的类型相同，它们都是指向内含 4 个 `int` 类型值的数组的指针。

注意，下面的声明不正确：

```
int sum2(int ar[][], int rows); // 错误的声明
```

前面介绍过，编译器会把数组表示法转换成指针表示法。例如，编译器会把 `ar[1]` 转换成 `ar+1`。编译器对 `ar+1` 求值，要知道 `ar` 所指向的对象大小。下面的声明：

```
int sum2(int ar[][4], int rows); // 有效声明
```

表示 `ar` 指向一个内含 4 个 `int` 类型值的数组（在我们的系统中，`ar` 指向的对象占 16 字节），所以 `ar+1` 的意思是“该地址加上 16 字节”。如果第 2 对方括号是空的，编译器就不知道该怎样处理。

也可以在第 1 对方括号中写上大小，如下所示，但是编译器会忽略该值：

```
int sum2(int ar[3][4], int rows); // 有效声明, 但是 3 将被忽略
```

与使用 `typedef` (第 5 章和第 14 章中讨论) 相比, 这种形式方便得多:

```
typedef int arr4[4];           // arr4 是一个内含 4 个 int 的数组
typedef arr4 arr3x4[3];        // arr3x4 是一个内含 3 个 arr4 的数组
int sum2(arr3x4 ar, int rows); // 与下面的声明相同
int sum2(int ar[3][4], int rows); // 与下面的声明相同
int sum2(int ar[][4], int rows); // 标准形式
```

一般而言, 声明一个指向 N 维数组的指针时, 只能省略最左边方括号中的值:

```
int sum4d(int ar[][12][20][30], int rows);
```

因为第 1 对方括号只用于表明这是一个指针, 而其他的方括号则用于描述指针所指向数据对象的类型。

下面的声明与该声明等价:

```
int sum4d(int (*ar)[12][20][30], int rows); // ar 是一个指针
```

这里, `ar` 指向一个 $12 \times 20 \times 30$ 的 `int` 数组。

10.8 变长数组 (VLA)

读者在学习处理二维数组的函数中可能不太理解, 为何只把数组的行数作为函数的形参, 而列数却内置在函数体内。例如, 函数定义如下:

```
#define COLS 4
int sum2d(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
    return tot;
}
```

假设声明了下列数组:

```
int array1[5][4];
int array2[100][4];
int array3[2][4];
```

可以用 `sum2d()` 函数分别计算这些数组的元素之和:

```
tot = sum2d(array1, 5); // 5 × 4 数组的元素之和
tot = sum2d(array2, 100); // 100 × 4 数组的元素之和
tot = sum2d(array3, 2); // 2 × 4 数组的元素之和
```

`sum2d()` 函数之所以能处理这些数组, 是因为这些数组的列数固定为 4, 而行数被传递给形参 `rows`, `rows` 是一个变量。但是如果要计算 6×5 的数组 (即 6 行 5 列), 就不能使用这个函数, 必须重新创建一个 `COLS` 为 5 的函数。因为 C 规定, 数组的维数必须是常量, 不能用变量来代替 `COLS`。

要创建一个能处理任意大小二维数组的函数, 比较繁琐 (必须把数组作为一维数组传递, 然后让函数计算每行的开始处)。而且, 这种方法不好处理 FORTRAN 的子例程, 这些子例程都允许在函数调用中指定两个维度。虽然 FORTRAN 是比较老的编程语言, 但是在过去的几十年里, 数值计算领域的专家已经用 FORTRAN 开发出许多有用的计算库。C 正逐渐替代 FORTRAN, 如果能直接转换现有的 FORTRAN 库就好了。

鉴于此，C99 新增了变长数组（*variable-length array*, VLA），允许使用变量表示数组的维度。如下所示：

```
int quarters = 4;
int regions = 5;
double sales[regions][quarters]; // 一个变长数组 (VLA)
```

前面提到过，变长数组有一些限制。变长数组必须是自动存储类别，这意味着无论在函数中声明还是作为函数形参声明，都不能使用 `static` 或 `extern` 存储类别说明符（第 12 章介绍）。而且，不能在声明中初始化它们。最终，C11 把变长数组作为一个可选特性，而不是必须强制实现的特性。

注意 变长数组不能改变大小

变长数组中的“变”不是指可以修改已创建数组的大小。一旦创建了变长数组，它的大小则保持不变。这里的“变”指的是：在创建数组时，可以使用变量指定数组的维度。

由于变长数组是 C 语言的新特性，目前完全支持这一特性的编译器不多。下面我们来看一个简单的例子：如何编写一个函数，计算 `int` 的二维数组所有元素之和。

首先，要声明一个带二维变长数组参数的函数，如下所示：

```
int sum2d(int rows, int cols, int ar[rows][cols]); // ar 是一个变长数组 (VLA)
```

注意前两个形参（`rows` 和 `cols`）用作第 3 个形参二维数组 `ar` 的两个维度。因为 `ar` 的声明要使用 `rows` 和 `cols`，所以在形参列表中必须在声明 `ar` 之前先声明这两个形参。因此，下面的原型是错误的：

```
int sum2d(int ar[rows][cols], int rows, int cols); // 无效的顺序
```

C99/C11 标准规定，可以省略原型中的形参名，但是在这种情况下，必须用星号来代替省略的维度：

```
int sum2d(int, int, int ar[*][*]); // ar 是一个变长数组 (VLA)，省略了维度形参名
```

其次，该函数的定义如下：

```
int sum2d(int rows, int cols, int ar[rows][cols])
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < cols; c++)
            tot += ar[r][c];
    return tot;
}
```

该函数除函数头与传统的 C 函数（程序清单 10.17）不同外，还把符号常量 `COLS` 替换成变量 `cols`。这是因为在函数头中使用了变长数组。由于用变量代表行数和列数，所以新的 `sum2d()` 现在可以处理任意大小的二维 `int` 数组，如程序清单 10.18 所示。但是，该程序要求编译器支持变长数组特性。另外，该程序还演示了以变长数组作为形参的函数既可处理传统 C 数组，也可处理变长数组。

程序清单 10.18 vararr2d.c 程序

```
//vararr2d.c -- 使用变长数组的函数
#include <stdio.h>
#define ROWS 3
#define COLS 4
int sum2d(int rows, int cols, int ar[rows][cols]);
```

```

int main(void)
{
    int i, j;
    int rs = 3;
    int cs = 10;
    int junk[ROWS][COLS] = {
        { 2, 4, 6, 8 },
        { 3, 5, 7, 9 },
        { 12, 10, 8, 6 }
    };

    int morejunk[ROWS - 1][COLS + 2] = {
        { 20, 30, 40, 50, 60, 70 },
        { 5, 6, 7, 8, 9, 10 }
    };

    int varr[rs][cs]; // 变长数组 (VLA)

    for (i = 0; i < rs; i++)
        for (j = 0; j < cs; j++)
            varr[i][j] = i * j + j;

    printf("3x5 array\n");
    printf("Sum of all elements = %d\n", sum2d(ROWS, COLS, junk));

    printf("2x6 array\n");
    printf("Sum of all elements = %d\n", sum2d(ROWS - 1, COLS + 2, morejunk));

    printf("3x10 VLA\n");
    printf("Sum of all elements = %d\n", sum2d(rs, cs, varr));

    return 0;
}

// 带变长数组形参的函数
int sum2d(int rows, int cols, int ar[rows][cols])
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < cols; c++)
            tot += ar[r][c];

    return tot;
}

```

下面是该程序的输出：

```

3x5 array
Sum of all elements = 80
2x6 array
Sum of all elements = 315
3x10 VLA
Sum of all elements = 270

```

需要注意的是，在函数定义的形参列表中声明的变长数组并未实际创建数组。和传统的语法类似，变长数组名实际上是一个指针。这说明带变长数组形参的函数实际上是在原始数组中处理数组，因此可以修改传入的数组。下面的代码段指出指针和实际数组是何时声明的：

```

int thing[10][6];
twoset(10, 6, thing);
...
}

void twoset (int n, int m, int ar[n][m]) // ar 是一个指向数组（内含 m 个 int 类型的值）的指针
{
    int temp[n][m]; // temp 是一个 n×m 的 int 数组
    temp[0][0] = 2; // 设置 temp 的一个元素为 2
    ar[0][0] = 2; // 设置 thing[0][0] 为 2
}

```

如上代码所示调用 `twoset()` 时，`ar` 成为指向 `thing[0]` 的指针，`temp` 被创建为 10×6 的数组。因为 `ar` 和 `thing` 都是指向 `thing[0]` 的指针，`ar[0][0]` 与 `thing[0][0]` 访问的数据位置相同。

const 和数组大小

是否可以在声明数组时使用 `const` 变量？

```

const int SZ = 80;
...
double ar[SZ]; // 是否允许?

```

C90 标准不允许（也可能允许）。数组的大小必须是给定的整型常量表达式，可以是整型常量组合，如 `20`、`sizeof` 表达式或其他不是 `const` 的内容。由于 C 实现可以扩大整型常量表达式的范围，所以可能会允许使用 `const`，但是这种代码可能无法移植。

C99/C11 标准允许在声明变长数组时使用 `const` 变量。所以该数组的定义必须是声明在块中的自动存储类数组。

变长数组还允许动态内存分配，这说明可以在程序运行时指定数组的大小。普通 C 数组都是静态内存分配，即在编译时确定数组的大小。由于数组大小是常量，所以编译器在编译时就知道了。第 12 章将详细介绍动态内存分配。

10.9 复合字面量

假设给带 `int` 类型形参的函数传递一个值，要传递 `int` 类型的变量，但是也可以传递 `int` 类型常量，如 `5`。在 C99 标准以前，对于带数组形参的函数，情况不同，可以传递数组，但是没有等价的数组常量。C99 新增了复合字面量（*compound literal*）。字面量是除符号常量外的常量。例如，`5` 是 `int` 类型字面量，`81.3` 是 `double` 类型的字面量，`'Y'` 是 `char` 类型的字面量，“`elephant`”是字符串字面量。发布 C99 标准的委员会认为，如果有代表数组和结构内容的复合字面量，在编程时会更方便。

对于数组，复合字面量类似数组初始化列表，前面是用括号括起来的类型名。例如，下面是一个普通的数组声明：

```
int diva[2] = {10, 20};
```

下面的复合字面量创建了一个和 `diva` 数组相同的匿名数组，也有两个 `int` 类型的值：

```
(int [2]) {10, 20}      // 复合字面量
```

注意，去掉声明中的数组名，留下的 int [2]即是复合字面量的类型名。

初始化有数组名的数组时可以省略数组大小，复合字面量也可以省略大小，编译器会自动计算数组当前的元素个数：

```
(int []) {50, 20, 90} // 内含 3 个元素的复合字面量
```

因为复合字面量是匿名的，所以不能先创建然后再使用它，必须在创建的同时使用它。使用指针记录地址就是一种用法。也就是说，可以这样用：

```
int * pt1;
pt1 = (int [2]) {10, 20};
```

注意，该复合字面量的字面常量与上面创建的 diva 数组的字面常量完全相同。与有数组名的数组类似，复合字面量的类型名也代表首元素的地址，所以可以把它赋给指向 int 的指针。然后便可使用这个指针。例如，本例中*pt1 是 10，pt1[1] 是 20。

还可以把复合字面量作为实际参数传递给带有匹配形式参数的函数：

```
int sum(const int ar[], int n);
...
int total3;
total3 = sum((int []) {4, 4, 4, 5, 5, 5}, 6);
```

这里，第 1 个实参是内含 6 个 int 类型值的数组，和数组名类似，这同时也是该数组首元素的地址。这种用法的好处是，把信息传入函数前不必先创建数组，这是复合字面量的典型用法。

可以把这种用法应用于二维数组或多维数组。例如，下面的代码演示了如何创建二维 int 数组并储存其地址：

```
int (*pt2)[4];      // 声明一个指向二维数组的指针，该数组内含 2 个数组元素，
                     // 每个元素是内含 4 个 int 类型值的数组
pt2 = (int [2][4]) { {1, 2, 3, -9}, {4, 5, 6, -8} };
```

如上所示，该复合字面量的类型是 int [2][4]，即一个 2×4 的 int 数组。

程序清单 10.19 把上述例子放进一个完整的程序中。

程序清单 10.19 flc.c 程序

```
// flc.c -- 有趣的常量
#include <stdio.h>
#define COLS 4
int sum2d(const int ar[][COLS], int rows);
int sum(const int ar[], int n);
int main(void)
{
    int total1, total2, total3;
    int * pt1;
    int (*pt2)[COLS];

    pt1 = (int[2]) { 10, 20 };
    pt2 = (int[2][COLS]) { {1, 2, 3, -9}, {4, 5, 6, -8} };

    total1 = sum(pt1, 2);
    total2 = sum2d(pt2, 2);
    total3 = sum((int []) { 4, 4, 4, 5, 5, 5 }, 6);
    printf("total1 = %d\n", total1);
    printf("total2 = %d\n", total2);
```

```

printf("total3 = %d\n", total3);

return 0;
}

int sum(const int ar [], int n)
{
    int i;
    int total = 0;

    for (i = 0; i < n; i++)
        total += ar[i];

    return total;
}

int sum2d(const int ar [] [COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r] [c];

    return tot;
}

```

要支持 C99 的编译器才能正常运行该程序示例（目前并不是所有的编译器都支持），其输出如下：

```

total1 = 30
total2 = 4
total3 = 27

```

记住，复合字面量是提供只临时需要的值的一种手段。复合字面量具有块作用域（第 12 章将介绍相关内容），这意味着一旦离开定义复合字面量的块，程序将无法保证该字面量是否存在。也就是说，复合字面量的定义在最内层的花括号中。

10.10 关键概念

数组用于储存相同类型的数据。C 把数组看作是派生类型，因为数组是建立在其他类型的基础上。也就是说，无法简单地声明一个数组。在声明数组时必须说明其元素的类型，如 `int` 类型的数组、`float` 类型的数组，或其他类型的数组。所谓的其他类型也可以是数组类型，这种情况下，创建的是数组的数组（或称为二维数组）。

通常编写一个函数来处理数组，这样在特定的函数中解决特定的问题，有助于实现程序的模块化。在把数组名作为实际参数时，传递给函数的不是整个数组，而是数组的地址（因此，函数对应的形式参数是指针）。为了处理数组，函数必须知道从何处开始读取数据和要处理多少个数组元素。数组地址提供了“地址”，“元素个数”可以内置在函数中或作为单独的参数传递。第 2 种方法更普遍，因为这样做可以让同一个函数处理不同大小的数组。

数组和指针的关系密切，同一个操作可以用数组表示法或指针表示法。它们之间的关系允许你在处理

数组的函数中使用数组表示法，即使函数的形式参数是一个指针，而不是数组。

对于传统的 C 数组，必须用常量表达式指明数组的大小，所以数组大小在编译时就已确定。C99/C11 新增了变长数组，可以用变量表示数组大小。这意味着变长数组的大小延迟到程序运行时才确定。

10.11 本章小结

数组是一组数据类型相同的元素。数组元素按顺序储存在内存中，通过整数下标（或索引）可以访问各元素。在 C 中，数组首元素的下标是 0，所以对于内含 n 个元素的数组，其最后一个元素的下标是 n-1。作为程序员，要确保使用有效的数组下标，因为编译器和运行的程序都不会检查下标的有效性。

声明一个简单的一维数组形式如下：

```
type name [ size ];
```

这里，type 是数组中每个元素的数据类型，name 是数组名，size 是数组元素的个数。对于传统的 C 数组，要求 size 是整型常量表达式。但是 C99/C11 允许使用整型非常量表达式。这种情况下的数组被称为变长数组。

C 把数组名解释为该数组首元素的地址。换言之，数组名与指向该数组首元素的指针等价。概括地说，数组和指针的关系十分密切。如果 ar 是一个数组，那么表达式 ar[i] 和 *(ar+i) 等价。

对于 C 语言而言，不能把整个数组作为参数传递给函数，但是可以传递数组的地址。然后函数可以使用户传入的地址操控原始数组。如果函数没有修改原始数组的意图，应在声明函数的形式参数时使用关键字 const。在被调函数中可以使用数组表示法或指针表示法，无论用哪种表示法，实际上使用的都是指针变量。

指针加上一个整数或递增指针，指针的值以所指向对象的大小为单位改变。也就是说，如果 pd 指向一个数组的 8 字节 double 类型值，那么 pd 加 1 意味着其值加 8，以便它指向该数组的下一个元素。

二维数组即是数组的数组。例如，下面声明了一个二维数组：

```
double sales[5][12];
```

该数组名为 sales，有 5 个元素（一维数组），每个元素都是一个内含 12 个 double 类型值的数组。第 1 个一维数组是 sales[0]，第 2 个一维数组是 sales[1]，以此类推，每个元素都是内含 12 个 double 类型值的数组。使用第 2 个下标可以访问这些一维数组中的特定元素。例如，sales[2][5] 是 sales[2] 的第 6 个元素，而 sales[2] 是 sales 的第 3 个元素。

C 语言传递多维数组的传统方法是把数组名（即数组的地址）传递给类型匹配的指针形参。声明这样的指针形参要指定所有的数组维度，除了第 1 个维度。传递的第 1 个维度通常作为第 2 个参数。例如，为了处理前面声明的 sales 数组，函数原型和函数调用如下：

```
void display(double ar[][12], int rows);
...
display(sales, 5);
```

变长数组提供第 2 种语法，把数组维度作为参数传递。在这种情况下，对应函数原型和函数调用如下：

```
void display(int rows, int cols, double ar[rows][cols]);
...
display(5, 12, sales);
```

虽然上述讨论中使用的是 int 类型的数组和 double 类型的数组，其他类型的数组也是如此。然而，字符串有一些特殊的规则，这是由于其末尾的空字符所致。有了这个空字符，不用传递数组的大小，函数通过检测字符串的末尾也知道在何处停止。我们将在第 11 章中详细介绍。

10.12 复习题

复习题的参考答案在附录 A 中。

- 下面的程序将打印什么内容？

```
#include <stdio.h>
int main(void)
{
    int ref[] = { 8, 4, 0, 2 };
    int *ptr;
    int index;

    for (index = 0, ptr = ref; index < 4; index++, ptr++)
        printf("%d %d\n", ref[index], *ptr);
    return 0;
}
```

- 在复习题 1 中，ref 有多少个元素？
- 在复习题 1 中，ref 的地址是什么？ref + 1 是什么意思？++ref 指向什么？
- 在下面的代码中，*ptr 和*(ptr + 2) 的值分别是什么？

a.

```
int *ptr;
int torf[2][2] = {12, 14, 16};
ptr = torf[0];
```

b.

```
int * ptr;
int fort[2][2] = { {12}, {14,16} };
ptr = fort[0];
```

- 在下面的代码中，**ptr 和** (ptr + 1) 的值分别是什么？

a.

```
int (*ptr)[2];
int torf[2][2] = {12, 14, 16};
ptr = torf;
```

b.

```
int (*ptr)[2];
int fort[2][2] = { {12}, {14,16} };
ptr = fort;
```

- 假设有下面的声明：

```
int grid[30][100];
```

- 用 1 种写法表示 grid[22][56]
- 用 2 种写法表示 grid[22][0]
- 用 3 种写法表示 grid[0][0]

- 正确声明以下各变量：

- digits 是一个内含 10 个 int 类型值的数组
- rates 是一个内含 6 个 float 类型值的数组

- c. mat 是一个内含 3 个元素的数组，每个元素都是内含 5 个整数的数组
d. psa 是一个内含 20 个元素的数组，每个元素都是指向 int 的指针
e. pstr 是一个指向数组的指针，该数组内含 20 个 char 类型的值
8.
a. 声明一个内含 6 个 int 类型值的数组，并初始化各元素为 1、2、4、8、16、32
b. 用数组表示法表示 a 声明的数组的第 3 个元素（其值为 4）
c. 假设编译器支持 C99/C11 标准，声明一个内含 100 个 int 类型值的数组，并初始化最后一个元素为 -1，其他元素不考虑
d. 假设编译器支持 C99/C11 标准，声明一个内含 100 个 int 类型值的数组，并初始化下标为 5、10、11、12、3 的元素为 101，其他元素不考虑
9. 内含 10 个元素的数组下标范围是什么？
10. 假设有下面的声明：
- ```
float rootbeer[10], things[10][5], *pf, value = 2.2;
int i = 3;
```
- 判断以下各项是否有效：
- a. rootbeer[2] = value;
  - b. scanf("%f", &rootbeer );
  - c. rootbeer = value;
  - d. printf("%f", rootbeer);
  - e. things[4][4] = rootbeer[3];
  - f. things[5] = rootbeer;
  - g. pf = value;
  - h. pf = rootbeer;
11. 声明一个  $800 \times 600$  的 int 类型数组。
12. 下面声明了 3 个数组：
- ```
double trots[20];
short clops[10][30];
long shots[5][10][15];
```
- a. 分别以传统方式和以变长数组为参数的方式编写处理 trots 数组的 void 函数原型和函数调用
 - b. 分别以传统方式和以变长数组为参数的方式编写处理 clops 数组的 void 函数原型和函数调用
 - c. 分别以传统方式和以变长数组为参数的方式编写处理 shots 数组的 void 函数原型和函数调用
13. 下面有两个函数原型：
- ```
void show(const double ar[], int n); // n 是数组元素的个数
void show2(const double ar2[][3], int n); // n 是二维数组的行数
```
- a. 编写一个函数调用，把一个内含 8、3、9 和 2 的复合字面量传递给 show() 函数。
  - b. 编写一个函数调用，把一个 2 行 3 列的复合字面量（8、3、9 作为第 1 行，5、4、1 作为第 2 行）传递给 show2() 函数。

## 10.13 编程练习

1. 修改程序清单 10.7 的 rain.c 程序，用指针进行计算（仍然要声明并初始化数组）。

2. 编写一个程序，初始化一个 `double` 类型的数组，然后把该数组的内容拷贝至 3 个其他数组中（在 `main()` 中声明这 4 个数组）。使用带数组表示法的函数进行第 1 份拷贝。使用带指针表示法和指针递增的函数进行第 2 份拷贝。把目标数组名、源数组名和待拷贝的元素个数作为前两个函数的参数。第 3 个函数以目标数组名、源数组名和指向源数组最后一个元素后面的元素的指针。也就是说，给定以下声明，则函数调用如下所示：

```
double source[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
double target1[5];
double target2[5];
double target3[5];
copy_arr(target1, source, 5);
copy_ptr(target2, source, 5);

copy_ptrs(target3, source, source + 5);
```

3. 编写一个函数，返回储存在 `int` 类型数组中的最大值，并在一个简单的程序中测试该函数。
4. 编写一个函数，返回储存在 `double` 类型数组中最大值的下标，并在一个简单的程序中测试该函数。
5. 编写一个函数，返回储存在 `double` 类型数组中最大值和最小值的差值，并在一个简单的程序中测试该函数。
6. 编写一个函数，把 `double` 类型数组中的数据倒序排列，并在一个简单的程序中测试该函数。
7. 编写一个程序，初始化一个 `double` 类型的二维数组，使用编程练习 2 中的一个拷贝函数把该数组中的数据拷贝至另一个二维数组中（因为二维数组是数组的数组，所以可以使用处理一维数组的拷贝函数来处理数组中的每个子数组）。
8. 使用编程练习 2 中的拷贝函数，把一个内含 7 个元素的数组中第 3~第 5 个元素拷贝至内含 3 个元素的数组中。该函数本身不需要修改，只需要选择合适实际参数（实际参数不需要是数组名和数组大小，只需要是数组元素的地址和待处理元素的个数）。
9. 编写一个程序，初始化一个 `double` 类型的  $3 \times 5$  二维数组，使用一个处理变长数组的函数将其拷贝至另一个二维数组中。还要编写一个以变长数组为形参的函数以显示两个数组的内容。这两个函数应该能处理任意  $N \times M$  数组（如果编译器不支持变长数组，就使用传统 C 函数处理  $N \times 5$  的数组）。
10. 编写一个函数，把两个数组中相对应的元素相加，然后把结果储存到第 3 个数组中。也就是说，如果数组 1 中包含的值是 2、4、5、8，数组 2 中包含的值是 1、0、4、6，那么该函数把 3、4、9、14 赋给第 3 个数组。函数接受 3 个数组名和一个数组大小。在一个简单的程序中测试该函数。
11. 编写一个程序，声明一个 `int` 类型的  $3 \times 5$  二维数组，并用合适的值初始化它。该程序打印数组中的值，然后各值翻倍（即是原值的 2 倍），并显示出各元素的新值。编写一个函数显示数组的内容，再编写一个函数把各元素的值翻倍。这两个函数都以函数名和行数作为参数。
12. 重写程序清单 10.7 的 `rain.c` 程序，把 `main()` 中的主要任务都改成用函数来完成。
13. 编写一个程序，提示用户输入 3 组数，每组数包含 5 个 `double` 类型的数（假设用户都正确地响应，不会输入非数值数据）。该程序应完成下列任务。
  - a. 把用户输入的数据储存在  $3 \times 5$  的数组中
  - b. 计算每组（5 个）数据的平均值
  - c. 计算所有数据的平均值
  - d. 找出这 15 个数据中的最大值

e. 打印结果

每个任务都要用单独的函数来完成（使用传统 C 处理数组的方式）。完成任务 b，要编写一个计算并返回一维数组平均值的函数，利用循环调用该函数 3 次。对于处理其他任务的函数，应该把整个数组作为参数，完成任务 c 和 d 的函数应把结果返回主调函数。

14. 以变长数组作为函数形参，完成编程练习 13。



# 第 11 章

## 字符串和字符串函数

本章介绍以下内容：

- 函数：gets()、gets\_s()、fgets()、puts()、fputs()、strcat()、strncat()、strcmp()、strncmp()、strcpy()、strncpy()、sprintf()、strchr()
- 创建并使用字符串
- 使用 C 库中的字符和字符串函数，并创建自定义的字符串函数
- 使用命令行参数

字符串是 C 语言中最有用、最重要的数据类型之一。虽然我们一直在使用字符串，但是要学的东西还有很多。C 库提供大量的函数用于读写字符串、拷贝字符串、比较字符串、合并字符串、查找字符串等。通过本章的学习，读者将进一步提高自己的编程水平。

### 11.1 表示字符串和字符串 I/O

第 4 章介绍过，字符串是以空字符 (\0) 结尾的 char 类型数组。因此，可以把上一章学到的数组和指针的知识应用于字符串。不过，由于字符串十分常用，所以 C 提供了许多专门用于处理字符串的函数。本章将讨论字符串的性质、如何声明并初始化字符串、如何在程序中输入和输出字符串，以及如何操控字符串。

程序清单 11.1 演示了在程序中表示字符串的几种方式。

程序清单 11.1 strings1.c 程序

```
// strings1.c
#include <stdio.h>
#define MSG "I am a symbolic string constant."
#define MAXLENGTH 81
int main(void)
{
 char words[MAXLENGTH] = "I am a string in an array.";
 const char * pt1 = "Something is pointing at me.";
 puts("Here are some strings:");
 puts(MSG);
 puts(words);
 puts(pt1);
 words[8] = 'p';
 puts(words);

 return 0;
}
```

和 printf() 函数一样，puts() 函数也属于 stdio.h 系列的输入/输出函数。但是，与 printf()

不同的是, `puts()` 函数只显示字符串, 而且自动在显示的字符串末尾加上换行符。下面是该程序的输出:

```
Here are some strings:
I am an old-fashioned symbolic string constant.
I am a string in an array.
Something is pointing at me.
I am a spring in an array.
```

我们先分析一下该程序中定义字符串的几种方法, 然后再讲解把字符串读入程序涉及的一些操作, 最后学习如何输出字符串。

### 11.1.1 在程序中定义字符串

程序清单 11.1 中使用了多种方法 (即字符串常量、`char` 类型数组、指向 `char` 的指针) 定义字符串。程序应该确保有足够的空间储存字符串, 这一点我们稍后讨论。

#### 1. 字符串字面量 (字符串常量)

用双引号括起来的内容称为字符串字面量 (*string literal*), 也叫作字符串常量 (*string constant*)。双引号中的字符和编译器自动加入末尾的 `\0` 字符, 都作为字符串储存在内存中, 所以 "`I am a symbolic stringconstant.`"、"`I am a string in an array.`"、"`Something is pointed at me.`"、"`Here are some strings:`" 都是字符串字面量。

从 ANSI C 标准起, 如果字符串字面量之间没有间隔, 或者用空白字符分隔, C 会将其视为串联起来的字符串字面量。例如:

```
char greeting[50] = "Hello, and"" how are" " you"
 " today!";
```

与下面的代码等价:

```
char greeting[50] = "Hello, and how are you today!";
```

如果要在字符串内部使用双引号, 必须在双引号前面加上一个反斜杠 (\):

```
printf("\"Run, Spot, run!\" exclaimed Dick.\n");
```

输出如下:

```
"Run, Spot, run!" exclaimed Dick.
```

字符串常量属于静态存储类别 (*static storage class*), 这说明如果在函数中使用字符串常量, 该字符串只会被储存一次, 在整个程序的生命期内存在, 即使函数被调用多次。用双引号括起来的内容被视为指向该字符串储存位置的指针。这类似于把数组名作为指向该数组位置的指针。如果确实如此, 程序清单 11.2 中的程序会输出什么?

#### 程序清单 11.2 strptr.c 程序

---

```
/* strptr.c -- 把字符串看作指针 */
#include <stdio.h>
int main(void)
{
 printf("%s, %p, %c\n", "We", "are", *"space farers");

 return 0;
}
```

---

`printf()` 根据 `%s` 转换说明打印 `We`, 根据 `%p` 转换说明打印一个地址。因此, 如果 "`are`" 代表一个地址, `printf()` 将打印该字符串首字符的地址 (如果使用 ANSI 之前的实现, 可能要用 `%u` 或 `%lu` 代替 `%p`)。

最后，`*"space farers"`表示该字符串所指向地址上储存的值，应该是字符串`"space farers"`的首字符。是否真的是这样？下面是该程序的输出：

```
We, 0x100000f61, s
```

## 2. 字符串数组和初始化

定义字符串数组时，必须让编译器知道需要多少空间。一种方法是用足够空间的数组储存字符串。在下面的声明中，用指定的字符串初始化数组`m1`：

```
const char m1[40] = "Limit yourself to one line's worth.;"
```

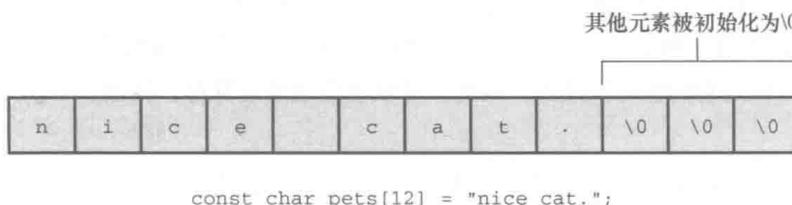
`const` 表明不会更改这个字符串。

这种形式的初始化比标准的数组初始化形式简单得多：

```
const char m1[40] = { 'L', 'i', 'm', ' ', 't', ' ', 'y', 'o', 'u', 'r', 's', 'e', 'l',
 'f', ' ', 't', 'o', ' ', 'o', 'n', 'e', ' ', 'l', 'i', 'n', 'e',
 '\\", 's', ' ', 'w', 'o', 'r', 't', 'h', '.', '\0' };
```

注意最后的空字符。没有这个空字符，这就不是一个字符串，而是一个字符数组。

在指定数组大小时，要确保数组的元素个数至少比字符串长度多 1（为了容纳空字符）。所有未被使用的元素都被自动初始化为 0（这里的 0 指的是 `char` 形式的空字符，不是数字字符 0），如图 11.1 所示。



```
const char pets[12] = "nice cat.;"
```

图 11.1 初始化数组

通常，让编译器确定数组的大小很方便。回忆一下，省略数组初始化声明中的大小，编译器会自动计算数组的大小：

```
const char m2[] = "If you can't think of anything, fake it.;"
```

让编译器确定初始化字符数组的大小很合理。因为处理字符串的函数通常都不知道数组的大小，这些函数通过查找字符串末尾的空字符确定字符串在何处结束。

让编译器计算数组的大小只能用在初始化数组时。如果创建一个稍后再填充的数组，就必须在声明时指定大小。声明数组时，数组大小必须是可求值的整数。在 C99 新增变长数组之前，数组的大小必须是整型常量，包括由整型常量组成的表达式。

```
int n = 8;
char cookies[1]; // 有效
char cakes[2 + 5]; // 有效，数组大小是整型常量表达式
char pies[2*sizeof(long double) + 1]; // 有效
char crumbs[n]; // 在 C99 标准之前无效，C99 标准之后这种数组是变长数组
```

字符数组名和其他数组名一样，是该数组首元素的地址。因此，假设有下面的初始化：

```
char car[10] = "Tata";
```

那么，以下表达式都为真：

```
car == &car[0]、*car == 'T'、*(car+1) == car[1] == 'a'。
```

还可以使用指针表示法创建字符串。例如，程序清单 11.1 中使用了下面的声明：

```
const char * pt1 = "Something is pointing at me.;"
```

该声明和下面的声明几乎相同：

```
const char ar1[] = "Something is pointing at me.;"
```

以上两个声明表明，`pt1` 和 `ar1` 都是该字符串的地址。在这两种情况下，带双引号的字符串本身决定了预留给字符串的存储空间。尽管如此，这两种形式并不完全相同。

### 3. 数组和指针

数组形式和指针形式有何不同？以上面的声明为例，数组形式 (`ar1[]`) 在计算机的内存中分配为一个内含 29 个元素的数组（每个元素对应一个字符，还加上一个末尾的空字符 '`\0`'），每个元素被初始化为字符串字面量对应的字符。通常，字符串都作为可执行文件的一部分储存在数据段中。当把程序载入内存时，也载入了程序中的字符串。字符串储存在静态存储区 (*static memory*) 中。但是，程序在开始运行时才会为该数组分配内存。此时，才将字符串拷贝到数组中（第 12 章将详细讲解）。注意，此时字符串有两个副本。一个是在静态内存中的字符串字面量，另一个是储存在 `ar1` 数组中的字符串。

此后，编译器便把数组名 `ar1` 识别为该数组首元素地址 (`&ar1[0]`) 的别名。这里关键要理解，在数组形式中，`ar1` 是地址常量。不能更改 `ar1`，如果改变了 `ar1`，则意味着改变了数组的存储位置（即地址）。可以进行类似 `ar1+1` 这样的操作，标识数组的下一个元素。但是不允许进行 `++ar1` 这样的操作。递增运算符只能用于变量名前（或概括地说，只能用于可修改的左值），不能用于常量。

指针形式 (`*pt1`) 也使得编译器为字符串在静态存储区预留 29 个元素的空间。另外，一旦开始执行程序，它会为指针变量 `pt1` 留出一个储存位置，并把字符串的地址储存在指针变量中。该变量最初指向该字符串的首字符，但是它的值可以改变。因此，可以使用递增运算符。例如，`++pt1` 将指向第 2 个字符 (`o`)。

字符串字面量被视为 `const` 数据。由于 `pt1` 指向这个 `const` 数据，所以应该把 `pt1` 声明为指向 `const` 数据的指针。这意味着不能用 `pt1` 改变它所指向的数据，但是仍然可以改变 `pt1` 的值（即，`pt1` 指向的位置）。如果把一个字符串字面量拷贝给一个数组，就可以随意改变数据，除非把数组声明为 `const`。

总之，初始化数组把静态存储区的字符串拷贝到数组中，而初始化指针只把字符串的地址拷贝给指针。程序清单 11.3 演示了这一点。

程序清单 11.3 addresses.c 程序

```
// addresses.c -- 字符串的地址
#define MSG "I'm special"

#include <stdio.h>
int main()
{
 char ar[] = MSG;
 const char *pt = MSG;
 printf("address of \"I'm special\": %p \n", "I'm special");
 printf(" address ar: %p\n", ar);
 printf(" address pt: %p\n", pt);
 printf(" address of MSG: %p\n", MSG);
 printf("address of \"I'm special\": %p \n", "I'm special");

 return 0;
}
```

下面是在我们的系统中运行该程序后的输出：

```

address of "I'm special": 0x100000f10
 address ar: 0x7fff5fbff858
 address pt: 0x100000f10
 address of MSG: 0x100000f10
address of "I'm special": 0x100000f10

```

该程序的输出说明了什么？第一，pt 和 MSG 的地址相同，而 ar 的地址不同，这与我们前面讨论的内容一致。第二，虽然字符串字面量 "I'm special" 在程序的两个 printf() 函数中出现了两次，但是编译器只使用了一个存储位置，而且与 MSG 的地址相同。编译器可以把多次使用的相同字面量储存在一处或多处。另一个编译器可能在不同的位置储存 3 个 "I'm special"。第三，静态数据使用的内存与 ar 使用的动态内存不同。不仅值不同，特定编译器甚至使用不同的位数表示两种内存。

数组和指针表示字符串的区别是否很重要？通常不太重要，但是这取决于想用程序做什么。我们来进一步讨论这个主题。

#### 4. 数组和指针的区别

初始化字符数组来储存字符串和初始化指针来指向字符串有何区别（“指向字符串”的意思是指向字符串的首字符）？例如，假设有下面两个声明：

```

char heart[] = "I love Tillie!";
const char *head = "I love Millie!";

```

两者主要的区别是：数组名 heart 是常量，而指针名 head 是变量。那么，实际使用有什么区别？

首先，两者都可以使用数组表示法：

```

for (i = 0; i < 6; i++)
 putchar(heart[i]);
putchar('\n');
for (i = 0; i < 6; i++)
 putchar(head[i]);
putchar('\n');

```

上面两段代码的输出是：

```

I love
I love

```

其次，两者都能进行指针加法操作：

```

for (i = 0; i < 6; i++)
 putchar(*(heart + i));
putchar('\n');
for (i = 0; i < 6; i++)
 putchar(*(head + i));
putchar('\n');

```

输出如下：

```

I love
I love

```

但是，只有指针表示法可以进行递增操作：

```

while (*head != '\0') /* 在字符串末尾处停止 */
 putchar(*(head++)); /* 打印字符，指针指向下一个位置 */

```

这段代码的输出如下：

```
I love Millie!
```

假设想让 head 和 heart 统一，可以这样做：

```
head = heart; /* head 现在指向数组 heart */
```

这使得 head 指针指向 heart 数组的首元素。

但是，不能这样做：

```
heart = head; /* 非法构造，不能这样写 */
```

这类似于 `x = 3;` 和 `3 = x;` 的情况。赋值运算符的左侧必须是变量（或概括地说是可修改的左值），如 `*pt_int`。顺带一提，`head = heart;` 不会导致 head 指向的字符串消失，这样做只是改变了储存在 head 中的地址。除非已经保存了 "I love Millie!" 的地址，否则当 head 指向别处时，就无法再访问该字符串。

