

# 第 14 章

## 结构和其他数据形式

本章介绍以下内容：

- 关键字：struct、union、typedef
- 运算符：.、->
- 什么是 C 结构，如何创建结构模板和结构变量
- 如何访问结构的成员，如何编写处理结构的函数
- 联合和指向函数的指针

设计程序时，最重要的步骤之一是选择表示数据的方法。在许多情况下，简单变量甚至是数组还不够。为此，C 提供了结构变量（structure variable）提高你表示数据的能力，它能让你创造新的形式。如果熟悉 Pascal 的记录（record），应该很容易理解结构。如果不懂 Pascal 也没关系，本章将详细介绍 C 结构。我们先通过一个示例来分析为何需要 C 结构，学习如何创建和使用结构。

### 14.1 示例问题：创建图书目录

Gwen Glenn 要打印一份图书目录。她想打印每本书的各种信息：书名、作者、出版社、版权日期、页数、册数和价格。其中的一些项目（如，书名）可以储存在字符数组中，其他项目需要一个 int 数组或 float 数组。用 7 个不同的数组分别记录每一项比较繁琐，尤其是 Gwen 还想创建多份列表：一份按书名排序、一份按作者排序、一份按价格排序等。如果能把图书目录的信息都包含在一个数组里更好，其中每个元素包含一本书的相关信息。

因此，Gwen 需要一种既能包含字符串又能包含数字的数据形式，而且还要保持各信息的独立。C 结构就满足这种情况下需求。我们通过一个示例演示如何创建和使用数组。但是，示例进行了一些限制。第一，该程序示例演示的书目只包含书名、作者和价格。第二，只有一本书的数目。当然，别忘了这只是进行了限制，我们在后面将扩展该程序。请看程序清单 14.1 及其输出，然后阅读后面的一些要点。

程序清单 14.1 book.c 程序

```
/* book.c -- 一本书的图书目录 */
#include <stdio.h>
#include <string.h>
char * s_gets(char * st, int n);
#define MAXTITL 41    /* 书名的最大长度 + 1      */
#define MAXAUTL 31    /* 作者姓名的最大长度 + 1 */

struct book {          /* 结构模版：标记是 book */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
```

```

}; /* 结构模版结束 */

int main(void)
{
    struct book library; /* 把 library 声明为一个 book 类型的变量 */

    printf("Please enter the book title.\n");
    s_gets(library.title, MAXTITL); /* 访问 title 部分 */
    printf("Now enter the author.\n");
    s_gets(library.author, MAXAUTL);
    printf("Now enter the value.\n");
    scanf("%f", &library.value);
    printf("%s by %s: $%.2f\n", library.title,
           library.author, library.value);
    printf("%s: \"%s\" ($%.2f)\n", library.author,
           library.title, library.value);
    printf("Done.\n");

    return 0;
}

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; // 处理输入行中剩余的字符
    }
    return ret_val;
}

```

我们使用前面章节中介绍的 `s_gets()` 函数去掉 `fgets()` 储存在字符串中的换行符。下面是该例的一个运行示例：

```

Please enter the book title.
Chicken of the Andes
Now enter the author.
Disma Lapoult
Now enter the value.
29.99
Chicken of the Andes by Disma Lapoult: $29.99
Disma Lapoult: "Chicken of the Andes" ($29.99)
Done.

```

程序清单 14.1 中创建的结构有 3 部分，每个部分都称为成员（*member*）或字段（*field*）。这 3 部分中，一部分储存书名，一部分储存作者名，一部分储存价格。下面是必须掌握的 3 个技巧：

- 为结构建立一个格式或样式；
- 声明一个适合该样式的变量；
- 访问结构变量的各个部分。

## 14.2 建立结构声明

结构声明 (*structure declaration*) 描述了一个结构的组织布局。声明类似下面这样：

```
struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
```

该声明描述了一个由两个字符数组和一个 `float` 类型变量组成的结构。该声明并未创建实际的数据对象，只描述了该对象由什么组成。（有时，我们把结构声明称为模板，因为它勾勒出结构是如何储存数据的。如果读者知道 C++ 的模板，此模板非彼模板，C++ 中的模板更为强大。）我们来分析一些细节。首先是关键字 `struct`，它表明跟在其后的是一个结构，后面是一个可选的标记（该例中是 `book`），稍后程序中可以使用该标记引用该结构。所以，我们在后面的程序中可以这样声明：

```
struct book library;
```

这把 `library` 声明为一个使用 `book` 结构布局的结构变量。

在结构声明中，用一对花括号括起来的是结构成员列表。每个成员都用自己的声明来描述。例如，`title` 部分是一个内含 `MAXTITL` 个元素的 `char` 类型数组。成员可以是任意一种 C 的数据类型，甚至可以是其他结构！右花括号后面的分号是声明所必需的，表示结构布局定义结束。可以把这个声明放在所有函数的外部（如本例所示），也可以放在一个函数定义的内部。如果把结构声明置于一个函数的内部，它的标记就只限于该函数内部使用。如果把结构声明置于函数的外部，那么该声明之后的所有函数都能使用它的标记。例如，在程序的另一个函数中，可以这样声明：

```
struct book dickens;
```

这样，该函数便创建了一个结构变量 `dickens`，该变量的结构布局是 `book`。

结构的标记名是可选的。但是以程序示例中的方式建立结构时（在一处定义结构布局，在另一处定义实际的结构变量），必须使用标记。我们学完如何定义结构变量后，再来看这一点。

## 14.3 定义结构变量

结构有两层含义。一层含义是“结构布局”，刚才已经讨论过了。结构布局告诉编译器如何表示数据，但是它并未让编译器为数据分配空间。下一步是创建一个结构变量，即是结构的另一层含义。程序中创建结构变量的一行是：

```
struct book library;
```

编译器执行这行代码便创建了一个结构变量 `library`。编译器使用 `book` 模板为该变量分配空间：一个内含 `MAXTITL` 个元素的 `char` 数组、一个内含 `MAXAUTL` 个元素的 `char` 数组和一个 `float` 类型的变量。这些存储空间都与一个名称 `library` 结合在一起（见图 14.1）。

在结构变量的声明中，`struct book` 所起的作用相当于一般声明中的 `int` 或 `float`。例如，可以定义两个 `struct book` 类型的变量，或者甚至是指向 `struct book` 类型结构的指针：

```
struct book doyle, panshin, * ptbook;
```

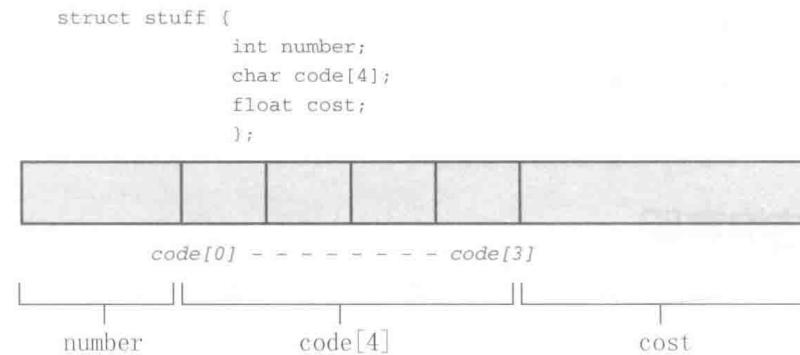


图 14.1 一个结构的内存分配

结构变量 doyle 和 pashin 中都包含 title、author 和 value 部分。指针 ptbook 可以指向 doyle、pashin 或任何其他 book 类型的结构变量。从本质上讲，book 结构声明创建了一个名为 struct book 的新类型。

就计算机而言，下面的声明：

```
struct book library;
```

是以下声明的简化：

```

struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
} library; /* 声明的右花括号后跟变量名 */

```

换言之，声明结构的过程和定义结构变量的过程可以组合成一个步骤。如下所示，组合后的结构声明和结构变量定义不需要使用结构标记：

```

struct { /* 无结构标记 */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
} library;

```

然而，如果打算多次使用结构模板，就要使用带标记的形式；或者，使用本章后面介绍的 `typedef`。

这是定义结构变量的一个方面，在这个例子中，并未初始化结构变量。

### 14.3.1 初始化结构

初始化变量和数组如下：

```
int count = 0;
int fibo[7] = {0,1,1,2,3,5,8};
```

结构变量是否也可以这样初始化？是的，可以。初始化一个结构变量（ANSI 之前，不能用自动变量初始化结构；ANSI 之后可以用任意存储类别）与初始化数组的语法类似：

```

struct book library = {
    "The Pious Pirate and the Devious Damsel",
    "Renee Vivotte",
    1.95
};

```

简而言之，我们使用在一对花括号中括起来的初始化列表进行初始化，各初始化项用逗号分隔。因此，`title` 成员可以被初始化为一个字符串，`value` 成员可以被初始化为一个数字。为了让初始化项与结构中

各成员的关联更加明显，我们让每个成员的初始化项独占一行。这样做只是为了提高代码的可读性，对编译器而言，只需要用逗号分隔各成员的初始化项即可。

### 注意 初始化结构和类别储存期

第 12 章中提到过，如果初始化静态存储期的变量（如，静态外部链接、静态内部链接或静态无链接），必须使用常量值。这同样适用于结构。如果初始化一个静态存储期的结构，初始化列表中的值必须是常量表达式。如果是自动存储期，初始化列表中的值可以不是常量。

## 14.3.2 访问结构成员

结构类似于一个“超级数组”，这个超级数组中，可以是一个元素为 `char` 类型，下一个元素为 `float` 类型，下一个元素为 `int` 数组。可以通过数组下标单独访问数组中的各元素，那么，如何访问结构中的成员？使用结构成员运算符——点`(.)`访问结构中的成员。例如，`library.value` 即访问 `library` 的 `value` 部分。可以像使用任何 `float` 类型变量那样使用 `library.value`。与此类似，可以像使用字符数组那样使用 `library.title`。因此，程序清单 14.1 中的程序中有 `s_gets(library.title, MAXTITL);` 和 `scanf("%f", &library.value);` 这样的代码。

本质上，`.title`、`.author` 和 `.value` 的作用相当于 `book` 结构的下标。

注意，虽然 `library` 是一个结构，但是 `library.value` 是一个 `float` 类型的变量，可以像使用其他 `float` 类型变量那样使用它。例如，`scanf("%f", ...)` 需要一个 `float` 类型变量的地址，而 `&library.float` 正好符合要求。`.` 比 `&` 的优先级高，因此这个表达式和 `&(library.float)` 一样。

如果还有一个相同类型的结构变量，可以用相同的方法：

```
struct book bill, newt;

s_gets(bill.title, MAXTITL);
s_gets(newt.title, MAXTITL);
```

`.title` 引用 `book` 结构的第 1 个成员。注意，程序清单 14.1 中的程序以两种不同的格式打印了 `library` 结构变量中的内容。这说明可以自行决定如何使用结构成员。

## 14.3.3 结构的初始化器

C99 和 C11 为结构提供了指定初始化器 (*designated initializer*)<sup>1</sup>，其语法与数组的指定初始化器类似。但是，结构的指定初始化器使用点运算符和成员名（而不是方括号和下标）标识特定的元素。例如，只初始化 `book` 结构的 `value` 成员，可以这样做：

```
struct book surprise = { .value = 10.99};
```

可以按照任意顺序使用指定初始化器：

```
struct book gift = { .value = 25.99,
                     .author = "James Broadfool",
                     .title = "Rue for the Toad"};
```

与数组类似，在指定初始化器后面的普通初始化器，为指定成员后面的成员提供初始值。另外，对特定成员的最后一次赋值才是它实际获得的值。例如，考虑下面的代码：

<sup>1</sup> 也被称为标记化结构初始化语法。——译者注

```
struct book gift= { .value = 18.90,
                    .author = "Philionna Pestle",
                    0.25};
```

赋给 value 的值是 0.25，因为它在结构声明中紧跟在 author 成员之后。新值 0.25 取代了之前的 18.9。在学习了结构的基本知识后，可以进一步了解结构的一些相关类型。

## 14.4 结构数组

接下来，我们要把程序清单 14.1 的程序扩展成可以处理多本书。显然，每本书的基本信息都可以用一个 book 类型的结构变量来表示。为描述两本书，需要使用两个变量，以此类推。可以使用这一类型的结构数组来处理多本书。在下一个程序中（程序清单 14.2）就创建了一个这样的数组。如果你使用 Borland C/C++，请参阅本节后面的“Borland C 和浮点数”。

### 结构和内存

manybook.c 程序创建了一个内含 100 个结构变量的数组。由于该数组是自动存储类别的对象，其中的信息被储存在栈（stack）中。如此大的数组需要很大一块内存，这可能会导致一些问题。如果在运行时出现错误，可能抱怨栈大小或栈溢出，你的编译器可能使用了一个默认大小的栈，这个栈对于该例而言太小。要修正这个问题，可以使用编译器选项设置栈大小为 10000，以容纳这个结构数组；或者可以创建静态或外部数组（这样，编译器就不会把数组放在栈中）；或者可以减小数组大小为 16。为何不一开始就使用较小的数组？这是为了让读者意识到栈大小的潜在问题，以便今后再遇到类似的问题，可以自己处理好。

程序清单 14.2 manybook.c 程序

```
/* manybook.c -- 包含多本书的图书目录 */
#include <stdio.h>
#include <string.h>
char * s_gets(char * st, int n);
#define MAXTITL 40
#define MAXAUTL 40
#define MAXBKS 100      /* 书籍的最大数量 */

struct book {           /* 简历 book 模板      */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book library[MAXBKS]; /* book 类型结构的数组 */
    int count = 0;
    int index;

    printf("Please enter the book title.\n");
    printf("Press [enter] at the start of a line to stop.\n");
    while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
        && library[count].title[0] != '\0')
    {
        printf("Now enter the author.\n");
        s_gets(library[count].author, MAXAUTL);
        library[count].value = 0.0;
        count++;
    }
}
```

```

s_gets(library[count].author, MAXAUTL);
printf("Now enter the value.\n");
scanf("%f", &library[count++].value);
while (getchar() != '\n')
    continue; /* 清理输入行 */
if (count < MAXBKS)
    printf("Enter the next title.\n");
}

if (count > 0)
{
    printf("Here is the list of your books:\n");
    for (index = 0; index < count; index++)
        printf("%s by %s: $%.2f\n", library[index].title,
               library[index].author, library[index].value);
}
else
    printf("No books? Too bad.\n");

return 0;
}

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; // 处理输入行中剩余的字符
    }
    return ret_val;
}

```

下面是该程序的一个输出示例：

```

Please enter the book title.
Press [enter] at the start of a line to stop.
My Life as a Budgie
Now enter the author.
Mack Zackles
Now enter the value.
12.95
Enter the next title.
... (此处省略了许多内容) ...
Here is the list of your books:
My Life as a Budgie by Mack Zackles: $12.95
Thought and Unthought Rethought by Kindra Schlagmeyer: $43.50
Concerto for Financial Instruments by Filmore Walletz: $49.99
The CEO Power Diet by Buster Downsize: $19.25

```

```
C++ Primer Plus by Stephen Prata: $59.99
Fact Avoidance: Perception as Reality by Polly Bull: $19.97
Coping with Coping by Dr. Rubin Thonkawker: $0.02
Diaphanous Frivolity by Neda McFey: $29.99
Murder Wore a Bikini by Mickey Splat: $18.95
A History of Buvania, Volume 8, by Prince Nikoli Buvan: $50.04
Mastering Your Digital Watch, 5nd Edition, by Miklos Mysz: $28.95
A Foregone Confusion by Phalty Reasoner: $5.99
Outsourcing Government: Selection vs. Election by Ima Pundit: $33.33
```

## Borland C 和浮点数

如果程序不使用浮点数，旧式的 Borland C 编译器会尝试使用小版本的 `scanf()` 来压缩程序。然而，如果在一个结构数组中只有一个浮点值（如程序清单 14.2 中那样），那么这种编译器（DOS 的 Borland C/C++ 3.1 之前的版本，不是 Borland C/C++ 4.0）就无法发现它存在。结果，编译器会生成如下消息：

```
scanf : floating point formats not linked
Abnormal program termination
```

一种解决方案是在程序中添加下面的代码：

```
#include <math.h>
double dummy = sin(0.0);
```

这段代码强制编译器载入浮点版本的 `scanf()`。

首先，我们学习如何声明结构数组和如何访问数组中的结构成员。然后，着重分析该程序的两个方面。

### 14.4.1 声明结构数组

声明结构数组和声明其他类型的数组类似。下面是一个声明结构数组的例子：

```
struct book library[MAXBKS];
```

以上代码把 `library` 声明为一个内含 `MAXBKS` 个元素的数组。数组的每个元素都是一个 `book` 类型的数组。因此，`library[0]` 是第 1 个 `book` 类型的结构变量，`library[1]` 是第 2 个 `book` 类型的结构变量，以此类推。参看图 14.2 可以帮助读者理解。数组名 `library` 本身不是结构名，它是一个数组名，该数组中的每个元素都是 `struct book` 类型的结构变量。

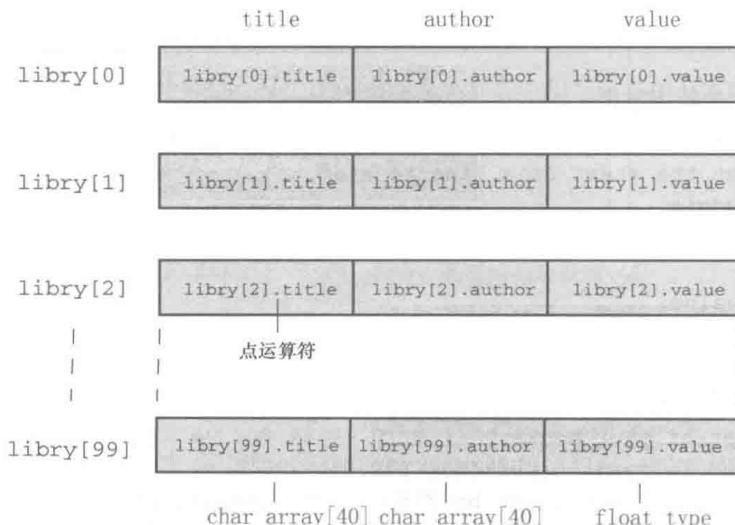


图 14.2 一个结构数组 `library[MAXBKS]`

## 14.4.2 标识结构数组的成员

为了标识结构数组中的成员，可以采用访问单独结构的规则：在结构名后面加一个点运算符，再在点运算符后面写上成员名。如下所示：

```
library[0].value /* 第 1 个数组元素与 value 相关联 */
library[4].title /* 第 5 个数组元素与 title 相关联 */
```

注意，数组下标紧跟在 library 后面，不是成员名后面：

```
library.value[2] // 错误
library[2].value // 正确
```

使用 library[2].value 的原因是：library[2]是结构变量名，正如 library[1]是另一个变量名。

顺带一提，下面的表达式代表什么？

```
library[2].title[4]
```

这是 library 数组第 3 个结构变量（library[2]部分）中书名的第 5 个字符（title[4]部分）。以程序清单 14.2 的输出为例，这个字符是 e。该例指出，点运算符右侧的下标作用于各个成员，点运算符左侧的下标作用与结构数组。

最后，总结一下：

```
library           // 一个 book 结构的数组
library[2]        // 一个数组元素，该元素是 book 结构
library[2].title  // 一个 char 数组 (library[2] 的 title 成员)
library[2].title[4] // 数组中 library[2] 元素的 title 成员的一个字符
```

下面，我们来讨论一下这个程序。

## 14.4.3 程序讨论

较之程序清单 14.1，该程序主要的改动之处是：插入一个 while 循环读取多个项。该循环的条件测试是：

```
while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
      && library[count].title[0] != '\0')
```

表达式 s\_gets(library[count].title, MAXTITL) 读取一个字符串作为书名，如果 s\_gets() 尝试读到文件结尾后面，该表达式则返回 NULL。表达式 library[count].title[0] != '\0' 判断字符串中的首字符是否是空字符（即，该字符串是否是空字符串）。如果在一行开始处用户按下 Enter 键，相当于输入了一个空字符串，循环将结束。程序中还检查了图书的数量，以免超出数组的大小。

然后，该程序中有如下几行：

```
while (getchar() != '\n')
    continue; /* 清理输入行 */
```

前面章节介绍过，这段代码弥补了 scanf() 函数遇到空格和换行符就结束读取的问题。当用户输入书的价格时，可能输入如下信息：

```
12.50[Enter]
```

其传送的字符序列如下：

```
12.50\n
```

scanf() 函数接受 1、2、..、5 和 0，但是把\n 留在输入序列中。如果没有上面两行清理输入行的代码，就会把留在输入序列中的换行符当作空行读入，程序以为用户发送了停止输入的信号。我们插入的这

两行代码只会在输入序列中查找并删除\n，不会处理其他字符。这样 `s_gets()` 就可以重新开始下一次输入。

## 14.5 嵌套结构

有时，在一个结构中包含另一个结构（即嵌套结构）很方便。例如，Shalala Pirosky 创建了一个有关她朋友信息的结构。显然，结构中需要一个成员表示朋友的姓名。然而，名字可以用一个数组来表示，其中包含名和姓这两个成员。程序清单 14.3 是一个简单的示例。

程序清单 14.3 friend.c 程序

```
// friend.c -- 嵌套结构示例
#include <stdio.h>
#define LEN 20
const char * msgs[5] =
{
    "    Thank you for the wonderful evening, ",
    "You certainly prove that a ",
    "is a special kind of guy. We must get together",
    "over a delicious ",
    " and have a few laughs"
};

struct names {           // 第 1 个结构
    char first[LEN];
    char last[LEN];
};

struct guy {             // 第 2 个结构
    struct names handle; // 嵌套结构
    char favfood[LEN];
    char job[LEN];
    float income;
};

int main(void)
{
    struct guy fellow = {           // 初始化一个结构变量
        {"Ewen", "Villard" },
        "grilled salmon",
        "personality coach",
        68112.00
    };

    printf("Dear %s, \n\n", fellow.handle.first);
    printf("%s%s.\n", msgs[0], fellow.handle.first);
    printf("%s%s\n", msgs[1], fellow.job);
    printf("%s\n", msgs[2]);
    printf("%s%s%s", msgs[3], fellow.favfood, msgs[4]);
    if (fellow.income > 150000.0)
        puts("!!!");
    else if (fellow.income > 75000.0)
        puts("!!");
    else
        puts("!!!");
}
```

```

    puts(".");
    printf("\n%40s%s\n", " ", "See you soon,");
    printf("%40s%s\n", " ", "Shalala");

    return 0;
}

```

下面是该程序的输出：

Dear Ewen,

Thank you for the wonderful evening, Ewen.  
 You certainly prove that a personality coach  
 is a special kind of guy. We must get together  
 over a delicious grilled salmon and have a few laughs.

See you soon,  
 Shalala

首先，注意如何在结构声明中创建嵌套结构。和声明 int 类型变量一样，进行简单的声明：

struct names handle;

该声明表明 handle 是一个 struct name 类型的变量。当然，文件中也应包含结构 names 的声明。

其次，注意如何访问嵌套结构的成员，这需要使用两次点运算符：

printf("Hello, %s!\n", fellow.handle.first);

从左往右解释 fellow.handle.first:

(fellow.handle).first

也就是说，找到 fellow，然后找到 fellow 的 handle 的成员，再找到 handle 的 first 成员。

## 14.6 指向结构的指针

喜欢使用指针的人一定很高兴能使用指向结构的指针。至少有 4 个理由可以解释为何要使用指向结构的指针。第一，就像指向数组的指针比数组本身更容易操控（如，排序问题）一样，指向结构的指针通常比结构本身更容易操控。第二，在一些早期的 C 实现中，结构不能作为参数传递给函数，但是可以传递指向结构的指针。第三，即使能传递一个结构，传递指针通常更有效率。第四，一些用于表示数据的结构中包含指向其他结构的指针。

下面的程序（程序清单 14.4）演示了如何定义指向结构的指针和如何用这样的指针访问结构的成员。

**程序清单 14.4 friends.c 程序**

```

/* friends.c -- 使用指向结构的指针 */
#include <stdio.h>
#define LEN 20

struct names {
    char first[LEN];
    char last[LEN];
};

struct guy {
    struct names handle;
    char favfood[LEN];
    char job[LEN];
}

```

```

    float income;
};

int main(void)
{
    struct guy fellow[2] = {
        { { "Ewen", "Villard" },
          "grilled salmon",
          "personality coach",
          68112.00
        },
        { { "Rodney", "Swillbelly" },
          "tripe",
          "tabloid editor",
          432400.00
        }
    };
    struct guy * him; /* 这是一个指向结构的指针 */

    printf("address #1: %p #2: %p\n", &fellow[0], &fellow[1]);
    him = &fellow[0]; /* 告诉编译器该指针指向何处 */
    printf("pointer #1: %p #2: %p\n", him, him + 1);
    printf("him->income is $%.2f: (*him).income is $%.2f\n",
           him->income, (*him).income);
    him++; /* 指向下一个结构 */
    printf("him->favfood is %s: him->handle.last is %s\n",
           him->favfood, him->handle.last);

    return 0;
}

```

该程序的输出如下：

```

address #1: 0x7fff5fbff820 #2: 0x7fff5fbff874
pointer #1: 0x7fff5fbff820 #2: 0x7fff5fbff874
him->income is $68112.00: (*him).income is $68112.00
him->favfood is tripe: him->handle.last is Swillbelly

```

我们先来看如何创建指向 guy 类型结构的指针，然后再分析如何通过该指针指定结构的成员。

## 14.6.1 声明和初始化结构指针

声明结构指针很简单：

```
struct guy * him;
```

首先是关键字 struct，其次是结构标记 guy，然后是一个星号 (\*)，其后跟着指针名。这个语法和其他指针声明一样。

该声明并未创建一个新的结构，但是指针 him 现在可以指向任意现有的 guy 类型的结构。例如，如果 barney 是一个 guy 类型的结构，可以这样写：

```
him = &barney;
```

和数组不同的是，结构名并不是结构的地址，因此要在结构名前面加上&运算符。

在本例中，fellow 是一个结构数组，这意味着 fellow[0] 是一个结构。所以，要让 him 指向 fellow[0]，可以这样写：

```
him = &fellow[0];
```

输出的前两行说明赋值成功。比较这两行发现，him 指向 fellow[0]，him + 1 指向 fellow[1]。注意，him 加 1 相当于 him 指向的地址加 84。在十六进制中， $874 - 820 = 54$ （十六进制）= 84（十进制），因为每个 guy 结构都占用 84 字节的内存：names.first 占用 20 字节，names.last 占用 20 字节，favfood 占用 20 字节，job 占用 20 字节，income 占用 4 字节（假设系统中 float 占用 4 字节）。顺带一提，在有些系统中，一个结构的大小可能大于它各成员大小之和。这是因为系统对数据进行校准的过程中产生了一些“缝隙”。例如，有些系统必须把每个成员都放在偶数地址上，或 4 的倍数的地址上。在这种系统中，结构的内部就存在未使用的“缝隙”。

## 14.6.2 用指针访问成员

指针 him 指向结构变量 fellow[0]，如何通过 him 获得 fellow[0] 的成员的值？程序清单 14.4 中的第 3 行输出演示了两种方法。

第 1 种方法也是最常用的方法：使用-> 运算符。该运算符由一个连接号（-）后跟一个大于号（>）组成。我们有下面的关系：

如果 him == &barney，那么 him->income 即是 barney.income

如果 him == &fellow[0]，那么 him->income 即是 fellow[0].income

换句话说，-> 运算符后面的结构指针和 . 运算符后面的结构名工作方式相同（不能写成 him.income，因为 him 不是结构名）。

这里要着重理解 him 是一个指针，但是 him->income 是该指针所指向结构的一个成员。所以在该例中，him->income 是一个 float 类型的变量。

第 2 种方法是，以这样的顺序指定结构成员的值：如果 him == &fellow[0]，那么 \*him == fellow[0]，因为 & 和 \* 是一对互逆运算符。因此，可以做以下替代：

```
fellow[0].income == (*him).income
```

必须要使用圆括号，因为 . 运算符比 \* 运算符的优先级高。

总之，如果 him 是指向 guy 类型结构 barney 的指针，下面的关系恒成立：

```
barney.income == (*him).income == him->income // 假设 him == &barney
```

接下来，我们来学习结构和函数的交互。

## 14.7 向函数传递结构的信息

函数的参数把值传递给函数。每个值都是一个数字——可能是 int 类型、float 类型，可能是 ASCII 字符码，或者是一个地址。然而，一个结构比一个单独的值复杂，所以难怪以前的 C 实现不允许把结构作为参数传递给函数。当前的实现已经移除了这个限制，ANSI C 允许把结构作为参数使用。所以程序员可以选择传递结构本身，还是传递指向结构的指针。如果你只关心结构中的某一部分，也可以把结构的成员作为参数。我们接下来将分析这 3 种传递方式，首先介绍以结构成员作为参数的情况。

### 14.7.1 传递结构成员

只要结构成员是一个具有单个值的数据类型（即，int 及其相关类型、char、float、double 或指针），便可把它作为参数传递给接受该特定类型的函数。程序清单 14.5 中的财务分析程序（初级版本）演示了这一点，该程序把客户的银行账户添加到他/她的储蓄和贷款账户中。

## 程序清单 14.5 funds1.c 程序

```
/* funds1.c -- 把结构成员作为参数传递 */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char      bank[FUNDLEN];
    double    bankfund;
    char      save[FUNDLEN];
    double    savefund;
};

double sum(double, double);

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n",
           sum(stan.bankfund, stan.savefund));
    return 0;
}

/* 两个 double 类型的数相加 */
double sum(double x, double y)
{
    return(x + y);
}
```

运行该程序后输出如下：

```
Stan has a total of $12576.21.
```

看来，这样传递参数没问题。注意，`sum()` 函数既不知道也不关心实际的参数是否是结构的成员，它只要求传入的数据是 `double` 类型。

当然，如果需要在被调函数中修改主调函数中成员的值，就要传递成员的地址：

```
modify(&stan.bankfund);
```

这是一个更改银行账户的函数。

把结构的信息告诉函数的第 2 种方法是，让被调函数知道自己正在处理一个结构。

## 14.7.2 传递结构的地址

我们继续解决前面的问题，但是这次把结构的地址作为参数。由于函数要处理 `funds` 结构，所以必须声明 `funds` 结构。如程序清单 14.6 所示。

## 程序清单 14.6 funds2.c 程序

```
/* funds2.c -- 传递指向结构的指针 */
```

```

#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char      bank[FUNDLEN];
    double    bankfund;
    char      save[FUNDLEN];
    double    savefund;
};

double sum(const struct funds *); /* 参数是一个指针 */

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n", sum(&stan));

    return 0;
}

double sum(const struct funds * money)
{
    return(money->bankfund + money->savefund);
}

```

运行该程序后输出如下：

```
Stan has a total of $12576.21.
```

`sum()` 函数使用指向 `funds` 结构的指针 (`money`) 作为它的参数。把地址 `&stan` 传递给该函数，使得指针 `money` 指向结构 `stan`。然后通过`->` 运算符获取 `stan.bankfund` 和 `stan.savefund` 的值。由于该函数不能改变指针所指向值的内容，所以把 `money` 声明为一个指向 `const` 的指针。

虽然该函数并未使用其他成员，但是也可以访问它们。注意，必须使用`&` 运算符来获取结构的地址。和数组名不同，结构名只是其地址的别名。

### 14.7.3 传递结构

对于允许把结构作为参数的编译器，可以把程序清单 14.6 重写为程序清单 14.7。

程序清单 14.7 `funds3.c` 程序

---

```

/* funds3.c -- 传递一个结构 */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char      bank[FUNDLEN];
    double    bankfund;
    char      save[FUNDLEN];
    double    savefund;
};

```

```

    double savefund;
};

double sum(struct funds moolah); /* 参数是一个结构 */

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n", sum(stan));

    return 0;
}

double sum(struct funds moolah)
{
    return(moolah.bankfund + moolah.savefund);
}

```

下面是运行该程序后的输出：

```
Stan has a total of $12576.21.
```

该程序把程序清单 14.6 中指向 `struct funds` 类型的结构指针 `money` 替换成 `struct funds` 类型的结构变量 `moolah`。调用 `sum()` 时，编译器根据 `funds` 模板创建了一个名为 `moolah` 的自动结构变量。然后，该结构的各成员被初始化为 `stan` 结构变量相应成员的值的副本。因此，程序使用原来结构的副本进行计算，然而，传递指针的程序清单 14.6 使用的是原始的结构进行计算。由于 `moolah` 是一个结构，所以该程序使用 `moolah.bankfund`，而不是 `moolah->bankfund`。另一方面，由于 `money` 是指针，不是结构，所以程序清单 14.6 使用的是 `monet->bankfund`。

#### 14.7.4 其他结构特性

现在的 C 允许把一个结构赋值给另一个结构，但是数组不能这样做。也就是说，如果 `n_data` 和 `o_data` 都是相同类型的结构，可以这样做：

```
o_data = n_data; // 把一个结构赋值给另一个结构
```

这条语句把 `n_data` 的每个成员的值都赋给 `o_data` 的相应成员。即使成员是数组，也能完成赋值。另外，还可以把一个结构初始化为相同类型的另一个结构：

```

struct names right_field = {"Ruthie", "George"};
struct names captain = right_field; // 把一个结构初始化为另一个结构

```

现在的 C（包括 ANSI C），函数不仅能把结构本身作为参数传递，还能把结构作为返回值返回。把结构作为函数参数可以把结构的信息传送给函数；把结构作为返回值的函数能把结构的信息从被调函数传回主调函数。结构指针也允许这种双向通信，因此可以选择任一种方法来解决编程问题。我们通过另一组程序示例来演示这两种方法。

为了对比这两种方法，我们先编写一个程序以传递指针的方式处理结构，然后以传递结构和返回结构的方式重写该程序。

## 程序清单 14.8 names1.c 程序

```

/* names1.c -- 使用指向结构的指针 */
#include <stdio.h>
#include <string.h>

#define NLEN 30

struct namect {
    char fname[NLEN];
    char lname[NLEN];
    int letters;
};

void getinfo(struct namect * );
void makeinfo(struct namect * );
void showinfo(const struct namect * );
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
    return 0;
}

void getinfo(struct namect * pst)
{
    printf("Please enter your first name.\n");
    s_gets(pst->fname, NLEN);
    printf("Please enter your last name.\n");
    s_gets(pst->lname, NLEN);
}

void makeinfo(struct namect * pst)
{
    pst->letters = strlen(pst->fname) +strlen(pst->lname);
}

void showinfo(const struct namect * pst)
{
    printf("%s %s, your name contains %d letters.\n",
           pst->fname, pst->lname, pst->letters);
}

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;       // 处理输入行的剩余字符
    }
    return ret_val;
}

```

下面是编译并运行该程序后的一个输出示例：

```

Please enter your first name.
Viola
Please enter your last name.
Plunderfest
Viola Plunderfest, your name contains 16 letters.

