

# 高级数据表示

本章介绍以下内容：

- 函数：进一步学习 `malloc()`
- 使用 C 表示不同类型的数据
- 新的算法，从概念上增强开发程序的能力
- 抽象数据类型（ADT）

学习计算机语言和学习音乐、木工或工程学一样。首先，要学会使用工具：学习如何演奏音阶、如何使用锤子等，然后解决各种问题，如降落、滑行以及平衡物体之类。到目前为止，读者一直在本书中学习和练习各种编程技能，如创建变量、结构、函数等。然而，如果想提高到更高层次时，工具是次要的，真正的挑战是设计和创建一个项目。本章将重点介绍这个更高的层次，教会读者如何把项目看作一个整体。本章涉及的内容可能比较难，但是这些内容非常有价值，将帮助读者从编程新手成长为老手。

我们先从程序设计的关键部分，即程序表示数据的方式开始。通常，程序开发最重要的部分是找到程序中表示数据的好方法。正确地表示数据可以更容易地编写程序其余部分。到目前为止，读者应该很熟悉 C 的内置类型：简单变量、数组、指针、结构和联合。

然而，找出正确的数据表示不仅仅是选择一种数据类型，还要考虑必须进行哪些操作。也就是说，必须确定如何储存数据，并且为数据类型定义有效的操作。例如，C 实现通常把 `int` 类型和指针类型都储存为整数，但是这两种类型的有效操作不相同。例如，两个整数可以相乘，但是两个指针不能相乘；可以用\* 运算符解引用指针，但是对整数这样做毫无意义。C 语言为它的基本类型都定义了有效的操作。但是，当你要设计数据表示的方案时，你可能需要自己定义有效操作。在 C 语言中，可以把所需的操作设计成 C 函数来表示。简而言之，设计一种数据类型包括设计如何储存该数据类型和设计一系列管理该数据的函数。

本章还会介绍一些算法 (*algorithm*)，即操控数据的方法。作为一名程序员，应该掌握这些可以反复解决类似问题的处理方法。

本章将进一步研究设计数据类型的过程，这是一个把算法和数据表示相匹配的过程。期间会用到一些常见的数据形式，如队列、列表和二叉树。

本章还将介绍抽象数据类型（ADT）的概念。抽象数据类型以面向问题而不是面向语言的方式，把解决问题的方法和数据表示结合起来。设计一个 ADT 后，可以在不同的环境中复用。理解 ADT 可以为将来学习面向对象程序设计（OOP）以及 C++ 语言做好准备。

## 17.1 研究数据表示

我们先从数据开始。假设要创建一个地址簿程序。应该使用什么数据形式储存信息？由于储存的每一项都包含多种信息，用结构来表示每一项很合适。如何表示多个项？是否用标准的结构数组？还是动态数组？还是一些其他形式？各项是否按字母顺序排列？是否要按照邮政编码（或地区编码）查找各项？需要

执行的行为将影响如何储存信息？简而言之，在开始编写代码之前，要在程序设计方面做很多决定。

如何表示储存在内存中的位图图像？位图图像中的每个像素在屏幕上都单独设置。在以前黑白屏的年代，可以使用一个计算机位（1 或 0）来表示一个像素点（开或闭），因此称之为位图。对于彩色显示器而言，如果 8 位表示一个像素，可以得到 256 种颜色。现在行业标准已发展到 65536 色（每像素 16 位）、16777216 色（每像素 24 位）、2147483 色（每像素 32 位），甚至更多。如果有 32 位色，且显示器有  $2560 \times 1440$  的分辨率，则需要将近 1.18 亿位（14M）来表示一个屏幕的位图图像。是用这种方法表示，还是开发一种压缩信息的方法？是有损压缩（丢失相对次要的数据）还是无损压缩（没有丢失数据）？再次提醒读者注意，在开始编写代码之前，需要做很多程序设计方面的决定。

我们来处理一个数据表示的示例。假设要编写一个程序，让用户输入一年内看过的所有电影（包括 DVD 和蓝光光碟）。要储存每部影片的各种信息，如片名、发行年份、导演、主演、片长、影片的种类（喜剧、科幻、爱情等）、评级等。建议使用一个结构储存每部电影，一个数组储存一年内看过的电影。为简单起见，我们规定结构中只有两个成员：片名和评级（0~10）。程序清单 17.1 演示了一个基本的实现。

程序清单 17.1 films1.c 程序

```
/* films1.c -- 使用一个结构数组 */
#include <stdio.h>
#include <string.h>
#define TSIZE      45 /* 储存片名的数组大小 */
#define FMAX       5   /* 影片的最大数量 */

struct film {
    char title[TSIZE];
    int rating;
};

char * s_gets(char str[], int lim);
int main(void)
{
    struct film movies[FMAX];
    int i = 0;
    int j;

    puts("Enter first movie title:");
    while (i < FMAX && s_gets(movies[i].title, TSIZE) != NULL &&
           movies[i].title[0] != '\0')
    {
        puts("Enter your rating <0-10>:");
        scanf("%d", &movies[i++].rating);
        while (getchar() != '\n')
            continue;
        puts("Enter next movie title (empty line to stop):");
    }
    if (i == 0)
        printf("No data entered. ");
    else
        printf("Here is the movie list:\n");

    for (j = 0; j < i; j++)
        printf("Movie: %s Rating: %d\n", movies[j].title, movies[j].rating);
    printf("Bye!\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; // 处理剩余输入行
    }
    return ret_val;
}

```

该程序创建了一个结构数组，然后把用户输入的数据储存在数组中。直到数组已满（用 FMAX 进行判断）或者到达文件结尾（用 NULL 进行判断），或者用户在首行按下 Enter 键（用'\0'进行判断），输入才会终止。

这样设计程序有点问题。首先，该程序很可能会浪费许多空间，因为大部分的片名都不会超过 40 个字符。但是，有些片名的确很长，如 *The Discreet Charm of the Bourgeoisie* 和 *Won Ton Ton, The Dog Who Saved Hollywood*。其次，许多人会觉得每年 5 部电影的限制太严格了。当然，也可以放宽这个限制，但是，要多大才合适？有些人每年可以看 500 部电影，因此可以把 FMAX 改为 500。但是，对有些人而言，这可能仍然不够，而对有些人而言一年根本看不了这么多部电影，这样就浪费了大量的内存。另外，一些编译器对自动存储类别变量（如 movies）可用的内存数量设置了一个默认的限制，如此大型的数组可能会超过默认设置的值。可以把数组声明为静态或外部数组，或者设置编译器使用更大的栈来解决这个问题。但是，这样做并不能根本解决问题。

该程序真正的问题是，数据表示太不灵活。程序在编译时确定所需内存量，其实在运行时确定会更好。要解决这个问题，应该使用动态内存分配来表示数据。可以这样做：

```

#define TSIZE 45 /* 储存片名的数组大小 */
struct film {
    char title[TSIZE];
    int rating;
};

...
int n, i;
struct film * movies; /* 指向结构的指针 */

...
printf("Enter the maximum number of movies you'll enter:\n");
scanf("%d", &n);
movies = (struct film *) malloc(n * sizeof(struct film));

```

第 12 章介绍过，可以像使用数组名那样使用指针 movies。

while (i < FMAX && s\_gets(movies[i].title, TSIZE) != NULL && movies[i].title[0] != '\0')  
使用 malloc()，可以推迟到程序运行时才确定数组中的元素数量。所以，如果只需要 20 个元素，程

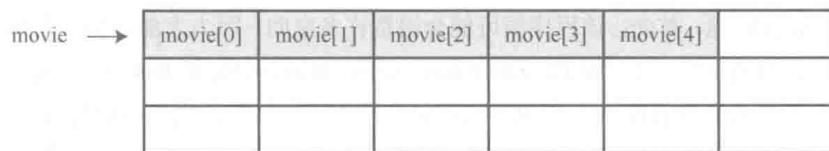
序就不必分配存放 500 个元素的空间。但是，这样做的前提是，用户要为元素个数提供正确的值。

## 17.2 从数组到链表

理想的情况是，用户可以不确定地添加数据（或者不断添加数据直到用完内存量），而不是先指定要输入多少项，也不用让程序分配多余的空间。这可以通过在输入每一项后调用 `malloc()` 分配正好能储存该项的空间。如果用户输入 3 部影片，程序就调用 `malloc()` 3 次；如果用户输入 300 部影片，程序就调用 `malloc()` 300 次。

不过，我们又制造了另一个麻烦。比较一下，一种方法是调用 `malloc()` 一次，为 300 个 `filem` 结构请求分配足够的空间；另一种方法是调用 `malloc()` 300 次，分别为每个 `file` 结构请求分配足够的空间。前者分配的是连续的内存块，只需要一个单独的指向 `struct` 变量 (`film`) 的指针，该指针指向已分配块中的第 1 个结构。简单的数组表示法让指针访问块中的每个结构，如前面代码段所示。第 2 种方法的问题是，无法保证每次调用 `malloc()` 都能分配到连续的内存块。这意味着结构不一定被连续储存（见图 17.1）。因此，与第 1 种方法储存一个指向 300 个结构块的指针相比，你需要储存 300 个指针，每个指针指向一个单独储存的结构。

```
struct film * movie;
movie = (struct film *) malloc(5*sizeof(struct film));
```



```
int i;
struct film * movies[s];
for (i = 0; i < s; i++)
    movies[i] = (struct films *) malloc(sizeof(struct films));
```

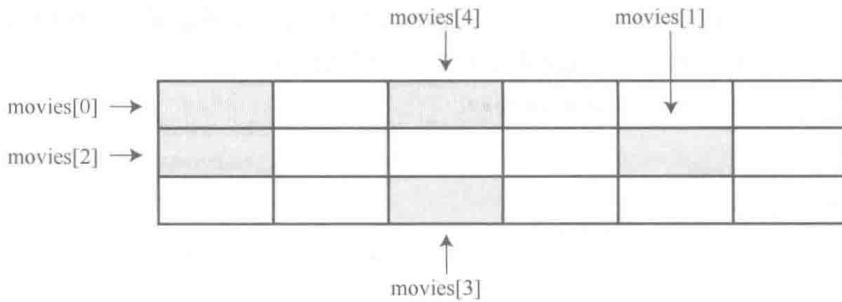


图 17.1 一块内存中分配结构和单独分配结构

一种解决方法是创建一个大型的指针数组，并在分配新结构时逐个给这些指针赋值，但是我们不打算使用这种方法：

```
#define TSIZE 45 /* 储存片名的数组大小 */
#define FMAX 500 /* 影片的最大数量 */
struct film {
    char title[TSIZE];
```

```

    int rating;
};

...
struct film * movies[FMAX]; /* 结构指针数组 */
int i;
...
movies[i] = (struct film *) malloc (sizeof (struct film));

```

如果用不完 500 个指针，这种方法节约了大量的内存，因为内含 500 个指针的数组比内含 500 个结构的数组所占的内存少得多。尽管如此，如果用不到 500 个指针，还是浪费了不少空间。而且，这样还是有 500 个结构的限制。

还有一种更好的方法。每次使用 `malloc()` 为新结构分配空间时，也为新指针分配空间。但是，还得需要另一个指针来跟踪新分配的指针，用于跟踪新指针的指针本身，也需要一个指针来跟踪，以此类推。要重新定义结构才能解决这个潜在的问题，即每个结构中包含指向 `next` 结构的指针。然后，当创建新结构时，可以把该结构的地址储存在上一个结构中。简而言之，可以这样定义 `film` 结构：

```

#define TSIZE 45 /* 储存片名的数组大小 */
struct film {
    char title[TSIZE];
    int rating;
    struct film * next;
};

```

虽然结构不能含有与本身类型相同的结构，但是可以含有指向同类型结构的指针。这种定义是定义链表（*linked list*）的基础，链表中的每一项都包含着在何处能找到下一项的信息。

在学习链表的代码之前，我们先从概念上理解一个链表。假设用户输入的片名是 `Modern Times`，等级为 10。程序将为 `film` 类型的结构分配空间，把字符串 `Modern Times` 拷贝到结构中的 `title` 成员中，然后设置 `rating` 成员为 10。为了表明该结构后面没有其他结构，程序要把 `next` 成员指针设置为 `NULL`（`NULL` 是一个定义在 `stdio.h` 头文件中的符号常量，表示空指针）。当然，还需要一个单独的指针储存第 1 个结构的地址，该指针被称为头指针（*head pointer*）。头指针指向链表中的第 1 项。图 17.2 演示了这种结构（为节约图片空间，压缩了 `title` 成员中的空白）。

```

#define TSIZE 45
struct film {
    char title[TSIZE];
    int rating;
    struct film * next;
};
struct film * head;

```

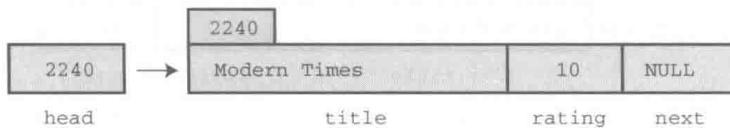


图 17.2 链表中的第 1 个项

现在，假设用户输入第 2 部电影及其评级，如 `Midnight in Paris` 和 8。程序为第 2 个 `film` 类型结构分配空间，把新结构的地址储存在第 1 个结构的 `next` 成员中（擦写了之前储存在该成员中的 `NULL`），这样链表中第 1 个结构中的 `next` 指针指向第 2 个结构。然后程序把 `Midnight in Paris` 和 8 拷贝到新结构中，并把第 2 个结构中的 `next` 成员设置为 `NULL`，表明该结构是链表中的最后一个结构。图 17.3 演示了这两个项。

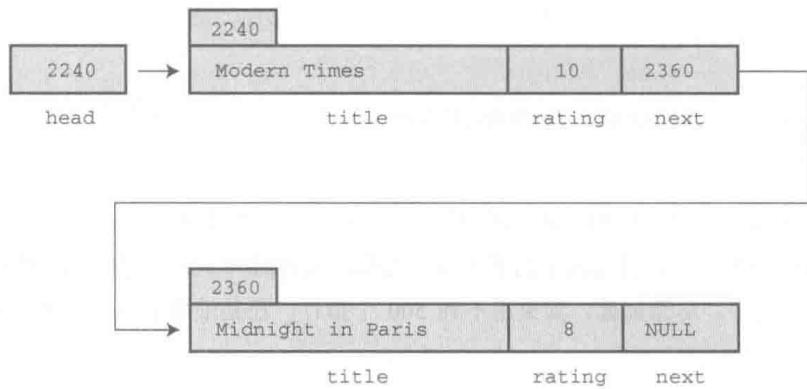


图 17.3 链表中的两个项

每加入一部新电影，就以相同的方式来处理。新结构的地址将储存在上一个结构中，新信息储存在新结构中，而且新结构中的 next 成员设置为 NULL。从而建立起如图 17.4 所示的链表。