另外，还可以改变 heart 数组中元素的信息：

```
heart[7] = 'M'; 或者*(heart + 7) = 'M';
```

数组的元素是变量（除非数组被声明为 `const`），但是数组名不是变量。

我们来看一下未使用 `const` 限定符的指针初始化：

```
char * word = "frame";
```

是否能使用该指针修改这个字符串？

```
word[1] = 'l'; // 是否允许？
```

编译器可能允许这样做，但是对当前的 C 标准而言，这样的行为是未定义的。例如，这样的语句可能导致内存访问错误。原因前面提到过，编译器可以使用内存中的一个副本表示所有完全相同的字符串字面量。例如，下面的语句都引用字符串 "Klingon" 的一个内存位置：

```
char * p1 = "Klingon";
p1[0] = 'F'; // ok?
printf("Klingon");
printf(": Beware the %s!\n", "Klingon");
```

也就是说，编译器可以用相同的地址替换每个 "Klingon" 实例。如果编译器使用这种单次副本表示法，并允许 `p1[0]` 修改 'F'，那将影响所有使用该字符串的代码。所以上述语句打印字符串字面量 "Klingon" 时实际上显示的是 "Flingon"：

```
Flingon: Beware the Flingons!
```

实际上在过去，一些编译器由于这方面的原因，其行为难以捉摸，而另一些编译器则导致程序异常中断。因此，建议在把指针初始化为字符串字面量时使用 `const` 限定符：

```
const char * p1 = "Klingon"; // 推荐用法
```

然而，把非 `const` 数组初始化为字符串字面量却不会导致类似的问题。因为数组获得的是原始字符串的副本。

总之，如果不修改字符串，不要用指针指向字符串字面量。

## 5. 字符串数组

如果创建一个字符数组会很方便，可以通过数组下标访问多个不同的字符串。程序清单 11.4 演示了两种方法：指向字符串的指针数组和 `char` 类型数组的数组。

### 程序清单 11.4 arrchar.c 程序

```
// arrchar.c -- 指针数组，字符串数组
#include <stdio.h>
#define SLEN 40
#define LIM 5
int main(void)
{
```

```

const char *mytalents[LIM] = {
 "Adding numbers swiftly",
 "Multiplying accurately", "Stashing data",
 "Following instructions to the letter",
 "Understanding the C language"
};

char yourtalents[LIM][SLEN] = {
 "Walking in a straight line",
 "Sleeping", "Watching television",
 "Mailing letters", "Reading email"
};

int i;

puts("Let's compare talents.");
printf("%-36s %-25s\n", "My Talents", "Your Talents");
for (i = 0; i < LIM; i++)
 printf("%-36s %-25s\n", mytalents[i], yourtalents[i]);
printf("\nsizeof mytalents: %zd, sizeof yourtalents: %zd\n",
 sizeof(mytalents), sizeof(yourtalents));

return 0;
}

```

下面是该程序的输出：

|                                      |                            |
|--------------------------------------|----------------------------|
| Let's compare talents.               |                            |
| My Talents                           | Your Talents               |
| Adding numbers swiftly               | Walking in a straight line |
| Multiplying accurately               | Sleeping                   |
| Stashing data                        | Watching television        |
| Following instructions to the letter | Mailing letters            |
| Understanding the C language         | Reading email              |

sizeof mytalents: 40, sizeof yourtalents: 200

从某些方面来看，`mytalents` 和 `yourtalents` 非常相似。两者都代表 5 个字符串。使用一个下标时都分别表示一个字符串，如 `mytalents[0]` 和 `yourtalents[0]`；使用两个下标时都分别表示一个字符，例如 `mytalents[1][2]` 表示 `mytalents` 数组中第 2 个指针所指向的字符串的第 3 个字符 'l'，`yourtalents[1][2]` 表示 `yourtalents` 数组的第 2 个字符串的第 3 个字符 'e'。而且，两者的初始化方式也相同。

但是，它们也有区别。`mytalents` 数组是一个内含 5 个指针的数组，在我们的系统中共占用 40 字节。而 `yourtalents` 是一个内含 5 个数组的数组，每个数组内含 40 个 `char` 类型的值，共占用 200 字节。所以，虽然 `mytalents[0]` 和 `yourtalents[0]` 都分别表示一个字符串，但 `mytalents` 和 `yourtalents` 的类型并不相同。`mytalents` 中的指针指向初始化时所用的字符串字面量的位置，这些字符串字面量被储存在静态内存中；而 `yourtalents` 中的数组则储存着字符串字面量的副本，所以每个字符串都被储存了两次。此外，为字符串数组分配内存的使用率较低。`yourtalents` 中的每个元素的大小必须相同，而且必须是能储存最长字符串的大小。

我们可以把 `yourtalents` 想象成矩形二维数组，每行的长度都是 40 字节；把 `mytalents` 想象成不规则的数组，每行的长度不同。图 11.2 演示了这两种数组的情况（实际上，`mytalents` 数组的指针元素所指向的字符串不必储存在连续的内存中，图中所示只是为了强调两种数组的不同）。

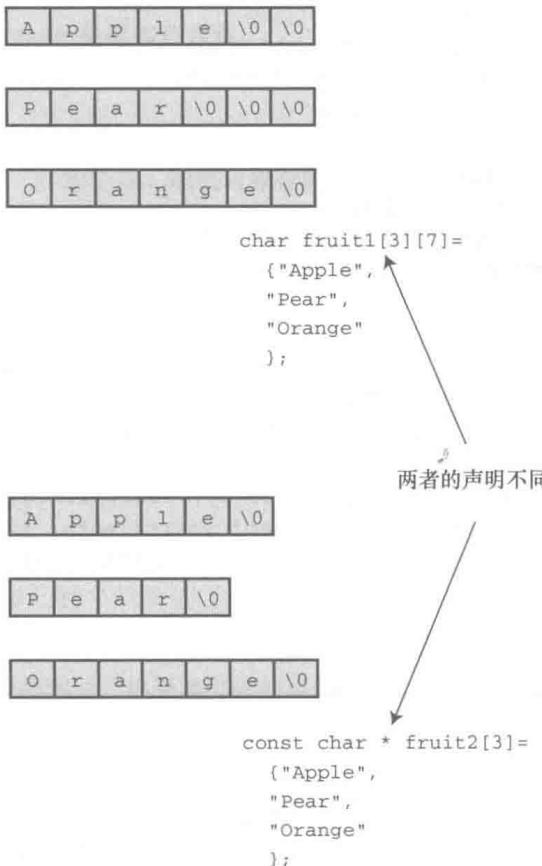


图 11.2 矩形数组和不规则数组

综上所述，如果要用数组表示一系列待显示的字符串，请使用指针数组，因为它比二维字符数组的效率高。但是，指针数组也有自身的缺点。`mytalents` 中的指针指向的字符串字面量不能更改；而 `yourtalentsde` 中的内容可以更改。所以，如果要改变字符串或为字符串输入预留空间，不要使用指向字符串字面量的指针。

### 11.1.2 指针和字符串

读者可能已经注意到了，在讨论字符串时或多或少会涉及指针。实际上，字符串的绝大多数操作都是通过指针完成的。例如，考虑程序清单 11.5 中的程序。

程序清单 11.5 p\_and\_s.c 程序

```
/* p_and_s.c -- 指针和字符串 */
#include <stdio.h>
int main(void)
{
 const char * mesg = "Don't be a fool!";
 const char * copy;

 copy = mesg;
 printf("%s\n", copy);
 printf("mesg = %s; &mesg = %p; value = %p\n", mesg, &mesg, mesg);
 printf("copy = %s; © = %p; value = %p\n", copy, ©, copy);
```

```
 return 0;
}
```

## 注意

如果编译器不识别%p, 用%u或%lu代替%p。

你可能认为该程序拷贝了字符串"Don't be a fool!", 程序的输出似乎也验证了你的猜测:

```
Don't be a fool!
mesg = Don't be a fool!; &mesg = 0x0012ff48; value = 0x0040a000
copy = Don't be a fool!; © = 0x0012ff44; value = 0x0040a000
```

我们来仔细分析最后两个 printf() 的输出。首先第 1 项, mesg 和 copy 都以字符串形式输出 (%s 转换说明)。这里没问题, 两个字符串都是"Don't be a fool!"。

接着第 2 项, 打印两个指针的地址。如上输出所示, 指针 mesg 和 copy 分别储存在地址为 0x0012ff48 和 0x0012ff44 的内存中。

注意最后一项, 显示两个指针的值。所谓指针的值就是它储存的地址。mesg 和 copy 的值都是 0x0040a000, 说明它们都指向同一个位置。因此, 程序并未拷贝字符串。语句 copy = mesg; 把 mesg 的值赋给 copy, 即让 copy 也指向 mesg 指向的字符串。

为什么要这样做? 为何不拷贝整个字符串? 假设数组有 50 个元素, 考虑一下哪种方法更效率: 拷贝一个地址还是拷贝整个数组? 通常, 程序要完成某项操作只需要知道地址就可以了。如果确实需要拷贝整个数组, 可以使用 strcpy() 或 strncpy() 函数, 本章稍后介绍这两个函数。

我们已经讨论了如何在程序中定义字符串, 接下来看看如何从键盘输入字符串。

## 11.2 字符串输入

如果想把一个字符串读入程序, 首先必须预留储存该字符串的空间, 然后用输入函数获取该字符串。

### 11.2.1 分配空间

要做的第 1 件事是分配空间, 以储存稍后读入的字符串。前面提到过, 这意味着必须要为字符串分配足够的空间。不要指望计算机在读取字符串时顺便计算它的长度, 然后再分配空间(计算机不会这样做, 除非你编写一个处理这些任务的函数)。假设编写了如下代码:

```
char *name;
scanf ("%s", name);
```

虽然可能会通过编译(编译器很可能给出警告), 但是在读入 name 时, name 可能会擦写掉程序中的数据或代码, 从而导致程序异常中止。因为 scanf() 要把信息拷贝至参数指定的地址上, 而此时该参数是个未初始化的指针, name 可能会指向任何地方。大多数程序员都认为出现这种情况很搞笑, 但仅限于评价别人的程序时。

最简单的方法是在声明时显式指明数组的大小:

```
char name[81];
```

现在 name 是一个已分配块(81 字节)的地址。还有一种方法是使用 C 库函数来分配内存, 第 12 章将详细介绍。

为字符串分配内存后，便可读入字符串。C 库提供了许多读取字符串的函数：`scanf()`、`gets()` 和 `fgets()`。我们先讨论最常用 `gets()` 函数。

## 11.2.2 不幸的 `gets()` 函数

在读取字符串时，`scanf()` 和转换说明`%s` 只能读取一个单词。可是在程序中经常要读取一整行输入，而不仅仅是一个单词。许多年前，`gets()` 函数就用于处理这种情况。`gets()` 函数简单易用，它读取整行输入，直至遇到换行符，然后丢弃换行符，储存其余字符，并在这些字符的末尾添加一个空字符使其成为一个 C 字符串。它经常和 `puts()` 函数配对使用，该函数用于显示字符串，并在末尾添加换行符。程序清单 11.6 中演示了这两个函数的用法。

程序清单 11.6 `getsputs.c` 程序

```
/* getsputs.c -- 使用 gets() 和 puts() */
#include <stdio.h>
#define STLEN 81
int main(void)
{
 char words[STLEN];

 puts("Enter a string, please.");
 gets(words); // 典型用法
 printf("Your string twice:\n");
 printf("%s\n", words);
 puts(words);
 puts("Done.");

 return 0;
}
```

下面是该程序在某些编译器（或者至少是旧式编译器）中的运行示例：

```
Enter a string, please.
I want to learn about string theory!
Your string twice:
I want to learn about string theory!
I want to learn about string theory!
Done.
```

整行输入（除了换行符）都被储存在 `words` 中，`puts(words)` 和 `printf("%s\n, words")` 的效果相同。

下面是该程序在另一个编译器中的输出示例：

```
Enter a string, please.
warning: this program uses gets(), which is unsafe.
Oh, no!
Your string twice:
Oh, no!
Oh, no!
Done.
```

编译器在输出中插入了一行警告消息。每次运行这个程序，都会显示这行消息。但是，并非所有的编译器都会这样做。其他编译器可能在编译过程中给出警告，但不会引起你的注意。

这是怎么回事？问题出在 `gets()` 唯一的参数是 `words`，它无法检查数组是否装得下输入行。上一章

介绍过，数组名会被转换成该数组首元素的地址，因此，`gets()`函数只知道数组的开始处，并不知道数组中有多少个元素。

如果输入的字符串过长，会导致缓冲区溢出（*buffer overflow*），即多余的字符超出了指定的目标空间。如果这些多余的字符只是占用了尚未使用的内存，就不会立即出现问题；如果它们擦写掉程序中的其他数据，会导致程序异常中止；或者还有其他情况。为了让输入的字符串容易溢出，把程序中的`STLEN`设置为5，程序的输出如下：

```
Enter a string, please.
warning: this program uses gets(), which is unsafe.
I think I'll be just fine.
Your string twice:
I think I'll be just fine.
I think I'll be just fine.
Done.
Segmentation fault: 11
```

“Segmentation fault”（分段错误）似乎不是个好提示，的确如此。在UNIX系统中，这条消息说明该程序试图访问未分配的内存。

C 提供解决某些编程问题的方法可能会导致陷入另一个尴尬棘手的困境。但是，为什么要特别提到`gets()`函数？因为该函数的不安全行为造成了安全隐患。过去，有些人通过系统编程，利用`gets()`插入和运行一些破坏系统安全的代码。

不久，C 编程社区的许多人都建议在编程时摒弃`gets()`。制定 C99 标准的委员会把这些建议放入了标准，承认了`gets()`的问题并建议不要再使用它。尽管如此，在标准中保留`gets()`也合情合理，因为现有程序中含有大量使用该函数的代码。而且，只要使用得当，它的确是一个很方便的函数。

好景不长，C11 标准委员会采取了更强硬的态度，直接从标准中废除了`gets()`函数。既然标准已经发布，那么编译器就必须根据标准来调整支持什么，不支持什么。然而在实际应用中，编译器为了能兼容以前的代码，大部分都继续支持`gets()`函数。不过，我们使用的编译器，可没那么大方。

### 11.2.3 `gets()`的替代品

过去通常用`fgets()`来代替`gets()`，`fgets()`函数稍微复杂些，在处理输入方面与`gets()`略有不同。C11 标准新增的`gets_s()`函数也可代替`gets()`。该函数与`gets()`函数更接近，而且可以替换现有代码中的`gets()`。但是，它是`stdio.h`输入/输出函数系列中的可选扩展，所以支持 C11 的编译器也不一定支持它。

#### 1. `fgets()`函数（和`fputs()`）

`fgets()`函数通过第 2 个参数限制读入的字符数来解决溢出的问题。该函数专门设计用于处理文件输入，所以一般情况下可能不太好用。`fgets()`和`gets()`的区别如下。

- `fgets()`函数的第 2 个参数指明了读入字符的最大数量。如果该参数的值是 n，那么`fgets()`将读入 n-1 个字符，或者读到遇到的第一个换行符为止。
- 如果`fgets()`读到一个换行符，会把它储存在字符串中。这点与`gets()`不同，`gets()`会丢弃换行符。
- `fgets()`函数的第 3 个参数指明要读入的文件。如果读入从键盘输入的数据，则以`stdin`（标准输入）作为参数，该标识符定义在`stdio.h`中。

因为`fgets()`函数把换行符放在字符串的末尾（假设输入行不溢出），通常要与`fputs()`函数（和

`puts()` 类似) 配对使用, 除非该函数不在字符串末尾添加换行符。`fputs()` 函数的第 2 个参数指明它要写入的文件。如果要显示在计算机显示器上, 应使用 `stdout` (标准输出) 作为该参数。程序清单 11.7 演示了 `fgets()` 和 `fputs()` 函数的用法。

程序清单 11.7 `fgetsl.c` 程序

```
/* fgetsl.c -- 使用 fgets() 和 fputs() */
#include <stdio.h>
#define STLEN 14
int main(void)
{
 char words[STLEN];

 puts("Enter a string, please.");
 fgets(words, STLEN, stdin);
 printf("Your string twice (puts(), then fputs()):\n");
 puts(words);
 fputs(words, stdout);
 puts("Enter another string, please.");
 fgets(words, STLEN, stdin);
 printf("Your string twice (puts(), then fputs()):\n");
 puts(words);
 fputs(words, stdout);
 puts("Done.");
}

return 0;
}
```

下面是该程序的输出示例:

```
Enter a string, please.
apple pie
Your string twice (puts(), then fputs()):
apple pie

apple pie
Enter another string, please.
strawberry shortcake
Your string twice (puts(), then fputs()):
strawberry sh
strawberry shDone.
```

第 1 行输入, `apple pie`, 比 `fgets()` 读入的整行输入短, 因此, `apple pie\n\0` 被储存在数组中。所以当 `puts()` 显示该字符串时又在末尾添加了换行符, 因此 `apple pie` 后面有一行空行。因为 `fputs()` 不在字符串末尾添加换行符, 所以并未打印出空行。

第 2 行输入, `strawberry shortcake`, 超过了大小的限制, 所以 `fgets()` 只读入了 13 个字符, 并把 `strawberry sh\0` 储存在数组中。再次提醒读者注意, `puts()` 函数会在待输出字符串末尾添加一个换行符, 而 `fputs()` 不会这样做。

`fputs()` 函数返回指向 `char` 的指针。如果一切进行顺利, 该函数返回的地址与传入的第 1 个参数相同。但是, 如果函数读到文件结尾, 它将返回一个特殊的指针: 空指针 (*null pointer*)。该指针保证不会指向有效的数据, 所以可用于标识这种特殊情况。在代码中, 可以用数字 0 来代替, 不过在 C 语言中用宏 `NUL` 来代替更常见 (如果在读入数据时出现某些错误, 该函数也返回 `NUL`)。程序清单 11.8 演示

为了一个简单的循环，读入并显示用户输入的内容，直到 `fgets()` 读到文件结尾或空行（即，首字符是换行符）。

#### 程序清单 11.8 fgets2.c 程序

---

```
/* fgets2.c -- 使用 fgets() 和 fputs() */
#include <stdio.h>
#define STLEN 10
int main(void)
{
 char words[STLEN];

 puts("Enter strings (empty line to quit):");
 while (fgets(words, STLEN, stdin) != NULL && words[0] != '\n')
 fputs(words, stdout);
 puts("Done.");
}

return 0;
}
```

---

下面是该程序的输出示例：

```
Enter strings (empty line to quit):
By the way, the gets() function
By the way, the gets() function
also returns a null pointer if it
also returns a null pointer if it
encounters end-of-file.
encounters end-of-file.
```

Done.

有意思，虽然 `STLEN` 被设置为 10，但是该程序似乎在处理过长的输入时完全没问题。程序中的 `fgets()` 一次读入 `STLEN - 1` 个字符（该例中为 9 个字符）。所以，一开始它只读入了“`By the wa`”，并储存为 `By the wa\0`；接着 `fputs()` 打印该字符串，而且并未换行。然后 `while` 循环进入下一轮迭代，`fgets()` 继续从剩余的输入中读入数据，即读入“`y, the ge`”并储存为 `y, the ge\0`；接着 `fputs()` 在刚才打印字符串的这一行接着打印第 2 次读入的字符串。然后 `while` 进入下一轮迭代，`fgets()` 继续读取输入、`fputs()` 打印字符串，这一过程循环进行，直到读入最后的“`tion\n`”。`fgets()` 将其储存为 `tion\n\0`，`fputs()` 打印该字符串，由于字符串中的 `\n`，光标被移至下一行开始处。

系统使用缓冲的 I/O。这意味着用户在按下 **Return** 键之前，输入都被储存在临时存储区（即，缓冲区）中。按下 **Return** 键就在输入中增加了一个换行符，并把整行输入发送给 `fgets()`。对于输出，`fputs()` 把字符发送给另一个缓冲区，当发送换行符时，缓冲区中的内容被发送至屏幕上。

`fgets()` 储存换行符有好处也有坏处。坏处是你可能并不想把换行符储存在字符串中，这样的换行符会带来一些麻烦。好处是对于储存的字符串而言，检查末尾是否有换行符可以判断是否读取了一整行。如果不是一整行，要妥善处理一行中剩下的字符。

首先，如何处理掉换行符？一个方法是在已储存的字符串中查找换行符，并将其替换成空字符：

```
while (words[i] != '\n') // 假设\n在words中
 i++;
words[i] = '\0';
```

其次，如果仍有字符串留在输入行怎么办？一个可行的办法是，如果目标数组装不下一整行输入，就

丢弃那些多出的字符：

```
while (getchar() != '\n') // 读取但不储存输入，包括\n
 continue;
```

程序清单 11.9 在程序清单 11.8 的基础上添加了一部分测试代码。该程序读取输入行，删除储存在字符串中的换行符，如果没有换行符，则丢弃数组装不下的字符。

程序清单 11.9 fgets3.c 程序

---

```
/* fgets3.c -- 使用 fgets() */
#include <stdio.h>
#define STLEN 10
int main(void)
{
 char words[STLEN];
 int i;

 puts("Enter strings (empty line to quit):");
 while (fgets(words, STLEN, stdin) != NULL && words[0] != '\n')
 {
 i = 0;
 while (words[i] != '\n' && words[i] != '\0')
 i++;
 if (words[i] == '\n')
 words[i] = '\0';
 else // 如果 word[i] == '\0' 则执行这部分代码
 while (getchar() != '\n')
 continue;
 puts(words);
 }
 puts("done");
 return 0;
}
```

---

### 循环

```
while (words[i] != '\n' && words[i] != '\0')
 i++;
```

遍历字符串，直至遇到换行符或空字符。如果先遇到换行符，下面的 `if` 语句就将其替换成空字符；如果先遇到空字符，`else` 部分便丢弃输入行的剩余字符。下面是该程序的输出示例：

```
Enter strings (empty line to quit):
```

```
This
This
program seems
program s
unwilling to accept long lines.
unwilling
But it doesn't get stuck on long
But it do
lines either.
lines eit
```

```
done
```

### 空字符和空指针

程序清单 11.9 中出现了空字符和空指针。从概念上看，两者完全不同。空字符（或'\\0'）是用于标记 C 字符串末尾的字符，其对应字符编码是 0。由于其他字符的编码不可能是 0，所以不可能是字符串的一部分。

空指针（或 NULL）有一个值，该值不会与任何数据的有效地址对应。通常，函数使用它返回一个有效地址表示某些特殊情况发生，例如遇到文件结尾或未能按预期执行。

空字符是整数类型，而空指针是指针类型。两者有时容易混淆的原因是：它们都可以用数值 0 来表示。但是，从概念上看，两者是不同类型的 0。另外，空字符是一个字符，占 1 字节；而空指针是一个地址，通常占 4 字节。

## 2. gets\_s() 函数

C11 新增的 gets\_s() 函数（可选）和 fgets() 类似，用一个参数限制读入的字符数。假设把程序清单 11.9 中的 fgets() 换成 gets\_s()，其他内容不变，那么下面的代码将把一行输入中的前 9 个字符读入 words 数组中，假设末尾有换行符：

```
gets_s(words, STLEN);
```

gets\_s() 与 fgets() 的区别如下。

- gets\_s() 只从标准输入中读取数据，所以不需要第 3 个参数。
- 如果 gets\_s() 读到换行符，会丢弃它而不是储存它。
- 如果 gets\_s() 读到最大字符数都没有读到换行符，会执行以下几步。首先把目标数组中的首字符设置为空字符，读取并丢弃随后的输入直至读到换行符或文件结尾，然后返回空指针。接着，调用依赖实现的“处理函数”（或你选择的其他函数），可能会中止或退出程序。

第 2 个特性说明，只要输入行未超过最大字符数，gets\_s() 和 gets() 几乎一样，完全可以用 gets\_s() 替换 gets()。第 3 个特性说明，要使用这个函数还需要进一步学习。

我们来比较一下 gets()、fgets() 和 gets\_s() 的适用性。如果目标存储区装得下输入行，3 个函数都没问题。但是 fgets() 会保留输入末尾的换行符作为字符串的一部分，要编写额外的代码将其替换成空字符。

如果输入行太长会怎样？使用 gets() 不安全，它会擦写现有数据，存在安全隐患。gets\_s() 函数很安全，但是，如果并不希望程序中止或退出，就要知道如何编写特殊的“处理函数”。另外，如果打算让程序继续运行，gets\_s() 会丢弃该输入行的其余字符，无论你是否需要。由此可见，当输入太长，超过数组可容纳的字符数时，fgets() 函数最容易使用，而且可以选择不同的处理方式。如果要让程序继续使用输入行中超出的字符，可以参考程序清单 11.8 中的处理方法。如果想丢弃输入行的超出字符，可以参考程序清单 11.9 中的处理方法。

所以，当输入与预期不符时，gets\_s() 完全没有 fgets() 函数方便、灵活。也许这也是 gets\_s() 只作为 C 库的可选扩展的原因之一。鉴于此，fgets() 通常是处理类似情况的最佳选择。

## 3. s\_gets() 函数

程序清单 11.9 演示了 fgets() 函数的一种用法：读取整行输入并用空字符代替换行符，或者读取一部分输入，并丢弃其余部分。既然没有处理这种情况的标准函数，我们就创建一个，在后面的程序中会用得上。程序清单 11.10 提供了一个这样的函数。

## 程序清单 11.10 s\_gets() 函数

```

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val) // 即, ret_val != NULL
 {
 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

如果 `fgets()` 返回 `NULL`, 说明读到文件结尾或出现读取错误, `s_gets()` 函数跳过了这个过程。它模仿程序清单 11.9 的处理方法, 如果字符串中出现换行符, 就用空字符替换它; 如果字符串中出现空字符, 就丢弃该输入行的其余字符, 然后返回与 `fgets()` 相同的值。我们在后面的示例中将讨论 `fgets()` 函数。

也许读者想了解为什么要丢弃过长输入行中的余下字符。这是因为, 输入行中多出来的字符会被留在缓冲区中, 成为下一次读取语句的输入。例如, 如果下一条读取语句要读取的是 `double` 类型的值, 就可能导致程序崩溃。丢弃输入行余下的字符保证了读取语句与键盘输入同步。

我们设计的 `s_gets()` 函数并不完美, 它最严重的缺陷是遇到不合适的输入时毫无反应。它丢弃多余的字符时, 既不通知程序也不告知用户。但是, 用来替换前面程序示例中的 `gets()` 足够了。

#### 11.2.4 scanf() 函数

我们再来研究一下 `scanf()`。前面的程序中用 `scanf()` 和 `%s` 转换说明读取字符串。`scanf()` 和 `gets()` 或 `fgets()` 的区别在于它们如何确定字符串的末尾: `scanf()` 更像是“获取单词”函数, 而不是“获取字符串”函数; 如果预留的存储区装得下输入行, `gets()` 和 `fgets()` 会读取第 1 个换行符之前所有的字符。`scanf()` 函数有两种方法确定输入结束。无论哪种方法, 都从第 1 个非空白字符作为字符串的开始。如果使用 `%s` 转换说明, 以下一个空白字符(空行、空格、制表符或换行符)作为字符串的结束(字符串不包括空白字符)。如果指定了字段宽度, 如 `%10s`, 那么 `scanf()` 将读取 10 个字符或读到第 1 个空白字符停止(先满足的条件即是结束输入的条件), 见图 11.3。

| 输入语句                             | 原输入序列*       | name中的内容 | 剩余输入序列  |
|----------------------------------|--------------|----------|---------|
| <code>scanf('%s', name);</code>  | Fleebert□Hup | Fleebert | □Hup    |
| <code>scanf('%5s', name);</code> | Fleebert□Hup | Fleeb    | ert□Hup |
| <code>scanf('%5s', name);</code> | Ann□Ular     | Ann      | □Ular   |

\*□表示空格字符

图 11.3 字段宽度和 `scanf()`

前面介绍过，`scanf()`函数返回一个整数值，该值等于`scanf()`成功读取的项数或EOF（读到文件结尾时返回EOF）。

程序清单11.11演示了在`scanf()`函数中指定字段宽度的用法。

程序清单11.11 `scan_str.c`程序

---

```
/* scan_str.c -- 使用 scanf() */
#include <stdio.h>
int main(void)
{
 char name1[11], name2[11];
 int count;

 printf("Please enter 2 names.\n");
 count = scanf("%5s %10s", name1, name2);
 printf("I read the %d names %s and %s.\n", count, name1, name2);

 return 0;
}
```

---

下面是该程序的3个输出示例：

```
Please enter 2 names.
Jesse Jukes
I read the 2 names Jesse and Jukes.
```

```
Please enter 2 names.
Liza Applebottham
I read the 2 names Liza and Applebotth.
```

```
Please enter 2 names.
Portensia Callowit
I read the 2 names Porte and nsia.
```

第1个输出示例，两个名字的字符个数都未超过字段宽度。第2个输出示例，只读入了Applebottham的前10个字符Applebotth（因为使用了%10s转换说明）。第3个输出示例，Portensia的后4个字符nsia被写入name2中，因为第2次调用`scanf()`时，从上一次调用结束的地方继续读取数据。在该例中，读取的仍是Portensia中的字母。

根据输入数据的性质，用`fgets()`读取从键盘输入的数据更合适。例如，`scanf()`无法完整读取书名或歌曲名，除非这些名称是一个单词。`scanf()`的典型用法是读取并转换混合数据类型为某种标准形式。例如，如果输入行包含一种工具名、库存量和单价，就可以使用`scanf()`。否则可能要自己拼凑一个函数处理一些输入检查。如果一次只输入一个单词，用`scanf()`也没问题。

`scanf()`和`gets()`类似，也存在一些潜在的缺点。如果输入行的内容过长，`scanf()`也会导致数据溢出。不过，在%`s`转换说明中使用字段宽度可防止溢出。

## 11.3 字符串输出

讨论完字符串输入，接下来我们讨论字符串输出。C有3个标准库函数用于打印字符串：`put()`、`fputs()`和`printf()`。

### 11.3.1 puts() 函数

puts() 函数很容易使用，只需把字符串的地址作为参数传递给它即可。程序清单 11.12 演示了 puts() 的一些用法。

程序清单 11.12 put\_out.c 程序

---

```
/* put_out.c -- 使用 puts() */
#include <stdio.h>
#define DEF "I am a #defined string."
int main(void)
{
 char str1[80] = "An array was initialized to me.";
 const char * str2 = "A pointer was initialized to me.";

 puts("I'm an argument to puts().");
 puts(DEF);
 puts(str1);
 puts(str2);
 puts(&str1[5]);
 puts(str2 + 4);

 return 0;
}
```

---

该程序的输出如下：

```
I'm an argument to puts().
I am a #defined string.
An array was initialized to me.
A pointer was initialized to me.
ray was initialized to me.
inter was initialized to me.
```

如上所示，每个字符串独占一行，因为 puts() 在显示字符串时会自动在其末尾添加一个换行符。

该程序示例再次说明，用双引号括起来的内容是字符串常量，且被视为该字符串的地址。另外，储存字符串的数组名也被看作是地址。在第 5 个 puts() 调用中，表达式&str1[5] 是 str1 数组的第 6 个元素 (r)，puts() 从该元素开始输出。与此类似，第 6 个 puts() 调用中，str2+4 指向储存"pointer"中 i 的存储单元，puts() 从这里开始输出。

puts() 如何知道在何处停止？该函数在遇到空字符时就停止输出，所以必须确保有空字符。不要模仿程序清单 11.13 中的程序！

程序清单 11.13 nono.c 程序

---

```
/* nono.c -- 千万不要模仿！ */
#include <stdio.h>
int main(void)
{
 char side_a[] = "Side A";
 char dont[] = { 'W', 'O', 'W', '!' };
 char side_b[] = "Side B";

 puts(dont); /* dont 不是一个字符串 */

 return 0;
}
```

---

由于 `dont` 缺少一个表示结束的空字符，所以它不是一个字符串，因此 `puts()` 不知道在何处停止。它会一直打印 `dont` 后面内存中的内容，直到发现一个空字符为止。为了让 `puts()` 能尽快读到空字符，我们把 `dont` 放在 `side_a` 和 `side_b` 之间。下面是该程序的一个运行示例：

```
WOW!Side A
```

我们使用的编译器把 `side_a` 数组储存在 `dont` 数组之后，所以 `puts()` 一直输出至遇到 `side_a` 中的空字符。你所使用的编译器输出的内容可能不同，这取决于编译器如何在内存中储存数据。如果删除程序中的 `side_a` 和 `side_b` 数组会怎样？通常内存中有许多空字符，如果幸运的话，`puts()` 很快就会发现一个。但是，这样做很不靠谱。

### 11.3.2 `fputs()` 函数

`fputs()` 函数是 `puts()` 针对文件定制的版本。它们的区别如下。

- `fputs()` 函数的第 2 个参数指明要写入数据的文件。如果要打印在显示器上，可以用定义在 `stdio.h` 中的 `stdout`（标准输出）作为该参数。
- 与 `puts()` 不同，`fputs()` 不会在输出的末尾添加换行符。

注意，`gets()` 丢弃输入中的换行符，但是 `puts()` 在输出中添加换行符。另一方面，`fgets()` 保留输入中的换行符，`fputs()` 不在输出中添加换行符。假设要编写一个循环，读取一行输入，另起一行打印出该输入。可以这样写：

```
char line[81];
while (gets(line)) // 与 while (gets(line) != NULL) 相同
 puts(line);
```

如果 `gets()` 读到文件结尾会返回空指针。对空指针求值为 0（即为假），这样便可结束循环。或者，可以这样写：

```
char line[81];
while (fgets(line, 81, stdin))
 fputs(line, stdout);
```

第一个循环（使用 `gets()` 和 `puts()` 的 `while` 循环），`line` 数组中的字符串显示在下一行，因为 `puts()` 在字符串末尾添加了一个换行符。第二个循环（使用 `fgets()` 和 `fputs()` 的 `while` 循环），`line` 数组中的字符串也显示在下一行，因为 `fgets()` 把换行符储存在字符串末尾。注意，如果混合使用 `fgets()` 输入和 `puts()` 输出，每个待显示的字符串末尾就会有两个换行符。这里关键要注意：`puts()` 应与 `gets()` 配对使用，`fputs()` 应与 `fgets()` 配对使用。

我们在这里提到已被废弃的 `gets()`，并不是鼓励使用它，而是为了让读者了解它的用法。如果今后遇到包含该函数的代码，不至于看不懂。

### 11.3.3 `printf()` 函数

在第 4 章中，我们详细讨论过 `printf()` 函数的用法。和 `puts()` 一样，`printf()` 也把字符串的地址作为参数。`printf()` 函数用起来没有 `puts()` 函数那么方便，但是它更加多才多艺，因为它可以格式化不同的数据类型。

与 `puts()` 不同的是，`printf()` 不会自动在每个字符串末尾加上一个换行符。因此，必须在参数中指明应该在哪里使用换行符。例如：

```
printf("%s\n", string);
```

和下面的语句效果相同：

```
puts(string);
```

如上所示, `printf()` 的形式更复杂些, 需要输入更多代码, 而且计算机执行的时间也更长 (但是你觉察不到)。然而, 使用 `printf()` 打印多个字符串更加简单。例如, 下面的语句把 Well、用户名和一个 `#define` 定义的字符串打印在一行:

```
printf("Well, %s, %s\n", name, MSG);
```

## 11.4 自定义输入/输出函数

不一定非要使用 C 库中的标准函数, 如果无法使用这些函数或者不想用它们, 完全可以在 `getchar()` 和 `putchar()` 的基础上自定义所需的函数。假设你需要一个类似 `puts()` 但是不会自动添加换行符的函数。程序清单 11.14 给出了一个这样的函数。

程序清单 11.14 `put1()` 函数

---

```
/* put1.c -- 打印字符串, 不添加\n */
#include <stdio.h>
void put1(const char * string)/* 不会改变字符串 */
{
 while (*string != '\0')
 putchar(*string++);
}
```

---

指向 `char` 的指针 `string` 最初指向传入参数的首元素。因为该函数不会改变传入的字符串, 所以形参使用了 `const` 限定符。打印了首元素的内容后, 指针递增 1, 指向下一个元素。`while` 循环重复这一过程, 直到指针指向包含空字符的元素。记住, `++` 的优先级高于 `*`, 因此 `putchar(*string++)` 打印 `string` 指向的值, 递增的是 `string` 本身, 而不是递增它所指向的字符。

可以把 `put1.c` 程序作为编写字符串处理函数的模型。因为每个字符串都以空字符结尾, 所以不用给函数传递字符串的大小。函数依次处理每个字符, 直至遇到空字符。

用数组表示法编写这个函数稍微复杂些:

```
int i = 0;
while (string[i] != '\0')
 putchar(string[i++]);
```

要为数组索引创建一个额外的变量。

许多 C 程序员会在 `while` 循环中使用下面的测试条件:

```
while (*string)
```

当 `string` 指向空字符时, `*string` 的值是 0, 即测试条件为假, `while` 循环结束。这种方法比上面两种方法简洁。但是, 如果不熟悉 C 语言, 可能觉察不出来。这种处理方法很普遍, 作为 C 程序员应该熟悉这种写法。

### 注意

为什么程序清单 11.14 中的形式参数是 `const char * string`, 而不是 `const char string[]`? 从技术方面看, 两者等价且都有效。使用带方括号的写法是为了提醒用户: 该函数处理的是数组。然而, 如果要处理字符串, 实际参数可以是数组名、用双引号括起来的字符串, 或声明为 `char *` 类型的变量。用 `const char * string` 可以提醒用户: 实际参数不一定是数组。

假设要设计一个类似 `puts()` 的函数，而且该函数还给出待打印字符的个数。如程序清单 11.15 所示，添加一个功能很简单。

#### 程序清单 11.15 put2.c 程序

---

```
/* put2.c -- 打印一个字符串，并统计打印的字符数 */
#include <stdio.h>
int put2(const char * string)
{
 int count = 0;
 while (*string) /* 常规用法 */
 {
 putchar(*string++);
 count++;
 }
 putchar('\n'); /* 不统计换行符 */

 return(count);
}
```

---

下面的函数调用将打印字符串 `pizza`:

```
put1("pizza");
```

下面的调用将返回统计的字符数，并将其赋给 `num`（该例中，`num` 的值是 5）：

```
num = put2("pizza");
```

程序清单 11.16 使用一个简单的驱动程序测试 `put1()` 和 `put2()`，并演示了嵌套函数的调用。

#### 程序清单 11.16 .c 程序

---