```

该程序把任务分配给 3 个函数来完成，都在 main() 中调用。每调用一个函数就把 person 结构的地址传递给它。

getinfo() 函数把结构的信息从自身传递给 main()。该函数通过与用户交互获得姓名，并通过 pst 指针定位，将其放入 person 结构中。由于 pst->lname 意味着 pst 指向结构的 lname 成员，这使得 pst->lname 等价于 char 数组的名称，因此做 s\_gets() 的参数很合适。注意，虽然 getinfo() 给 main() 提供了信息，但是它并未使用返回机制，所以其返回类型是 void。

makeinfo() 函数使用双向传输方式传送信息。通过使用指向 person 的指针，该指针定位了储存在该结构中的名和姓。该函数使用 C 库函数 strlen() 分别计算名和姓中的字母总数，然后使用 person 的地址储存两数之和。同样，makeinfo() 函数的返回类型也是 void。

showinfo() 函数使用一个指针定位待打印的信息。因为该函数不改变数组的内容，所以将其声明为 const。

所有这些操作中，只有一个结构变量 person，每个函数都使用该结构变量的地址来访问它。一个函数把信息从自身传回主调函数，一个函数把信息从主调函数传给自身，一个函数通过双向传输来传递信息。

现在，我们来看如何使用结构参数和返回值来完成相同任务。第一，为了传递结构本身，函数的参数必须是 person，而不是&person。那么，相应的形式参数应声明为 struct namect，而不是指向该类型的指针。第二，可以通过返回一个结构，把结构的信息返回给 main()。程序清单 14.9 演示了不使用指针的版本。

#### 程序清单 14.9 names2.c 程序

```

/* names2.c -- 传递并返回结构 */
#include <stdio.h>
#include <string.h>

#define NLEN 30
struct namect {
    char fname[NLEN];
    char lname[NLEN];
    int letters;
};

struct namect getinfo(void);
struct namect makeinfo(struct namect);

```

```

void showinfo(struct namect);
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    person = getinfo();
    person = makeinfo(person);
    showinfo(person);

    return 0;
}

struct namect getinfo(void)
{
    struct namect temp;
    printf("Please enter your first name.\n");
    s_gets(temp.fname, NLEN);
    printf("Please enter your last name.\n");
    s_gets(temp.lname, NLEN);

    return temp;
}

struct namect makeinfo(struct namect info)
{
    info.letters = strlen(info.fname) + strlen(info.lname);

    return info;
}

void showinfo(struct namect info)
{
    printf("%s %s, your name contains %d letters.\n",
           info.fname, info.lname, info.letters);
}

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; // 处理输入行的剩余部分
    }
    return ret_val;
}

```

该版本最终的输出和前面版本相同，但是它使用了不同的方式。程序中的每个函数都创建了自己的 person 备份，所以该程序使用了 4 个不同的结构，不像前面的版本只使用一个结构。

例如，考虑 makeinfo() 函数。在第 1 个程序中，传递的是 person 的地址，该函数实际上处理的是 person 的值。在第 2 个版本的程序中，创建了一个新的结构 info。储存在 person 中的值被拷贝到 info 中，函数处理的是这个副本。因此，统计完字母个数后，计算结果储存在 info 中，而不是 person 中。然而，返回机制弥补了这一点。makeinfo() 中的这行代码：

```
return info;
```

与 main() 中的这行结合：

```
person = makeinfo(person);
```

把储存在 info 中的值拷贝到 person 中。注意，必须把 makeinfo() 函数声明为 struct namect 类型，所以该函数要返回一个结构。

## 14.7.5 结构和结构指针的选择

假设要编写一个与结构相关的函数，是用结构指针作为参数，还是用结构作为参数和返回值？两者各有优缺点。

把指针作为参数有两个优点：无论是以前还是现在的 C 实现都能使用这种方法，而且执行起来很快，只需要传递一个地址。缺点是无法保护数据。被调函数中的某些操作可能会意外影响原来结构中的数据。不过，ANSI C 新增的 const 限定符解决了这个问题。例如，如果在程序清单 14.8 中，showinfo() 函数中的代码改变了结构的任意成员，编译器会捕获这个错误。

把结构作为参数传递的优点是，函数处理的是原始数据的副本，这保护了原始数据。另外，代码风格也更清楚。假设定义了下面的结构类型：

```
struct vector {double x; double y;};
```

如果用 vector 类型的结构 ans 储存相同类型结构 a 和 b 的和，就要把结构作为参数和返回值：

```
struct vector ans, a, b;
struct vector sum_vect(struct vector, struct vector);
...
ans = sum_vect(a,b);
```

对程序员而言，上面的版本比用指针传递的版本更自然。指针版本如下：

```
struct vector ans, a, b;
void sum_vect(const struct vector *, const struct vector *, struct vector *);
...
sum_vect(&a, &b, &ans);
```

另外，如果使用指针版本，程序员必须记住总和的地址应该是第 1 个参数还是第 2 个参数的地址。

传递结构的两个缺点是：较老版本的实现可能无法处理这样的代码，而且传递结构浪费时间和存储空间。尤其是把大型结构传递给函数，而它只使用结构中的一两个成员时特别浪费。这种情况下传递指针或只传递函数所需的成员更合理。

通常，程序员为了追求效率会使用结构指针作为函数参数，如需防止原始数据被意外修改，使用 const 限定符。按值传递结构是处理小型结构最常用的方法。

## 14.7.6 结构中的字符数组和字符指针

到目前为止，我们在结构中都使用字符数组来储存字符串。是否可以使用指向 char 的指针来代替字符数组？例如，程序清单 14.3 中有如下声明：

```
#define LEN 20
struct names {
    char first[LEN];
    char last[LEN];
};
```

其中的结构声明是否可以这样写：

```
struct pnames {
    char * first;
    char * last;
};
```

当然可以，但是如果理解这样做的含义，可能会有麻烦。考虑下面的代码：

```
struct names veep = {"Talia", "Summers"};
struct pnames treas = {"Brad", "Fallingjaw"};
printf("%s and %s\n", veep.first, treas.first);
```

以上代码都没问题，也能正常运行，但是思考一下字符串被储存在何处。对于 struct names 类型的结构变量 veep，以上字符串都储存在结构内部，结构总共要分配 40 字节储存姓名。然而，对于 struct pnames 类型的结构变量 treas，以上字符串储存在编译器储存常量的地方。结构本身只储存了两个地址，在我们的系统中共占 16 字节。尤其是，struct pnames 结构不用为字符串分配任何存储空间。它使用的储存在别处的字符串（如，字符串常量或数组中的字符串）。简而言之，在 pnames 结构变量中的指针应该只用来在程序中管理那些已分配和在别处分配的字符串。

我们看看这种限制在什么情况下出问题。考虑下面的代码：

```
struct names accountant;
struct pnames attorney;
puts("Enter the last name of your accountant:");
scanf("%s", accountant.last);
puts("Enter the last name of your attorney:");
scanf("%s", attorney.last); /* 这里有一个潜在的危险 */
```

就语法而言，这段代码没问题。但是，用户的输入储存到哪里去了？对于会计师 (accountant)，他的名储存在 accountant 结构变量的 last 成员中，该结构中有一个储存字符串的数组。对于律师 (attorney)，scanf() 把字符串放到 attorney.last 表示的地址上。由于这是未经初始化的变量，地址可以是任何值，因此程序可以把名放在任何地方。如果走运的话，程序不会出问题，至少暂时不会出问题，否则这一操作会导致程序崩溃。实际上，如果程序能正常运行并不是好事，因为这意味着一个未被觉察的危险潜伏在程序中。

因此，如果要用结构储存字符串，用字符数组作为成员比较简单。用指向 char 的指针也行，但是误用会导致严重的问题。

### 14.7.7 结构、指针和 malloc()

如果使用 malloc() 分配内存并使用指针储存该地址，那么在结构中使用指针处理字符串就比较合理。这种方法的优点是，可以请求 malloc() 为字符串分配合适的存储空间。可以要求用 4 字节储存 "Joe" 和用 18 字节储存 "Rasolofomasoandro"。用这种方法改写程序清单 14.9 并不费劲。主要是更改结构声明（用指针代替数组）和提供一个新版本的 getinfo() 函数。新的结构声明如下：

```
struct namect {
    char * fname; // 用指针代替数组
    char * lname;
    int letters;
};
```

新版本的 getinfo() 把用户的输入读入临时数组中，调用 malloc() 函数分配存储空间，并把字符串拷贝到新分配的存储空间中。对名和姓都要这样做：

```
void getinfo (struct namect * pst)
{
    char temp[SLEN];
    printf("Please enter your first name.\n");
    s_gets(temp, SLEN);
    // 分配内存储存名
    pst->fname = (char *) malloc(strlen(temp) + 1);
    // 把名拷贝到已分配的内存
    strcpy(pst->fname, temp);
    printf("Please enter your last name.\n");
    s_gets(temp, SLEN);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}
```

要理解这两个字符串都未储存在结构中，它们储存在 malloc() 分配的内存块中。然而，结构中储存着这两个字符串的地址，处理字符串的函数通常都要使用字符串的地址。因此，不用修改程序中的其他函数。

第 12 章建议，应该成对使用 malloc() 和 free()。因此，还要在程序中添加一个新的函数 cleanup()，用于释放程序动态分配的内存。如程序清单 14.10 所示。

程序清单 14.10 names3.c 程序

```
// names3.c -- 使用指针和 malloc()
#include <stdio.h>
#include <string.h> // 提供 strcpy()、strlen() 的原型
#include <stdlib.h> // 提供 malloc()、free() 的原型
#define SLEN 81
struct namect {
    char * fname; // 使用指针
    char * lname;
    int letters;
};

void getinfo(struct namect *); // 分配内存
void makeinfo(struct namect *);
void showinfo(const struct namect *);
void cleanup(struct namect *); // 调用该函数时释放内存
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
    cleanup(&person);

    return 0;
}
```

```

void getinfo(struct namect * pst)
{
    char temp[SLEN];
    printf("Please enter your first name.\n");
    s_gets(temp, SLEN);
    // 分配内存以储存名
    pst->fname = (char *) malloc(strlen(temp) + 1);
    // 把名拷贝到动态分配的内存中
    strcpy(pst->fname, temp);
    printf("Please enter your last name.\n");
    s_gets(temp, SLEN);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}

void makeinfo(struct namect * pst)
{
    pst->letters = strlen(pst->fname) +
        strlen(pst->lname);
}

void showinfo(const struct namect * pst)
{
    printf("%s %s, your name contains %d letters.\n",
           pst->fname, pst->lname, pst->letters);
}

void cleanup(struct namect * pst)
{
    free(pst->fname);
    free(pst->lname);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;
    if (fgets(st, n, stdin))
    {
        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; // 处理输入行的剩余部分
    }
    return ret_val;
}

```

下面是该程序的输出：

```
Please enter your first name.  
Floresiensis  
Please enter your last name.  
Mann  
Floresiensis Mann, your name contains 16 letters.
```

### 14.7.8 复合字面量和结构 (C99)

C99 的复合字面量特性可用于结构和数组。如果只需要一个临时结构值，复合字面量很好用。例如，可以使用复合字面量创建一个数组作为函数的参数或赋给另一个结构。语法是把类型名放在圆括号中，后面紧跟一个用花括号括起来的初始化列表。例如，下面是 struct book 类型的复合字面量：

```
(struct book) {"The Idiot", "Fyodor Dostoyevsky", 6.99}
```

程序清单 14.11 中的程序示例，使用复合字面量为一个结构变量提供两个可替换的值（在撰写本书时，并不是所有的编译器都支持这个特性，不过这是时间的问题）。

程序清单 14.11 complit.c 程序

---

```
/* complit.c -- 复合字面量 */  
#include <stdio.h>  
#define MAXTITL 41  
#define MAXAUTL 31  
  
struct book {           // 结构模版：标记是 book  
    char title[MAXTITL];  
    char author[MAXAUTL];  
    float value;  
};  
  
int main(void)  
{  
    struct book readfirst;  
    int score;  
  
    printf("Enter test score: ");  
    scanf("%d", &score);  
  
    if (score >= 84)  
        readfirst = (struct book) {"Crime and Punishment",  
                                  "Fyodor Dostoyevsky",  
                                  11.25};  
    else  
        readfirst = (struct book) {"Mr. Bouncy's Nice Hat",  
                                  "Fred Winsome",  
                                  5.99};  
  
    printf("Your assigned reading:\n");  
    printf("%s by %s: $%.2f\n", readfirst.title,  
          readfirst.author, readfirst.value);  
  
    return 0;  
}
```

还可以把复合字面量作为函数的参数。如果函数接受一个结构，可以把复合字面量作为实际参数传递：

```
struct rect {double x; double y;};
double rect_area(struct rect r){return r.x * r.y;}
...
double area;
area = rect_area( (struct rect) {10.5, 20.0});
```

值 210 被赋给 area。

如果函数接受一个地址，可以传递复合字面量的地址：

```
struct rect {double x; double y;};
double rect_areap(struct rect * rp){return rp->x * rp->y;}
...
double area;
area = rect_areap( &(struct rect) {10.5, 20.0});
```

值 210 被赋给 area。

复合字面量在所有函数的外部，具有静态存储期；如果复合字面量在块中，则具有自动存储期。复合字面量和普通初始化列表的语法规则相同。这意味着，可以在复合字面量中使用指定初始化器。

### 14.7.9 伸缩型数组成员 (C99)

C99 新增了一个特性：伸缩型数组成员 (*flexible array member*)，利用这项特性声明的结构，其最后一个数组成员具有一些特性。第 1 个特性是，该数组不会立即存在。第 2 个特性是，使用这个伸缩型数组成员可以编写合适的代码，就好像它确实存在并具有所需数目的元素一样。这可能听起来很奇怪，所以我们来一步步地创建和使用一个带伸缩型数组成员的结构。

首先，声明一个伸缩型数组成员有如下规则：

- 伸缩型数组成员必须是结构的最后一个成员；
- 结构中必须至少有一个成员；
- 伸缩数组的声明类似于普通数组，只是它的方括号中是空的。

下面用一个示例来解释以上几点：

```
struct flex
{
    int count;
    double average;
    double scores[]; // 伸缩型数组成员
};
```

声明一个 struct flex 类型的结构变量时，不能用 scores 做任何事，因为没有给这个数组预留存储空间。实际上，C99 的意图并不是让你声明 struct flex 类型的变量，而是希望你声明一个指向 struct flex 类型的指针，然后用 malloc() 来分配足够的空间，以储存 struct flex 类型结构的常规内容和伸缩型数组成员所需的额外空间。例如，假设用 scores 表示一个内含 5 个 double 类型值的数组，可以这样做：

```
struct flex * pf; // 声明一个指针
// 请求为一个结构和一个数组分配存储空间
pf = malloc(sizeof(struct flex) + 5 * sizeof(double));
```

现在有足够的存储空间储存 count、average 和一个内含 5 个 double 类型值的数组。可以用指针 pf 访问这些成员：

```
pf->count = 5;           // 设置 count 成员
pf->scores[2] = 18.5;    // 访问数组成员的一个元素
```

程序清单 14.13 进一步扩展了这个例子，让伸缩型数组成员在第 1 种情况下表示 5 个值，在第 2 种情况下代表 9 个值。该程序也演示了如何编写一个函数处理带伸缩型数组元素的结构。

## 程序清单 14.12 flexmemb.c 程序

```
// flexmemb.c -- 伸缩型数组成员 (C99 新增特性)
#include <stdio.h>
#include <stdlib.h>

struct flex
{
    size_t count;
    double average;
    double scores [];
    // 伸缩型数组成员
};

void showFlex(const struct flex * p);

int main(void)
{
    struct flex * pf1, *pf2;
    int n = 5;
    int i;
    int tot = 0;

    // 为结构和数组分配存储空间
    pf1 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf1->count = n;
    for (i = 0; i < n; i++)
    {
        pf1->scores[i] = 20.0 - i;
        tot += pf1->scores[i];
    }
    pf1->average = tot / n;
    showFlex(pf1);

    n = 9;
    tot = 0;
    pf2 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf2->count = n;
    for (i = 0; i < n; i++)
    {
        pf2->scores[i] = 20.0 - i / 2.0;
        tot += pf2->scores[i];
    }
    pf2->average = tot / n;
    showFlex(pf2);
    free(pf1);
    free(pf2);

    return 0;
}

void showFlex(const struct flex * p)
{
```

```

int i;
printf("Scores : ");
for (i = 0; i < p->count; i++)
    printf("%g ", p->scores[i]);
printf("\nAverage: %g\n", p->average);
}

```

下面是该程序的输出：

```

Scores : 20 19 18 17 16
Average: 18
Scores : 20 19.5 19 18.5 18 17.5 17 16.5 16
Average: 17

```

带伸缩型数组成员的结构确实有一些特殊的处理要求。第一，不能用结构进行赋值或拷贝：

```

struct flex * pf1, *pf2; // *pf1 和*pf2 都是结构
...
*pf2 = *pf1;           // 不要这样做

```

这样做只能拷贝除伸缩型数组成员以外的其他成员。确实要进行拷贝，应使用 `memcpy()` 函数（第 16 章中介绍）。

第二，不要以按值方式把这种结构传递给结构。原因相同，按值传递一个参数与赋值类似。要把结构的地址传递给函数。

第三，不要使用带伸缩型数组成员的结构作为数组成员或另一个结构的成员。

这种类似于在结构中最后一个成员是伸缩型数组的情况，称为 `struct hack`。除了伸缩型数组成员在声明时用空的方括号外，`struct hack` 特指大小为 0 的数组。然而，`struct hack` 是针对特殊编译器（GCC）的，不属于 C 标准。这种伸缩型数组成员方法是标准认可的编程技巧。

## 14.7.10 匿名结构（C11）

匿名结构是一个没有名称的结构成员。为了理解它的工作原理，我们先考虑如何创建嵌套结构：

```

struct names
{
    char first[20];
    char last[20];
};

struct person
{
    int id;
    struct names name; // 嵌套结构成员
};

struct person ted = {8483, {"Ted", "Grass"}};

```

这里，`name` 成员是一个嵌套结构，可以通过类似 `ted.name.first` 的表达式访问“ted”：

```
puts(ted.name.first);
```

在 C11 中，可以用嵌套的匿名成员结构定义 `person`：

```

struct person
{
    int id;
    struct {char first[20]; char last[20];}; // 匿名结构
};

```

初始化 `ted` 的方式相同：

```
struct person ted = {8483, {"Ted", "Grass"}};
```

但是，在访问 `ted` 时简化了步骤，只需把 `first` 看作是 `person` 的成员那样使用它：

```
puts(ted.first);
```

当然，也可以把 `first` 和 `last` 直接作为 `person` 的成员，删除嵌套循环。匿名特性在嵌套联合中更加有用，我们在本章后面介绍。

### 14.7.11 使用结构数组的函数

假设一个函数要处理一个结构数组。由于数组名就是该数组的地址，所以可以把它传递给函数。另外，该函数还需访问结构模板。为了理解该函数的工作原理，程序清单 14.13 把前面的金融程序扩展为两人，所以需要一个内含两个 `funds` 结构的数组。

程序清单 14.13 funds4.c 程序

```
/* funds4.c -- 把结构数组传递给函数 */
#include <stdio.h>
#define FUNDLEN 50
#define N 2

struct funds {
    char     bank[FUNDLEN];
    double   bankfund;
    char     save[FUNDLEN];
    double   savefund;
};

double sum(const struct funds money [], int n);

int main(void)
{
    struct funds jones[N] = {
        {
            "Garlic-Melon Bank",
            4032.27,
            "Lucky's Savings and Loan",
            8543.94
        },
        {
            "Honest Jack's Bank",
            3620.88,
            "Party Time Savings",
            3802.91
        }
    };

    printf("The Joneses have a total of $%.2f.\n", sum(jones, N));

    return 0;
}

double sum(const struct funds money [], int n)
{
    double total;
```

```

int i;

for (i = 0, total = 0; i < n; i++)
    total += money[i].bankfund + money[i].savefund;

return(total);
}

```

该程序的输出如下：

The Joneses have a total of \$20000.00.

(读者也许认为这个总和有些巧合!)

数组名 `jones` 是该数组的地址，即该数组首元素 (`jones[0]`) 的地址。因此，指针 `money` 的初始值相当于通过下面的表达式获得：

```
money = &jones[0];
```

因为 `money` 指向 `jones` 数组的首元素，所以 `money[0]` 是该数组的另一个名称。与此类似，`money[1]` 是第 2 个元素。每个元素都是一个 `funds` 类型的结构，所以都可以使用点运算符 (.) 来访问 `funds` 类型结构的成员。

下面是几个要点。

- 可以把数组名作为数组中第 1 个结构的地址传递给函数。
- 然后可以用数组表示法访问数组中的其他结构。注意下面的函数调用与使用数组名效果相同：
 

```
sum(&jones[0], N)
```

 因为 `jones` 和 `&jones[0]` 的地址相同，使用数组名是传递结构地址的一种间接的方法。
- 由于 `sum()` 函数不能改变原始数据，所以该函数使用了 ANSI C 的限定符 `const`。

## 14.8 把结构内容保存到文件中

由于结构可以储存不同类型的信息，所以它是构建数据库的重要工具。例如，可以用一个结构储存雇员或汽车零件的相关信息。最终，我们要把这些信息储存在文件中，并且能再次检索。数据库文件可以包含任意数量的此类数据对象。储存在一个结构中的整套信息被称为记录 (*record*)，单独的项被称为字段 (*field*)。本节我们来探讨这个主题。

或许储存记录最没效率的方法是用 `fprintf()`。例如，回忆程序清单 14.1 中的 `book` 结构：

```

#define MAXTITL 40
#define MAXAUTL 40
struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

```

如果 `pbook` 标识一个文件流，那么通过下面这条语句可以把信息储存在 `struct book` 类型的结构变量 `primer` 中：

```
fprintf(pbooks, "%s %s %.2f\n", primer.title, primer.author, primer.value);
```

对于一些结构（如，有 30 个成员的结构），这个方法用起来很不方便。另外，在检索时还存在问题，因为程序要知道一个字段结束和另一个字段开始的位置。虽然用固定字段宽度的格式可以解决这个问题（例如，`"%39s%39s%8.2f"`），但是这个方法仍然很笨拙。

更好的方案是使用 `fread()` 和 `fwrite()` 函数读写结构大小的单元。回忆一下，这两个函数使用与程序相同的二进制表示法。例如：

```
fwrite(&primer, sizeof(struct book), 1, pbooks);
```

定位到 `primer` 结构变量开始的位置，并把结构中所有的字节都拷贝到与 `pbooks` 相关的文件中。`sizeof(struct book)` 告诉函数待拷贝的一块数据的大小，1 表明一次拷贝一块数据。带相同参数的 `fread()` 函数从文件中拷贝一块结构大小的数据到 `&primer` 指向的位置。简而言之，这两个函数一次读写整个记录，而不是一个字段。

以二进制表示法储存数据的缺点是，不同的系统可能使用不同的二进制表示法，所以数据文件可能不具可移植性。甚至同一个系统，不同编译器设置也可能导致不同的二进制布局。

### 14.8.1 保存结构的程序示例

为了演示如何在程序中使用这些函数，我们把程序清单 14.2 修改为一个新的版本（即程序清单 14.14），把书名保存在 `book.dat` 文件中。如果该文件已存在，程序将显示它当前的内容，然后允许在文件中添加内容（如果你使用的是早期的 Borland 编译器，请参阅程序清单 14.2 后面的“Borland C 和浮点数”）。

程序清单 14.14 `booksave.c` 程序

```
/* booksave.c -- 在文件中保存结构中的内容 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXTITL 40
#define MAXAUTL 40
#define MAXBKS 10           /* 最大书籍数量 */
char * s_gets(char * st, int n);
struct book {             /* 建立 book 模板 */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book library[MAXBKS]; /* 结构数组 */
    int count = 0;
    int index, filecount;
    FILE * pbooks;
    int size = sizeof(struct book);

    if ((pbooks = fopen("book.dat", "a+b")) == NULL)
    {
        fputs("Can't open book.dat file\n", stderr);
        exit(1);
    }

    rewind(pbooks);           /* 定位到文件开始 */
    while (count < MAXBKS && fread(&library[count], size,
        1, pbooks) == 1)
    {
        if (count == 0)
```

```

    puts("Current contents of book.dat:");
    printf("%s by %s: $%.2f\n", library[count].title,
           library[count].author, library[count].value);
    count++;
}
filecount = count;
if (count == MAXBKS)
{
    fputs("The book.dat file is full.", stderr);
    exit(2);
}

puts("Please add new book titles.");
puts("Press [enter] at the start of a line to stop.");
while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
    && library[count].title[0] != '\0')
{
    puts("Now enter the author.");
    s_gets(library[count].author, MAXAUTL);
    puts("Now enter the value.");
    scanf("%f", &library[count++].value);
    while (getchar() != '\n')
        continue; /* 清理输入行 */
    if (count < MAXBKS)
        puts("Enter the next title.");
}

if (count > 0)
{
    puts("Here is the list of your books:");
    for (index = 0; index < count; index++)
        printf("%s by %s: $%.2f\n", library[index].title,
               library[index].author, library[index].value);
    fwrite(&library[filecount], size, count - filecount,
          pbooks);
}
else
    puts("No books? Too bad.\n");

puts("Bye.\n");
fclose(pbooks);

return 0;
}

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;      // 清理输入行
    }
    return ret_val;
}

```

我们先看几个运行示例，然后再讨论程序中的要点。

```

$ booksave
Please add new book titles.
Press [enter] at the start of a line to stop.
Metric Merriment
Now enter the author.
Polly Poetica
Now enter the value.
18.99
Enter the next title.
Deadly Farce
Now enter the author.
Dudley Forse
Now enter the value.
15.99
Enter the next title.
[enter]
Here is the list of your books:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
Bye.
$ booksave
Current contents of book.dat:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
Please add new book titles.
The Third Jar
Now enter the author.
Nellie Nostrum
Now enter the value.
22.99
Enter the next title.
[enter]
Here is the list of your books:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
The Third Jar by Nellie Nostrum: $22.99
Bye.
$
```

再次运行 booksave.c 程序把这 3 本书作为当前的文件记录打印出来。

## 14.8.2 程序要点

首先，以 "a+b" 模式打开文件。a+部分允许程序读取整个文件并在文件的末尾添加内容。b 是 ANSI 的一种标识方法，表明程序将使用二进制文件格式。对于不接受 b 模式的 UNIX 系统，可以省略 b，因为

UNIX 只有一种文件形式。对于早期的 ANSI 实现，要找出和 `b` 等价的表示法。

我们选择二进制模式是因为 `fread()` 和 `fwrite()` 函数要使用二进制文件。虽然结构中有些内容是文本，但是 `value` 成员不是文本。如果使用文本编辑器查看 `book.dat`，该结构本文部分的内容显示正常，但是数值部分的内容不可读，甚至会导致文本编辑器出现乱码。

`rewrite()` 函数确保文件指针位于文件开始处，为读文件做好准备。

第 1 个 `while` 循环每次把一个结构读到结构数组中，当数组已满或读完文件时停止。变量 `filecount` 统计已读结构的数量。

第 2 个 `while` 按下循环提示用户进行输入，并接受用户的输入。和程序清单 14.2 一样，当数组已满或用户在一行的开始处按下 `Enter` 键时，循环结束。注意，该循环开始时 `count` 变量的值是第 1 个循环结束后的值。该循环把新输入项添加到数组的末尾。

然后 `for` 循环打印文件和用户输入的数据。因为该文件是以附加模式打开，所以新写入的内容添加到文件现有内容的末尾。

我们本可以用一个循环在文件末尾一次添加一个结构，但还是决定用 `fwrite()` 一次写入一块数据。对表达式 `count - filecount` 求值得新添加的书籍数量，然后调用 `fwrite()` 把结构大小的块写入文件。由于表达式 `&library[filecount]` 是数组中第 1 个新结构的地址，所以拷贝就从这里开始。

也许该例是把结构写入文件和检索它们的最简单的方法，但是这种方法浪费存储空间，因为这还保存了结构中未使用的部分。该结构的大小是  $2 \times 40 \times \text{sizeof}(\text{char}) + \text{sizeof}(\text{float})$ ，在我们的系统中共 84 字节。实际上不是每个输入项都需要这么多空间。但是，让每个输入块的大小相同在检索数据时很方便。

另一个方法是使用可变大小的记录。为了方便读取文件中的这种记录，每个记录以数值字段规定记录的大小。这比上一种方法复杂。通常，这种方法涉及接下来要介绍的“链式结构”和第 16 章的动态内存分配。

## 14.9 链式结构

在结束讨论结构之前，我们想简要介绍一下结构的多种用途之一：创建新的数据形式。计算机用户已经开发出的一些数据形式比我们提到过的数组和简单结构更有效地解决特定的问题。这些形式包括队列、二叉树、堆、哈希表和图表。许多这样的形式都由链式结构（*linked structure*）组成。通常，每个结构都包含一两个数据项和一两个指向其他同类型结构的指针。这些指针把一个结构和另一个结构链接起来，并提供一种路径能遍历整个彼此链接的结构。例如，图 14.3 演示了一个二叉树结构，每个单独的结构（或节点）都和它下面的两个结构（或节点）相连。

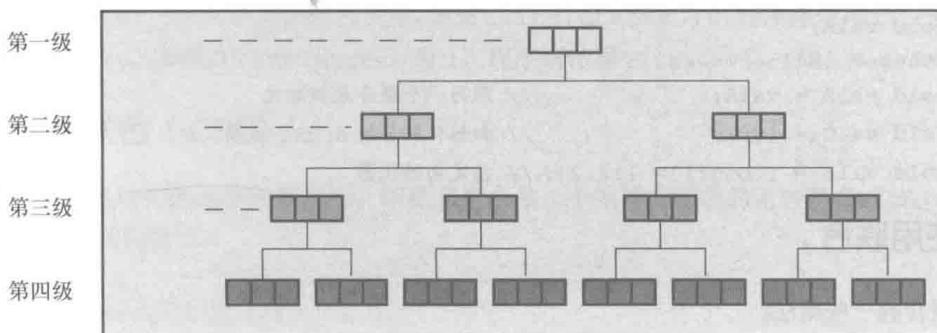


图 14.3 一个二叉树结构

图 14.3 中显示的分级或树状的结构是否比数组高效？考虑一个有 10 级节点的树的情况。它有  $2^{10}-1$ （或 1023）个节点，可以储存 1023 个单词。如果这些单词以某种规则排列，那么可以从最顶层开始，逐级向下

移动查找单词，最多只需移动 9 次便可找到任意单词。如果把这些单词都放在一个数组中，最多要查找 1023 个元素才能找出所需的单词。

如果你对这些高级概念感兴趣，可以阅读一些关于数据结构的书籍。使用 C 结构，可以创建和使用那些书中介绍的各种数据形式。另外，第 17 章中也介绍了一些高级数据形式。

本章对结构的概念介绍至此为止，第 17 章中会给出链式结构的例子。下面，我们介绍 C 语言中的联合、枚举和 `typedef`。

## 14.10 联合简介

联合（union）是一种数据类型，它能在同一个内存空间中储存不同的数据类型（不是同时储存）。其典型的用法是，设计一种表以储存既无规律、事先也不知道顺序的混合类型。使用联合类型的数组，其中的联合都大小相等，每个联合可以储存各种数据类型。

创建联合和创建结构的方式相同，需要一个联合模板和联合变量。可以用一个步骤定义联合，也可以用联合标记分两步定义。下面是一个带标记的联合模板：

```
union hold {
    int digit;
    double bigfl;
    char letter;
};
```

根据以上形式声明的结构可以储存一个 `int` 类型、一个 `double` 类型和 `char` 类型的值。然而，声明的联合只能储存一个 `int` 类型的值或一个 `double` 类型的值或 `char` 类型的值。

下面定义了 3 个与 `hold` 类型相关的变量：

```
union hold fit;           // hold 类型的联合变量
union hold save[10];      // 内含 10 个联合变量的数组
union hold * pu;          // 指向 hold 类型联合变量的指针
```

第 1 个声明创建了一个单独的联合变量 `fit`。编译器分配足够的空间以便它能储存联合声明中占用最大字节的类型。在本例中，占用空间最大的是 `double` 类型的数据。在我们的系统中，`double` 类型占 64 位，即 8 字节。第 2 个声明创建了一个数组 `save`，内含 10 个元素，每个元素都是 8 字节。第 3 个声明创建了一个指针，该指针变量储存 `hold` 类型联合变量的地址。

可以初始化联合。需要注意的是，联合只能储存一个值，这与结构不同。有 3 种初始化的方法：把一个联合初始化为另一个同类型的联合；初始化联合的第一个元素；或者根据 C99 标准，使用指定初始化器：

```
union hold valA;
valA.letter = 'R';
union hold valB = valA;           // 用另一个联合来初始化
union hold valC = {88};          // 初始化联合的 digit 成员
union hold valD = {.bigfl = 118.2}; // 指定初始化器
```

### 14.10.1 使用联合

下面是联合的一些用法：

```
fit.digit = 23;   // 把 23 储存在 fit，占 2 字节
fit.bigfl = 2.0;  // 清除 23，储存 2.0，占 8 字节
fit.letter = 'h'; // 清除 2.0，储存 h，占 1 字节
```

点运算符表示正在使用哪种数据类型。在联合中，一次只储存一个值。即使有足够的空间，也不能同时储存一个 char 类型值和一个 int 类型值。编写代码时要注意当前储存在联合中的数据类型。

和用指针访问结构使用->运算符一样，用指针访问联合时也要使用->运算符：

```
pu = &fit;
x = pu->digit; // 相当于 x = fit.digit
```

不要像下面的语句序列这样：

```
fit.letter = 'A';
fignum = 3.02*fit.bigfl; // 错误
```

以上语句序列是错误的，因为储存在 fit 中的是 char 类型，但是下一行却假定 fit 中的内容是 double 类型。

不过，用一个成员把值储存在一个联合中，然后用另一个成员查看内容，这种做法有时很有用。下一章的程序清单 15.4 就给出了一个这样的例子。

联合的另一种用法是，在结构中储存与其成员有从属关系的信息。例如，假设用一个结构表示一辆汽车。如果汽车属于驾驶者，就要用一个结构成员来描述这个所有者。如果汽车被租赁，那么需要一个成员来描述其租赁公司。可以用下面的代码来完成：

```
struct owner {
    char socsecurity[12];
    ...
};

struct leasecompany {
    char name[40];
    char headquarters[40];
    ...
};

union data {
    struct owner owncar;
    struct leasecompany leasecar;
};

struct car_data {
    char make[15];
    int status; /* 私有为 0，租赁为 1 */
    union data ownerinfo;
    ...
};
```

假设 flits 是 car\_data 类型的结构变量，如果 flits.status 为 0，程序将使用 flits.ownerinfo.owncar.socsecurity，如果 flits.status 为 1，程序则使用 flits.ownerinfo.leasecar.name。

## 14.10.2 匿名联合 (C11)

匿名联合和匿名结构的工作原理相同，即匿名联合是一个结构或联合的无名联合成员。例如，我们重新定义 car\_data 结构如下：

```
struct owner {
    char socsecurity[12];
    ...
};

struct leasecompany {
    char name[40];
    char headquarters[40];
}
```

```

    ...
};

struct car_data {
    char make[15];
    int status; /* 私有为 0, 租赁为 1 */
    union {
        struct owner owncar;
        struct leasecompany leasecar;
    };
    ...
};

```

现在, 如果 flits 是 car\_data 类型的结构变量, 可以用 flits.owncar.socsecurity 代替 flits.ownerinfo.owncar.socsecurity。

### 总结: 结构和联合运算符

成员运算符: .