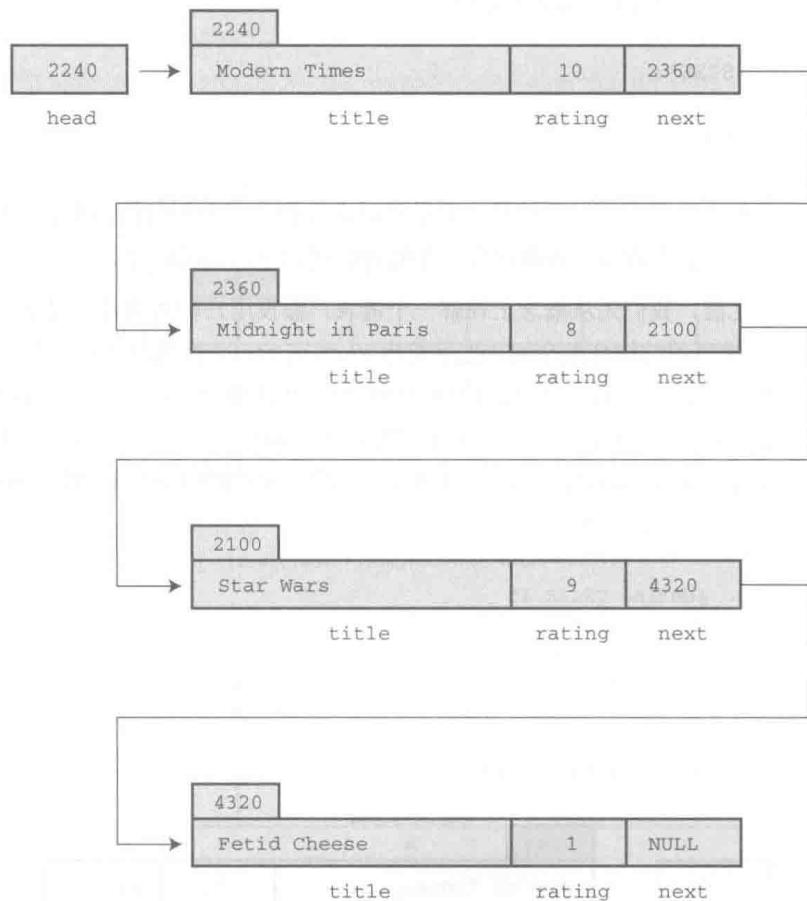


图 17.4 链表中的多个项

假设要显示这个链表，每显示一项，就可以根据该项中已储存的地址来定位下一个待显示的项。然而，这种方案能正常运行，还需要一个指针储存链表中第 1 项的地址，因为链表中没有其他项储存该项的地址。此时，头指针就派上了用场。

## 17.2.1 使用链表

从概念上了解了链表的工作原理，接着我们来实现它。程序清单 17.2 修改了程序清单 17.1，用链表而

不是数组来储存电影信息。

程序清单 17.2 films2.c 程序

```
/* films2.c -- 使用结构链表 */
#include <stdio.h>
#include <stdlib.h>           /* 提供 malloc() 原型 */
#include <string.h>            /* 提供 strcpy() 原型 */
#define TSIZE    45             /* 储存片名的数组大小 */

struct film {
    char title[TSIZE];
    int rating;
    struct film * next;      /* 指向链表中的下一个结构 */
};

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

int main(void)
{
    struct film * head = NULL;
    struct film * prev, *current;
    char input[TSIZE];

/* 收集并储存信息 */
    puts("Enter first movie title:");
    while (s_gets(input, TSIZE) != NULL && input[0] != '\0')
    {
        current = (struct film *) malloc(sizeof(struct film));
        if (head == NULL)          /* 第 1 个结构 */
            head = current;
        else                        /* 后续的结构 */
            prev->next = current;
        current->next = NULL;
        strcpy(current->title, input);
        puts("Enter your rating <0-10>:");
        scanf("%d", &current->rating);
        while (getchar() != '\n')
            continue;
        puts("Enter next movie title (empty line to stop):");
        prev = current;
    }

/* 显示电影列表 */
    if (head == NULL)
        printf("No data entered. ");
    else
        printf("Here is the movie list:\n");
    current = head;
    while (current != NULL)
    {
        printf("Movie: %s Rating: %d\n",
               current->title, current->rating);
        current = current->next;
    }
}
```

```

/* 完成任务，释放已分配的内存 */
current = head;
while (current != NULL)
{
    current = head;
    head = current->next;
    free(current);
}
printf("Bye!\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; // 处理剩余输入行
    }
    return ret_val;
}

```

该程序用链表执行两个任务。第 1 个任务是，构造一个链表，把用户输入的数据储存在链表中。第 2 个任务是，显示链表。显示链表的任务比较简单，所以我们先来讨论它。

## 1. 显示链表

显示链表从设置一个指向第 1 个结构的指针（名为 `current`）开始。由于头指针（名为 `head`）已经指向链表中的第 1 个结构，所以可以用下面的代码来完成：

```
current = head;
```

然后，可以使用指针表示法访问结构的成员：

```
printf("Movie: %s Rating: %d\n", current->title, current->rating);
```

下一步是根据储存在该结构中 `next` 成员中的信息，重新设置 `current` 指针指向链表中的下一个结构。代码如下：

```
current = current->next;
```

完成这些之后，再重复整个过程。当显示到链表中最后一个项时，`current` 将被设置为 `NULL`，因为这是链表最后一个结构中 `next` 成员的值。

```
while (current != NULL)
{
    printf("Movie: %s Rating: %d\n", current->title, current->rating);
    current = current->next;
}
```

遍历链表时，为何不直接使用 head 指针，而要重新创建一个新指针（current）？因为如果使用 head 会改变 head 中的值，程序就找不到链表的开始处。

## 2. 创建链表

创建链表涉及下面 3 步：

- (1) 使用 malloc() 为结构分配足够的空间；
- (2) 储存结构的地址；
- (3) 把当前信息拷贝到结构中。

如无必要不用创建一个结构，所以程序使用临时存储区（input 数组）获取用户输入的电影名。如果用户通过键盘模拟 EOF 或输入一行空行，将退出下面的循环：

```
while (s_gets(input, TSIZE) != NULL && input[0] != '\0')
```

如果用户进行输入，程序就分配一个结构的空间，并将其地址赋给指针变量 current：

```
current = (struct film *) malloc(sizeof(struct film));
```

链表中第 1 个结构的地址应储存在指针变量 head 中。随后每个结构的地址应储存在其前一个结构的 next 成员中。因此，程序要知道它处理的是否是第 1 个结构。最简单的方法是在程序开始时，把 head 指针初始化为 NULL。然后，程序可以使用 head 的值进行判断：

```
if (head == NULL) /* 第 1 个结构*/
    head = current;
else /* subsequent structures */
    prev->next = current;
```

在上面的代码中，指针 prev 指向上一次分配的结构。

接下来，必须为结构成员设置合适的值。尤其是，把 next 成员设置为 NULL，表明当前结构是链表的最后一个结构。还要把 input 数组中的电影名拷贝到 title 成员中，而且要给 rating 成员提供一个值。如下代码所示：

```
current->next = NULL;
strcpy(current->title, input);
puts("Enter your rating <0-10>:");
scanf("%d", &current->rating);
```

由于 s\_gets() 限制了只能输入 TSIZE-1 个字符，所以用 strcpy() 函数把 input 数组中的字符串拷贝到 title 成员很安全。

最后，要为下一次输入做好准备。尤其是，要设置 prev 指向当前结构。因为在用户输入下一部电影且程序为新结构分配空间后，当前结构将成为新结构的上一个结构，所以程序在循环末尾这样设置该指针：

```
prev = current;
```

程序是否能正常运行？下面是该程序的一个运行示例：

```
Enter first movie title:
Spirited Away
Enter your rating <0-10>:
9
Enter next movie title (empty line to stop):
The Duelists
Enter your rating <0-10>:
8
Enter next movie title (empty line to stop):
Devil Dog: The Mound of Hound
Enter your rating <0-10>:
```

```

1
Enter next movie title (empty line to stop):

Here is the movie list:
Movie: Spirited Away Rating: 9
Movie: The Duelists Rating: 8
Movie: Devil Dog: The Mound of Hound Rating: 1
Bye!

```

### 3. 释放链表

在许多环境中，程序结束时都会自动释放 `malloc()` 分配的内存。但是，最好还是成对调用 `malloc()` 和 `free()`。因此，程序在清理内存时为每个已分配的结构都调用了 `free()` 函数：

```

current = head;
while (current != NULL)
{
    current = head;
    head = current->next;
    free(current);
}

```

#### 17.2.2 反思

`films2.c` 程序还有些不足。例如，程序没有检查 `malloc()` 是否成功请求到内存，也无法删除链表中的项。这些不足可以弥补。例如，添加代码检查 `malloc()` 的返回值是否是 `NULL`（返回 `NULL` 说明未获得所需内存）。如果程序要删除链表中的项，还要编写更多的代码。

这种用特定方法解决特定问题，并且在需要时才添加相关功能的编程方式通常不是最好的解决方案。另一方面，通常都无法预料程序要完成的所有任务。随着编程项目越来越大，一个程序员或编程团队事先计划好一切模式，越来越不现实。很多成功的大型程序都是由成功的小型程序逐步发展而来。

如果要修改程序，首先应该强调最初的设计，并简化其他细节。程序清单 17.2 中的程序示例没有遵循这个原则，它把概念模型和代码细节混在一起。例如，该程序的概念模型是在一个链表中添加项，但是程序却把一些细节（如，`malloc()` 和 `current->next` 指针）放在最明显的位置，没有突出接口。如果程序能以某种方式强调给链表添加项，并隐藏具体的处理细节（如调用内存管理函数和设置指针）会更好。把用户接口和代码细节分开的程序，更容易理解和更新。学习下面的内容就可以实现这些目标。

## 17.3 抽象数据类型 (ADT)

在编程时，应该根据编程问题匹配合适的数据类型。例如，用 `int` 类型代表你有多少双鞋，用 `float` 或 `double` 类型代表每双鞋的价格。在前面的电影示例中，数据构成了链表，每个链表项由电影名（C 字符串）和评级（一个 `int` 类型值）。C 中没有与之匹配的基本类型，所以我们定义了一个结构代表单独的项，然后设计了一些方法把一系列结构构成一个链表。本质上，我们使用 C 语言的功能设计了一种符合程序要求的新数据类型。但是，我们的做法并不系统。现在，我们用更系统的方法来定义数据类型。

什么是类型？类型特指两类信息：属性和操作。例如，`int` 类型的属性是它代表一个整数值，因此它共享整数的属性。允许对 `int` 类型进行算术操作是：改变 `int` 类型值的符号、两个 `int` 类型值相加、相减、相乘、相除、求模。当声明一个 `int` 类型的变量时，就表明了只能对该变量进行这些操作。

## 注意 整数属性

C 的 int 类型背后是一个更抽象的整数概念。数学家已经用正式的抽象方式定义了整数的属性。例如，假设 N 和 M 是整数，那么  $N+M=M+N$ ；假设 S、Q 也是整数，如果  $N+M=S$ ，而且  $N+Q=S$ ，那么  $M=Q$ 。可以认为数学家提供了整数的抽象概念，而 C 则实现了这一抽象概念。注意，实现整数的算术运算是表示整数必不可少的部分。如果只是储存值，并未在算术表达式中使用，int 类型就没那么有用了。还要注意的是，C 并未很好地实现整数。例如，整数是无穷大的数，但是 2 字节的 int 类型只能表示 65536 个整数。因此，不要混淆抽象概念和具体的实现。

假设要定义一个新的数据类型。首先，必须提供储存数据的方法，例如设计一个结构。其次，必须提供操控数据的方法。例如，考虑 `films2.c` 程序（程序清单 17.2）。该程序用链接的结构来储存信息，而且通过代码实现了如何添加和显示信息。尽管如此，该程序并未清楚地表明正在创建一个新类型。我们应该怎么做？

计算机科学领域已开发了一种定义新类型的好方法，用 3 个步骤完成从抽象到具体的过程。

1. 提供类型属性和相关操作的抽象描述。这些描述既不能依赖特定的实现，也不能依赖特定的编程语言。这种正式的抽象描述被称为抽象数据类型 (ADT)。

2. 开发一个实现 ADT 的编程接口。也就是说，指明如何储存数据和执行所需操作的函数。例如在 C 中，可以提供结构定义和操控该结构的函数原型。这些作用于用户定义类型的函数相当于作用于 C 基本类型的内置运算符。需要使用该新类型的程序员可以使用这个接口进行编程。

3. 编写代码实现接口。这一步至关重要，但是使用该新类型的程序员无需了解具体的实现细节。

我们再次以前面的电影项目为例来熟悉这个过程，并用新方法重新完成这个示例。

### 17.3.1 建立抽象

从根本上讲，电影项目所需的是一个项链表。每一项包含电影名和评级。你所需的操作是把新项添加到链表的末尾和显示链表中的内容。我们把需要处理这些需求的抽象类型叫作链表。链表具有哪些属性？首先，链表应该能储存一系列的项。也就是说，链表能储存多个项，而且这些项以某种方式排列，这样才能描述链表的第一项、第二项或最后一项。其次，链表类型应该提供一些操作，如在链表中添加新项。下面是链表的一些有用的操作：

- 初始化一个空链表；
- 在链表末尾添加一个新项；
- 确定链表是否为空；
- 确定链表是否已满；
- 确定链表中的项数；
- 访问链表中的每一项执行某些操作，如显示该项。

对该电影项目而言，暂时不需要其他操作。但是一般的链表还应包含以下操作：

- 在链表的任意位置插入一个项；
- 移除链表中的一个项；
- 在链表中检索一个项（不改变链表）；

- 用另一个项替换链表中的一个项;
- 在链表中搜索一个项。

非正式但抽象的链表定义是：链表是一个能储存一系列项目且可以对其进行所需操作的数据对象。该定义既未说明链表中可以储存什么项，也未指定是用数组、结构还是其他数据形式来储存项，而且并未规定用什么方法来实现操作（如，查找链表中元素的个数）。这些细节都留给实现完成。

为了让示例尽量简单，我们采用一种简化的链表作为抽象数据类型。它只包含电影项目中的所需属性。该类型总结如下：

|       |   |
|-------|---|
| 类型名:  | 简单链表  |
| 类型属性: | 可以储存一系列项  |
| 类型操作: | 初始化链表为空<br>确定链表为空<br>确定链表已满<br>确定链表中的项数<br>在链表末尾添加项<br>遍历链表，处理链表中的项<br>清空链表 |