```
//put_put.c -- 用户自定义输出函数
#include <stdio.h>
void put1(const char *);
int put2(const char *);

int main(void)
{
 put1("If I'd as much money");
 put1(" as I could spend,\n");
 printf("I count %d characters.\n",
 put2("I never would cry old chairs to mend."));

 return 0;
}

void put1(const char * string)
{
 while (*string) /* 与 *string != '\0' 相同 */
 putchar(*string++);
}

int put2(const char * string)
{
 int count = 0;
 while (*string)
```

```

 putchar(*string++);
 count++;
 }
 putchar('\n');

 return(count);
}

```

程序中使用 `printf()` 打印 `put2()` 的值，但是为了获得 `put2()` 的返回值，计算机必须先执行 `put2()`，因此在打印字符数之前先打印了传递给该函数的字符串。下面是该程序的输出：

```

If I'd as much money as I could spend,
I never would cry old chairs to mend.
I count 37 characters.

```

## 11.5 字符串函数

C 库提供了多个处理字符串的函数，ANSI C 把这些函数的原型放在 `string.h` 头文件中。其中最常用的函数有 `strlen()`、`strcat()`、`strcmp()`、`strncmp()`、`strcpy()` 和 `strncpy()`。另外，还有 `sprintf()` 函数，其原型在 `stdio.h` 头文件中。欲了解 `string.h` 系列函数的完整列表，请查阅附录 B 中的参考资料 V “新增 C99 和 C11 的标准 ANSI C 库”。

### 11.5.1 `strlen()` 函数

`strlen()` 函数用于统计字符串的长度。下面的函数可以缩短字符串的长度，其中用到了 `strlen()`：

```

void fit(char *string, unsigned int size)
{
 if (strlen(string) > size)
 string[size] = '\0';
}

```

该函数要改变字符串，所以函数头在声明形式参数 `string` 时没有使用 `const` 限定符。

程序清单 11.17 中的程序测试了 `fit()` 函数。注意代码中使用了 C 字符串常量的串联特性。

程序清单 11.17 `test_fit.c` 程序

```

/* test_fit.c -- 使用缩短字符串长度的函数 */
#include <stdio.h>
#include <string.h> /* 内含字符串函数原型 */
void fit(char *, unsigned int);

int main(void)
{
 char mesg [] = "Things should be as simple as possible,"
 " but not simpler.";

 puts(mesg);
 fit(mesg, 38);
 puts(mesg);
 puts("Let's look at some more of the string.");
 puts(mesg + 39);

 return 0;
}

```

```

void fit(char *string, unsigned int size)
{
 if (strlen(string) > size)
 string[size] = '\0';
}

```

下面是该程序的输出：

```

Things should be as simple as possible, but not simpler.
Things should be as simple as possible
Let's look at some more of the string.
but not simpler.

```

`fit()` 函数把第 39 个元素的逗号替换成'\0'字符。`puts()` 函数在空字符处停止输出，并忽略其余字符。然而，这些字符还在缓冲区中，下面的函数调用把这些字符打印了出来：

```
puts(msg + 8);
```

表达式 `msg + 39` 是 `msg[39]` 的地址，该地址上储存的是空格字符。所以 `put()` 显示该字符并继续输出直至遇到原来字符串中的空字符。图 11.4 演示了这一过程。

原始字符串：



调用 `fit (msg,7)` 之后的字符串

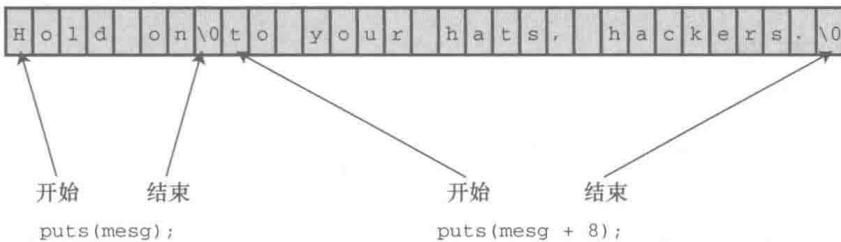


图 11.4 `puts()` 函数和空字符

## 注意

一些 ANSI 之前的系统使用 `strings.h` 头文件，而有些系统可能根本没有字符串头文件。

`string.h` 头文件中包含了 C 字符串函数系列的原型，因此程序清单 11.17 要包含该头文件。

## 11.5.2 `strcat()` 函数

`strcat()`（用于拼接字符串）函数接受两个字符串作为参数。该函数把第 2 个字符串的备份附加在第 1 个字符串末尾，并把拼接后形成的新字符串作为第 1 个字符串，第 2 个字符串不变。`strcat()` 函数的类型是 `char *`（即，指向 `char` 的指针）。`strcat()` 函数返回第 1 个参数，即拼接第 2 个字符串后的第 1 个字符串的地址。

程序清单 11.18 演示了 `strcat()` 的用法。该程序还使用了程序清单 11.10 的 `s_gets()` 函数。回忆一下，该函数使用 `fgets()` 读取一整行，如果有换行符，将其替换成空字符。

## 程序清单 11.18 str\_cat.c 程序

```

/* str_cat.c -- 拼接两个字符串 */
#include <stdio.h>
#include <string.h> /* strcat() 函数的原型在该头文件中 */
#define SIZE 80
char * s_gets(char * st, int n);
int main(void)
{
 char flower[SIZE];
 char addon [] = "s smell like old shoes.";

 puts("What is your favorite flower?");
 if (s_gets(flower, SIZE))
 {
 strcat(flower, addon);
 puts(flower);
 puts(addon);
 }
 else
 puts("End of file encountered!");
 puts("bye");

 return 0;
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

该程序的输出示例如下：

```

What is your favorite flower?
wonderflower
wonderflowers smell like old shoes.
s smell like old shoes.
bye

```

从以上输出可以看出，flower 改变了，而 addon 保持不变。

### 11.5.3 strncat() 函数

strcat() 函数无法检查第 1 个数组是否能容纳第 2 个字符串。如果分配给第 1 个数组的空间不够大，多出来的字符溢出到相邻存储单元时就会出问题。当然，可以像程序清单 11.15 那样，用 strlen() 查看第 1 个数组的长度。注意，要给拼接后的字符串长度加 1 才够空间存放末尾的空字符。或者，用 strncat()，该函数的第 3 个参数指定了最大添加字符数。例如，strncat(bugs, addon, 13) 将把 addon 字符串的内容附加给 bugs，在加到第 13 个字符或遇到空字符时停止。因此，算上空字符（无论哪种情况都要添加空字符），bugs 数组应该足够大，以容纳原始字符串（不包含空字符）、添加原始字符串在后面的 13 个字符和末尾的空字符。程序清单 11.19 使用这种方法，计算 available 变量的值，用于表示允许添加的最大字符数。

程序清单 11.19 join\_chk.c 程序

```
/* join_chk.c -- 拼接两个字符串，检查第 1 个数组的大小 */
#include <stdio.h>
#include <string.h>
#define SIZE 30
#define BUGSIZE 13
char * s_gets(char * st, int n);
int main(void)
{
 char flower[SIZE];
 char addon [] = "s smell like old shoes.";
 char bug[BUGSIZE];
 int available;

 puts("What is your favorite flower?");
 s_gets(flower, SIZE);
 if ((strlen(addon) + strlen(flower) + 1) <= SIZE)
 strcat(flower, addon);
 puts(flower);
 puts("What is your favorite bug?");
 s_gets(bug, BUGSIZE);
 available = BUGSIZE - strlen(bug) - 1;
 strncat(bug, addon, available);
 puts(bug);

 return 0;
}
char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
```

```

 continue;
 }
 return ret_val;
}

```

下面是该程序的运行示例：

```

What is your favorite flower?
Rose
Roses smell like old shoes.
What is your favorite bug?
Aphid
Aphids smell

```

读者可能已经注意到，`strcat()` 和 `gets()` 类似，也会导致缓冲区溢出。为什么 C11 标准不废弃 `strcat()`，只留下 `strncat()`？为何对 `gets()` 那么残忍？这也许是由于 `gets()` 造成的安全隐患来自于使用该程序的人，而 `strcat()` 暴露的问题是那些粗心的程序员造成的。无法控制用户会进行什么操作，但是，可以控制你的程序做什么。C 语言相信程序员，因此程序员有责任确保 `strcat()` 的使用安全。

#### 11.5.4 `strcmp()` 函数

假设要把用户的响应与已储存的字符串作比较，如程序清单 11.20 所示。

程序清单 11.20 nogo.c 程序

```

/* nogo.c -- 该程序是否能正常运行? */
#include <stdio.h>
#define ANSWER "Grant"
#define SIZE 40
char * s_gets(char * st, int n);

int main(void)
{
 char try[SIZE];

 puts("Who is buried in Grant's tomb?");
 s_gets(try, SIZE);
 while (try != ANSWER)
 {
 puts("No, that's wrong. Try again.");
 s_gets(try, SIZE);
 }
 puts("That's right!");

 return 0;
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 while (st[i] != '\n' && st[i] != '\0')

```

```

 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
}
return ret_val;
}

```

这个程序看上去没问题，但是运行后却不对劲。ANSWER 和 try 都是指针，所以 try != ANSWER 检查的不是两个字符串是否相等，而是这两个字符串的地址是否相同。因为 ANSWER 和 try 储存在不同的位置，所以这两个地址不可能相同，因此，无论用户输入什么，程序都提示输入不正确。这真让人沮丧。

该函数要比较的是字符串的内容，不是字符串的地址。读者可以自己设计一个函数，也可以使用 C 标准库中的 strcmp() 函数（用于字符串比较）。该函数通过比较运算符来比较字符串，就像比较数字一样。如果两个字符串参数相同，该函数就返回 0，否则返回非零值。修改后的版本如程序清单 11.21 所示。

程序清单 11.21 compare.c 程序

```

/* compare.c -- 该程序可以正常运行 */
#include <stdio.h>
#include <string.h> // strcmp() 函数的原型在该头文件中

#define ANSWER "Grant"
#define SIZE 40
char * s_gets(char * st, int n);

int main(void)
{
 char try[SIZE];

 puts("Who is buried in Grant's tomb?");
 s_gets(try, SIZE);
 while (strcmp(try, ANSWER) != 0)
 {
 puts("No, that's wrong. Try again.");
 s_gets(try, SIZE);
 }
 puts("That's right!");

 return 0;
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')

```

```

 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

## 注意

由于非零值都为“真”，所以许多经验丰富的 C 程序员会把该例 main() 中的 while 循环头写成：

```
while (strcmp(try, ANSWER))
```

strcmp() 函数比较的是字符串，不是整个数组，这是非常好的功能。虽然数组 try 占用了 40 字节，而储存在其中的“Grant”只占用了 6 字节（还有一个用来放空字符），strcmp() 函数只会比较 try 中第 1 个空字符前面的部分。所以，可以用 strcmp() 比较储存在不同大小数组中的字符串。

如果用户输入 GRANT、grant 或 Ulysses S. Grant 会怎样？程序会告知用户输入错误。希望程序更友好，必须把所有正确答案的可能性包含其中。这里可以使用一些小技巧。例如，可以使用#define 定义类似 GRANT 这样的答案，并编写一个函数把输入的内容都转换成小写，就解决了大小写的问题。但是，还要考虑一些其他错误的形式，这些留给读者完成。

### 1. strcmp() 的返回值

如果 strcmp() 比较的字符串不同，它会返回什么值？请看程序清单 11.22 的程序示例。

**程序清单 11.22 compback.c 程序**

```

/* compback.c -- strcmp() 的返回值 */
#include <stdio.h>
#include <string.h>
int main(void)
{
 printf("strcmp(\"A\", \"A\") is ");
 printf("%d\n", strcmp("A", "A"));

 printf("strcmp(\"A\", \"B\") is ");
 printf("%d\n", strcmp("A", "B"));

 printf("strcmp(\"B\", \"A\") is ");
 printf("%d\n", strcmp("B", "A"));

 printf("strcmp(\"C\", \"A\") is ");
 printf("%d\n", strcmp("C", "A"));

 printf("strcmp(\"Z\", \"a\") is ");
 printf("%d\n", strcmp("Z", "a"));

 printf("strcmp(\"apples\", \"apple\") is ");
 printf("%d\n", strcmp("apples", "apple"));

 return 0;
}

```

在我们的系统中运行该程序，输出如下：

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 1
strcmp("Z", "a") is -1
strcmp("apples", "apple") is 1
```

`strcmp()` 比较“`A`”和本身，返回 0；比较“`A`”和“`B`”，返回 -1；比较“`B`”和“`A`”，返回 1。这说明，如果在字母表中第 1 个字符串位于第 2 个字符串前面，`strcmp()` 中就返回负数；反之，`strcmp()` 则返回正数。所以，`strcmp()` 比较“`C`”和“`A`”，返回 1。其他系统可能返回 2，即两者的 ASCII 码之差。ASCII 标准规定，在字母表中，如果第 1 个字符串在第 2 个字符串前面，`strcmp()` 返回一个负数；如果两个字符串相同，`strcmp()` 返回 0；如果第 1 个字符串在第 2 个字符串后面，`strcmp()` 返回正数。然而，返回的具体值取决于实现。例如，下面给出在不同实现中的输出，该实现返回两个字符的差值：

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 2
strcmp("Z", "a") is -7
strcmp("apples", "apple") is 115
```

如果两个字符串开始的几个字符都相同会怎样？一般而言，`strcmp()` 会依次比较每个字符，直到发现第 1 对不同的字符为止。然后，返回相应的值。例如，在上面的最后一个例子中，“`apples`”和“`apple`”只有最后一对字符不同（“`apples`”的 s 和“`apple`”的空字符）。由于空字符在 ASCII 中排第 1。字符 s 一定在它后面，所以 `strcmp()` 返回一个正数。

最后一个例子表明，`strcmp()` 比较所有的字符，不只是字母。所以，与其说该函数按字母顺序进行比较，不如说是按机器排序序列（*machine collating sequence*）进行比较，即根据字符的数值进行比较（通常都使用 ASCII 值）。在 ASCII 中，大写字母在小写字母前面，所以 `strcmp("Z", "a")` 返回的是负值。

大多数情况下，`strcmp()` 返回的具体值并不重要，我们只在意该值是 0 还是非 0（即，比较的两个字符串是否相等）。或者按字母排序字符串，在这种情况下，需要知道比较的结果是为正、为负还是为 0。

## 注意

`strcmp()` 函数比较的是字符串，不是字符，所以其参数应该是字符串（如“`apples`”和“`A`”），而不是字符（如‘`A`’）。但是，`char` 类型实际上是整数类型，所以可以使用关系运算符来比较字符。假设 `word` 是储存在 `char` 类型数组中的字符串，`ch` 是 `char` 类型的变量，下面的语句都有效：

```
if (strcmp(word, "quit") == 0) // 使用 strcmp() 比较字符串
 puts("Bye!");
if (ch == 'q') // 使用 == 比较字符
 puts("Bye!");
```

尽管如此，不要使用 `ch` 或‘`q`’作为 `strcmp()` 的参数。

程序清单 11.23 用 `strcmp()` 函数检查程序是否要停止读取输入。

### 程序清单 11.23 `quit_chk.c` 程序

---

```
/* quit_chk.c -- 某程序的开始部分 */
#include <stdio.h>
#include <string.h>
```

```

#define SIZE 80
#define LIM 10
#define STOP "quit"
char * s_gets(char * st, int n);

int main(void)
{
 char input[LIM][SIZE];
 int ct = 0;

 printf("Enter up to %d lines (type quit to quit):\n", LIM);
 while (ct < LIM && s_gets(input[ct], SIZE) != NULL &&
 strcmp(input[ct], STOP) != 0)
 {
 ct++;
 }
 printf("%d strings entered\n", ct);

 return 0;
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

该程序在读到 EOF 字符（这种情况下 `s_gets()` 返回 `NULL`）、用户输入 `quit` 或输入项达到 `LIM` 时退出。

顺带一提，有时输入空行（即，只按下 **Enter** 键或 **Return** 键）表示结束输入更方便。为实现这一功能，只需修改一下 `while` 循环的条件即可：

```
while (ct < LIM && s_gets(input[ct], SIZE) != NULL && input[ct][0] != '\0')
```

这里，`input[ct]` 是刚输入的字符串，`input[ct][0]` 是该字符串的第一个字符。如果用户输入空行，`s_gets()` 便会把该行第一个字符（换行符）替换成空字符。所以，下面的表达式用于检测空行：

```
input[ct][0] != '\0'
```

## 2. `strcmp()` 函数

`strcmp()` 函数比较字符串中的字符，直到发现不同的字符为止，这一过程可能会持续到字符串的末尾。而 `strncmp()` 函数在比较两个字符串时，可以比较到字符不同的地方，也可以只比较第 3 个参数指定的字

符数。例如，要查找以"astro"开头的字符串，可以限定函数只查找这 5 个字符。程序清单 11.24 演示了该函数的用法。

程序清单 11.24 starsrch.c 程序

---

```
/* starsrch.c -- 使用 strncmp() */
#include <stdio.h>
#include <string.h>
#define LISTSIZE 6
int main()
{
 const char * list[LISTSIZE] =
 {
 "astronomy", "astounding",
 "astrophysics", "ostracize",
 "asterism", "astrophobia"
 };
 int count = 0;
 int i;

 for (i = 0; i < LISTSIZE; i++)
 if (strncmp(list[i], "astro", 5) == 0)
 {
 printf("Found: %s\n", list[i]);
 count++;
 }
 printf("The list contained %d words beginning"
 " with astro.\n", count);

 return 0;
}
```

---

下面是该程序的输出：

```
Found: astronomy
Found: astrophysics
Found: astrophobia
The list contained 3 words beginning with astro.
```

## 11.5.5 strcpy() 和 strncpy() 函数

前面提到过，如果 pts1 和 pts2 都是指向字符串的指针，那么下面语句拷贝的是字符串的地址而不是字符串本身：

```
pts2 = pts1;
```

如果希望拷贝整个字符串，要使用 strcpy() 函数。程序清单 11.25 要求用户输入以 q 开头的单词。该程序把输入拷贝至一个临时数组中，如果第 1 个字母是 q，程序调用 strcpy() 把整个字符串从临时数组拷贝至目标数组中。strcpy() 函数相当于字符串赋值运算符。

程序清单 11.25 copy1.c 程序

---

```
/* copy1.c -- 演示 strcpy() */
#include <stdio.h>
#include <string.h> // strcpy() 的原型在该头文件中
#define SIZE 40
#define LIM 5
```

```

char * s_gets(char * st, int n);

int main(void)
{
 char qwords[LIM][SIZE];
 char temp[SIZE];
 int i = 0;

 printf("Enter %d words beginning with q:\n", LIM);
 while (i < LIM && s_gets(temp, SIZE))
 {
 if (temp[0] != 'q')
 printf("%s doesn't begin with q!\n", temp);
 else
 {
 strcpy(qwords[i], temp);
 i++;
 }
 }
 puts("Here are the words accepted:");
 for (i = 0; i < LIM; i++)
 puts(qwords[i]);

 return 0;
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

下面是该程序的运行示例：

```

Enter 5 words beginning with q:
quackery
quasar
quilt
quotient
no more
no more doesn't begin with q!
quiz
Here are the words accepted:

```

```
quackery
quasar
quilt
quotient
quiz
```

注意，只有在输入以 q 开头的单词后才会递增计数器 i，而且该程序通过比较字符进行判断：

```
if (temp[0] != 'q')
```

这行代码的意思是：temp 中的第 1 个字符是否是 q？当然，也可以通过比较字符串进行判断：

```
if (strcmp(temp, "q", 1) != 0)
```

这行代码的意思是：temp 字符串和"q"的第 1 个元素是否相等？

请注意，`strcpy()` 第 2 个参数（`temp`）指向的字符串被拷贝至第 1 个参数（`qword[i]`）指向的数据组中。拷贝出来的字符串被称为目标字符串，最初的字符串被称为源字符串。参考赋值表达式语句，很容易记住 `strcpy()` 参数的顺序，即第 1 个是目标字符串，第 2 个是源字符串。

```
char target[20];
int x;
x = 50; /* 数字赋值 */
strcpy(target, "Hi ho!"); /* 字符串赋值 */
target = "So long"; /* 语法错误 */
```

程序员有责任确保目标数组有足够的空间容纳源字符串的副本。下面的代码有点问题：

```
char * str;
strcpy(str, "The C of Tranquility"); // 有问题
```

`strcpy()` 把"The C of Tranquility"拷贝至 str 指向的地址上，但是 str 未被初始化，所以该字符串可能被拷贝到任意的地方！

总之，`strcpy()` 接受两个字符串指针作为参数，可以把指向源字符串的第 2 个指针声明为指针、数组名或字符串常量；而指向源字符串副本的第 1 个指针应指向一个数据对象（如，数组），且该对象有足够的空间储存源字符串的副本。记住，声明数组将分配储存数据的空间，而声明指针只分配储存一个地址的空间。

## 1. `strcpy()` 的其他属性

`strcpy()` 函数还有两个有用的属性。第一，`strcpy()` 的返回类型是 `char *`，该函数返回的是第 1 个参数的值，即一个字符的地址。第二，第 1 个参数不必指向数组的开始。这个属性可用于拷贝数组的一部分。程序清单 11.26 演示了该函数的这两个属性。

程序清单 11.26 copy2.c 程序

```
/* copy2.c -- 使用 strcpy() */
#include <stdio.h>
#include <string.h> // 提供 strcpy() 的函数原型
#define WORDS "beast"
#define SIZE 40

int main(void)
{
 const char * orig = WORDS;
 char copy[SIZE] = "Be the best that you can be.";
 char * ps;

 puts(orig);
```

```

 puts(copy);
 ps = strcpy(copy + 7, orig);
 puts(copy);
 puts(ps);

 return 0;
}

```

下面是该程序的输出：

```

beast
Be the best that you can be.
Be the beast
beast

```

注意，`strcpy()`把源字符串中的空字符也拷贝在内。在该例中，空字符覆盖了 `copy` 数组中 `that` 的第 1 个 `t`（见图 11.5）。注意，由于第 1 个参数是 `copy + 7`，所以 `ps` 指向 `copy` 中的第 8 个元素（下标为 7）。因此 `puts(ps)` 从该处开始打印字符串。

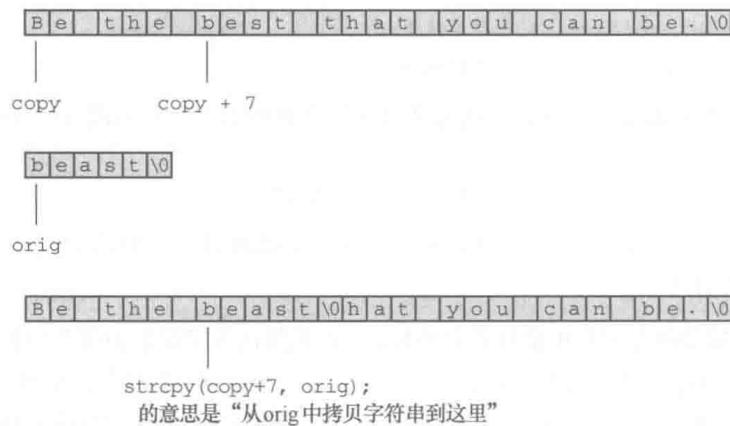


图 11.5 使用指针 `strcpy()` 函数

## 2. 更谨慎的选择：`strncpy()`

`strcpy()` 和 `strcat()` 都有同样的问题，它们都不能检查目标空间是否能容纳源字符串的副本。拷贝字符串用 `strncpy()` 更安全，该函数的第 3 个参数指明可拷贝的最大字符数。程序清单 11.27 用 `strncpy()` 替代程序清单 11.25 中的 `strcpy()`。为了演示目标空间装不下源字符串的副本会发生什么情况，该程序使用了一个相当小的目标字符串（共 7 个元素，包含 6 个字符）。

### 程序清单 11.27 copy3.c 程序

```

/* copy3.c -- 使用 strncpy() */
#include <stdio.h>
#include <string.h> /* 提供 strncpy() 的函数原型 */
#define SIZE 40
#define TARGSIZE 7
#define LIM 5
char * s_gets(char * st, int n);

int main(void)
{
 char qwords[LIM][TARGSIZE];
 char temp[SIZE];

```

```

int i = 0;

printf("Enter %d words beginning with q:\n", LIM);
while (i < LIM && s_gets(temp, SIZE))
{
 if (temp[0] != 'q')
 printf("%s doesn't begin with q!\n", temp);
 else
 {
 strncpy(qwords[i], temp, TARGSIZE - 1);
 qwords[i][TARGSIZE - 1] = '\0';
 i++;
 }
}
puts("Here are the words accepted:");
for (i = 0; i < LIM; i++)
 puts(qwords[i]);

return 0;
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

下面是该程序的运行示例:

Enter 5 words beginning with q:

quack  
quadratic  
quisling  
quota  
quagga

Here are the words accepted:

quack  
quadra  
quisli  
quota  
quagga

`strncpy(target, source, n)` 把 `source` 中的 `n` 个字符或空字符之前的字符（先满足哪个条件就

拷贝到何处) 拷贝至 target 中。因此, 如果 source 中的字符数小于 n, 则拷贝整个字符串, 包括空字符。但是, `strncpy()` 拷贝字符串的长度不会超过 n, 如果拷贝到第 n 个字符时还未拷贝完整个源字符串, 就不会拷贝空字符。所以, 拷贝的副本中不一定有空字符。鉴于此, 该程序把 n 设置为比目标数组大小少 1 (`TARGSIZE-1`), 然后把数组最后一个元素设置为空字符:

```
strncpy(qwords[i], temp, TARGSIZE - 1);
qwords[i][TARGSIZE - 1] = '\0';
```

这样做确保储存的是一个字符串。如果目标空间能容纳源字符串的副本, 那么从源字符串拷贝的空字符便是该副本的结尾; 如果目标空间装不下副本, 则把副本最后一个元素设置为空字符。

## 11.5.6 `sprintf()` 函数

`sprintf()` 函数声明在 `stdio.h` 中, 而不是在 `string.h` 中。该函数和 `printf()` 类似, 但是它是把数据写入字符串, 而不是打印在显示器上。因此, 该函数可以把多个元素组合成一个字符串。`sprintf()` 的第 1 个参数是目标字符串的地址。其余参数和 `printf()` 相同, 即格式字符串和待写入项的列表。

程序清单 11.28 中的程序用 `printf()` 把 3 个项 (两个字符串和一个数字) 组合成一个字符串。注意, `sprintf()` 的用法和 `printf()` 相同, 只不过 `sprintf()` 把组合后的字符串储存在数组 `formal` 中而不是显示在屏幕上。

程序清单 11.28 `format.c` 程序

```
/* format.c -- 格式化字符串 */
#include <stdio.h>
#define MAX 20
char * s_gets(char * st, int n);

int main(void)
{
 char first[MAX];
 char last[MAX];
 char formal[2 * MAX + 10];
 double prize;

 puts("Enter your first name:");
 s_gets(first, MAX);
 puts("Enter your last name:");
 s_gets(last, MAX);
 puts("Enter your prize money:");
 scanf("%lf", &prize);
 sprintf(formal, "%s, %-19s: $%6.2f\n", last, first, prize);
 puts(formal);

 return 0;
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
```

```

 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

下面是该程序的运行示例：

```

Enter your first name:
Annie
Enter your last name:
von Wurstkasse
Enter your prize money:
25000
von Wurstkasse, Annie : $25000.00

```

`sprintf()` 函数获取输入，并将其格式化为标准形式，然后把格式化后的字符串储存在 `formal` 中。

## 11.5.7 其他字符串函数

ANSI C 库有 20 多个用于处理字符串的函数，下面总结了一些常用的函数。

■ `char *strcpy(char * restrict s1, const char * restrict s2);`

该函数把 `s2` 指向的字符串（包括空字符）拷贝至 `s1` 指向的位置，返回值是 `s1`。

■ `char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`

该函数把 `s2` 指向的字符串拷贝至 `s1` 指向的位置，拷贝的字符数不超过 `n`，其返回值是 `s1`。该函数不会拷贝空字符后面的字符，如果源字符串的字符少于 `n` 个，目标字符串就以拷贝的空字符结尾；如果源字符串有 `n` 个或超过 `n` 个字符，就不拷贝空字符。

■ `char *strcat(char * restrict s1, const char * restrict s2);`

该函数把 `s2` 指向的字符串拷贝至 `s1` 指向的字符串末尾。`s2` 字符串的第一个字符将覆盖 `s1` 字符串末尾的空字符。该函数返回 `s1`。

■ `char *strncat(char * restrict s1, const char * restrict s2, size_t n);`

该函数把 `s2` 字符串中的 `n` 个字符拷贝至 `s1` 字符串末尾。`s2` 字符串的第一个字符将覆盖 `s1` 字符串末尾的空字符。不会拷贝 `s2` 字符串中空字符和其后的字符，并在拷贝字符的末尾添加一个空字符。该函数返回 `s1`。

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

如果 `s1` 字符串在机器排序序列中位于 `s2` 字符串的后面，该函数返回一个正数；如果两个字符串相等，则返回 0；如果 `s1` 字符串在机器排序序列中位于 `s2` 字符串的前面，则返回一个负数。

■ `int strncmp(const char * s1, const char * s2, size_t n);`

该函数的作用和 `strcmp()` 类似，不同的是，该函数在比较 `n` 个字符后或遇到第 1 个空字符时停止比较。

■ `char *strchr(const char * s, int c);`

如果 s 字符串中包含 c 字符，该函数返回指向 s 字符串首位置的指针（末尾的空字符也是字符串的一部分，所以在查找范围内）；如果在字符串 s 中未找到 c 字符，该函数则返回空指针。

■ `char *strpbrk(const char * s1, const char * s2);`

如果 s1 字符串中包含 s2 字符串中的任意字符，该函数返回指向 s1 字符串首位置的指针；如果在 s1 字符串中未找到任何 s2 字符串中的字符，则返回空字符串。

■ `char *strrchr(const char * s, int c);`

该函数返回 s 字符串中 c 字符的最后一次出现的位置（末尾的空字符也是字符串的一部分，所以在查找范围内）。如果未找到 c 字符，则返回空指针。

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

该函数返回指向 s1 字符串中 s2 字符串出现的首位置。如果在 s1 中没有找到 s2，则返回空指针。

■ `size_t strlen(const char * s);`

该函数返回 s 字符串中的字符数，不包括末尾的空字符。

请注意，那些使用 `const` 关键字的函数原型表明，函数不会更改字符串。例如，下面的函数原型：

```
char *strcpy(char * restrict s1, const char * restrict s2);
```

表明不能更改 s2 指向的字符串，至少不能在 `strcpy()` 函数中更改。但是可以更改 s1 指向的字符串。这样做很合理，因为 s1 是目标字符串，要改变，而 s2 是源字符串，不能更改。

关键字 `restrict` 将在第 12 章中介绍，该关键字限制了函数参数的用法。例如，不能把字符串拷贝给本身。

第 5 章中讨论过，`size_t` 类型是 `sizeof` 运算符返回的类型。C 规定 `sizeof` 运算符返回一个整数类型，但是并未指定是哪种整数类型，所以 `size_t` 在一个系统中可以是 `unsigned int`，而在另一个系统中可以是 `unsigned long`。`string.h` 头文件针对特定系统定义了 `size_t`，或者参考其他有 `size_t` 定义的头文件。

前面提到过，参考资料 V 中列出了 `string.h` 系列的所有函数。除提供 ANSI 标准要求的函数外，许多实现还提供一些其他函数。应查看你所使用的 C 实现文档，了解可以使用哪些函数。

我们来看一下其中一个函数的简单用法。前面学过的 `fgets()` 读入一行输入时，在目标字符串的末尾添加换行符。我们自定义的 `s_gets()` 函数通过 `while` 循环检测换行符。其实，这里可以用 `strchr()` 代替 `s_gets()`。首先，使用 `strchr()` 查找换行符（如果有的话）。如果该函数发现了换行符，将返回该换行符的地址，然后便可用空字符替换该位置上的换行符：

```
char line[80];
char * find;

fgets(line, 80, stdin);
find = strchr(line, '\n'); // 查找换行符
if (find) // 如果没找到换行符，返回 NULL
 *find = '\0'; // 把该处的字符替换为空字符
```

如果 `strchr()` 未找到换行符，`fgets()` 在达到行末尾之前就达到了它能读取的最大字符数。可以像在 `s_gets()` 中那样，给 `if` 添加一个 `else` 来处理这种情况。

接下来，我们看一个处理字符串的完整程序。

## 11.6 字符串示例：字符串排序

我们来处理一个按字母表顺序排序字符串的实际问题。准备名单表、创建索引和许多其他情况下都会用到字符串排序。该程序主要是用 `strcmp()` 函数来确定两个字符串的顺序。一般的做法是读取字符串函数、排序字符串并打印出来。之前，我们设计了一个读取字符串的方案，该程序就用到这个方案。打印字符串没问题。程序使用标准的排序算法，稍后解释。我们使用了一个小技巧，看看读者是否能明白。程序清单 11.29 演示了这个程序。

程序清单 11.29 `sort_str.c` 程序

```
/* sort_str.c -- 读入字符串，并排序字符串 */
#include <stdio.h>
#include <string.h>
#define SIZE 81 /* 限制字符串长度，包括 \0 */
#define LIM 20 /* 可读入的最多行数 */
#define HALT "" /* 空字符串停止输入 */
void stsrt(char *strings [], int num); /* 字符串排序函数 */
char * s_gets(char * st, int n);

int main(void)
{
 char input[LIM] [SIZE]; /* 储存输入的数组 */
 char *ptstr[LIM]; /* 内含指针变量的数组 */
 int ct = 0; /* 输入计数 */
 int k; /* 输出计数 */

 printf("Input up to %d lines, and I will sort them.\n", LIM);
 printf("To stop, press the Enter key at a line's start.\n");
 while (ct < LIM && s_gets(input[ct], SIZE) != NULL
 && input[ct][0] != '\0')
 {
 ptstr[ct] = input[ct]; /* 设置指针指向字符串 */
 ct++;
 }
 stsrt(ptstr, ct); /* 字符串排序函数 */
 puts("\nHere's the sorted list:\n");
 for (k = 0; k < ct; k++)
 puts(ptstr[k]); /* 排序后的指针 */

 return 0;
}

/* 字符串-指针-排序函数 */
void stsrt(char *strings [], int num)
{
 char *temp;
 int top, seek;

 for (top = 0; top < num - 1; top++)
 for (seek = top + 1; seek < num; seek++)
 if (strcmp(strings[top], strings[seek]) > 0)
```

```

 {
 temp = strings[top];
 strings[top] = strings[seek];
 strings[seek] = temp;
 }
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

我们用一首童谣来测试该程序：

```

Input up to 20 lines, and I will sort them.
To stop, press the Enter key at a line's start.
O that I was where I would be,
Then would I be where I am not;
But where I am I must be,
And where I would be I can not.

```

Here's the sorted list:

```

And where I would be I can not.
But where I am I must be,
O that I was where I would be,
Then would I be where I am not;

```

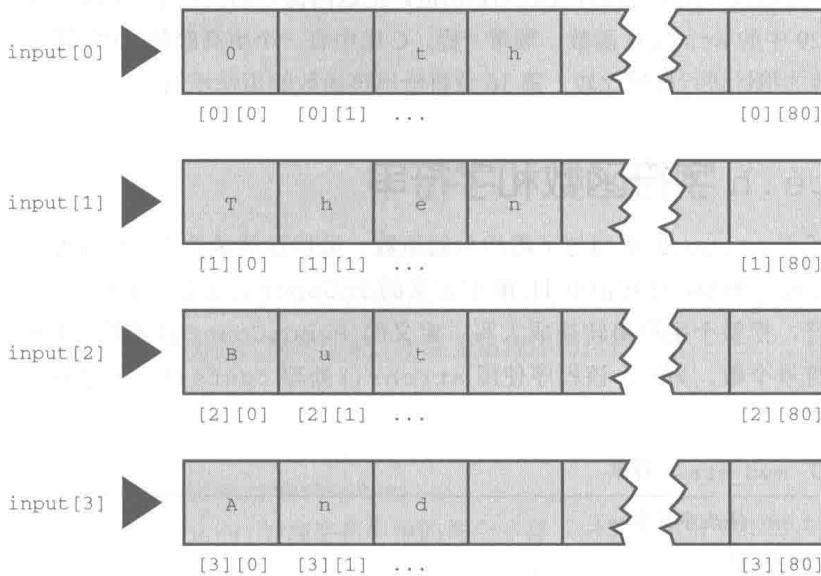
看来经过排序后，这首童谣的内容未受影响。

### 11.6.1 排序指针而非字符串

该程序的巧妙之处在于排序的是指向字符串的指针，而不是字符串本身。我们来分析一下具体怎么做。最初，ptrst[0]被设置为input[0]，ptrst[1]被设置为input[1]，以此类推。这意味着指针ptrst[i]指向数组input[i]的首字符。每个input[i]都是一个内含81个元素的数组，每个ptrst[i]都是一个单独的变量。排序过程把ptrst重新排列，并未改变input。例如，如果按字母顺序input[1]在input[0]前面，程序便交换指向它们的指针（即ptrst[0]指向input[1]的开始，而ptrst[1]指向input[0]的开始）。这样做比用strcpy()交换两个input字符串的内容简单得多，而且还保留了input数组中的原始顺序。图11.6从另一个视角演示了这一过程。

排序前：

ptrst[0] 指向input[0]  
ptrst[1] 指向input[1]  
等等



排序后：

ptrst[0] 指向input[3]  
ptrst[1] 指向input[2]  
等等

图 11.6 排序字符串指针

## 11.6.2 选择排序算法

我们采用选择排序算法 (*selection sort algorithm*) 来排序指针。具体做法是，利用 `for` 循环依次把每个元素与首元素比较。如果待比较的元素在当前首元素的前面，则交换两者。循环结束时，首元素包含的指针指向机器排序序列最靠前的字符串。然后外层 `for` 循环重复这一过程，这次从 `input` 的第 2 个元素开始。当内层循环执行完毕时，`ptrst` 中的第 2 个元素指向排在第 2 的字符串。这一过程持续到所有元素都已排序完毕。

现在来进一步分析选择排序的过程。下面是排序过程的伪代码：

```
for n = 首元素至 n = 倒数第 2 个元素,
 找出剩余元素中的最大值，并将其放在第 n 个元素中
```

具体过程如下。首先，从  $n = 0$  开始，遍历整个数组找出最大值元素，那该元素与第 1 个元素交换；然后设置  $n = 1$ ，遍历除第 1 个元素以外的其他元素，在其余元素中找出最大值元素，把该元素与第 2 个元素交换；重复这一过程直至倒数第 2 个元素为止。现在只剩下两个元素。比较这两个元素，把较大者放在倒数第 2 的位置上。这样，数组中的最小元素就在最后的位置上。

这看起来用 `for` 循环就能完成任务，但是我们还要更详细地分析“查找和放置”的过程。在剩余项中查找最大值的方法是，比较数组剩余元素的第 1 个元素和第 2 个元素。如果第 2 个元素比第 1 个元素大，交换两者。现在比较数组剩余元素的第 1 个元素和第 3 个元素，如果第 3 个元素比较大，交换两者。每次交换都把较大的元素移至顶部。继续这一过程直到比较第 1 个元素和最后一个元素。比较完毕后，最大值元素现在是剩余数组的首元素。已经排除了该数组的首元素，但是其他元素还是一团糟。下面是排序过程的伪代码：

for n - 第 2 个元素至最后一个元素，

比较第 n 个元素与第 1 个元素，如果第 n 个元素更大，交换这两个元素的值

看上去用一个 for 循环也能搞定。只不过要把它嵌套在刚才的 for 循环中。外层循环指明正在处理数组的哪一个元素，内层循环找出应储存在该元素的值。把这两部分伪代码结合起来，翻译成 C 代码，就得到了程序清单 11.29 中的 stsrt() 函数。顺带一提，C 库中有一个更高级的排序函数：qsort()。该函数使用一个指向函数的指针进行排序比较。第 16 章将给出该函数的用法示例。

## 11.7 ctype.h 字符函数和字符串

第 7 章中介绍了 ctype.h 系列与字符相关的函数。虽然这些函数不能处理整个字符串，但是可以处理字符串中的字符。例如，程序清单 11.30 中定义的 ToUpper() 函数，利用 toupper() 函数处理字符串中的每个字符，把整个字符串转换成大写；定义的 PunctCount() 函数，利用 ispunct() 统计字符串中的标点符号个数。另外，该程序使用 strchr() 处理 fgets() 读入字符串的换行符（如果有的话）。

程序清单 11.30 mod\_str.c 程序

```
/* mod_str.c -- 修改字符串 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LIMIT 81
void ToUpper(char *);
int PunctCount(const char *);

int main(void)
{
 char line[LIMIT];
 char * find;

 puts("Please enter a line:");
 fgets(line, LIMIT, stdin);
 find = strchr(line, '\n'); // 查找换行符
 if (find) // 如果地址不是 NULL,
 *find = '\0'; // 用空字符替换
 ToUpper(line);
 puts(line);
 printf("That line has %d punctuation characters.\n", PunctCount(line));

 return 0;
}

void ToUpper(char * str)
{
 while (*str)
 {
 *str = toupper(*str);
 str++;
 }
}
```

```

int PunctCount(const char * str)
{
 int ct = 0;
 while (*str)
 {
 if (ispunct(*str))
 ct++;
 str++;
 }

 return ct;
}

```

`while (*str)` 循环处理 `str` 指向的字符串中的每个字符，直至遇到空字符。此时`*str` 的值为 0（空字符的编码值为 0），即循环条件为假，循环结束。下面是该程序的运行示例：

```

Please enter a line:
Me? You talkin' to me? Get outta here!
ME? YOU TALKIN' TO ME? GET OUTTA HERE!
That line has 4 punctuation characters.