一般注释:

该运算符与结构或联合名一起使用, 指定结构或联合的一个成员。如果 name 是一个结构的名称, member 是该结构模版指定的一个成员名, 下面标识了该结构的这个成员:

name.member

name.member 的类型就是 member 的类型。联合使用成员运算符的方式与结构相同。

示例:

```

struct {
    int code;
    float cost;
} item;
item.code = 1265;

```

间接成员运算符: ->

一般注释:

该运算符和指向结构或联合的指针一起使用, 标识结构或联合的一个成员。假设 ptrstr 是指向结构的指针, member 是该结构模版指定的一个成员, 那么:

ptrstr->member

标识了指向结构的成员。联合使用间接成员运算符的方式与结构相同。

示例:

```

struct {
    int code;
    float cost;
} item, * ptrst;
ptrst = &item;
ptrst->code = 3451;

```

最后一条语句把一个 int 类型的值赋给 item 的 code 成员。如下 3 个表达式是等价的:

ptrst->code      item.code      (\*ptrst).code

## 14.11 枚举类型

可以用枚举类型 (*enumerated type*) 声明符号名称来表示整型常量。使用 enum 关键字, 可以创建一个新“类型”并指定它可具有的值 (实际上, enum 常量是 int 类型, 因此, 只要能使用 int 类型的地方就

可以使用枚举类型)。枚举类型的是提高程序的可读性。它的语法与结构的语法相同。例如，可以这样声明：

```
enum spectrum {red, orange, yellow, green, blue, violet};  
enum spectrum color;
```

第1个声明创建了 spectrum 作为标记名，允许把 enum spectrum 作为一个类型名使用。第2个声明使 color 作为该类型的变量。第1个声明中花括号内的标识符枚举了 spectrum 变量可能有的值。因此，color 可能的值是 red、orange、yellow 等。这些符号常量被称为枚举符 (enumerator)。然后，便可这样用：

```
int c;  
color = blue;  
if (color == yellow)  
    ...;  
for (color = red; color <= violet; color++)  
    ...;
```

虽然枚举符 (如 red 和 blue) 是 int 类型，但是枚举变量可以是任意整数类型，前提是该整数类型可以储存枚举常量。例如，spectrum 的枚举符范围是 0~5，所以编译器可以用 unsigned char 来表示 color 变量。

顺带一提，C 枚举的一些特性并不适用于 C++。例如，C 允许枚举变量使用++运算符，但是 C++ 标准不允许。所以，如果编写的代码将来会并入 C++ 程序，那么必须把上面例子中的 color 声明为 int 类型，才能 C 和 C++ 都兼容。

### 14.11.1 enum 常量

blue 和 red 到底是什么？从技术层面看，它们是 int 类型的常量。例如，假定有前面的枚举声明，可以这样写：

```
printf("red = %d, orange = %d\n", red, orange);
```

其输出如下：

```
red = 0, orange = 1
```

red 成为一个有名称的常量，代表整数 0。类似地，其他标识符都是有名称的常量，分别代表 1~5。只要是能使用整型常量的地方就可以使用枚举常量。例如，在声明数组时，可以用枚举常量表示数组的大小；在 switch 语句中，可以把枚举常量作为标签。

### 14.11.2 默认值

默认情况下，枚举列表中的常量都被赋予 0、1、2 等。因此，下面的声明中 nina 的值是 3：

```
enum kids {nippy, slats, skippy, nina, liz};
```

### 14.11.3 赋值

在枚举声明中，可以为枚举常量指定整数值：

```
enum levels {low = 100, medium = 500, high = 2000};
```

如果只给一个枚举常量赋值，没有对后面的枚举常量赋值，那么后面的常量会被赋予后续的值。例如，假设没有如下的声明：

```
enum feline {cat, lynx = 10, puma, tiger};
```

那么，cat 的值是 0 (默认)，lynx、puma 和 tiger 的值分别是 10、11、12。

#### 14.11.4 enum 的用法

枚举类型的目的就是为了提高程序的可读性和可维护性。如果要处理颜色，使用 red 和 blue 比使用 0 和 1 更直观。注意，枚举类型只能在内部使用。如果要输入 color 中 orange 的值，只能输入 1，而不是单词 orange。或者，让程序先读入字符串"orange"，再将其转换为 orange 代表的值。

因为枚举类型是整数类型，所以可以在表达式中以使用整数变量的方式使用 enum 变量。它们用在 case 语句中很方便。

程序清单 14.15 演示了一个使用 enum 的小程序。该程序示例使用默认值的方案，把 red 的值设置为 0，使之成为指向字符串"red"的指针的索引。

程序清单 14.15 enum.c 程序

```
/* enum.c -- 使用枚举类型的值 */
#include <stdio.h>
#include <string.h>    // 提供 strcmp()、strchr() 函数的原型
#include <stdbool.h>   // C99 特性
char * s_gets(char * st, int n);

enum spectrum { red, orange, yellow, green, blue, violet };
const char * colors [] = { "red", "orange", "yellow",
"green", "blue", "violet" };
#define LEN 30

int main(void)
{
    char choice[LEN];
    enum spectrum color;
    bool color_is_found = false;

    puts("Enter a color (empty line to quit):");
    while (s_gets(choice, LEN) != NULL && choice[0] != '\0')
    {
        for (color = red; color <= violet; color++)
        {
            if (strcmp(choice, colors[color]) == 0)
            {
                color_is_found = true;
                break;
            }
        }
        if (color_is_found)
            switch (color)
            {
                case red: puts("Roses are red.");
                break;
                case orange: puts("Poppies are orange.");
                break;
                case yellow: puts("Sunflowers are yellow.");
                break;
                case green: puts("Grass is green.");
                break;
                case blue: puts("Bluebells are blue.");
                break;
            }
    }
}
```

```

        break;
    case violet: puts("Violets are violet.");
        break;
    }
    else
        printf("I don't know about the color %s.\n", choice);
    color_is_found = false;
    puts("Next color, please (empty line to quit):");
}
puts("Goodbye!");

return 0;
}

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;       // 清理输入行
    }
    return ret_val;
}

```

当输入的字符串与 color 数组的成员指向的字符串相匹配时，for 循环结束。如果循环找到匹配的颜色，程序就用枚举变量的值与作为 case 标签的枚举常量匹配。下面是该程序的一个运行示例：

```

Enter a color (empty line to quit):
blue
Bluebells are blue.
Next color, please (empty line to quit):
orange
Poppies are orange.
Next color, please (empty line to quit):
purple
I don't know about the color purple.
Next color, please (empty line to quit):

Goodbye!

```

## 14.11.5 共享名称空间

C 语言使用名称空间 (*namespace*) 标识程序中的各部分，即通过名称来识别。作用域是名称空间概念的一部分：两个不同作用域的同名变量不冲突；两个相同作用域的同名变量冲突。名称空间是分类别的。在特定作用域中的结构标记、联合标记和枚举标记都共享相同的名称空间，该名称空间与普通变量使用的空间不同。这意味着在相同作用域中变量和标记的名称可以相同，不会引起冲突，但是不能在相同作用域

中声明两个同名标签或同名变量。例如，在 C 中，下面的代码不会产生冲突：

```
struct rect { double x; double y; };
int rect; // 在 C 中不会产生冲突
```

尽管如此，以两种不同的方式使用相同的标识符会造成混乱。另外，C++不允许这样做，因为它把标记名和变量名放在相同的名称空间中。

## 14.12 `typedef` 简介

`typedef` 工具是一个高级数据特性，利用 `typedef` 可以为某一类型自定义名称。这方面与 `#define` 类似，但是两者有 3 处不同：

- 与 `#define` 不同，`typedef` 创建的符号名只受限于类型，不能用于值。
- `typedef` 由编译器解释，不是预处理器。
- 在其受限范围内，`typedef` 比 `#define` 更灵活。

下面介绍 `typedef` 的工作原理。假设要用 `BYTE` 表示 1 字节的数组。只需像定义个 `char` 类型变量一样定义 `BYTE`，然后在定义前面加上关键字 `typedef` 即可：

```
typedef unsigned char BYTE;
```

随后，便可使用 `BYTE` 来定义变量：

```
BYTE x, y[10], * z;
```

该定义的作用域取决于 `typedef` 定义所在的位置。如果定义在函数中，就具有局部作用域，受限于定义所在的函数。如果定义在函数外面，就具有文件作用域。

通常，`typedef` 定义中用大写字母表示被定义的名称，以提醒用户这个类型名实际上是一个符号缩写。当然，也可以用小写：

```
typedef unsigned char byte;
```

`typedef` 中使用的名称遵循变量的命名规则。

为现有类型创建一个名称，看上去真是多此一举，但是它有时的确很有用。在前面的示例中，用 `BYTE` 替代 `unsigned char` 表明你打算用 `BYTE` 类型的变量表示数字，而不是字符码。使用 `typedef` 还能提高程序的可移植性。例如，我们之前提到的 `sizeof` 运算符的返回类型：`size_t` 类型，以及 `time()` 函数的返回类型：`time_t` 类型。`C` 标准规定 `sizeof` 和 `time()` 返回整数类型，但是让实现来决定具体是什么整数类型。其原因是，`C` 标准委员会认为没有哪个类型对于所有的计算机平台都是最优选择。所以，标准委员会决定建立一个新的类型名（如，`time_t`），并让实现使用 `typedef` 来设置它的具体类型。以这样的方式，`C` 标准提供以下通用原型：

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

`time_t` 在一个系统中是 `unsigned long`，在另一个系统中可以是 `unsigned long long`。只要包含 `time.h` 头文件，程序就能访问合适的定义，你也可以在代码中声明 `time_t` 类型的变量。

`typedef` 的一些特性与 `#define` 的功能重合。例如：

```
#define BYTE unsigned char
```

这使预处理器用 `BYTE` 替换 `unsigned char`。但是也有 `#define` 没有的功能：

```
typedef char * STRING;
```

没有 `typedef` 关键字，编译器将把 `STRING` 识别为一个指向 `char` 的指针变量。有了 `typedef` 关键字，编译器则把 `STRING` 解释成一个类型的标识符，该类型是指向 `char` 的指针。因此：

```
STRING name, sign;
```

相当于：

```
char * name, * sign;
```

但是，如果这样假设：

```
#define STRING char *
```

然后，下面的声明：

```
STRING name, sign;
```

将被翻译成：

```
char * name, sign;
```

这导致只有 name 才是指针。

还可以把 `typedef` 用于结构：

```
typedef struct complex {
    float real;
    float imag;
} COMPLEX;
```

然后便可使用 `COMPLEX` 类型代替 `complex` 结构来表示复数。使用 `typedef` 的第 1 个原因是：为经常出现的类型创建一个方便、易识别的类型名。例如，前面的例子中，许多人更倾向于使用 `STRING` 或与其等价的标记。

用 `typedef` 来命名一个结构类型时，可以省略该结构的标签：

```
typedef struct {double x; double y;} rect;
```

假设这样使用 `typedef` 定义的类型名：

```
rect r1 = {3.0, 6.0};
rect r2;
```

以上代码将被翻译成：

```
struct {double x; double y;} r1= {3.0, 6.0};
struct {double x; double y;} r2;
r2 = r1;
```

这两个结构在声明时都没有标记，它们的成员完全相同（成员名及其类型都匹配），C 认为这两个结构的类型相同，所以 `r1` 和 `r2` 间的赋值是有效操作。

使用 `typedef` 的第 2 个原因是：`typedef` 常用于给复杂的类型命名。例如，下面的声明：

```
typedef char (* FRPTC ()) [5];
```

把 `FRPTC` 声明为一个函数类型，该函数返回一个指针，该指针指向内含 5 个 `char` 类型元素的数组（参见下一节的讨论）。

使用 `typedef` 时要记住，`typedef` 并没有创建任何新类型，它只是为某个已存在的类型增加了一个方便使用的标签。以前面的 `STRING` 为例，这意味着我们创建的 `STRING` 类型变量可以作为实参传递给以指向 `char` 指针作为形参的函数。

通过结构、联合和 `typedef`，C 提供了有效处理数据的工具和处理可移植数据的工具。

## 14.13 其他复杂的声明

C 允许用户自定义数据形式。虽然我们常用的是一些简单的形式，但是根据需要有时还会用到一些复杂的形式。在一些复杂的声明中，常包含下面的符号，如表 14.1 所示：

表14.1 声明时可使用的符号

符号	含义
*	表示一个指针
()	表示一个函数
[]	表示一个数组

下面是一些较复杂的声明示例：

```
int board[8][8];           // 声明一个内含 int 数组的数组
int ** ptr;                // 声明一个指向指针的指针，被指向的指针指向 int
int * risks[10];           // 声明一个内含 10 个元素的数组，每个元素都是一个指向 int 的指针
int (* rusks)[10];         // 声明一个指向数组的指针，该数组内含 10 个 int 类型的值
int * oof[3][4];           // 声明一个 3×4 的二维数组，每个元素都是指向 int 的指针
int (* uuf)[3][4];         // 声明一个指向 3×4 二维数组的指针，该数组中内含 int 类型值
int (* uof[3])[4];          // 声明一个内含 3 个指针元素的数组，其中每个指针都指向一个内含 4 个 int 类型元素的数组
```

要看懂以上声明，关键要理解\*、()和[]的优先级。记住下面几条规则。

1. 数组名后面的[]和函数名后面的()具有相同的优先级。它们比\*（解引用运算符）的优先级高。因此下面声明的 risk 是一个指针数组，不是指向数组的指针：

```
int * risks[10];
```

2. []和()的优先级相同，由于都是从左往右结合，所以下面的声明中，在应用方括号之前，\*先与 rusks 结合。因此 rusks 是一个指向数组的指针，该数组内含 10 个 int 类型的元素：

```
int (* rusks)[10];
```

3. []和()都是从左往右结合。因此下面声明的 goods 是一个由 12 个内含 50 个 int 类型值的数组组成的二维数组，不是一个有 50 个内含 12 个 int 类型值的数组组成的二维数组：

```
int goods[12][50];
```

把以上规则应用于下面的声明：

```
int * oof[3][4];
```

[3]比\*的优先级高，由于从左往右结合，所以[3]先与 oof 结合。因此，oof 首先是一个内含 3 个元素的数组。然后再与[4]结合，所以 oof 的每个元素都是内含 4 个元素的数组。\*说明这些元素都是指针。最后，int 表明了这 4 个元素都是指向 int 的指针。因此，这条声明要表达的是：oof 是一个内含 3 个元素的数组，其中每个元素是由 4 个指向 int 的指针组成的数组。简而言之，oof 是一个 3×4 的二维数组，每个元素都是指向 int 的指针。编译器要为 12 个指针预留存储空间。

现在来看下面的声明：

```
int (* uuf)[3][4];
```

圆括号使得\*先与 uuf 结合，说明 uuf 是一个指针，所以 uuf 是一个指向 3×4 的 int 类型二维数组的指针。编译器要为一个指针预留存储空间。

根据这些规则，还可以声明：

```
char * fump(int);           // 返回字符指针的函数
char (* frump)(int);        // 指向函数的指针，该函数的返回类型为 char
char (* flump[3])(int);     // 内含 3 个指针的数组，每个指针都指向返回类型为 char 的函数
```

这 3 个函数都接受 int 类型的参数。

可以使用 `typedef` 建立一系列相关类型：

```
typedef int arr5[5];
typedef arr5 * p_arr5;
typedef p_arr5 arrp10[10];
arr5 togs; // togs 是一个内含 5 个 int 类型值的数组
p_arr5 p2; // p2 是一个指向数组的指针，该数组内含 5 个 int 类型的值
arrp10 ap; // ap 是一个内含 10 个指针的数组，每个指针都指向一个内含 5 个 int 类型值的数组
```

如果把这些放入结构中，声明会更复杂。至于应用，我们就不再进一步讨论了。

## 14.14 函数和指针

通过上一节的学习可知，可以声明一个指向函数的指针。这个复杂的玩意儿到底有何用处？通常，函数指针常用作另一个函数的参数，告诉该函数要使用哪一个函数。例如，排序数组涉及比较两个元素，以确定先后。如果元素是数字，可以使用`>`运算符；如果元素是字符串或结构，就要调用函数进行比较。`C`库中的`qsort()`函数可以处理任意类型的数组，但是要告诉`qsort()`使用哪个函数来比较元素。为此，`qsort()`函数的参数列表中，有一个参数接受指向函数的指针。然后，`qsort()`函数使用该函数提供的方案进行排序，无论这个数组中的元素是整数、字符串还是结构。

我们来进一步研究函数指针。首先，什么是函数指针？假设有一个指向`int`类型变量的指针，该指针储存着这个`int`类型变量储存在内存位置的地址。同样，函数也有地址，因为函数的机器语言实现由载入内存的代码组成。指向函数的指针中储存着函数代码的起始处的地址。

其次，声明一个数据指针时，必须声明指针所指向的数据类型。声明一个函数指针时，必须声明指针指向的函数类型。为了指明函数类型，要指明函数签名，即函数的返回类型和形参类型。例如，考虑下面的函数原型：

```
void ToUpper(char *); // 把字符串中的字符转换成大写字母
```

`ToUpper()`函数的类型是“带`char *`类型参数、返回类型是`void`的函数”。下面声明了一个指针`pf`指向该函数类型：

```
void (*pf)(char *); // pf 是一个指向函数的指针
```

从该声明可以看出，第 1 对圆括号把`*`和`pf`括起来，表明`pf`是一个指向函数的指针。因此，`(*pf)`是一个参数列表为`(char *)`、返回类型为`void`的函数。注意，把函数名`ToUpper`替换为表达式`(*pf)`是创建指向函数指针最简单的方式。所以，如果想声明一个指向某类型函数的指针，可以写出该函数的原型后把函数名替换成`(*pf)`形式的表达式，创建函数指针声明。前面提到过，由于运算符优先级的规则，在声明函数指针时必须把`*`和指针名括起来。如果省略第 1 个圆括号会导致完全不同的情况：

```
void *pf(char *); // pf 是一个返回字符指针的函数
```

### 提示

要声明一个指向特定类型函数的指针，可以先声明一个该类型的函数，然后把函数名替换成`(*pf)`形式的表达式。然后，`pf`就成为指向该类型函数的指针。

声明了函数指针后，可以把类型匹配的函数地址赋给它。在这种上下文中，函数名可以用于表示函数的地址：

```
void ToUpper(char *);
```

```

void ToLower(char *);
int round(double);
void (*pf)(char *);

pf = ToUpper;      // 有效, ToUpper 是该类型函数的地址
pf = ToLower;      // 有效, ToUpper 是该类型函数的地址
pf = round;        // 无效, round 与指针类型不匹配
pf = ToLower();    // 无效, ToLower() 不是地址

```

最后一条语句是无效的，不仅因为 `ToLower()` 不是地址，而且 `ToLower()` 的返回类型是 `void`，它没有返回值，不能在赋值语句中进行赋值。注意，指针 `pf` 可以指向其他带 `char *` 类型参数、返回类型是 `void` 的函数，不能指向其他类型的函数。

既然可以用数据指针访问数据，也可以用函数指针访问函数。奇怪的是，有两种逻辑上不一致的语法可以这样做，下面解释：

```

void ToUpper(char *);
void ToLower(char *);
void (*pf)(char *);
char mis[] = "Nina Metier";
pf = ToUpper;
(*pf)(mis);    // 把 ToUpper 作用于 (语法 1)
pf = ToLower;
pf(mis);       // 把 ToLower 作用于 (语法 2)

```

这两种方法看上去都合情合理。先分析第 1 种方法：由于 `pf` 指向 `ToUpper` 函数，那么 `*pf` 就相当于 `ToUpper` 函数，所以表达式 `(*pf)(mis)` 和 `ToUpper(mis)` 相同。从 `ToUpper` 函数和 `pf` 的声明就能看出，`ToUpper` 和 `(*pf)` 是等价的。第 2 种方法：由于函数名是指针，那么指针和函数名可以互换使用，所以 `pf(mis)` 和 `ToUpper(mis)` 相同。从 `pf` 的赋值表达式语句就能看出 `ToUpper` 和 `pf` 是等价的。由于历史的原因，贝尔实验室的 C 和 UNIX 的开发者采用第 1 种形式，而伯克利的 UNIX 推广者却采用第 2 种形式。K&R C 不允许第 2 种形式。但是，为了与现有代码兼容，ANSI C 认为这两种形式（本例中是 `(*pf)(mis)` 和 `pf(mis)`）等价。后续的标准也延续了这种矛盾的和谐。

作为函数的参数是数据指针最常见的用法之一，函数指针亦如此。例如，考虑下面的函数原型：

```
void show(void (* fp)(char *), char * str);
```

这看上去让人头晕。它声明了两个形参：`fp` 和 `str`。`fp` 形参是一个函数指针，`str` 是一个数据指针。更具体地说，`fp` 指向的函数接受 `char *` 类型的参数，其返回类型为 `void`；`str` 指向一个 `char` 类型的值。因此，假设有上面的声明，可以这样调用函数：

```
show(ToLower, mis);    /* show() 使用 ToLower() 函数: fp = ToLower */
show(pf, mis);         /* show() 使用 pf 指向的函数: fp = pf */
```

`show()` 如何使用传入的函数指针？是用 `fp()` 语法还是 `(*fp)()` 语法调用函数：

```

void show(void (* fp)(char *), char * str)
{
    (*fp)(str); /* 把所选函数作用于 str */
    puts(str);   /* 显示结果 */
}

```

例如，这里的 `show()` 首先用 `fp` 指向的函数转换 `str`，然后显示转换后的字符串。

顺带一提，把带返回值的函数作为参数传递给另一个函数有两种不同的方法。例如，考虑下面的语句：

```
function1(sqrt);      /* 传递 sqrt() 函数的地址 */
function2(sqrt(4.0)); /* 传递 sqrt() 函数的返回值 */
```

第1条语句传递的是 `sqrt()` 函数的地址，假设 `function1()` 在其代码中会使用该函数。第2条语句先调用 `sqrt()` 函数，然后求值，并把返回值（该例中是 2.0）传递给 `function2()`。

程序清单 14.16 中的程序通过 `show()` 函数来演示这些要点，该函数以各种转换函数作为参数。该程序也演示了一些处理菜单的有用技巧。

程序清单 14.16 func\_ptr.c 程序

```
// func_ptr.c -- 使用函数指针
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LEN 81
char * s_gets(char * st, int n);
char showmenu(void);
void eatline(void);           // 读取至行末尾
void show(void(*fp)(char *), char * str);
void ToUpper(char *);         // 把字符串转换为大写
void ToLower(char *);         // 把字符串转换为小写
void Transpose(char *);       // 大小写转置
void Dummy(char *);           // 不更改字符串

int main(void)
{
    char line[LEN];
    char copy[LEN];
    char choice;
    void(*pfun)(char *); // 声明一个函数指针，被指向的函数接受 char *类型的参数，无返回值

    puts("Enter a string (empty line to quit):");
    while (s_gets(line, LEN) != NULL && line[0] != '\0')
    {
        while ((choice = showmenu()) != 'n')
        {
            switch (choice) // switch 语句设置指针
            {
                case 'u': pfun = ToUpper; break;
                case 'l': pfun = ToLower; break;
                case 't': pfun = Transpose; break;
                case 'o': pfun = Dummy; break;
            }
            strcpy(copy, line); // 为 show() 函数拷贝一份
            show(pfun, copy); // 根据用户的选择，使用选定的函数
        }
        puts("Enter a string (empty line to quit):");
    }
    puts("Bye!");
}

return 0;
}

char showmenu(void)
{
    char ans;
```

```

puts("Enter menu choice:");
puts("u) uppercase      l) lowercase");
puts("t) transposed case o) original case");
puts("n) next string");
ans = getchar();           // 获取用户的输入
ans = tolower(ans);        // 转换为小写
eatline();                 // 清理输入行
while (strchr("ulton", ans) == NULL)
{
    puts("Please enter a u, l, t, o, or n:");
    ans = tolower(getchar());
    eatline();
}

return ans;
}

void eatline(void)
{
    while (getchar() != '\n')
        continue;
}

void ToUpper(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
}

void ToLower(char * str)
{
    while (*str)
    {
        *str = tolower(*str);
        str++;
    }
}

void Transpose(char * str)
{
    while (*str)
    {
        if (islower(*str))
            *str = toupper(*str);
        else if (isupper(*str))
            *str = tolower(*str);
        str++;
    }
}

void Dummy(char * str)
{
}

```

```

// 不改变字符串
}

void show(void(*fp)(char *), char * str)
{
    (*fp)(str); // 把用户选定的函数作用于 str
    puts(str); // 显示结果
}

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; // 清理输入行中剩余的字符
    }
    return ret_val;
}

```

下面是该程序的输出示例：

```

Enter a string (empty line to quit):
Does C make you feel loopy?
Enter menu choice:
u) uppercase l) lowercase
t) transposed case o) original case
n) next string
t
DOES c MAKE YOU FEEL LOOPY?
Enter menu choice:
u) uppercase l) lowercase
t) transposed case o) original case
n) next string
l
does c make you feel loopy?
Enter menu choice:
u) uppercase l) lowercase
t) transposed case o) original case
n) next string
n
Enter a string (empty line to quit):

```

Bye!

注意，ToUpper()、ToLower()、Transpose()和Dummy()函数的类型都相同，所以这4个函数都可以赋给pfun指针。该程序把pfun作为show()的参数，但是也可以直接把这4个函数中的任一个函

数名作为参数，如 `show(Transpose, copy)`。

这种情况下，可以使用 `typedef`。例如，该程序中可以这样写：

```
typedef void (*V_FP_CHARP)(char *);
void show(V_FP_CHARP fp, char *);
```

`V_FP_CHARP pfun;`

如果还想更复杂一些，可以声明并初始化一个函数指针的数组：

```
V_FP_CHARP arpf[4] = {ToUpper, ToLower, Transpose, Dummy};
```

然后把 `showmenu()` 函数的返回类型改为 `int`，如果用户输入 `u`，则返回 0；如果用户输入 `1`，则返回 2；如果用户输入 `t`，则返回 2，以此类推。可以把程序中的 `switch` 语句替换成下面的 `while` 循环：

```
index = showmenu();
while (index >= 0 && index <= 3)
{
    strcpy(copy, line);           /* 为 show() 拷贝一份 */
    show(arpf[index], copy);     /* 使用选定的函数 */
    index = showmenu();
}
```

虽然没有函数数组，但是可以有函数指针数组。

以上介绍了使用函数名的 4 种方法：定义函数、声明函数、调用函数和作为指针。图 14.4 进行了总结。

函数原型中的函数名：	<code>int comp(int x, int y);</code>
函数调用中的函数名：函数定义中的函数名：	<code>status = comp(q,r);</code>
	<code>int comp(intx, inty)</code>
	{ ... }
在赋值表达式语句中作为指针的函数名：	<code>pfunct = comp;</code>
作为指针参数的函数名：	<code>slowsort(arr,n,comp);</code>

图 14.4 函数名的用法

至于如何处理菜单，`showmenu()` 函数给出了几种技巧。首先，下面的代码：

```
ans = getchar();      // 获取用户输入
ans = tolower(ans);  // 转换成小写
```

和

```
ans = tolower(getchar());
```

演示了转换用户输入的两种方法。这两种方法都可以把用户输入的字符转换为一种大小写形式，这样就不用检测用户输入的是 '`u`' 还是 '`U`'，等等。

`eatline()` 函数丢弃输入行中的剩余字符，在处理这两种情况时很有用。第一，用户为了输入一个选择，输入一个字符，然后按下 `Enter` 键，将产生一个换行符。如果不处理这个换行符，它将成为下一次读取的第一个字符。第二，假设用户输入的是整个单词 `uppercase`，而不是一个字母 `u`。如果没有 `eatline()` 函数，程序会把 `uppercase` 中的字符作为用户的响应依次读取。有了 `eatline()`，程序会读取 `u` 字符并丢弃输入行中剩余的字符。

其次，`showmenu()` 函数的设计意图是，只给程序返回正确的选项。为完成这项任务，程序使用了 `string.h` 头文件中的标准库函数 `strchr()`：

```
while (strchr("ulton", ans) == NULL)
```

该函数在字符串 "ulton" 中查找字符 `ans` 首次出现的位置，并返回一个指向该字符的指针。如果没有找到该字符，则返回空指针。因此，上面的 `while` 循环头可以用下面的 `while` 循环头代替，但是上面的

用起来更方便：

```
while (ans != 'u' && ans != 'l' && ans != 't' && ans != 'o' && ans != 'n')
```

待检查的项越多，使用 `strchr()` 就越方便。

## 14.15 关键概念

我们在编程中要表示的信息通常不只是一个数字或一些列数字。程序可能要处理具有多种属性的实体。例如，通过姓名、地址、电话号码和其他信息表示一名客户；或者，通过电影名、发行人、播放时长、售价等表示一部电影 DVD。C 结构可以把这些信息都放在一个单元内。在组织程序时这很重要，因为这样可以把相关的信息都储存在一处，而不是分散储存在多个变量中。

设计结构时，开发一个与之配套的函数包通常很有用。例如，写一个以结构（或结构的地址）为参数的函数打印结构内容，比用一堆 `printf()` 语句强得多。因为只需要一个参数就能打印结构中的所有信息。如果把信息放到零散的变量中，每个部分都需要一个参数。另外，如果要在结构中增加一个成员，只需重写函数，不必改写函数调用。这在修改结构时很方便。

联合声明与结构声明类似。但是，联合的成员共享相同的存储空间，而且在联合中同一时间内只能有一个成员。实质上，可以在联合变量中储存一个类型不唯一的值。

`enum` 工具提供一种定义符号常量的方法，`typedef` 工具提供一种为基本或派生类型创建新标识符的方法。

指向函数的指针提供一种告诉函数应使用哪一个函数的方法。

## 14.16 本章小结

C 结构提供在相同的数据对象中储存多个不同类型数据项的方法。可以使用标记来标识一个具体的结构模板，并声明该类型的变量。通过成员点运算符（`.`）可以使用结构模版中的标签来访问结构的各个成员。

如果有一个指向结构的指针，可以用该指针和间接成员运算符（`->`）代替结构名和点运算符来访问结构的各成员。和数组不同，结构名不是结构的地址，要在结构名前使用`&`运算符才能获得结构的地址。

一贯以来，与结构相关的函数都使用指向结构的指针作为参数。现在的 C 允许把结构作为参数传递，作为返回值和同类型结构之间赋值。然而，传递结构的地址通常更有效。

联合使用与结构相同的语法。然而，联合的成员共享一个共同的存储空间。联合同一时间内只能储存一个单独的数据项，不像结构那样同时储存多种数据类型。也就是说，结构可以同时储存一个 `int` 类型数据、一个 `double` 类型数据和一个 `char` 类型数据，而相应的联合只能保存一个 `int` 类型数据，或者一个 `double` 类型数据，或者一个 `char` 类型数据。

通过枚举可以创建一系列代表整型常量（枚举常量）的符号和定义相关联的枚举类型。

`typedef` 工具可用于建立 C 标准类型的别名或缩写。

函数名代表函数的地址，可以把函数的地址作为参数传递给其他函数，然后这些函数就可以使用被指向的函数。如果把特定函数的地址赋给一个名为 `pf` 的函数指针，可以通过以下两种方式调用该函数：

```
#include <math.h> /* 提供 sin() 函数的原型: double sin(double) */

...
double (*pdf)(double);
double x;
pdf = sin;
```

```
x = (*pdf)(1.2); // 调用 sin(1.2)
x = pdf(1.2); // 同样调用 sin(1.2)
```

## 14.17 复习题

复习题的参考答案在附录 A 中。

- 下面的结构模板有什么问题：

```
structure {
    char itable;
    int num[20];
    char * togs
}
```

- 下面是程序的一部分，输出是什么？

```
#include <stdio.h>
struct house {
    float sqft;
    int rooms;
    int stories;
    char address[40];
};
int main(void)
{
    struct house fruzt = {1560.0, 6, 1, "22 Spiffo Road"};
    struct house *sign;

    sign = &fruzt;
    printf("%d %d\n", fruzt.rooms, sign->stories);
    printf("%s \n", fruzt.address);
    printf("%c %c\n", sign->address[3], fruzt.address[4]);
    return 0;
}
```

- 设计一个结构模板储存一个月份名、该月份名的 3 个字母缩写、该月的天数以及月份号。
- 定义一个数组，内含 12 个结构（第 3 题的结构类型）并初始化为一个年份（非闰年）。
- 编写一个函数，用户提供月份号，该函数就返回一年中到该月为止（包括该月）的总天数。假设在所有函数的外部声明了第 3 题的结构模版和一个该类型结构的数组。
- a. 假设有下面的 `typedef`，声明一个内含 10 个指定结构的数组。然后，单独给成员赋值（或等价字符串），使第 3 个元素表示一个焦距长度有 500mm，孔径为 f/2.0 的 Remarkata 镜头。

```
typedef struct lens { /* 描述镜头 */
    float foclen; /* 焦距长度，单位为 mm */
    float fstop; /* 孔径 */
    char brand[30]; /* 品牌名称 */
} LENS;
```

b. 重写 a，在声明中使用一个待指定初始化器的初始化列表，而不是对每个成员单独赋值。

- 考虑下面程序片段：

```
struct name {
    char first[20];
    char last[20];
};
```

```

struct bem {
    int limbs;
    struct name title;
    char type[30];
};

struct bem * pb;
struct bem deb = {
    6,
    { "Berbnazel", "Gwolkapwolk" },
    "Arcturan"
};

```

pb = &deb;

- a. 下面的语句分别打印什么?

```

printf("%d\n", deb.limbs);
printf("%s\n", pb->type);
printf("%s\n", pb->type + 2);

```

- b. 如何用结构表示法(两种方法)表示"Gwolkapwolk"?

- c. 编写一个函数,以 bem 结构的地址作为参数,并以下面的形式输出结构的内容(假定结构模板在一个名为 starfolk.h 的头文件中):

Berbnazel Gwolkapwolk is a 6-limbed Arcturan.