下一步是为开发简单链表 ADT 开发一个 C 接口。

### 17.3.2 建立接口

这个简单链表的接口有两个部分。第 1 部分是描述如何表示数据，第 2 部分是描述实现 ADT 操作的函数。例如，要设计在链表中添加项的函数和报告链表中项数的函数。接口设计应尽量与 ADT 的描述保持一致。因此，应该用某种通用的 Item 类型而不是一些特殊类型，如 int 或 struct film。可以用 C 的 typedef 功能来定义所需的 Item 类型：

```
#define TSIZE 45 /* 储存电影名的数组大小 */
struct film
{
    char title[TSIZE];
    int rating;
};
typedef struct film Item;
```

然后，就可以在定义的其余部分使用 Item 类型。如果以后需要其他数据形式的链表，可以重新定义 Item 类型，不必更改其余的接口定义。

定义了 Item 之后，现在必须确定如何储存这种类型的项。实际上这一步属于实现步骤，但是现在决定好可以让示例更简单些。在 films2.c 程序中用链接的结构处理得很好，所以，我们在这里也采用相同的方法：

```
typedef struct node
{
    Item item;
    struct node * next;
} Node;
typedef Node * List;
```

在链表的实现中，每一个链节叫作节点 (node)。每个节点包含形成链表内容的信息和指向下一个节点

的指针。为了强调这个术语，我们把 node 作为节点结构的标记名，并使用 `typedef` 把 Node 作为 `struct node` 结构的类型名。最后，为了管理链表，还需要一个指向链表开始处的指针，我们使用 `typedef` 把 List 作为该类型的指针名。因此，下面的声明：

```
List movies;
```

创建了该链表所需类型的指针 `movies`。

这是否是定义 List 类型的唯一方法？不是。例如，还可以添加一个变量记录项数：

```
typedef struct list
{
    Node * head; /* 指向链表头的指针 */
    int size;     /* 链表中的项数 */
} List;          /* List 的另一种定义 */
```

可以像稍后的程序示例中那样，添加第 2 个指针储存链表的末尾。现在，我们还是使用 List 类型的第一种定义。这里要着重理解下面的声明创建了一个链表，而不一个指向节点的指针或一个结构：

```
List movies;
```

`movies` 代表的确切数据应该是接口层次不可见的实现细节。

例如，程序启动后应把头指针初始化为 `NULL`。但是，不要使用下面这样的代码：

```
movies = NULL;
```

为什么？因为稍后你会发现 List 类型的结构实现更好，所以应这样初始化：

```
movies.next = NULL;
movies.size = 0;
```

使用 List 的人都不用担心这些细节，只要能使用下面的代码就行：

```
InitializeList(movies);
```

使用该类型的程序员只需知道用 `InitializeList()` 函数来初始化链表，不必了解 List 类型变量的实现细节。这是数据隐藏的一个示例，数据隐藏是一种从编程的更高层次隐藏数据表示细节的艺术。

为了指导用户使用，可以在函数原型前面提供以下注释：

```
/* 操作：初始化一个链表           */
/* 前提条件：plist 指向一个链表 */
/* 后置条件：该链表初始化为空       */
void InitializeList(List * plist);
```

这里要注意 3 点。第 1，注释中的“前提条件”(*precondition*) 是调用该函数前应具备的条件。例如，需要一个待初始化的链表。第 2，注释中的“后置条件”(*postcondition*) 是执行完该函数后的情况。第 3，该函数的参数是一个指向链表的指针，而不是一个链表。所以应该这样调用该函数：

```
InitializeList(&movies);
```

由于按值传递参数，所以该函数只能通过指向该变量的指针才能更改主调程序传入的变量。这里，由于语言的限制使得接口和抽象描述略有区别。

C 语言把所有类型和函数的信息集合成一个软件包的方法是：把类型定义和函数原型（包括前提条件和后置条件注释）放在一个头文件中。该文件应该提供程序员使用该类型所需的所有信息。程序清单 17.3 给出了一个简单链表类型的头文件。该程序定义了一个特定的结构作为 `Item` 类型，然后根据 `Item` 定义了 `Node`，再根据 `Node` 定义了 `List`。然后，把表示链表操作的函数设计为接受 `Item` 类型和 `List` 类型的参数。如果函数要修改一个参数，那么该参数的类型应是指向相应类型的指针，而不是该类型。在头文件中，把组成函数名的每个单词的首字母大写，以这种方式表明这些函数是接口包的一部分。另外，该文件使用第 16 章介绍的`#ifndef` 指令，防止多次包含一个文件。如果编译器不支持 C99 的 `bool` 类型，可以用下

面的代码：

```
enum bool {false, true}; /* 把 bool 定义为类型, false 和 true 是该类型的值 */
```

替换下面的头文件：

```
#include <stdbool.h> /* C99 特性 */
```

### 程序清单 17.3 list.h 接口头文件

```
/* list.h -- 简单链表类型的头文件 */
#ifndef LIST_H_
#define LIST_H_
#include <stdbool.h> /* C99 特性 */

/* 特定程序的声明 */

#define TSIZE      45 /* 储存电影名的数组大小 */
struct film
{
    char title[TSIZE];
    int rating;
};

/* 一般类型定义 */

typedef struct film Item;

typedef struct node
{
    Item item;
    struct node * next;
} Node;

typedef Node * List;

/* 函数原型 */

/* 操作：      初始化一个链表 */
/* 前提条件：  plist 指向一个链表 */
/* 后置条件：  链表初始化为空 */
void InitializeList(List * plist);

/* 操作：      确定链表是否为空定义, plist 指向一个已初始化的链表 */
/* 后置条件：  如果链表为空, 该函数返回 true; 否则返回 false */
bool ListIsEmpty(const List *plist);

/* 操作：      确定链表是否已满, plist 指向一个已初始化的链表 */
/* 后置条件：  如果链表已满, 该函数返回真; 否则返回假 */
bool ListIsFull(const List *plist);

/* 操作：      确定链表中的项数, plist 指向一个已初始化的链表 */
/* 后置条件：  该函数返回链表中的项数 */
unsigned int ListItemCount(const List *plist);
```

```

/* 操作: 在链表的末尾添加项 */
/* 前提条件: item 是一个待添加至链表的项, plist 指向一个已初始化的链表 */
/* 后置条件: 如果可以, 该函数在链表末尾添加一个项, 且返回 true; 否则返回 false */
bool AddItem(Item item, List * plist);

/* 操作: 把函数作用于链表中的每一项 */
/* plist 指向一个已初始化的链表 */
/* pfun 指向一个函数, 该函数接受一个 Item 类型的参数, 且无返回值 */
/* 后置条件: pfun 指向的函数作用于链表中的每一项一次 */
void Traverse(const List * plist, void(*pfun)(Item item));

/* 操作: 释放已分配的内存 (如果有的话) */
/* plist 指向一个已初始化的链表 */
/* 后置条件: 释放了为链表分配的所有内存, 链表设置为空 */
void EmptyTheList(List * plist);

#endif

```

只有 `InitializeList()`、`AddItem()` 和 `EmptyTheList()` 函数要修改链表，因此从技术角度看，这些函数需要一个指针参数。然而，如果某些函数接受 `List` 类型的变量作为参数，而其他函数却接受 `List` 类型的地址作为参数，用户会很困惑。因此，为了减轻用户的负担，所有的函数均使用指针参数。

头文件中的一个函数原型比其他原型复杂：

```

/* 操作: 把函数作用于链表中的每一项 */
/* plist 指向一个已初始化的链表 */
/* pfun 指向一个函数, 该函数接受一个 Item 类型的参数, 且无返回值 */
/* 后置条件: pfun 指向的函数作用于链表中的每一项一次 */
void Traverse(const List * plist, void(*pfun)(Item item));

```

参数 `pfun` 是一个指向函数的指针，它指向的函数接受 `item` 值且无返回值。第 14 章中介绍过，可以把函数指针作为参数传递给另一个函数，然后该函数就可以使用这个被指针指向的函数。例如，该例中可以让 `pfun` 指向显示链表项的函数。然后把 `Traverse()` 函数把该函数作用于链表中的每一项，显示链表中的内容。

### 17.3.3 使用接口

我们的目标是，使用这个接口编写程序，但是不必知道具体的实现细节（如，不知道函数的实现细节）。在编写具体函数之前，我们先编写电影程序的一个新版本。由于接口要使用 `List` 和 `Item` 类型，所以该程序也应使用这些类型。下面是编写该程序的一个伪代码方案。

创建一个 `List` 类型的变量。

创建一个 `Item` 类型的变量。

初始化链表为空。

当链表未满且有输入时：

把输入读取到 `Item` 类型的变量中。

在链表末尾添加项。

访问链表中的每个项并显示它们。

程序清单 17.4 中的程序按照以上伪代码来编写，其中还加入了一些错误检查。注意该程序利用了 list.h（程序清单 17.3）中描述的接口。另外，还需注意，链表中含有 showmovies() 函数的代码，它与 Traverse() 的原型一致。因此，程序可以把指针 showmovies 传递给 Traverse()，这样 Traverse() 可以把 showmovies() 函数应用于链表中的每一项（回忆一下，函数名是指向该函数的指针）。

## 程序清单 17.4 films3.c 程序

```
/* films3.c -- 使用抽象数据类型 (ADT) 风格的链表 */
/* 与 list.c 一起编译 */
#include <stdio.h>
#include <stdlib.h>      /* 提供 exit() 的原型 */
#include "list.h"         /* 定义 List、Item */
void showmovies(Item item);
char * s_gets(char * st, int n);
int main(void)
{
    List movies;
    Item temp;

    /* 初始化 */
    InitializeList(&movies);
    if (ListIsFull(&movies))
    {
        fprintf(stderr, "No memory available! Bye!\n");
        exit(1);
    }

    /* 获得用户输入并储存 */
    puts("Enter first movie title:");
    while (s_gets(temp.title, TSIZE) != NULL && temp.title[0] != '\0')
    {
        puts("Enter your rating <0-10>:");
        scanf("%d", &temp.rating);
        while (getchar() != '\n')
            continue;
        if (AddItem(temp, &movies) == false)
        {
            fprintf(stderr, "Problem allocating memory\n");
            break;
        }
        if (ListIsFull(&movies))
        {
            puts("The list is now full.");
            break;
        }
        puts("Enter next movie title (empty line to stop):");
    }

    /* 显示 */
    if (ListIsEmpty(&movies))
        printf("No data entered. ");
    else
    {
```

```

        printf("Here is the movie list:\n");
        Traverse(&movies, showmovies);
    }
    printf("You entered %d movies.\n", ListItemCount(&movies));

    /* 清理 */
    EmptyTheList(&movies);
    printf("Bye!\n");

    return 0;
}

void showmovies(Item item)
{
    printf("Movie: %s Rating: %d\n", item.title,
           item.rating);
}

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; // 处理输入行的剩余内容
    }
    return ret_val;
}

```

#### 17.3.4 实现接口

当然，我们还是必须实现 List 接口。C 方法是把函数定义统一放在 list.c 文件中。然后，整个程序由 list.h (定义数据结构和提供用户接口的原型)、list.c (提供函数代码实现接口) 和 films3.c (把链表接口应用于特定编程问题的源代码文件) 组成。程序清单 17.5 演示了 list.c 的一种实现。要运行该程序，必须把 films3.c 和 list.c 一起编译和链接 (可以复习一下第 9 章关于编译多文件程序的内容)。list.h、list.c 和 films3.c 组成了整个程序 (见图 17.5)。

程序清单 17.5 list.c 实现文件

---

```

/* list.c -- 支持链表操作的函数 */
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

```

```

/* 局部函数原型 */
static void CopyToNode(Item item, Node * pnode);

/* 接口函数 */
/* 把链表设置为空 */
void InitializeList(List * plist)
{
    *plist = NULL;
}

/* 如果链表为空，返回 true */
bool ListIsEmpty(const List * plist)
{
    if (*plist == NULL)
        return true;
    else
        return false;
}

/* 如果链表已满，返回 true */
bool ListIsFull(const List * plist)
{
    Node * pt;
    bool full;

    pt = (Node *)malloc(sizeof(Node));
    if (pt == NULL)
        full = true;
    else
        full = false;
    free(pt);

    return full;
}

/* 返回节点的数量 */
unsigned int ListItemCount(const List * plist)
{
    unsigned int count = 0;
    Node * pnode = *plist; /* 设置链表的开始 */

    while (pnode != NULL)
    {
        ++count;
        pnode = pnode->next; /* 设置下一个节点 */
    }

    return count;
}

/* 创建储存项的节点，并将其添加至由 plist 指向的链表末尾（较慢的实现） */
bool AddItem(Item item, List * plist)
{
    Node * pnew;

```

```

Node * scan = *plist;

pnew = (Node *) malloc(sizeof(Node));
if (pnew == NULL)
    return false; /* 失败时退出函数 */

CopyToNode(item, pnew);
pnew->next = NULL;
if (scan == NULL)           /* 空链表, 所以把 */
    *plist = pnew;          /* pnew放在链表的开头 */
else
{
    while (scan->next != NULL)
        scan = scan->next; /* 找到链表的末尾 */
    scan->next = pnew;     /* 把pnew添加到链表的末尾 */
}

return true;
}

/* 访问每个节点并执行pfun指向的函数 */
void Traverse(const List * plist, void(*pfun)(Item item))
{
    Node * pnode = *plist; /* 设置链表的开始 */

    while (pnode != NULL)
    {
        (*pfun)(pnode->item); /* 把函数应用于链表中的项 */
        pnode = pnode->next; /* 前进到下一项 */
    }
}

/* 释放由malloc()分配的内存 */
/* 设置链表指针为NULL */
void EmptyTheList(List * plist)
{
    Node * psave;

    while (*plist != NULL)
    {
        psave = (*plist)->next; /* 保存下一个节点的地址 */
        free(*plist);            /* 释放当前节点 */
        *plist = psave;          /* 前进至下一个节点 */
    }
}

/* 局部函数定义 */
/* 把一个项拷贝到节点中 */
static void CopyToNode(Item item, Node * pnode)
{
    pnode->item = item; /* 拷贝结构 */
}

```

```

list.h
/*
 * list.h--简单链表类型的头文件 */
/* 特定的程序声明 */
#define TSIZE 45 /* 储存电影名的数组大小 */
struct film
{
    char title[TSIZE];
    int rating;
};

void Traverse (List l, void (* pfun)(Item item) );