```

`ToUpper()` 函数利用 `toupper()` 处理字符串中的每个字符（由于 C 区分大小写，所以这是两个不同的函数名）。根据 ANSI C 中的定义，`toupper()` 函数只改变小写字符。但是一些很旧的 C 实现不会自动检查大小写，所以以前的代码通常会这样写：

```

if (islower(*str)) /* ANSI C 之前的做法 -- 在转换大小写之前先检查 */
 *str = toupper(*str);

```

顺带一提，`ctype.h` 中的函数通常作为宏（*macro*）来实现。这些 C 预处理器宏的作用很像函数，但是两者有一些重要的区别。我们在第 16 章再讨论关于宏的内容。

该程序使用 `fgets()` 和 `strchr()` 组合，读取一行输入并把换行符替换成空字符。这种方法与使用 `s_gets()` 的区别是：`s_gets()` 会处理输入行剩余字符（如果有的话），为下一次输入做好准备。而本例只有一条输入语句，就没必要进行多余的步骤。

## 11.8 命令行参数

在图形界面普及之前都使用命令行界面。DOS 和 UNIX 就是例子。Linux 终端提供类 UNIX 命令行环境。命令行（*command line*）是在命令行环境中，用户为运行程序输入命令的行。假设一个文件中有一个名为 `fuss` 的程序。在 UNIX 环境中运行该程序的命令行是：

```
$ fuss
```

或者在 Windows 命令提示模式下是：

```
C> fuss
```

命令行参数（*command-line argument*）是同一行的附加项。如下例：

```
$ fuss -r Ginger
```

一个 C 程序可以读取并使用这些附加项（见图 11.7）。

程序清单 11.27 是一个典型的例子，该程序通过 `main()` 的参数读取这些附加项。

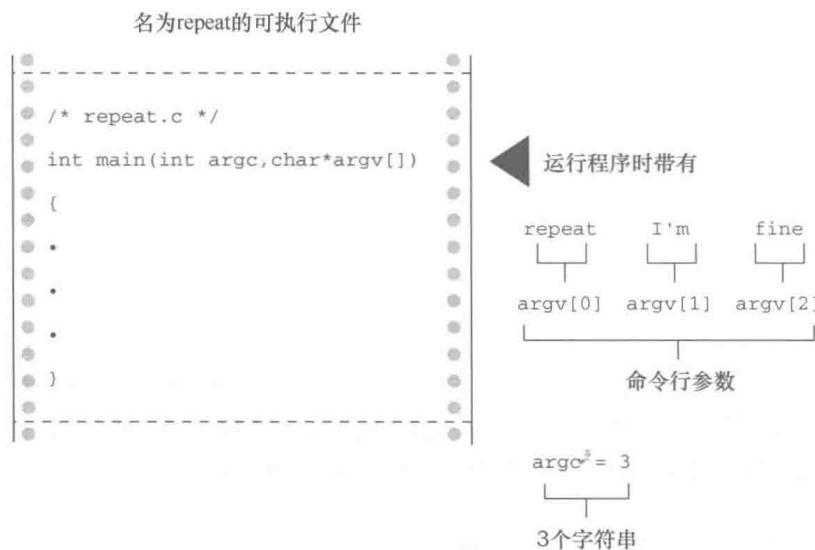


图 11.7 命令行参数

## 程序清单 11.31 repeat.c 程序

```
/* repeat.c -- 带参数的 main() */
#include <stdio.h>
int main(int argc, char *argv[])
{
 int count;

 printf("The command line has %d arguments:\n", argc - 1);
 for (count = 1; count < argc; count++)
 printf("%d: %s\n", count, argv[count]);
 printf("\n");

 return 0;
}
```

把该程序编译为可执行文件 repeat。下面是通过命令行运行该程序后的输出：

```
C>repeat Resistance is futile
The command line has 3 arguments:
1: Resistance
2: is
3: futile
```

由此可见该程序为何名为 repeat。下面我们解释一下它的运行原理。

C 编译器允许 main() 没有参数或者有两个参数（一些实现允许 main() 有更多参数，属于对标准的扩展）。main() 有两个参数时，第 1 个参数是命令行中的字符串数量。过去，这个 int 类型的参数被称为 argc（表示参数计数 *(argument count)*）。系统用空格表示一个字符串的结束和下一个字符串的开始。因此，上面的 repeat 示例中包括命令名共有 4 个字符串，其中后 3 个供 repeat 使用。该程序把命令行字符串储存在内存中，并把每个字符串的地址储存在指针数组中。而该数组的地址则被储存在 main() 的第 2 个参数中。按照惯例，这个指向指针的指针称为 argv（表示参数值 *[argument value]*）。如果系统允许（一些操作系统不允许这样），就把程序本身的名称赋给 argv[0]，然后把随后的第 1 个字符串赋给 argv[1]，以此类推。在我们的例子中，有下面的关系：

`argv[0]` 指向 `repeat` (对大部分系统而言)

`argv[1]` 指向 `Resistance`

`argv[2]` 指向 `is`

`argv[3]` 指向 `futile`

程序清单 11.31 的程序通过一个 `for` 循环依次打印每个字符串。`printf()` 中的`%s` 转换说明表明，要提供一个字符串的地址作为参数，而指针数组中的每个元素 (`argv[0]`、`argv[1]` 等) 都是这样的地址。

`main()` 中的形参形式与其他带形参的函数相同。许多程序员用不同的形式声明 `argv`:

```
int main(int argc, char **argv)
```

`char **argv` 与 `char *argv[]` 等价。也就是说，`argv` 是一个指向指针的指针，它所指向的指针指向 `char`。因此，即使在原始定义中，`argv` 也是指向指针（该指针指向 `char`）的指针。两种形式都可以使用，但我们认为第 1 种形式更清楚地表明 `argv` 表示一系列字符串。

顺带一提，许多环境（包括 UNIX 和 DOS）都允许用双引号把多个单词括起来形成一个参数。例如：

```
repeat "I am hungry" now
```

这行命令把字符串 "I am hungry" 赋给 `argv[1]`，把 "now" 赋给 `argv[2]`。

## 11.8.1 集成环境中的命令行参数

Windows 集成环境（如 Xcode、Microsoft Visual C++ 和 Embarcadero C++ Builder）都不用命令行运行程序。有些环境中项目对话框，为特定项目指定命令行参数。其他环境中，可以在 IDE 中编译程序，然后打开 MS-DOS 窗口在命令行模式中运行程序。但是，如果你的系统有一个运行命令行的编译器（如 GCC）会更简单。

## 11.8.2 Macintosh 中的命令行参数

如果使用 Xcode 4.6（或类似的版本），可以在 **Product** 菜单中选择 **Scheme** 选项来提供命令行参数，编辑 **Scheme**，运行。然后选择 **Argument** 标签，在 **Launch** 的 **Arguments Pass** 中输入参数。

或者进入 Mac 的 Terminal 模式和 UNIX 的命令行环境。然后，可以找到程序可执行代码的目录（UNIX 的文件夹），或者下载命令行工具，使用 `gcc` 或 `clang` 编译程序。

## 11.9 把字符串转换为数字

数字既能以字符串形式储存，也能以数值形式储存。把数字储存为字符串就是储存数字字符。例如，数字 213 以 '2'、'1'、'3'、'\0' 的形式被储存在字符串数组中。以数值形式储存 213，储存的是 `int` 类型的值。

C 要求用数值形式进行数值运算（如，加法和比较）。但是在屏幕上显示数字则要求字符串形式，因为屏幕显示的是字符。`printf()` 和 `sprintf()` 函数，通过 `%d` 和其他转换说明，把数字从数值形式转换为字符串形式，`scanf()` 可以把输入字符串转换为数值形式。C 还有一些函数专门用于把字符串形式转换成数值形式。

假设你编写的程序需要使用数值命令形参，但是命令形参数被读取为字符串。因此，要使用数值必须先把字符串转换为数字。如果需要整数，可以使用 `atoi()` 函数（用于把字母数字转换成整数），该函数接受一个字符串作为参数，返回相应的整数值。程序清单 11.32 中的程序示例演示了该函数的用法。

## 程序清单 11.32 hello.c 程序

```
/* hello.c -- 把命令行参数转换为数字 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
 int i, times;

 if (argc < 2 || (times = atoi(argv[1])) < 1)
 printf("Usage: %s positive-number\n", argv[0]);
 else
 for (i = 0; i < times; i++)
 puts("Hello, good looking!");
 return 0;
}
```

该程序的运行示例：

```
$ hello 3
Hello, good looking!
Hello, good looking!
Hello, good looking!
```

\$ 是 UNIX 和 Linux 的提示符（一些 UNIX 系统使用%）。命令行参数 3 被储存为字符串 3\0。atoi() 函数把该字符串转换为整数值 3，然后该值被赋给 times。该值确定了执行 for 循环的次数。

如果运行该程序时没有提供命令行参数，那么 argc < 2 为真，程序给出一条提示信息后结束。如果 times 为 0 或负数，情况也是如此。C 语言逻辑运算符的求值顺序保证了如果 argc < 2，就不会对 atoi(argv[1]) 求值。

如果字符串仅以整数开头，atoi() 函数也能处理，它只把开头的整数转换为字符。例如，atoi("42regular") 将返回整数 42。如果在命令行输入 hello what 会怎样？在我们所用的 C 实现中，如果命令行参数不是数字，atoi() 函数返回 0。然而 C 标准规定，这种情况下的行为是未定义的。因此，使用有错误检测功能的 strtol() 函数（马上介绍）会更安全。

该程序中包含了 stdlib.h 头文件，因为从 ANSI C 开始，该头文件中包含了 atoi() 函数的原型。除此之外，还包含了 atof() 和 atol() 函数的原型。atof() 函数把字符串转换成 double 类型的值，atol() 函数把字符串转换成 long 类型的值。atof() 和 atol() 的工作原理和 atoi() 类似，因此它们分别返回 double 类型和 long 类型。

ANSI C 还提供一套更智能的函数：strtol() 把字符串转换成 long 类型的值，strtoul() 把字符串转换成 unsigned long 类型的值，strtod() 把字符串转换成 double 类型的值。这些函数的智能之处在于识别和报告字符串中的首字符是否是数字。而且，strtol() 和 strtoul() 还可以指定数字的进制。

下面的程序示例中涉及 strtol() 函数，其原型如下：

```
long strtol(const char * restrict nptr, char ** restrict endptr, int base);
```

这里，nptr 是指向待转换字符串的指针，endptr 是一个指针的地址，该指针被设置为标识输入数字结束字符的地址，base 表示以什么进制写入数字。程序清单 11.33 演示了该函数的用法。

## 程序清单 11.33 strcnvt.c 程序

```
/* strcnvt.c -- 使用 strtol() */
```

```

#include <stdio.h>
#include <stdlib.h>
#define LIM 30
char * s_gets(char * st, int n);

int main()
{
 char number[LIM];
 char * end;
 long value;

 puts("Enter a number (empty line to quit):");
 while (s_gets(number, LIM) && number[0] != '\0')
 {
 value = strtol(number, &end, 10); /* 十进制 */
 printf("base 10 input, base 10 output: %ld, stopped at %s (%d)\n",
 value, end, *end);
 value = strtol(number, &end, 16); /* 十六进制 */
 printf("base 16 input, base 10 output: %ld, stopped at %s (%d)\n",
 value, end, *end);
 puts("Next number:");
 }
 puts("Bye!\n");

 return 0;
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 int i = 0;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 while (st[i] != '\n' && st[i] != '\0')
 i++;
 if (st[i] == '\n')
 st[i] = '\0';
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

下面是该程序的输出示例：

```

Enter a number (empty line to quit):
10
base 10 input, base 10 output: 10, stopped at (0)
base 16 input, base 10 output: 16, stopped at (0)
Next number:
10atom
base 10 input, base 10 output: 10, stopped at atom (97)

```

```
base 16 input, base 10 output: 266, stopped at tom (116)
```

Next number:

Bye!

首先注意，当 `base` 分别为 10 和 16 时，字符串 "10" 分别被转换成数字 10 和 16。还要注意，如果 `end` 指向一个字符，`*end` 就是一个字符。因此，第 1 次转换在读到空字符时结束，此时 `end` 指向空字符。打印 `end` 会显示一个空字符串，以 `%d` 转换说明输出 `*end` 显示的是空字符的 ASCII 码。

对于第 2 个输入的字符串，当 `base` 为 10 时，`end` 的值是 '`a`' 字符的地址。所以打印 `end` 显示的是字符串 "atom"，打印 `*end` 显示的是 '`a`' 字符的 ASCII 码。然而，当 `base` 为 16 时，'`a`' 字符被识别为一个有效的十六进制数，`strtol()` 函数把十六进制数 `10a` 转换成十进制数 266。

`strtol()` 函数最多可以转换三十六进制，'`a`'~'`z`' 字符都可用作数字。`strtoul()` 函数与该函数类似，但是它把字符串转换成无符号值。`strtod()` 函数只以十进制转换，因此它值需要两个参数。

许多实现使用 `itoa()` 和 `ftoa()` 函数分别把整数和浮点数转换成字符串。但是这两个函数并不是 C 标准库的成员，可以用 `sprintf()` 函数代替它们，因为 `sprintf()` 的兼容性更好。

## 11.10 关键概念

许多程序都要处理文本数据。一个程序可能要求用户输入姓名、公司列表、地址、一种蕨类植物的学名、音乐剧的演员等。毕竟，我们用言语与现实世界互动，使用文本的例子不计其数。C 程序通过字符串的方式来处理它们。

字符串，无论是由字符数组、指针还是字符串常量标识，都储存为包含字符编码的一系列字节，并以空字符串结尾。C 提供库函数处理字符串，查找字符串并分析它们。尤其要牢记，应该使用 `strcmp()` 来代替关系运算符，当比较字符串时，应该使用 `strcpy()` 或 `strncpy()` 代替赋值运算符把字符串赋给字符数组。

## 11.11 本章小结

C 字符串是一系列 `char` 类型的字符，以空字符 ('\0') 结尾。字符串可以储存在字符数组中。字符串还可以用字符串常量来表示，里面都是字符，括在双引号中（空字符除外）。编译器提供空字符。因此，"joy" 被储存为 4 个字符 j、o、y 和 \0。`strlen()` 函数可以统计字符串的长度，空字符不计算在内。

字符串常量也叫作字符串——字面量，可用于初始化字符数组。为了容纳末尾的空字符，数组大小应该至少比容纳的数组长度多 1。也可以用字符串常量初始化指向 `char` 的指针。

函数使用指向字符串首字符的指针来表示待处理的字符串。通常，对应的实际参数是数组名、指针变量或用双引号括起来的字符串。无论是哪种情况，传递的都是首字符的地址。一般而言，没必要传递字符串的长度，因为函数可以通过末尾的空字符确定字符串的结束。

`fgets()` 函数获取一行输入，`puts()` 和 `fputs()` 函数显示一行输出。它们都是 `stdio.h` 头文件中的函数，用于代替已被弃用的 `gets()`。

C 库中有多个字符串处理函数。在 ANSI C 中，这些函数都声明在 `string.h` 文件中。C 库中还有许多字符处理函数，声明在 `ctype.h` 文件中。

给 `main()` 函数提供两个合适的形式参数，可以让程序访问命令行参数。第 1 个参数通常是 `int` 类型的 `argc`，其值是命令行的单词数量。第 2 个参数通常是一个指向数组的指针 `argv`，数组内含指向 `char` 的指针。每个指向 `char` 的指针都指向一个命令行参数字符串，`argv[0]` 指向命令名称，`argv[1]` 指向第一个命令行参数，以此类推。

`atoi()`、`atol()`和`atof()`函数把字符串形式的数字分别转换成`int`、`long`和`double`类型的数字。`strtol()`、`strtoul()`和`strtod()`函数把字符串形式的数字分别转换成`long`、`unsigned long`和`double`类型的数字。

## 11.12 复习题

复习题的参考答案在附录A中。

1. 下面字符串的声明有什么问题？

```
int main(void)
{
 char name[] = {'F', 'e', 's', 's' };
 ...
}
```

2. 下面的程序会打印什么？

```
#include <stdio.h>
int main(void)
{
 char note[] = "See you at the snack bar.";
 char *ptr;

 ptr = note;
 puts(ptr);
 puts(++ptr);
 note[7] = '\0';
 puts(note);
 puts(++ptr);
 return 0;
}
```

3. 下面的程序会打印什么？

```
#include <stdio.h>
#include <string.h>
int main(void)
{
 char food [] = "Yummy";
 char *ptr;

 ptr = food + strlen(food);
 while (--ptr >= food)
 puts(ptr);
 return 0;
}
```

4. 下面的程序会打印什么？

```
#include <stdio.h>
#include <string.h>
int main(void)
{
 char goldwyn[40] = "art of it all ";
 char samuel[40] = "I read p";
 const char * quote = "the way through.";

 strcat(goldwyn, quote);
 strcat(samuel, goldwyn);
```

```

 puts(samuel);
 return 0;
}

```

5. 下面的练习涉及字符串、循环、指针和递增指针。首先，假设定义了下面的函数：

```

#include <stdio.h>
char *pr(char *str)
{
 char *pc;

 pc = str;
 while (*pc)
 putchar(*pc++);
 do {
 putchar(--pc);
 } while (pc - str);
 return (pc);
}

```

考虑下面的函数调用：

x = pr("Ho Ho Ho!");

- a. 将打印什么？
- b. x 是什么类型？
- c. x 的值是什么？
- d. 表达式--pc 是什么意思？与--\*pc 有何不同？
- e. 如果用--pc 替换--\*pc，会打印什么？
- f. 两个 while 循环用来测试什么？
- g. 如果 pr() 函数的参数是空字符串，会怎样？
- h. 必须在主调函数中做什么，才能让 pr() 函数正常运行？

6. 假设有如下声明：

```
char sign = '$';
```

sign 占用多少字节的内存？'\$' 占用多少字节的内存？"\$" 占用多少字节的内存？

7. 下面的程序会打印出什么？

```

#include <stdio.h>
#include <string.h>
#define M1 "How are ya, sweetie? "
char M2[40] = "Beat the clock.";
char * M3 = "chat";
int main(void)
{
 char words[80];
 printf(M1);
 puts(M1);
 puts(M2);
 puts(M2 + 1);
 strcpy(words, M2);
 strcat(words, " Win a toy.");
 puts(words);
 words[4] = '\0';
 puts(words);
}

```

```

while (*M3)
 puts(M3++);
puts(--M3);
puts(--M3);
M3 = M1;
puts(M3);
return 0;
}

```

8. 下面的程序会打印出什么？

```

#include <stdio.h>
int main(void)
{
 char str1 [] = "gawsie";
 char str2 [] = "bletonism";
 char *ps;
 int i = 0;
 for (ps = str1; *ps != '\0'; ps++) {
 if (*ps == 'a' || *ps == 'e')
 putchar(*ps);
 else
 (*ps)--;
 putchar(*ps);
 }
 putchar('\n');
 while (str2[i] != '\0') {
 printf("%c", i % 3 ? str2[i] : '*');
 ++i;
 }
 return 0;
}

```

9. 本章定义的 `s_gets()` 函数，用指针表示法代替数组表示法便可减少一个变量 `i`。请改写该函数。
10. `strlen()` 函数接受一个指向字符串的指针作为参数，并返回该字符串的长度。请编写一个这样的函数。
11. 本章定义的 `s_gets()` 函数，可以用 `strchr()` 函数代替其中的 `while` 循环来查找换行符。请改写该函数。
12. 设计一个函数，接受一个指向字符串的指针，返回指向该字符串第 1 个空格字符的指针，或如果未找到空格字符，则返回空指针。
13. 重写程序清单 11.21，使用 `ctype.h` 头文件中的函数，以便无论用户选择大写还是小写，该程序都能正确识别答案。

## 11.13 编程练习

1. 设计并测试一个函数，从输入中获取下 `n` 个字符（包括空白、制表符、换行符），把结果储存在一个数组里，它的地址被传递作为一个参数。
2. 修改并编程练习 1 的函数，在 `n` 个字符后停止，或在读到第 1 个空白、制表符或换行符时停止，哪个先遇到哪个停止。不能只使用 `scanf()`。
3. 设计并测试一个函数，从一行输入中把一个单词读入一个数组中，并丢弃输入行中的其余字符。该函数应该跳过第 1 个非空白字符前面的所有空白。将一个单词定义为没有空白、制表符或换行符的

字符串序列。

4. 设计并测试一个函数，它类似编程练习 3 的描述，只不过它接受第 2 个参数指明可读取的最大字符数。
5. 设计并测试一个函数，搜索第 1 个函数形参指定的字符串，在其中查找第 2 个函数形参指定的字符首次出现的位置。如果成功，该函数返指向该字符的指针，如果在字符串中未找到指定字符，则返回空指针（该函数的功能与 `strchr()` 函数相同）。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。
6. 编写一个名为 `is_within()` 的函数，接受一个字符和一个指向字符串的指针作为两个函数形参。如果指定字符在字符串中，该函数返回一个非零值（即为真）。否则，返回 0（即为假）。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。
7. `strncpy(s1, s2, n)` 函数把 `s2` 中的 `n` 个字符拷贝至 `s1` 中，截断 `s2`，或者有必要的话在末尾添加空字符。如果 `s2` 的长度是 `n` 或多于 `n`，目标字符串不能以空字符结尾。该函数返回 `s1`。自己编写一个这样的函数，名为 `mystrncpy()`。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。
8. 编写一个名为 `string_in()` 的函数，接受两个指向字符串的指针作为参数。如果第 2 个字符串中包含第 1 个字符串，该函数将返回第 1 个字符串开始的地址。例如，`string_in("hats", "at")` 将返回 `hats` 中 `a` 的地址。否则，该函数返回空指针。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。
9. 编写一个函数，把字符串中的内容用其反序字符串代替。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。
10. 编写一个函数接受一个字符串作为参数，并删除字符串中的空格。在一个程序中测试该函数，使用循环读取输入行，直到用户输入一行空行。该程序应该应用该函数只每个输入的字符串，并显示处理后的字符串。
11. 编写一个函数，读入 10 个字符串或者读到 EOF 时停止。该程序为用户提供一个有 5 个选项的菜单：打印源字符串列表、以 ASCII 中的顺序打印字符串、按长度递增顺序打印字符串、按字符串中第 1 个单词的长度打印字符串、退出。菜单可以循环显示，除非用户选择退出选项。当然，该程序要能真正完成菜单中各选项的功能。
12. 编写一个程序，读取输入，直至读到 EOF，报告读入的单词数、大写字母数、小写字母数、标点符号数和数字字符数。使用 `ctype.h` 头文件中的函数。
13. 编写一个程序，反序显示命令行参数的单词。例如，命令行参数是 `see you later`，该程序应打印 `later you see`。
14. 编写一个通过命令行运行的程序计算幂。第 1 个命令行参数是 `double` 类型的数，作为幂的底数，第 2 个参数是整数，作为幂的指数。
15. 使用字符分类函数实现 `atoi()` 函数。如果输入的字符串不是纯数字，该函数返回 0。
16. 编写一个程序读取输入，直至读到文件结尾，然后把字符串打印出来。该程序识别和实现下面的命令行参数：

|    |            |
|----|------------|
| -p | 按原样打印      |
| -u | 把输入全部转换成大写 |
| -l | 把输入全部转换成小写 |

如果没有命令行参数，则让程序像是使用了 `-p` 参数那样运行。

# 存储类别、链接和内存管理

本章介绍以下内容：

- 关键字：auto、extern、static、register、const、volatile、restricted、\_Thread\_local、\_Atomic
- 函数：rand()、srand()、time()、malloc()、calloc()、free()
- 如何确定变量的作用域（可见的范围）和生命周期（它存在多长时间）
- 设计更复杂的程序

C 语言能让程序员恰到好处地控制程序，这是它的优势之一。程序员通过 C 的内存管理系统指定变量的作用域和生命周期，实现对程序的控制。合理使用内存储存数据是设计程序的一个要点。

## 12.1 存储类别

C 提供了多种不同的模型或存储类别 (*storage class*) 在内存中储存数据。要理解这些存储类别，先要复习一些概念和术语。

本书目前所有编程示例中使用的数据都储存在内存中。从硬件方面来看，被储存的每个值都占用一定的物理内存，C 语言把这样的一块内存称为对象 (*object*)。对象可以储存一个或多个值。一个对象可能并未储存实际的值，但是它在储存适当的值时一定具有相应的大小（面向对象编程中的对象指的是类对象，其定义包括数据和允许对数据进行的操作，C 不是面向对象编程语言）。

从软件方面来看，程序需要一种方法访问对象。这可以通过声明变量来完成：

```
int entity = 3;
```

该声明创建了一个名为 entity 的标识符 (*identifier*)。标识符是一个名称，在这种情况下，标识符可以用来指定 (*designate*) 特定对象的内容。标识符遵循变量的命名规则（第 2 章介绍过）。在该例中，标识符 entity 即是软件（即 C 程序）指定硬件内存中的对象的方式。该声明还提供了储存在对象中的值。

变量名不是指定对象的唯一途径。考虑下面的声明：

```
int * pt = &entity;
int ranks[10];
```

第 1 行声明中，pt 是一个标识符，它指定了一个储存地址的对象。但是，表达式 \*pt 不是标识符，因为它不是一个名称。然而，它确实指定了一个对象，在这种情况下，它与 entity 指定的对象相同。一般而言，那些指定对象的表达式被称为左值（第 5 章介绍过）。所以，entity 既是标识符也是左值；\*pt 既是表达式也是左值。按照这个思路，ranks + 2 \* entity 既不是标识符（不是名称），也不是左值（它不指定内存位置上的内容）。但是表达式 \*(ranks + 2 \* entity) 是一个左值，因为它的的确指定了特定内存位置的值，即 ranks 数组的第 7 个元素。顺带一提，ranks 的声明创建了一个可容纳 10 个 int 类型元素的对象，该数组的每个元素也是一个对象。

所有这些示例中，如果可以使用左值改变对象中的值，该左值就是一个可修改的左值(*modifiable lvalue*)。现在，考虑下面的声明：

```
const char * pc = "Behold a string literal!";
```

程序根据该声明把相应的字符串字面量储存在内存中，内含这些字符值的数组就是一个对象。由于数组中的每个字符都能被单独访问，所以每个字符也是一个对象。该声明还创建了一个标识符为 pc 的对象，储存着字符串的地址。由于可以设置 pc 重新指向其他字符串，所以标识符 pc 是一个可修改的左值。`const` 只能保证被 pc 指向的字符串内容不被修改，但是无法保证 pc 不指向别的字符串。由于`*pc` 指定了储存'B'字符的数据对象，所以`*pc` 是一个左值，但不是一个可修改的左值。与此类似，因为字符串字面量本身指定了储存字符串的对象，所以它也是一个左值，但不是可修改的左值。

可以用存储期(*storage duration*)描述对象，所谓存储期是指对象在内存中保留了多长时间。标识符用于访问对象，可以用作用域(*scope*)和链接(*linkage*)描述标识符，标识符的作用域和链接表明了程序的哪些部分可以使用它。不同的存储类别具有不同的存储期、作用域和链接。标识符可以在源代码的多文件中共享、可用于特定文件的任意函数中、可仅限于特定函数中使用，甚至只在函数中的某部分使用。对象可存在于程序的执行期，也可以仅存在于它所在函数的执行期。对于并发编程，对象可以在特定线程的执行期存在。可以通过函数调用的方式显式分配和释放内存。

我们先学习作用域、链接和存储期的含义，再介绍具体的存储类别。

### 12.1.1 作用域

作用域描述程序中可访问标识符的区域。一个 C 变量的作用域可以是块作用域、函数作用域、函数原型作用域或文件作用域。到目前为止，本书程序示例中使用的变量几乎都具有块作用域。块是用一对花括号括起来的代码区域。例如，整个函数体是一个块，函数中的任意复合语句也是一个块。定义在块中的变量具有块作用域(*block scope*)，块作用域变量的可见范围是从定义处到包含该定义的块的末尾。另外，虽然函数的形式参数声明在函数的左花括号之前，但是它们也具有块作用域，属于函数体这个块。所以到目前为止，我们使用的局部变量(包括函数的形式参数)都具有块作用域。因此，下面代码中的变量 cleo 和 patrick 都具有块作用域：

```
double blocky(double cleo)
{
 double patrick = 0.0;
 ...
 return patrick;
}
```

声明在内层块中的变量，其作用域仅局限于该声明所在的块：

```
double blocky(double cleo)
{
 double patrick = 0.0;
 int i;
 for (i = 0; i < 10; i++)
 {
 double q = cleo * i; // q 的作用域开始
 ...
 patrick *= q;
 } // q 的作用域结束
 ...
 return patrick;
}
```

在该例中，`q` 的作用域仅限于内层块，只有内层块中的代码才能访问 `q`。

以前，具有块作用域的变量都必须声明在块的开头。C99 标准放宽了这一限制，允许在块中的任意位置声明变量。因此，对于 `for` 的循环头，现在可以这样写：

```
for (int i = 0; i < 10; i++)
 printf("A C99 feature: i = %d", i);
```

为适应这个新特性，C99 把块的概念扩展到包括 `for` 循环、`while` 循环、`do while` 循环和 `if` 语句所控制的代码，即使这些代码没有用花括号括起来，也算是块的一部分。所以，上面 `for` 循环中的变量 `i` 被视为 `for` 循环块的一部分，它的作用域仅限于 `for` 循环。一旦程序离开 `for` 循环，就不能再访问 `i`。

函数作用域 (*function scope*) 仅用于 `goto` 语句的标签。这意味着即使一个标签首次出现在函数的内层块中，它的作用域也延伸至整个函数。如果在两个块中使用相同的标签会很混乱，标签的函数作用域防止了这样的事情发生。

函数原型作用域 (*function prototype scope*) 用于函数原型中的形参名（变量名），如下所示：

```
int mighty(int mouse, double large);
```

函数原型作用域的范围是从形参定义处到原型声明结束。这意味着，编译器在处理函数原型中的形参时只关心它的类型，而形参名（如果有的话）通常无关紧要。而且，即使有形参名，也不必与函数定义中的形参名相匹配。只有在变长数组中，形参名才有用：

```
void use_a_VLA(int n, int m, ar[n][m]);
```

方括号中必须使用在函数原型中已声明的名称。

变量的定义在函数的外面，具有文件作用域 (*file scope*)。具有文件作用域的变量，从它的定义处到该定义所在文件的末尾均可见。考虑下面的例子：

```
#include <stdio.h>
int units = 0; /* 该变量具有文件作用域 */
void critic(void);
int main(void)
{
 ...
}
void critic(void)
{
 ...
}
```

这里，变量 `units` 具有文件作用域，`main()` 和 `critic()` 函数都可以使用它（更准确地说，`units` 具有外部链接文件作用域，稍后讲解）。由于这样的变量可用于多个函数，所以文件作用域变量也称为全局变量 (*global variable*)。

## 注意 翻译单元和文件

你认为的多个文件在编译器中可能以一个文件出现。例如，通常在源代码 (.c 扩展名) 中包含一个或多个头文件 (.h 扩展名)。头文件会依次包含其他头文件，所以会包含多个单独的物理文件。但是，C 预处理实际上是用包含的头文件内容替换 `#include` 指令。所以，编译器源代码文件和所有的头文件都看成是一个包含信息的单独文件。这个文件被称为翻译单元 (*translation unit*)。描述一个具有文件作用域的变量时，它的实际可见范围是整个翻译单元。如果程序由多个源代码文件组成，那么该程序也将由多个翻译单元组成。每个翻译单元均对应一个源代码文件和它所包含的文件。

## 12.1.2 链接

接下来，我们介绍链接。C 变量有 3 种链接属性：外部链接、内部链接或无链接。具有块作用域、函数作用域或函数原型作用域的变量都是无链接变量。这意味着这些变量属于定义它们的块、函数或原型私有。具有文件作用域的变量可以是外部链接或内部链接。外部链接变量可以在多文件程序中使用，内部链接变量只能在一个翻译单元中使用。

### 注意 正式和非正式术语

C 标准用“内部链接的文件作用域”描述仅限于一个翻译单元（即一个源代码文件和它所包含的头文件）的作用域，用“外部链接的文件作用域”描述可延伸至其他翻译单元的作用域。但是，对程序员而言这些术语太长了。一些程序员把“内部链接的文件作用域”简称为“文件作用域”，把“外部链接的文件作用域”简称为“全局作用域”或“程序作用域”。

如何知道文件作用域变量是内部链接还是外部链接？可以查看外部定义中是否使用了存储类别说明符 `static`：

```
int giants = 5; // 文件作用域，外部链接
static int dodgers = 3; // 文件作用域，内部链接
int main()
{
 ...
}
```

该文件和同一程序的其他文件都可以使用变量 `giants`。而变量 `dodgers` 属文件私有，该文件中的任意函数都可使用它。

## 12.1.3 存储期

作用域和链接描述了标识符的可见性。存储期描述了通过这些标识符访问的对象的生存期。C 对象有 4 种存储期：静态存储期、线程存储期、自动存储期、动态分配存储期。

如果对象具有静态存储期，那么它在程序的执行期间一直存在。文件作用域变量具有静态存储期。注意，对于文件作用域变量，关键字 `static` 表明了其链接属性，而非存储期。以 `static` 声明的文件作用域变量具有内部链接。但是无论是内部链接还是外部链接，所有的文件作用域变量都具有静态存储期。

线程存储期用于并发程序设计，程序执行可被分为多个线程。具有线程存储期的对象，从被声明时到线程结束一直存在。以关键字 `_Thread_local` 声明一个对象时，每个线程都获得该变量的私有备份。

块作用域的变量通常都具有自动存储期。当程序进入定义这些变量的块时，为这些变量分配内存；当退出这个块时，释放刚才为变量分配的内存。这种做法相当于把自动变量占用的内存视为一个可重复使用的工作区或暂存区。例如，一个函数调用结束后，其变量占用的内存可用于储存下一个被调用函数的变量。

变长数组稍有不同，它们的存储期从声明处到块的末尾，而不是从块的开始处到块的末尾。

我们到目前为止使用的局部变量都是自动类别。例如，在下面的代码中，变量 `number` 和 `index` 在每次调用 `bore()` 函数时被创建，在离开函数时被销毁：

```

void bore(int number)
{
 int index;
 for (index = 0; index < number; index++)
 puts("They don't make them the way they used to.\n");
 return 0;
}

```

然而，块作用域变量也能具有静态存储期。为了创建这样的变量，要把变量声明在块中，且在声明前面加上关键字 `static`:

```

void more(int number)
{
 int index;
 static int ct = 0;
 ...
 return 0;
}

```

这里，变量 `ct` 储存在静态内存中，它从程序被载入到程序结束期间都存在。但是，它的作用域定义在 `more()` 函数块中。只有在执行该函数时，程序才能使用 `ct` 访问它所指定的对象（但是，该函数可以给其他函数提供该存储区的地址以便间接访问该对象，例如通过指针形参或返回值）。

C 使用作用域、链接和存储期为变量定义了多种存储方案。本书不涉及并发程序设计，所以不再赘述这方面的内容。已分配存储期在本章后面介绍。因此，剩下 5 种存储类别：自动、寄存器、静态块作用域、静态外部链接、静态内部链接，如表 12.1 所列。现在，我们已经介绍了作用域、链接和存储期，接下来将详细讨论这些存储类别。

表 12.1 5 种存储类别

| 存储类别   | 存储期 | 作用域 | 链接 | 声明方式                            |
|--------|-----|-----|----|---------------------------------|
| 自动     | 自动  | 块   | 无  | 块内                              |
| 寄存器    | 自动  | 块   | 无  | 块内，使用关键字 <code>register</code>  |
| 静态外部链接 | 静态  | 文件  | 外部 | 所有函数外                           |
| 静态内部链接 | 静态  | 文件  | 内部 | 所有函数外，使用关键字 <code>static</code> |
| 静态无链接  | 静态  | 块   | 无  | 块内，使用关键字 <code>static</code>    |

## 12.1.4 自动变量

属于自动存储类别的变量具有自动存储期、块作用域且无链接。默认情况下，声明在块或函数头中的任何变量都属于自动存储类别。为了更清楚地表达你的意图（例如，为了表明有意覆盖一个外部变量定义，或者强调不要把该变量改为其他存储类别），可以显式使用关键字 `auto`，如下所示：