8. 考虑下面的声明:

```

struct fullname {
    char fname[20];
    char lname[20];
};

struct bard {
    struct fullname name;
    int born;
    int died;
};

struct bard willie;
struct bard *pt = &willie;

```

- a. 用 willie 标识符标识 willie 结构的 born 成员。

- b. 用 pt 标识符标识 willie 结构的 born 成员。

- c. 调用 scanf() 读入一个用 willie 标识符标识的 born 成员的值。

- d. 调用 scanf() 读入一个用 pt 标识符标识的 born 成员的值。

- e. 调用 scanf() 读入一个用 willie 标识符标识的 name 成员中 lname 成员的值。

- f. 调用 scanf() 读入一个用 pt 标识符标识的 name 成员中 lname 成员的值。

- g. 构造一个标识符,标识 willie 结构变量所表示的姓名中名的第 3 个字母(英文的名在前)。

- h. 构造一个表达式,表示 willie 结构变量所表示的名和姓中的字母总数。

9. 定义一个结构模板以储存这些项:汽车名、马力、EPA(美国环保局)城市交通 MPG(每加仑燃料行驶的英里数)评级、轴距和出厂年份。使用 car 作为该模版的标记。

10. 假设有如下结构:

```

struct gas {
    float distance;
    float gals;
}

```

```
    float mpg;
};
```

- a. 设计一个函数，接受 struct gas 类型的参数。假设传入的结构包含 distance 和 gals 信息。该函数为 mpg 成员计算正确的值，并把值返回该结构。
- b. 设计一个函数，接受 struct gas 类型的参数。假设传入的结构包含 distance 和 gals 信息。该函数为 mpg 成员计算正确的值，并把该值赋给合适的成员。
11. 声明一个标记为 choices 的枚举，把枚举常量 no、yes 和 maybe 分别设置为 0、1、2。
12. 声明一个指向函数的指针，该函数返回指向 char 的指针，接受一个指向 char 的指针和一个 char 类型的值。
13. 声明 4 个函数，并初始化一个指向这些函数的指针数组。每个函数都接受两个 double 类型的参数，返回 double 类型的值。另外，用两种方法使用该数组调用带 10.0 和 2.5 实参的第 2 个函数。

## 14.18 编程练习

1. 重新编写复习题 5，用月份名的拼写代替月份号（别忘了使用 strcmp()）。在一个简单的程序中测试该函数。
2. 编写一个函数，提示用户输入日、月和年。月份可以是月份号、月份名或月份名缩写。然后该程序应返回一年中到用户指定日子（包括这一天）的总天数。
3. 修改程序清单 14.2 中的图书目录程序，使其按照输入图书的顺序输出图书的信息，然后按照标题字母的声明输出图书的信息，最后按照价格的升序输出图书的信息。
4. 编写一个程序，创建一个有两个成员的结构模板：
  - a. 第 1 个成员是社会保险号，第 2 个成员是一个有 3 个成员的结构，第 1 个成员代表名，第 2 个成员代表中间名，第 3 个成员表示姓。创建并初始化一个内含 5 个该类型结构的数组。该程序以下面的格式打印数据：  
Dribble, Flossie M. -- 302039823  
如果有中间名，只打印它的第 1 个字母，后面加一个点(.)；如果没有中间名，则不用打印点。编写一个程序进行打印，把结构数组传递给这个函数。
  - b. 修改 a 部分，传递结构的值而不是结构的地址。
5. 编写一个程序满足下面的要求。
  - a. 外部定义一个有两个成员的结构模板 name：一个字符串储存名，一个字符串储存姓。
  - b. 外部定义一个有 3 个成员的结构模板 student：一个 name 类型的结构，一个 grade 数组储存 3 个浮点型分数，一个变量储存 3 个分数平均数。
  - c. 在 main() 函数中声明一个内含 CSIZE (CSIZE = 4) 个 student 类型结构的数组，并初始化这些结构的名字部分。用函数执行 g、e、f 和 g 中描述的任务。
  - d. 以交互的方式获取每个学生的成绩，提示用户输入学生的姓名和分数。把分数储存到 grade 数组相应的结构中。可以在 main() 函数或其他函数中用循环来完成。
  - e. 计算每个结构的平均分，并把计算后的值赋给合适的成员。
  - f. 打印每个结构的信息。

- g. 打印班级的平均分，即所有结构的数值成员的平均值。
6. 一个文本文件中保存着一个垒球队的信息。每行数据都是这样排列：
- ```
4 Jessie Joybat 5 2 1 1
```
- 第1项是球员号，为方便起见，其范围是0~18。第2项是球员的名。第3项是球员的姓。名和姓都是一个单词。第4项是官方统计的球员上场次数。接着3项分别是击中数、走垒数和打点(RBI)。文件可能包含多场比赛的数据，所以同一位球员可能有多行数据，而且同一位球员的多行数据之间可能有其他球员的数据。编写一个程序，把数据储存到一个结构数组中。该结构中的成员要分别表示球员的名、姓、上场次数、击中数、走垒数、打点和安打率(稍后计算)。可以使用球员号作为数组的索引。该程序要读到文件结尾，并统计每位球员的各项累计总和。
- 世界棒球统计与之相关。例如，一次走垒和触垒中的失误不计入上场次数，但是可能产生一个RBI。但是该程序要做的是像下面描述的一样读取和处理数据文件，不会关心数据的实际含义。
- 要实现这些功能，最简单的方法是把结构的内容都初始化为零，把文件中的数据读入临时变量中，然后将其加入相应的结构中。程序读完文件后，应计算每位球员的安打率，并把计算结果储存到结构的相应成员中。计算安打率是用球员的累计击中数除以上场累计次数。这是一个浮点数计算。最后，程序结合整个球队的统计数据，一行显示一位球员的累计数据。
7. 修改程序清单14.14，从文件中读取每条记录并显示出来，允许用户删除记录或修改记录的内容。如果删除记录，把空出来的空间留给下一个要读入的记录。要修改现有的文件内容，必须用“r+b”模式，而不是“a+b”模式。而且，必须更加注意定位文件指针，防止新加入的记录覆盖现有记录。最简单的方法是改动储存在内存中的所有数据，然后再把最后的信息写入文件。跟踪的一个方法是在book结构中添加一个成员表示是否该项被删除。
8. 巨人航空公司的机群由12个座位的飞机组成。它每天飞行一个航班。根据下面的要求，编写一个座位预订程序。
- 该程序使用一个内含12个结构的数组。每个结构中包括：一个成员表示座位编号、一个成员表示座位是否已被预订、一个成员表示预订人的名、一个成员表示预订人的姓。
  - 该程序显示下面的菜单：
- ```
To choose a function, enter its letter label:  
a) Show number of empty seats  
b) Show list of empty seats  
c) Show alphabetical list of seats  
d) Assign a customer to a seat assignment  
e) Delete a seat assignment  
f) Quit
```
- 该程序能成功执行上面给出的菜单。选择d)和e)要提示用户进行额外输入，每个选项都能让用户中止输入。
  - 执行特定程序后，该程序再次显示菜单，除非用户选择f)。
9. 巨人航空公司(编程练习8)需要另一架飞机(容量相同)，每天飞4班(航班102、311、444和519)。把程序扩展为可以处理4个航班。用一个顶层菜单提供航班选择和退出。选择一个特定航班，就会出现和编程练习8类似的菜单。但是该菜单要添加一个新选项：确认座位分配。而且，菜单中的退出是返回顶层菜单。每次显示都要指明当前正在处理的航班号。另外，座位分配显示要指明确认状态。
10. 编写一个程序，通过一个函数指针数组实现菜单。例如，选择菜单中的a，将激活由该数组第1

个元素指向的函数。

11. 编写一个名为 `transform()` 的函数，接受 4 个参数：内含 `double` 类型数据的源数组名、内含 `double` 类型数据的目标数组名、一个表示数组元素个数的 `int` 类型参数、函数名（或等价的函数指针）。`transform()` 函数应把指定函数应用于源数组中的每个元素，并把返回值储存在目标数组中。例如：

```
transform(source, target, 100, sin);
```

该声明会把 `target[0]` 设置为 `sin(source[0])`，等等，共有 100 个元素。在一个程序中调用 `transform()` 4 次，以测试该函数。分别使用 `math.h` 函数库中的两个函数以及自定义的两个函数作为参数。

# 第 15 章

## 位操作

本章介绍以下内容：

- 运算符：~、&、|、^、  
<<、>>  
&=、|=、^=、>>=、<<=
- 二进制、十进制和十六进制记数法（复习）
- 处理一个值中的位的两个 C 工具：位运算符和位字段
- 关键字：\_Alignas、\_Alignof

在 C 语言中，可以单独操控变量中的位。读者可能好奇，竟然有人想这样做。有时必须单独操控位，而且非常有用。例如，通常向硬件设备发送一两个字节来控制这些设备，其中每个位（bit）都有特定的含义。另外，与文件相关的操作系统信息经常被储存，通过使用特定位表明特定项。许多压缩和加密操作都是直接处理单独的位。高级语言一般不会处理这级别的细节，C 在提供高级语言便利的同时，还能在为汇编语言所保留的级别上工作，这使其成为编写设备驱动程序和嵌入式代码的首选语言。

首先要介绍位、字节、二进制记数法和其他进制记数系统的一些背景知识。

### 15.1 二进制数、位和字节

通常都是基于数字 10 来书写数字。例如 2157 的千位是 2，百位是 1，十位是 5，个位是 7，可以写成：

$$2 \times 1000 + 1 \times 100 + 5 \times 10 + 7 \times 1$$

注意，1000 是 10 的立方（即 3 次幂），100 是 10 的平方（即 2 次幂），10 是 10 的 1 次幂，而且 10（以及任意正数）的 0 次幂是 1。因此，2157 也可以写成：

$$2 \times 10^3 + 1 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

因为这种书写数字的方法是基于 10 的幂，所以称以 10 为基底书写 2157。

姑且认为十进制系统得以发展是得益于我们都有 10 根手指。从某种意义上讲，计算机的位只有 2 根手指，因为它只能被设置为 0 或 1，关闭或打开。因此，计算机适用基底为 2 的数制系统。它用 2 的幂而不是 10 的幂。以 2 为基底表示的数字被称为二进制数（binary number）。二进制中的 2 和十进制中的 10 作用相同。例如，二进制数 1101 可表示为：

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

以十进制数表示为：

$$1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13$$

用二进制系统可以把任意整数（如果有足够的位）表示为 0 和 1 的组合。由于数字计算机通过关闭和打开状态的组合来表示信息，这两种状态分别用 0 和 1 来表示，所以使用这套数制系统非常方便。接下来，我们来学习二进制系统如何表示 1 字节的整数。

### 15.1.1 二进制整数

通常，1 字节包含 8 位。C 语言用字节 (byte) 表示储存系统字符集所需的大小，所以 C 字节可能是 8 位、9 位、16 位或其他值。不过，描述存储器芯片和数据传输率中所用的字节指的是 8 位字节。为了简化起见，本章假设 1 字节是 8 位（计算机界通常用八位组 (octet) 这个术语特指 8 位字节）。可以从左往右给这 8 位分别编号为 7~0。在 1 字节中，编号是 7 的位被称为高阶位 (high-order bit)，编号是 0 的位被称为低阶位 (low-order bit)。每 1 位的编号对应 2 的相应指数。因此，可以根据图 15.1 所示的例子理解字节。



图 15.1 位编号和位值

这里，128 是 2 的 7 次幂，以此类推。该字节能表示的最大数字是把所有位都设置为 1：11111111。这个二进制数的值是：

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

而该字节最小的二进制数是 00000000，其值为 0。因此，1 字节可储存 0~255 范围内的数字，总共 256 个值。或者，通过不同的方式解释位组合 (bit pattern)，程序可以用 1 字节储存 -128~+127 范围内的整数，总共还是 256 个值。例如，通常 `unsigned char` 用 1 字节表示的范围是 0~255，而 `signed char` 用 1 字节表示的范围是 -128~+127。

### 15.1.2 有符号整数

如何表示有符号整数取决于硬件，而不是 C 语言。也许表示有符号数最简单的方式是用 1 位（如，高阶位）储存符号，只剩下 7 位表示数字本身（假设储存在 1 字节中）。用这种符号量 (sign-magnitude) 表示法，10000001 表示 -1，00000001 表示 1。因此，其表示范围是 -127~+127。

这种方法的缺点是有两个 0：+0 和 -0。这很容易混淆，而且用两个位组合来表示一个值也有些浪费。

二进制补码 (two's-complement) 方法避免了这个问题，是当今最常用的系统。我们将以 1 字节为例，讨论这种方法。二进制补码用 1 字节中的后 7 位表示 0~127，高阶位设置为 0。目前，这种方法和符号量的方法相同。另外，如果高阶位是 1，表示的值为负。这两种方法的区别在于如何确定负值。从一个 9 位组合 100000000 (256 的二进制形式) 减去一个负数的位组合，结果是该负值的量。例如，假设一个负值的位组合是 10000000，作为一个无符号字节，该组合为表示 128；作为一个有符号值，该组合表示负值（编码是 7 的位为 1），而且值为 100000000-10000000，即 1000000 (128)。因此，该数是 -128 (在符号量表示法中，该位组合表示 -0)。类似地，10000001 是 -127，11111111 是 -1。该方法可以表示 -128~+127 范围内的数。

要得到一个二进制补码数的相反数，最简单的方法是反转每一位（即 0 变为 1，1 变为 0），然后加 1。因为 1 是 00000001，那么 -1 则是 11111110+1，或 11111111。这与上面的介绍一致。

二进制反码 (one's-complement) 方法通过反转位组合中的每一位形成一个负数。例如，00000001 是 1，那么 11111110 是 -1。这种方法也有一个 -0：11111111。该方法能表示 -127~+127 之间的数。

### 15.1.3 二进制浮点数

浮点数分两部分储存：二进制小数和二进制指数。下面我们将详细介绍。

#### 1. 二进制小数

一个普通的浮点数 0.527，表示如下：

$$5/10 + 2/100 + 7/1000$$

从左往右，各分母都是 10 的递增次幂。在二进制小数中，使用 2 的幂作为分母，所以二进制小数 .101 表示为：

$$1/2 + 0/4 + 1/8$$

用十进制表示法为：

$$0.50 + 0.00 + 0.125$$

即是 0.625。

许多分数（如， $1/3$ ）不能用十进制表示法精确地表示。与此类似，许多分数也不能用二进制表示法准确地表示。实际上，二进制表示法只能精确地表示多个  $1/2$  的幂的和。因此， $3/4$  和  $7/8$  可以精确地表示为二进制小数，但是  $1/3$  和  $2/5$  却不能。

#### 2. 浮点数表示法

为了在计算机中表示一个浮点数，要留出若干位（因系统而异）储存二进制分数，其他位储存指数。一般而言，数字的实际值是由二进制小数乘以 2 的指定次幂组成。例如，一个浮点数乘以 4，那么二进制小数不变，其指数乘以 2，二进制分数不变。如果一份浮点数乘以一个不是 2 的幂的数，会改变二进制小数部分，如有必要，也会改变指数部分。

## 15.2 其他进制数

计算机界通常使用八进制记数系统和十六进制记数系统。因为 8 和 16 都是 2 的幂，这些系统比十进制系统更接近计算机的二进制系统。

### 15.2.1 八进制

八进制（octal）是指八进制记数系统。该系统基于 8 的幂，用 0~7 表示数字（正如十进制用 0~9 表示数字一样）。例如，八进制数 451（在 C 中写作 0451）表示为：

$$4 \times 8^2 + 5 \times 8^1 + 1 \times 8^0 = 297 \text{ (十进制)}$$

了解八进制的一个简单的方法是，每个八进制位对应 3 个二进制位。表 15.1 列出了这种对应关系。这种关系使得八进制与二进制之间的转换很容易。例如，八进制数 0377 的二进制形式是 11111111。即，用 111 替代 0377 中的最后一个 7，再用 111 替换倒数第 2 个 7，最后用 011 替代 3，并舍去第 1 位的 0。这表明比 0377 大的八进制要用多个字节表示。这是八进制唯一不方便的地方：一个 3 位的八进制数可能要用 9 位二进制数来表示。注意，将八进制数转换为二进制形式时，不能去掉中间的 0。例如，八进制数 0173 的二进制形式是 01111011，不是 01111111。

表 15.1 与八进制位等价的二进制位

八进制位	等价的二进制位	八进制位	等价的二进制位
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

## 15.2.2 十六进制

十六进制 (*hexadecimal* 或 *hex*) 是指十六进制记数系统。该系统基于 16 的幂，用 0~15 表示数字。但是，由于没有单独的数 (*digit*, 即 0~9 这样单独一位的数) 表示 10~15，所以用字母 A~F 来表示。例如，十六进制数 A3F (在 C 中写作 0xA3F) 表示为：

$$10 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 = 2623 \text{ (十进制)}$$

由于 A 表示 10, F 表示 15。在 C 语言中，A~F 既可用小写也可用大写。因此，2623 也可写作 0xa3f。

每个十六进制位都对应一个 4 位的二进制数 (即 4 个二进制位)，那么两个十六进制位恰好对应一个 8 位字节。第 1 个十六进制表示前 4 位，第 2 个十六进制表示后 4 位。因此，十六进制很适合表示字节值。

表 15.2 列出了各进制之间的对应关系。例如，十六进制值 0xC2 可转换为 11000010。相反，二进制值 11010101 可以看作是 1101 0101，可转换为 0xD5。

表 15.2 十进制、十六进制和等价的二进制

十进制	十六进制	等价二进制	十进制	十六进制	等价二进制
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

介绍了位和字节的相关内容，接下来我们研究 C 用位和字节进行哪些操作。C 有两个操控位的工具。第 1 个工具是一套 (6 个) 作用于位的按位运算符。第 2 个工具是字段 (*field*) 数据形式，用于访问 int 中的位。下面将简要介绍这些 C 的特性。

## 15.3 C 按位运算符

C 提供按位逻辑运算符和移位运算符。在下面的例子中，为了方便读者了解位的操作，我们用二进制记数法写出值。但是在实际的程序中不必这样，用一般形式的整型变量或常量即可。例如，在程序中用 25 或 031 或 0x19，而不是 00011001。另外，下面的例子均使用 8 位二进制数，从左往右每位的编号为 7~0。

### 15.3.1 按位逻辑运算符

4个按位逻辑运算符都用于整型数据，包括char。之所以叫作按位(bitwise)运算，是因为这些操作都是针对每一个位进行，不影响它左右两边的位。不要把这些运算符与常规的逻辑运算符(&&、||和!)混淆，常规的逻辑运算符操作的是整个值。

#### 1. 二进制反码或按位取反：~

一元运算符~把1变为0，把0变为1。如下例子所示：

```
~(10011010) // 表达式
(01100101) // 结果值
```

假设val的类型是unsigned char，已被赋值为2。在二进制中，00000010表示2。那么，~val的值是11111101，即253。注意，该运算符不会改变val的值，就像 $3 * \text{val}$ 不会改变val的值一样，val仍然是2。但是，该运算符确实创建了一个可以使用或赋值的新值：

```
newval = ~val;
printf("%d", ~val);
```

如果要把val的值改为~val，使用下面这条语句：

```
val = ~val;
```

#### 2. 按位与：&

二元运算符&通过逐位比较两个运算对象，生成一个新值。对于每个位，只有两个运算对象中相应的位都为1时，结果才为1（从真/假方面看，只有当两个位都为真时，结果才为真）。因此，对下面的表达式求值：

```
(10010011) & (00111101) // 表达式
```

由于两个运算对象中编号为4和0的位都为1，得：

```
(00010001) // 结果值
```

C有一个按位与和赋值结合的运算符：&=。下面两条语句产生的最终结果相同：

```
val &= 0377;
val = val & 0377;
```

#### 3. 按位或：|

二元运算符|，通过逐位比较两个运算对象，生成一个新值。对于每个位，如果两个运算对象中相应的位为1，结果就为1（从真/假方面看，如果两个运算对象中相应的一个位为真或两个位都为真，那么结果为真）。因此，对下面的表达式求值：

```
(10010011) | (00111101) // 表达式
```

除了编号为6的位，这两个运算对象的其他位至少有一个位为1，得：

```
(10111111) // 结果值
```

C有一个按位或和赋值结合的运算符：|=。下面两条语句产生的最终作用相同：

```
val |= 0377;
val = val | 0377;
```

#### 4. 按位异或：^

二元运算符^逐位比较两个运算对象。对于每个位，如果两个运算对象中相应的位一个为1（但不是两个为1），结果为1（从真/假方面看，如果两个运算对象中相应的一个位为真且不是两个为同为1，那么结

果为真)。因此, 对下面表达式求值:

```
(10010011) ^ (00111101) // 表达式
```

编号为 0 的位都是 1, 所以结果为 0, 得:

```
(10101110) // 结果值
```

C 有一个按位异或和赋值结合的运算符: ^=。下面两条语句产生的最终作用相同:

```
val ^= 0377;
val = val ^= 0377;
```

### 15.3.2 用法: 掩码

按位与运算符常用于掩码 (mask)。所谓掩码指的是一些设置为开 (1) 或关 (0) 的位组合。要明白称其为掩码的原因, 先来看通过&把一个量与掩码结合后发生什么情况。例如, 假设定义符号常量 MASK 为 2 (即, 二进制形式为 00000010), 只有 1 号位是 1, 其他位都是 0。下面的语句:

```
flags = flags & MASK;
```

把 flags 中除 1 号位以外的所有位都设置为 0, 因为使用按位与运算符 (&) 任何位与 0 组合都得 0。1 号位的值不变 (如果 1 号位是 1, 那么 1&1 得 1; 如果 1 号位是 0, 那么 0&1 也得 0)。这个过程叫作“使用掩码”, 因为掩码中的 0 隐藏了 flags 中相应的位。

可以这样类比: 把掩码中的 0 看作不透明, 1 看作透明。表达式 flags & MASK 相当于用掩码覆盖在 flags 的位组合上, 只有 MASK 为 1 的位才可见 (见图 15.2)。

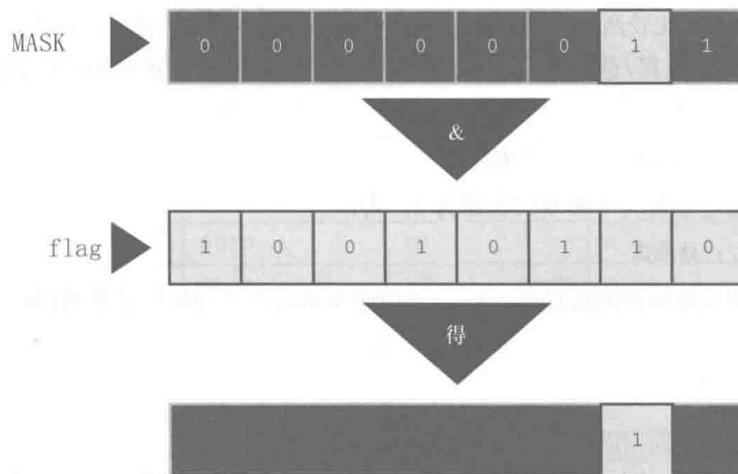


图 15.2 掩码示例

用 &= 运算符可以简化前面的代码, 如下所示:

```
flags &= MASK;
```

下面这条语句是按位与的一种常见用法:

```
ch &= 0xff; /* 或者 ch &= 0377; */
```

前面介绍过 0xff 的二进制形式是 11111111, 八进制形式是 0377。这个掩码保持 ch 中最后 8 位不变, 其他位都设置为 0。无论 ch 原来是 8 位、16 位或是其他更多位, 最终的值都被修改为 1 个 8 位字节。在该例中, 掩码的宽度为 8 位。

### 15.3.3 用法: 打开位 (设置位)

有时, 需要打开一个值中的特定位, 同时保持其他位不变。例如, 一台 IBM PC 通过向端口发送值来

控制硬件。例如，为了打开内置扬声器，必须打开 1 号位，同时保持其他位不变。这种情况可以使用按位或运算符 (`|`)。

以上一节的 `flags` 和 `MASK`（只有 1 号位为 1）为例。下面的语句：

```
flags = flags | MASK;
```

把 `flags` 的 1 号位设置为 1，且其他位不变。因为使用 `|` 运算符，任何位与 0 组合，结果都为本身；任何位与 1 组合，结果都为 1。

例如，假设 `flags` 是 00001111，`MASK` 是 10110110。下面的表达式：

```
flags | MASK
```

即是：

```
(00001111) | (10110110) // 表达式
```

其结果为：

```
(10111111) // 结果值
```

`MASK` 中为 1 的位，`flags` 与其对应的位也为 1。`MASK` 中为 0 的位，`flags` 与其对应的位不变。

用 `|=` 运算符可以简化上面的代码，如下所示：

```
flags |= MASK;
```

同样，这种方法根据 `MASK` 中为 1 的位，把 `flags` 中对应的位设置为 1，其他位不变。

#### 15.3.4 用法：关闭位（清空位）

和打开特定的位类似，有时也需要在不影响其他位的情况下关闭指定的位。假设要关闭变量 `flags` 中的 1 号位。同样，`MASK` 只有 1 号位为 1（即，打开）。可以这样做：

```
flags = flags & ~MASK;
```

由于 `MASK` 除 1 号位为 1 以外，其他位全为 0，所以 `~MASK` 除 1 号位为 0 以外，其他位全为 1。使用 `&`，任何位与 1 组合都得本身，所以这条语句保持 1 号位不变，改变其他各位。另外，使用 `&`，任何位与 0 组合都得 0。所以无论 1 号位的初始值是什么，都将其设置为 0。

例如，假设 `flags` 是 00001111，`MASK` 是 10110110。下面的表达式：

```
flags & ~MASK
```

即是：

```
(00001111) & ~(10110110) // 表达式
```

其结果为：

```
(00001001) // 结果值
```

`MASK` 中为 1 的位在结果中都被设置（清空）为 0。`flags` 中与 `MASK` 为 0 的位相应的位在结果中都未改变。

可以使用下面的简化形式：

```
flags &= ~MASK;
```

#### 15.3.5 用法：切换位

切换位指的是打开已关闭的位，或关闭已打开的位。可以使用按位异或运算符 (`^`) 切换位。也就是说，假设 `b` 是一个位（1 或 0），如果 `b` 为 1，则 `1^b` 为 0；如果 `b` 为 0，则 `1^b` 为 1。另外，无论 `b` 为 1 还是 0，`0^b` 均为 `b`。因此，如果使用 `^` 组合一个值和一个掩码，将切换该值与 `MASK` 为 1 的位相对应的位，该

值与 MASK 为 0 的位相对应的位不变。要切换 flags 中的 1 号位，可以使用下面两种方法：

```
flags = flags ^ MASK;
flags ^= MASK;
```

例如，假设 flags 是 00001111，MASK 是 10110110。表达式：

```
flags ^ MASK
```

即是：

```
(00001111) ^ (10110110) // 表达式
```

其结果为：

```
(10111001) // 结果值
```

flags 中与 MASK 为 1 的位相对应的位都被切换了，MASK 为 0 的位相对应的位不变。

### 15.3.6 用法：检查位的值

前面介绍了如何改变位的值。有时，需要检查某位的值。例如，flags 中 1 号位是否被设置为 1？不能这样直接比较 flags 和 MASK：

```
if (flags == MASK)
    puts("Wow!"); /* 不能正常工作 */
```

这样做即使 flags 的 1 号位为 1，其他位的值会导致比较结果为假。因此，必须覆盖 flags 中的其他位，只用 1 号位和 MASK 比较：

```
if ((flags & MASK) == MASK)
    puts("Wow!");
```

由于按位运算符的优先级比 == 低，所以必须在 flags & MASK 周围加上圆括号。

为了避免信息漏过边界，掩码至少要与其覆盖的值宽度相同。

### 15.3.7 移位运算符

下面介绍 C 的移位运算符。移位运算符向左或向右移动位。同样，我们在示例中仍然使用二进制数，有助于读者理解其工作原理。

#### 1. 左移：<<

左移运算符 (<<) 将其左侧运算对象每一位的值向左移动其右侧运算对象指定的位数。左侧运算对象移出左末端位的值丢失，用 0 填充空出的位置。下面的例子中，每一位都向左移动两个位置：

```
(10001010) << 2 // 表达式
(00101000) // 结果值
```

该操作产生了一个新的位值，但是不改变其运算对象。例如，假设 stonk 为 1，那么 stonk<<2 为 4，但是 stonk 本身不变，仍为 1。可以使用左移赋值运算符 (<<=) 来更改变量的值。该运算符将变量中的位向左移动其右侧运算对象给定值的位数。如下例：

```
int stonk = 1;
int onkoo;
onkoo = stonk << 2; /* 把 4 赋给 onkoo */
stonk <<= 2; /* 把 stonk 的值改为 4 */
```

#### 2. 右移：>>

右移运算符 (>>) 将其左侧运算对象每一位的值向右移动其右侧运算对象指定的位数。左侧运算对象

移出右末端位的值丢。对于无符号类型，用 0 填充空出的位置；对于有符号类型，其结果取决于机器。空出的位置可用 0 填充，或者用符号位（即，最左端的位）的副本填充：

```
(10001010) >> 2      // 表达式，有符号值
(00100010)           // 在某些系统中的结果值
(10001010) >> 2      // 表达式，有符号值
(11100010)           // 在另一些系统上的结果值
```

下面是无符号值的例子：

```
(10001010) >> 2      // 表达式，无符号值
(00100010)           // 所有系统都得到该结果值
```

每个位向右移动两个位置，空出的位用 0 填充。

右移赋值运算符 ( $>>=$ ) 将其左侧的变量向右移动指定数量的位数。如下所示：

```
int sweet = 16;
int ooosw;

ooosw = sweet >> 3;    // ooosw = 2, sweet 的值仍然为 16
sweet >>=3;           // sweet 的值为 2
```

### 3. 用法：移位运算符

移位运算符针对 2 的幂提供快速有效的乘法和除法：

number << n	number 乘以 2 的 n 次幂
number >> n	如果 number 为非负，则用 number 除以 2 的 n 次幂

这些移位运算符类似于在十进制中移动小数点来乘以或除以 10。

移位运算符还可用于从较大单元中提取一些位。例如，假设用一个 `unsigned long` 类型的值表示颜色值，低阶位字节储存红色的强度，下一个字节储存绿色的强度，第 3 个字节储存蓝色的强度。随后你希望把每种颜色的强度分别储存在 3 个不同的 `unsigned char` 类型的变量中。那么，可以使用下面的语句：

```
#define BYTE_MASK 0xff
unsigned long color = 0x002a162f;
unsigned char blue, green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
```

以上代码中，使用右移运算符将 8 位颜色值移动至低阶字节，然后使用掩码技术把低阶字节赋给指定的变量。

## 15.3.8 编程示例

在第 9 章中，我们用递归的方法编写了一个程序，把数字转换为二进制形式（程序清单 9.8）。现在，要用移位运算符来解决相同的问题。程序清单 15.1 中的程序，读取用户从键盘输入的整数，将该整数和一个字符串地址传递给 `itobs()` 函数（`itobs` 表示 *integer to binary string*，即整数转换成二进制字符串）。然后，该函数使用移位运算符计算出正确的 1 和 0 的组合，并将其放入字符串中。

### 程序清单 15.1 binbit.c 程序

---

```
/* binbit.c -- 使用位操作显示二进制 */
#include <stdio.h>
#include <limits.h> // 提供 CHAR_BIT 的定义，CHAR_BIT 表示每字节的位数
char * itobs(int, char *);
```

```

void show_bstr(const char *);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;

    puts("Enter integers and see them in binary.");
    puts("Non-numeric input terminates program.");
    while (scanf("%d", &number) == 1)
    {
        itobs(number, bin_str);
        printf("%d is ", number);
        show_bstr(bin_str);
        putchar('\n');
    }
    puts("Bye!");
}

return 0;
}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';

    return ps;
}

/*4 位一组显示二进制字符串 */
void show_bstr(const char * str)
{
    int i = 0;

    while (str[i]) /* 不是一个空字符 */
    {
        putchar(str[i]);
        if (++i % 4 == 0 && str[i])
            putchar(' ');
    }
}

```

程序清单 15.1 使用 limits.h 中的 CHAR\_BIT 宏，该宏表示 char 中的位数。sizeof 运算符返回 char 的大小，所以表达式 CHAR\_BIT \* sizeof(int) 表示 int 类型的位数。bin\_str 数组的元素个数是 CHAR\_BIT \* sizeof(int) + 1，留出一个位置给末尾的空字符。

itobs() 函数返回的地址与传入的地址相同，可以把该函数作为 printf() 的参数。在该函数中，首次执行 for 循环时，对 01 & n 求值。01 是一个八进制形式的掩码，该掩码除 0 号位是 1 之外，其他所有位都为 0。因此，01 & n 就是 n 最后一位的值。该值为 0 或 1。但是对数组而言，需要的是字符'0'或

字符'1'。该值加上'0'即可完成这种转换（假设按顺序编码的数字，如 ASCII）。其结果存放在数组中倒数第 2 个元素中（最后一个元素用来存放空字符）。

顺带一提，用 `1 & n` 或 `01 & n` 都可以。我们用八进制 1 而不是十进制 1，只是为了更接近计算机的表达方式。

然后，循环执行 `i--` 和 `n >>= 1`。`i--` 移动到数组的前一个元素，`n >>= 1` 使 `n` 中的所有位向右移一个位置。进入下一轮迭代时，循环中处理的是 `n` 中新的最右端的值。然后，把该值储存在倒数第 3 个元素中，以此类推。`itobs()` 函数用这种方式从右往左填充数组。

可以使用 `printf()` 或 `puts()` 函数显示最终的字符串，但是程序清单 15.1 中定义了 `show_bstr()` 函数，以 4 位一组打印字符串，方便阅读。

下面的该程序的运行示例：

```
Enter integers and see them in binary.
Non-numeric input terminates program.
7
7 is 0000 0000 0000 0000 0000 0000 0000 0111
2013
2013 is 0000 0000 0000 0000 0000 0111 1101 1101
-1
-1 is 1111 1111 1111 1111 1111 1111 1111 1111
32123
32123 is 0000 0000 0000 0000 0111 1101 0111 1011
q
Bye!
```

### 15.3.9 另一个例子

我们来看另一个例子。这次要编写的函数用于切换一个值中的后 `n` 位，待处理值和 `n` 都是函数的参数。

`~` 运算符切换一个字节的所有位，而不是选定的少数位。但是，`^` 运算符（按位异或）可用于切换单个位。假设创建了一个掩码，把后 `n` 位设置为 1，其余位设置为 0。然后使用`^`组合掩码和待切换的值便可切换该值的最后 `n` 位，而且其他位不变。方法如下：

```
int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;

    while (bits-- > 0)
    {
        mask |= bitval;
        bitval <= 1;
    }
    return num ^ mask;
}
```

`while` 循环用于创建所需的掩码。最初，`mask` 的所有位都为 0。第 1 轮循环将 `mask` 的 0 号位设置为 1。然后第 2 轮循环将 `mask` 的 1 号位设置为 1，以此类推。循环 `bits` 次，`mask` 的后 `bits` 位就都被设置为 1。最后，`num ^ mask` 运算即得所需的结果。

我们把这个函数放入前面的程序中，测试该函数。如程序清单 15.2 所示。

## 程序清单 15.2 invert4.c 程序

```

/* invert4.c -- 使用位操作显示二进制 */
#include <stdio.h>
#include <limits.h>
char * itobs(int, char *);
void show_bstr(const char *);
int invert_end(int num, int bits);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];

    int number;

    puts("Enter integers and see them in binary.");
    puts("Non-numeric input terminates program.");
    while (scanf("%d", &number) == 1)
    {
        itobs(number, bin_str);
        printf("%d is\n", number);
        show_bstr(bin_str);
        putchar('\n');
        number = invert_end(number, 4);
        printf("Inverting the last 4 bits gives\n");
        show_bstr(itobs(number, bin_str));
        putchar('\n');
    }
    puts("Bye!");
}

return 0;
}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';

    return ps;
}

/* 以 4 位为一组，显示二进制字符串 */
void show_bstr(const char * str)
{
    int i = 0;

    while (str[i]) /* 不是空字符串 */
    {
        putchar(str[i]);
        if (++i % 4 == 0 && str[i])
            putchar(' ');
    }
}

```

```

    }

}

int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;

    while (bits-- > 0)
    {
        mask |= bitval;
        bitval <<= 1;
    }

    return num ^ mask;
}

```

下面是该程序的一个运行示例：

```

Enter integers and see them in binary.
Non-numeric input terminates program.
7
7 is
0000 0000 0000 0000 0000 0000 0000 0111
Inverting the last 4 bits gives
0000 0000 0000 0000 0000 0000 0000 1000
12541
12541 is
0000 0000 0000 0000 0011 0000 1111 1101
Inverting the last 4 bits gives
0000 0000 0000 0000 0011 0000 1111 0010
q
Bye!