```

```

list.c
/*
 * list.c--支持链表操作的函数 */
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

.

.

/* 把一个项拷贝到节点中 */
static void CopyToNode (Item item, Node * pnode)
{
    pnode->item = item; /* 拷贝结构 */
}

```

```

films3.c
/*
 * films3.c--使用抽象数据类型(ADT)风格的链表 */
#include <stdio.h>
#include <stdlib.h> /* 提供exit()的原型 */
#include "list.h"
void showmovies(Item item);

int main(void)
{
.
.
}

```

图 17.5 电影程序的 3 个部分

## 1. 程序的一些注释

`list.c` 文件有几个需要注意的地方。首先，该文件演示了什么情况下使用内部链接函数。如第 12 章所述，具有内部链接的函数只能在其声明所在的文件夹可见。在实现接口时，有时编写一个辅助函数（不作为正式接口的一部分）很方便。例如，使用 `CopyToNode()` 函数把一个 `Item` 类型的值拷贝到 `Item` 类型的变量中。由于该函数是实现的一部分，但不是接口的一部分，所以我们使用 `static` 存储类别说明符把它隐藏在 `list.c` 文件中。接下来，讨论其他函数。

`InitializeList()` 函数将链表初始化为空。在我们的实现中，这意味着把 `List` 类型的变量设置为 `NULL`。前面提到过，这要求把指向 `List` 类型变量的指针传递给该函数。

`ListIsEmpty()` 函数很简单，但是它的前提条件是，当链表为空时，链表变量被设置为 `NULL`。因此，在首次调用 `ListIsEmpty()` 函数之前初始化链表非常重要。另外，如果要扩展接口添加删除项的功能，那么当最后一个项被删除时，应该确保该删除函数重置链表为空。对链表而言，链表的大小取决于可用内存量。`ListIsFull()` 函数尝试为新项分配空间。如果分配失败，说明链表已满；如果分配成功，则必须释放刚才分配的内存供真正的项所用。

`ListItemCount()` 函数使用常用的链表算法遍历链表，同时统计链表中的项：

```

unsigned int ListItemCount(const List * plist)
{
    unsigned int count = 0;
    Node * pnode = *plist; /* 设置链表的开始 */

    while (pnode != NULL)
    {
        ++count;
        pnode = pnode->next; /* 设置下一个节点 */
    }

    return count;
}

```

AddItem() 函数是这些函数中最复杂的：

```

bool AddItem(Item item, List * plist)
{
    Node * pnew;
    Node * scan = *plist;

    pnew = (Node *) malloc(sizeof(Node));
    if (pnew == NULL)
        return false; /* 失败时退出函数 */

    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (scan == NULL) /* 空链表, 所以把 */
        *plist = pnew; /* pnew 放在链表的开头 */
    else
    {
        while (scan->next != NULL)
            scan = scan->next; /* 找到链表的末尾 */
        scan->next = pnew; /* 把 pnew 添加到链表的末尾 */
    }

    return true;
}

```

AddItem() 函数首先为新节点分配空间。如果分配成功，该函数使用 CopyToNode() 把项拷贝到新节点中。然后把该节点的 next 成员设置为 NULL。这表明该节点是链表中的最后一个节点。最后，完成创建节点并为其成员赋正确的值之后，该函数把该节点添加到链表的末尾。如果该项是添加到链表的第一个项，需要把头指针设置为指向第 1 项（记住，头指针的地址是传递给 AddItem() 函数的第 2 个参数，所以 \*plist 就是头指针的值）。否则，代码继续在链表中前进，直到发现被设置为 NULL 的 next 成员。此时，该节点就是当前的最后一个节点，所以，函数重置它的 next 成员指向新节点。

要养成良好的编程习惯，给链表添加项之前应调用 ListIsFull() 函数。但是，用户可能并未这样做，所以在 AddItem() 函数内部检查 malloc() 是否分配成功。而且，用户还可能在调用 ListIsFull() 和调用 AddItem() 函数之间做其他事情分配了内存，所以最好还是检查 malloc() 是否分配成功。

Traverse() 函数与 ListItemCount() 函数类似，不过它还把一个指针函数作用于链表中的每一项。

```

void Traverse (const List * plist, void (* pfun)(Item item) )
{
    Node * pnode = *plist; /* 设置链表的开始 */

```

```

while (pnode != NULL)
{
    (*pfun) (pnode->item); /* 把函数应用于该项 */
    pnode = pnode->next;   /* 前进至下一个项 */
}
}

```

pnode->item 代表储存在节点中的数据，pnode->next 标识链表中的下一个节点。如下函数调用：  
 Traverse(movies, showmovies);

把 showmovies() 函数应用于链表中的每一项。

最后，EmptyTheList() 函数释放了之前 malloc() 分配的内存：

```

void EmptyTheList(List * plist)
{
    Node * psave;

    while (*plist != NULL)
    {
        psave = (*plist)->next;      /* 保存下一个节点的地址 */
        free(*plist);                /* 释放当前节点 */
        *plist = psave;              /* 前进至下一个节点 */
    }
}

```

该函数的实现通过把 List 类型的变量设置为 NULL 来表明一个空链表。因此，要把 List 类型变量的地址传递给该函数，以便函数重置。由于 List 已经是一个指针，所以 plist 是一个指向指针的指针。因此，在上面的代码中，\*plist 是指向 Node 的指针。当到达链表末尾时，\*plist 为 NULL，表明原始的实际参数现在被设置为 NULL。

代码中要保存下一节点的地址，因为原则上调用了 free() 会使当前节点（即\*plist 指向的节点）的内容不可用。

### 提示 const 的限制

多个处理链表的函数都把 const List \* plist 作为形参，表明这些函数不会更改链表。这里，const 确实提供了一些保护。它防止了\*plist（即 plist 所指向的量）被修改。在该程序中，plist 指向 movies，所以 const 防止了这些函数修改 movies。因此，在 ListItemCount() 中，不允许有类似下面的代码：

```
*plist = (*plist)->next; // 如果*plist 是 const，不允许这样做
```

因为改变\*plist 就改变了 movies，将导致程序无法跟踪数据。然而，\*plist 和 movies 都被看作是 const 并不意味着\*plist 或 movies 指向的数据是 const。例如，可以编写下面的代码：

```
(*plist)->item.rating = 3; // 即使*plist 是 const，也可以这样做
```

因为上面的代码并未改变\*plist，它改变的是\*plist 指向的数据。由此可见，不要指望 const 能捕获到意外修改数据的程序错误。

## 2. 考虑你要做的

现在花点时间来评估 ADT 方法做了什么。首先，比较程序清单 17.2 和程序清单 17.4。这两个程序都使用相同的内存分配方法（动态分配链接的结构）解决电影链表的问题，但是程序清单 17.2 暴露了所有的编

程细节，把 `malloc()` 和 `prev->next` 这样的代码都公之于众。而程序清单 17.4 隐藏了这些细节，并用与任务直接相关的方式表达程序。也就是说，该程序讨论的是创建链表和向链表中添加项，而不是调用内存函数或重置指针。简而言之，程序清单 17.4 是根据待解决的问题来表达程序，而不是根据解决问题所需的具体工具来表达程序。ADT 版本可读性更高，而且针对的是最终的用户所关心的问题。

其次，`list.h` 和 `list.c` 文件一起组成了可复用的资源。如果需要另一个简单的链表，也可以使用这些文件。假设你需要储存亲戚的一些信息：姓名、关系、地址和电话号码，那么先要在 `list.h` 文件中重新定义 `Item` 类型：

```
typedef struct itemtag
{
    char fname[14];
    char lname [24];
    char relationship[36];
    char address [60];
    char phonenum[20];
} Item;
```

然后……只需要做这些就行了。因为所有处理简单链表的函数都与 `Item` 类型有关。根据不同的情况，有时还要重新定义 `CopyToNode()` 函数。例如，当项是一个数组时，就不能通过赋值来拷贝。

另一个要点是，用户接口是根据抽象链表操作定义的，不是根据某些特定的数据表示和算法来定义。这样，不用重写最后的程序就能随意修改实现。例如，当前使用的 `AddItem()` 函数效率不高，因为它总是从链表第 1 个项开始，然后搜索至链表末尾。可以通过保存链表结尾处的地址来解决这个问题。例如，可以这样重新定义 `List` 类型：

```
typedef struct list
{
    Node * head; /* 指向链表的开头 */
    Node * end; /* 指向链表的末尾 */
} List;
```

当然，还要根据新的定义重写处理链表的函数，但是不用修改程序清单 17.4 中的内容。对大型编程项目而言，这种把实现和最终接口隔离的做法相当有用。这称为数据隐藏，因为对终端用户隐藏了数据表示的细节。

注意，这种特殊的 ADT 甚至不要求以链表的方式实现简单链表。下面是另一种方法：

```
#define MAXSIZE 100
typedef struct list
{
    Item entries[MAXSIZE]; /* 项数组 */
    int items; /* 其中的项数 */
} List;
```

这样做也需要重写 `list.c` 文件，但是使用 `list` 的程序不用修改。

最后，考虑这种方法给程序开发过程带来了哪些好处。如果程序运行出现问题，可以把问题定位到具体的函数上。如果想用更好的方法来完成某个任务（如，添加项），只需重写相应的函数即可。如果需要新功能，可以添加一个新的函数。如果觉得数组或双向链表更好，可以重写实现的代码，不用修改使用实现的程序。

## 17.4 队列 ADT

在 C 语言中使用抽象数据类型方法编程包含以下 3 个步骤。

1. 以抽象、通用的方式描述一个类型，包括该类型的操作。

2. 设计一个函数接口表示这个新类型。
3. 编写具体代码实现这个接口。

前面已经把这种方法应用到简单链表中。现在，把这种方法应用于更复杂的数据类型：队列。

### 17.4.1 定义队列抽象数据类型

队列（queue）是具有两个特殊属性的链表。第一，新项只能添加到链表的末尾。从这方面看，队列与简单链表类似。第二，只能从链表的开头移除项。可以把队列想象成排队买票的人。你从队尾加入队列，买完票后从队首离开。队列是一种“先进先出”（first in, first out，缩写为 FIFO）的数据形式，就像排队买票的队伍一样（前提是没有人插队）。接下来，我们建立一个非正式的抽象定义：

|       |  |
|-------|--|
| 类型名：  | 队列   |
| 类型属性： | 可以储存一系列项   |
| 类型操作： | 初始化队列为空<br>确定队列为空<br>确定队列已满<br>确定队列中的项数<br>在队列末尾添加项<br>在队列开头删除或恢复项<br>清空队列 |

### 17.4.2 定义一个接口

接口定义放在 queue.h 文件中。我们使用 C 的 `typedef` 工具创建两个类型名：`Item` 和 `Queue`。相应结构的具体实现应该是 queue.h 文件的一部分，但是从概念上来看，应该在实现阶段才设计结构。现在，只是假定已经定义了这些类型，着重考虑函数的原型。

首先，考虑初始化。这涉及改变 `Queue` 类型，所以该函数应该以 `Queue` 的地址作为参数：

```
void InitializeQueue (Queue * pq);
```

接下来，确定队列是否为空或已满的函数应返回真或假值。这里，假设 C99 的 `stdbool.h` 头文件可用。如果该文件不可用，可以使用 `int` 类型或自己定义 `bool` 类型。由于该函数不更改队列，所以接受 `Queue` 类型的参数。但是，传递 `Queue` 的地址更快，更节省内存，这取决于 `Queue` 类型的对象大小。这次我们尝试这种方法。这样做的好处是，所有的函数都以地址作为参数，而不像 `List` 示例那样。为了表明这些函数不更改队列，可以且应该使用 `const` 限定符：

```
bool QueueIsFull(const Queue * pq);
bool QueueIsEmpty (const Queue * pq);
```

指针 `pq` 指向 `Queue` 数据对象，不能通过 `pq` 这个代理更改数据。可以定义一个类似该函数的原型，返回队列的项数：

```
int QueueItemCount (const Queue * pq);
```

在队列末尾添加项涉及标识项和队列。这次要更改队列，所以有必要（而不是可选）使用指针。该函数的返回类型可以是 `void`，或者通过返回值来表示是否成功添加项。我们采用后者：

```
bool EnQueue(Item item, Queue * pq);
```

最后，删除项有多种方法。如果把项定义为结构或一种基本类型，可以通过函数返回待删除的项。函

数的参数可以是 Queue 类型或指向 Queue 的指针。因此，可能是下面这样的原型：

```
Item DeQueue (Queue q);
```

然而，下面的原型会更合适一些：

```
bool DeQueue (Item * pitem, Queue * pq);
```

从队列中待删除的项储存在 pitem 指针指向的位置，函数的返回值表明是否删除成功。

清空队列的函数所需的唯一参数是队列的地址，可以使用下面的函数原型：

```
void EmptyTheQueue (Queue * pq);
```

### 17.4.3 实现接口数据表示

第一步是确定在队列中使用何种 C 数据形式。有可能是数组。数组的优点是方便使用，而且向数组的末尾添加项很简单。问题是如何从队列的开头删除项。类比于排队买票的队列，从队列的开头删除一个项包括拷贝数组首元素的值和把数组剩余各项依次向前移动一个位置。编程实现这个过程很简单，但是会浪费大量的计算机时间（见图 17.6）。

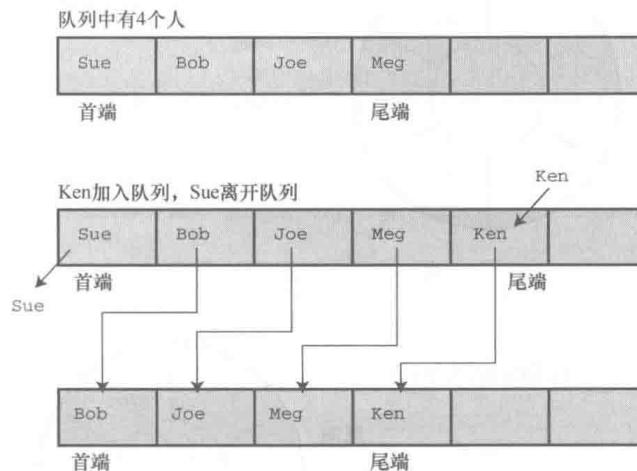


图 17.6 用数组实现队列

第二种解决数组队列删除问题的方法是改变队列首端的位置，其余元素不动（见图 17.7）。

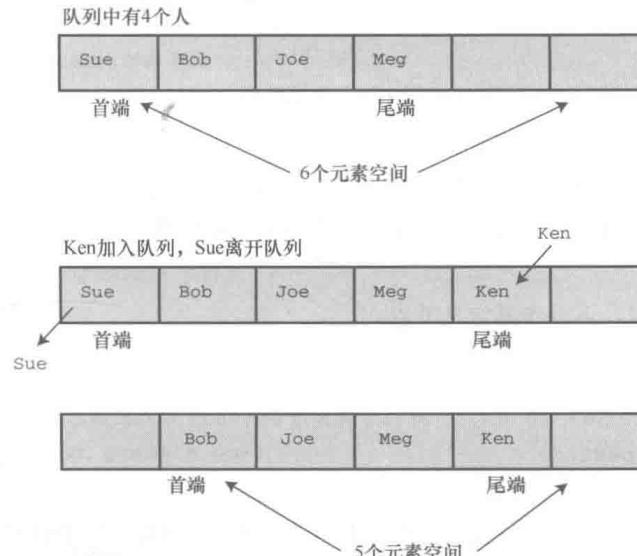


图 17.7 重新定义首元素

解决这种问题的一个好方法是，使队列成为环形。这意味着把数组的首尾相连，即数组的首元素紧跟在最后一个元素后面。这样，当到达数组末尾时，如果首元素空出，就可以把新添加的项储存到这些空出的元素中（见图 17.8）。可以想象在一张条形的纸上画出数组，然后把数组的首尾粘起来形成一个环。当然，要做一些标记，以免尾端超过首端。