```

int main(void)
{
 auto int plox;
}

```

关键字 `auto` 是存储类别说明符 (*storage-class specifier*)。`auto` 关键字在 C++ 中的用法完全不同，如果编写 C/C++ 兼容的程序，最好不要使用 `auto` 作为存储类别说明符。

块作用域和无链接意味着只有在变量定义所在的块中才能通过变量名访问该变量（当然，参数用于传递变量的值和地址给另一个函数，但是这是间接的方法）。另一个函数可以使用同名变量，但是该变量是储存在不同内存位置上的另一个变量。

变量具有自动存储期意味着，程序在进入该变量声明所在的块时变量存在，程序在退出该块时变量消失。原来该变量占用的内存位置现在可做他用。

接下来分析一下嵌套块的情况。块中声明的变量仅限于该块及其包含的块使用。

```
int loop(int n)
{
 int m; // m 的作用域
 scanf("%d", &m);
 {
 int i; // m 和 i 的作用域
 for (i = m; i < n; i++)
 puts("i is local to a sub-block\n");
 }
 return m; // m 的作用域, i 已经消失
}
```

在上面的代码中，`i` 仅在内层块中可见。如果在内层块的前面或后面使用 `i`，编译器会报错。通常，在设计程序时用不到这个特性。然而，如果这个变量仅供该块使用，那么在块中就近定义该变量也很方便。这样，可以在靠近使用变量的地方记录其含义。另外，这样的变量只有在使用时才占用内存。变量 `n` 和 `m` 分别定义在函数头和外层块中，它们的作用域是整个函数，而且在调用函数到函数结束期间都一直存在。

如果内层块中声明的变量与外层块中的变量同名会怎样？内层块会隐藏外层块的定义。但是离开内层块后，外层块变量的作用域又回到了原来的作用域。程序清单 12.1 演示了这一过程。

程序清单 12.1 hiding.c 程序

```
// hiding.c -- 块中的变量
#include <stdio.h>
int main()
{
 int x = 30; // 原始的 x

 printf("x in outer block: %d at %p\n", x, &x);
 {
 int x = 77; // 新的 x, 隐藏了原始的 x
 printf("x in inner block: %d at %p\n", x, &x);
 }
 printf("x in outer block: %d at %p\n", x, &x);
 while (x++ < 33) // 原始的 x
 {
 int x = 100; // 新的 x, 隐藏了原始的 x
 x++;
 printf("x in while loop: %d at %p\n", x, &x);
 }
 printf("x in outer block: %d at %p\n", x, &x);

 return 0;
}
```

下面是该程序的输出：

```
x in outer block: 30 at 0x7fff5fbff8c8
x in inner block: 77 at 0x7fff5fbff8c4
```

```
x in outer block: 30 at 0x7fff5fbff8c8
x in while loop: 101 at 0x7fff5fbff8c0
x in while loop: 101 at 0x7fff5fbff8c0
x in while loop: 101 at 0x7fff5fbff8c0
x in outer block: 34 at 0x7fff5fbff8c8
```

首先，程序创建了变量 `x` 并初始化为 30，如第 1 条 `printf()` 语句所示。然后，定义了一个新的变量 `x`，并设置为 77，如第 2 条 `printf()` 语句所示。根据显示的地址可知，新变量隐藏了原始的 `x`。第 3 条 `printf()` 语句位于第 1 个内层块后面，显示的是原始的 `x` 的值，这说明原始的 `x` 既没有消失也不曾改变。

也许该程序最难懂的是 `while` 循环。`while` 循环的测试条件中使用的是原始的 `x`：

```
while(x++ < 33)
```

在该循环中，程序创建了第 3 个 `x` 变量，该变量只定义在 `while` 循环中。所以，当执行到循环体中的 `x++` 时，递增为 101 的是新的 `x`，然后 `printf()` 语句显示了该值。每轮迭代结束，新的 `x` 变量就消失。然后循环的测试条件使用并递增原始的 `x`，再次进入循环体，再次创建新的 `x`。在该例中，这个 `x` 被创建和销毁了 3 次。注意，该循环必须在测试条件中递增 `x`，因为如果在循环体中递增 `x`，那么递增的是循环体中创建的 `x`，而非测试条件中使用的原始 `x`。

我们使用的编译器在创建 `while` 循环体中的 `x` 时，并未复用内层块中 `x` 占用的内存，但是有些编译器会这样做。

该程序示例的用意不是鼓励读者要编写类似的代码（根据 C 的命名规则，要想出别的变量名并不难），而是为了解释在内层块中定义变量的具体情况。

## 1. 没有花括号的块

前面提到一个 C99 特性：作为循环或 `if` 语句的一部分，即使不使用花括号（`{}`），也是一个块。更完整地说，整个循环是它所在块的子块（*sub-block*），循环体是整个循环块的子块。与此类似，`if` 语句是一个块，与其相关联的子语句是 `if` 语句的子块。这些规则会影响到声明的变量和这些变量的作用域。程序清单 12.2 演示了 `for` 循环中该特性的用法。

程序清单 12.2 forc99.c 程序

---

```
// forc99.c -- 新的 C99 块规则
#include <stdio.h>
int main()
{
 int n = 8;

 printf(" Initially, n = %d at %p\n", n, &n);
 for (int n = 1; n < 3; n++)
 printf(" loop 1: n = %d at %p\n", n, &n);
 printf("After loop 1, n = %d at %p\n", n, &n);
 for (int n = 1; n < 3; n++)
 {
 printf(" loop 2 index n = %d at %p\n", n, &n);
 int n = 6;
 printf(" loop 2: n = %d at %p\n", n, &n);
 n++;
 }
 printf("After loop 2, n = %d at %p\n", n, &n);

 return 0;
}
```

---

假设编译器支持 C 语言的这个新特性，该程序的输出如下：

```
Initially, n = 8 at 0xffff5fbff8c8
loop 1: n = 1 at 0xffff5fbff8c4
loop 1: n = 2 at 0xffff5fbff8c4
After loop 1, n = 8 at 0xffff5fbff8c8
loop 2 index n = 1 at 0xffff5fbff8c0
loop 2: n = 6 at 0xffff5fbff8bc
loop 2 index n = 2 at 0xffff5fbff8c0
loop 2: n = 6 at 0xffff5fbff8bc
After loop 2, n = 8 at 0xffff5fbff8c8
```

第 1 个 for 循环头中声明的 n，其作用域作用至循环末尾，而且隐藏了原始的 n。但是，离开循环后，原始的 n 又起作用了。

第 2 个 for 循环头中声明的 n 作为循环的索引，隐藏了原始的 n。然后，在循环体中又声明了一个 n，隐藏了索引 n。结束一轮迭代后，声明在循环体中的 n 消失，循环头使用索引 n 进行测试。当整个循环结束时，原始的 n 又起作用了。再次提醒读者注意，没必要在程序中使用相同的变量名。如果用了，各变量的情况如上所述。

### 注意 支持 C99 和 C11

有些编译器并不支持 C99/C11 的这些作用域规则（Microsoft Visual Studio 2012 就是其中之一）。有些编译器会提供激活这些规则的选项。例如，撰写本书时，gcc 默认支持了 C99 的许多特性，但是要用 -std=c99 选项激活程序清单 12.2 中使用的特性：

```
gcc -std=c99 forc99.c
```

与此类似，gcc 或 clang 都要使用 -std=c1x 或 -std=c11 选项，才支持 C11 特性。

## 2. 自动变量的初始化

自动变量不会初始化，除非显式初始化它。考虑下面的声明：

```
int main(void)
{
 int repid;
 int tents = 5;
```

tents 变量被初始化为 5，但是 repid 变量的值是之前占用分配给 repid 的空间中的任意值（如果说有的话），别指望这个值是 0。可以用非常量表达式（*non-constant expression*）初始化自动变量，前提是所用的变量已在前面定义过：

```
int main(void)
{
 int ruth = 1;
 int rance = 5 * ruth; // 使用之前定义的变量
```

### 12.1.5 寄存器变量

变量通常储存在计算机内存中。如果幸运的话，寄存器变量储存在 CPU 的寄存器中，或者概括地说，储存在最快的可用内存中。与普通变量相比，访问和处理这些变量的速度更快。由于寄存器变量储存在寄存器而非内存中，所以无法获取寄存器变量的地址。绝大多数方面，寄存器变量和自动变量都一样。也就是说，它们都是块作用域、无链接和自动存储期。使用存储类别说明符 register 便可声明寄存器变量：

```
int main(void)
{
 register int quick;
```

我们刚才说“如果幸运的话”，是因为声明变量为 `register` 类别与直接命令相比更像是一种请求。编译器必须根据寄存器或最快可用内存的数量衡量你的请求，或者直接忽略你的请求，所以可能不会如你所愿。在这种情况下，寄存器变量就变成普通的自动变量。即使是这样，仍然不能对该变量使用地址运算符。

在函数头中使用关键字 `register`，便可请求形参是寄存器变量：

```
void macho(register int n)
```

可声明为 `register` 的数据类型有限。例如，处理器中的寄存器可能没有足够大的空间来储存 `double` 类型的值。

## 12.1.6 块作用域的静态变量

静态变量 (*static variable*) 听起来自相矛盾，像是一个不可变的变量。实际上，静态的意思是该变量在内存中原地不动，并不是说它的值不变。具有文件作用域的变量自动具有（也必须是）静态存储期。前面提到过，可以创建具有静态存储期、块作用域的局部变量。这些变量和自动变量一样，具有相同的作用域，但是程序离开它们所在的函数后，这些变量不会消失。也就是说，这种变量具有块作用域、无链接，但是具有静态存储期。计算机在多次函数调用之间会记录它们的值。在块中（提供块作用域和无链接）以存储类别说明符 `static`（提供静态存储期）声明这种变量。程序清单 12.3 演示了一个这样的例子。

程序清单 12.3 loc\_stat.c 程序

---

```
/* loc_stat.c -- 使用局部静态变量 */
#include <stdio.h>
void trystat(void);

int main(void)
{
 int count;

 for (count = 1; count <= 3; count++)
 {
 printf("Here comes iteration %d:\n", count);
 trystat();
 }

 return 0;
}

void trystat(void)
{
 int fade = 1;
 static int stay = 1;

 printf("fade = %d and stay = %d\n", fade++, stay++);
}
```

---

注意，`trystat()` 函数先打印再递增变量的值。该程序的输出如下：

```
Here comes iteration 1:
fade = 1 and stay = 1
```

```
Here comes iteration 2:
fade = 1 and stay = 2
Here comes iteration 3:
fade = 1 and stay = 3
```

静态变量 `stay` 保存了它被递增 1 后的值，但是 `fade` 变量每次都是 1。这表明了初始化的不同：每次调用 `trystat()` 都会初始化 `fade`，但是 `stay` 只在编译 `strstat()` 时被初始化一次。如果未显式初始化静态变量，它们会被初始化为 0。

下面两个声明很相似：

```
int fade = 1;
static int stay = 1;
```

第 1 条声明确实是 `trystat()` 函数的一部分，每次调用该函数时都会执行这条声明。这是运行时行为。第 2 条声明实际上并不是 `trystat()` 函数的一部分。如果逐步调试该程序会发现，程序似乎跳过了这条声明。这是因为静态变量和外部变量在程序被载入内存时已执行完毕。把这条声明放在 `trystat()` 函数中是为了告诉编译器只有 `trystat()` 函数才能看到该变量。这条声明并未在运行时执行。

不能在函数的形参中使用 `static`：

```
int wontwork(static int flu); // 不允许
```

“局部静态变量”是描述具有块作用域的静态变量的另一个术语。阅读一些老的 C 文献时会发现，这种存储类别被称为内部静态存储类别 (*internal static storage class*)。这里的内部指的是函数内部，而非内部链接。

### 12.1.7 外部链接的静态变量

外部链接的静态变量具有文件作用域、外部链接和静态存储期。该类别有时称为外部存储类别 (*external storage class*)，属于该类别的变量称为外部变量 (*external variable*)。把变量的定义性声明 (*defining declaration*) 放在所有函数的外面便创建了外部变量。当然，为了指出该函数使用了外部变量，可以在函数中用关键字 `extern` 再次声明。如果一个源代码文件使用的外部变量定义在另一个源代码文件中，则必须用 `extern` 在该文件中声明该变量。如下所示：

```
int Errupt; /* 外部定义的变量 */
double Up[100]; /* 外部定义的数组 */
extern char Coal; /* 如果 Coal 被定义在另一个文件， */
 /* 则必须这样声明 */

void next(void);
int main(void)
{
 extern int Errupt; /* 可选的声明 */

 extern double Up[]; /* 可选的声明 */
 ...
}
void next(void)
{
 ...
}
```

注意，在 `main()` 中声明 `Up` 数组时（这是可选的声明）不用指明数组大小，因为第 1 次声明已经提供了数组大小信息。`main()` 中的两条 `extern` 声明完全可以省略，因为外部变量具有文件作用域，所以 `Errupt` 和 `Up` 从声明处到文件结尾都可见。它们出现在那里，仅为了说明 `main()` 函数要使用这两个变量。

如果省略掉函数中的 `extern` 关键字，相当于创建了一个自动变量。去掉下面声明中的 `extern`:

```
extern int Errupt;
```

便成为:

```
int Errupt;
```

这使得编译器在 `main()` 中创建了一个名为 `Errupt` 的自动变量。它是一个独立的局部变量，与原来的外部变量 `Errupt` 不同。该局部变量仅 `main()` 中可见，但是外部变量 `Errupt` 对于该文件的其他函数（如 `next()`）也可见。简而言之，在执行块中的语句时，块作用域中的变量将“隐藏”文件作用域中的同名变量。如果不得已要使用与外部变量同名的局部变量，可以在局部变量的声明中使用 `auto` 存储类别说明符明确表达这种意图。

外部变量具有静态存储期。因此，无论程序执行到 `main()`、`next()` 还是其他函数，数组 `Up` 及其值都一直存在。

下面 3 个示例演示了外部和自动变量的一些使用情况。示例 1 中有一个外部变量 `Hocus`。该变量对 `main()` 和 `magic()` 均可见。

```
/* 示例 1 */
int Hocus;
int magic();
int main(void)
{
 extern int Hocus; // Hocus 之前已声明为外部变量
 ...
}
int magic()
{
 extern int Hocus; // 与上面的 Hocus 是同一个变量
 ...
}
```

示例 2 中有一个外部变量 `Hocus`，对两个函数均可见。这次，在默认情况下对 `magic()` 可见。

```
/*示例 2 */
int Hocus;
int magic();
int main(void)
{
 extern int Hocus; // Hocus 之前已声明为外部变量
 ...
}
int magic()
{
 //并未在该函数中声明 Hocus，但是仍可使用该变量
 ...
}
```

在示例 3 中，创建了 4 个独立的变量。`main()` 中的 `Hocus` 变量默认是自动变量，属于 `main()` 私有。`magic()` 中的 `Hocus` 变量被显式声明为自动，只有 `magic()` 可用。外部变量 `Hocus` 对 `main()` 和 `magic()` 均不可见，但是对该文件中未创建局部 `Hocus` 变量的其他函数可见。最后，`Pocus` 是外部变量，`magic()` 可见，但是 `main()` 不可见，因为 `Pocus` 被声明在 `main()` 后面。

```
/* 示例 3 */
int Hocus;
int magic();
```

```

int main(void)
{
 int Hocus; // 声明 Hocus，默认是自动变量
 ...
}

int Pocus;
int magic()
{
 auto int Hocus; // 把局部变量 Hocus 显式声明为自动变量
 ...
}

```

这 3 个示例演示了外部变量的作用域是：从声明处到文件结尾。除此之外，还说明了外部变量的生命周期。外部变量 Hocus 和 Pocus 在程序运行中一直存在，因为它们不受限于任何函数，不会在某个函数返回后就消失。

## 1. 初始化外部变量

外部变量和自动变量类似，也可以被显式初始化。与自动变量不同的是，如果未初始化外部变量，它们会被自动初始化为 0。这一原则也适用于外部定义的数组元素。与自动变量的情况不同，只能使用常量表达式初始化文件作用域变量：

```

int x = 10; // 没问题，10 是常量
int y = 3 + 20; // 没问题，用于初始化的是常量表达式
size_t z = sizeof(int); // 没问题，用于初始化的是常量表达式
int x2 = 2 * x; // 不行，x 是变量

```

(只要不是变长数组，`sizeof` 表达式可被视为常量表达式。)

## 2. 使用外部变量

下面来看一个使用外部变量的示例。假设有两个函数 `main()` 和 `critic()`，它们都要访问变量 `units`。可以把 `units` 声明在这两个函数的上面，如程序清单 12.4 所示（注意：该例的目的是演示外部变量的工作原理，并非它的典型用法）。

程序清单 12.4 `global.c` 程序

```

/* global.c -- 使用外部变量 */
#include <stdio.h>
int units = 0; /* 外部变量 */
void critic(void);
int main(void)
{
 extern int units; /* 可选的重复声明 */

 printf("How many pounds to a firkin of butter?\n");
 scanf("%d", &units);
 while (units != 56)
 critic();
 printf("You must have looked it up!\n");

 return 0;
}

void critic(void)
{

```

```

/* 删除了可选的重复声明 */
printf("No luck, my friend. Try again.\n");
scanf("%d", &units);
}

```

下面是该程序的输出示例：

```

How many pounds to a firkin of butter?
14
No luck, my friend. Try again.
56
You must have looked it up!

```

注意，`critic()`是如何读取 `units` 的第 2 个值的。当 `while` 循环结束时，`main()` 也知道 `units` 的新值。所以 `main()` 函数和 `critic()` 都可以通过标识符 `units` 访问相同的变量。用 C 的术语来描述是，`units` 具有文件作用域、外部链接和静态存储期。

把 `units` 定义在所有函数定义外面（即外部），`units` 便是一个外部变量，对 `units` 定义下面的所有函数均可见。因此，`critics()` 可以直接使用 `units` 变量。

类似地，`main()` 也可直接访问 `units`。但是，`main()` 中确实有如下声明：

```
extern int units;
```

本例中，以上声明主要是为了指出该函数要使用这个外部变量。存储类别说明符 `extern` 告诉编译器，该函数中任何使用 `units` 的地方都引用同一个定义在函数外部的变量。再次强调，`main()` 和 `critic()` 使用的都是外部定义的 `units`。

### 3. 外部名称

C99 和 C11 标准都要求编译器识别局部标识符的前 63 个字符和外部标识符的前 31 个字符。这修订了以前的标准，即编译器识别局部标识符前 31 个字符和外部标识符前 6 个字符。你所用的编译器可能还执行以前的规则。外部变量名比局部变量名的规则严格，是因为外部变量名还要遵循局部环境规则，所受的限制更多。

### 4. 定义和声明

下面进一步介绍定义变量和声明变量的区别。考虑下面的例子：

```

int tern = 1; /* tern 被定义 */
main()
{
 extern int tern; /* 使用在别处定义的 tern */
}

```

这里，`tern` 被声明了两次。第 1 次声明为变量预留了存储空间，该声明构成了变量的定义。第 2 次声明只告诉编译器使用之前已创建的 `tern` 变量，所以这不是定义。第 1 次声明被称为定义式声明 (*defining declaration*)，第 2 次声明被称为引用式声明 (*referencing declaration*)。关键字 `extern` 表明该声明不是定义，因为它指示编译器去别处查询其定义。

假设这样写：

```

extern int tern;
int main(void)
{
}

```

编译器会假设 `tern` 实际的定义在该程序的别处，也许在别的文件中。该声明并不会引起分配存储空间。因此，不要用关键字 `extern` 创建外部定义，只用它来引用现有的外部定义。

外部变量只能初始化一次，且必须在定义该变量时进行。假设有下面的代码：

```
// file_one.c
char permis = 'N';
...
// file_two.c
extern char permis = 'Y'; /* 错误 */

file_two 中的声明是错误的，因为 file_one.c 中的定义式声明已经创建并初始化了 permis。
```

## 12.1.8 内部链接的静态变量

该存储类别的变量具有静态存储期、文件作用域和内部链接。在所有函数外部（这点与外部变量相同），用存储类别说明符 `static` 定义的变量具有这种存储类别：

```
static int svil = 1; // 静态变量，内部链接
int main(void)
{
```

这种变量过去称为外部静态变量 (*external static variable*)，但是这个术语有点自相矛盾（这些变量具有内部链接）。但是，没有合适的新简称，所以只能用内部链接的静态变量 (*static variable with internal linkage*)。普通的外部变量可用于同一程序中任意文件中的函数，但是内部链接的静态变量只能用于同一个文件中的函数。可以使用存储类别说明符 `extern`，在函数中重复声明任何具有文件作用域的变量。这样的声明并不会改变其链接属性。考虑下面的代码：

```
int traveler = 1; // 外部链接
static int stayhome = 1; // 内部链接
int main()
{
 extern int traveler; // 使用定义在别处的 traveler
 extern int stayhome; // 使用定义在别处的 stayhome
 ...
}
```

对于该程序所在的翻译单元，`traveler` 和 `stayhome` 都具有文件作用域，但是只有 `traveler` 可用于其他翻译单元（因为它具有外部链接）。这两个声明都使用了 `extern` 关键字，指明了 `main()` 中使用的这两个变量的定义都在别处，但是这并未改变 `stayhome` 的内部链接属性。

## 12.1.9 多文件

只有当程序由多个翻译单元组成时，才体现区别内部链接和外部链接的重要性。接下来简要介绍一下。

复杂的 C 程序通常由多个单独的源代码文件组成。有时，这些文件可能要共享一个外部变量。C 通过在一个文件中进行定义式声明，然后在其他文件中进行引用式声明来实现共享。也就是说，除了一个定义式声明外，其他声明都要使用 `extern` 关键字。而且，只有定义式声明才能初始化变量。

注意，如果外部变量定义在一个文件中，那么其他文件在使用该变量之前必须先声明它（用 `extern` 关键字）。也就是说，在某文件中对外部变量进行定义式声明只是单方面允许其他文件使用该变量，其他文件在用 `extern` 声明之前不能直接使用它。

过去，不同的编译器遵循不同的规则。例如，许多 UNIX 系统允许在多个文件中不使用 `extern` 关键字声明变量，前提是只有一个带初始化的声明。编译器会把文件中一个带初始化的声明视为该变量的定义。

## 12.1.10 存储类别说明符

读者可能已经注意到了，关键字 `static` 和 `extern` 的含义取决于上下文。C 语言有 6 个关键字作为存储类别说明符：`auto`、`register`、`static`、`extern`、`_Thread_local` 和 `typedef`。`typedef` 关键字与任何内存存储无关，把它归于此类有一些语法上的原因。尤其是，在绝大多数情况下，不能在声明中使用多个存储类别说明符，所以这意味着不能使用多个存储类别说明符作为 `typedef` 的一部分。唯一例外的是 `_Thread_local`，它可以和 `static` 或 `extern` 一起使用。

`auto` 说明符表明变量是自动存储期，只能用于块作用域的变量声明中。由于在块中声明的变量本身就具有自动存储期，所以使用 `auto` 主要是为了明确表达要使用与外部变量同名的局部变量的意图。

`register` 说明符也只用于块作用域的变量，它把变量归为寄存器存储类别，请求最快速度访问该变量。同时，还保护了该变量的地址不被获取。

用 `static` 说明符创建的对象具有静态存储期，载入程序时创建对象，当程序结束时对象消失。如果 `static` 用于文件作用域声明，作用域受限于该文件。如果 `static` 用于块作用域声明，作用域则受限于该块。因此，只要程序在运行对象就存在并保留其值，但是只有在执行块内的代码时，才能通过标识符访问。块作用域的静态变量无链接。文件作用域的静态变量具有内部链接。

`extern` 说明符表明声明的变量定义在别处。如果包含 `extern` 的声明具有文件作用域，则引用的变量必须具有外部链接。如果包含 `extern` 的声明具有块作用域，则引用的变量可能具有外部链接或内部链接，这取决于该变量的定义式声明。

### 小结：存储类别

自动变量具有块作用域、无链接、自动存储期。它们是局部变量，属于其定义所在块（通常指函数）私有。寄存器变量的属性和自动变量相同，但是编译器会使用更快的内存或寄存器储存它们。不能获取寄存器变量的地址。

具有静态存储期的变量可以具有外部链接、内部链接或无链接。在同一个文件所有函数的外部声明的变量是外部变量，具有文件作用域、外部链接和静态存储期。如果在这种声明前面加上关键字 `static`，那么其声明的变量具有文件作用域、内部链接和静态存储期。如果在函数中用 `static` 声明一个变量，则该变量具有块作用域、无链接、静态存储期。

具有自动存储期的变量，程序在进入该变量的声明所在块时才为其分配内存，在退出该块时释放之前分配的内存。如果未初始化，自动变量中是垃圾值。程序在编译时为具有静态存储期的变量分配内存，并在程序的运行过程中一直保留这块内存。如果未初始化，这样的变量会被设置为 0。

具有块作用域的变量是局部的，属于包含该声明的块私有。具有文件作用域的变量对文件（或翻译单元）中位于其声明后面的所有函数可见。具有外部链接的文件作用域变量，可用于该程序的其他翻译单元。具有内部链接的文件作用域变量，只能用于其声明所在的文件内。

下面用一个简短的程序使用了 5 种存储类别。该程序包含两个文件（程序清单 12.5 和程序清单 12.6），所以必须使用多文件编译（参见第 9 章或参看编译器的指导手册）。该示例仅为了让读者熟悉 5 种存储类别的用法，并不是提供设计模型，好的设计可以不需要使用文件作用域变量。

#### 程序清单 12.5 parta.c 程序

---

```
// parta.c --- 不同的存储类别
```

```

// 与 partb.c 一起编译
#include <stdio.h>
void report_count();
void accumulate(int k);
int count = 0; // 文件作用域, 外部链接

int main(void)
{
 int value; // 自动变量
 register int i; // 寄存器变量

 printf("Enter a positive integer (0 to quit): ");
 while (scanf("%d", &value) == 1 && value > 0)
 {
 ++count; // 使用文件作用域变量
 for (i = value; i >= 0; i--)
 accumulate(i);
 printf("Enter a positive integer (0 to quit): ");
 }
 report_count();

 return 0;
}

void report_count()
{
 printf("Loop executed %d times\n", count);
}

```

---

程序清单 12.6 partb.c 程序

---

```

// partb.c -- 程序的其余部分
// 与 parta.c 一起编译
#include <stdio.h>

extern int count; // 引用式声明, 外部链接

static int total = 0; // 静态定义, 内部链接
void accumulate(int k); // 函数原型

void accumulate(int k) // k 具有块作用域, 无链接
{
 static int subtotal = 0; // 静态, 无链接

 if (k <= 0)
 {
 printf("loop cycle: %d\n", count);
 printf("subtotal: %d; total: %d\n", subtotal, total);
 subtotal = 0;
 }
 else
 {
 subtotal += k;
 }
}

```

```

 total += k;
}
}

```

在该程序中，块作用域的静态变量 subtotal 统计每次 while 循环传入 accumulate() 函数的总数，具有文件作用域、内部链接的变量 total 统计所有传入 accumulate() 函数的总数。当传入负值时，accumulate() 函数报告 total 和 subtotal 的值，并在报告后重置 subtotal 为 0。由于 parta.c 调用了 accumulate() 函数，所以必须包含 accumulate() 函数的原型。而 partb.c 只包含了 accumulate() 函数的定义，并未在文件中调用该函数，所以其原型为可选（即省略原型也不影响使用）。该函数使用了外部变量 count 统计 main() 中的 while 循环迭代的次数（顺带一提，对于该程序，没必要使用外部变量把 parta.c 和 partb.c 的代码弄得这么复杂）。在 parta.c 中，main() 和 report\_count() 共享 count。

下面是程序的运行示例：

```

Enter a positive integer (0 to quit): 5
loop cycle: 1
subtotal: 15; total: 15
Enter a positive integer (0 to quit): 10
loop cycle: 2
subtotal: 55; total: 70
Enter a positive integer (0 to quit): 2
loop cycle: 3
subtotal: 3; total: 73
Enter a positive integer (0 to quit): 0
Loop executed 3 times

```

## 12.1.11 存储类别和函数

函数也有存储类别，可以是外部函数（默认）或静态函数。C99 新增了第 3 种类别——内联函数，将在第 16 章中介绍。外部函数可以被其他文件的函数访问，但是静态函数只能用于其定义所在的文件。假设一个文件中包含了以下函数原型：

```

double gamma(double); /* 该函数默认为外部函数 */
static double beta(int, int);
extern double delta(double, int);

```

在同一个程序中，其他文件中的函数可以调用 gamma() 和 delta()，但是不能调用 beta()，因为以 static 存储类别说明符创建的函数属于特定模块私有。这样做避免了名称冲突的问题，由于 beta() 受限于它所在的文件，所以在其他文件中可以使用与之同名的函数。

通常的做法是：用 extern 关键字声明定义在其他文件中的函数。这样做是为了表明当前文件中使用的函数被定义在别处。除非使用 static 关键字，否则一般函数声明都默认为 extern。

## 12.1.12 存储类别的选择

对于“使用哪种存储类别”的回答绝大多数是“自动存储类别”，要知道默认类别就是自动存储类别。初学者会认为外部存储类别很不错，为何不把所有的变量都设置成外部变量，这样就不必使用参数和指针在函数间传递信息了。然而，这背后隐藏着一个陷阱。如果这样做，A() 函数可能违背你的意图，私下修改 B() 函数使用的变量。多年来，无数程序员的经验表明，随意使用外部存储类别的变量导致的后果远远超过了它所带来的便利。

唯一例外的是 const 数据。因为它们在初始化后就不会被修改，所以不用担心它们被意外篡改：

```
const int DAYS = 7;
const char * MSGS[3] = {"Yes", "No", "Maybe"};
```

保护性程序设计的黄金法则是：“按需知道”原则。尽量在函数内部解决该函数的任务，只共享那些需要共享的变量。除自动存储类别外，其他存储类别也很有用。不过，在使用某类别之前先要考虑一下是否有必要这样做。

## 12.2 随机数函数和静态变量

学习了不同存储类别的概念后，我们来看几个相关的程序。首先，来看一个使用内部链接的静态变量的函数：随机数函数。ANSI C 库提供了 `rand()` 函数生成随机数。生成随机数有多种算法，ANSI C 允许 C 实现针对特定机器使用最佳算法。然而，ANSI C 标准还提供了一个可移植的标准算法，在不同系统中生成相同的随机数。实际上，`rand()` 是“伪随机数生成器”，意思是可预测生成数字的实际序列。但是，数字在其取值范围内均匀分布。

为了看清楚程序内部的情况，我们使用可移植的 ANSI 版本，而不是编译器内置的 `rand()` 函数。可移植版本的方案开始于一个“种子”数字。该函数使用该种子生成新的数，这个新数又成为新的种子。然后，新种子可用于生成更新的种子，以此类推。该方案要行之有效，随机数函数必须记录它上一次被调用时所使用的种子。这里需要一个静态变量。程序清单 12.7 演示了版本 0（稍后给出版本 1）。

程序清单 12.7 `rand0.c` 函数文件

---

```
/* rand0.c --生成随机数*/
/* 使用 ANSI C 可移植算法 */
static unsigned long int next = 1; /* 种子 */

unsigned int rand0(void)
{
 /* 生成伪随机数的魔术公式 */
 next = next * 1103515245 + 12345;
 return (unsigned int) (next / 65536) % 32768;
}
```

---

在程序清单 12.7 中，静态变量 `next` 的初始值是 1，其值在每次调用 `rand0()` 函数时都会被修改（通过魔术公式）。该函数是用于返回一个 0~32767 之间的值。注意，`next` 是具有内部链接的静态变量（并非无链接）。这是为了方便稍后扩展本例，供同一个文件中的其他函数共享。

程序清单 12.8 是测试 `rand0()` 函数的一个简单的驱动程序。

程序清单 12.8 `r_drive0.c` 驱动程序

---

```
/* r_drive0.c -- 测试 rand0() 函数 */
/* 与 rand0.c 一起编译 */
#include <stdio.h>
extern unsigned int rand0(void);

int main(void)
{
 int count;

 for (count = 0; count < 5; count++)
 printf("%d\n", rand0());
```

```

 return 0;
}

```

该程序也需要多文件编译。程序清单 12.7 和程序清单 12.8 分别使用一个文件。程序清单 12.8 中的 `extern` 关键字提醒读者 `rand0()` 被定义在其他文件中，在这个文件中不要求写出该函数原型。输出如下：

```

16838
5758
10113
17515
31051

```

程序输出的数字看上去是随机的，再次运行程序后，输出如下：

```

16838
5758
10113
17515
31051

```

看来，这两次的输出完全相同，这体现了“伪随机”的一个方面。每次主程序运行，都开始于相同的种子 1。可以引入另一个函数 `srand1()` 重置种子来解决这个问题。关键是要让 `next` 成为只供 `rand1()` 和 `srand1()` 访问的内部链接静态变量（`srand1()` 相当于 C 库中的 `srand()` 函数）。把 `srand1()` 加入 `rand1()` 所在的文件中。程序清单 12.9 给出了修改后的文件。

#### 程序清单 12.9 s\_and\_r.c 文件程序

```

/* s_and_r.c -- 包含 rand1() 和 srand1() 的文件 */
/* 使用 ANSI C 可移植算法 */
static unsigned long int next = 1; /* 种子 */

int rand1(void)
{
 /*生成伪随机数的魔术公式*/
 next = next * 1103515245 + 12345;
 return (unsigned int) (next / 65536) % 32768;
}

void srand1(unsigned int seed)
{
 next = seed;
}

```

注意，`next` 是具有内部链接的文件作用域静态变量。这意味着 `rand1()` 和 `srand1()` 都可以使用它，但是其他文件中的函数无法访问它。使用程序清单 12.10 的驱动程序测试这两个函数。

#### 程序清单 12.10 r\_drive1.c 驱动程序

```

/* r_drive1.c -- 测试 rand1() 和 srand1() */
/* 与 s_and_r.c 一起编译 */
#include <stdio.h>
#include <stdlib.h>
extern void srand1(unsigned int x);
extern int rand1(void);

```

```

int main(void)
{
 int count;
 unsigned seed;

 printf("Please enter your choice for seed.\n");
 while (scanf("%u", &seed) == 1)
 {
 srand1(seed); /* 重置种子 */
 for (count = 0; count < 5; count++)
 printf("%d\n", rand1());
 printf("Please enter next seed (q to quit):\n");
 }
 printf("Done\n");
}

return 0;
}

```

编译两个文件，运行该程序后，其输出如下：

```

1
16838
5758
10113
17515
31051
Please enter next seed (q to quit):
513
20067
23475
8955
20841
15324
Please enter next seed (q to quit):
q
Done

```

设置 seed 的值为 1，输出的结果与前面程序相同。但是设置 seed 的值为 513 后就得到了新的结果。

### 注意 自动重置种子

如果 C 实现允许访问一些可变的量（如，时钟系统），可以用这些值（可能会被截断）初始化种子值。例如，ANSI C 有一个 `time()` 函数返回系统时间。虽然时间单元因系统而异，但是重点是该返回值是一个可进行运算的类型，而且其值随着时间变化而变化。`time()` 返回值的类型名是 `time_t`，具体类型与系统有关。这没关系，我们可以使用强制类型转换：

```
#include <time.h> /* 提供 time() 的 ANSI 原型 */
srand1((unsigned int) time(0)); /* 初始化种子 */
```

一般而言，`time()` 接受的参数是一个 `time_t` 类型对象的地址，而时间值就储存在传入的地址上。当然，也可以传入空指针 (0) 作为参数，这种情况下，只能通过返回值机制来提供值。

可以把这个技巧应用于标准的 ANSI C 函数 `srand()` 和 `rand()` 中。如果使用这些函数，要在文件中包含 `stdlib.c` 头文件。实际上，既然已经明白了 `srand1()` 和 `rand1()` 如何使用内部链接的静态变量，

你也可以使用编译器提供的版本。我们将在下一个示例中这样做。

## 12.3 掷骰子

我们将要模拟一个非常流行的游戏——掷骰子。骰子的形式多种多样，最普遍的是使用两个6面骰子。在一些冒险游戏中，会使用5种骰子：4面、6面、8面、12面和20面。聪明的古希腊人证明了只有5种正多面体，它们的所有面都具有相同的形状和大小。各种不同类型的骰子就是根据这些正多面体发展而来。也可以做成其他面数的，但是其所有的面不会都相等，因此各个面朝上的几率就不同。

计算机计算不用考虑几何的限制，所以可以设计任意面数的电子骰子。我们先从6面开始。

我们想获得1~6的随机数。然而，`rand()`生成的随机数在0~`RAND_MAX`之间。`RAND_MAX`被定义在`stdlib.h`中，其值通常是`INT_MAX`。因此，需要进行一些调整，方法如下。

1. 把随机数求模6，获得的整数在0~5之间。
2. 结果加1，新值在1~6之间。
3. 为方便以后扩展，把第1步中的数字6替换成骰子面数。

下面的代码实现了这3个步骤：

```
#include <stdlib.h> /* 提供 rand() 的原型 */
int rollem(int sides)
{
 int roll;

 roll = rand() % sides + 1;
 return roll;
}
```

我们还想用一个函数提示用户选择任意面数的骰子，并返回点数总和。如程序清单12.11所示。

**程序清单 12.11 diceroll.c 程序**

---

```
/* diceroll.c -- 掷骰子模拟程序 */
/* 与 mandydice.c 一起编译 */

#include "diceroll.h"
#include <stdio.h>
#include <stdlib.h> /* 提供库函数 rand() 的原型 */

int roll_count = 0; /* 外部链接 */

static int rollem(int sides) /* 该函数属于该文件私有 */
{
 int roll;

 roll = rand() % sides + 1;
 ++roll_count; /* 计算函数调用次数 */

 return roll;
}

int roll_n_dice(int dice, int sides)
{
 int d;
```

```

int total = 0;
if (sides < 2)
{
 printf("Need at least 2 sides.\n");
 return -2;
}
if (dice < 1)
{
 printf("Need at least 1 die.\n");
 return -1;
}

for (d = 0; d < dice; d++)
 total += rollem(sides);

return total;
}

```

该文件加入了新元素。第一，rollem() 函数属于该文件私有，它是 roll\_n\_dice() 的辅助函数。第二，为了演示外部链接的特性，该文件声明了一个外部变量 roll\_count。该变量统计调用 rollem() 函数的次数。这样设计有点蹩脚，仅为了演示外部变量的特性。第三，该文件包含以下预处理指令：