```

## 15.4 位字段

操控位的第 2 种方法是位字段 (*bit field*)。位字段是一个 `signed int` 或 `unsigned int` 类型变量中的一组相邻的位 (C99 和 C11 新增了 `_Bool` 类型的位字段)。位字段通过一个结构声明来建立，该结构声明为每个字段提供标签，并确定该字段的宽度。例如，下面的声明建立了一个 4 个 1 位的字段：

```

struct {
    unsigned int autfd : 1;
    unsigned int bldfc : 1;
    unsigned int undln : 1;
    unsigned int itals : 1;
} prnt;

```

根据该声明，`prnt` 包含 4 个 1 位的字段。现在，可以通过普通的结构成员运算符 (.) 单独给这些字段赋值：

```

prnt.itals = 0;
prnt.undln = 1;

```

由于每个字段恰好为 1 位，所以只能为其赋值 1 或 0。变量 `prnt` 被储存在 `int` 大小的内存单元中，但是在本例中只使用了其中的 4 位。

带有位字段的结构提供一种记录设置的方便途径。许多设置（如，字体的粗体或斜体）就是简单的二

选一。例如，开或关、真或假。如果只需要使用 1 位，就不需要使用整个变量。内含位字段的结构允许在一个存储单元中储存多个设置。

有时，某些设置也有多个选择，因此需要多位来表示。这没问题，字段不限制 1 位大小。可以使用如下的代码：

```
struct {
    unsigned int code1 : 2;
    unsigned int code2 : 2;
    unsigned int code3 : 8;
} prcode;
```

以上代码创建了两个 2 位的字段和一个 8 位的字段。可以这样赋值：

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 102;
```

但是，要确保所赋的值不超出字段可容纳的范围。

如果声明的总位数超过了一个 `unsigned int` 类型的大小会怎样？会用到下一个 `unsigned int` 类型的存储位置。一个字段不允许跨越两个 `unsigned int` 之间的边界。编译器会自动移动跨界的字段，保持 `unsigned int` 的边界对齐。一旦发生这种情况，第 1 个 `unsigned int` 中会留下一个未命名的“洞”。

可以用未命名的字段宽度“填充”未命名的“洞”。使用一个宽度为 0 的未命名字段迫使下一个字段与下一个整数对齐：

```
struct {
    unsigned int field1 : 1;
    unsigned int : 2;
    unsigned int field2 : 1;
    unsigned int : 0;
    unsigned int field3 : 1;
} stuff;
```

这里，在 `stuff.field1` 和 `stuff.field2` 之间，有一个 2 位的空隙；`stuff.field3` 将储存在下一个 `unsigned int` 中。

字段储存在一个 `int` 中的顺序取决于机器。在有些机器上，存储的顺序是从左往右，而在另一些机器上，是从右往左。另外，不同的机器中两个字段边界的位置也有区别。由于这些原因，位字段通常都不容易移植。尽管如此，有些情况却要用到这种不可移植的特性。例如，以特定硬件设备所用的形式储存数据。

### 15.4.1 位字段示例

通常，把位字段作为一种更紧凑储存数据的方式。例如，假设要在屏幕上表示一个方框的属性。为简化问题，我们假设方框具有如下属性：

- 方框是透明的或不透明的；
- 方框的填充色选自以下调色板：黑色、红色、绿色、黄色、蓝色、紫色、青色或白色；
- 边框可见或隐藏；
- 边框颜色与填充色使用相同的调色板；
- 边框可以使用实线、点线或虚线样式。

可以使用单独的变量或全长 (*full-sized*) 结构成员来表示每个属性，但是这样做有些浪费位。例如，只需 1 位即可表示方框是透明还是不透明；只需 1 位即可表示边框是显示还是隐藏。8 种颜色可以用 3 位单元的 8 个可能的

值来表示，而3种边框样式也只需2位单元即可表示。总共10位就足够表示方框的5个属性设置。

一种方案是：一个字节储存方框内部（透明和填充色）的属性，一个字节储存方框边框的属性，每个字节中的空隙用未命名字段填充。`struct box_props` 声明如下：

```
struct box_props {
    bool opaque           : 1 ;
    unsigned int fill_color : 3 ;
    unsigned int          : 4 ;
    bool show_border      : 1 ;
    unsigned int border_color : 3 ;
    unsigned int border_style : 2 ;
    unsigned int          : 2 ;
};
```

加上未命名的字段，该结构共占用16位。如果不使用填充，该结构占用10位。但是要记住，C以`unsigned int`作为位字段结构的基本布局单元。因此，即使一个结构唯一的成员是1位字段，该结构的大小也是一个`unsigned int`类型的大小，`unsigned int`在我们的系统中是32位。另外，以上代码假设C99新增的`_Bool`类型可用，在`stdbool.h`中，`bool`是`_Bool`的别名。

对于`opaque`成员，1表示方框不透明，0表示透明。`show_border`成员也用类似的方法。对于颜色，可以用简单的RGB（即red-green-blue的缩写）表示。这些颜色都是三原色的混合。显示器通过混合红、绿、蓝像素来产生不同的颜色。在早期的计算机色彩中，每个像素都可以打开或关闭，所以可以使用用1位来表示三原色中每个二进制颜色的亮度。常用的顺序是，左侧位表示蓝色亮度、中间位表示绿色亮度、右侧位表示红色亮度。表15.3列出了这8种可能的组合。`fill_color`成员和`border_color`成员可以使用这些组合。最后，`border_style`成员可以使用0、1、2来表示实线、点线和虚线样式。

表 15.3 简单的颜色表示

位组合	十进制	颜色
000	0	黑色
001	1	红色
010	2	绿色
011	3	黄色
100	4	蓝色
101	5	紫色
110	6	青色
111	7	白色

程序清单15.3中的程序使用`box_props`结构，该程序用#define创建供结构成员使用的符号常量。注意，只打开一位即可表示三原色之一。其他颜色用三原色的组合来表示。例如，紫色由打开的蓝色位和红色位组成，所以，紫色可表示为BLUE|RED。

程序清单 15.3 fields.c 程序

```
/* fields.c -- 定义并使用字段 */
#include <stdio.h>
#include <stdbool.h> // C99 定义了 bool、true、false

/* 线的样式 */
#define SOLID 0
#define DOTTED 1
#define DASHED 2
```

```

/* 三原色 */
#define BLUE    4
#define GREEN   2
#define RED     1
/* 混合色 */
#define BLACK   0
#define YELLOW  (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN    (GREEN | BLUE)
#define WHITE   (RED | GREEN | BLUE)

const char * colors[8] = { "black", "red", "green", "yellow",
"blue", "magenta", "cyan", "white" };

struct box_props {
    bool opaque : 1;           // 或者 unsigned int (C99 以前)
    unsigned int fill_color : 3;
    unsigned int : 4;
    bool show_border : 1;      // 或者 unsigned int (C99 以前)
    unsigned int border_color : 3;
    unsigned int border_style : 2;
    unsigned int : 2;
};

void show_settings(const struct box_props * pb);

int main(void)
{
    /* 创建并初始化 box_props 结构 */
    struct box_props box = { true, YELLOW, true, GREEN, DASHED };

    printf("Original box settings:\n");
    show_settings(&box);

    box.opaque = false;
    box.fill_color = WHITE;
    box.border_color = MAGENTA;
    box.border_style = SOLID;
    printf("\nModified box settings:\n");
    show_settings(&box);

    return 0;
}

void show_settings(const struct box_props * pb)
{
    printf("Box is %s.\n",
        pb->opaque == true ? "opaque" : "transparent");
    printf("The fill color is %s.\n", colors[pb->fill_color]);
    printf("Border %s.\n",
        pb->show_border == true ? "shown" : "not shown");
    printf("The border color is %s.\n", colors[pb->border_color]);
    printf("The border style is ");
    switch (pb->border_style)
    {

```

```

case SOLID: printf("solid.\n"); break;
case DOTTED: printf("dotted.\n"); break;
case DASHED: printf("dashed.\n"); break;
default:      printf("unknown type.\n");
}

```

下面是该程序的输出:

```

Original box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

```

```

Modified box settings:
Box is transparent.
The fill color is white.
Border shown.
The border color is magenta.
The border style is solid.

```

该程序要注意几个要点。首先，初始化位字段结构与初始化普通结构的语法相同：

```
struct box_props box = {YES, YELLOW, YES, GREEN, DASHED};
```

类似地，也可以给位字段成员赋值：

```
box.fill_color = WHITE;
```

另外，switch语句中也可以使用位字段成员，甚至还可以把位字段成员用作数组的下标：

```
printf("The fill color is %s.\n", colors[pb->fill_color]);
```

注意，根据colors数组的定义，每个索引对应一个表示颜色的字符串，而每种颜色都把索引值作为该颜色的数值。例如，索引1对应字符串"red"，枚举常量red的值是1。

## 15.4.2 位字段和按位运算符

在同类型的编程问题中，位字段和按位运算符是两种可替换的方法，用哪种方法都可以。例如，前面的例子中，使用和unsigned int类型大小相同的结构储存图形框的信息。也可使用unsigned int变量储存相同的信息。如果不想用结构成员表示法来访问不同的部分，也可以使用按位运算符来操作。一般而言，这种方法比较麻烦。接下来，我们来研究这两种方法（程序中使用了这两种方法，仅为了解释它们的区别，我们并不鼓励这样做）。

可以通过一个联合把结构方法和位方法放在一起。假定声明了struct box\_props类型，然后这样声明联合：

```

union Views /* 把数据看作结构或 unsigned short 类型的变量 */
{
    struct box_props st_view;
    unsigned short us_view;
};

```

在某些系统中，unsigned int和box\_props类型的结构都占用16位内存。但是，在其他系统中（例如我们使用的系统），unsigned int和box\_props都是32位。无论哪种情况，通过联合，都可以使用st\_view成员把一块内存看作是一个结构，或者使用us\_view成员把相同的内存块看作是一个unsigned short。结构的哪一个位字段与unsigned short中的哪一位对应？这取决于实现和硬件。下面的程序示例假设从字节的低阶位端到高阶位端载入结构。也就是说，结构中的第1个位字段对应计算机字的0号位（为简化起见，图15.3以16位单元演示了这种情况）。

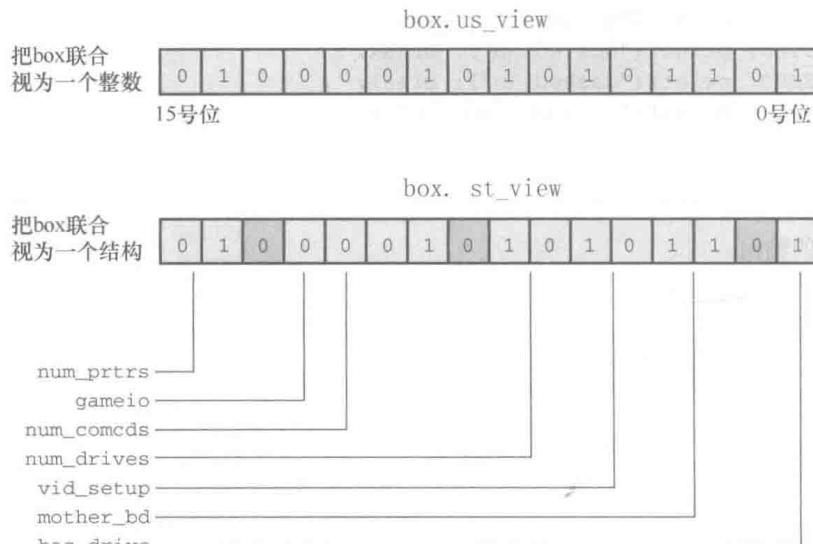


图 15.3 作为整数和结构的联合

程序清单 15.4 使用 Views 联合来比较位字段和按位运算符这两种方法。在该程序中，`box` 是 View 联合，所以 `box.st_view` 是一个使用位字段的 `box_props` 类型的结构，`box.us_view` 把相同的数据看作是一个 `unsigned short` 类型的变量。联合只允许初始化第 1 个成员，所以初始化值必须与结构相匹配。该程序分别通过两个函数显示 `box` 的属性，一个函数接受一个结构，一个函数接受一个 `unsigned short` 类型的值。这两种方法都能访问数据，但是所用的技术不同。该程序还使用了本章前面定义的 `itobs()` 函数，以二进制字符串形式显示数据，以便读者查看每个位的开闭情况。

## 程序清单 15.4 dualview.c 程序

```
/* dualview.c -- 位字段和按位运算符 */
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

/* 位字段符号常量 */
/* 边框线样式 */
#define SOLID      0
#define DOTTED     1
#define DASHED     2
/* 三原色 */
#define BLUE       4
#define GREEN      2
#define RED        1
/* 混合颜色 */
#define BLACK      0
#define YELLOW     (RED | GREEN)
#define MAGENTA   (RED | BLUE)
#define CYAN      (GREEN | BLUE)
#define WHITE     (RED | GREEN | BLUE)

/* 按位方法中用到的符号常量 */
#define OPAQUE     0x1
#define FILL_BLUE  0x8
#define FILL_GREEN 0x4
#define FILL_RED   0x2
#define FILL_MASK  0xE
#define BORDER    0x100
#define BORDER_BLUE 0x800
```

```

#define BORDER_GREEN 0x400
#define BORDER_RED0x 200
#define BORDER_MASK 0xE00
#define B_SOLID 0
#define B_DOTTED 0x1000
#define B_DASHED 0x2000
#define STYLE_MASK0x 3000

const char * colors[8] = { "black", "red", "green", "yellow", "blue", "magenta",
"cyan", "white" };

struct box_props {

    bool opaque : 1;
    unsigned int fill_color : 3;
    unsigned int : 4;
    bool show_border : 1;
    unsigned int border_color : 3;
    unsigned int border_style : 2;
    unsigned int : 2;
};

union Views /* 把数据看作结构或 unsigned short 类型的变量 */
{
    struct box_props st_view;
    unsigned short us_view;
};

void show_settings(const struct box_props * pb);
void show_settings1(unsigned short);
char * itobs(int n, char * ps);

int main(void)
{
    /* 创建Views 联合，并初始化 initialize struct box view */
    union Views box = { { true, YELLOW, true, GREEN, DASHED } };
    char bin_str[8 * sizeof(unsigned int) + 1];

    printf("Original box settings:\n");
    show_settings(&box.st_view);
    printf("\nBox settings using unsigned int view:\n");
    show_settings1(box.us_view);

    printf("bits are %s\n",
        itobs(box.us_view, bin_str));
    box.us_view &= ~FILL_MASK; /* 把表示填充色的位清 0 */
    box.us_view |= (FILL_BLUE | FILL_GREEN); /* 重置填充色 */
    box.us_view ^= OPAQUE; /* 切换是否透明的位 */
    box.us_view |= BORDER_RED; /* 错误的方法 */
    box.us_view &= ~STYLE_MASK; /* 把样式的位清 0 */
    box.us_view |= B_DOTTED; /* 把样式设置为点 */
    printf("\nModified box settings:\n");
    show_settings(&box.st_view);
    printf("\nBox settings using unsigned int view:\n");
    show_settings1(box.us_view);
    printf("bits are %s\n",
        itobs(box.us_view, bin_str));

    return 0;
}

```

```

}

void show_settings(const struct box_props * pb)
{
    printf("Box is %s.\n",
           pb->opaque == true ? "opaque" : "transparent");
    printf("The fill color is %s.\n", colors[pb->fill_color]);
    printf("Border %s.\n",
           pb->show_border == true ? "shown" : "not shown");
    printf("The border color is %s.\n", colors[pb->border_color]);
    printf("The border style is ");
    switch (pb->border_style)
    {
        case SOLID: printf("solid.\n"); break;
        case DOTTED: printf("dotted.\n"); break;
        case DASHED: printf("dashed.\n"); break;
        default:     printf("unknown type.\n");
    }
}

void show_settings1(unsigned short us)
{
    printf("box is %s.\n",
           (us & OPAQUE) == OPAQUE ? "opaque" : "transparent");
    printf("The fill color is %s.\n",
           colors[(us >> 1) & 07]);
    printf("Border %s.\n",
           (us & BORDER) == BORDER ? "shown" : "not shown");
    printf("The border style is ");
    switch (us & STYLE_MASK)
    {
        case B_SOLID : printf("solid.\n"); break;
        case B_DOTTED: printf("dotted.\n"); break;
        case B_DASHED: printf("dashed.\n"); break;
        default       : printf("unknown type.\n");
    }
    printf("The border color is %s.\n",
           colors[(us >> 9) & 07]);
}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';

    return ps;
}

```

下面是该程序的输出：

```

Original box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.
Box settings using unsigned int view:

```

```

box is opaque.
The fill color is yellow.
Border shown.
The border style is dashed.
The border color is green.
bits are 00000000000000000000000000000000100101000000111
Modified box settings:
Box is transparent.
The fill color is cyan.
Border shown.
The border color is yellow.
The border style is dotted.
Box settings using unsigned int view:
box is transparent.
The fill color is cyan.
Border not shown.
The border style is dotted.
The border color is yellow.
bits are 0000000000000000000000000000000010111000001100

```

这里要讨论几个要点。位字段视图和按位视图的区别是，按位视图需要位置信息。例如，程序中使用 BLUE 表示蓝色，该符号常量的数值为 4。但是，由于结构排列数据的方式，实际储存蓝色设置的是 3 号位（位的编号从 0 开始，参见图 15.1），而且储存边框为蓝色的设置是 11 号位。因此，该程序定义了一些新的符号常量：

```
#define FILL_BLUE      0x8
#define BORDER_BLUE    0x800
```

这里，0x8 是 3 号位为 1 时的值，0x800 是 11 号位为 1 时的值。可以使用第 1 个符号常量设置填充色的蓝色位，用第 2 个符号常量设置边框颜色的蓝色位。用十六进制记数法更容易看出要设置二进制的哪一位，由于十六进制的每一位代表二进制的 4 位，那么 0x8 的位组合是 1000，而 0x800 的位组合是 1000000000，0x800 的位组合比 0x8 后面多 8 个 0。但是以等价的十进制来看就没那么明显，0x8 是 8，0x800 是 2048。

如果值是 2 的幂，那么可以使用左移运算符来表示值。例如，可以用下面的#define 分别替换上面的#define：

```
#define FILL_BLUE      1<<3
#define BORDER_BLUE    1<<11
```

这里，<<的右侧是 2 的指数，也就是说，0x8 是  $2^3$ ，0x800 是  $2^{11}$ 。同样，表达式 1<<n 指的是第 n 位为 1 的整数。1<<11 是常量表达式，在编译时求值。

可以使用枚举代替#define 创建符号常量。例如，可以这样做：

```
enum { OPAQUE = 0x1, FILL_BLUE = 0x8, FILL_GREEN = 0x4, FILL_RED = 0x2,
       FILL_MASK = 0xE, BORDER = 0x100, BORDER_BLUE = 0x800,
       BORDER_GREEN = 0x400, BORDER_RED = 0x200, BORDER_MASK = 0xE00,
       B_DOTTED = 0x1000, B_DASHED = 0x2000, STYLE_MASK = 0x3000};
```

如果不想要创建枚举变量，就不用在声明中使用标记。

注意，按位运算符改变设置更加复杂。例如，要设置填充色为青色。只打开蓝色位和绿色位是不够的：

```
box.us_view |= (FILL_BLUE | FILL_GREEN); /* 重置填充色 */
```

问题是该颜色还依赖于红色位的设置。如果已经设置了该位（比如对于黄色），这行代码保留了红色位的设置，而且还设置了蓝色位和绿色位，结果是产生白色。解决这个问题最简单的方法是在设置新值前关闭所有的颜色位。因此，程序中使用了下面两行代码：

```
box.us_view &= ~FILL_MASK;           /* 把表示填充色的位清 0 */
box.us_view |= (FILL_BLUE | FILL_GREEN); /* 重置填充色 */
```

如果不先关闭所有的相关位，程序中演示了这种情况：

```
box.us_view |= BORDER_RED;          /* 错误的方法 */
```

因为 BORDER\_GREEN 位已经设置过了，所以结果颜色是 BORDER\_GREEN | BORDER\_RED，被解释为黄色。

这种情况下，位字段版本更简单：

```
box.st_view.fill_color = CYAN; /*等价的位字段方法*/
```

这种方法不用先清空所有的位。而且，使用位字段成员时，可以为边框和框内填充色使用相同的颜色值。但是用按位运算符的方法则要使用不同的值（这些值反映实际位的位置）。

其次，比较下面两个打印语句：

```
printf("The border color is %s.\n", colors[pb->border_color]);
printf("The border color is %s.\n", colors[(us >> 9) & 07]);
```

第 1 条语句中，表达式 pb->border\_color 的值在 0~7 的范围内，所以该表达式可用作 colors 数组的索引。用按位运算符获得相同的信息更加复杂。一种方法是使用 us>>9 把边框颜色右移至最右端（0 号位~2 号位），然后把该值与掩码 07 组合，关闭除了最右端 3 位以外所有的位。这样结果也在 0~7 的范围内，可作为 colors 数组的索引。

### 警告

位字段和位的位置之间的相互对应因实现而异。例如，在早期的 Macintosh PowerPC 上运行程序

清单 15.4，输出如下：

```
Original box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

Box settings using unsigned int view:
box is transparent.
The fill color is black.
Border not shown.
The border style is solid.
The border color is black.
bits are 10110000101010000000000000000000000000000

Modified box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

Box settings using unsigned int view:
box is opaque.
The fill color is cyan.
Border shown.
The border style is dotted.
The border color is red.
bits are 10110000101010000001001000001101
```

该输出的二进制位与程序示例 15.4 不同，Macintosh PowerPC 把结构载入内存的方式不同。特别是，它把第 1 位字段载入最高阶位，而不是最低阶位。所以结构表示法储存在前 16 位（与 PC 中的顺序不同），而 unsigned int 表示法则储存在后 16 位。因此，对于 Macintosh，程序清单 15.4 中关于位的位置的假设是错误的，使用按位运算符改变透明设置和填充色设置时，也弄错了位。

## 15.5 对齐特性 (C11)

C11 的对齐特性比用位填充字节更自然，它们还代表了 C 在处理硬件相关问题上的能力。在这种上下文中，对齐指的是如何安排对象在内存中的位置。例如，为了效率最大化，系统可能要把一个 `double` 类型的值储存在 4 字节内存地址上，但却允许把 `char` 储存在任意地址。大部分程序员都对对齐不以为然。但是，有些情况又受益于对齐控制。例如，把数据从一个硬件位置转移到另一个位置，或者调用指令同时操作多个数据项。

`_Alignof` 运算符给出一个类型的对齐要求，在关键字 `_Alignof` 后面的圆括号中写上类型名即可：

```
size_t d_align = _Alignof(float);
```

假设 `d_align` 的值是 4，意思是 `float` 类型对象的对齐要求是 4。也就是说，4 是储存该类型值相邻地址的字节数。一般而言，对齐值都应该是 2 的非负整数次幂。较大的对齐值被称为 `stricter` 或 `stronger`，较小的对齐值被称为 `weaker`。

可以使用 `_Alignas` 说明符指定一个变量或类型的对齐值。但是，不应该要求该值小于基本对齐值。例如，如果 `float` 类型的对齐要求是 4，不要请求其对齐值是 1 或 2。该说明符用作声明的一部分，说明符后面的圆括号内包含对齐值或类型：

```
_Alignas(double) char c1;
_ALIGNAS(8) char c2;
unsigned char _Alignas(long double) c_arr[sizeof(long double)];
```

### 注意

撰写本书时，Clang (3.2 版本) 要求 `_Alignas(type)` 说明符在类型说明符后面，如上面第 3 行代码所示。但是，无论 `_Alignas(type)` 说明符在类型说明符的前面还是后面，GCC 4.7.3 都能识别，后来 Clang 3.3 版本也支持了这两种顺序。

程序清单 15.5 中的程序演示了 `_Alignas` 和 `_Alignof` 的用法。

程序清单 15.5 align.c 程序

```
// align.c -- 使用 _Alignof 和 _Alignas (C11)

#include <stdio.h>
int main(void)
{
    double dx;
    char ca;
    char cx;
    double dz;
    char cb;
    char _Alignas(double) cz;

    printf("char alignment: %zd\n", _Alignof(char));
    printf("double alignment: %zd\n", _Alignof(double));
    printf("&dx: %p\n", &dx);
    printf("&ca: %p\n", &ca);
    printf("&cx: %p\n", &cx);
    printf("&dz: %p\n", &dz);
```

```

printf("&cb: %p\n", &cb);
printf("&cz: %p\n", &cz);

return 0;
}

```

该程序的输出如下：

```

char alignment:    1
double alignment: 8
&dx: 0x7fff5fbff660
&ca: 0x7fff5fbff65f
&cx: 0x7fff5fbff65e
&dz: 0x7fff5fbff650
&cb: 0x7fff5fbff64f
&cz: 0x7fff5fbff648

```

在我们的系统中，`double` 的对齐值是 8，这意味着地址的类型对齐可以被 8 整除。以 0 或 8 结尾的十六进制地址可被 8 整除。这就是地址常用两个 `double` 类型的变量和 `char` 类型的变量 `cz`（该变量是 `double` 对齐值）。因为 `char` 的对齐值是 1，对于普通的 `char` 类型变量，编译器可以使用任何地址。

在程序中包含 `stdalign.h` 头文件后，就可以把 `alignas` 和 `alignof` 分别作为 `_Alignas` 和 `_Alignof` 的别名。这样做可以与 C++ 关键字匹配。

C11 在 `stdlib.h` 库还添加了一个新的内存分配函数，用于对齐动态分配的内存。该函数的原型如下：

```
void *aligned_alloc(size_t alignment, size_t size);
```

第 1 个参数代表指定的对齐，第 2 个参数是所需的字节数，其值应是第 1 个参数的倍数。与其他内存分配函数一样，要使用 `free()` 函数释放之前分配的内存。

## 15.6 关键概念

C 区别于许多高级语言的特性之一是访问整数中单独位的能力。该特性通常是与硬件设备和操作系统交互的关键。

C 有两种访问位的方法。一种方法是通过按位运算符，另一种方法是在结构中创建位字段。

C11 新增了检查内存对齐要求的功能，而且可以指定比基本对齐值更大的对齐值。

通常（但不总是），使用这些特性的程序仅限于特定的硬件平台或操作系统，而且设计为不可移植的。

## 15.7 本章小结

计算硬件与二进制记数系统密不可分，因为二进制数的 1 和 0 可用于表示计算机内存和寄存器中位的开闭状态。虽然 C 不允许以二进制形式书写数字，但是它识别与二进制相关的八进制和十六进制记数法。正如每个二进制数字表示 1 位一样，每个八进制位代表 3 位，每个十六进制位代表 4 位。这种关系使得二进制转为八进制或十六进制较为简单。

C 提供多种按位运算符，之所以称为按位是因为它们单独操作一个值中的每个位。`~` 运算符将其运算对象的每一位取反，将 1 转为 0，0 转为 1。按位与运算符（`&`）通过两个运算对象形成一个值。如果两运算对象中相同号位都为 1，那么该值中对应的位为 1；否则，该位为 0。按位或运算符（`|`）同样通过两个运算对象形成一个值。如果两运算对象中相同号位有一个为 1 或都为 1，那么该值中对应的位为 1；否则，该位为 0。按位异或运算符（`^`）也有类似的操作，只有两运算对象中相同号位有一个为 1 时，结果值中对

应的位才为 1。

C 还有左移（`<<`）和右移（`>>`）运算符。这两个运算符使位组合中的所有位都向左或向右移动指定数量的位，以形成一个新值。对于左移运算符，空出的位置设为 0。对于右移运算符，如果是无符号类型的值，空出的位设为 0；如果是有符号类型的值，右移运算符的行为取决于实现。

可以在结构中使用位字段操控一个值中的单独位或多组位。具体细节因实现而异。

可以使用 `_Alignas` 强制执行数据存储区上的对齐要求。

这些位工具帮助 C 程序处理硬件问题，因此它们通常用于依赖实现的场合中。

## 15.8 复习题

复习题的参考答案在附录 A 中。

1. 把下面的十进制转换为二进制：

- a. 3
- b. 13
- c. 59
- d. 119

2. 将下面的二进制值转换为十进制、八进制和十六进制的形式：

- a. 00010101
- b. 01010101
- c. 01001100
- d. 10011101

3. 对下面的表达式求值，假设每个值都为 8 位：

- a. `~3`
- b. `3 & 6`
- c. `3 | 6`
- d. `1 | 6`
- e. `3 ^ 6`
- f. `7 >> 1`
- g. `7 << 2`

4. 对下面的表达式求值，假设每个值都为 8 位：

- a. `~0`
- b. `!0`
- c. `2 & 4`
- d. `2 && 4`
- e. `2 | 4`
- f. `2 || 4`

g. `5 << 3`

5. 因为 ASCII 码只使用最后 7 位，所以有时需要用掩码关闭其他位，其相应的二进制掩码是什么？分别用十进制、八进制和十六进制来表示这个掩码。

6. 程序清单 15.2 中，可以把下面的代码：

```
while (bits-- > 0)
{
    mask |= bitval;
    bitval <= 1;
}
```

替换成：

```
while (bits-- > 0)
{
    mask += bitval;
    bitval *= 2;
}
```

程序照常工作。这是否意味着`*=2`等同于`<<=1`? `+=`是否等同于`|=`?

7. a. Tinkerbell 计算机有一个硬件字节可读入程序。该字节包含以下信息：

位	含义
0~1	1.4MB 软盘驱动器的数量
2	未使用
3~4	CD-ROM 驱动器数量
5	未使用
6~7	硬盘驱动器数量

Tinkerbell 和 IBM PC 一样，从右往左填充结构位字段。创建一个适合存放这些信息的位字段模板。

b. Klinkerbll 与 Tinkerbell 类似，但是它从左往右填充结构位字段。请为 Klinkerbll 创建一个相应的位字段模板。

## 15.9 编程练习

1. 编写一个函数，把二进制字符串转换为一个数值。例如，有下面的语句：

```
char * pbin = "01001001";
```

那么把 `pbin` 作为参数传递给该函数后，它应该返回一个 `int` 类型的值 25。

2. 编写一个程序，通过命令行参数读取两个二进制字符串，对这两个二进制数使用`~`运算符、`&`运算符、`|`运算符和`^`运算符，并以二进制字符串形式打印结果（如果无法使用命令行环境，可以通过交互式让程序读取字符串）。

3. 编写一个函数，接受一个 `int` 类型的参数，并返回该参数中打开位的数量。在一个程序中测试该函数。

4. 编写一个程序，接受两个 `int` 类型的参数：一个是值；一个是位的位置。如果指定位的位置为 1，该函数返回 1；否则返回 0。在一个程序中测试该函数。

5. 编写一个函数，把一个 `unsigned int` 类型值中的所有位向左旋转指定数量的位。例如，

`rotate_1(x, 4)` 把 `x` 中所有位向左移动 4 个位置，而且从最左端移出的位会重新出现在右端。也就是说，把高阶位移出的位放入低阶位。在一个程序中测试该函数。

6. 设计一个位字段结构以储存下面的信息。

字体 ID: 0 ~ 255 之间的一个数；

字体大小: 0 ~ 127 之间的一个数；

对齐: 0 ~ 2 之间的一个数，表示左对齐、居中、右对齐；

加粗: 开 (1) 或闭 (0)；

斜体: 开 (1) 或闭 (0)；

在一个程序中使用该结构来打印字体参数，并使用循环菜单来让用户改变参数。例如，该程序的一个运行示例如下：

```
ID SIZE ALIGNMENT  B   I   U
 1   12    left     off off off

f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
s

Enter font size (0-127): 36

ID SIZE ALIGNMENT  B   I   U
 1   36    left     off off off

f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
a

Select alignment:
l)left   c)center   r)right
r

ID SIZE ALIGNMENT  B   I   U
 1   36   right     off off off
f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
i

ID SIZE ALIGNMENT  B   I   U
 1   36   right     off  on off

f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
q

Bye!
```

该程序要使用按位与运算符 (&) 和合适的掩码来把字体 ID 和字体大小信息转换到指定的范围内。

7. 编写一个与编程练习 6 功能相同的程序，使用 `unsigned long` 类型的变量储存字体信息，并且使用按位运算符而不是位成员来管理这些信息。



# 第 16 章

## C 预处理器和 C 库

本章介绍以下内容：

- 预处理指令：#define、#include、#ifdef、#else、#endif、#ifndef、#if、#elif、#line、#error、#pragma
- 关键字：\_Generic、\_Noreturn、\_Static\_assert
- 函数/宏：sqrt()、atan()、atan2()、exit()、atexit()、assert()、memcpy()、memmove()、va\_start()、va\_arg()、va\_copy()、va\_end()
- C 预处理器的其他功能
- 通用选择表达式
- 内联函数
- C 库概述和一些特殊用途的方便函数

C 语言建立在适当的关键字、表达式、语句以及使用它们的规则上。然而，C 标准不仅描述 C 语言，还描述如何执行 C 预处理器、C 标准库有哪些函数，以及详述这些函数的工作原理。本章将介绍 C 预处理器和 C 库，我们先从 C 预处理器开始。

C 预处理器在程序执行之前查看程序（故称之为预处理器）。根据程序中的预处理器指令，预处理器把符号缩写替换为其表示的内容。预处理器可以包含程序所需的其他文件，可以选择让编译器查看哪些代码。预处理器并不知道 C。基本上它的工作是把一些文本转换成另外一些文本。这样描述预处理器无法体现它的真正效用和价值，我们将在本章举例说明。前面的程序示例中也有很多#define 和#include 的例子。下面，我们先总结一下已学过的预处理指令，再介绍一些新的知识点。

### 16.1 翻译程序的第一步

在预处理之前，编译器必须对该程序进行一些翻译处理。首先，编译器把源代码中出现的字符映射到源字符集。该过程处理多字节字符和三字符序列——字符扩展让 C 更加国际化（详见附录 B“参考资料 VII，扩展字符支持”）。

第二，编译器定位每个反斜杠后面跟着换行符的实例，并删除它们。也就是说，把下面两个物理行 (physical line)：