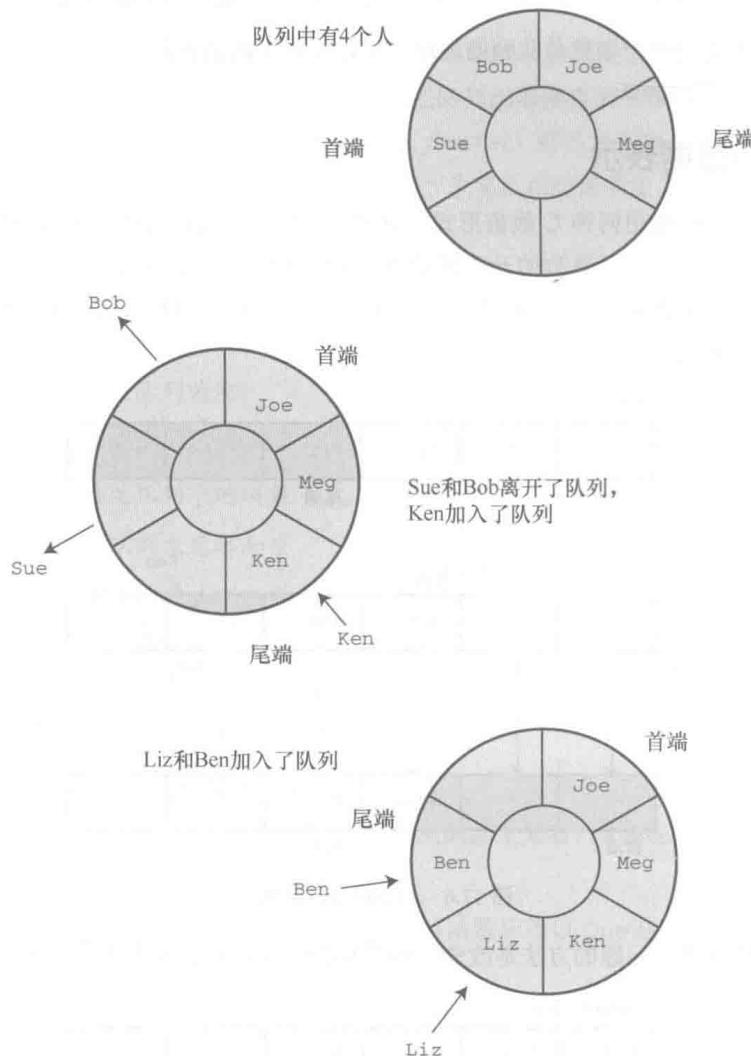


图 17.8 环形队列

另一种方法是使用链表。使用链表的好处是删除首项时不必移动其余元素，只需重置头指针指向新的首元素即可。由于我们已经讨论过链表，所以采用这个方案。我们用一个整数队列开始测试：

```
typedef int Item;
```

链表由节点组成，所以，下一步是定义节点：

```
typedef struct node
{
    Item item;
    struct node * next;
} Node;
```

对队列而言，要保存首尾项，这可以使用指针来完成。另外，可以用一个计数器来记录队列中的项数。因此，该结构应由两个指针成员和一个 int 类型的成员构成：

```

typedef struct queue
{
    Node * front; /* 指向队列首项的指针 */
    Node * rear; /* 指向队列尾项的指针 */
    int items; /* 队列中的项数 */
} Queue;

```

注意，Queue 是一个内含 3 个成员的结构，所以用指向队列的指针作为参数比直接用队列作为参数节约了时间和空间。

接下来，考虑队列的大小。对链表而言，其大小受限于可用的内存量，因此链表不要太大。例如，可能使用一个队列模拟飞机等待在机场着陆。如果等待的飞机数量太多，新到的飞机就应该改到其他机场降落。我们把队列的最大长度设置为 10。程序清单 17.6 包含了队列接口的原型和定义。Item 类型留给用户定义。使用该接口时，可以根据特定的程序插入合适的定义。

程序清单 17.6 queue.h 接口头文件

```

/* queue.h -- Queue 的接口 */
#ifndef _QUEUE_H_
#define _QUEUE_H_
#include <stdbool.h>

// 在这里插入 Item 类型的定义，例如
typedef int Item; // 用于 use_q.c
// 或者 typedef struct item {int gumption; int charisma;} Item;

#define MAXQUEUE 10

typedef struct node
{
    Item item;
    struct node * next;
} Node;

typedef struct queue
{
    Node * front; /* 指向队列首项的指针 */
    Node * rear; /* 指向队列尾项的指针 */
    int items; /* 队列中的项数 */
} Queue;

/* 操作： 初始化队列 */
/* 前提条件： pq 指向一个队列 */
/* 后置条件： 队列被初始化为空 */
void InitializeQueue(Queue * pq);

/* 操作： 检查队列是否已满 */
/* 前提条件： pq 指向之前被初始化的队列 */
/* 后置条件： 如果队列已满则返回 true，否则返回 false */
bool QueueIsFull(const Queue * pq);

/* 操作： 检查队列是否为空 */
/* 前提条件： pq 指向之前被初始化的队列 */
/* 后置条件： 如果队列为空则返回 true，否则返回 false */

```

```

bool QueueIsEmpty(const Queue *pq);

/* 操作:      确定队列中的项数
/* 前提条件:  pq 指向之前被初始化的队列
/* 后置条件:  返回队列中的项数
int QueueItemCount(const Queue * pq);

/* 操作:      在队列末尾添加项
/* 前提条件:  pq 指向之前被初始化的队列
/*          item 是要被添加在队列末尾的项
/* 后置条件:  如果队列不为空, item 将被添加在队列的末尾,
/*          该函数返回 true; 否则, 队列不改变, 该函数返回 false */
bool EnQueue(Item item, Queue * pq);

/* 操作:      从队列的开头删除项
/* 前提条件:  pq 指向之前被初始化的队列
/* 后置条件:  如果队列不为空, 队列首端的 item 将被拷贝到*pitem 中
/*          并被删除, 且函数返回 true;
/*          如果该操作使得队列为空, 则重置队列为空
/*          如果队列在操作前为空, 该函数返回 false
bool DeQueue(Item *pitem, Queue * pq);

/* 操作:      清空队列
/* 前提条件:  pq 指向之前被初始化的队列
/* 后置条件:  队列被清空
void EmptyTheQueue(Queue * pq);

#endif

```

## 1. 实现接口函数

接下来, 我们编写接口代码。首先, 初始化队列为空, 这里“空”的意思是把指向队列首项和尾项的指针设置为 NULL, 并把项数 (items 成员) 设置为 0:

```

void InitializeQueue(Queue * pq)
{
    pq->front = pq->rear = NULL;
    pq->items = 0;
}

```

这样, 通过检查 items 的值可以很方便地了解到队列是否已满、是否为空和确定队列的项数:

```

bool QueueIsFull(const Queue * pq)
{
    return pq->items == MAXQUEUE;
}

bool QueueIsEmpty(const Queue * pq)
{
    return pq->items == 0;
}

int QueueItemCount(const Queue * pq)
{
    return pq->items;
}

```

把项添加到队列中，包括以下几个步骤：

- (1) 创建一个新节点；
- (2) 把项拷贝到节点中；
- (3) 设置节点的 next 指针为 NULL，表明该节点是最后一个节点；
- (4) 设置当前尾节点的 next 指针指向新节点，把新节点链接到队列中；
- (5) 把 rear 指针指向新节点，以便找到最后的节点；
- (6) 项数加 1。

函数还要处理两种特殊情况。第一种情况，如果队列为空，应该把 front 指针设置为指向新节点。因为如果队列中只有一个节点，那么这个节点既是首节点也是尾节点。第二种情况是，如果函数不能为节点分配所需内存，则必须执行一些动作。因为大多数情况下我们都使用小型队列，这种情况很少发生，所以，如果程序运行的内存不足，我们只是通过函数终止程序。EnQueue() 的代码如下：

```
bool EnQueue(Item item, Queue * pq)
{
    Node * pnew;

    if (QueueIsFull(pq))
        return false;
    pnew = (Node *)malloc( sizeof(Node));
    if (pnew == NULL)
    {
        fprintf(stderr, "Unable to allocate memory!\n");
        exit(1);
    }
    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (QueueIsEmpty(pq))
        pq->front = pnew; /* 项位于队列前端 */
    else
        pq->rear->next = pnew; /* 链接到队列尾端 */
    pq->rear = pnew; /* 记录队列尾端的位置 */
    pq->items++; /* 队列项数加 1 */

    return true;
}
```

CopyToNode() 函数是静态函数，用于把项拷贝到节点中：

```
static void CopyToNode(Item item, Node * pn)
{
    pn->item = item;
}
```

从队列的首端删除项，涉及以下几个步骤：

- (1) 把项拷贝到给定的变量中；
- (2) 释放空出的节点使用的内存空间；
- (3) 重置首指针指向队列中的下一个项；
- (4) 如果删除最后一项，把首指针和尾指针都重置为 NULL；
- (5) 项数减 1。

下面的代码完成了这些步骤：

```
bool DeQueue(Item * pitem, Queue * pq)
{
    Node * pt;

    if (QueueIsEmpty(pq))
        return false;
    CopyToItem(pq->front, pitem);
    pt = pq->front;
    pq->front = pq->front->next;
    free(pt);
    pq->items--;
    if (pq->items == 0)
        pq->rear = NULL;

    return true;
}
```

关于指针要注意两点。第一，删除最后一项时，代码中并未显式设置 `front` 指针为 `NULL`，因为已经设置 `front` 指针指向被删除节点的 `next` 指针。如果该节点不是最后一个节点，那么它的 `next` 指针就为 `NULL`。第二，代码使用临时指针 (`pt`) 储存待删除节点的位置。因为指向首节点的正式指针 (`pt->front`) 被重置为指向下一个节点，所以如果没有临时指针，程序就不知道该释放哪块内存。

我们使用 `DeQueue()` 函数清空队列。循环调用 `DeQueue()` 函数直到队列为空：

```
void EmptyTheQueue(Queue * pq)
{
    Item dummy;
    while (!QueueIsEmpty(pq))
        DeQueue(&dummy, pq);
}
```

### 注意 保持纯正的 ADT

定义 ADT 接口后，应该只使用接口函数处理数据类型。例如，`Dequeue()` 依赖 `EnQueue()` 函数来正确设置指针和把 `rear` 节点的 `next` 指针设置为 `NULL`。如果在一个使用 ADT 的程序中，决定直接操控队列的某些部分，有可能破坏接口包中函数之间的协作关系。

程序清单 17.7 演示了该接口中的所有函数，包括 `EnQueue()` 函数中用到的 `CopyToItem()` 函数。

#### 程序清单 17.7 queue.c 实现文件

```
/* queue.c -- Queue 类型的实现 */
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

/* 局部函数 */
static void CopyToNode(Item item, Node * pn);
static void CopyToItem(Node * pn, Item * pi);

void InitializeQueue(Queue * pq)
{
    pq->front = pq->rear = NULL;
```

```

    pq->items = 0;
}

bool QueueIsFull(const Queue * pq)
{
    return pq->items == MAXQUEUE;
}

bool QueueIsEmpty(const Queue * pq)
{
    return pq->items == 0;
}

int QueueItemCount(const Queue * pq)
{
    return pq->items;
}

bool EnQueue(Item item, Queue * pq)
{
    Node * pnew;

    if (QueueIsFull(pq))
        return false;
    pnew = (Node *) malloc(sizeof(Node));
    if (pnew == NULL)
    {
        fprintf(stderr, "Unable to allocate memory!\n");
        exit(1);
    }
    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (QueueIsEmpty(pq))
        pq->front = pnew; /* 项位于队列的前端 */
    else
        pq->rear->next = pnew; /* 链接到队列的尾端 */
    pq->rear = pnew; /* 记录队列尾端的位置 */
    pq->items++; /* 队列项数加 1 */

    return true;
}

bool DeQueue(Item * pitem, Queue * pq)
{
    Node * pt;

    if (QueueIsEmpty(pq))
        return false;
    CopyToItem(pq->front, pitem);
    pt = pq->front;
    pq->front = pq->front->next;
    free(pt);
    pq->items--;
    if (pq->items == 0)
        pq->rear = NULL;
}

```

```

    return true;
}

/* 清空队列 */
void EmptyTheQueue(Queue * pq)
{
    Item dummy;
    while (!QueueIsEmpty(pq))
        DeQueue(&dummy, pq);
}

/* 局部函数 */

static void CopyToNode(Item item, Node * pn)
{
    pn->item = item;
}

static void CopyToItem(Node * pn, Item * pi)
{
    *pi = pn->item;
}

```

#### 17.4.4 测试队列

在重要程序中使用一个新的设计（如，队列包）之前，应该先测试该设计。测试的一种方法是，编写一个小程序。这样的程序称为驱动程序 (*driver*)，其唯一的用途是进行测试。例如，程序清单 17.8 使用一个添加和删除整数的队列。在运行该程序之前，要确保 `queue.h` 中包含下面这行代码：

```
typedef int item;
```

记住，还必须链接 `queue.c` 和 `use_q.c`。

**程序清单 17.8 use\_q.c 程序**

---

```

/* use_q.c -- 驱动程序测试 Queue 接口 */
/* 与 queue.c 一起编译 */
#include <stdio.h>
#include "queue.h" /* 定义 Queue、Item */

int main(void)
{
    Queue line;
    Item temp;
    char ch;

    InitializeQueue(&line);
    puts("Testing the Queue interface. Type a to add a value,");
    puts("type d to delete a value, and type q to quit.");
    while ((ch = getchar()) != 'q')
    {
        if (ch != 'a' && ch != 'd') /* 忽略其他输出 */
            continue;

```

```

if (ch == 'a')
{
    printf("Integer to add: ");
    scanf("%d", &temp);
    if (!QueueIsFull(&line))
    {
        printf("Putting %d into queue\n", temp);
        EnQueue(temp, &line);
    }
    else
        puts("Queue is full!");
}
else
{
    if (QueueIsEmpty(&line))
        puts("Nothing to delete!");
    else
    {
        DeQueue(&temp, &line);
        printf("Removing %d from queue\n", temp);
    }
}
printf("%d items in queue\n", QueueItemCount(&line));
puts("Type a to add, d to delete, q to quit:");
}
EmptyTheQueue(&line);
puts("Bye!");

return 0;
}

```

下面是一个运行示例。除了这样测试，还应该测试当队列已满后，实现是否能正常运行。

```

Testing the Queue interface. Type a to add a value,
type d to delete a value, and type q to quit.

a
Integer to add: 40
Putting 40 into queue
1 items in queue
Type a to add, d to delete, q to quit:
a
Integer to add: 20
Putting 20 into queue
2 items in queue
Type a to add, d to delete, q to quit:
a
Integer to add: 55
Putting 55 into queue
3 items in queue
Type a to add, d to delete, q to quit:
d
Removing 40 from queue
2 items in queue
Type a to add, d to delete, q to quit:
d
Removing 20 from queue

```

```

1 items in queue
Type a to add, d to delete, q to quit:
d
Removing 55 from queue
0 items in queue
Type a to add, d to delete, q to quit:
d
Nothing to delete!
0 items in queue
Type a to add, d to delete, q to quit:
q
Bye!