```
#include "diceroll.h"
```

如果使用标准库函数，如 rand()，要在当前文件中包含标准头文件（对 rand() 而言要包含 stdlib.h），而不是声明该函数。因为头文件中已经包含了正确的函数原型。我们效仿这一做法，把 roll\_n\_dice() 函数的原型放在 diceroll.h 头文件中。把文件名放在双引号中而不是尖括号中，指示编译器在本地查找文件，而不是到编译器存放标准头文件的位置去查找文件。“本地查找”的含义取决于具体的实现。一些常见的实现把头文件与源代码文件或工程文件（如果编译器使用它们的话）放在相同的目录或文件夹中。程序清单 12.12 是头文件中的内容。

程序清单 12.12 diceroll.h 文件

---

```
//diceroll.h
extern int roll_count;

int roll_n_dice(int dice, int sides);
```

---

该头文件中包含一个函数原型和一个 extern 声明。由于 diceroll.c 文件包含了该文件，diceroll.c 实际上包含了 roll\_count 的两个声明：

```
extern int roll_count; // 头文件中的声明（引用式声明）
int roll_count = 0; // 源代码文件中的声明（定义式声明）
```

这样做没问题。一个变量只能有一个定义式声明，但是带 extern 的声明是引用式声明，可以有多个引用式声明。

使用 roll\_n\_dice() 函数的程序都要包含 diceroll.c 头文件。包含该头文件后，程序便可使用 roll\_n\_dice() 函数和 roll\_count 变量。如程序清单 12.13 所示。

程序清单 12.13 manydice.c 文件

---

```
/* manydice.c -- 多次掷骰子的模拟程序 */
/* 与 diceroll.c 一起编译 */
#include <stdio.h>
#include <stdlib.h> /* 为库函数 srand() 提供原型 */
```

```

#include <time.h> /* 为 time() 提供原型 */
#include "diceroll.h" /* 为 roll_n_dice() 提供原型, 为 roll_count 变量提供声明 */

int main(void)
{
 int dice, roll;
 int sides;
 int status;

 srand((unsigned int) time(0)); /* 随机种子 */
 printf("Enter the number of sides per die, 0 to stop.\n");
 while (scanf("%d", &sides) == 1 && sides > 0)
 {
 printf("How many dice?\n");
 if ((status = scanf("%d", &dice)) != 1)
 {
 if (status == EOF)
 break; /* 退出循环 */
 else
 {
 printf("You should have entered an integer.");
 printf(" Let's begin again.\n");
 while (getchar() != '\n')
 continue; /* 处理错误的输入 */
 printf("How many sides? Enter 0 to stop.\n");
 continue; /* 进入循环的下一轮迭代 */
 }
 }
 roll = roll_n_dice(dice, sides);
 printf("You have rolled a %d using %d %d-sided dice.\n",
 roll, dice, sides);
 printf("How many sides? Enter 0 to stop.\n");
 }
 printf("The rollm() function was called %d times.\n",
 roll_count); /* 使用外部变量 */
 printf("GOOD FORTUNE TO YOU!\n");

 return 0;
}

```

要与包含程序清单 12.11 的文件一起编译该文件。可以把程序清单 12.11、12.12 和 12.13 都放在同一文件夹或目录中。运行该程序，下面是一个输出示例：

```

Enter the number of sides per die, 0 to stop.
6
How many dice?
2
You have rolled a 12 using 2 6-sided dice.
How many sides? Enter 0 to stop.
6
How many dice?
2
You have rolled a 4 using 2 6-sided dice.

```

```
How many sides? Enter 0 to stop.
6
How many dice?
2
You have rolled a 5 using 2 6-sided dice.
How many sides? Enter 0 to stop.
0
The rollm() function was called 6 times.
GOOD FORTUNE TO YOU!
```

因为该程序使用了 `srand()` 随机生成随机数种子，所以大多数情况下，即使输入相同也很难得到相同的输出。注意，`manydice.c` 中的 `main()` 访问了定义在 `diceroll.c` 中的 `roll_count` 变量。

有 3 种情况可以导致外层 `while` 循环结束：`side` 小于 1、输入类型不匹配（此时 `scanf()` 返回 0）、遇到文件结尾（返回值是 `EOF`）。为了读取骰子的点数，该程序处理文件结尾的方式（退出 `while` 循环）与处理类型不匹配（进入循环的下一轮迭代）的情况不同。

可以通过多种方式使用 `roll_n_dice()`。`sides` 等于 2 时，程序模仿掷硬币，“正面朝上”为 2，“反面朝上”为 1（或者反过来表示也行）。很容易修改该程序单独显示点数的结果，或者构建一个骰子模拟器。如果要掷多次骰子（如在一些角色扮演类游戏中），可以很容易地修改程序以输出类似的结果：

```
Enter the number of sets; enter q to stop.
18
How many sides and how many dice?
6 3
Here are 18 sets of 3 6-sided throws.
12 10 6 9 8 14 8 15 9 14 12 17 11 7 10
13 8 14
How many sets? Enter q to stop.
q
```

`rand1()` 或 `rand()`（不是 `rollm()`）还可以用来创建一个猜数字程序，让计算机选定一个数字，你来猜。读者感兴趣的话可以自己编写这个程序。

## 12.4 分配内存：`malloc()` 和 `free()`

我们前面讨论的存储类别有一个共同之处：在确定用哪种存储类别后，根据已制定好的内存管理规则，将自动选择其作用域和存储期。然而，还有更灵活地选择，即用库函数分配和管理内存。

首先，回顾一下内存分配。所有程序都必须预留足够的内存来储存程序使用的数据。这些内存中有些是自动分配的。例如，以下声明：

```
float x;
char place[] = "Dancing Oxen Creek";
```

为一个 `float` 类型的值和一个字符串预留了足够的内存，或者可以显式指定分配一定数量的内存：

```
int plates[100];
```

该声明预留了 100 个内存位置，每个位置都用于储存 `int` 类型的值。声明还为内存提供了一个标识符。因此，可以使用 `x` 或 `place` 识别数据。回忆一下，静态数据在程序载入内存时分配，而自动数据在程序执行块时分配，并在程序离开该块时销毁。

C 能做的不止这些。可以在程序运行时分配更多的内存。主要的工具是 `malloc()` 函数，该函数接受一个参数：所需的内存字节数。`malloc()` 函数会找到合适的空闲内存块，这样的内存是匿名的。也就是说，`malloc()` 分配内存，但是不会为其赋名。然而，它确实返回动态分配内存块的首字节地址。因此，可以把该地址赋给一个指针变量，并使用指针访问这块内存。因为 `char` 表示 1 字节，`malloc()` 的返回类型通常

被定义为指向 `char` 的指针。然而，从 ANSI C 标准开始，C 使用一个新的类型：指向 `void` 的指针。该类型相当于一个“通用指针”。`malloc()` 函数可用于返回指向数组的指针、指向结构的指针等，所以通常该函数的返回值会被强制转换为匹配的类型。在 ANSI C 中，应该坚持使用强制类型转换，提高代码的可读性。然而，把指向 `void` 的指针赋给任意类型的指针完全不用考虑类型匹配的问题。如果 `malloc()` 分配内存失败，将返回空指针。

我们试着用 `malloc()` 创建一个数组。除了用 `malloc()` 在程序运行时请求一块内存，还需要一个指针记录这块内存的位置。例如，考虑下面的代码：

```
double * ptd;
ptd = (double *) malloc(30 * sizeof(double));
```

以上代码为 30 个 `double` 类型的值请求内存空间，并设置 `ptd` 指向该位置。注意，指针 `ptd` 被声明为指向一个 `double` 类型，而不是指向内含 30 个 `double` 类型值的块。回忆一下，数组名是该数组首元素的地址。因此，如果让 `ptd` 指向这个块的首元素，便可像使用数组名一样使用它。也就是说，可以使用表达式 `ptd[0]` 访问该块的首元素，`ptd[1]` 访问第 2 个元素，以此类推。根据前面所学的知识，可以使用数组名来表示指针，也可以用指针来表示数组。

现在，我们有 3 种创建数组的方法。

- 声明数组时，用常量表达式表示数组的维度，用数组名访问数组的元素。可以用静态内存或自动内存创建这种数组。
- 声明变长数组（C99 新增的特性）时，用变量表达式表示数组的维度，用数组名访问数组的元素。具有这种特性的数组只能在自动内存中创建。
- 声明一个指针，调用 `malloc()`，将其返回值赋给指针，使用指针访问数组的元素。该指针可以是静态的或自动的。

使用第 2 种和第 3 种方法可以创建动态数组（*dynamic array*）。这种数组和普通数组不同，可以在程序运行时选择数组的大小和分配内存。例如，假设 `n` 是一个整型变量。在 C99 之前，不能这样做：

```
double item[n]; /* C99 之前：n 不允许是变量 */
```

但是，可以这样做：

```
ptd = (double *) malloc(n * sizeof(double)); /* 可以 */
```

如你所见，这比变长数组更灵活。

通常，`malloc()` 要与 `free()` 配套使用。`free()` 函数的参数是之前 `malloc()` 返回的地址，该函数释放之前 `malloc()` 分配的内存。因此，动态分配内存的存储期从调用 `malloc()` 分配内存到调用 `free()` 释放内存为止。设想 `malloc()` 和 `free()` 管理着一个内存池。每次调用 `malloc()` 分配内存给程序使用，每次调用 `free()` 把内存归还内存池中，这样便可重复使用这些内存。`free()` 的参数应该是一个指针，指向由 `malloc()` 分配的一块内存。不能用 `free()` 释放通过其他方式（如，声明一个数组）分配的内存。`malloc()` 和 `free()` 的原型都在 `stdlib.h` 头文件中。

使用 `malloc()`，程序可以在运行时才确定数组大小。如程序清单 12.14 所示，它把内存块的地址赋给指针 `ptd`，然后便可以使用数组名的方式使用 `ptd`。另外，如果内存分配失败，可以调用 `exit()` 函数结束程序，其原型在 `stdlib.h` 中。`EXIT_FAILURE` 的值也被定义在 `stdlib.h` 中。标准提供了两个返回值以保证在所有操作系统中都能正常工作：`EXIT_SUCCESS`（或者，相当于 0）表示普通的程序结束，`EXIT_FAILURE` 表示程序异常中止。一些操作系统（包括 UNIX、Linux 和 Windows）还接受一些表示其他运行错误的整数值。

## 程序清单 12.14 dyn\_arr.c 程序

```

/* dyn_arr.c -- 动态分配数组 */
#include <stdio.h>
#include <stdlib.h> /* 为 malloc()、free() 提供原型 */

int main(void)
{
 double * ptd;
 int max;
 int number;
 int i = 0;

 puts("What is the maximum number of type double entries?");
 if (scanf("%d", &max) != 1)
 {
 puts("Number not correctly entered -- bye.");
 exit(EXIT_FAILURE);
 }
 ptd = (double *) malloc(max * sizeof(double));
 if (ptd == NULL)
 {
 puts("Memory allocation failed. Goodbye.");
 exit(EXIT_FAILURE);
 }
 /* ptd 现在指向有 max 个元素的数组 */
 puts("Enter the values (q to quit):");
 while (i < max && scanf("%lf", &ptd[i]) == 1)
 ++i;
 printf("Here are your %d entries:\n", number = i);
 for (i = 0; i < number; i++)
 {
 printf("%7.2f ", ptd[i]);
 if (i % 7 == 6)
 putchar('\n');
 }
 if (i % 7 != 0)
 putchar('\n');
 puts("Done.");
 free(ptd);

 return 0;
}

```

下面是该程序的运行示例。程序通过交互的方式让用户先确定数组的大小，我们设置数组大小为 5。虽然我们后来输入了 6 个数，但程序也只处理前 5 个数。

What is the maximum number of entries?

5

Enter the values (q to quit):

20 30 35 25 40 80

Here are your 5 entries:

20.00 30.00 35.00 25.00 40.00

Done.

该程序通过以下代码获取数组的大小：

```

if (scanf("%d", &max) != 1)
{
 puts("Number not correctly entered -- bye.");
 exit(EXIT_FAILURE);
}

```

接下来，分配足够的内存空间以储存用户要存入的所有数，然后把动态分配的内存地址赋给指针 ptd：

```
ptd = (double *) malloc(max * sizeof(double));
```

在 C 中，不一定要使用强制类型转换 (double \*)，但是在 C++ 中必须使用。所以，使用强制类型转换更容易把 C 程序转换为 C++ 程序。

malloc() 可能分配不到所需的内存。在这种情况下，该函数返回空指针，程序结束：

```

if (ptd == NULL)
{
 puts("Memory allocation failed. Goodbye.");
 exit(EXIT_FAILURE);
}

```

如果程序成功分配内存，便可把 ptd 视为一个有 max 个元素的数组名。

注意，free() 函数位于程序的末尾，它释放了 malloc() 函数分配的内存。free() 函数只释放其参数指向的内存块。一些操作系统在程序结束时会自动释放动态分配的内存，但是有些系统不会。为保险起见，请使用 free()，不要依赖操作系统来清理。

使用动态数组有什么好处？从本例来看，使用动态数组给程序带来了更多灵活性。假设你已经知道，在大多数情况下程序所用的数组都不会超过 100 个元素，但是有时程序确实需要 10000 个元素。要是按照平时的做法，你不得不为这种情况声明一个内含 10000 个元素的数组。基本上这样做是在浪费内存。如果需要 10001 个元素，该程序就会出错。这种情况下，可以使用一个动态数组调整程序以适应不同的情况。

## 12.4.1 free() 的重要性

静态内存的数量在编译时是固定的，在程序运行期间也不会改变。自动变量使用的内存数量在程序执行期间自动增加或减少。但是动态分配的内存数量只会增加，除非用 free() 进行释放。例如，假设有一个创建数组临时副本的函数，其代码框架如下：

```

...
int main()
{
 double glad[2000];
 int i;
 ...
 for (i = 0; i < 1000; i++)
 gobble(glad, 2000);
 ...
}
void gobble(double ar[], int n)
{
 double * temp = (double *) malloc(n * sizeof(double));
 ... /* free(temp); // 假设忘记使用 free() */
}

```

第 1 次调用 gobble() 时，它创建了指针 temp，并调用 malloc() 分配了 16000 字节的内存（假设 double 为 8 字节）。假设如代码注释所示，遗漏了 free()。当函数结束时，作为自动变量的指针 temp 也会消失。但是它所指向的 16000 字节的内存却仍然存在。由于 temp 指针已被销毁，所以无法访问这块

内存，它也不能被重复使用，因为代码中没有调用 `free()` 释放这块内存。

第 2 次调用 `gobble()` 时，它又创建了指针 `temp`，并调用 `malloc()` 分配了 16000 字节的内存。第 1 次分配的 16000 字节内存已不可用，所以 `malloc()` 分配了另外一块 16000 字节的内存。当函数结束时，该内存块也无法被再访问和再使用。

循环要执行 1000 次，所以在循环结束时，内存池中有 1600 万字节被占用。实际上，也许在循环结束之前就已耗尽所有的内存。这类问题被称为内存泄漏 (*memory leak*)。在函数末尾处调用 `free()` 函数可避免这类问题发生。

## 12.4.2 `calloc()` 函数

分配内存还可以使用 `calloc()`，典型的用法如下：

```
long * newmem;
newmem = (long *)calloc(100, sizeof (long));
```

和 `malloc()` 类似，在 ANSI 之前，`calloc()` 也返回指向 `char` 的指针；在 ANSI 之后，返回指向 `void` 的指针。如果要储存不同的类型，应使用强制类型转换运算符。`calloc()` 函数接受两个无符号整数作为参数（ANSI 规定是 `size_t` 类型）。第 1 个参数是所需的存储单元数量，第 2 个参数是存储单元的大小（以字节为单位）。在该例中，`long` 为 4 字节，所以，前面的代码创建了 100 个 4 字节的存储单元，总共 400 字节。

用 `sizeof(long)` 而不是 4，提高了代码的可移植性。这样，在其他 `long` 不是 4 字节的系统中也能正常工作。

`calloc()` 函数还有一个特性：它把块中的所有位都设置为 0（注意，在某些硬件系统中，不是把所有位都设置为 0 来表示浮点值 0）。

`free()` 函数也可用于释放 `calloc()` 分配的内存。

动态内存分配是许多高级程序设计技巧的关键。我们将在第 17 章中详细讲解。有些编译器可能还提供其他内存管理函数，有些可以移植，有些不可以。读者可以抽时间看一下。

## 12.4.3 动态内存分配和变长数组

变长数组（VLA）和调用 `malloc()` 在功能上有些重合。例如，两者都可用于创建在运行时确定大小的数组：

```
int vlamal()
{
 int n;
 int * pi;
 scanf("%d", &n);
 pi = (int *) malloc (n * sizeof(int));
 int ar[n];// 变长数组
 pi[2] = ar[2] = -5;
 ...
}
```

不同的是，变长数组是自动存储类型。因此，程序在离开变长数组定义所在的块时（该例中，即 `vlamal()` 函数结束时），变长数组占用的内存空间会被自动释放，不必使用 `free()`。另一方面，用 `malloc()` 创建的数组不必局限在一个函数内访问。例如，可以这样做：被调函数创建一个数组并返回指针，供主调函数访问，然后主调函数在末尾调用 `free()` 释放之前被调函数分配的内存。另外，`free()`

所用的指针变量可以与 `malloc()` 的指针变量不同，但是两个指针必须储存相同的地址。但是，不能释放同一块内存两次。

对多维数组而言，使用变长数组更方便。当然，也可以用 `malloc()` 创建二维数组，但是语法比较繁琐。如果编译器不支持变长数组特性，就只能固定二维数组的维度，如下所示：

```
int n = 5;
int m = 6;
int ar2[n][m]; // n×m 的变长数组 (VLA)
int (* p2)[6]; // C99 之前的写法
int (* p3)[m]; // 要求支持变长数组
p2 = (int (*)[6]) malloc(n * 6 * sizeof(int)); // n×6 数组
p3 = (int (*)[m]) malloc(n * m * sizeof(int)); // n×m 数组 (要求支持变长数组)
ar2[1][2] = p2[1][2] = 12;
```

先复习一下指针声明。由于 `malloc()` 函数返回一个指针，所以 `p2` 必须是一个指向合适类型的指针。  
第 1 个指针声明：

```
int (* p2)[6]; // C99 之前的写法
```

表明 `p2` 指向一个内含 6 个 `int` 类型值的数组。因此，`p2[i]` 代表一个由 6 个整数构成的元素，`p2[i][j]` 代表一个整数。

第 2 个指针声明用一个变量指定 `p3` 所指向数组的大小。因此，`p3` 代表一个指向变长数组的指针，这行代码不能在 C90 标准中运行。

#### 12.4.4 存储类别和动态内存分配

存储类别和动态内存分配有何联系？我们来看一个理想化模型。可以认为程序把它可用的内存分为 3 部分：一部分供具有外部链接、内部链接和无链接的静态变量使用；一部分供自动变量使用；一部分供动态内存分配。

静态存储类别的变量所用的内存数量在编译时确定，只要程序还在运行，就可访问储存在该部分的数据。该类别的变量在程序开始执行时被创建，在程序结束时被销毁。

然而，自动存储类别的变量在程序进入变量定义所在块时存在，在程序离开块时消失。因此，随着程序调用函数和函数结束，自动变量所用的内存数量也相应地增加和减少。这部分的内存通常作为栈来处理，这意味着新创建的变量按顺序加入内存，然后以相反的顺序销毁。

动态分配的内存调用 `malloc()` 或相关函数时存在，在调用 `free()` 后释放。这部分的内存由程序员管理，而不是一套规则。所以内存块可以在一个函数中创建，在另一个函数中销毁。正是因为这样，这部分的内存用于动态内存分配会支离破碎。也就是说，未使用的内存块分散在已使用的内存块之间。另外，使用动态内存通常比使用栈内存慢。

总而言之，程序把静态对象、自动对象和动态分配的对象储存在不同的区域。

##### 程序清单 12.15 where.c 程序

---

```
// where.c -- 数据被储存在何处？

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

int static_store = 30;
const char * pcg = "String Literal";
```

```

int main()
{
 int static_store = 30;
 char static_string [] = "Static String";
 int * pi;
 char * pcl;

 pi = (int *) malloc(sizeof(int));
 *pi = 35;
 pcl = (char *) malloc(strlen("Dynamic String") + 1);
 strcpy(pcl, "Dynamic String");

 printf("static_store: %d at %p\n", static_store, &static_store);
 printf(" auto_store: %d at %p\n", auto_store, &auto_store);
 printf(" *pi: %d at %p\n", *pi, pi);
 printf(" %s at %p\n", static_string, static_string);
 printf(" %s at %p\n", pcl, pcl);
 printf(" %s at %p\n", "Quoted String", "Quoted String");
 free(pi);
 free(pcl);

 return 0;
}

```

在我们的系统中，该程序的输入如下：

```

static_store: 30 at 00378000
auto_store: 40 at 0049FB8C
*pi: 35 at 008E9BA0
String Literal at 00375858
Auto char Array at 0049FB74
Dynamic String at 008E9BD0
Quoted String at 00375908

```

如上所示，静态数据（包括字符串字面量）占用一个区域，自动数据占用另一个区域，动态分配的数据占用第 3 个区域（通常被称为内存堆或自由内存）。

## 12.5 ANSI C 类型限定符

我们通常用类型和存储类别来描述一个变量。C90 还新增了两个属性：恒常性 (*constancy*) 和易变性 (*volatility*)。这两个属性可以分别用关键字 `const` 和 `volatile` 来声明，以这两个关键字创建的类型是限定类型 (*qualified type*)。C99 标准新增了第 3 个限定符：`restrict`，用于提高编译器优化。C11 标准新增了第 4 个限定符：`_Atomic`。C11 提供一个可选库，由 `stdatomic.h` 管理，以支持并发程序设计，而且 `_Atomic` 是可选支持项。

C99 为类型限定符增加了一个新属性：它们现在是幂等的 (*idempotent*)！这个属性听起来很强大，其实意思是在一条声明中多次使用同一个限定符，多余的限定符将被忽略：

```
const const const int n = 6; // 与 const int n = 6; 相同
```

有了这个新属性，就可以编写类似下面的代码：

```
typedef const int zip;
const zip q = 8;
```

## 12.5.1 const 类型限定符

第4章和第10章中介绍过 `const`。以 `const` 关键字声明的对象，其值不能通过赋值或递增、递减来修改。在 ANSI 兼容的编译器中，以下代码：

```
const int nochange; /* 限定 nochange 的值不能被修改 */
nchange = 12; /* 不允许 */
```

编译器会报错。但是，可以初始化 `const` 变量。因此，下面的代码没问题：

```
const int nochange = 12; /* 没问题 */
```

该声明让 `nchange` 成为只读变量。初始化后，就不能再改变它的值。

可以用 `const` 关键字创建不允许修改的数组：

```
const int days1[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

### 1. 在指针和形参声明中使用 `const`

声明普通变量和数组时使用 `const` 关键字很简单。指针则复杂一些，因为要区分是限定指针本身为 `const` 还是限定指针指向的值为 `const`。下面的声明：

```
const float * pf; /* pf 指向一个 float 类型的 const 值 */
```

创建了 `pf` 指向的值不能被改变，而 `pt` 本身的值可以改变。例如，可以设置该指针指向其他 `const` 值。相比之下，下面的声明：

```
float * const pt; /* pt 是一个 const 指针 */
```

创建的指针 `pt` 本身的值不能更改。`pt` 必须指向同一个地址，但是它所指向的值可以改变。下面的声明：

```
const float * const ptr;
```

表明 `ptr` 既不能指向别处，它所指向的值也不能改变。

还可以把 `const` 放在第 3 个位置：

```
float const * pfc; // 与 const float * pfc; 相同
```

如注释所示，把 `const` 放在类型名之后、`*`之前，说明该指针不能用于改变它所指向的值。简而言之，`const` 放在`*`左侧任意位置，限定了指针指向的数据不能改变；`const` 放在`*`的右侧，限定了指针本身不能改变。

`const` 关键字的常见用法是声明为函数形参的指针。例如，假设有一个函数要调用 `display()` 显示一个数组的内容。要把数组名作为实际参数传递给该函数，但是数组名是一个地址。该函数可能会更改主调函数中的数据，但是下面的原型保证了数据不会被更改：

```
void display(const int array[], int limit);
```

在函数原型和函数头，形参声明 `const int array[]` 与 `const int * array` 相同，所以该声明表明不能更改 `array` 指向的数据。

ANSI C 库遵循这种做法。如果一个指针仅用于给函数访问值，应将其声明为一个指向 `const` 限定类型的指针。如果要用指针更改主调函数中的数据，就不使用 `const` 关键字。例如，ANSI C 中的 `strcat()` 原型如下：

```
char *strcat(char * restrict s1, const char * restrict s2);
```

回忆一下，`strcat()` 函数在第 1 个字符串的末尾添加第 2 个字符串的副本。这更改了第 1 个字符串，但是未更改第 1 个字符串。上面的声明体现了这一点。

## 2. 对全局数据使用 const

前面讲过，使用全局变量是一种冒险的方法，因为这样做暴露了数据，程序的任何部分都能更改数据。如果把数据设置为 `const`，就可避免这样的危险，因此用 `const` 限定符声明全局数据很合理。可以创建 `const` 变量、`const` 数组和 `const` 结构（结构是一种复合数据类型，将在下一章介绍）。

然而，在文件间共享 `const` 数据要小心。可以采用两个策略。第一，遵循外部变量的常用规则，即在一个文件中使用定义式声明，在其他文件中使用引用式声明（用 `extern` 关键字）：

```
/* file1.c -- 定义了一些外部 const 变量 */
const double PI = 3.14159;
const char * MONTHS[12] = { "January", "February", "March", "April", "May",
 "June", "July", "August", "September", "October",
 "November", "December" };
```

```
/* file2.c -- 使用定义在别处的外部 const 变量 */
extern const double PI;
extern const * MONTHS [];
```

另一种方案是，把 `const` 变量放在一个头文件中，然后在其他文件中包含该头文件：

```
/* constant.h -- 定义了一些外部 const 变量 */
static const double PI = 3.14159;
static const char * MONTHS[12] = {"January", "February", "March", "April", "May",
 "June", "July", "August", "September", "October",
 "November", "December"};
```

```
/* file1.c -- 使用定义在别处的外部 const 变量 */
#include "constant.h"
```

```
/* file2.c -- 使用定义在别处的外部 const 变量 */
#include "constant.h"
```

这种方案必须在头文件中用关键字 `static` 声明全局 `const` 变量。如果去掉 `static`，那么在 `file1.c` 和 `file2.c` 中包含 `constant.h` 将导致每个文件中都有一个相同标识符的定义式声明，C 标准不允许这样做（然而，有些编译器允许）。实际上，这种方案相当于给每个文件提供了一个单独的数据副本<sup>1</sup>。由于每个副本只对该文件可见，所以无法用这些数据和其他文件通信。不过没关系，它们都是完全相同（每个文件都包含相同的头文件）的 `const` 数据（声明时使用了 `const` 关键字），这不是问题。

头文件方案的好处是，方便你偷懒，不用惦记着在一个文件中使用定义式声明，在其他文件中使用引用式声明。所有的文件都只需包含同一个头文件即可。但它的缺点是，数据是重复的。对于前面的例子而言，这不算什么问题，但是如果 `const` 数据包含庞大的数组，就不能视而不见了。

### 12.5.2 volatile 类型限定符

`volatile` 限定符告知计算机，代理（而不是变量所在的程序）可以改变该变量的值。通常，它被用于硬件地址以及在其他程序或同时运行的线程中共享数据。例如，一个地址上可能储存着当前的时钟时间，无论程序做什么，地址上的值都随时间的变化而改变。或者一个地址用于接受另一台计算机传入的信息。

`volatile` 的语法和 `const` 一样：

```
volatile int loc1; /* loc1 是一个易变的位置 */
volatile int * ploc; /* ploc 是一个指向易变的位置的指针 */
```

<sup>1</sup> 注意，以 `static` 声明的文件作用域变量具有内部链接属性。——译者注

以上代码把 loc1 声明为 volatile 变量，把 ploc 声明为指向 volatile 变量的指针。

读者可能认为 volatile 是个可有可无的概念，为何 ANSI 委员会把 volatile 关键字放入标准？原因是它涉及编译器的优化。例如，假设有下面的代码：

```
val1 = x;
/* 一些不使用 x 的代码 */
val2 = x;
```

智能的（进行优化的）编译器会注意到以上代码使用了两次 x，但并未改变它的值。于是编译器把 x 的值临时储存在寄存器中，然后在 val2 需要使用 x 时，才从寄存器中（而不是从原始内存位置上）读取 x 的值，以节约时间。这个过程被称为高速缓存（caching）。通常，高速缓存是个不错的优化方案，但如果一些其他代理在以上两条语句之间改变了 x 的值，就不能这样优化了。如果没有 volatile 关键字，编译器就不知道这种事情是否会发生。因此，为安全起见，编译器不会进行高速缓存。这是在 ANSI 之前的情况。现在，如果声明中没有 volatile 关键字，编译器会假定变量的值在使用过程中不变，然后再尝试优化代码。

可以同时用 const 和 volatile 限定一个值。例如，通常用 const 把硬件时钟设置为程序不能更改的变量，但是可以通过代理改变，这时用 volatile。只能在声明中同时使用这两个限定符，它们的顺序不重要，如下所示：

```
volatile const int loc;
const volatile int * ploc;
```

### 12.5.3 restrict 类型限定符

restrict 关键字允许编译器优化某部分代码以更好地支持计算。它只能用于指针，表明该指针是访问数据对象的唯一且初始的方式。要弄明白为什么这样做有用，先看几个例子。考虑下面的代码：

```
int ar[10];
int * restrict restar = (int *) malloc(10 * sizeof(int));
int * par = ar;
```

这里，指针 restar 是访问由 malloc() 所分配内存的唯一且初始的方式。因此，可以用 restrict 关键字限定它。而指针 par 既不是访问 ar 数组中数据的初始方式，也不是唯一方式。所以不用把它设置为 restrict。

现在考虑下面稍复杂的例子，其中 n 是 int 类型：

```
for (n = 0; n < 10; n++)
{
 par[n] += 5;
 restar[n] += 5;
 ar[n] *= 2;
 par[n] += 3;
 restar[n] += 3;
}
```

由于之前声明了 restar 是访问它所指向的数据块的唯一且初始的方式，编译器可以把涉及 restar 的两条语句替换成下面这条语句，效果相同：

```
restar[n] += 8; /* 可以进行替换 */
```

但是，如果把与 par 相关的两条语句替换成下面的语句，将导致计算错误：

```
par[n] += 8; /* 给出错误的结果 */
```

这是因为 for 循环在 par 两次访问相同的数据之间，用 ar 改变了该数据的值。

在本例中，如果未使用 `restrict` 关键字，编译器就必须假设最坏的情况（即，在两次使用指针之间，其他的标识符可能已经改变了数据）。如果用了 `restrict` 关键字，编译器就可以选择捷径优化计算。

`restrict` 限定符还可用于函数形参中的指针。这意味着编译器可以假定在函数体内其他标识符不会修改该指针指向的数据，而且编译器可以尝试对其优化，使其不做别的用途。例如，C 库有两个函数用于把一个位置上的字节拷贝到另一个位置。在 C99 中，这两个函数的原型是：

```
void * memcpy(void * restrict s1, const void * restrict s2, size_t n);
void * memmove(void * s1, const void * s2, size_t n);
```

这两个函数都从位置 `s2` 把 `n` 字节拷贝到位置 `s1`。`memcpy()` 函数要求两个位置不重叠，但是 `memmove()` 没有这样的要求。声明 `s1` 和 `s2` 为 `restrict` 说明这两个指针都是访问相应数据的唯一方式，所以它们不能访问相同块的数据。这满足了 `memcpy()` 无重叠的要求。`memmove()` 函数允许重叠，它在拷贝数据时不得不更小心，以防在使用数据之前就先覆盖了数据。

`restrict` 关键字有两个读者。一个是编译器，该关键字告知编译器可以自由假定一些优化方案。另一个读者是用户，该关键字告知用户要使用满足 `restrict` 要求的参数。总而言之，编译器不会检查用户是否遵循这一限制，但是无视它后果自负。

#### 12.5.4 `_Atomic` 类型限定符 (C11)

并发程序设计把程序执行分成可以同时执行的多个线程。这给程序设计带来了新的挑战，包括如何管理访问相同数据的不同线程。C11 通过包含可选的头文件 `stdatomic.h` 和 `threads.h`，提供了一些可选的（不是必须实现的）管理方法。值得注意的是，要通过各种宏函数来访问原子类型。当一个线程对一个原子类型的对象执行原子操作时，其他线程不能访问该对象。例如，下面的代码：

```
int hogs; // 普通声明
hogs = 12; // 普通赋值
```

可以替换成：

```
_Atomic int hogs; // hogs 是一个原子类型的变量
atomic_store(&hogs, 12); // stdatomic.h 中的宏
```

这里，在 `hogs` 中储存 12 是一个原子过程，其他线程不能访问 `hogs`。

编写这种代码的前提是，编译器要支持这一新特性。

#### 12.5.5 旧关键字的新位置

C99 允许把类型限定符和存储类别说明符 `static` 放在函数原型和函数头的形式参数的初始方括号中。对于类型限定符而言，这样做为现有功能提供了一个替代的语法。例如，下面是旧式语法的声明：

```
void ofmouth(int * const a1, int * restrict a2, int n); // 以前的风格
```

该声明表明 `a1` 是一个指向 `int` 的 `const` 指针，这意味着不能更改指针本身，可以更改指针指向的数据。除此之外，还表明 `a2` 是一个 `restrict` 指针，如上一节所述。新的等价语法如下：

```
void ofmouth(int a1[const], int a2[restrict], int n); // C99 允许
```

根据新标准，在声明函数形参时，指针表示法和数组表示法都可以使用这两个限定符。

`static` 的情况不同，因为新标准为 `static` 引入了一种与以前用法不相关的新用法。现在，`static` 除了表明静态存储类别变量的作用域或链接外，新的用法告知编译器如何使用形式参数。例如，考虑下面的原型：

```
double stick(double ar[static 20]);
```

`static` 的这种用法表明，函数调用中的实际参数应该是一个指向数组首元素的指针，且该数组至少

有 20 个元素。这种用法的目的是让编译器使用这些信息优化函数的编码。为何给 `static` 新增一个完全不同的用法？C 标准委员会不愿意创建新的关键字，因为这样会让以前用新关键字作为标识符的程序无效。所以，他们会尽量利用现有的关键字，尽量不添加新的关键字。

`restrict` 关键字有两个读者。一个是编译器，该关键字告知编译器可以自由假定一些优化方案。另一个读者是用户，该关键字告知用户要使用满足 `restrict` 要求的参数。

## 12.6 关键概念

C 提供多种管理内存的模型。除了熟悉这些模型外，还要学会如何选择不同的类别。大多数情况下，最好选择自动变量。如果要使用其他类别，应该有充分的理由。通常，使用自动变量、函数形参和返回值进行函数间的通信比使用全局变量安全。但是，保持不变的数据适合用全局变量。

应该尽量理解静态内存、自动内存和动态分配内存的属性。尤其要注意：静态内存的数量在编译时确定；静态数据在载入程序时被载入内存。在程序运行时，自动变量被分配或释放，所以自动变量占用的内存数量随着程序的运行会不断变化。可以把自动内存看作是可重复利用的工作区。动态分配的内存也会增加和减少，但是这个过程由函数调用控制，不是自动进行的。

## 12.7 本章小结

内存用于存储程序中的数据，由存储期、作用域和链接表征。存储期可以是静态的、自动的或动态分配的。如果是静态存储期，在程序开始执行时分配内存，并在程序运行时都存在。如果是自动存储期，在程序进入变量定义所在块时分配变量的内存，在程序离开块时释放内存。如果是动态分配存储期，在调用 `malloc()`（或相关函数）时分配内存，在调用 `free()` 函数时释放内存。

作用域决定程序的哪些部分可以访问某数据。定义在所有函数之外的变量具有文件作用域，对位于该变量声明之后的所有函数可见。定义在块或作为函数形参内的变量具有块作用域，只对该块以及它包含的嵌套块可见。

链接描述定义在程序某翻译单元中的变量可被链接的程度。具有块作用域的变量是局部变量，无链接。具有文件作用域的变量可以是内部链接或外部链接。内部链接意味着只有其定义所在的文件才能使用该变量。外部链接意味着其他文件使用也可以使用该变量。

下面是 C 的 5 种存储类别（不包括线程的概念）。

- **自动**—在块中不带存储类别说明符或带 `auto` 存储类别说明符声明的变量（或作为函数头中的形参）属于自动存储类别，具有自动存储期、块作用域、无链接。如果未初始化自动变量，它的值是未定义的。
- **寄存器**—在块中带 `register` 存储类别说明符声明的变量（或作为函数头中的形参）属于寄存器存储类别，具有自动存储期、块作用域、无链接，且无法获取其地址。把一个变量声明为寄存器变量即请求编译器将其储存到访问速度最快的区域。如果未初始化寄存器变量，它的值是未定义的。
- **静态、无链接**—在块中带 `static` 存储类别说明符声明的变量属于“静态、无链接”存储类别，具有静态存储期、块作用域、无链接。只在编译时被初始化一次。如果未显式初始化，它的字节都被设置为 0。
- **静态、外部链接**—在所有函数外部且没有使用 `static` 存储类别说明符声明的变量属于“静态、外部链接”存储类别，具有静态存储期、文件作用域、外部链接。只能在编译器被初始化一次。如果未显式初始化，它的字节都被设置为 0。
- **静态、内部链接**—在所有函数外部且使用了 `static` 存储类别说明符声明的变量属于“静态、内部链接”存储类别，具有静态存储期、文件作用域、内部链接。只能在编译器被初始化一次。如果未显式初始化，它的字节都被设置为 0。

动态分配的内存由 `malloc()` (或相关) 函数分配, 该函数返回一个指向指定字节数内存块的指针。这块内存被 `free()` 函数释放后便可重复使用, `free()` 函数以该内存块的地址作为参数。

类型限定符 `const`、`volatile`、`restrict` 和 `_Atomic`。`const` 限定符限定数据在程序运行时不能改变。对指针使用 `const` 时, 可限定指针本身不能改变或指针指向的数据不能改变, 这取决于 `const` 在指针声明中的位置。`volatile` 限定符表明, 限定的数据除了被当前程序修改外还可以被其他进程修改。该限定符的目的是警告编译器不要进行假定的优化。`restrict` 限定符也是为了方便编译器设置优化方案。`restrict` 限定的指针是访问它所指向数据的唯一途径。

## 12.8 复习题

复习题的参考答案在附录 A 中。

1. 哪些类别的变量可以成为它所在函数的局部变量?
2. 哪些类别的变量在它所在程序的运行期一直存在?
3. 哪些类别的变量可以被多个文件使用? 哪些类别的变量仅限于在一个文件中使用?
4. 块作用域变量具有什么链接属性?
5. `extern` 关键字有什么用途?
6. 考虑下面两行代码, 就输出的结果而言有何异同:

```
int * p1 = (int *)malloc(100 * sizeof(int));
int * p1 = (int *)calloc(100, sizeof(int));
```

7. 下面的变量对哪些函数可见? 程序是否有误?

```
/* 文件 1 */
int daisy;
int main(void)
{
 int lily;
 ...
}
int petal()
{
 extern int daisy, lily;
 ...
}
/* 文件 2 */
extern int daisy;
static int lily;
int rose;
int stem()
{
 int rose;
 ...
}
void root()
{
 ...
}
```

8. 下面程序会打印什么?