```
printf("That's wond\
erful!\n");
```

转换成一个逻辑行 (logical line)：

```
printf("That's wonderful\n!");
```

注意，在这种场合中，“换行符”的意思是通过按下 Enter 键在源代码文件中换行所生成的字符，而不是指符号表征\n。

由于预处理表达式的长度必须是一个逻辑行，所以这一步为预处理器做好了准备工作。一个逻辑行可以是多个物理行。

第三，编译器把文本划分成预处理记号序列、空白序列和注释序列（记号是由空格、制表符或换行符分隔的项，详见 16.2.1）。这里要注意的是，编译器将用一个空格字符替换每一条注释。因此，下面的代码：

```
int/* 这看起来并不像一个空格*/fox;
```

将变成：

```
int fox;
```

而且，实现可以用一个空格替换所有的空白字符序列（不包括换行符）。最后，程序已经准备好进入预处理阶段，预处理器查找一行中以#号开始的预处理指令。

## 16.2 明示常量：#define

#define 预处理器指令和其他预处理器指令一样，以#号作为一行的开始。ANSI 和后来的标准都允许#号前面有空格或制表符，而且还允许在#号和指令的其余部分之间有空格。但是旧版本的 C 要求指令从一行最左边开始，而且#号和指令其余部分之间不能有空格。指令可以出现在源文件的任何地方，其定义从指令出现的地方到该文件末尾有效。我们大量使用#define 指令来定义明示常量（*manifest constant*）（也叫做符号常量），但是该指令还有许多其他用途。程序清单 16.1 演示了#define 指令的一些用法和属性。

预处理器指令从#开始运行，到后面的第 1 个换行符为止。也就是说，指令的长度仅限于一行。然而，前面提到过，在预处理开始前，编译器会把多行物理行处理为一行逻辑行。

程序清单 16.1 preproc.c 程序

---

```
/* preproc.c -- 简单的预处理示例 */
#include <stdio.h>
#define TWO 2      /* 可以使用注释 */
#define OW "Consistency is the last refuge of the unimaginative. - Oscar Wilde" /* 反斜杠把该定义延续到下一行 */

#define FOUR TWO*TWO
#define PX printf("X is %d.\n", x)
#define FMT "X is %d.\n"

int main(void)
{
    int x = TWO;

    PX;
    x = FOUR;
    printf(FMT, x);
    printf("%s\n", OW);
    printf("TWO: OW\n");

    return 0;
}
```

---

每行#define（逻辑行）都由3部分组成。第1部分是#define指令本身。第2部分是选定的缩写，也称为宏。有些宏代表值（如本例），这些宏被称为类对象宏（*object-like macro*）。C语言还有类函数宏（*function-like macro*），稍后讨论。宏的名称中不允许有空格，而且必须遵循C变量的命名规则：只能使用字符、数字和下划线（\_）字符，而且首字符不能是数字。第3部分（指令行的其余部分）称为替换列表或替换体（见图16.1）。一旦预处理器在程序中找到宏的示实例后，就会用替换体代替该宏（也有例外，稍后解释）。从宏变成最终替换文本的过程称为宏展开（*macro expansion*）。注意，可以在#define行使用标准C注释。如前所述，每条注释都会被一个空格代替。

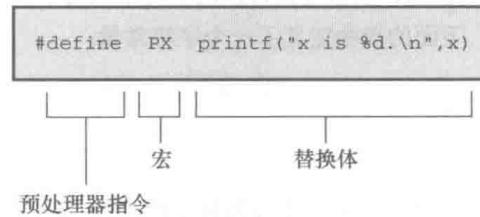


图16.1 类对象宏定义的组成

运行该程序示例后，输出如下：

```

X is 2.
X is 4.
Consistency is the last refuge of the unimaginative. - Oscar Wilde
TWO: OW
    
```

下面分析具体的过程。下面的语句：

```
int x = TWO;
```

变成了：

```
int x = 2;
```

2代替了TWO。而语句：

```
PX;
```

变成了：

```
printf("X is %d.\n", x);
```

这里同样进行了替换。这是一个新用法，到目前为止我们只是用宏来表示明示常量。从该例中可以看出，宏可以表示任何字符串，甚至可以表示整个C表达式。但是要注意，虽然PX是一个字符串常量，它只打印一个名为x的变量。

下一行也是一个新用法。读者可能认为FOUR被替换成4，但是实际的过程是：

```
x = FOUR;
```

变成了：

```
x = TWO*TWO;
```

即是：

```
x = 2*2;
```

宏展开到此处为止。由于编译器在编译期对所有的常量表达式（只包含常量的表达式）求值，所以预处理器不会进行实际的乘法运算，这一过程在编译时进行。预处理器不做计算，不对表达式求值，它只进行替换。

注意，宏定义还可以包含其他宏（一些编译器不支持这种嵌套功能）。

程序中的下一行：

```
printf (FMT, x);
```

变成了：

```
printf("X is %d.\n", x);
```

相应的字符串替换了 FMT。如果要多次使用某个冗长的字符串，这种方法比较方便。另外，也可以用下面的方法：

```
const char * fmt = "X is %d.\n";
```

然后可以把 fmt 作为 printf() 的格式字符串。

下一行中，用相应的字符串替换 OW。双引号使替换的字符串成为字符串常量。编译器把该字符串储存在以空字符结尾的数组中。因此，下面的指令定义了一个字符常量：

```
#define HAL 'Z'
```

而下面的指令则定义了一个字符串 (z\0)：

```
#define HAP "Z"
```

在程序示例 16.1 中，我们在一行的结尾加一个反斜杠字符使该行扩展至下一行：

```
#define OW "Consistency is the last refuge of the unimaginative. - Oscar Wilde"
```

注意，第 2 行要与第 1 行左对齐。如果这样做：

```
#define OW "Consistency is the last refuge of the unimaginative. - Oscar Wilde"
```

那么输出的内容是：

```
Consistency is the last refuge of the unimaginative. - Oscar Wilde
```

第 2 行开始到 tive 之间的空格也算是字符串的一部分。

一般而言，预处理器发现程序中的宏后，会用宏等价的替换文本进行替换。如果替换的字符串中还包含宏，则继续替换这些宏。唯一例外的是双引号中的宏。因此，下面的语句：

```
printf("TWO: OW");
```

打印的是 TWO: OW，而不是打印：

```
2: Consistency is the last refuge of the unimaginative. - Oscar Wilde
```

要打印这行，应该这样写：

```
printf("%d: %s\n", TWO, OW);
```

这行代码中，宏不在双引号内。

那么，何时使用字符常量？对于绝大部分数字常量，应该使用字符常量。如果在算式中用字符常量代替数字，常量名能更清楚地表达该数字的含义。如果是表示数组大小的数字，用符号常量后更容易改变数组的大小和循环次数。如果数字是系统代码（如，EOF），用符号常量表示的代码更容易移植（只需改变 EOF 的定义）。助记、易更改、可移植，这些都是符号常量很有价值的特性。

C 语言现在也支持 const 关键字，提供了更灵活的方法。用 const 可以创建在程序运行过程中不能改变的变量，可具有文件作用域或块作用域。另一方面，宏常量可用于指定标准数组的大小和 const 变量的初始值。

```
#define LIMIT 20
const int LIM = 50;
static int data1[LIMIT];           // 有效
static int data2[LIM];             // 无效
const int LIM2 = 2 * LIMIT;        // 有效
const int LIM3 = 2 * LIM;           // 无效
```

这里解释一下上面代码中的“无效”注释。在 C 中，非自动数组的大小应该是整型常量表达式，这意味着表示数组大小的必须是整型常量的组合（如 5）、枚举常量和 sizeof 表达式，不包括 const 声明的值（这也是 C++ 和 C 的区别之一，在 C++ 中可以把 const 值作为常量表达式的一部分）。但是，有的实现可能接受其他形式的常量表达式。例如，GCC 4.7.3 不允许 data2 的声明，但是 Clang 4.6 允许。

## 16.2.1 记号

从技术角度来看，可以把宏的替换体看作是记号（*token*）型字符串，而不是字符型字符串。C 预处理器记号是宏定义的替换体中单独的“词”。用空白把这些词分开。例如：

```
#define FOUR 2*2
```

该宏定义有一个记号：2\*2 序列。但是，下面的宏定义中：

```
#define SIX 2 * 3
```

有 3 个记号：2、\*、3。

替换体中有多个空格时，字符型字符串和记号型字符串的处理方式不同。考虑下面的定义：

```
#define EIGHT 4 * 8
```

如果预处理器把该替换体解释为字符型字符串，将用 4 \* 8 替换 EIGHT。即，额外的空格是替换体的一部分。如果预处理器把该替换体解释为记号型字符串，则用 3 个的记号 4 \* 8（分别由单个空格分隔）来替换 EIGHT。换而言之，解释为字符型字符串，把空格视为替换体的一部分；解释为记号型字符串，把空格视为替换体中各记号的分隔符。在实际应用中，一些 C 编译器把宏替换体视为字符串而不是记号。在比这个例子更复杂的情况下，两者的区别才有实际意义。

顺带一提，C 编译器处理记号的方式比预处理器复杂。由于编译器理解 C 语言的规则，所以不要求代码中用空格来分隔记号。例如，C 编译器可以把 2\*2 直接视为 3 个记号，因为它可以识别 2 是常量，\* 是运算符。

## 16.2.2 重定义常量

假设先把 LIMIT 定义为 20，稍后在该文件中又把它定义为 25。这个过程称为重定义常量。不同的实现采用不同的重定义方案。除非新定义与旧定义相同，否则有些实现会将其视为错误。另外一些实现允许重定义，但会给出警告。ANSI 标准采用第 1 种方案，只有新定义和旧定义完全相同才允许重定义。

具有相同的定义意味着替换体中的记号必须相同，且顺序也相同。因此，下面两个定义相同：

```
#define SIX 2 * 3
```

```
#define SIX 2 * 3
```

这两条定义都有 3 个相同的记号，额外的空格不算替换体的一部分。而下面的定义则与上面两条宏定义不同：

```
#define SIX 2*3
```

这条宏定义中只有一个记号，因此与前两条定义不同。如果需要重定义宏，使用 #undef 指令（稍后讨论）。

如果确实需要重定义常量，使用 const 关键字和作用域规则更容易些。

## 16.3 在#define 中使用参数

在#define 中使用参数可以创建外形和作用与函数类似的类函数宏。带有参数的宏看上去很像函数，因为这样的宏也使用圆括号。类函数宏定义的圆括号中可以有一个或多个参数，随后这些参数出现在替换

体中，如图 16.2 所示。

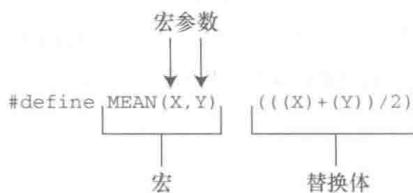


图 16.2 函数宏定义的组成

下面是一个类函数宏的示例：

```
#define SQUARE(X) X*X
```

在程序中可以这样用：

```
z = SQUARE(2);
```

这看上去像函数调用，但是它的行为和函数调用完全不同。程序清单 16.2 演示了类函数宏和另一个宏的用法。该示例中有一些陷阱，请读者仔细阅读序。

程序清单 16.2 mac\_arg.c 程序

```
/* mac_arg.c -- 带参数的宏 */
#include <stdio.h>
#define SQUARE(X) X*X
#define PR(X) printf("The result is %d.\n", X)
int main(void)
{
    int x = 5;
    int z;

    printf("x = %d\n", x);
    z = SQUARE(x);
    printf("Evaluating SQUARE(x): ");
    PR(z);
    z = SQUARE(2);
    printf("Evaluating SQUARE(2): ");
    PR(z);
    printf("Evaluating SQUARE(x+2): ");
    PR(SQUARE(x + 2));
    printf("Evaluating 100/SQUARE(2): ");
    PR(100 / SQUARE(2));
    printf("x is %d.\n", x);
    printf("Evaluating SQUARE(++x): ");
    PR(SQUARE(++x));
    printf("After incrementing, x is %x.\n", x);

    return 0;
}
```

SQUARE 宏的定义如下：

```
#define SQUARE(X) X*X
```

这里，SQUARE 是宏标识符，SQUARE(X) 中的 X 是宏参数，X\*X 是替换列表。程序清单 16.2 中出现 SQUARE(X) 的地方都会被 X\*X 替换。这与前面的示例不同，使用该宏时，既可以用 x，也可以用其他符号。宏定义中的 X 由宏调用中的符号代替。因此，SQUARE(2) 替换为 2\*2，x 实际上起到参数的作用。

然而，稍后你将看到，宏参数与函数参数不完全相同。下面是程序的输出。注意有些内容可能与我们的预期不符。实际上，你的编译器输出甚至与下面的结果完全不同。

```
x = 5
Evaluating SQUARE(x): The result is 25.
Evaluating SQUARE(2): The result is 4.
Evaluating SQUARE(x+2): The result is 17.
Evaluating 100/SQUARE(2): The result is 100.
x is 5.
Evaluating SQUARE(++x): The result is 42.
After incrementing, x is 7.
```

前两行与预期相符，但是接下来的结果有点奇怪。程序中设置 `x` 的值为 5，你可能认为 `SQUARE(x+2)` 应该是  $7 \times 7$ ，即 49。但是，输出的结果是 17，这不是一个平方值！导致这样结果的原因是，我们前面提到过，预处理器不做计算、不求值，只替换字符序列。预处理器把出现 `x` 的地方都替换成 `x+2`。因此，`x*x` 变成了 `x+2*x+2`。如果 `x` 为 5，那么该表达式的值为：

$$5+2*5+2 = 5 + 10 + 2 = 17$$

该例演示了函数调用和宏调用的重要区别。函数调用在程序运行时把参数的值传递给函数。宏调用在编译之前把参数记号传递给程序。这两个不同的过程发生在不同时期。是否可以修改宏定义让 `SQUARE(x+2)` 得 36？当然可以，要多加几个圆括号：

```
#define SQUARE(x) (x)*(x)
```

现在 `SQUARE(x+2)` 变成了 `(x+2)*(x+2)`，在替换字符串中使用圆括号就得到符合预期的乘法运算。

但是，这并未解决所有的问题。下面的输出行：

```
100/SQUARE(2)
```

将变成：

```
100/2*2
```

根据优先级规则，从左往右对表达式求值： $(100/2)*2$ ，即  $50*2$ ，得 100。把 `SQUARE(x)` 定义为下面的形式可以解决这种混乱：

```
#define SQUARE(x) (x*x)
```

这样修改定义后得  $100/(2*2)$ ，即  $100/4$ ，得 25。

要处理前面的两种情况，要这样定义：

```
#define SQUARE(x) ((x)*(x))
```

因此，必要时要使用足够多的圆括号来确保运算和结合的正确顺序。

尽管如此，这样做还是无法避免程序中最后一种情况的问题。`SQUARE(++x)` 变成了 `++x*x++x`，递增了两次 `x`，一次在乘法运算之前，一次在乘法运算之后：

$$++x*x++x = 6*7 = 42$$

由于标准并未对这类运算规定顺序，所以有些编译器得 7\*6。而有些编译器可能在乘法运算之前已经递增了 `x`，所以 7\*7 得 49。在 C 标准中，对该表达式求值的这种情况称为未定义行为。无论哪种情况，`x` 的开始值都是 5，虽然从代码上看只递增了一次，但是 `x` 的最终值是 7。

解决这个问题最简单的方法是，避免用 `++x` 作为宏参数。一般而言，不要在宏中使用递增或递减运算符。但是，`++x` 可作为函数参数，因为编译器会对 `++x` 求值得 5 后，再把 5 传递给函数。

### 16.3.1 用宏参数创建字符串：#运算符

下面是一个类函数宏：

```
#define PSQR(X) printf("The square of X is %d.\n", ((X)*(X)));
```

假设这样使用宏：

```
PSQR(8);
```

输出为：

```
The square of X is 64.
```

注意双引号字符串中的 X 被视为普通文本，而不是一个可被替换的记号。

C 允许在字符串中包含宏参数。在类函数宏的替换体中，#号作为一个预处理运算符，可以把记号转换成字符串。例如，如果 x 是一个宏形参，那么 #x 就是转换为字符串 "x" 的形参名。这个过程称为字符串化 (*stringizing*)。程序清单 16.3 演示了该过程的用法。

#### 程序清单 16.3 subst.c 程序

```
/* subst.c -- 在字符串中替换 */
#include <stdio.h>
#define PSQR(x) printf("The square of " #x " is %d.\n", ((x)*(x)))

int main(void)
{
    int y = 5;

    PSQR(y);
    PSQR(2 + 4);

    return 0;
}
```

该程序的输出如下：

```
The square of y is 25.
The square of 2 + 4 is 36.
```

调用第 1 个宏时，用 "y" 替换 #x。调用第 2 个宏时，用 "2 + 4" 替换 #x。ANSI C 字符串的串联特性将这些字符串与 printf() 语句的其他字符串组合，生成最终的字符串。例如，第 1 次调用变成：

```
printf("The square of " "y" " is %d.\n", ((y)*(y)));
```

然后，字符串串联功能将这 3 个相邻的字符串组合成一个字符串：

```
"The square of y is %d.\n"
```

### 16.3.2 预处理器黏合剂：##运算符

与 # 运算符类似，## 运算符可用于类函数宏的替换部分。而且，## 还可用于对象宏的替换部分。## 运算符把两个记号组合成一个记号。例如，可以这样做：

```
#define XNAME(n) x ## n
```

然后，宏 XNAME(4) 将展开为 x4。程序清单 16.4 演示了## 作为记号粘合剂的用法。

#### 程序清单 16.4 glue.c 程序

```
// glue.c -- 使用##运算符
#include <stdio.h>
#define XNAME(n) x ## n
#define PRINT_XN(n) printf("x" #n " = %d\n", x ## n);

int main(void)
```

```

{
    int XNAME(1) = 14;      // 变成 int x1 = 14;
    int XNAME(2) = 20;      // 变成 int x2 = 20;
    int x3 = 30;
    PRINT_XN(1);           // 变成 printf("x1 = %d\n", x1);
    PRINT_XN(2);           // 变成 printf("x2 = %d\n", x2);
    PRINT_XN(3);           // 变成 printf("x3 = %d\n", x3);
    return 0;
}

```

该程序的输出如下：

```

x1 = 14
x2 = 20
x3 = 30

```

注意，PRINT\_XN()宏用#运算符组合字符串，##运算符把记号组合为一个新的标识符。

### 16.3.3 变参宏：...和\_\_VA\_ARGS\_\_

一些函数（如printf()）接受数量可变的参数。stdvar.h头文件（本章后面介绍）提供了工具，让用户自定义带可变参数的函数。C99/C11也对宏提供了这样的工具。虽然标准中未使用“可变”（variadic）这个词，但是它已成为描述这种工具的通用词（虽然，C标准的索引添加了字符串化(stringizing)词条，但是，标准并未把固定参数的函数或宏称为固定函数和不变宏）。

通过把宏参数列表中最后的参数写成省略号（即，3个点...）来实现这一功能。这样，预定义宏

\_VA\_ARGS\_可用在替换部分中，表明省略号代表什么。例如，下面的定义：

```
#define PR(...) printf(_VA_ARGS_)
```

假设稍后调用该宏：

```
PR("Howdy");
PR("weight = %d, shipping = $%.2f\n", wt, sp);
```

对于第1次调用，\_VA\_ARGS\_展开为1个参数："Howdy"。

对于第2次调用，\_VA\_ARGS\_展开为3个参数："weight = %d, shipping = \$%.2f\n"、wt、sp。

因此，展开后的代码是：

```
printf("Howdy");
printf("weight = %d, shipping = $%.2f\n", wt, sp);
```

程序清单16.5演示了一个示例，该程序使用了字符串的串联功能和#运算符。

程序清单16.5 variadic.c程序

---

```

// variadic.c -- 变参宏
#include <stdio.h>
#include <math.h>
#define PR(X, ...) printf("Message " #X ":" " _VA_ARGS_ )

int main(void)
{
    double x = 48;
    double y;

    y = sqrt(x);

```

```

PR(1, "x = %g\n", x);
PR(2, "x = %.2f, y = %.4f\n", x, y);

return 0;
}

```

第 1 个宏调用, x 的值是 1, 所以 #x 变成 "1"。展开后成为:

```
print("Message " "1" ":" "x = %g\n", x);
```

然后, 串联 4 个字符, 把调用简化为:

```
print("Message 1: x = %g\n", x);
```

下面是该程序的输出:

```
Message 1: x = 48
Message 2: x = 48.00, y = 6.9282
```

记住, 省略号只能代替最后的宏参数:

```
#define WRONG(X, ..., Y) #X #__VA_ARGS__ #y //不能这样做
```

## 16.4 宏和函数的选择

有些编程任务既可以用带参数的宏完成, 也可以用函数完成。应该使用宏还是函数? 这没有硬性规定, 但是可以参考下面的情况。

使用宏比使用普通函数复杂一些, 稍有不慎会产生奇怪的副作用。一些编译器规定宏只能定义成一行。不过, 即使编译器没有这个限制, 也应该这样做。

宏和函数的选择实际上是时间和空间的权衡。宏生成内联代码, 即在程序中生成语句。如果调用 20 次宏, 即在程序中插入 20 行代码。如果调用函数 20 次, 程序中只有一份函数语句的副本, 所以节省了空间。然而另一方面, 程序的控制必须跳转至函数内, 随后再返回主调程序, 这显然比内联代码花费更多的时间。

宏的一个优点是, 不用担心变量类型 (这是因为宏处理的是字符串, 而不是实际的值)。因此, 只要能用 int 或 float 类型都可以使用 SQUARE(x) 宏。

C99 提供了第 3 种可替换的方法——内联函数。本章后面将介绍。

对于简单的函数, 程序员通常使用宏, 如下所示:

```
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
#define ABS(X) ((X) < 0 ? -(X) : (X))
#define ISSIGN(X) ((X) == '+' || (X) == '-' ? 1 : 0)
```

(如果 x 是一个代数符号字符, 最后一个宏的值为 1, 即为真。)

要注意以下几点。

- 记住宏名中不允许有空格, 但是在替换字符串中可以有空格。ANSI C 允许在参数列表中使用空格。
- 用圆括号把宏的参数和整个替换体括起来。这样能确保被括起来的部分在下面这样的表达式中正确地展开:
 

```
forks = 2 * MAX(guests + 3, last);
```
- 用大写字母表示宏函数的名称。该惯例不如用大写字母表示宏常量应用广泛。但是, 大写字母可以提醒程序员注意, 宏可能产生的副作用。
- 如果打算使用宏来加快程序的运行速度, 那么首先要确定使用宏和使用函数是否会导致较大差异。在程序中只使用一次的宏无法明显减少程序的运行时间。在嵌套循环中使用宏更有助于提高效率。

许多系统提供程序分析器以帮助程序员压缩程序中最耗时的部分。

假设你开发了一些方便的宏函数，是否每写一个新程序都要重写这些宏？如果使用#include 指令，就不用这样做了。

## 16.5 文件包含：#include

当预处理器发现#include 指令时，会查看后面的文件名并把文件的内容包含到当前文件中，即替换源文件中的#include 指令。这相当于把被包含文件的全部内容输入到源文件#include 指令所在的位置。#include 指令有两种形式：

```
#include <stdio.h>           ←文件名在尖括号中
#include "mystuff.h"         ←文件名在双引号中
```

在 UNIX 系统中，尖括号告诉预处理器在标准系统目录中查找该文件。双引号告诉预处理器首先在当前目录中（或文件名中指定的其他目录）查找该文件，如果未找到再查找标准系统目录：

```
#include <stdio.h>           ←查找系统目录
#include "hot.h"              ←查找当前工作目录
#include "/usr/biff/p.h"       ←查找/usr/biff 目录
```

集成开发环境（IDE）也有标准路径或系统头文件的路径。许多集成开发环境提供菜单选项，指定用尖括号时的查找路径。在 UNIX 中，使用双引号意味着先查找本地目录，但是具体查找哪个目录取决于编译器的设定。有些编译器会搜索源代码文件所在的目录，有些编译器则搜索当前的工作目录，还有些搜索项目文件所在的目录。

ANSI C 不为文件提供统一的目录模型，因为不同的计算机所用的系统不同。一般而言，命名文件的方法因系统而异，但是尖括号和双引号的规则与系统无关。

为什么要包含文件？因为编译器需要这些文件中的信息。例如，stdio.h 文件中通常包含 EOF、NULL、getchar() 和 putchar() 的定义。getchar() 和 putchar() 被定义为宏函数。此外，该文件中还包含 C 的其他 I/O 函数。

C 语言习惯用.h 后缀表示头文件，这些文件包含需要放在程序顶部的信息。头文件经常包含一些预处理器指令。有些头文件（如 stdio.h）由系统提供，当然你也可以创建自己的头文件。

包含一个大型头文件不一定显著增加程序的大小。在大部分情况下，头文件的内容是编译器生成最终代码时所需的信息，而不是添加到最终代码中的材料。

### 16.5.1 头文件示例

假设你开发了一个存放人名的结构，还编写了一些使用该结构的函数。可以把不同的声明放在头文件中。程序清单 16.6 演示了一个这样的例子。

程序清单 16.6 names\_st.h 头文件

---

```
// names_st.h -- names_st 结构的头文件
// 常量
#include <string.h>
#define SLEN 32

// 结构声明
struct names_st
{
```

```

char first[SLEN];
char last[SLEN];
};

// 类型定义
typedef struct names_st names;

// 函数原型
void get_names(names *);
void show_names(const names *);
char * s_gets(char * st, int n);

```

该头文件包含了一些头文件中常见的内容：#define 指令、结构声明、typedef 和函数原型。注意，这些内容是编译器在创建可执行代码时所需的信息，而不是可执行代码。为简单起见，这个特殊的头文件过于简单。通常，应该用#ifndef 和#define 防止多重包含头文件。我们稍后介绍这些内容。

可执行代码通常在源代码文件中，而不是在头文件中。例如，程序清单 16.7 中有头文件中函数原型的定义。该程序包含了 names\_st.h 头文件，所以编译器知道 names 类型。

程序清单 16.7 name\_st.c 源文件

```

// names_st.c -- 定义 names_st.h 中的函数
#include <stdio.h>
#include "names_st.h"      // 包含头文件

// 函数定义
void get_names(names * pn)
{
    printf("Please enter your first name: ");
    s_gets(pn->first, SLEN);

    printf("Please enter your last name: ");
    s_gets(pn->last, SLEN);
}

void show_names(const names * pn)
{
    printf("%s %s", pn->first, pn->last);
}

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; // 处理输入行中的剩余字符
    }
    return ret_val;
}

```

get\_names() 函数通过 s\_gets() 函数调用了 fgets() 函数，避免了目标数组溢出。程序清单 16.8 使用了程序清单 16.6 的头文件和程序清单 16.7 的源文件。

程序清单 16.8 useheader.c 程序

```

// useheader.c -- 使用 names_st 结构
#include <stdio.h>
#include "names_st.h"
// 记住要链接 names_st.c

int main(void)
{
    names candidate;

    get_names(&candidate);
    printf("Let's welcome ");
    show_names(&candidate);
    printf(" to this program!\n");
    return 0;
}

```

下面是该程序的输出：

```

Please enter your first name: Ian
Please enter your last name: Smersh
Let's welcome Ian Smersh to this program!

```

该程序要注意下面几点。

- 两个源代码文件都使用 names\_st 类型结构，所以它们都必须包含 names\_st.h 头文件。
- 必须编译和链接 names\_st.c 和 useheader.c 源代码文件。
- 声明和指令放在 names\_st.h 头文件中，函数定义放在 names\_st.c 源代码文件中。

## 16.5.2 使用头文件

浏览任何一个标准头文件都可以了解头文件的基本信息。头文件中最常用的形式如下。

- 明示常量——例如，stdio.h 中定义的 EOF、NULL 和 BUFSIZE（标准 I/O 缓冲区大小）。
- 宏函数——例如，getc(stdin) 通常用 getchar() 定义，而 getc() 经常用于定义较复杂的宏，头文件 ctype.h 通常包含 ctype 系列函数的宏定义。
- 函数声明——例如，string.h 头文件（一些旧的系统中是 strings.h）包含字符串函数系列的函数声明。在 ANSI C 和后面的标准中，函数声明都是函数原型形式。
- 结构模版定义——标准 I/O 函数使用 FILE 结构，该结构中包含了文件和与文件缓冲区相关的信息。FILE 结构在头文件 stdio.h 中。
- 类型定义——标准 I/O 函数使用指向 FILE 的指针作为参数。通常，stdio.h 用#define 或 typedef 把 FILE 定义为指向结构的指针。类似地，size\_t 和 time\_t 类型也定义在头文件中。

许多程序员都在程序中使用自己开发的标准头文件。如果开发一系列相关的函数或结构，那么这种方法特别有价值。

另外，还可以使用头文件声明外部变量供其他文件共享。例如，如果已经开发了共享某个变量的一系列函数，该变量报告某种状况（如，错误情况），这种方法就很有效。这种情况下，可以在包含这些函数声明的源代码文件定义一个文件作用域的外部链接变量：

```
int status = 0;           // 该变量具有文件作用域，在源代码文件
```

然后，可以在与源代码文件相关联的头文件中进行引用式声明：

```
extern int status;        // 在头文件中
```

这行代码会出现在包含了该头文件的文件中，这样使用该系列函数的文件都能使用这个变量。虽然源代码文件中包含该头文件后也包含了该声明，但是只要声明的类型一致，在一个文件中同时使用定义式声明和引用式声明没问题。

需要包含头文件的另一种情况是，使用具有文件作用域、内部链接和 `const` 限定符的变量或数组。`const` 防止值被意外修改，`static` 意味着每个包含该头文件的文件都获得一份副本。因此，不需要在一个文件中进行定义式声明，在其他文件中进行引用式声明。

`#include` 和 `#define` 指令是最常用的两个 C 预处理器特性。接下来，我们介绍一些其他指令。

## 16.6 其他指令

程序员可能要为不同的工作环境准备 C 程序和 C 库包。不同的环境可能使用不同的代码类型。预处理器提供一些指令，程序员通过修改 `#define` 的值即可生成可移植的代码。`#undef` 指令取消之前的 `#define` 定义。`#if`、`#ifdef`、`#ifndef`、`#else`、`#elif` 和 `#endif` 指令用于指定什么情况下编写哪些代码。`#line` 指令用于重置行和文件信息，`#error` 指令用于给出错误消息，`#pragma` 指令用于向编译器发出指令。

### 16.6.1 #undef 指令

`#undef` 指令用于“取消”已定义的 `#define` 指令。也就是说，假设有如下定义：

```
#define LIMIT 400
```

然后，下面的指令：

```
#undef LIMIT
```

将移除上面的定义。现在就可以把 `LIMIT` 重新定义为一个新值。即使原来没有定义 `LIMIT`，取消 `LIMIT` 的定义仍然有效。如果想使用一个名称，又不确定之前是否已经用过，为安全起见，可以用 `#undef` 指令取消该名字的定义。

### 16.6.2 从 C 预处理器角度看已定义

处理器在识别标识符时，遵循与 C 相同的规则：标识符可以由大写字母、小写字母、数字和下划线字符组成，且首字符不能是数字。当预处理器在预处理器指令中发现一个标识符时，它会把该标识符当作已定义的或未定义的。这里的已定义表示由预处理器定义。如果标识符是同一个文件中由前面的 `#define` 指令创建的宏名，而且没有用 `#undef` 指令关闭，那么该标识符是已定义的。如果标识符不是宏，假设是一个文件作用域的 C 变量，那么该标识符对预处理器而言就是未定义的。

已定义宏可以是对象宏，包括空宏或类函数宏：

```
#define LIMIT 1000           // LIMIT 是已定义的
```

```
#define GOOD           // GOOD 是已定义的
#define A(X) ((-(X))*(X)) // A 是已定义的
int q;                // q 不是宏，因此是未定义的
#undef GOOD            // GOOD 取消定义，是未定义的
```

注意，`#define` 宏的作用域从它在文件中的声明处开始，直到用`#undef` 指令取消宏为止，或延伸至文件尾（以二者中先满足的条件作为宏作用域的结束）。另外还要注意，如果宏通过头文件引入，那么`#define` 在文件中的位置取决于`#include` 指令的位置。

稍后将介绍几个预定义宏，如`_DATE_` 和 `_FILE_`。这些宏一定是已定义的，而且不能取消定义。

### 16.6.3 条件编译

可以使用其他指令创建条件编译（*conditional compilation*）。也就是说，可以使用这些指令告诉编译器根据编译时的条件执行或忽略信息（或代码）块。

#### 1. `#ifdef`、`#else` 和 `#endif` 指令

我们用一个简短的示例来演示条件编译的情况。考虑下面的代码：

```
#ifdef MAVIS
    #include "horse.h" // 如果已经用#define 定义了 MAVIS，则执行下面的指令
    #define STABLES 5
#else
    #include "cow.h"      //如果没有用#define 定义 MAVIS，则执行下面的指令
    #define STABLES 15
#endif
```

这里使用的较新的编译器和 ANSI 标准支持的缩进格式。如果使用旧的编译器，必须左对齐所有的指令或至少左对齐`#`号，如下所示：

```
#ifdef MAVIS
    #include "horse.h"      // 如果已经用#define 定义了 MAVIS，则执行下面的指令
    #define STABLES 5
#else
    #include "cow.h"        //如果没有用#define 定义 MAVIS，则执行下面的指令
    #define STABLES 15
#endif
```

`#ifdef` 指令说明，如果预处理器已定义了后面的标识符（MAVIS），则执行`#else` 或 `#endif` 指令之前的所有指令并编译所有 C 代码（先出现哪个指令就执行到哪里）。如果预处理器未定义 MAVIS，且有`#else` 指令，则执行`#else` 和 `#endif` 指令之间的所有代码。

`#ifdef` `#else` 很像 C 的 `if` `else`。两者的主要区别是，预处理器不识别用于标记块的花括号`( )`，因此它使用`#else`（如果需要）和`#endif`（必须存在）来标记指令块。这些指令结构可以嵌套。也可以用这些指令标记 C 语句块，如程序清单 16.9 所示。

程序清单 16.9 `ifdef.c` 程序

---