```

## 17.5 用队列进行模拟

经过测试，队列没问题。现在，我们用它来做一些有趣的事情。许多现实生活的情形都涉及队列。例如，在银行或超市的顾客队列、机场的飞机队列、多任务计算机系统中的任务队列等。我们可以用队列包来模拟这些情形。

假设 Sigmund Landers 在商业街设置了一个提供建议的摊位。顾客可以购买 1 分钟、2 分钟或 3 分钟的建议。为确保交通畅通，商业街规定每个摊位前排队等待的顾客最多为 10 人（相当于程序中的最大队列长度）。假设顾客都是随机出现的，并且他们花在咨询上的时间也是随机选择的（1 分钟、2 分钟、3 分钟）。那么 Sigmund 平均每小时要接待多少名顾客？每位顾客平均要花多长时间？排队等待的顾客平均有多少人？队列模拟能回答类似的问题。

首先，要确定在队列中放什么。可以根据顾客加入队列的时间和顾客咨询时花费的时间来描述每一位顾客。因此，可以这样定义 Item 类型。

```

typedef struct item
{
    long arrive;      /* 一位顾客加入队列的时间 */
    int processtime; /* 该顾客咨询时花费的时间 */
} Item;

```

要用队列包来处理这个结构，必须用 `typedef` 定义的 `Item` 替换上一个示例的 `int` 类型。这样做就不用担心队列的具体工作机制，可以集中精力分析实际问题，即模拟咨询 Sigmund 的顾客队列。

这里有一种方法，让时间以 1 分钟为单位递增。每递增 1 分钟，就检查是否有新顾客到来。如果有一位顾客且队列未满，将该顾客添加到队列中。这涉及把顾客到来的时间和顾客所需的咨询时间记录在 `Item` 类型的结构中，然后在队列中添加该项。然而，如果队列已满，就让这位顾客离开。为了做统计，要记录顾客的总数和被拒顾客（队列已满不能加入队列的人）的总数。

接下来，处理队列的首端。也就是说，如果队列不为空且前面的顾客没有在咨询 Sigmund，则删除队列首端的项。记住，该项中储存着这位顾客加入队列的时间，把该时间与当前时间作比较，就可得出该顾客在队列中等待的时间。该项还储存着这位顾客需要咨询的分钟数，即还要咨询 Sigmund 多长时间。因此还要用一个变量储存这个时长。如果 Sigmund 正忙，则不用让任何人离开队列。尽管如此，记录等待时间的变量应该递减 1。

核心代码类似下面这样，每一轮迭代对应 1 分钟的行为：

```

for (cycle = 0; cycle < cyclelimit; cycle++)
{
    if (newcustomer(min_per_cust))
    {

```

```

    if (QueueIsFull(&line))
        turnaways++;
    else
    {
        customers++;
        temp = customertime(cycle);
        EnQueue(temp, &line);
    }
}
if (wait_time <= 0 && !QueueIsEmpty(&line))
{
    DeQueue(&temp, &line);
    wait_time = temp.processtime;
    line_wait += cycle - temp.arrive;
    served++;
}
if (wait_time > 0)
    wait_time--;
sum_line += QueueItemCount(&line);
}

```

注意，时间的表示比较粗糙（1分钟），所以一小时最多60位顾客。下面是一些变量和函数的含义。

- min\_per\_cus 是顾客到达的平均间隔时间。
- newcustomer() 使用 C 的 rand() 函数确定在特定时间内是否有顾客到来。
- turnaways 是被拒绝的顾客数量。
- customers 是加入队列的顾客数量。
- temp 是表示新顾客的 Item 类型变量。
- customertime() 设置 temp 结构中的 arrive 和 processtime 成员。
- wait\_time 是 Sigmund 完成当前顾客的咨询还需多长时间。
- line\_wait 是到目前为止队列中所有顾客的等待总时间。
- served 是咨询过 Sigmund 的顾客数量。
- sum\_line 是到目前为止统计的队列长度。

如果到处都是 malloc()、free() 和指向节点的指针，整个程序代码会非常混乱和晦涩。队列包让你把注意力集中在模拟问题上，而不是编程细节上。

程序清单 17.9 演示了模拟商业街咨询摊位队列的完整代码。根据第 12 章介绍的方法，使用标准函数 rand()、srand() 和 time() 来产生随机数。另外要特别注意，必须用下面的代码更新 queue.h 中的 Item，该程序才能正常工作：

```

typedef struct item
{
    long arrive;          //一位顾客加入队列的时间
    int processtime;     //该顾客咨询时花费的时间
} Item;

```

记住，还要把 mall.c 和 queue.c 一起链接。

#### 程序清单 17.9 mall.c 程序

---

```

// mall.c -- 使用 Queue 接口
// 和 queue.c 一起编译

```

```

#include <stdio.h>
#include <stdlib.h>           // 提供 rand() 和 srand() 的原型
#include <time.h>             // 提供 time() 的原型
#include "queue.h"             // 更改 Item 的 typedef
#define MIN_PER_HR 60.0

bool newcustomer(double x);    // 是否有新顾客到来?
Item customertime(long when); // 设置顾客参数

int main(void)
{
    Queue line;
    Item temp;           // 新的顾客数据
    int hours;           // 模拟的小时数
    int perhour;          // 每小时平均多少位顾客
    long cycle, cyclelimit; // 循环计数器、计数器的上限
    long turnaways = 0;   // 因队列已满被拒的顾客数量
    long customers = 0;   // 加入队列的顾客数量
    long served = 0;      // 在模拟期间咨询过 Sigmund 的顾客数量
    long sum_line = 0;     // 累计的队列总长
    int wait_time = 0;     // 从当前到 Sigmund 空闲所需的时间
    double min_per_cust;  // 顾客到来的平均时间
    long line_wait = 0;    // 队列累计的等待时间

    InitializeQueue(&line);
    srand((unsigned int) time(0)); // rand() 随机初始化
    puts("Case Study: Sigmund Lander's Advice Booth");
    puts("Enter the number of simulation hours:");
    scanf("%d", &hours);
    cyclelimit = MIN_PER_HR * hours;
    puts("Enter the average number of customers per hour:");
    scanf("%d", &perhour);
    min_per_cust = MIN_PER_HR / perhour;

    for (cycle = 0; cycle < cyclelimit; cycle++)
    {
        if (newcustomer(min_per_cust))
        {
            if (QueueIsFull(&line))
                turnaways++;
            else
            {
                customers++;
                temp = customertime(cycle);
                EnQueue(temp, &line);
            }
        }
        if (wait_time <= 0 && !QueueIsEmpty(&line))
        {
            DeQueue(&temp, &line);
            wait_time = temp.processtime;
            line_wait += cycle - temp.arrive;
        }
    }
}

```

```

        served++;
    }
    if (wait_time > 0)
        wait_time--;
    sum_line += QueueItemCount(&line);
}

if (customers > 0)
{
    printf("customers accepted: %ld\n", customers);
    printf(" customers served: %ld\n", served);
    printf("      turnaways: %ld\n", turnaways);
    printf("average queue size: %.2f\n",
           (double) sum_line / cyclelimit);
    printf(" average wait time: %.2f minutes\n",
           (double) line_wait / served);
}
else
    puts("No customers!");
EmptyTheQueue(&line);
puts("Bye!");

return 0;
}

// x 是顾客到来的平均时间 (单位: 分钟)
// 如果 1 分钟内有顾客到来, 则返回 true
bool newcustomer(double x)
{
    if (rand() * x / RAND_MAX < 1)
        return true;
    else
        return false;
}

// when 是顾客到来的时间
// 该函数返回一个 Item 结构, 该顾客到达的时间设置为 when,
// 咨询时间设置为 1~3 的随机值
Item customertime(long when)
{
    Item cust;

    cust.processtime = rand() % 3 + 1;
    cust.arrive = when;

    return cust;
}

```

该程序允许用户指定模拟运行的小时数和每小时平均有多少位顾客。模拟时间较长得出的值较为平均, 模拟时间较短得出的值随时间的变化而随机变化。下面的运行示例解释了这一点(先保持每小时的顾客平均数量不变)。注意, 在模拟 80 小时和 800 小时的情况下, 平均队伍长度和等待时间基本相同。但是, 在模拟 1 小时的情况下这两个量差别很大, 而且与长时间模拟的情况差别也很大。这是因为小数量的统计样本往往更容易受相对变化的影响。

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
80
Enter the average number of customers per hour:
20
customers accepted: 1633
customers served: 1633
turnaways: 0
average queue size: 0.46
average wait time: 1.35 minutes
```

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
800
Enter the average number of customers per hour:
20
customers accepted: 16020
customers served: 16019
turnaways: 0
average queue size: 0.44
average wait time: 1.32 minutes
```

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
1
Enter the average number of customers per hour:
20
customers accepted: 20
customers served: 20
turnaways: 0
average queue size: 0.23
average wait time: 0.70 minutes
```

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
1
Enter the average number of customers per hour:
20
customers accepted: 22
customers served: 22
turnaways: 0
average queue size: 0.75
average wait time: 2.05 minutes
```

然后保持模拟的时间不变，改变每小时的顾客平均数量：

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
80
Enter the average number of customers per hour:
25
customers accepted: 1960
customers served: 1959
turnaways: 3
average queue size: 1.43
average wait time: 3.50 minutes
```

Case Study: Sigmund Lander's Advice Booth