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

```

char color = 'B';
void first(void);
void second(void);

int main(void)
{
 extern char color;

 printf("color in main() is %c\n", color);
 first();
 printf("color in main() is %c\n", color);
 second();
 printf("color in main() is %c\n", color);
 return 0;
}

void first(void)
{
 char color;

 color = 'R';
 printf("color in first() is %c\n", color);
}

void second(void)
{
 color = 'G';
 printf("color in second() is %c\n", color);
}

```

9. 假设文件的开始处有如下声明：

```

static int plink;
int value_ct(const int arr[], int value, int n);

```

- 以上声明表明了程序员的什么意图？
- 用 const int value 和 const int n 分别替换 int value 和 int n，是否对主调程序的值加强保护。

## 12.9 编程练习

- 不使用全局变量，重写程序清单 12.4。
- 在美国，通常以英里/加仑来计算油耗；在欧洲，以升/100 公里来计算。下面是程序的一部分，提示用户选择计算模式（美制或公制），然后接收数据并计算油耗。

```

// pe12-2b.c
// 与 pe12-2a.c 一起编译
#include <stdio.h>
#include "pe12-2a.h"
int main(void)
{
 int mode;

 printf("Enter 0 for metric mode, 1 for US mode: ");
 scanf("%d", &mode);
 while (mode >= 0)

```

```

{
 set_mode(mode);
 get_info();
 show_info();
 printf("Enter 0 for metric mode, 1 for US mode");
 printf(" (-1 to quit): ");
 scanf("%d", &mode);
}
printf("Done.\n");
return 0;
}

```

下面是是一些输出示例：

```

Enter 0 for metric mode, 1 for US mode: 0
Enter distance traveled in kilometers: 600
Enter fuel consumed in liters: 78.8
Fuel consumption is 13.13 liters per 100 km.
Enter 0 for metric mode, 1 for US mode (-1 to quit): 1
Enter distance traveled in miles: 434
Enter fuel consumed in gallons: 12.7
Fuel consumption is 34.2 miles per gallon.
Enter 0 for metric mode, 1 for US mode (-1 to quit): 3
Invalid mode specified. Mode 1(US) used.
Enter distance traveled in miles: 388
Enter fuel consumed in gallons: 15.3
Fuel consumption is 25.4 miles per gallon.
Enter 0 for metric mode, 1 for US mode (-1 to quit): -1
Done.

```

如果用户输入了不正确的模式，程序向用户给出提示消息并使用上一次输入的正确模式。请提供 pe12-2a.h 头文件和 pe12-2a.c 源文件。源代码文件应定义 3 个具有文件作用域、内部链接的变量。一个表示模式、一个表示距离、一个表示消耗的燃料。get\_info() 函数根据用户输入的模式提示用户输入相应数据，并将其储存到文件作用域变量中。show\_info() 函数根据设置的模式计算并显示油耗。可以假设用户输入的都是数值数据。

3. 重新设计编程练习 2，要求只使用自动变量。该程序提供的用户界面不变，即提示用户输入模式等。但是，函数调用要作相应变化。
4. 在一个循环中编写并测试一个函数，该函数返回它被调用的次数。
5. 编写一个程序，生成 100 个 1~10 范围内的随机数，并以降序排列（可以把第 11 章的排序算法稍加改动，便可用于整数排序，这里仅对整数排序）。
6. 编写一个程序，生成 1000 个 1~10 范围内的随机数。不用保存或打印这些数字，仅打印每个数出现的次数。用 10 个不同的种子值运行，生成的数字出现的次数是否相同？可以使用本章自定义的函数或 ANSI C 的 rand() 和 srand() 函数，它们的格式相同。这是一个测试特定随机数生成器随机性的方法。
7. 编写一个程序，按照程序清单 12.13 输出示例后面讨论的内容，修改该程序。使其输出类似：

```

Enter the number of sets; enter q to stop : 18
How many sides and how many dice? 6 3
Here are 18 sets of 3 6-sided throws.
12 10 6 9 8 14 8 15 9 14 12 17 11 7 10
13 8 14
How many sets? Enter q to stop: q

```

8. 下面是程序的一部分：

```
// pe12-8.c
#include <stdio.h>
int * make_array(int elem, int val);
void show_array(const int ar [], int n);
int main(void)
{
 int * pa;
 int size;
 int value;

 printf("Enter the number of elements: ");
 while (scanf("%d", &size) == 1 && size > 0)
 {
 printf("Enter the initialization value: ");
 scanf("%d", &value);
 pa = make_array(size, value);
 if (pa)
 {
 show_array(pa, size);
 free(pa);
 }
 printf("Enter the number of elements (<1 to quit): ");
 }
 printf("Done.\n");
 return 0;
}
```

提供 `make_array()` 和 `show_array()` 函数的定义，完成该程序。`make_array()` 函数接受两个参数，第 1 个参数是 `int` 类型数组的元素个数，第 2 个参数是要赋给每个元素的值。该函数调用 `malloc()` 创建一个大小合适的数组，将其每个元素设置为指定的值，并返回一个指向该数组的指针。`show_array()` 函数显示数组的内容，一行显示 8 个数。

9. 编写一个符合以下描述的函数。首先，询问用户需要输入多少个单词。然后，接收用户输入的单词，并显示出来，使用 `malloc()` 并回答第 1 个问题（即要输入多少个单词），创建一个动态数组，该数组内含相应的指向 `char` 的指针（注意，由于数组的每个元素都是指向 `char` 的指针，所以用于储存 `malloc()` 返回值的指针应该是一个指向指针的指针，且它所指向的指针指向 `char`）。在读取字符串时，该程序应该把单词读入一个临时的 `char` 数组，使用 `malloc()` 分配足够的存储空间来储存单词，并把地址存入该指针数组（该数组中每个元素都是指向 `char` 的指针）。然后，从临时数组中把单词拷贝到动态分配的存储空间中。因此，有一个字符指针数组，每个指针都指向一个对象，该对象的大小正好能容纳被储存的特定单词。下面是该程序的一个运行示例：

```
How many words do you wish to enter? 5
Enter 5 words now:
I enjoyed doing this exercise
Here are your words:
I
enjoyed
doing
this
exercise
```



# 第13章

## 文件输入/输出

本章介绍以下内容：

- 函数：`fopen()`、`getc()`、`putc()`、`exit()`、`fclose()`  
`fprintf()`、`fscanf()`、`fgets()`、`fputs()`  
`rewind()`、`fseek()`、`ftell()`、`fflush()`  
`fgetpos()`、`fsetpos()`、`feof()`、`ferror()`  
`ungetc()`、`setvbuf()`、`fread()`、`fwrite()`
- 如何使用 C 标准 I/O 系列的函数处理文件
- 文件模式和二进制模式、文本和二进制格式、缓冲和无缓冲 I/O
- 使用既可以顺序访问文件也可以随机访问文件的函数

文件是当今计算机系统不可或缺的部分。文件用于储存程序、文档、数据、书信、表格、图形、照片、视频和许多其他种类的信息。作为程序员，必须会编写创建文件和从文件读写数据的程序。本章将介绍相关的内容。

### 13.1 与文件进行通信

有时，需要程序从文件中读取信息或把信息写入文件。这种程序与文件交互的形式就是文件重定向（第 8 章介绍过）。这种方法很简单，但是有一定限制。例如，假设要编写一个交互程序，询问用户书名并把完整的书名列表保存在文件中。如果使用重定向，应该类似于：

```
books > bklist
```

用户的输入被重定向到 `bklist` 中。这样做不仅会把不符合要求的文本写入 `bklist`，而且用户也看不到要回答什么问题。

C 提供了更强大的文件通信方法，可以在程序中打开文件，然后使用特殊的 I/O 函数读取文件中的信息或把信息写入文件。在研究这些方法之前，先简要介绍一下文件的性质。

#### 13.1.1 文件是什么

文件 (*file*) 通常是在磁盘或固态硬盘上的一段已命名的存储区。对我们而言，`stdio.h` 就是一个文件的名称，该文件中包含一些有用的信息。然而，对操作系统而言，文件更复杂一些。例如，大型文件会被分开储存，或者包含一些额外的数据，方便操作系统确定文件的种类。然而，这都是操作系统所关心的，程序员关心的是 C 程序如何处理文件（除非你正在编写操作系统）。

C 把文件看作是一系列连续的字节，每个字节都能被单独读取。这与 UNIX 环境中（C 的发源地）的文件结构相对应。由于其他环境中可能无法完全对应这个模型，C 提供两种文件模式：文本模式和二进制模式。

#### 13.1.2 文本模式和二进制模式

首先，要区分文本内容和二进制内容、文本文件格式和二进制文件格式，以及文件的文本模式和二进

制模式。

所有文件的内容都以二进制形式（0 或 1）储存。但是，如果文件最初使用二进制编码的字符（例如，ASCII 或 Unicode）表示文本（就像 C 字符串那样），该文件就是文本文件，其中包含文本内容。如果文件中的二进制值代表机器语言代码或数值数据（使用相同的内部表示，假设，用于 long 或 double 类型的值）或图片或音乐编码，该文件就是二进制文件，其中包含二进制内容。

UNIX 用同一种文件格式处理文本文件和二进制文件的内容。不奇怪，鉴于 C 是作为开发 UNIX 的工具而创建的，C 和 UNIX 在文本中都使用\n（换行符）表示换行。UNIX 目录中有一个统计文件大小的计数，程序可使用该计数确定是否读到文件结尾。然而，其他系统在此之前已经有其他方法处理文件，专门用于保存文本。也就是说，其他系统已经有一种与 UNIX 模型不同的格式处理文本文件。例如，以前的 OS X Macintosh 文件用\r（回车符）表示新的一行。早期的 MS-DOS 文件用\r\n组合表示新的一行，用嵌入的 Ctrl+Z 字符表示文件结尾，即使实际文件用添加空字符的方法使其总大小是 256 的倍数（在 Windows 中，Notepad 仍然生成 MS-DOS 格式的文本文件，但是新的编辑器可能使用类 UNIX 格式居多）。其他系统可能保持文本文件中的每一行长度相同，如有必要，用空字符填充每一行，使其长度保持一致。或者，系统可能在每行的开始标出每行的长度。

为了规范文本文件的处理，C 提供两种访问文件的途径：二进制模式和文本模式。在二进制模式中，程序可以访问文件的每个字节。而在文本模式中，程序所见的内容和文件的实际内容不同。程序以文本模式读取文件时，把本地环境表示的行末尾或文件结尾映射为 C 模式。例如，C 程序在旧式 Macintosh 中以文本模式读取文件时，把文件中的\r 转换成\n；以文本模式写入文件时，把\n 转换成\r。或者，C 文本模式程序在 MS-DOS 平台读取文件时，把\r\n转换成\n；写入文件时，把\n 转换成\r\n。在其他环境中编写的文本模式程序也会做类似的转换。

除了以文本模式读写文本文件，还能以二进制模式读写文本文件。如果读写一个旧式 MS-DOS 文本文档，程序会看到文件中的\r 和\n 字符，不会发生映射（图 13.1 演示了一些文本）。如果要编写旧式 Mac 格式、MS-DOS 格式或 UNIX/Linux 格式的文件模式程序，应该使用二进制模式，这样程序才能确定实际的文件内容并执行相应的动作。

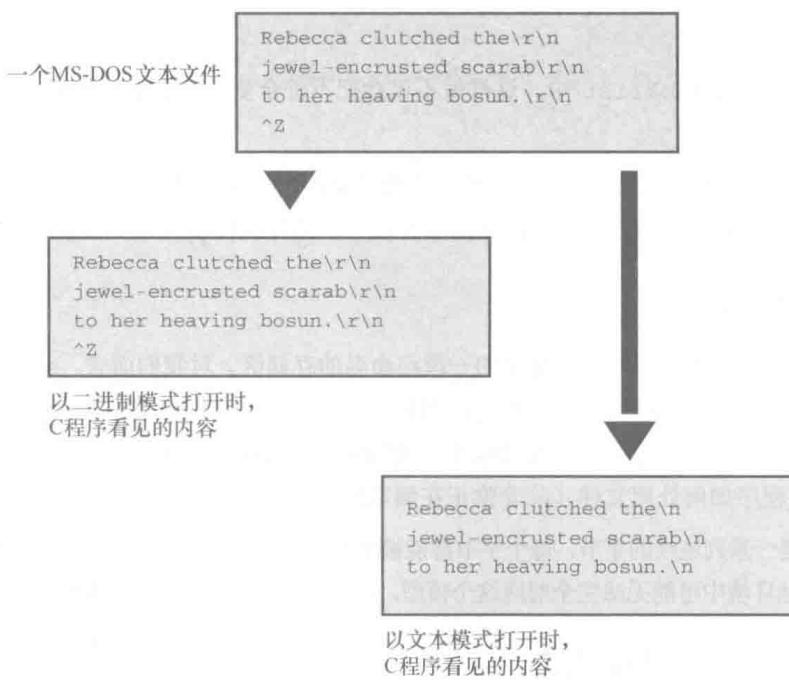


图 13.1 二进制模式和文本模式

虽然 C 提供了二进制模式和文本模式，但是这两种模式的实现可以相同。前面提到过，因为 UNIX 使用一种文件格式，这两种模式对于 UNIX 实现而言完全相同。Linux 也是如此。

### 13.1.3 I/O 的级别

除了选择文件的模式，大多数情况下，还可以选择 I/O 的两个级别（即处理文件访问的两个级别）。底层 I/O (*low-level I/O*) 使用操作系统提供的基本 I/O 服务。标准高级 I/O (*standard high-level I/O*) 使用 C 库的标准包和 `stdio.h` 头文件定义。因为无法保证所有的操作系统都使用相同的底层 I/O 模型，C 标准只支持标准 I/O 包。有些实现会提供底层库，但是 C 标准建立了可移植的 I/O 模型，我们主要讨论这些 I/O。

### 13.1.4 标准文件

C 程序会自动打开 3 个文件，它们被称为标准输入 (*standard input*)、标准输出 (*standard output*) 和标准错误输出 (*standard error output*)。在默认情况下，标准输入是系统的普通输入设备，通常为键盘；标准输出和标准错误输出是系统的普通输出设备，通常为显示屏。

通常，标准输入为程序提供输入，它是 `getchar()` 和 `scanf()` 使用的文件。程序通常输出到标准输出，它是 `putchar()`、`puts()` 和 `printf()` 使用的文件。第 8 章提到的重定向把其他文件视为标准输入或标准输出。标准错误输出提供了一个逻辑上不同的地方来发送错误消息。例如，如果使用重定向把输出发送给文件而不是屏幕，那么发送至标准错误输出的内容仍然会被发送到屏幕上。这样很好，因为如果把错误消息发送至文件，就只能打开文件才能看到。

## 13.2 标准 I/O

与底层 I/O 相比，标准 I/O 包除了可移植以外还有两个好处。第一，标准 I/O 有许多专门的函数简化了处理不同 I/O 的问题。例如，`printf()` 把不同形式的数据转换成与终端相适应的字符串输出。第二，输入和输出都是缓冲的。也就是说，一次转移一大块信息而不是一字节信息（通常至少 512 字节）。例如，当程序读取文件时，一块数据被拷贝到缓冲区（一块中介存储区域）。这种缓冲极大地提高了数据传输速率。程序可以检查缓冲区中的字节。缓冲在后台处理，所以让人有逐字符访问的错觉（如果使用底层 I/O，要自己完成大部分工作）。程序清单 13.1 演示了如何用标准 I/O 读取文件和统计文件中的字符数。我们将在后面几节讨论程序清单 13.1 中的一些特性。该程序使用命令行参数，如果你是 Windows 用户，在编译后必须在命令提示窗口运行该程序；如果你是 Macintosh 用户，最简单的方法是使用 Terminal 在命令行形式中编译并运行该程序。或者，如第 11 章所述，如果在 IDE 中运行该程序，可以使用 Xcode 的 Product 菜单提供命令行参数。或者也可以用 `puts()` 和 `fgets()` 函数替换命令行参数来获得文件名。

程序清单 13.1 `count.c` 程序

---

```
/* count.c -- 使用标准 I/O */
#include <stdio.h>
#include <stdlib.h> // 提供 exit() 的原型

int main(int argc, char *argv[])
{
 int ch; // 读取文件时，储存每个字符的地方
 FILE *fp; // “文件指针”
 unsigned long count = 0;
 if (argc != 2)
```

```

{
 printf("Usage: %s filename\n", argv[0]);
 exit(EXIT_FAILURE);
}
if ((fp = fopen(argv[1], "r")) == NULL)
{
 printf("Can't open %s\n", argv[1]);
 exit(EXIT_FAILURE);
}
while ((ch = getc(fp)) != EOF)
{
 putc(ch, stdout); // 与 putchar(ch); 相同
 count++;
}
fclose(fp);
printf("File %s has %lu characters\n", argv[1], count);

return 0;
}

```

### 13.2.1 检查命令行参数

首先，程序清单 13.1 中的程序检查 `argc` 的值，查看是否有命令行参数。如果没有，程序将打印一条消息并退出程序。字符串 `argv[0]` 是该程序的名称。显式使用 `argv[0]` 而不是程序名，错误消息的描述会随可执行文件名的改变而自动改变。这一特性在像 UNIX 这种允许单个文件具有多个文件名的环境中也很方便。但是，一些操作系统可能不识别 `argv[0]`，所以这种用法并非完全可移植。

`exit()` 函数关闭所有打开的文件并结束程序。`exit()` 的参数被传递给一些操作系统，包括 UNIX、Linux、Windows 和 MS-DOS，以供其他程序使用。通常的惯例是：正常结束的程序传递 0，异常结束的程序传递非零值。不同的退出值可用于区分程序失败的不同原因，这也是 UNIX 和 DOS 编程的通常做法。但是，并不是所有的操作系统都能识别相同范围内的返回值。因此，C 标准规定了一个最小的限制范围。尤其是，标准要求 0 或宏 `EXIT_SUCCESS` 用于表明成功结束程序，宏 `EXIT_FAILURE` 用于表明结束程序失败。这些宏和 `exit()` 原型都位于 `stdlib.h` 头文件中。

根据 ANSI C 的规定，在最初调用的 `main()` 中使用 `return` 与调用 `exit()` 的效果相同。因此，在 `main()`，下面的语句：

```
return 0;
```

和下面这条语句的作用相同：

```
exit(0);
```

但是要注意，我们说的是“最初的调用”。如果 `main()` 在一个递归程序中，`exit()` 仍然会终止程序，但是 `return` 只会把控制权交给上一级递归，直至最初的一级。然后 `return` 结束程序。`return` 和 `exit()` 的另一个区别是，即使在其他函数中（除 `main()` 以外）调用 `exit()` 也能结束整个程序。

### 13.2.2 `fopen()` 函数

继续分析程序清单 13.1，该程序使用 `fopen()` 函数打开文件。该函数声明在 `stdio.h` 中。它的第 1 个参数是待打开文件的名称，更确切地说是一个包含改文件名的字符串地址。第 2 个参数是一个字符串，指定待打开文件的模式。表 13.1 列出了 C 库提供的一些模式。

表 13.1 fopen() 的模式字符串

| 模式字符串                                                      | 含义                                                                |
|------------------------------------------------------------|-------------------------------------------------------------------|
| "r"                                                        | 以读模式打开文件                                                          |
| "w"                                                        | 以写模式打开文件，把现有文件的长度截为 0，如果文件不存在，则创建一个新文件                            |
| "a"                                                        | 以写模式打开文件，在现有文件末尾添加内容，如果文件不存在，则创建一个新文件                             |
| "r+"                                                       | 以更新模式打开文件（即可以读写文件）                                                |
| "w+"                                                       | 以更新模式打开文件（即，读和写），如果文件存在，则将其长度截为 0；如果文件不存在，则创建一个新文件                |
| "a+"                                                       | 以更新模式打开文件（即，读和写），在现有文件的末尾添加内容，如果文件不存在则创建一个新文件；可以读整个文件，但是只能从末尾添加内容 |
| "rb"、"wb"、"ab"、"ab+"、<br>"a+b"、"wb+"、"w+b"、<br>"ab+"、"a+b" | 与上一个模式类似，但是以二进制模式而不是文本模式打开文件                                      |
| "wx"、"wbx"、<br>"w+x"、"wb+x"或"w+bx"                         | (C11) 类似非 x 模式，但是如果文件已存在或以独占模式打开文件，则打开文件失败                        |

像 UNIX 和 Linux 这样只有一种文件类型的系统，带 b 字母的模式和不带 b 字母的模式相同。

新的 C11 新增了带 x 字母的写模式，与以前的写模式相比具有更多特性。第一，如果以传统的一种写模式打开一个现有文件，fopen() 会把该文件的长度截为 0，这样就丢失了该文件的内容。但是使用带 x 字母的写模式，即使 fopen() 操作失败，原文件的内容也不会被删除。第二，如果环境允许，x 模式的独占特性使得其他程序或线程无法访问正在被打开的文件。

### 警告

如果使用任何一种 "w" 模式（不带 x 字母）打开一个现有文件，该文件的内容会被删除，以便程序在一个空白文件中开始操作。然而，如果使用带 x 字母的任何一种模式，将无法打开一个现有文件。

程序成功打开文件后，fopen() 将返回文件指针 (*file pointer*)，其他 I/O 函数可以使用这个指针指定该文件。文件指针（该例中是 fp）的类型是指向 FILE 的指针，FILE 是一个定义在 stdio.h 中的派生类型。文件指针 fp 并不指向实际的文件，它指向一个包含文件信息的数据对象，其中包含操作文件的 I/O 函数所用的缓冲区信息。因为标准库中的 I/O 函数使用缓冲区，所以它们不仅要知道缓冲区的位置，还要知道缓冲区被填充的程度以及操作哪一个文件。标准 I/O 函数根据这些信息在必要时决定再次填充或清空缓冲区。fp 指向的数据对象包含了这些信息（该数据对象是一个 C 结构，将在第 14 章中介绍）。

### 13.2.3 getc() 和 putc() 函数

getc() 和 putc() 函数与 getchar() 和 putchar() 函数类似。所不同的是，要告诉 getc() 和 putc() 函数使用哪一个文件。下面这条语句的意思是“从标准输入中获取一个字符”：

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

然而，下面这条语句的意思是“从 fp 指定的文件中获取一个字符”：

```
ch = getc(fp);
```

与此类似，下面语句的意思是“把字符 ch 放入 FILE 指针 fpout 指定的文件中”：

```
putc(ch, fpout);
```

在 putc() 函数的参数列表中，第 1 个参数是待写入的字符，第 2 个参数是文件指针。

程序清单 13.1 把 stdout 作为 putc() 的第 2 个参数。stdout 作为与标准输出相关联的文件指针，定义在 stdio.h 中，所以 putc(ch, stdout) 与 putchar(ch) 的作用相同。实际上，putchar() 函数一般通过 putc() 来定义。与此类似，getchar() 也通过使用标准输入的 getc() 来定义。

为何该示例不用 putchar() 而要用 putc()？原因之一是为了介绍 putc() 函数；原因之二是，把 stdout 替换成别的参数，很容易将这段程序改写成文件输出。

### 13.2.4 文件结尾

从文件中读取数据的程序在读到文件结尾时要停止。如何告诉程序已经读到文件结尾？如果 getc() 函数在读取一个字符时发现是文件结尾，它将返回一个特殊值 EOF。所以 C 程序只有在读到超过文件末尾时才会发现文件的结尾（一些其他语言用一个特殊的函数在读取之前测试文件结尾，C 语言不同）。

为了避免读到空文件，应该使用入口条件循环（不是 do while 循环）进行文件输入。鉴于 getc()（和其他 C 输入函数）的设计，程序应该在进入循环体之前先尝试读取。如下面设计所示：

```
// 设计范例 #1
int ch; // 用 int 类型的变量储存 EOF
FILE * fp;
fp = fopen("wacky.txt", "r");
ch = getc(fp); // 获得初始输入
while (ch != EOF)
{
 putchar(ch); // 处理输入
 ch = getc(fp); // 获得下一个输入
}
```

以上代码可简化为：

```
// 设计范例 #2
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while ((ch = getc(fp)) != EOF)
{
 putchar(ch); // 处理输入
}
```

由于 ch = getc(fp) 是 while 测试条件的一部分，所以程序在进入循环体之前就读取了文件。不要设计成下面这样：

```
// 糟糕的设计（存在两个问题）
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while (ch != EOF) // 首次使用 ch 时，它的值尚未确定
{
 ch = getc(fp); // 获得输入
 putchar(ch); // 处理输入
}
```

第1个问题是，ch首次与EOF比较时，其值尚未确定。第2个问题是，如果getc()返回EOF，该循环会把EOF作为一个有效字符处理。这些问题都可以解决。例如，把ch初始化为一个哑值(dummy value)，再把一个if语句加入到循环中。但是，何必多此一举，直接使用上面的设计范例即可。

其他输入函数也会用到这种处理方案，它们在读到文件结尾时也会返回一个错误信号(EOF或NULL指针)。

### 13.2.5 fclose() 函数

fclose(fp)函数关闭fp指定的文件，必要时刷新缓冲区。对于较正式的程序，应该检查是否成功关闭文件。如果成功关闭，fclose()函数返回0，否则返回EOF：

```
if (fclose(fp) != 0)
 printf("Error in closing file %s\n", argv[1]);
```

如果磁盘已满、移动硬盘被移除或出现I/O错误，都会导致调用fclose()函数失败。

### 13.2.6 指向标准文件的指针

stdio.h头文件把3个文件指针与3个标准文件相关联，C程序会自动打开这3个标准文件。如表13.2所示：

表 13.2 标准文件和相关联的文件指针

| 标准文件 | 文件指针   | 通常使用的设备 |
|------|--------|---------|
| 标准输入 | stdin  | 键盘      |
| 标准输出 | stdout | 显示器     |
| 标准错误 | stderr | 显示器     |

这些文件指针都是指向FILE的指针，所以它们可用作标准I/O函数的参数，如fclose(fp)中的fp。接下来，我们用一个程序示例创建一个新文件，并写入内容。

## 13.3 一个简单的文件压缩程序

下面的程序示例把一个文件中选定的数据拷贝到另一个文件中。该程序同时打开了两个文件，以“r”模式打开一个，以“w”模式打开另一个。该程序(程序清单13.2)以保留每3个字符中的第1个字符的方式压缩第1个文件的内容。最后，把压缩后的文本存入第2个文件。第2个文件的名称是第1个文件名加上.red后缀(此处的red代表reduced)。使用命令行参数，同时打开多个文件，以及在原文件名后面加上后缀，都是相当有用的技巧。这种压缩方式有限，但是也有它的用途(很容易把该程序改成用标准I/O而不是命令行参数提供文件名)。

程序清单 13.2 reducto.c 程序

---

```
// reducto.c -把文件压缩成原来的1/3!
#include <stdio.h>
#include <stdlib.h> // 提供exit()的原型
#include <string.h> // 提供strcpy()、strcat()的原型
#define LEN 40

int main(int argc, char *argv [])
```

```

{
 FILE *in, *out; // 声明两个指向 FILE 的指针
 int ch;
 char name[LEN]; // 储存输出文件名
 int count = 0;

 // 检查命令行参数
 if (argc < 2)
 {
 fprintf(stderr, "Usage: %s filename\n", argv[0]);
 exit(EXIT_FAILURE);
 }
 // 设置输入
 if ((in = fopen(argv[1], "r")) == NULL)
 {
 fprintf(stderr, "I couldn't open the file \"%s\"\n",
 argv[1]);
 exit(EXIT_FAILURE);
 }
 // 设置输出
 strncpy(name, argv[1], LEN - 5); // 拷贝文件名
 name[LEN - 5] = '\0';
 strcat(name, ".red"); // 在文件名后添加.red
 if ((out = fopen(name, "w")) == NULL)
 {
 // 以写模式打开文件
 fprintf(stderr, "Can't create output file.\n");
 exit(3);
 }
 // 拷贝数据
 while ((ch = getc(in)) != EOF)
 if (count++ % 3 == 0)
 putc(ch, out); // 打印 3 个字符中的第 1 个字符
 // 收尾工作
 if (fclose(in) != 0 || fclose(out) != 0)
 fprintf(stderr, "Error in closing files\n");

 return 0;
}

```

假设可执行文件名是 `reducto`, 待读取的文件名为 `eddy`, 该文件中包含下面一行内容:

`So even Eddy came over ready.`

命令如下:

`reducto eddy`

待写入的文件名为 `eddy.red`。该程序把输出显示在 `eddy.red` 中, 而不是屏幕上。打开 `eddy.red`, 内容如下:

`Send money`

该程序示例演示了几个编程技巧。我们来仔细研究一下。

`fprintf()` 和 `printf()` 类似, 但是 `fprintf()` 的第 1 个参数必须是一个文件指针。程序中使用 `stderr` 指针把错误消息发送至标准错误, C 标准通常都这么做。

为了构造新的输出文件名，该程序使用 `strncpy()` 把名称 `eddy` 拷贝到数组 `name` 中。参数 `LEN-5` 为 `.red` 后缀和末尾的空字符预留了空间。如果 `argv[2]` 字符串比 `LEN-5` 长，就拷贝不了空字符。出现这种情况时，程序会添加空字符。调用 `strncpy()` 后，`name` 中的第一个空字符在调用 `strcat()` 函数时，被 `.red` 的 `.` 覆盖，生成了 `eddy.red`。程序中还检查了是否成功打开名为 `eddy.red` 的文件。这个步骤在一些环境中相当重要，因为像 `strange.c.red` 这样的文件名可能是无效的。例如，在传统的 DOS 环境中，不能在后缀名后面添加后缀名（MS-DOS 使用的方法是用 `.red` 替换现有后缀名，所以 `strange.c` 将变成 `strange.red`）。例如，可以用 `strchr()` 函数定位（如果有的话），然后只拷贝点前面的部分即可）。

该程序同时打开了两个文件，所以我们要声明两个 `FIFL` 指针。注意，程序都是单独打开和关闭每个文件。同时打开的文件数量是有限的，这个限制取决于系统和实现，范围一般是  $10\sim20$ 。相同的文件指针可以处理不同的文件，前提是这些文件不需要同时打开。

## 13.4 文件 I/O: fprintf()、fscanf()、fgets() 和 fputs()

前面章节介绍的 I/O 函数都类似于文件 I/O 函数。它们的主要区别是，文件 I/O 函数要用 `FILE` 指针指定待处理的文件。与 `getc()`、`putc()` 类似，这些函数都要求用指向 `FILE` 的指针（如，`stdout`）指定一个文件，或者使用 `fopen()` 的返回值。

### 13.4.1 fprintf() 和 fscanf() 函数

文件 I/O 函数 `fprintf()` 和 `fscanf()` 函数的工作方式与 `printf()` 和 `scanf()` 类似，区别在于前者需要用第 1 个参数指定待处理的文件。我们在前面用过 `fprintf()`。程序清单 13.3 演示了这两个文件 I/O 函数和 `rewind()` 函数的用法。

程序清单 13.3 addaword.c 程序

```
/* addaword.c -- 使用 fprintf()、fscanf() 和 rewind() */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 41

int main(void)
{
 FILE *fp;
 char words[MAX];

 if ((fp = fopen("wordy", "a+")) == NULL)
 {
 fprintf(stderr, "Can't open \"wordy\" file.\n");
 exit(EXIT_FAILURE);
 }

 puts("Enter words to add to the file; press the #");
 puts("key at the beginning of a line to terminate.");
 while ((fscanf(stdin, "%40s", words) == 1) && (words[0] != '#'))
 fprintf(fp, "%s\n", words);

 puts("File contents:");
 rewind(fp); /* 返回到文件开始处 */
 while (fscanf(fp, "%s", words) == 1)
```

```

 puts(words);
 puts("Done!");
 if (fclose(fp) != 0)
 fprintf(stderr, "Error closing file\n");

 return 0;
}

```

该程序可以在文件中添加单词。使用 "a+" 模式，程序可以对文件进行读写操作。首次使用该程序，它将创建 wordy 文件，以便把单词存入其中。随后再使用该程序，可以在 wordy 文件后面添加单词。虽然 "a+" 模式只允许在文件末尾添加内容，但是该模式下可以读整个文件。rewind() 函数让程序回到文件开始处，方便 while 循环打印整个文件的内容。注意，rewind() 接受一个文件指针作为参数。

下面是该程序在 UNIX 环境中的一个运行示例（可执行程序已重命名为 addword）：

```

$ addword
Enter words to add to the file; press the Enter
key at the beginning of a line to terminate.
The fabulous programmer
#
File contents:
The
fabulous
programmer
Done!
$ addword
Enter words to add to the file; press the Enter
key at the beginning of a line to terminate.
enchanted the
large
#
File contents:
The
fabulous
programmer
enchanted
the
large
Done!