```
/* ifdef.c -- 使用条件编译 */
#include <stdio.h>
#define JUST_CHECKING
#define LIMIT 4

int main(void)
{
```

```

int i;
int total = 0;

for (i = 1; i <= LIMIT; i++)
{
    total += 2 * i*i + 1;
#ifdef JUST_CHECKING
    printf("i=%d, running total = %d\n", i, total);
#endif
}
printf("Grand total = %d\n", total);

return 0;
}

```

编译并运行该程序后，输出如下：

```

i=1, running total = 3
i=2, running total = 12
i=3, running total = 31
i=4, running total = 64
Grand total = 64

```

如果省略 JUST\_CHECKING 定义（把它放在 C 注释中，或者使用`#undef`指令取消它的定义）并重新编译该程序，只会输出最后一行。可以用这种方法在调试程序。定义 JUST\_CHECKING 并合理使用`#ifdef`，编译器将执行用于调试的程序代码，打印中间值。调试结束后，可移除 JUST\_CHECKING 定义并重新编译。如果以后还需要使用这些信息，重新插入定义即可。这样做省去了再次输入额外打印语句的麻烦。`#ifdef`还可用于根据不同的 C 实现选择合适的代码块。

## 2. #ifndef 指令

`#ifndef` 指令与`#ifdef` 指令的用法类似，也可以和`#else`、`#endif`一起使用，但是它们的逻辑相反。`#ifndef` 指令判断后面的标识符是否是未定义的，常用于定义之前未定义的常量。如下所示：

```

/* arrays.h */
#ifndef SIZE
#define SIZE 100
#endif
(旧的实现可能不允许使用缩进的#define)

```

通常，包含多个头文件时，其中的文件可能包含了相同宏定义。`#ifndef` 指令可以防止相同的宏被重复定义。在首次定义一个宏的头文件中用`#ifndef` 指令激活定义，随后在其他头文件中的定义都被忽略。

`#ifndef` 指令还有另一种用法。假设有上面的 arrays.h 头文件，然后把下面一行代码放入一个头文件中：

```

#include "arrays.h"
SIZE 被定义为 100。但是，如果把下面的代码放入该头文件：
#define SIZE 10
#include "arrays.h"

```

SIZE 则被设置为 10。这里，当执行到`#include "arrays.h"`这行，处理 array.h 中的代码时，由于 SIZE 是已定义的，所以跳过了`#define SIZE 100` 这行代码。鉴于此，可以利用这种方法，用一个较小的数组测试程序。测试完毕后，移除`#define SIZE 10` 并重新编译。这样，就不用修改头文件数组本身了。

#ifndef 指令通常用于防止多次包含一个文件。也就是说，应该像下面这样设置头文件：

```
/* things.h */
#ifndef THINGS_H_
#define THINGS_H_
/* 省略了头文件中的其他内容*/
#endif
```

假设该文件被包含了多次。当预处理器首次发现该文件被包含时，THINGS\_H\_是未定义的，所以定义了THINGS\_H\_，并接着处理该文件的其他部分。当预处理器第2次发现该文件被包含时，THINGS\_H\_是已定义的，所以预处理器跳过了该文件的其他部分。

为什么要多次包含一个文件？最常见的原因是，许多被包含的文件中都包含着其他文件，所以显式包含的文件中可能包含着已经包含的其他文件。这有什么问题？在被包含的文件中有某些项（如，一些结构类型的声明）只能在一个文件中出现一次。C标准头文件使用#ifndef技巧避免重复包含。但是，这存在一个问题：如何确保待测试的标识符没有在别处定义。通常，实现的供应商使用这些方法解决这个问题：用文件名作为标识符、使用大写字母、用下划线字符代替文件名中的点字符、用下划线字符做前缀或后缀（可能使用两条下划线）。例如，查看stdio.h头文件，可以发现许多类似的代码：

```
#ifndef _STDIO_H
#define _STDIO_H
// 省略了文件的内容
#endif
```

你也可以这样做。但是，由于标准保留使用下划线作为前缀，所以在自己的代码中不要这样写，避免与标准头文件中的宏发生冲突。程序清单16.10修改了程序清单16.6中的头文件，使用#ifndef避免文件被重复包含。

程序清单16.10 names.c程序

---

```
// names.h --修订后的 names_st 头文件，避免重复包含

#ifndef NAMES_H_
#define NAMES_H_

// 明示常量
#define SLEN 32

// 结构声明
struct names_st
{
    char first[SLEN];
    char last[SLEN];
};

// 类型定义
typedef struct names_st names;

// 函数原型
void get_names(names *);
void show_names(const names *);
char * s_gets(char * st, int n);

#endif
```

---

用程序清单 16.11 的程序测试该头文件没问题，但是如果把清单 16.10 中的`#ifndef`保护删除后，程序就无法通过编译。

程序清单 16.11 doubincl.c 程序

---

```
// doubincl.c -- 包含头文件两次
#include <stdio.h>
#include "names.h"
#include "names.h" // 不小心第 2 次包含头文件

int main()
{
    names winner = { "Less", "Ismoor" };
    printf("The winner is %s %s.\n", winner.first,
           winner.last);
    return 0;
}
```

---

### 3. #if 和# elif 指令

`#if` 指令很像 C 语言中的 `if`。`#if` 后面跟整型常量表达式，如果表达式为非零，则表达式为真。可以在指令中使用 C 的关系运算符和逻辑运算符：

```
#if SYS == 1
#include "ibm.h"
#endif
```

可以按照 `if else` 的形式使用`#elif`（早期的实现不支持`#elif`）。例如，可以这样写：

```
#if SYS == 1
    #include "ibmpc.h"
#elif SYS == 2
    #include "vax.h"
#elif SYS == 3
    #include "mac.h"
#else
    #include "general.h"
#endif
```

较新的编译器提供另一种方法测试名称是否已定义，即用`#if defined (VAX)`代替`#ifdef VAX`。

这里，`defined` 是一个预处理运算符，如果它的参数是用`#defined` 定义过，则返回 1；否则返回 0。这种新方法的优点是，它可以和`#elif`一起使用。下面用这种形式重写前面的示例：

```
#if defined (IBMPC)
    #include "ibmpc.h"
#elif defined (VAX)
    #include "vax.h"
#elif defined (MAC)
    #include "mac.h"
#else
    #include "general.h"
#endif
```

如果在 VAX 机上运行这几行代码，那么应该在文件前面用下面的代码定义 VAX：

```
#define VAX
```

条件编译还有一个用途是让程序更容易移植。改变文件开头部分的几个关键的定义，即可根据不同的系统设置不同的值和包含不同的文件。

## 16.6.4 预定义宏

C 标准规定了一些预定义宏，如表 16.1 所列。

表 16.1 预 定 义 宏

宏	含义
__DATE__	预处理的日期 ("Mmm dd yyyy" 形式的字符串字面量，如 Nov 23 2013)
__FILE__	表示当前源代码文件名的字符串字面量
__LINE__	表示当前源代码文件中行号的整型常量
__STDC__	设置为 1 时，表明实现遵循 C 标准
__STDC_HOSTED__	本机环境设置为 1；否则设置为 0
__STDC_VERSION__	支持 C99 标准，设置为 199901L；支持 C11 标准，设置为 201112L
__TIME__	翻译代码的时间，格式为 "hh:mm:ss"

C99 标准提供一个名为 \_\_func\_\_ 的预定义标识符，它展开为一个代表函数名的字符串（该函数包含该标识符）。那么，\_\_func\_\_ 必须具有函数作用域，而从本质上讲宏具有文件作用域。因此，\_\_func\_\_ 是 C 语言的预定义标识符，而不是预定义宏。

程序清单 16.12 中使用了一些预定义宏和预定义标识符。注意，其中一些是 C99 新增的，所以不支持 C99 的编译器可能无法识别它们。如果使用 GCC，必须设置 -std=c99 或 -std=c11。

程序清单 16.12 predef.c 程序

```
// predef.c -- 预定义宏和预定义标识符
#include <stdio.h>
void why_me();

int main()
{
    printf("The file is %s.\n", __FILE__);
    printf("The date is %s.\n", __DATE__);
    printf("The time is %s.\n", __TIME__);
    printf("The version is %ld.\n", __STDC_VERSION__);
    printf("This is line %d.\n", __LINE__);
    printf("This function is %s\n", __func__);
    why_me();
}

return 0;
}

void why_me()
{
    printf("This function is %s\n", __func__);
    printf("This is line %d.\n", __LINE__);
}
```

下面是该程序的输出：

```
The file is predef.c.
The date is Sep 23 2013.
The time is 22:01:09.
```

```
The version is 201112.
This is line 11.
This function is main
This function is why_me
This is line 21.
```

## 16.6.5 #line 和#error

#line 指令重置 \_\_LINE\_\_ 和 \_\_FILE\_\_ 宏报告的行号和文件名。可以这样使用#line:

```
#line 1000          // 把当前行号重置为 1000
#line 10 "cool.c"    // 把行号重置为 10, 把文件名重置为 cool.c
```

#error 指令让预处理器发出一条错误消息，该消息包含指令中的文本。如果可能的话，编译过程应该中断。可以这样使用#error 指令：

```
#if __STDC_VERSION__ != 201112L
#error Not C11

#endif
```

编译以上代码生成后，输出如下：

```
$ gcc newish.c
newish.c:14:2: error: #error Not C11
$ gcc -std=c11 newish.c
$
```

如果编译器只支持旧标准，则会编译失败，如果支持 C11 标准，就能成功编译。

## 16.6.6 #pragma

在现在的编译器中，可以通过命令行参数或 IDE 菜单修改编译器的一些设置。#pragma 把编译器指令放入源代码中。例如，在开发 C99 时，标准被称为 C9X，可以使用下面的编译指示（*pragma*）让编译器支持 C9X：

```
#pragma c9x on
```

一般而言，编译器都有自己的编译指示集。例如，编译指示可能用于控制分配给自动变量的内存量，或者设置错误检查的严格程度，或者启用非标准语言特性等。C99 标准提供了 3 个标准编译指示，但是超出了本书讨论的范围。

C99 还提供 \_Pragma 预处理器运算符，该运算符把字符串转换成普通的编译指示。例如：

```
_Pragma("nonstandardtreatmenttypeB on")
```

等价于下面的指令：

```
#pragma nonstandardtreatmenttypeB on
```

由于该运算符不使用#符号，所以可以把它作为宏展开的一部分：

```
#define PRAGMA(X) _Pragma(#X)
#define LIMRG(X) PRAGMA(STDC CX_LIMITED_RANGE X)
```

然后，可以使用类似下面的代码：

```
LIMRG ( ON )
```

顺带一提，下面的定义看上去没问题，但实际上无法正常运行：

```
#define LIMRG(X) _Pragma(STDC CX_LIMITED_RANGE #X)
```

问题在于这行代码依赖字符串的串联功能，而预处理过程完成之后才会串联字符串。

\_Pragma 运算符完成“解字符串”(*destringizing*) 的工作，即把字符串中的转义序列转换成它所代表

的字符。因此，

```
_Pragma("use_bool \"true \"false")
```

变成了：

```
#pragma use_bool "true "false
```

## 16.6.7 泛型选择 (C11)

在程序设计中，泛型编程 (*generic programming*) 指那些没有特定类型，但是一旦指定一种类型，就可以转换成指定类型的代码。例如，C++在模板中可以创建泛型算法，然后编译器根据指定的类型自动使用实例化代码。C没有这种功能。然而，C11新增了一种表达式，叫作泛型选择表达式 (*generic selection expression*)，可根据表达式的类型（即表达式的类型是 `int`、`double` 还是其他类型）选择一个值。泛型选择表达式不是预处理器指令，但是在一些泛型编程中它常用作`#define` 宏定义的一部分。

下面是一个泛型选择表达式的示例：

```
_Generic(x, int: 0, float: 1, double: 2, default: 3)
```

`_Generic` 是 C11 的关键字。`_Generic` 后面的圆括号中包含多个用逗号分隔的项。第 1 个项是一个表达式，后面的每个项都由一个类型、一个冒号和一个值组成，如 `float: 1`。第 1 个项的类型匹配哪个标签，整个表达式的值是该标签后面的值。例如，假设上面表达式中 `x` 是 `int` 类型的变量，`x` 的类型匹配 `int:` 标签，那么整个表达式的值就是 0。如果没有与类型匹配的标签，表达式的值就是 `default:` 标签后面的值。泛型选择语句与 `switch` 语句类似，只是前者用表达式的类型匹配标签，而后者用表达式的值匹配标签。

下面是一个把泛型选择语句和宏定义组合的例子：

```
#define MYTYPE(X) _Generic((X), \
    int: "int", \
    float : "float", \
    double: "double", \
    default: "other"\
)
```

宏必须定义为一条逻辑行，但是可以用\把一条逻辑行分隔成多条物理行。在这种情况下，对泛型选择表达式求值得字符串。例如，对 `MYTYPE(5)` 求值得"int"，因为值 5 的类型与 `int:` 标签匹配。程序清单 16.13 演示了这种用法。

程序清单 16.13 mytype.c 程序

---

```
// mytype.c

#include <stdio.h>

#define MYTYPE(X) _Generic((X), \
    int: "int", \
    float : "float", \
    double: "double", \
    default: "other"\

)

int main(void)
{
    int d = 5;

    printf("%s\n", MYTYPE(d));      // d 是 int 类型
    printf("%s\n", MYTYPE(2.0*d)); // 2.0 * d 是 double 类型
```

```

printf("%s\n", MYTYPE(3L));      // 3L 是 long 类型
printf("%s\n", MYTYPE(&d));     // &d 的类型是 int *
return 0;
}

```

下面是该程序的输出：

```

int
double
other
other

```

MYTYPE() 最后两个示例所用的类型与标签不匹配，所以打印默认的字符串。可以使用更多类型标签来扩展宏的能力，但是该程序主要是为了演示 `_Generic` 的基本工作原理。

对一个泛型选择表达式求值时，程序不会先对第一个项求值，它只确定类型。只有匹配标签的类型后才会对表达式求值。

可以像使用独立类型（“泛型”）函数那样使用 `_Generic` 定义宏。本章后面介绍 `math` 库时会给出一个示例。

## 16.7 内联函数 (C99)

通常，函数调用都有一定的开销，因为函数的调用过程包括建立调用、传递参数、跳转到函数代码并返回。使用宏使代码内联，可以避免这样的开销。C99 还提供另一种方法：内联函数 (*inline function*)。读者可能顾名思义地认为内联函数会用内联代码替换函数调用。其实 C99 和 C11 标准中叙述的是：“把函数变成内联函数建议尽可能快地调用该函数，其具体效果由实现定义”。因此，把函数变成内联函数，编译器可能会用内联代码替换函数调用，并（或）执行一些其他的优化，但是也可能不起作用。

创建内联函数的定义有多种方法。标准规定具有内部链接的函数可以成为内联函数，还规定了内联函数的定义与调用该函数的代码必须在同一个文件中。因此，最简单的方法是使用函数说明符 `inline` 和存储类别说明符 `static`。通常，内联函数应定义在首次使用它的文件中，所以内联函数也相当于函数原型。如下所示：

```

#include <stdio.h>
inline static void eatline()    // 内联函数定义/原型
{
    while (getchar() != '\n')
        continue;
}

int main()
{
    ...
    eatline();           // 函数调用
    ...
}

```

编译器查看内联函数的定义（也是原型），可能会用函数体中的代码替换 `eatline()` 函数调用。也就是说，效果相当于在函数调用的位置输入函数体中的代码：

```

#include <stdio.h>
inline static void eatline() // 内联函数定义/原型
{
    while (getchar() != '\n')

```

```

    continue;
}

int main()
{
    ...
    while (getchar() != '\n') // 替换函数调用
        continue;
    ...
}

```

由于并未给内联函数预留单独的代码块，所以无法获得内联函数的地址（实际上可以获得地址，不过这样做之后，编译器会生成一个非内联函数）。另外，内联函数无法在调试器中显示。

内联函数应该比较短小。把较长的函数变成内联并未节约多少时间，因为执行函数体的时间比调用函数的时间长得多。

编译器优化内联函数必须知道该函数定义的内容。这意味着内联函数定义与函数调用必须在同一个文件中。鉴于此，一般情况下内联函数都具有内部链接。因此，如果程序有多个文件都要使用某个内联函数，那么这些文件中都必须包含该内联函数的定义。最简单的做法是，把内联函数定义放入头文件，并在使用该内联函数的文件中包含该头文件即可。

```

// eatline.h
#ifndef EATLINE_H_
#define EATLINE_H_
inline static void eatline()
{
    while (getchar() != '\n')
        continue;
}
#endif

```

一般都不在头文件中放置可执行代码，内联函数是个特例。因为内联函数具有内部链接，所以在多个文件中定义同一个内联函数不会产生什么问题。

与 C++ 不同的是，C 还允许混合使用内联函数定义和外部函数定义（具有外部链接的函数定义）。例如，一个程序中使用下面 3 个文件：

```

//file1.c
...
inline static double square(double);
double square(double x) { return x * x; }
int main()
{
    double q = square(1.3);
    ...

//file2.c
...
double square(double x) { return (int) (x*x); }
void spam(double v)
{
    double kv = square(v);
    ...

//file3.c
...

```

```

inline double square(double x) { return (int) (x * x + 0.5); }
void masp(double w)
{
    double kw = square(w);
    ...
}

```

如上述代码所示，3 个文件中都定义了 `square()` 函数。`file1.c` 文件中是 `inline static` 定义；`file2.c` 文件中是普通的函数定义（因此具有外部链接）；`file3.c` 文件中是 `inline` 定义，省略了 `static`。

3 个文件中的函数都调用了 `square()` 函数，这会发什么情况？`file1.c` 文件中的 `main()` 使用 `square()` 的局部 `static` 定义。由于该定义也是 `inline` 定义，所以编译器有可能优化代码，也许会内联该函数。`file2.c` 文件中，`spam()` 函数使用该文件中 `square()` 函数的定义，该定义具有外部链接，其他文件也可见。`file3.c` 文件中，编译器既可以使用该文件中 `square()` 函数的内联定义，也可以使用 `file2.c` 文件中的外部链接定义。如果像 `file3.c` 那样，省略 `file1.c` 文件 `inline` 定义中的 `static`，那么该 `inline` 定义被视为可替换的外部定义。

注意 GCC 在 C99 之前就使用一些不同的规则实现了内联函数，所以 GCC 可以根据当前编译器的标记来解释 `inline`。

## 16.8 `_Noreturn` 函数 (C11)

C99 新增 `inline` 关键字时，它是唯一的函数说明符（关键字 `extern` 和 `static` 是存储类别说明符，可应用于数据对象和函数）。C11 新增了第 2 个函数说明符 `_Noreturn`，表明调用完成后函数不返回主调函数。`exit()` 函数是 `_Noreturn` 函数的一个示例，一旦调用 `exit()`，它不会再返回主调函数。注意，这与 `void` 返回类型不同。`void` 类型的函数在执行完毕后返回主调函数，只是它不提供返回值。

`_Noreturn` 的目的是告诉用户和编译器，这个特殊的函数不会把控制返回主调程序。告诉用户以免滥用该函数，通知编译器可优化一些代码。

## 16.9 C 库

最初，并没有官方的 C 库。后来，基于 UNIX 的 C 实现成为了标准。ANSI C 委员会主要以这个标准为基础，开发了一个官方的标准库。在意识到 C 语言的应用范围不断扩大后，该委员会重新定义了这个库，使之可以应用于其他系统。

我们讨论过一些标准库中的 I/O 函数、字符函数和字符串函数。本章将介绍更多函数。不过，首先要学习如何使用库。

### 16.9.1 访问 C 库

如何访问 C 库取决于实现，因此你要了解当前系统的一般情况。首先，可以在多个不同的位置找到库函数。例如，`getchar()` 函数通常作为宏定义在 `stdio.h` 头文件中，而 `strlen()` 通常在库文件中。其次，不同的系统搜索这些函数的方法不同。下面介绍 3 种可能的方法。

#### 1. 自动访问

在一些系统中，只需编译程序，就可使用一些常用的库函数。

记住，在使用函数之前必须先声明函数的类型，通过包含合适的头文件即可完成。在描述库函数的用

户手册中，会指出使用某函数时应包含哪个头文件。但是在一些旧系统上，可能必须自己输入函数声明。再次提醒读者，用户手册中指明了函数类型。另外，附录 B “参考资料” 中根据头文件分组，总结了 ANSI C 库函数。

过去，不同的实现使用的头文件名不同。ANSI C 标准把库函数分为多个系列，每个系列的函数原型都放在一个特定的头文件中。

## 2. 文件包含

如果函数被定义为宏，那么可以通过`#include` 指令包含定义宏函数的文件。通常，类似的宏都放在合适名称的头文件中。例如，许多系统（包括所有的 ANSI C 系统）都有`ctype.h` 文件，该文件中包含了一些确定字符性质（如大写、数字等）的宏。

## 3. 库包含

在编译或链接程序的某些阶段，可能需要指定库选项。即使在自动检查标准库的系统中，也会有不常用的函数库。必须通过编译时选项显式指定这些库。注意，这个过程与包含头文件不同。头文件提供函数声明或原型，而库选项告诉系统到哪里查找函数代码。虽然这里无法涉及所有系统的细节，但是可以提醒读者应该注意什么。

### 16.9.2 使用库描述

篇幅有限，我们无法讨论完整的库。但是，可以看几个具有代表性的示例。首先，了解函数文档。

可以在多个地方找到函数文档。你所使用的系统可能有在线手册，集成开发环境通常都有在线帮助。C 实现的供应商可能提供描述库函数的纸质版用户手册，或者把这些材料放在 CD-ROM 中或网上。有些出版社也出版 C 库函数的参考手册。这些材料中，有些是一般材料，有些则是针对特定实现的。本书附录 B 中提供了一个库函数的总结。

阅读文档的关键是看懂函数头。许多内容随时间变化而变化。下面是旧的 UNIX 文档中，关于`read()` 的描述：

```
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

首先，给出了应该包含的文件，但是没有给出`fread()`、`ptr`、`sizeof(*ptr)` 或 `nitems` 的类型。过去，默认类型都是`int`，但是从描述中可以看出`ptr` 是一个指针（在早期的 C 中，指针被作为整数处理）。参数`stream` 声明为指向`FILE` 的指针。上面的函数声明中的第 2 个参数看上去像是`sizeof` 运算符，而实际上这个参数的值应该是`ptr` 所指向对象的大小。虽然用`sizeof` 作为参数没什么问题，但是用`int` 类型的值作为参数更符合语法。

后来，上面的描述变成了：

```
#include <stdio.h>

int fread(ptr, size, nitems, stream;
char *ptr;
int size, nitems;
FILE *stream;
```

现在，所有的类型都显式说明，`ptr` 作为指向`char` 的指针。

ANSI C90 标准提供了下面的描述：

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

首先，使用了新的函数原型格式。其次，改变了一些类型。`size_t` 类型被定义为 `sizeof` 运算符的返回值类型——无符号整数类型，通常是 `unsigned int` 或 `unsigned long`。`stddef.h` 文件中包含了 `size_t` 类型的 `typedef` 或 `#define` 定义。其他文件（包括 `stdio.h`）通过包含 `stddef.h` 来包含这个定义。许多函数（包括 `fread()`）的实际参数中都要使用 `sizeof` 运算符，形式参数的 `size_t` 类型中正好匹配这种常见的情况。

另外，ANSI C 把指向 `void` 的指针作为一种通用指针，用于指针指向不同类型的情况。例如，`fread()` 的第 1 个参数可能是指向一个 `double` 类型数组的指针，也可能是指向其他类型结构的指针。如果假设实际参数是一个指向内含 20 个 `double` 类型元素数组的指针，且形式参数是指向 `void` 的指针，那么编译器会选用合适的类型，不会出现类型冲突的问题。

C99/C11 标准在以上的描述中加入了新的关键字 `restrict`：

```
#include <stdio.h>
size_t fread(void * restrict ptr, size_t size, size_t nmemb, FILE * restrict stream);
```

接下来，我们讨论一些特殊的函数。

## 16.10 数学库

数学库中包含许多有用的数学函数。`math.h` 头文件提供这些函数的原型。表 16.2 中列出了一些声明在 `math.h` 中的函数。注意，函数中涉及的角度都以弧度为单位（1 弧度= $180/\pi=57.296$  度）。参考资料 V “新增 C99 和 C11 标准的 ANSI C 库”列出了 C99 和 C11 标准的所有函数。

表 16.2 ANSI C 标准的一些数学函数

原型	描述
<code>double acos(double x)</code>	返回余弦值为 <code>x</code> 的角度（0~ $\pi$ 弧度）
<code>double asin(double x)</code>	返回正弦值为 <code>x</code> 的角度（ $-\pi/2$ ~ $\pi/2$ 弧度）
<code>double atan(double x)</code>	返回正切值为 <code>x</code> 的角度（ $-\pi/2$ ~ $\pi/2$ 弧度）
<code>double atan2(double y, double x)</code>	返回正弦值为 <code>y/x</code> 的角度（ $-\pi$ ~ $\pi$ 弧度）
<code>double cos(double x)</code>	返回 <code>x</code> 的余弦值， <code>x</code> 的单位为弧度
<code>double sin(double x)</code>	返回 <code>x</code> 的正弦值， <code>x</code> 的单位为弧度
<code>double tan(double x)</code>	返回 <code>x</code> 的正切值， <code>x</code> 的单位为弧度
<code>double exp(double x)</code>	返回 <code>x</code> 的指数函数的值 ( $e^x$ )
<code>double log(double x)</code>	返回 <code>x</code> 的自然对数值
<code>double log10(double x)</code>	返回 <code>x</code> 的以 10 为底的对数值
<code>double pow(double x, double y)</code>	返回 <code>x</code> 的 <code>y</code> 次幂
<code>double sqrt(double x)</code>	返回 <code>x</code> 的平方值
<code>double cbrt(double x)</code>	返回 <code>x</code> 的立方值
<code>double ceil(double x)</code>	返回不小于 <code>x</code> 的最小整数值
<code>double fabs(double x)</code>	返回 <code>x</code> 的绝对值
<code>double floor(double x)</code>	返回不大于 <code>x</code> 的最大整数值

### 16.10.1 三角问题

我们可以使用数学库解决一些常见的问题：把 x/y 坐标转换为长度和角度。例如，在网格上画了一条线，该线条水平穿过了 4 个单元（x 的值），垂直穿过了 3 个单元（y 的值）。那么，该线的长度（量）和方向是什么？根据数学的三角公式可知：

$$\text{大小} = \text{square root } (x^2 + y^2)$$

$$\text{角度} = \arctan(y/x)$$

数学库提供平方根函数和一对反正切函数，所以可以用 C 程序表示这个问题。平方根函数是 `sqrt()`，接受一个 `double` 类型的参数，并返回参数的平方根，也是 `double` 类型。

`atan()` 函数接受一个 `double` 类型的参数（即正切值），并返回一个角度（该角度的正切值就是参数值）。但是，当线的 x 值和 y 值均为 -5 时，`atan()` 函数产生混乱。因为  $(-5) / (-5)$  得 1，所以 `atan()` 返回  $45^\circ$ ，该值与 x 和 y 均为 5 时的返回值相同。也就是说，`atan()` 无法区分角度相同但反向相反的线（实际上，`atan()` 返回值的单位是弧度而不是度，稍后介绍两者的转换）。

当然，C 库还提供了 `atan2()` 函数。它接受两个参数：x 的值和 y 的值。这样，通过检查 x 和 y 的正负号就可以得出正确的角度值。`atan2()` 和 `atan()` 均返回弧度值。把弧度转换为度，只需将弧度值乘以 180，再除以 pi 即可。pi 的值通过计算表达式  $4 * \text{atan}(1)$  得到。程序清单 16.13 演示了这些步骤。另外，学习该程序还复习了结构和 `typedef` 相关的知识。

程序清单 16.14 rect\_pol.c 程序

---

```
/* rect_pol.c -- 把直角坐标转换为极坐标 */
#include <stdio.h>
#include <math.h>

#define RAD_TO_DEG (180/(4 * atan(1)))

typedef struct polar_v {
    double magnitude;
    double angle;
} Polar_V;

typedef struct rect_v {
    double x;
    double y;
} Rect_V;

Polar_V rect_to_polar(Rect_V);

int main(void)
{
    Rect_V input;
    Polar_V result;

    puts("Enter x and y coordinates; enter q to quit:");
    while (scanf("%lf %lf", &input.x, &input.y) == 2)
    {
        result = rect_to_polar(input);
        printf("magnitude = %.2f, angle = %.2f\n",
               result.magnitude, result.angle);
    }
}
```

```

    puts("Bye.");

    return 0;
}

Polar_V rect_to_polar(Rect_V rv)
{
    Polar_V pv;

    pv.magnitude = sqrt(rv.x * rv.x + rv.y * rv.y);
    if (pv.magnitude == 0)
        pv.angle = 0.0;
    else
        pv.angle = RAD_TO_DEG * atan2(rv.y, rv.x);

    return pv;
}

```

下面是运行该程序后的一个输出示例：

```

Enter x and y coordinates; enter q to quit:
10 10
magnitude = 14.14, angle = 45.00
-12 -5
magnitude = 13.00, angle = -157.38
q
Bye.

```

如果编译时出现下面的消息：

```

Undefined: _sqrt
或
'sqrt': unresolved external

```

或者其他类似的消息，表明编译器链接器没有找到数学库。UNIX 系统会要求使用-lm 标记（flag）指示链接器搜索数学库：

```
cc rect_pol.c -lm
```

注意，-lm 标记在命令行的末尾。因为链接器在编译器编译 C 文件后才开始处理。在 Linux 中使用 GCC 编译器可能要这样写：

```
gcc rect_pol.c -lm
```

## 16.10.2 类型变体

基本的浮点型数学函数接受 double 类型的参数，并返回 double 类型的值。当然，也可以把 float 或 long double 类型的参数传递给这些函数，它们仍然能正常工作，因为这些类型的参数会被转换成 double 类型。这样做很方便，但并不是最好的处理方式。如果不需要双精度，那么用 float 类型的单精度值来计算会更快些。而且把 long double 类型的值传递给 double 类型的形参会损失精度，形参获得的值可能不是原来的值。为了解决这些潜在的问题，C 标准专门为 float 类型和 long double 类型提供了标准函数，即在原函数名前加上 f 或 l 前缀。因此，sqrtf() 是 sqrt() 的 float 版本，sqrtl() 是 sqrt() 的 long double 版本。

利用 C11 新增的泛型选择表达式定义一个泛型宏，根据参数类型选择最合适的数据函数版本。程序清单 16.15 演示了两种方法。

## 程序清单 16.15 generic.c 程序

```
// generic.c -- 定义泛型宏

#include <stdio.h>
#include <math.h>
#define RAD_TO_DEG (180/(4 * atanl(1)))

// 泛型平方根函数
#define SQRT(X) _Generic((X), \
    long double: sqrtl, \
    default: sqrt, \
    float: sqrtf)(X)

// 泛型正弦函数，角度的单位为度
#define SIN(X) _Generic((X), \
    long double: sinl((X)/RAD_TO_DEG), \
    default:     sin((X)/RAD_TO_DEG), \
    float:       sinf((X)/RAD_TO_DEG) \
)

int main(void)
{
    float x = 45.0f;
    double xx = 45.0;
    long double xxx = 45.0L;

    long double y = SQRT(x);
    long double yy = SQRT(xx);
    long double yyyy = SQRT(xxx);
    printf("%.17Lf\n", y);           // 匹配 float
    printf("%.17Lf\n", yy);          // 匹配 default
    printf("%.17Lf\n", yyyy);        // 匹配 long double
    int i = 45;
    yy = SQRT(i);                  // 匹配 default
    printf("%.17Lf\n", yy);
    yyyy = SIN(xxx);               // 匹配 long double
    printf("%.17Lf\n", yyyy);

    return 0;
}
```

下面是该程序的输出：

```
6.70820379257202148
6.70820393249936942
6.70820393249936909
6.70820393249936942
0.70710678118654752
```

如上所示，`SQRT(i)` 和 `SQRT(xx)` 的返回值相同，因为它们的参数类型分别是 `int` 和 `double`，所以只能与 `default` 标签对应。

有趣的一点是，如何让 `_Generic` 宏的行为像一个函数。`SIN()` 的定义也许提供了一个方法：每个带标号的值都是函数调用，所以 `_Generic` 表达式的值是一个特定的函数调用，如 `sinf((X)/RAD_TO_DEG)`，

用传入 `SIN()` 的参数替换 `x`。

`SQRT()` 的定义也许更简洁。`_Generic` 表达式的值就是函数名，如 `sinf`。函数的地址可以代替该函数名，所以 `_Generic` 表达式的值是一个指向函数的指针。然而，紧随整个 `_Generic` 表达式之后的是 `(x)`，函数指针(参数)表示函数指针。因此，这是一个带指定的参数的函数指针。

简而言之，对于 `SIN()`，函数调用在泛型选择表达式内部；而对于 `SQRT()`，先对泛型选择表达式求值得一个指针，然后通过该指针调用它所指向的函数。

### 16.10.3 tgmath.h 库 (C99)

C99 标准提供的 `tgmath.h` 头文件中定义了泛型类型宏，其效果与程序清单 16.15 类似。如果在 `math.h` 中为一个函数定义了 3 种类型 (`float`、`double` 和 `long double`) 的版本，那么 `tgmath.h` 文件就创建一个泛型类型宏，与原来 `double` 版本的函数名同名。例如，根据提供的参数类型，定义 `sqrt()` 宏展开为 `sqrtf()`、`sqrt()` 或 `sqrtl()` 函数。换言之，`sqrt()` 宏的行为和程序清单 16.15 中的 `SQRT()` 宏类似。

如果编译器支持复数运算，就会支持 `complex.h` 头文件，其中声明了与复数运算相关的函数。例如，声明有 `csqrtf()`、`csqrt()` 和 `csqrtn()`，这些函数分别返回 `float complex`、`double complex` 和 `long double complex` 类型的复数平方根。如果提供这些支持，那么 `tgmath.h` 中的 `sqrt()` 宏也能展开为相应的复数平方根函数。

如果包含了 `tgmath.h`，要调用 `sqrt()` 函数而不是 `sqrt()` 宏，可以用圆括号把被调用的函数名括起来：

```
#include <tgmath.h>
...
float x = 44.0;
double y;
y = sqrt(x);           // 调用宏，所以是 sqrtf(x)
y = (sqrt)(x);         // 调用函数 sqrt()
```

这样做没问题，因为类函数宏的名称必须用圆括号括起来。圆括号只会影响操作顺序，不会影响括起来的表达式，所以这样做得到的仍然是函数调用的结果。实际上，在讨论函数指针时提到过，由于 C 语言奇怪而矛盾的函数指针规则，还可以使用 `(*sqrt)()` 的形式来调用 `sqrt()` 函数。

不借助 C 标准以外的机制，C11 新增的 `_Generic` 表达式是实现 `tgmath.h` 最简单的方式。

## 16.11 通用工具库

通用工具库包含各种函数，包括随机数生成器、查找和排序函数、转换函数和内存管理函数。第 12 章介绍过 `rand()`、`srand()`、`malloc()` 和 `free()` 函数。在 ANSI C 标准中，这些函数的原型都在 `stdlib.h` 头文件中。附录 B 参考资料 V 列出了该系列的所有函数。现在，我们来进一步讨论其中的几个函数。

### 16.11.1 exit() 和 atexit() 函数

在前面的章节中我们已经在程序示例中用过 `exit()` 函数。而且，在 `main()` 返回系统时将自动调用 `exit()` 函数。ANSI 标准还新增了一些不错的功能，其中最重要的是可以指定在执行 `exit()` 时调用的特定函数。`atexit()` 函数通过退出时注册被调用的函数提供这种功能，`atexit()` 函数接受一个函数指针作为参数。程序清单 16.16 演示了它的用法。

## 程序清单 16.16 byebye.c 程序

```

/* byebye.c -- atexit() 示例 */
#include <stdio.h>
#include <stdlib.h>
void sign_off(void);
void too_bad(void);

int main(void)
{
    int n;

    atexit(sign_off);      /* 注册 sign_off() 函数 */
    puts("Enter an integer:");
    if (scanf("%d", &n) != 1)
    {
        puts("That's no integer!");
        atexit(too_bad); /* 注册 too_bad() 函数 */
        exit(EXIT_FAILURE);
    }
    printf("%d is %s.\n", n, (n % 2 == 0) ? "even" : "odd");

    return 0;
}

void sign_off(void)
{
    puts("Thus terminates another magnificent program from");
    puts("SeeSaw Software!");
}

void too_bad(void)
{
    puts("SeeSaw Software extends its heartfelt condolences");
    puts("to you upon the failure of your program.");
}

```

下面是该程序的一个运行示例：

```

Enter an integer:
212
212 is even.
Thus terminates another magnificent program from
SeeSaw Software!