```

Enter the number of simulation hours:
80
Enter the average number of customers per hour:
30
customers accepted: 2376
customers served: 2373
turnaways: 94
average queue size: 5.85
average wait time: 11.83 minutes

```

注意，随着每小时顾客平均数量的增加，顾客的平均等待时间迅速增加。在每小时 20 位顾客（80 小时模拟时间）的情况下，每位顾客的平均等待时间是 1.35 分钟；在每小时 25 位顾客的情况下，平均等待时间增加至 3.50 分钟；在每小时 30 位顾客的情况下，该数值攀升至 11.83 分钟。而且，这 3 种情况下被拒顾客分别从 0 位增加至 3 位最后陡增至 94 位。Sigmund 可以根据程序模拟的结果决定是否要增加一个摊位。

## 17.6 链表和数组

许多编程问题，如创建一个简单链表或队列，都可以用链表（指的是动态分配结构的序列链）或数组来处理。每种形式都有其优缺点，所以要根据具体问题的要求来决定选择哪一种形式。表 17.1 总结了链表和数组的性质。

表 17.1 比较数组和链表

| 数据形式 | 优点                   | 缺点                     |
|------|----------------------|------------------------|
| 数组   | C 直接支持<br>提供随机访问     | 在编译时确定大小<br>插入和删除元素很费时 |
| 链表   | 运行时确定大小<br>快速插入和删除元素 | 不能随机访问<br>用户必须提供编程支持   |

接下来，详细分析插入和删除元素的过程。在数组中插入元素，必须移动其他元素腾出空位插入新元素，如图 17.9 所示。新插入的元素离数组开头越近，要被移动的元素越多。然而，在链表中插入节点，只需给两个指针赋值，如图 17.10 所示。类似地，从数组中删除一个元素，也要移动许多相关的元素。但是从链表中删除节点，只需重新设置一个指针并释放被删除节点占用的内存即可。

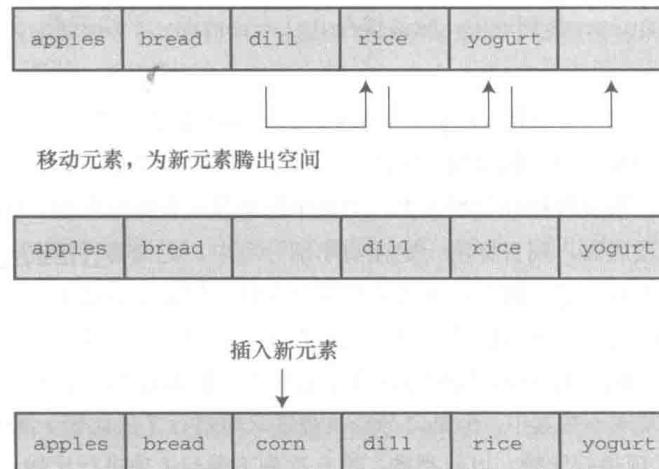


图 17.9 在数组中插入一个元素

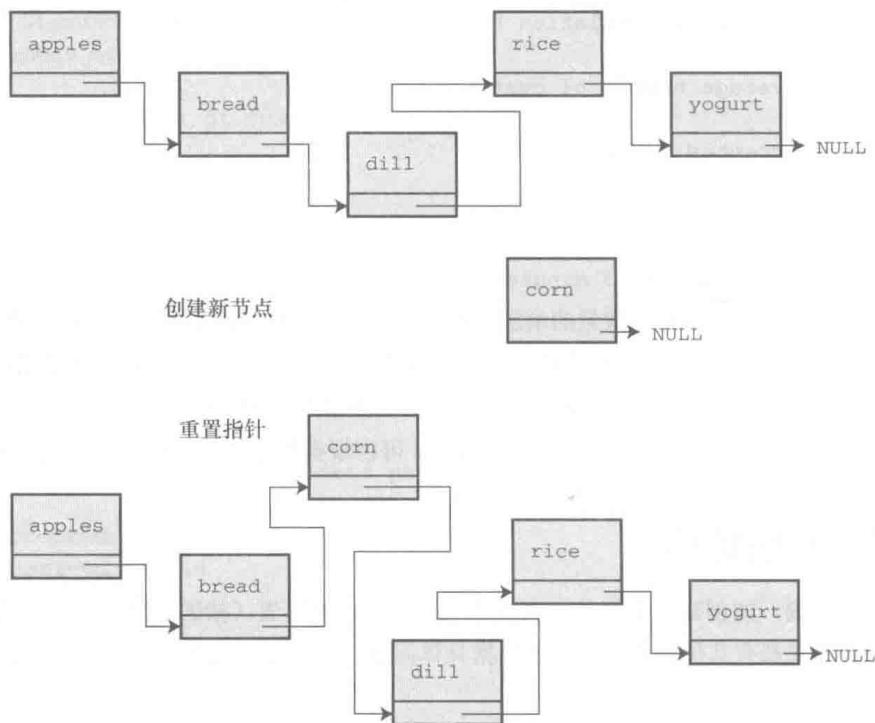


图 17.10 在链表中插入一个元素

接下来，考虑如何访问元素。对数组而言，可以使用数组下标直接访问该数组中的任意元素，这叫做随机访问 (*random access*)。对链表而言，必须从链表首节点开始，逐个节点移动到要访问的节点，这叫做顺序访问 (*sequential access*)。当然，也可以顺序访问数组。只需按顺序递增数组下标即可。在某些情况下，顺序访问足够了。例如，显示链表中的每一项，顺序访问就不错。其他情况用随机访问更合适。

假设要查找链表中的特定项。一种算法是从列表的开头开始按顺序查找，这叫做顺序查找 (*sequential search*)。如果项并未按某种顺序排列，则只能顺序查找。如果待查找的项不在链表里，必须查找完所有的项才知道该项不在链表中（在这种情况下可以使用并发编程，同时查找列表中的不同部分）。

我们可以先排序列表，以改进顺序查找。这样，就不必查找排在待查找项后面的项。例如，假设在一个按字母排序的列表中查找 Susan。从开头开始查找每一项，直到 Sylvia 都没有查找到 Susan。这时就可以退出查找，因为如果 Susan 在列表中，应该排在 Sylvia 前面。平均下来，这种方法查找不在列表中的项的时间减半。

对于一个排序的列表，用二分查找 (*binary search*) 比顺序查找好得多。下面分析二分查找的原理。首先，把待查找的项称为目标项，而且假设列表中的各项按字母排序。然后，比较列表的中间项和目标项。如果两者相等，查找结束；假设目标项在列表中，如果中间项排在目标项前面，则目标项一定在后半部分项中；如果中间项在目标项后面，则目标项一定在前半部分项中。无论哪种情况，两项比较的结果都确定了下次查找的范围只有列表的一半。接着，继续使用这种方法，把需要查找的剩下一半的中间项与目标项比较。同样，这种方法会确定下一次查找的范围是当前查找范围的一半。以此类推，直到找到目标项或最终发现列表中没有目标项（见图 17.11）。这种方法非常有效率。假如有 127 个项，顺序查找平均要进行 64 次比较才能找到目标项或发现不在其中。但是二分查找最多只用进行 7 次比较。第 1 次比较剩下 63 项进行比较，第 2 次比较剩下 31 项进行比较，以此类推，第 6 次剩下最后 1 项进行比较，第 7 次比较确定剩下的这个项是否是目标项。一般而言， $n$  次比较能处理有  $2^n - 1$  个元素的数组。所以项数越多，越能体现二分查找的优势。

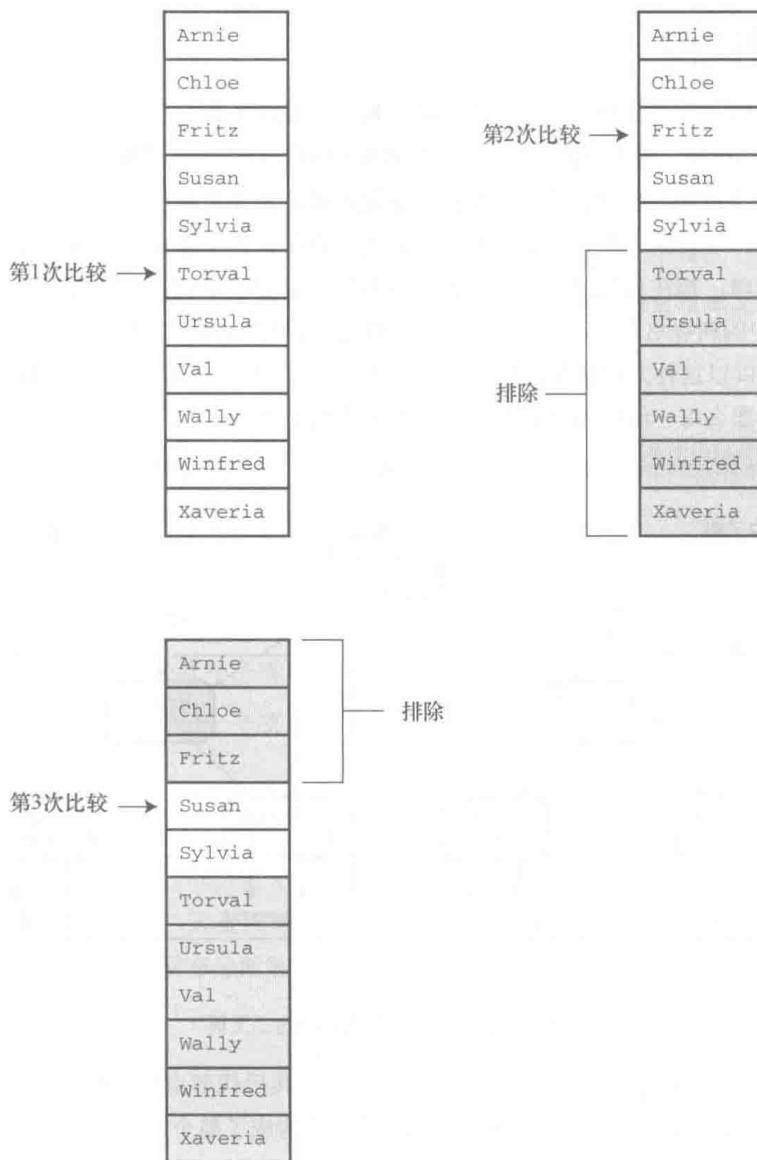


图 17.11 用二分查找法查找 Susan

用数组实现二分查找很简单，因为可以使用数组下标确定数组中任意部分的中点。只要把数组的首元素和尾元素的索引相加，得到的和再除以 2 即可。例如，内含 100 个元素的数组，首元素下标是 0，尾元素下标是 99，那么用于首次比较的中间项的下标应为  $(0+99)/2$ ，得 49（整数除法）。如果比较的结果是下标为 49 的元素在目标项的后面，那么目标项的下标应在 0~48 的范围内。所以，第 2 次比较的中间项的下标应为  $(0+48)/2$ ，得 24。如果中间项与目标项的比较结果是，中间项在目标项前面，那么第 3 次比较的中间项下标应为  $(25+48)/2$ ，得 36。这体现了随机访问的特性，可以从一个位置跳至另一个位置，不用一次访问两位置之间的项。但是，链表只支持顺序访问，不提供跳至中间节点的方法。所以在链表中不能使用二分查找。

如前所述，选择何种数据类型取决于具体的问题。如果因频繁地插入和删除项导致经常调整大小，而且不需要经常查找，选择链表会更好。如果只是偶尔插入或删除项，但是经常进行查找，使用数组会更好。

如果需要一种既支持频繁插入和删除项又支持频繁查找的数据形式，数组和链表都无法胜任，怎么办？这种情况下应该选择二叉查找树。

## 17.7 二叉查找树

二叉查找树是一种结合了二分查找策略的链接结构。二叉树的每个节点都包含一个项和两个指向其他节点（称为子节点）的指针。图 17.12 演示了二叉查找树中的节点是如何链接的。二叉树中的每个节点都包含两个子节点——左节点和右节点，其顺序按照如下规定确定：左节点的项在父节点的项前面，右节点的项在父节点的项后面。这种关系存在于每个有子节点的节点中。进一步而言，所有可以追溯其祖先回到一个父节点的左节点的项，都在该父节点项的前面；所有以一个父节点的右节点为祖先的项，都在该父节点项的后面。图 17.12 中的树以这种方式储存单词。有趣的是，与植物学的树相反，该树的顶部被称为根(*root*)。树具有分层组织，所以以这种方式储存的数据也以等级或层次组织。一般而言，每级都有上一级和下一级。如果二叉树是满的，那么每一级的节点数都是上一级节点数的两倍。

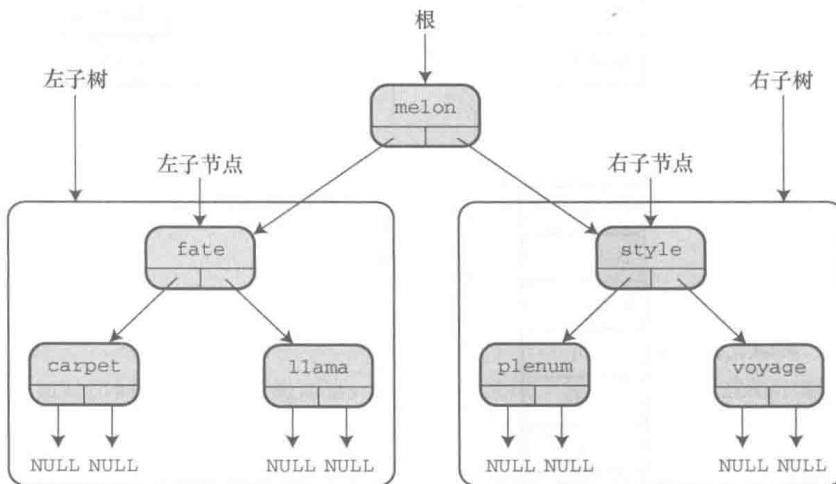


图 17.12 一个从存储单词的二叉树

二叉查找树中的每个节点是其后代节点的根，该节点与其后代节点构成称了一个子树 (*subtree*)。如图 17.12 所示，包含单词 *fate*、*carpet* 和 *llama* 的节点构成了整个二叉树的左子树，而单词 *voyage* 是 *style-plenum-voyage* 子树的右子树。

假设要在二叉树中查找一个项（即目标项）。如果目标项在根节点项的前面，则只需查找左子树；如果目标项在根节点项的后面，则只需查找右子树。因此，每次比较就排除半个树。假设查找左子树，这意味着目标项与左子节点项比较。如果目标项在左子节点项的前面，则只需查找其后代节点的左半部分，以此类推。与二分查找类似，每次比较都能排除一半的可能匹配项。

我们用这种方法来查找 *puppy* 是否在图 17.12 的二叉树中。比较 *puppy* 和 *melon*（根节点项），如果 *puppy* 在该树中，一定在右子树中。因此，在右子树中比较 *puppy* 和 *style*，发现 *puppy* 在 *style* 前面，所以必须链接到其左节点。然后发现该节点是 *plenum*，在 *puppy* 前面。现在要向下链接到该节点的右子节点，但是没有右子节点了。所以经过 3 次比较后发现 *puppy* 不在该树中。

二叉查找树在链式结构中结合了二分查找的效率。但是，这样编程的代价是构建一个二叉树比创建一个链表更复杂。下面我们在下一个 ADT 项目中创建一个二叉树。

### 17.7.1 二叉树 ADT

和前面一样，先从概括地定义二叉树开始。该定义假设树不包含相同的项。许多操作与链表相同，区

别在于数据层次的安排。下面建立一个非正式的树定义：

|       |  |
|-------|--|
| 类型名：  | 二叉查找树  |
| 类型属性： | 二叉树要么是空节点的集合（空树），要么是有一个根节点的节点集合<br>每个节点都有两个子树，叫做左子树和右子树<br>每个子树本身也是一个二叉树，也有可能是空树<br>二叉查找树是一个有序的二叉树，每个节点包含一个项，<br>左子树的所有项都在根节点项的前面，右子树的所有项都在根节点项的后面 |
| 类型操作： | 初始化树为空<br>确定树是否为空<br>确定树是否已满<br>确定树中的项数<br>在树中添加一个项<br>在树中删除一个项<br>在树中查找一个项<br>在树中访问一个项<br>清空树   |

## 17.7.2 二叉查找树接口

原则上，可以用多种方法实现二叉查找树，甚至可以通过操控数组下标用数组来实现。但是，实现二叉查找树最直接的方法是通过指针动态分配链式节点。因此我们这样定义：

```
typedef SOMETHING Item;

typedef struct trnode
{
    Item item;
    struct trnode * left;
    struct trnode * right;
} Trn;

typedef struct tree
{
    Trnode * root;
    int size;
} Tree;
```

每个节点包含一个项、一个指向左子节点的指针和一个指向右子节点的指针。可以把 Tree 定义为指向 Trnode 的指针类型，因为只需要知道根节点的位置就可访问整个树。然而，使用有成员大小的结构能很方便地记录树的大小。

我们要开发一个维护 Nerfville 宠物俱乐部的花名册，每一项都包含宠物名和宠物的种类。程序清单 17.10 就是该花名册的接口。我们把树的大小限制为 10，较小的树便于在树已满时测试程序的行为是否正确。当然，你也可以把 MAXITEMS 设置为更大的值。

## 程序清单 17.10 tree.h 接口头文件

```

/* tree.h -- 二叉查找树
 * 树种不允许有重复的项 */
#ifndef _TREE_H_
#define _TREE_H_
#include <stdbool.h>

/* 根据具体情况重新定义 Item */
#define SLEN 20
typedef struct item
{
    char petname[SLEN];
    char petkind[SLEN];
} Item;

#define MAXITEMS 10

typedef struct trnode
{
    Item item;
    struct trnode * left;      /* 指向左分支的指针 */
    struct trnode * right;     /* 指向右分支的指针 */
} Trnode;

typedef struct tree
{
    Trnode * root; /* 指向根节点的指针 */           */
    int size;       /* 树的项数 */                  */
} Tree;