```

如你所见，fprintf() 和 fscanf() 的工作方式与 printf() 和 scanf() 类似。但是，与 putc() 不同的是，fprintf() 和 fscanf() 函数都把 FILE 指针作为第 1 个参数，而不是最后一个参数。

### 13.4.2 fgets() 和 fputs() 函数

第 11 章时介绍过 fgets() 函数。它的第 1 个参数和 gets() 函数一样，也是表示储存输入位置的地址 (char \* 类型)；第 2 个参数是一个整数，表示待输入字符串的大小<sup>1</sup>；最后一个参数是文件指针，指定待读取的文件。下面是一个调用该函数的例子：

```
fgets(buf, STLEN, fp);
```

这里，buf 是 char 类型数组的名称，STLEN 是字符串的大小，fp 是指向 FILE 的指针。

fgets() 函数读取输入直到第 1 个换行符的后面，或读到文件结尾，或者读取 STLEN-1 个字符（以

<sup>1</sup> 注意，字符串大小和字符串长度不同。前者指该字符串占用多少空间，后者指该字符串的字符个数。——译者注

上面的 fgets() 为例)。然后, fgets() 在末尾添加一个空字符使之成为一个字符串。字符串的大小是其字符数加上一个空字符。如果 fgets() 在读到字符上限之前已读完一整行, 它会把表示行结尾的换行符放在空字符前面。fgets() 函数在遇到 EOF 时将返回 NULL 值, 可以利用这一机制检查是否到达文件结尾; 如果未遇到 EOF 则之前返回传给它的地址。

fputs() 函数接受两个参数: 第 1 个是字符串的地址; 第 2 个是文件指针。该函数根据传入地址找到的字符串写入指定的文件中。和 puts() 函数不同, fputs() 在打印字符串时不会在其末尾添加换行符。下面是一个调用该函数的例子:

```
fputs(buf, fp);
```

这里, buf 是字符串的地址, fp 用于指定目标文件。

由于 fgets() 保留了换行符, fputs() 就不会再添加换行符, 它们配合得非常好。如第 11 章的程序清单 11.8 所示, 即使输入行比 SLEN 长, 这两个函数依然处理得很好。

## 13.5 随机访问: fseek() 和 ftell()

有了 fseek() 函数, 便可把文件看作是数组, 在 fopen() 打开的文件中直接移动到任意字节处。我们创建一个程序 (程序清单 13.4) 演示 fseek() 和 ftell() 的用法。注意, fseek() 有 3 个参数, 返回 int 类型的值; ftell() 函数返回一个 long 类型的值, 表示文件中的当前位置。

程序清单 13.4 reverse.c 程序

```
/* reverse.c -- 倒序显示文件的内容 */
#include <stdio.h>
#include <stdlib.h>
#define CNTL_Z '\032' /* DOS 文本文件中的文件结尾标记 */
#define SLEN 81
int main(void)
{
 char file[SLEN];
 char ch;
 FILE *fp;
 long count, last;

 puts("Enter the name of the file to be processed:");
 scanf("%80s", file);
 if ((fp = fopen(file, "rb")) == NULL)
 {
 /* 只读模式 */
 printf("reverse can't open %s\n", file);
 exit(EXIT_FAILURE);
 }

 fseek(fp, 0L, SEEK_END); /* 定位到文件末尾 */
 last = ftell(fp);
 for (count = 1L; count <= last; count++)
 {
 fseek(fp, -count, SEEK_END); /* 回退 */
 ch = getc(fp);
 if (ch != CNTL_Z && ch != '\r') /* MS-DOS 文件 */
 putchar(ch);
 }
 putchar('\n');
```

```

fclose(fp);

return 0;
}

```

下面是对一个文件的输出：

```
Enter the name of the file to be processed:
```

```
Cluv
```

```
.C ni eno naht ylevol erom margorp a
ees reven llahs I taht kniht I
```

该程序使用二进制模式，以便处理 MS-DOS 文本和 UNIX 文件。但是，在使用其他格式文本文件的环境中可能无法正常工作。

### 注意

如果通过命令行环境运行该程序，待处理文件要和可执行文件在同一个目录（或文件夹）中。如果在 IDE 中运行该程序，具体查找方案序因实现而异。例如，默认情况下，Microsoft Visual Studio 2012 在源代码所在的目录中查找，而 Xcode 4.6 则在可执行文件所在的目录中查找。

接下来，我们要讨论 3 个问题：`fseek()` 和 `ftell()` 函数的工作原理、如何使用二进制流、如何让程序可移植。

#### 13.5.1 `fseek()` 和 `ftell()` 的工作原理

`fseek()` 的第 1 个参数是 `FILE` 指针，指向待查找的文件，`open()` 应该已打开该文件。

`fseek()` 的第 2 个参数是偏移量 (*offset*)。该参数表示从起始点开始要移动的距离（参见表 13.3 列出的起始点模式）。该参数必须是一个 `long` 类型的值，可以为正（前移）、负（后移）或 0（保持不动）。

`fseek()` 的第 3 个参数是模式，该参数确定起始点。根据 ANSI 标准，在 `stdio.h` 头文件中规定了几个表示模式的明示常量 (*manifest constant*)，如表 13.3 所示。

表 13.3 文件的起始点模式

| 模式                    | 偏移量的起始点 |
|-----------------------|---------|
| <code>SEEK_SET</code> | 文件开始处   |
| <code>SEEK_CUR</code> | 当前位置    |
| <code>SEEK_END</code> | 文件末尾    |

旧的实现可能缺少这些定义，可以使用数值 `0L`、`1L`、`2L` 分别表示这 3 种模式。`L` 后缀表明其值是 `long` 类型。或者，实现可能把这些明示常量定义在别的头文件中。如果不确定，请查阅实现的使用手册或在线帮助。

下面是调用 `fseek()` 函数的一些示例，`fp` 是一个文件指针：

```

fseek(fp, 0L, SEEK_SET); // 定位至文件开始处
fseek(fp, 10L, SEEK_SET); // 定位至文件中的第 10 个字节
fseek(fp, 2L, SEEK_CUR); // 从文件当前位置前移 2 个字节

```

```
fseek(fp, 0L, SEEK_END); // 定位至文件结尾
fseek(fp, -10L, SEEK_END); // 从文件结尾处回退 10 个字节
```

对于这些调用还有一些限制，我们稍后再讨论。

如果一切正常，fseek()的返回值为0；如果出现错误（如试图移动的距离超出文件的范围），其返回值为-1。

ftell()函数的返回类型是long，它返回的是当前的位置。ANSI C把它定义在stdio.h中。在最初实现的UNIX中，ftell()通过返回距文件开始处的字节数来确定文件的位置。文件的第一个字节到文件开始处的距离是0，以此类推。ANSI C规定，该定义适用于以二进制模式打开的文件，以文件模式打开文件的情况不同。这也是程序清单13.4以二进制模式打开文件的原因。

下面，我们来分析程序清单13.4中的基本要素。首先，下面的语句：

```
fseek(fp, 0L, SEEK_END);
```

把当前位置设置为距文件末尾0字节偏移量。也就是说，该语句把当前位置设置在文件结尾。下一条语句：

```
last = ftell(fp);
```

把从文件开始处到文件结尾的字节数赋给last。

然后是一个for循环：

```
for (count = 1L; count <= last; count++)
{
 fseek(fp, -count, SEEK_END); /* go backward */
 ch = getc(fp);
}
```

第1轮迭代，把程序定位到文件结尾的第一个字符（即，文件的最后一个字符）。然后，程序打印该字符。下一轮迭代把程序定位到前一个字符，并打印该字符。重复这一过程直至到达文件的第一个字符，并打印。

## 13.5.2 二进制模式和文本模式

我们设计的程序清单13.4在UNIX和MS-DOS环境下都可以运行。UNIX只有一种文件格式，所以不需要进行特殊的转换。然而MS-DOS要格外注意。许多MS-DOS编辑器都用Ctrl+Z标记文本文件的结尾。以文本模式打开这样的文件时，C能识别这个作为文件结尾标记的字符。但是，以二进制模式打开相同的文件时，Ctrl+Z字符被看作是文件中的一个字符，而实际的文件结尾符在该字符的后面。文件结尾符可能紧跟在Ctrl+Z字符后面，或者文件中可能用空字符填充，使该文件的大小是256的倍数。在DOS环境下不会打印空字符，程序清单13.4中就包含了防止打印Ctrl+Z字符的代码。

二进制模式和文本模式的另一个不同之处是：MS-DOS用\r\n组合表示文本文件换行。以文本模式打开相同的文件时，C程序把\r\n“看成”\n。但是，以二进制模式打开该文件时，程序能看见这两个字符。因此，程序清单13.4中还包含了不打印\r的代码。通常，UNIX文本文件既没有Ctrl+Z，也没有\r，所以这部分代码不会影响大部分UNIX文本文件。

ftell()函数在文本模式和二进制模式中的工作方式不同。许多系统的文本文件格式与UNIX的模型有很大不同，导致从文件开始处统计的字节数成为一个毫无意义的值。ANSI C规定，对于文本模式，ftell()返回的值可以作为fseek()的第2个参数。对于MS-DOS，ftell()返回的值把\r\n当作一个字节计数。

## 13.5.3 可移植性

理论上，fseek()和ftell()应该符合UNIX模型。但是，不同系统存在着差异，有时确实无法做到

与 UNIX 模型一致。因此，ANSI 对这些函数降低了要求。下面是一些限制。

- 在二进制模式中，实现不必支持 SEEK\_END 模式。因此无法保证程序清单 13.4 的可移植性。移植性更高的方法是逐字节读取整个文件直到文件末尾。C 预处理器的条件编译指令（第 16 章介绍）提供了一种系统方法来处理这种情况。
- 在文本模式中，只有以下调用能保证其相应的行为。

| 函数调用                                          | 效果                                                                                        |
|-----------------------------------------------|-------------------------------------------------------------------------------------------|
| <code>fseek(file, 0L, SEEK_SET)</code>        | 定位至文件开始处                                                                                  |
| <code>fseek(file, 0L, SEEK_CUR)</code>        | 保持当前位置不动                                                                                  |
| <code>fseek(file, 0L, SEEK_END)</code>        | 定位至文件结尾                                                                                   |
| <code>fseek(file, ftell-pos, SEEK_SET)</code> | 到距文件开始处 <code>ftell-pos</code> 的位置，<br><code>ftell-pos</code> 是 <code>ftell()</code> 的返回值 |

不过，许多常见的环境都支持更多的行为。

### 13.5.4 fgetpos() 和 fsetpos() 函数

`fseek()` 和 `ftell()` 潜在的问题是，它们都把文件大小限制在 `long` 类型能表示的范围内。也许 20 亿字节看起来相当大，但是随着存储设备的容量迅猛增长，文件也越来越大。鉴于此，ANSI C 新增了两个处理较大文件的新定位函数：`fgetpos()` 和 `fsetpos()`。这两个函数不使用 `long` 类型的值表示位置，它们使用一种新类型：`fpos_t`（代表 file position type，文件定位类型）。`fpos_t` 类型不是基本类型，它根据其他类型来定义。`fpos_t` 类型的变量或数据对象可以在文件中指定一个位置，它不能是数组类型，除此之外，没有其他限制。实现可以提供一个满足特殊平台要求的类型，例如，`fpos_t` 可以实现为结构。

ANSI C 定义了如何使用 `fpos_t` 类型。`fgetpos()` 函数的原型如下：

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

调用该函数时，它把 `fpos_t` 类型的值放在 `pos` 指向的位置上，该值描述了文件中的一个位置。如果成功，`fgetpos()` 函数返回 0；如果失败，返回非 0。

`fsetpos()` 函数的原型如下：

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

调用该函数时，使用 `pos` 指向位置上的 `fpos_t` 类型值来设置文件指针指向该值指定的位置。如果成功，`fsetpos()` 函数返回 0；如果失败，则返回非 0。`fpos_t` 类型的值应通过之前调用 `fgetpos()` 获得。

## 13.6 标准 I/O 的机理

我们在前面学习了标准 I/O 包的特性，本节研究一个典型的概念模型，分析标准 I/O 的工作原理。

通常，使用标准 I/O 的第 1 步是调用 `fopen()` 打开文件（前面介绍过，C 程序会自动打开 3 种标准文件）。`fopen()` 函数不仅打开一个文件，还创建了一个缓冲区（在读写模式下会创建两个缓冲区）以及一个包含文件和缓冲区数据的结构。另外，`fopen()` 返回一个指向该结构的指针，以便其他函数知道如何找到该结构。假设把该指针赋给一个指针变量 `fp`，我们说 `fopen()` 函数“打开一个流”。如果以文本模式打开该文件，就获得一个文本流；如果以二进制模式打开该文件，就获得一个二进制流。

这个结构通常包含一个指定流中当前位置的文件位置指示器。除此之外，它还包含错误和文件结

尾的指示器、一个指向缓冲区开始处的指针、一个文件标识符和一个计数（统计实际拷贝进缓冲区的字节数）。

我们主要考虑文件输入。通常，使用标准 I/O 的第 2 步是调用一个定义在 `stdio.h` 中的输入函数，如 `fscanf()`、`getc()` 或 `fgets()`。一调用这些函数，文件中的数据块就被拷贝到缓冲区中。缓冲区的大小因实现而异，一般是 512 字节或是它的倍数，如 4096 或 16384（随着计算机硬盘容量越来越大，缓冲区的大小也越来越大）。最初调用函数，除了填充缓冲区外，还要设置 `fp` 所指向的结构中的值。尤其要设置流中的当前位置和拷贝进缓冲区的字节数。通常，当前位置从字节 0 开始。

在初始化结构和缓冲区后，输入函数按要求从缓冲区中读取数据。在它读取数据时，文件位置指示器被设置为指向刚读取字符的下一个字符。由于 `stdio.h` 系列的所有输入函数都使用相同的缓冲区，所以调用任何一个函数都将从上一次函数停止调用的位置开始。

当输入函数发现已读完缓冲区中的所有字符时，会请求把下一个缓冲大小的数据块从文件拷贝到该缓冲区中。以这种方式，输入函数可以读取文件中的所有内容，直到文件结尾。函数在读取缓冲区中的最后一个字符后，把结尾指示器设置为真。于是，下一次被调用的输入函数将返回 `EOF`。

输出函数以类似的方式把数据写入缓冲区。当缓冲区被填满时，数据将被拷贝至文件中。

## 13.7 其他标准 I/O 函数

ANSI 标准库的标准 I/O 系列有几十个函数。虽然在这里无法一一列举，但是我们会简要地介绍一些，让读者对它们有一个大概的了解。这里列出函数的原型，表明函数的参数和返回类型。我们要讨论的这些函数，除了 `setvbuf()`，其他函数均可在 ANSI 之前的实现中使用。参考资料 V 的“新增 C99 和 C11 的标准 ANSI C 库”中列出了全部的 ANSI C 标准 I/O 包。

### 13.7.1 int ungetc(int c, FILE \*fp) 函数

`int ungetc()` 函数把 `c` 指定的字符放回输入流中。如果把一个字符放回输入流，下次调用标准输入函数时将读取该字符（见图 13.2）。例如，假设要读取下一个冒号之前的所有字符，但是不包括冒号本身，可以使用 `getchar()` 或 `getc()` 函数读取字符到冒号，然后使用 `ungetc()` 函数把冒号放回输入流中。ANSI C 标准保证每次只会放回一个字符。如果实现允许把一行中的多个字符放回输入流，那么下一次输入函数读入的字符顺序与放回时的顺序相反。

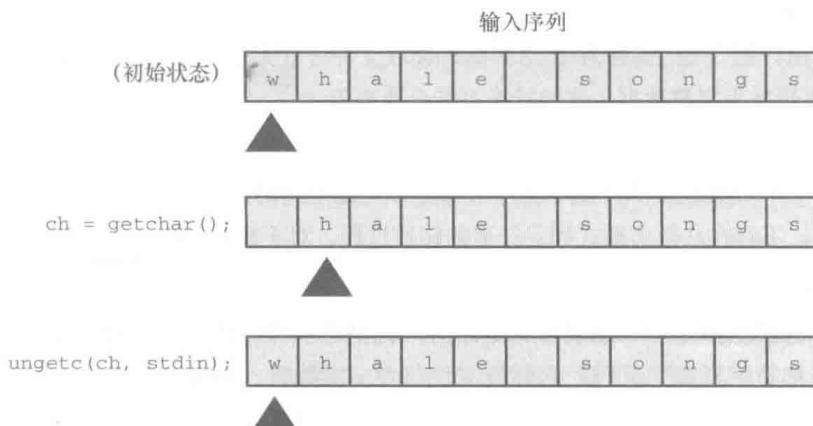


图 13.2 `ungets()` 函数

### 13.7.2 int fflush() 函数

`fflush()` 函数的原型如下：

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

调用 `fflush()` 函数引起输出缓冲区中所有的未写入数据被发送到 `fp` 指定的输出文件。这个过程称为刷新缓冲区。如果 `fp` 是空指针，所有输出缓冲区都被刷新。在输入流中使用 `fflush()` 函数的效果是未定义的。只要最近一次操作不是输入操作，就可以用该函数来更新流（任何读写模式）。

### 13.7.3 int setvbuf() 函数

`setvbuf()` 函数的原型是：

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

`setvbuf()` 函数创建了一个供标准 I/O 函数替换使用的缓冲区。在打开文件后且未对流进行其他操作之前，调用该函数。指针 `fp` 识别待处理的流，`buf` 指向待使用的存储区。如果 `buf` 的值不是 `NULL`，则必须创建一个缓冲区。例如，声明一个内含 1024 个字符的数组，并传递该数组的地址。然而，如果把 `NULL` 作为 `buf` 的值，该函数会为自己分配一个缓冲区。变量 `size` 告诉 `setvbuf()` 数组的大小 (`size_t` 是一种派生的整数类型，第 5 章介绍过)。`mode` 的选择如下：`_IOFBF` 表示完全缓冲（在缓冲区满时刷新）；`_IOLBF` 表示行缓冲（在缓冲区满时或写入一个换行符时）；`_IONBF` 表示无缓冲。如果操作成功，函数返回 0，否则返回一个非零值。

假设一个程序要储存一种数据对象，每个数据对象的大小是 3000 字节。可以使用 `setvbuf()` 函数创建一个缓冲区，其大小是该数据对象大小的倍数。

### 13.7.4 二进制 I/O：fread() 和 fwrite()

介绍 `fread()` 和 `fwrite()` 函数之前，先要了解一些背景知识。之前用到的标准 I/O 函数都是面向文本的，用于处理字符和字符串。如何要在文件中保存数值数据？用 `fprintf()` 函数和 `%f` 转换说明只是把数值保存为字符串。例如，下面的代码：

```
double num = 1./3.;
fprintf(fp, "%f", num);
```

把 `num` 储存为 8 个字符：0.333333。使用 `%.2f` 转换说明将其储存为 4 个字符：0.33，用 `%.12f` 转换说明则将其储存为 14 个字符：0.333333333333。改变转换说明将改变储存该值所需的空间数量，也会导致储存不同的值。把 `num` 储存为 0.33 后，读取文件时就无法将其恢复为更高的精度。一般而言，`fprintf()` 把数值转换为字符数据，这种转换可能会改变值。

为保证数值在储存前后一致，最精确的做法是使用与计算机相同的位组合来储存。因此，`double` 类型的值应该储存在一个 `double` 大小的单元中。如果以程序所用的表示法把数据储存在文件中，则称以二进制形式储存数据。不存在从数值形式到字符串的转换过程。对于标准 I/O，`fread()` 和 `fwrite` 函数用于以二进制形式处理数据（见图 13.3）。

实际上，所有的数据都是以二进制形式储存的，甚至连字符都以字符码的二进制表示来储存。如果文件中的所有数据都被解释成字符码，则称该文件包含文本数据。如果部分或所有的数据都被解释成二进制形式的数值数据，则称该文件包含二进制数据（另外，用数据表示机器语言指令的文件都是二进制文件）。

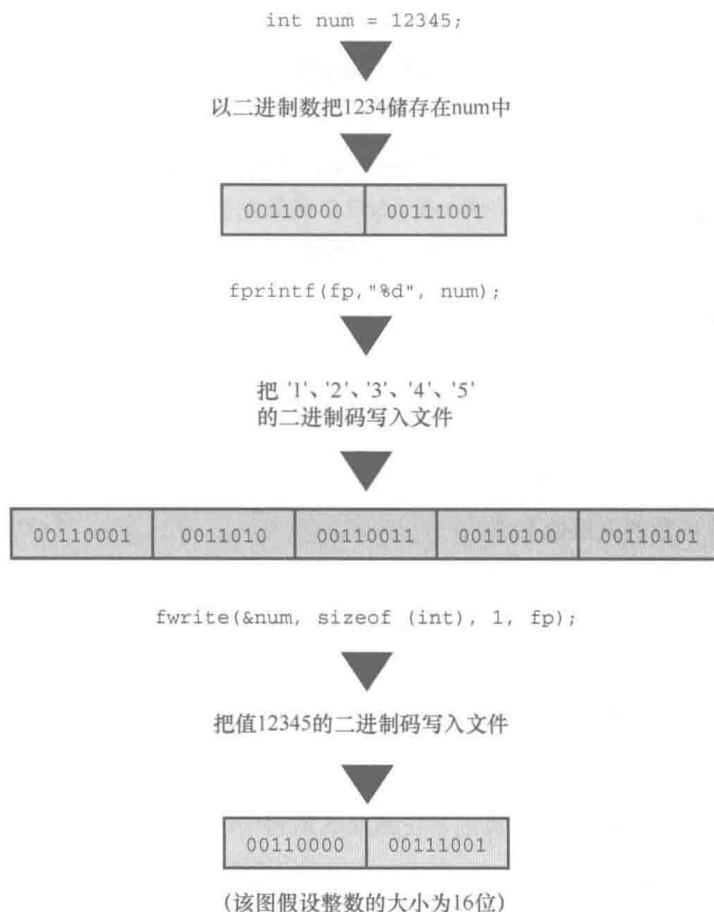


图 13.3 二进制输出和文本输出

二进制和文本的用法很容易混淆。ANSI C 和许多操作系统都识别两种文件格式：二进制和文本。能以二进制数据或文本数据形式存储或读取信息。可以用二进制模式打开文本格式的文件，可以把文本储存在二进制形式的文件中。可以调用 `getc()` 拷贝包含二进制数据的文件。然而，一般而言，用二进制模式在二进制格式文件中储存二进制数据。类似地，最常用的还是以文本格式打开文本文件中的文本数据（通常文字处理器生成的文件都是二进制文件，因为这些文件中包含了大量非文本信息，如字体和格式等）。

### 13.7.5 size\_t fwrite() 函数

`fwrite()` 函数的原型如下：

```
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb, FILE * restrict fp);
```

`fwrite()` 函数把二进制数据写入文件。`size_t` 是根据标准 C 类型定义的类型，它是 `sizeof` 运算符返回的类型，通常是 `unsigned int`，但是实现可以选择使用其他类型。指针 `ptr` 是待写入数据块的地址。`size` 表示待写入数据块的大小（以字节为单位），`nmemb` 表示待写入数据块的数量。和其他函数一样，`fp` 指定待写入的文件。例如，要保存一个大小为 256 字节的数据对象（如数组），可以这样做：

```
char buffer[256];
fwrite(buffer, 256, 1, fp);
```

以上调用把一块 256 字节的数据从 `buffer` 写入文件。另举一例，要保存一个内含 10 个 `double` 类型值的数组，可以这样做：

```
double earnings[10];
fwrite(earnings, sizeof(double), 10, fp);
```

以上调用把 `earnings` 数组中的数据写入文件，数据被分成 10 块，每块都是 `double` 的大小。

注意 `fwrite()` 原型中的 `const void * restrict ptr` 声明。`fwrite()` 的一个问题是，它的第 1 个参数不是固定的类型。例如，第 1 个例子中使用 `buffer`，其类型是指向 `char` 的指针；而第 2 个例子中使用 `earnings`，其类型是指向 `double` 的指针。在 ANSI C 函数原型中，这些实际参数都被转换成指向 `void` 的指针类型，这种指针可作为一种通用类型指针（在 ANSI C 之前，这些参数使用 `char *` 类型，需要把实参强制转换成 `char *` 类型）。

`fwrite()` 函数返回成功写入项的数量。正常情况下，该返回值就是 `nmemb`，但如果出现写入错误，返回值会比 `nmemb` 小。

### 13.7.6 size\_t fread() 函数

`size_t fread()` 函数的原型如下：

```
size_t fread(void * restrict ptr, size_t size, size_t nmemb, FILE * restrict fp);
```

`fread()` 函数接受的参数和 `fwrite()` 函数相同。在 `fread()` 函数中，`ptr` 是待读取文件数据在内存中的地址，`fp` 指定待读取的文件。该函数用于读取被 `fwrite()` 写入文件的数据。例如，要恢复上例中保存的内含 10 个 `double` 类型值的数组，可以这样做：

```
double earnings[10];
fread(earnings, sizeof(double), 10, fp);
```

该调用把 10 个 `double` 大小的值拷贝进 `earnings` 数组中。

`fread()` 函数返回成功读取项的数量。正常情况下，该返回值就是 `nmemb`，但如果出现读取错误或读到文件结尾，该返回值就会比 `nmemb` 小。

### 13.7.7 int feof(FILE \*fp) 和 int ferror(FILE \*fp) 函数

如果标准输入函数返回 `EOF`，则通常表明函数已到达文件结尾。然而，出现读取错误时，函数也会返回 `EOF`。`feof()` 和 `ferror()` 函数用于区分这两种情况。当上一次输入调用检测到文件结尾时，`feof()` 函数返回一个非零值，否则返回 0。当读或写出现错误，`ferror()` 函数返回一个非零值，否则返回 0。

### 13.7.8 一个程序示例

接下来，我们用一个程序示例说明这些函数的用法。该程序把一系列文件中的内容附加在另一个文件的末尾。该程序存在一个问题：如何给文件传递信息。可以通过交互或使用命令行参数来完成，我们先采用交互式的方法。下面列出了程序的设计方案。

- 询问目标文件的名称并打开它。
- 使用一个循环询问源文件。
- 以读模式依次打开每个源文件，并将其添加到目标文件的末尾。

为演示 `setvbuf()` 函数的用法，该程序将使用它指定一个不同的缓冲区大小。下一步是细化程序打开目标文件的步骤：

1. 以附加模式打开目标文件；
2. 如果打开失败，则退出程序；
3. 为该文件创建一个 4096 字节的缓冲区；

4. 如果创建失败，则退出程序。

与此类似，通过以下具体步骤细化拷贝部分：

1. 如果该文件与目标文件相同，则跳至下一个文件；
2. 如果以读模式无法打开文件，则跳至下一个文件；
3. 把文件内容添加至目标文件末尾。

最后，程序回到目标文件的开始处，显示当前整个文件的内容。

作为练习，我们使用 `fread()` 和 `fwrite()` 函数进行拷贝。程序清单 13.5 给出了这个程序。

#### 程序清单 13.5 append.c 程序

```
/* append.c -- 把文件附加到另一个文件末尾 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 4096
#define SLEN 81
void append(FILE *source, FILE *dest);
char * s_gets(char * st, int n);

int main(void)
{
 FILE *fa, *fs; // fa 指向目标文件, fs 指向源文件
 int files = 0; // 附加的文件数量
 char file_app[SLEN]; // 目标文件名
 char file_src[SLEN]; // 源文件名
 int ch;

 puts("Enter name of destination file:");
 s_gets(file_app, SLEN);
 if ((fa = fopen(file_app, "a+")) == NULL)
 {
 fprintf(stderr, "Can't open %s\n", file_app);
 exit(EXIT_FAILURE);
 }
 if (setvbuf(fa, NULL, _IOFBF, BUFSIZE) != 0)
 {
 fputs("Can't create output buffer\n", stderr);
 exit(EXIT_FAILURE);
 }
 puts("Enter name of first source file (empty line to quit):");
 while (s_gets(file_src, SLEN) && file_src[0] != '\0')
 {
 if (strcmp(file_src, file_app) == 0)
 fputs("Can't append file to itself\n", stderr);
 else if ((fs = fopen(file_src, "r")) == NULL)
 fprintf(stderr, "Can't open %s\n", file_src);
 else
 {
 if (setvbuf(fs, NULL, _IOFBF, BUFSIZE) != 0)
 {
 fputs("Can't create input buffer\n", stderr);
 continue;
 }
 // 从 fs 读取数据并写入 fa
 // ...
 }
 }
}
```

```

 }
 append(fs, fa);
 if (ferror(fs) != 0)
 fprintf(stderr, "Error in reading file %s.\n",
 file_src);
 if (ferror(fa) != 0)
 fprintf(stderr, "Error in writing file %s.\n",
 file_app);
 fclose(fs);
 files++;
 printf("File %s appended.\n", file_src);
 puts("Next file (empty line to quit):");
}
}

printf("Done appending. %d files appended.\n", files);
rewind(fa);
printf("%s contents:\n", file_app);
while ((ch = getc(fa)) != EOF)
 putchar(ch);
puts("Done displaying.");
fclose(fa);

return 0;
}

void append(FILE *source, FILE *dest)
{
 size_t bytes;
 static char temp[BUFSIZE]; // 只分配一次

 while ((bytes = fread(temp, sizeof(char), BUFSIZE, source)) > 0)
 fwrite(temp, sizeof(char), bytes, dest);
}

char * s_gets(char * st, int n)
{
 char * ret_val;
 char * find;

 ret_val = fgets(st, n, stdin);
 if (ret_val)
 {
 find = strchr(st, '\n'); // 查找换行符
 if (find) // 如果地址不是 NULL,
 *find = '\0'; // 在此处放置一个空字符
 else
 while (getchar() != '\n')
 continue;
 }
 return ret_val;
}

```

如果 `setvbuf()` 无法创建缓冲区，则返回一个非零值，然后终止程序。可以用类似的代码为正在拷贝的文件创建一块 4096 字节的缓冲区。把 `NULL` 作为 `setvbuf()` 的第 2 个参数，便可让函数分配缓冲区的

存储空间。

该程序获取文件名所用的函数是 `s_gets()`，而不是 `scanf()`，因为 `scanf()` 会跳过空白，因此无法检测到空行。该程序还用 `s_gets()` 代替 `fgets()`，因为后者在字符串中保留换行符。

以下代码防止程序把文件附加在自身末尾：

```
if (strcmp(file_src, file_app) == 0)
 fputs("Can't append file to itself\n", stderr);
```

参数 `file_app` 表示目标文件名，`file_src` 表示正在处理的文件名。

`append()` 函数完成拷贝任务。该函数使用 `fread()` 和 `fwrite()` 一次拷贝 4096 字节，而不是一次拷贝 1 字节：

```
void append(FILE *source, FILE *dest)
{
 size_t bytes;
 static char temp[BUFSIZE]; // 只分配一次
 while ((bytes = fread(temp, sizeof(char), BUFSIZE, source)) > 0)
 fwrite(temp, sizeof(char), bytes, dest);
}
```

因为是以附加模式打开由 `dest` 指定的文件，所以所有的源文件都被依次添加至目标文件的末尾。注意，`temp` 数组具有静态存储期（意思是在编译时分配该数组，不是在每次调用 `append()` 函数时分配）和块作用域（意思是该数组属于它所在的函数私有）。

该程序示例使用文本模式的文件。使用“`ab+`”和“`rb`”模式可以处理二进制文件。

### 13.7.9 用二进制 I/O 进行随机访问

随机访问是用二进制 I/O 写入二进制文件最常用的方式，我们来看一个简短的例子。程序清单 13.6 中的程序创建了一个储存 `double` 类型数字的文件，然后让用户访问这些内容。

程序清单 13.6 `randbin.c` 程序

---

```
/* randbin.c -- 用二进制 I/O 进行随机访问 */
#include <stdio.h>
#include <stdlib.h>
#define ARSIZE 1000

int main()
{
 double numbers[ARSIZE];
 double value;
 const char * file = "numbers.dat";
 int i;
 long pos;
 FILE *iofile;

 // 创建一组 double 类型的值
 for (i = 0; i < ARSIZE; i++)
 numbers[i] = 100.0 * i + 1.0 / (i + 1);
 // 尝试打开文件
 if ((iofile = fopen(file, "wb")) == NULL)
 {
 fprintf(stderr, "Could not open %s for output.\n", file);
 exit(EXIT_FAILURE);
 }
```

```

}

// 以二进制格式把数组写入文件
fwrite(numbers, sizeof(double), ARSIZE, iofile);
fclose(iofile);
if ((iofile = fopen(file, "rb")) == NULL)
{
 fprintf(stderr,
 "Could not open %s for random access.\n", file);
 exit(EXIT_FAILURE);
}

// 从文件中读取选定的内容
printf("Enter an index in the range 0-%d.\n", ARSIZE - 1);
while (scanf("%d", &i) == 1 && i >= 0 && i < ARSIZE)
{
 pos = (long) i * sizeof(double); // 计算偏移量
 fseek(iofile, pos, SEEK_SET); // 定位到此处
 fread(&value, sizeof(double), 1, iofile);
 printf("The value there is %f.\n", value);
 printf("Next index (out of range to quit):\n");
}
// 完成
fclose(iofile);
puts("Bye!");

return 0;
}

```

首先，该程序创建了一个数组，并在该数组中存放了一些值。然后，程序以二进制模式创建了一个名为 numbers.dat 的文件，并使用 `fwrite()` 把数组中的内容拷贝到文件中。内存中数组的所有 double 类型值的位组合（每个位组合都是 64 位）都被拷贝至文件中。不能用文本编辑器读取最后的二进制文件，因为无法把文件中的值转换成字符串。然而，储存在文件中的每个值都与储存在内存中的值完全相同，没有损失任何精确度。此外，每个值在文件中也同样占用 64 位存储空间，所以可以很容易地计算出每个值的位置。

程序的第 2 部分用于打开待读取的文件，提示用户输入一个值的索引。程序通过把索引值和 double 类型值占用的字节相乘，即可得出文件中的一个位置。然后，程序调用 `fseek()` 定位到该位置，用 `fread()` 读取该位置上的数据值。注意，这里并未使用转换说明。`fread()` 从已定位的位置开始，拷贝 8 字节到内存中地址为`&value` 的位置。然后，使用 `printf()` 显示 `value`。下面是该程序的一个运行示例：

```

Enter an index in the range 0-999.
500
The value there is 50000.001996.
Next index (out of range to quit):
900
The value there is 90000.001110.
Next index (out of range to quit):
0
The value there is 1.000000.
Next index (out of range to quit):
-1
Bye!

```

## 13.8 关键概念

C 程序把输入看作是字节流，输入流来源于文件、输入设备（如键盘），或者甚至是另一个程序的输出。类似地，C 程序把输出也看作是字节流，输出流的目的地可以是文件、视频显示等。

C 如何解释输入流或输出流取决于所使用的输入/输出函数。程序可以不做任何改动地读取和存储字节，或者把字节依次解释成字符，随后可以把这些字符解释成普通文本以用文本表示数字。类似地，对于输出，所使用的函数决定了二进制值是被原样转移，还是被转换成文本或以文本表示数字。如果要在不损失精度的前提下保存或恢复数值数据，请使用二进制模式以及 `fread()` 和 `fwrite()` 函数。如果打算保存文本信息并创建能在普通文本编辑器查看的文本，请使用文本模式和函数（如 `getc()` 和 `fprintf()`）。

要访问文件，必须创建文件指针（类型是 `FILE *`）并把指针与特定文件名相关联。随后的代码就可以使用这个指针（而不是文件名）来处理该文件。

要重点理解 C 如何处理文件结尾。通常，用于读取文件的程序使用一个循环读取输入，直至到达文件结尾。C 输入函数在读过文件结尾后才会检测到文件结尾，这意味着应该在尝试读取之后立即判断是否是文件结尾。可以使用 13.2.4 节中“设计范例”中的双文件输入模式。

## 13.9 本章小结

对于大多数 C 程序而言，写入文件和读取文件必不可少。为此，绝大部分 C 实现都提供底层 I/O 和标准高级 I/O。因为 ANSI C 库考虑到可移植性，包含了标准 I/O 包，但是未提供底层 I/O。

标准 I/O 包自动创建输入和输出缓冲区以加快数据传输。`fclose()` 函数为标准 I/O 打开一个文件，并创建一个用于存储文件和缓冲区信息的结构。`fclose()` 函数返回指向该结构的指针，其他函数可以使用该指针指定待处理的文件。`feof()` 和 `ferror()` 函数报告 I/O 操作失败的原因。

C 把输入视为字节流。如果使用 `fread()` 函数，C 把输入看作是二进制值并将其储存在指定存储位置。如果使用 `fscanf()`、`getc()`、`fgets()` 或其他相关函数，C 则将每个字节看作是字符码。然后 `fscanf()` 和 `scanf()` 函数尝试把字符码翻译成转换说明指定的其他类型。例如，输入一个值 23，%f 转换说明会把 23 翻译成一个浮点值，%d 转换说明会把 23 翻译成一个整数值，%s 转换说明则会把 23 储存为字符串。`getc()` 和 `fgetc()` 系列函数把输入作为字符码储存，将其作为单独的字符保存在字符变量中或作为字符串储存在字符数组中。类似地，`fwrite()` 将二进制数据直接放入输出流，而其他输出函数把非字符数据转换成用字符表示后才将其放入输出流。

ANSI C 提供两种文件打开模式：二进制和文本。以二进制模式打开文件时，可以逐字节读取文件；以文本模式打开文件时，会把文件内容从文本的系统表示法映射为 C 表示法。对于 UNIX 和 Linux 系统，这两种模式完全相同。

通常，输入函数 `getc()`、`fgets()`、`fscanf()` 和 `fread()` 都从文件开始处按顺序读取文件。然而，`fseek()` 和 `ftell()` 函数让程序可以随机访问文件中的任意位置。`fgetpos()` 和 `fsetpos()` 把类似的功能扩展至更大的文件。与文本模式相比，二进制模式更容易进行随机访问。

## 13.10 复习题

复习题的参考答案在附录 A 中。

- 下面的程序有什么问题？

```
int main(void)
{
```

```

int * fp;
int k;

fp = fopen("gelatin");
for (k = 0; k < 30; k++)
 fputs(fp, "Nanette eats gelatin.");
fclose("gelatin");
return 0;
}

```

2. 下面的程序完成什么任务？（假设在命令行环境中运行）

```

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main(int argc, char *argv[])
{
 int ch;
 FILE *fp;
 if (argc < 2)
 exit(EXIT_FAILURE);
 if ((fp = fopen(argv[1], "r")) == NULL)
 exit(EXIT_FAILURE);
 while ((ch = getc(fp)) != EOF)
 if (isdigit(ch))
 putchar(ch);
 fclose(fp);

 return 0;
}

```

3. 假设程序中有下列语句：

```

#include <stdio.h>
FILE * fp1,* fp2;
char ch;

fp1 = fopen("terky", "r");
fp2 = fopen("jerky", "w");

```

另外，假设成功打开了两个文件。补全下面函数调用中缺少的参数：

- a. ch = getc();
  - b. fprintf( , "%c\n", );
  - c. putc( , );
  - d. fclose(); /\* 关闭 terky 文件 \*/
4. 编写一个程序，不接受任何命令行参数或接受一个命令行参数。如果有一个参数，将其解释为文件名；如果没有参数，使用标准输入（stdin）作为输入。假设输入完全是浮点数。该程序要计算和报告输入数字的算术平均值。
5. 编写一个程序，接受两个命令行参数。第 1 个参数是字符，第 2 个参数是文件名。要求该程序只打印文件中包含给定字符的那些行。

### 注意

C 程序根据'\n'识别文件中的行。假设所有行都不超过 256 个字符，你可能会想到用 fgets()。

6. 二进制文件和文本文件有何区别？二进制流和文本流有何区别？
7.
  - a. 分别用 `fprintf()` 和 `fwrite()` 储存 8238201 有何区别？
  - b. 分别用 `putc()` 和 `fwrite()` 储存字符 S 有何区别？
8. 下面语句的区别是什么？
 

```
printf("Hello, %s\n", name);
fprintf(stdout, "Hello, %s\n", name);
fprintf(stderr, "Hello, %s\n", name);
```
9. “`a+`”、“`r+`”和“`w+`”模式打开的文件都是可读写的。哪种模式更适合用来更改文件中已有的内容？

## 13.11 编程练习

1. 修改程序清单 13.1 中的程序，要求提示用户输入文件名，并读取用户输入的信息，不使用命令行参数。
2. 编写一个文件拷贝程序，该程序通过命令行获取原始文件名和拷贝文件名。尽量使用标准 I/O 和二进制模式。
3. 编写一个文件拷贝程序，提示用户输入文本文件名，并以该文件名作为原始文件名和输出文件名。该程序要使用 `ctype.h` 中的 `toupper()` 函数，在写入到输出文件时把所有文本转换成大写。使用标准 I/O 和文本模式。
4. 编写一个程序，按顺序在屏幕上显示命令行中列出的所有文件。使用 `argc` 控制循环。
5. 修改程序清单 13.5 中的程序，用命令行界面代替交互式界面。
6. 使用命令行参数的程序依赖于用户的内存如何正确地使用它们。重写程序清单 13.2 中的程序，不使用命令行参数，而是提示用户输入所需信息。
7. 编写一个程序打开两个文件。可以使用命令行参数或提示用户输入文件名。
  - a. 该程序以这样的顺序打印：打印第 1 个文件的第 1 行，第 2 个文件的第 1 行，第 1 个文件的第 2 行，第 2 个文件的第 2 行，以此类推，打印到行数较多文件的最后一行。
  - b. 修改该程序，把行号相同的行打印成一行。
8. 编写一个程序，以一个字符和任意文件名作为命令行参数。如果字符后面没有参数，该程序读取标准输入；否则，程序依次打开每个文件并报告每个文件中该字符出现的次数。文件名和字符本身也要一同报告。程序应包含错误检查，以确定参数数量是否正确和是否能打开文件。如果无法打开文件，程序应报告这一情况，然后继续处理下一个文件。
9. 修改程序清单 13.3 中的程序，从 1 开始，根据加入列表的顺序为每个单词编号。当程序下次运行时，确保新的单词编号接着上次的编号开始。
10. 编写一个程序打开一个文本文件，通过交互方式获得文件名。通过一个循环，提示用户输入一个文件位置。然后该程序打印从该位置开始到下一个换行符之前的内容。用户输入负数或非数值字符可以结束输入循环。
11. 编写一个程序，接受两个命令行参数。第 1 个参数是一个字符串，第 2 个参数是一个文件名。然后该程序查找该文件，打印文件中包含该字符串的所有行。因为该任务是面向行而不是面向字符的，所以要使用 `fgets()` 而不是 `getc()`。使用标准 C 库函数 `strstr()`（11.5.7 节简要介绍过）在每一行中查找指定字符串。假设文件中的所有行都不超过 255 个字符。

12. 创建一个文本文件，内含 20 行，每行 30 个整数。这些整数都在 0~9 之间，用空格分开。该文件是用数字表示一张图片，0~9 表示逐渐增加的灰度。编写一个程序，把文件中的内容读入一个  $20 \times 30$  的 int 数组中。一种把这些数字转换为图片的粗略方法是：该程序使用数组中的值初始化一个  $20 \times 31$  的字符数组，用值 0 对应空格字符，1 对应点字符，以此类推。数字越大表示字符所占的空间越大。例如，用# 表示 9。每行的最后一个字符（第 31 个）是空字符，这样该数组包含了 20 个字符串。最后，程序显示最终的图片（即，打印所有的字符串），并将结果储存在文本文件中。例如，下面是开始的数据：

根据以上描述选择特定的输出字符，最终输出如下：

13. 用变长数组 (VLA) 代替标准数组, 完成编程练习 12。
  14. 数字图像, 尤其是从宇宙飞船发回的数字图像, 可能会包含一些失真。为编程练习 12 添加消除失真的函数。该函数把每个值与它上下左右相邻的值作比较, 如果该值与其周围相邻值的差都大于 1, 则用所有相邻值的平均值 (四舍五入为整数) 代替该值。注意, 与边界上的点相邻的点少于 4 个, 所以做特殊处理。