```

如果在 IDE 中运行，可能看不到最后两行。下面是另一个运行示例：

```

Enter an integer:
what?
That's no integer!
SeeSaw Software extends its heartfelt condolences
to you upon the failure of your program.
Thus terminates another magnificent program from
SeeSaw Software!

```

在 IDE 中运行，可能看不到最后 4 行。

接下来，我们讨论 atexit() 和 exit() 的参数。

## 1. atexit() 函数的用法

这个函数使用函数指针。要使用 `atexit()` 函数，只需把退出时要调用的函数地址传递给 `atexit()` 即可。函数名作为函数参数时相当于该函数的地址，所以该程序中把 `sign_off` 或 `too_bad` 作为参数。然后，`atexit()` 注册函数列表中的函数，当调用 `exit()` 时就会执行这些函数。ANSI 保证，在这个列表中至少可以放 32 个函数。最后调用 `exit()` 函数时，`exit()` 会执行这些函数（执行顺序与列表中的函数顺序相反，即最后添加的函数最先执行）。

注意，输入失败时，会调用 `sign_off()` 和 `too_bad()` 函数；但是输入成功时只会调用 `sign_off()`。因为只有输入失败时，才会进入 `if` 语句中注册 `too_bad()`。另外还要注意，最先调用的是最后一个被注册的函数。

`atexit()` 注册的函数（如 `sign_off()` 和 `too_bad()`）应该不带任何参数且返回类型为 `void`。通常，这些函数会执行一些清理任务，例如更新监视程序的文件或重置环境变量。

注意，即使没有显式调用 `exit()`，还是会调用 `sign_off()`，因为 `main()` 结束时会隐式调用 `exit()`。

## 2. exit() 函数的用法

`exit()` 执行完 `atexit()` 指定的函数后，会完成一些清理工作：刷新所有输出流、关闭所有打开的流和关闭由标准 I/O 函数 `tmpfile()` 创建的临时文件。然后 `exit()` 把控制权返回主机环境，如果可能的话，向主机环境报告终止状态。通常，UNIX 程序使用 0 表示成功终止，用非零值表示终止失败。UNIX 返回的代码并不适用于所有的系统，所以 ANSI C 为了可移植性的要求，定义了一个名为 `EXIT_FAILURE` 的宏表示终止失败。类似地，ANSI C 还定义了 `EXIT_SUCCESS` 表示成功终止。不过，`exit()` 函数也接受 0 表示成功终止。在 ANSI C 中，在非递归的 `main()` 中使用 `exit()` 函数等价于使用关键字 `return`。尽管如此，在 `main()` 以外的函数中使用 `exit()` 也会终止整个程序。

### 16.11.2 qsort() 函数

对较大型的数组而言，“快速排序”方法是最有效的排序算法之一。该算法由 C.A.R.Hoare 于 1962 年开发。它把数组不断分成更小的数组，直到变成单元素数组。首先，把数组分成两部分，一部分的值都小于另一部分的值。这个过程一直持续到数组完全排序好为止。

快速排序算法在 C 实现中的名称是 `qsort()`。`qsort()` 函数排序数组的数据对象，其原型如下：

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

第 1 个参数是指针，指向待排序数组的首元素。ANSI C 允许把指向任何数据类型的指针强制转换成指向 `void` 的指针，因此，`qsort()` 的第 1 个实际参数可以引用任何类型的数组。

第 2 个参数是待排序项的数量。函数原型把该值转换为 `size_t` 类型。前面提到过，`size_t` 定义在标准头文件中，是 `sizeof` 运算符返回的整数类型。

由于 `qsort()` 把第 1 个参数转换为 `void` 指针，所以 `qsort()` 不知道数组中每个元素的大小。为此，函数原型用第 3 个参数补偿这一信息，显式指明待排序数组中每个元素的大小。例如，如果排序 `double` 类型的数组，那么第 3 个参数应该是 `sizeof(double)`。

最后，`qsort()` 还需要一个指向函数的指针，这个被指针指向的比较函数用于确定排序的顺序。该函数应接受两个参数：分别指向待比较两项的指针。如果第 1 项的值大于第 2 项，比较函数则返回正数；如果两项相同，则返回 0；如果第 1 项的值小于第 2 项，则返回负数。`qsort()` 根据给定的其他信息计算出两个指针的值，然后把它们传递给比较函数。

`qsort()`原型中的第4个函数确定了比较函数的形式：

```
int (*compar)(const void *, const void *)
```

这表明 `qsort()`最后一个参数是一个指向函数的指针，该函数返回 `int` 类型的值且接受两个指向 `const void` 的指针作为参数，这两个指针指向待比较项。

程序清单 16.17 和后面的讨论解释了如何定义一个比较函数，以及如何使用 `qsort()`。该程序创建了一个内含随机浮点值的数组，并排序了这个数组。

程序清单 16.17 `qsorcer.c` 程序

---

```
/* qsorcer.c -- 用 qsort() 排序一组数字 */
#include <stdio.h>
#include <stdlib.h>

#define NUM 40
void fillarray(double ar [], int n);
void showarray(const double ar [], int n);
int mycomp(const void * p1, const void * p2);

int main(void)
{
    double vals[NUM];
    fillarray(vals, NUM);
    puts("Random list:");
    showarray(vals, NUM);
    qsort(vals, NUM, sizeof(double), mycomp);
    puts("\nSorted list:");
    showarray(vals, NUM);
    return 0;
}

void fillarray(double ar [], int n)
{
    int index;

    for (index = 0; index < n; index++)
        ar[index] = (double) rand() / ((double) rand() + 0.1);
}

void showarray(const double ar [], int n)
{
    int index;

    for (index = 0; index < n; index++)
    {
        printf("%9.4f ", ar[index]);
        if (index % 6 == 5)
            putchar('\n');
    }
    if (index % 6 != 0)
        putchar('\n');
}

/* 按从小到大的顺序排序 */
int mycomp(const void * p1, const void * p2)
```

```

{
    /* 要使用指向 double 的指针来访问这两个值 */
    const double * a1 = (const double *) p1;
    const double * a2 = (const double *) p2;

    if (*a1 < *a2)
        return -1;
    else if (*a1 == *a2)
        return 0;
    else
        return 1;
}

```

下面是该程序的运行示例：

```

Random list:
0.0001  1.6475  2.4332  0.0693  0.7268  0.7383
24.0357 0.1009  87.1828  5.7361  0.6079  0.6330
1.6058  0.1406  0.5933  1.1943  5.5295  2.2426
0.8364  2.7127  0.2514  0.9593  8.9635  0.7139
0.6249  1.6044  0.8649  2.1577  0.5420  15.0123
1.7931  1.6183  1.9973  2.9333  12.8512  1.3034
0.3032  1.1406  18.7880  0.9887

Sorted list:
0.0001  0.0693  0.1009  0.1406  0.2514  0.3032
0.5420  0.5933  0.6079  0.6249  0.6330  0.7139
0.7268  0.7383  0.8364  0.8649  0.9593  0.9887
1.1406  1.1943  1.3034  1.6044  1.6058  1.6183
1.6475  1.7931  1.9973  2.1577  2.2426  2.4332
2.7127  2.9333  5.5295  5.7361  8.9635  12.8512
15.0123 18.7880  24.0357  87.1828

```

接下来分析两点：qsort() 的用法和 mycomp() 的定义。

## 1. qsort() 的用法

qsort() 函数排序数组的数据对象。该函数的 ANSI 原型如下：

```

void qsort (void *base, size_t nmemb, size_t size,
            int (*compar) (const void *, const void *));

```

第 1 个参数值指向待排序数组首元素的指针。在该程序中，实际参数是 double 类型的数组名 vals，因此指针指向该数组的首元素。根据该函数的原型，参数 vals 会被强制转换成指向 void 的指针。由于 ANSI C 允许把指向任何数据类型的指针强制转换成指向 void 的指针，所以 qsort() 的第 1 个实际参数可以引用任何类型的数组。

第 2 个参数是待排序项的数量。在程序清单 16.17 中是 NUM，即数组元素的数量。函数原型把该值转换为 size\_t 类型。

第 3 个参数是数组中每个元素占用的空间大小，本例中为 sizeof(double)。

最后一个参数是 mycomp，这里函数名即是函数的地址，该函数用于比较元素。

## 2. mycomp() 的定义

前面提到过，qsort() 的原型中规定了比较函数的形式：

```

int (*compar) (const void *, const void *)

```

这表明 `qsort()` 最后一个参数是一个指向函数的指针，该函数返回 `int` 类型的值且接受两个指向 `const void` 的指针作为参数。程序中 `mycomp()` 使用的就是这个原型：

```
int mycomp(const void * p1, const void * p2);
```

记住，函数名作为参数时即是指向该函数的指针。因此，`mycomp` 与 `compar` 原型相匹配。

`qsort()` 函数把两个待比较元素的地址传递给比较函数。在该程序中，把待比较的两个 `double` 类型值的地址赋给 `p1` 和 `p2`。注意，`qsort()` 的第 1 个参数引用整个数组，比较函数中的两个参数引用数组中的两个元素。这里存在一个问题。为了比较指针所指向的值，必须解引用指针。因为值是 `double` 类型，所以要把指针解引用为 `double` 类型的值。然而，`qsort()` 要求指针指向 `void`。要解决这个问题，必须在比较函数的内部声明两个类型正确的指针，并初始化它们分别指向作为参数传入的值：

```
/* 按从小到大的顺序排序值 */
int mycomp(const void * p1, const void * p2)
{
    /* 使用指向 double 类型的指针访问值 */
    const double * a1 = (const double *) p1;
    const double * a2 = (const double *) p2;
    if (*a1 < *a2)
        return -1;
    else if (*a1 == *a2)
        return 0;
    else
        return 1;
}
```

简而言之，为了让该方法具有通用性，`qsort()` 和比较函数使用了指向 `void` 的指针。因此，必须把数组中每个元素的大小明确告诉 `qsort()`，并且在比较函数的定义中，必须把该函数的指针参数转换为对具体应用而言类型正确的指针。

### 注意 C 和 C++ 中的 `void*`

C 和 C++ 对待指向 `void` 的指针有所不同。在这两种语言中，都可以把任何类型的指针赋给 `void` 类型的指针。例如，程序清单 16.17 中，`qsort()` 的函数调用中把 `double*` 指针赋给 `void*` 指针。但是，C++ 要求在把 `void*` 指针赋给任何类型的指针时必须进行强制类型转换。而 C 没有这样的要求。

例如，程序清单 16.17 中的 `mycomp()` 函数，就使用了这样的强制类型转换：

```
const double * a1 = (const double *) p1;
```

这种强制类型转换，在 C 中是可选的，但在 C++ 中是必须的。因为两种语言都使用强制类型转换，所以遵循 C++ 的要求也无不可。将来如果要把该程序转成 C++，就不必更改这部分的代码。

下面再来看一个比较函数的例子。假设有下面的声明：

```
struct names {
    char first[40];
    char last[40];
};

struct names staff[100];
```

如何调用 `qsort()`？模仿程序清单 16.17 中 `qsort()` 的函数调用，应该是这样：

```
qsort(staff, 100, sizeof(struct names), comp);
```

这里 `comp` 是比较函数的函数名。那么，应如何编写这个函数？假设要先按姓排序，如果同姓再按名

排序，可以这样编写该函数：

```
#include <string.h>
int comp(const void * p1, const void * p2) /* 该函数的形式必须是这样 */
{
    /* 得到正确类型的指针 */
    const struct names *ps1 = (const struct names *) p1;
    const struct names *ps2 = (const struct names *) p2;
    int res;
    res = strcmp(ps1->last, ps2->last); /* 比较姓 */
    if (res != 0)
        return res;
    else /* 如果同姓，则比较名 */
        return strcmp(ps1->first, ps2->first);
}
```

该函数使用 `strcmp()` 函数进行比较。`strcmp()` 的返回值与比较函数的要求相匹配。注意，通过指针访问结构成员时必须使用`->`运算符。

## 16.12 断言库

`assert.h` 头文件支持的断言库是一个用于辅助调试程序的小型库。它由 `assert()` 宏组成，接受一个整型表达式作为参数。如果表达式求值为假（非零），`assert()` 宏就在标准错误流（`stderr`）中写入一条错误信息，并调用 `abort()` 函数终止程序（`abort()` 函数的原型在 `stdlib.h` 头文件中）。`assert()` 宏是为了标识出程序中某些条件为真的关键位置，如果其中的一个具体条件为假，就用 `assert()` 语句终止程序。通常，`assert()` 的参数是一个条件表达式或逻辑表达式。如果 `assert()` 中止了程序，它首先会显示失败的测试、包含测试的文件名和行号。

### 16.12.1 assert 的用法

程序清单 16.18 演示了一个使用 `assert` 的小程序。在求平方根之前，该程序断言 `z` 是否大于或等于 0。程序还错误地减去一个值而不是加上一个值，故意让 `z` 得到不合适的值。

程序清单 16.18 assert.c 程序

```
/* assert.c -- 使用 assert() */
#include <stdio.h>
#include <math.h>
#include <assert.h>
int main()
{
    double x, y, z;

    puts("Enter a pair of numbers (0 0 to quit): ");
    while (scanf("%lf%lf", &x, &y) == 2
           && (x != 0 || y != 0))
    {
        z = x * x - y * y; /* 应该用 + */
        assert(z >= 0);
        printf("answer is %f\n", sqrt(z));
        puts("Next pair of numbers: ");
    }
    puts("Done");
```

```

    return 0;
}

```

下面是该程序的运行示例：

```

Enter a pair of numbers (0 0 to quit):
4 3
answer is 2.645751
Next pair of numbers:
5 3
answer is 4.000000
Next pair of numbers:
3 5
Assertion failed: (z >= 0), function main, file /Users/assert.c, line 14.

```

具体的错误提示因编译器而异。让人困惑的是，这条消息可能不是指明  $z \geq 0$ ，而是指明没有满足  $z \geq 0$  的条件。

用 if 语句也能完成类似的任务：

```

if (z < 0)
{
    puts("z less than 0");
    abort();
}

```

但是，使用 assert() 有几个好处：它不仅能自动标识文件和出问题的行号，还有一种无需更改代码就能开启或关闭 assert() 的机制。如果认为已经排除了程序的 bug，就可以把下面的宏定义写在包含 assert.h 的位置前面：

```
#define NDEBUG
```

并重新编译程序，这样编译器就会禁用文件中的所有 assert() 语句。如果程序又出现问题，可以移除这条#define 指令（或者把它注释掉），然后重新编译程序，这样就重新启用了 assert() 语句。

## 16.12.2 \_Static\_assert (C11)

assert() 表达式是在运行时进行检查。C11 新增了一个特性：\_Static\_assert 声明，可以在编译时检查 assert() 表达式。因此，assert() 可以导致正在运行的程序中止，而 \_Static\_assert() 可以导致程序无法通过编译。\_Static\_assert() 接受两个参数。第 1 个参数是整型常量表达式，第 2 个参数是一个字符串。如果第 1 个表达式求值为 0（或\_False），编译器会显示字符串，而且不编译该程序。看看程序清单 16.19 的小程序，然后查看 assert() 和 \_Static\_assert() 的区别。

**程序清单 16.19 statasrt.c 程序**

```

// statasrt.c
#include <stdio.h>
#include <limits.h>
_Static_assert(CHAR_BIT == 16, "16-bit char falsely assumed");
int main(void)
{
    puts("char is 16 bits.");
    return 0;
}

```

下面是在命令行编译的示例：

```
$ clang statasrt.c
statasrt.c:4:1: error: static_assert failed "16-bit char falsely assumed"
_Static_assert(CHAR_BIT == 16, "16-bit char falsely assumed");
^
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
1 error generated.
$
```

根据语法, `_Static_assert()` 被视为声明。因此, 它可以出现在函数中, 或者在这种情况下出现在函数的外部。

`_Static_assert` 要求它的第 1 个参数是整型常量表达式, 这保证了能在编译期求值 (`sizeof` 表达式被视为整型常量)。不能用程序清单 16.18 中的 `assert` 代替 `_Static_assert`, 因为 `assert` 中作为测试表达式的 `z > 0` 不是常量表达式, 要到程序运行时才求值。当然, 可以在程序清单 16.19 的 `main()` 函数中使用 `assert(CHAR_BIT == 16)`, 但这会在编译和运行程序后才生成一条错误信息, 很没效率。

## 16.13 string.h 库中的 `memcpy()` 和 `memmove()`

不能把一个数组赋给另一个数组, 所以要通过循环把数组中的每个元素赋给另一个数组相应的元素。有一个例外的情况是: 使用 `strcpy()` 和 `strncpy()` 函数来处理字符串。`memcpy()` 和 `memmove()` 函数提供类似的方法处理任意类型的数组。下面是这两个函数的原型:

```
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 `s2` 指向的位置拷贝 `n` 字节到 `s1` 指向的位置, 而且都返回 `s1` 的值。所不同的是, `memcpy()` 的参数带关键字 `restrict`, 即 `memcpy()` 假设两个内存区域之间没有重叠; 而 `memmove()` 不作这样的假设, 所以拷贝过程类似于先把所有字节拷贝到一个临时缓冲区, 然后再拷贝到最终目的地。如果使用 `memcpy()` 时, 两区域出现重叠会怎样? 其行为是未定义的, 这意味着该函数可能正常工作, 也可能失败。编译器不会在本不该使用 `memcpy()` 时禁止你使用, 作为程序员, 在使用该函数时有责任确保两个区域不重叠。

由于这两个函数设计用于处理任何数据类型, 所有它们的参数都是两个指向 `void` 的指针。C 允许把任何类型的指针赋给 `void *` 类型的指针。如此宽容导致函数无法知道待拷贝数据的类型。因此, 这两个函数使用第 3 个参数指明待拷贝的字节数。注意, 对数组而言, 字节数一般与元素个数不同。如果要拷贝数组中 10 个 `double` 类型的元素, 要使用 `10*sizeof(double)`, 而不是 10。

程序清单 16.20 中的程序使用了这两个函数。该程序假设 `double` 类型是 `int` 类型的两倍大小。另外, 该程序还使用了 C11 的 `_Static_assert` 特性测试断言。

程序清单 16.20 mems.c 程序

```
// mems.c -- 使用 memcpy() 和 memmove()
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define SIZE 10
void show_array(const int ar [], int n);
// 如果编译器不支持C11的_Static_assert, 可以注释掉下面这行
_Static_assert(sizeof(double) == 2 * sizeof(int), "double not twice int size");
int main()
{
    int values[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int target[SIZE];
```

```

double curious[SIZE / 2] = { 2.0, 2.0e5, 2.0e10, 2.0e20, 5.0e30 };

puts("memcpy() used:");
puts("values (original data): ");
show_array(values, SIZE);
memcpy(target, values, SIZE * sizeof(int));
puts("target (copy of values):");
show_array(target, SIZE);

puts("\nUsing memmove() with overlapping ranges:");
memmove(values + 2, values, 5 * sizeof(int));
puts("values -- elements 0-4 copied to 2-6:");
show_array(values, SIZE);

puts("\nUsing memcpy() to copy double to int:");
memcpy(target, curious, (SIZE / 2) * sizeof(double));
puts("target -- 5 doubles into 10 int positions:");
show_array(target, SIZE / 2);
show_array(target + 5, SIZE / 2);

return 0;
}

void show_array(const int ar [], int n)
{
    int i;

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

```

下面是该程序的输出：

```

memcpy() used:
values (original data):
1 2 3 4 5 6 7 8 9 10
target (copy of values):
1 2 3 4 5 6 7 8 9 10

Using memmove() with overlapping ranges:
values -- elements 0-4 copied to 2-6:
1 2 1 2 3 4 5 8 9 10

Using memcpy() to copy double to int:
target -- 5 doubles into 10 int positions:
0 1073741824 0 1091070464 536870912
1108516959 2025163840 1143320349 -2012696540 1179618799

```

程序中最后一次调用 memcpy() 从 double 类型数组中把数据拷贝到 int 类型数组中，这演示了 memcpy() 函数不知道也不关心数据的类型，它只负责从一个位置把一些字节拷贝到另一个位置（例如，从结构中拷贝数据到字符数组中）。而且，拷贝过程中也不会进行数据转换。如果用循环对数组中的每个元素赋值，double 类型的值会在赋值过程被转换为 int 类型的值。这种情况下，按原样拷贝字节，然后程序把这些位组合解释成 int 类型。

## 16.14 可变参数: stdarg.h

本章前面提到过变参宏, 即该宏可以接受可变数量的参数。stdarg.h 头文件为函数提供了一个类似的功能, 但是用法比较复杂。必须按如下步骤进行:

1. 提供一个使用省略号的函数原型;
2. 在函数定义中创建一个 va\_list 类型的变量;
3. 用宏把该变量初始化为一个参数列表;
4. 用宏访问参数列表;
5. 用宏完成清理工作。

接下来详细分析这些步骤。这种函数的原型应该有一个形参列表, 其中至少有一个形参和一个省略号:

```
void f1(int n, ...);           // 有效
int f2(const char * s, int k, ...); // 有效
char f3(char c1, ..., char c2); // 无效, 省略号不在最后
double f3(...);                // 无效, 没有形参
```

最右边的形参 (即省略号的前一个形参) 起着特殊的作用, 标准中用 *parmN* 这个术语来描述该形参。在上面的例子中, 第 1 行 *f1()* 中 *parmN* 为 *n*, 第 2 行 *f2()* 中 *parmN* 为 *k*。传递给该形参的实际参数是省略号部分代表的参数数量。例如, 可以这样使用前面声明的 *f1()* 函数:

```
f1(2, 200, 400);           // 2 个额外的参数
f1(4, 13, 117, 18, 23);    // 4 个额外的参数
```

接下来, 声明在 stdarg.h 中的 va\_list 类型代表一种用于储存形参对应的形参列表中省略号部分的数据对象。变参函数的定义起始部分类似下面这样:

```
double sum(int lim,...)
{
    va_list ap; // 声明一个储存参数的对象
```

在该例中, *lim* 是 *parmN* 形参, 它表明变参列表中参数的数量。

然后, 该函数将使用定义在 stdarg.h 中的 va\_start() 宏, 把参数列表拷贝到 va\_list 类型的变量中。该宏有两个参数: va\_list 类型的变量和 *parmN* 形参。接着上面的例子讨论, va\_list 类型的变量是 *ap*, *parmN* 形参是 *lim*。所以, 应这样调用它:

```
va_start(ap, lim); // 把 ap 初始化为参数列表
```

下一步是访问参数列表的内容, 这涉及使用另一个宏 va\_arg()。该宏接受两个参数: 一个 va\_list 类型的变量和一个类型名。第 1 次调用 va\_arg() 时, 它返回参数列表的第一项; 第 2 次调用时返回第二项, 以此类推。表示类型的参数指定了返回值的类型。例如, 如果参数列表中的第一个参数是 double 类型, 第 2 个参数是 int 类型, 可以这样做:

```
double tic;
int toc;
...
tic = va_arg(ap, double); // 检索第 1 个参数
toc = va_arg(ap, int);    // 检索第 2 个参数
```

注意, 传入的参数类型必须与宏参数的类型相匹配。如果第 1 个参数是 10.0, 上面 *tic* 那行代码可以正常工作。但是如果参数是 10, 这行代码可能会出错。这里不会像赋值那样把 double 类型自动转换成 int 类型。

最后, 要使用 `va_end()` 宏完成清理工作。例如, 释放动态分配用于储存参数的内存。该宏接受一个 `va_list` 类型的变量:

```
va_end(ap); // 清理工作
```

调用 `va_end(ap)` 后, 只有用 `va_start` 重新初始化 `ap` 后, 才能使用变量 `ap`。

因为 `va_arg()` 不提供退回之前参数的方法, 所以有必要保存 `va_list` 类型变量的副本。C99 新增了一个宏用于处理这种情况: `va_copy()`。该宏接受两个 `va_list` 类型的变量作为参数, 它把第 2 个参数拷贝给第 1 个参数:

```
va_list ap;
va_list apcopy;
double
double tic;
int toc;
...
va_start(ap, lim); // 把 ap 初始化为一个参数列表
va_copy(apcopy, ap); // 把 apcopy 作为 ap 的副本
tic = va_arg(ap, double); // 检索第 1 个参数
toc = va_arg(ap, int); // 检索第 2 个参数
```

此时, 即使删除了 `ap`, 也可以从 `apcopy` 中检索两个参数。

程序清单 16.21 中的程序示例中演示了如何创建这样的函数, 该函数对可变参数求和。`sum()` 的第 1 个参数是待求和项的数目。

程序清单 16.21 varargs.c 程序

---

```
//varargs.c -- use variable number of arguments
#include <stdio.h>
#include <stdarg.h>
double sum(int, ...);

int main(void)
{
    double s, t;

    s = sum(3, 1.1, 2.5, 13.3);
    t = sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1);
    printf("return value for "
          "sum(3, 1.1, 2.5, 13.3): %g\n", s);
    printf("return value for "
          "sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): %g\n", t);

    return 0;
}

double sum(int lim, ...)
{
    va_list ap; // 声明一个对象储存参数
    double tot = 0;
    int i;

    va_start(ap, lim); // 把 ap 初始化为参数列表
    for (i = 0; i < lim; i++)
        tot += va_arg(ap, double); // 访问参数列表中的每一项
```

```

va_end(ap);           // 清理工作

return tot;
}

```

下面是该程序的输出：

```

return value for sum(3, 1.1, 2.5, 13.3):      16.9
return value for sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): 31.6

```

查看程序中的运算可以发现，第 1 次调用 `sum()` 时对 3 个数求和，第 2 次调用时对 6 个数求和。总而言之，使用变参函数比使用变参宏更复杂，但是函数的应用范围更广。

## 16.15 关键概念

C 标准不仅描述 C 语言，还描述了组成 C 语言的软件包、C 预处理器和 C 标准库。通过预处理器可以控制编译过程、列出要替换的内容、指明要编译的代码行和影响编译器其他方面的行为。C 库扩展了 C 语言的作用范围，为许多编程问题提供现成的解决方案。

## 16.16 本章小结

C 预处理器和 C 库是 C 语言的两个重要的附件。C 预处理器遵循预处理器指令，在编译源代码之前调整源代码。C 库提供许多有助于完成各种任务的函数，包括输入、输出、文件处理、内存管理、排序与搜索、数学运算、字符串处理等。附录 B 的参考资料 V 中列出了完整的 ANSI C 库。

## 16.17 复习题

- 下面的几组代码由一个或多个宏组成，其后是使用宏的源代码。在每种情况下代码的结果是什么？这些代码是否是有效代码？（假设其中的变量已声明）

a.

```
#define FPM 5280 /*每英里的英尺数*/
dist = FPM * miles;
```

b.

```
#define FEET 4
#define POD FEET + FEET
plot = FEET * POD;
```

c.

```
#define SIX = 6;
nex = SIX;
```

d.

```
#define NEW(X) X + 5
y = NEW(y);
berg = NEW(berg) * lob;
est = NEW(berg) / NEW(y);
nilp = lob * NEW(-berg);
```

- 修改复习题 1 中 d 部分的定义，使其更可靠。
- 定义一个宏函数，返回两值中的较小值。

4. 定义 EVEN\_GT(X, Y) 宏，如果 X 为偶数且大于 Y，该宏返回 1。
5. 定义一个宏函数，打印两个表达式及其值。例如，若参数为 3+4 和 4\*12，则打印：  
3+4 is 7 and 4\*12 is 48
6. 创建#define 指令完成下面的任务。
  - a. 创建一个值为 25 的命名常量。
  - b. SPACE 表示空格字符。
  - c. PS() 代表打印空格字符。
  - d. BIG(X) 代表 X 的值加 3。
  - e. SUMSQ(X, Y) 代表 X 和 Y 的平方和。
7. 定义一个宏，以下面的格式打印名称、值和 int 类型变量的地址：  
name: fop; value: 23; address: ff464016
8. 假设在测试程序时要暂时跳过一块代码，如何在不移除这块代码的前提下完成这项任务？
9. 编写一段代码，如果定义了 PR\_DATE 宏，则打印预处理的日期。
10. 内联函数部分讨论了 3 种不同版本的 square() 函数。从行为方面看，这 3 种版本的函数有何不同？
11. 创建一个使用泛型选择表达式的宏，如果宏参数是 \_Bool 类型，对"boolean"求值，否则对"not boolean"求值。
12. 下面的程序有什么错误？
 

```
#include <stdio.h>
int main(int argc, char argv[])
{
    printf("The square root of %f is %f\n", argv[1], sqrt(argv[1]));
}
```
13. 假设 scores 是内含 1000 个 int 类型元素的数组，要按降序排序该数组中的值。假设你使用 qsort() 和 comp() 比较函数。
  - a. 如何正确调用 qsort()？
  - b. 如何正确定义 comp()？
14. 假设 data1 是内含 100 个 double 类型元素的数组，data2 是内含 300 个 double 类型元素的数组。
  - a. 编写 memcpy() 的函数调用，把 data2 中的前 100 个元素拷贝到 data1 中。
  - b. 编写 memcpy() 的函数调用，把 data2 中的后 100 个元素拷贝到 data1 中。

## 16.18 编程练习

1. 开发一个包含你需要的预处理器定义的头文件。
2. 两数的调和平均数这样计算：先得到两数的倒数，然后计算两个倒数的平均值，最后取计算结果的倒数。使用#define 指令定义一个宏“函数”，执行该运算。编写一个简单的程序测试该宏。
3. 极坐标用向量的模（即向量的长度）和向量相对 x 轴逆时针旋转的角度来描述该向量。直角坐标用向量的 x 轴和 y 轴的坐标来描述该向量（见图 16.3）。编写一个程序，读取向量的模和角度（单位：

度), 然后显示 x 轴和 y 轴的坐标。相关方程如下:

$$x = r \cos A \quad y = r \sin A$$

需要一个函数来完成转换, 该函数接受一个包含极坐标的结构, 并返回一个包含直角坐标的结构(或返回指向该结构的指针)。

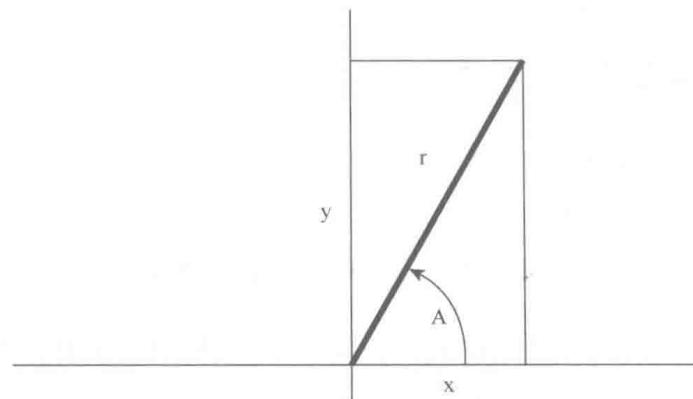


图 16.3 直角坐标和极坐标

#### 4. ANSI 库这样描述 `clock()` 函数的特性:

```
#include <time.h>
clock_t clock (void);
```

这里, `clock_t` 是定义在 `time.h` 中的类型。该函数返回处理器时间, 其单位取决于实现(如果处理器时间不可用或无法表示, 该函数将返回-1)。然而, `CLOCKS_PER_SEC`(也定义在 `time.h` 中)是每秒处理器时间单位的数量。因此, 两个 `clock()` 返回值的差值除以 `CLOCKS_PER_SEC` 得到两次调用之间经过的秒数。在进行除法运算之前, 把值的类型强制转换成 `double` 类型, 可以将时间精确到小数点以后。编写一个函数, 接受一个 `double` 类型的参数表示时间延迟数, 然后在这段时间运行一个循环。编写一个简单的程序测试该函数。

5. 编写一个函数接受这些参数: 内含 `int` 类型元素的数组名、数组的大小和一个代表选取次数的值。该函数从数组中随机选择指定数量的元素, 并打印它们。每个元素只能选择一次(模拟抽奖数字或挑选陪审团成员)。另外, 如果你的实现有 `time()`(第 12 章讨论过)或类似的函数, 可在 `srand()` 中使用这个函数的输出来初始化随机数生成器 `rand()`。编写一个简单的程序测试该函数。
6. 修改程序清单 16.17, 使用 `struct names` 元素(在程序清单 16.17 后面的讨论中定义过), 而不是 `double` 类型的数组。使用较少的元素, 并用选定的名字显式初始化数组。
7. 下面是使用变参函数的一个程序段:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
void show_array(const double ar[], int n);
double * new_d_array(int n, ...);

int main()
{
    double * p1;
    double * p2;

    p1 = new_d_array(5, 1.2, 2.3, 3.4, 4.5, 5.6);
    p2 = new_d_array(4, 100.0, 20.00, 8.08, -1890.0);
```

```
show_array(p1, 5);
show_array(p2, 4);
free(p1);
free(p2);

return 0;
}
```

`new_d_array()` 函数接受一个 `int` 类型的参数和 `double` 类型的参数。该函数返回一个指针，指向由 `malloc()` 分配的内存块。`int` 类型的参数指定了动态数组中的元素个数，`double` 类型的值用于初始化元素(第 1 个值赋给第 1 个元素，以此类推)。编写 `show_array()` 和 `new_d_array()` 函数的代码，完成这个程序。