/* 函数原型 */

/* 操作：      把树初始化为空*/
/* 前提条件：  ptree 指向一个树 */          */
/* 后置条件：  树被初始化为空 */          */
void InitializeTree(Tree * ptree);

/* 操作：      确定树是否为空 */            */
/* 前提条件：  ptree 指向一个树 */          */
/* 后置条件：  如果树为空，该函数返回 true */ */
/*           否则，返回 false */             */
bool TreeIsEmpty(const Tree * ptree);

/* 操作：      确定树是否已满 */            */
/* 前提条件：  ptree 指向一个树 */          */
/* 后置条件：  如果树已满，该函数返回 true */ */
/*           否则，返回 false */             */
bool TreeIsFull(const Tree * ptree);

/* 操作：      确定树的项数 */            */
/* 前提条件：  ptree 指向一个树 */          */

```

```

/* 后置条件： 返回树的项数 */  

int TreeItemCount(const Tree * ptree);  
  

/* 操作： 在树中添加一个项 */  

/* 前提条件： pi 是待添加项的地址 */  

/* ptree 指向一个已初始化的树 */  

/* 后置条件： 如果可以添加，该函数将在树中添加一个项 */  

/* 并返回 true；否则，返回 false */  

bool AddItem(const Item * pi, Tree * ptree);  
  

/* 操作： 在树中查找一个项 */  

/* 前提条件： pi 指向一个项 */  

/* ptree 指向一个已初始化的树 */  

/* 后置条件： 如果在树中添加一个项，该函数返回 true */  

/* 否则，返回 false */  

bool InTree(const Item * pi, const Tree * ptree);  
  

/* 操作： 从树中删除一个项 */  

/* 前提条件： pi 是删除项的地址 */  

/* ptree 指向一个已初始化的树 */  

/* 后置条件： 如果从树中成功删除一个项，该函数返回 true */  

/* 否则，返回 false */  

bool DeleteItem(const Item * pi, Tree * ptree);  
  

/* 操作： 把函数应用于树中的每一项 */  

/* 前提条件： ptree 指向一个树 */  

/* pfun 指向一个函数， */  

/* 该函数接受一个 Item 类型的参数，并无返回值*/  

/* 后置条件： pfun 指向的这个函数为树中的每一项执行一次 */  

void Traverse(const Tree * ptree, void(*pfun)(Item item));  
  

/* 操作： 删除树中的所有内容 */  

/* 前提条件： ptree 指向一个已初始化的树 */  

/* 后置条件： 树为空 */  

void DeleteAll(Tree * ptree);  
  

#endif

```

### 17.7.3 二叉树的实现

接下来，我们要实现 tree.h 中的每个函数。InitializeTree()、EmptyTree()、FullTree() 和 TreeItems() 函数都很简单，与链表 ADT、队列 ADT 类似，所以下面着重讲解其他函数。

#### 1. 添加项

在树中添加一个项，首先要检查该树是否有空间放得下一个项。由于我们定义二叉树时规定其中的项不能重复，所以接下来要检查树中是否有该项。通过这两步检查后，便可创建一个新节点，把待添加项拷贝到该节点中，并设置节点的左指针和右指针都为 NULL。这表明该节点没有子节点。然后，更新 Tree 结

构的 size 成员，统计新增了一项。接下来，必须找出应该把这个新节点放在树中的哪个位置。如果树为空，则应设置根节点指针指向该新节点。否则，遍历树找到合适的位置放置该节点。AddItem() 函数就根据这个思路来实现，并把一些工作交给几个尚未定义的函数：SeekItem()、MakeNode() 和 AddNode()。

```
bool AddItem(const Item * pi, Tree * ptree)
{
    Trnode * new_node;
    if (TreeIsFull(ptree))
    {
        fprintf(stderr, "Tree is full\n");
        return false; /* 提前返回 */
    }
    if (SeekItem(pi, ptree).child != NULL)
    {
        fprintf(stderr, "Attempted to add duplicate item\n");
        return false; /* 提前返回 */
    }
    new_node = MakeNode(pi); /* 指向新节点 */
    if (new_node == NULL)
    {
        fprintf(stderr, "Couldn't create node\n");
        return false; /* 提前返回 */
    }
    /* 成功创建了一个新节点 */
    ptree->size++;
    if (ptree->root == NULL) /* 情况 1：树为空 */
        ptree->root = new_node; /* 新节点是根节点 */
    else /* 情况 2：树不为空 */
        AddNode(new_node, ptree->root); /* 在树中添加一个节点 */
    return true; /* 成功返回 */
}
```

SeekItem()、MakeNode() 和 AddNode() 函数不是 Tree 类型公共接口的一部分。它们是隐藏在 tree.c 文件中的静态函数，处理实现的细节（如节点、指针和结构），不属于公共接口。

MakeNode() 函数相当简单，它处理动态内存分配和初始化节点。该函数的参数是指向新项的指针，其返回值是指向新节点的指针。如果 malloc() 无法分配所需的内存，则返回空指针。只有成功分配了内存，MakeNode() 函数才会初始化新节点。下面是 MakeNode() 的代码：

```
static Trnode * MakeNode(const Item * pi)
{
    Trnode * new_node;
    new_node = (Trnode *) malloc(sizeof(Trnode));
    if (new_node != NULL)
    {
        new_node->item = *pi;
        new_node->left = NULL;
        new_node->right = NULL;
    }
    return new_node;
}
```

AddNode() 函数是二叉查找树中最麻烦的第 2 个函数。它必须确定新节点的位置，然后添加新节点。具体来说，该函数要比较新项和根项，以确定应该把新项放在左子树还是右子树中。如果新项是一个数字，

则使用<和>进行比较；如果新项是一个字符串，则使用 `strcmp()` 函数来比较。但是，该项是内含两个字符串的结构，所以，必须自定义用于比较的函数。如果新项应放在左子树中，`ToLeft()` 函数（稍后定义）返回 `true`；如果新项应放在右子树中，`ToRight()` 函数（稍后定义）返回 `true`。这两个函数分别相当于<和>。假设把新项放在左子树中。如果左子树为空，`AddNode()` 函数只需让左子节点指针指向新项即可。如果左子树不为空怎么办？此时，`AddNode()` 函数应该把新项和左子节点中的项做比较，以确定新项应该放在该子节点的左子树还是右子树。这个过程一直持续到函数发现一个空子树为止，并在此处添加新节点。递归是一种实现这种查找过程的方法，即把 `AddNode()` 函数应用于子节点，而不是根节点。当左子树或右子树为空时，即当 `root->left` 或 `root->right` 为 `NULL` 时，函数的递归调用序列结束。记住，`root` 是指向当前子树顶部的指针，所以每次递归调用它都指向一个新的下一级子树（递归详见第 9 章）。

```
static void AddNode(Trnode * new_node, Trnode * root)
{
    if (ToLeft(&new_node->item, &root->item))
    {
        if (root->left == NULL) /* 空子树 */
            root->left = new_node; /* 所以，在此处添加节点 */
        else
            AddNode(new_node, root->left); /* 否则，处理该子树*/
    }
    else if (ToRight(&new_node->item, &root->item))
    {
        if (root->right == NULL)
            root->right = new_node;
        else
            AddNode(new_node, root->right);
    }
    else /* 不应含有重复的项 */
    {
        fprintf(stderr, "location error in AddNode()\n");
        exit(1);
    }
}
```

`ToLeft()` 和 `ToRight()` 函数依赖于 `Item` 类型的性质。Nerfville 宠物俱乐部的成员名按字母排序。如果两个宠物名相同，按其种类排序。如果种类也相同，这两项属于重复项，根据该二叉树的定义，这是不允许的。回忆一下，如果标准 C 库函数 `strcmp()` 中的第 1 个参数表示的字符串在第 2 个参数表示的字符串前面，该函数则返回负数；如果两个字符串相同，该函数则返回 0；如果第 1 个字符串在第 2 个字符串后面，该函数则返回正数。`ToRight()` 函数的实现代码与该函数类似。通过这两个函数完成比较，而不是直接在 `AddNode()` 函数中直接比较，这样的代码更容易适应新的要求。当需要比较不同的数据形式时，就不必重写整个 `AddNode()` 函数，只需重写 `Toleft()` 和 `ToRight()` 即可。

```
static bool ToLeft(const Item * i1, const Item * i2)
{
    int compl;
    if ((compl = strcmp(i1->petname, i2->petname)) < 0)
        return true;
    else if (compl == 0 &&
             strcmp(i1->petkind, i2->petkind) < 0)
        return true;
    else
        return false;
}
```

## 2. 查找项

3个接口函数都要在树中查找特定项：AddItem()、InItem()和DeleteItem()。这些函数的实现中使用SeekItem()函数进行查找。DeleteItem()函数有一个额外的要求：该函数要知道待删除项的父节点，以便在删除子节点后更新父节点指向子节点的指针。因此，我们设计SeekItem()函数返回的结构包含两个指针：一个指针指向包含项的节点（如果未找到指定项则为NULL）；一个指针指向父节点（如果该节点为根节点，即没有父节点，则为NULL）。这个结构类型的定义如下：

```
typedef struct pair {
    Trnode * parent;
    Trnode * child;
} Pair;
```

SeekItem()函数可以用递归的方式实现。但是，为了给读者介绍更多编程技巧，我们这次使用while循环处理树中从上到下的查找。和AddNode()一样，SeekItem()也使用ToLeft()和ToRight()在树中导航。开始时，SeekItem()设置look.child指针指向该树的根节点，然后沿着目标项所在的路径重置look.child指向后续的子树。同时，设置look.parent指向后续的父节点。如果没有找到匹配的项，look.child则被设置为NULL。如果在根节点找到匹配的项，则设置look.parent为NULL，因为根节点没有父节点。下面是SeekItem()函数的实现代码：

```
static Pair SeekItem(const Item * pi, const Tree * ptree)
{
    Pair look;
    look.parent = NULL;
    look.child = ptree->root;
    if (look.child == NULL)
        return look; /* 提前退出 */
    while (look.child != NULL)
    {
        if (ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->left;
        }
        else if (ToRight(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->right;
        }
        else /* 如果前两种情况都不满足，则必定是相等的情况 */
            break; /* look.child 目标项的节点 */
    }
    return look; /* 成功返回 */
}
```

注意，如果SeekItem()函数返回一个结构，那么该函数可以与结构成员运算符一起使用。例如，AddItem()函数中有如下的代码：

```
if (SeekItem(pi, ptree).child != NULL)
```

有了SeekItem()函数后，编写InTree()公共接口函数就很简单了：

```
bool InTree(const Item * pi, const Tree * ptree)
{
    return (SeekItem(pi, ptree).child == NULL) ? false : true;
}
```

### 3. 考虑删除项

删除项是最复杂的任务，因为必须重新连接剩余的子树形成有效的树。在准备编写这部分代码之前，必须明确需要做什么。

图 17.13 演示了最简单的情况。待删除的节点没有子节点，这样的节点被称为叶节点 (*leaf*)。这种情况只需把父节点中的指针重置为 NULL，并使用 `free()` 函数释放已删除节点所占用的内存。

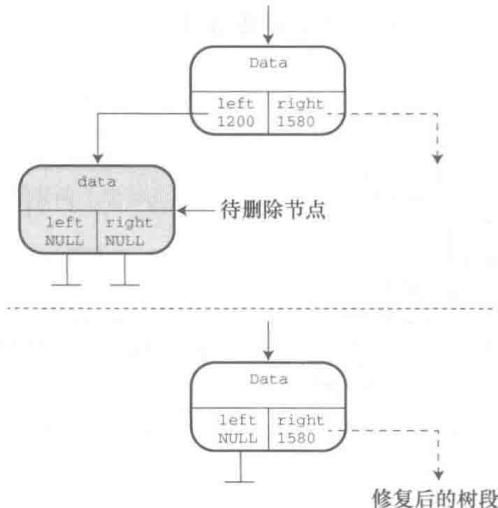


图 17.13 删除一个叶节点

删除带有一个子节点的情况比较复杂。删除该节点会导致其子树与其他部分分离。为了修正这种情况，要把被删除节点父节点中储存该节点的地址更新为该节点子树的地址（见图 17.14）。

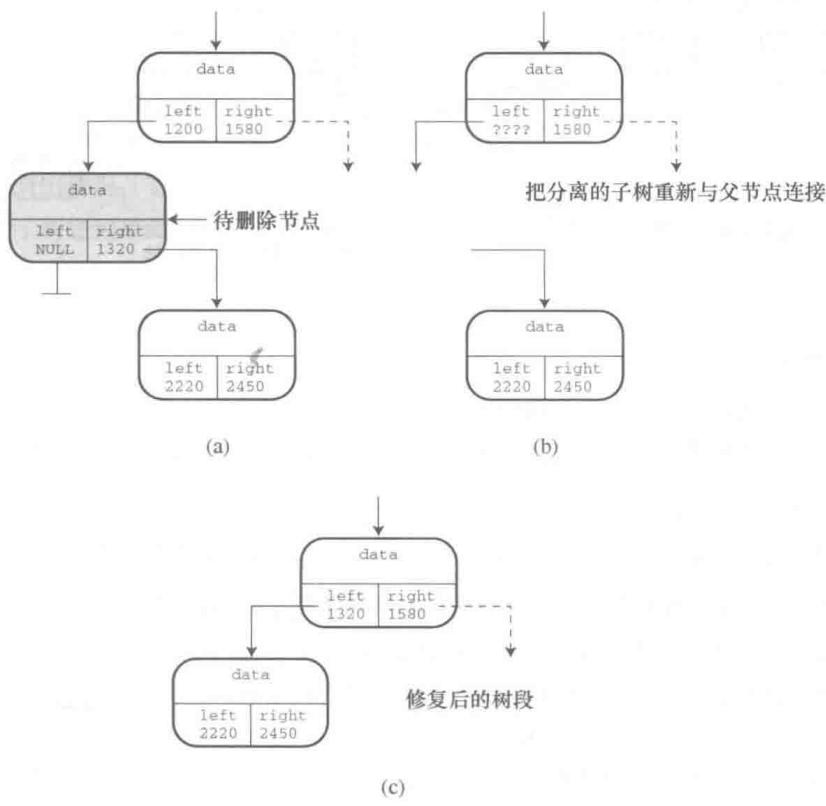


图 17.14 删除有一个子节点的节点

最后一种情况是删除有两个子树的节点。其中一个子树（如左子树）可连接在被删除节点之前连接的位置。但是，另一个子树怎么处理？牢记树的基本设计：左子树的所有项都在父节点项的前面，右子树的所有项都在父节点项的后面。也就是说，右子树的所有项都在左子树所有项的后面。而且，因为该右子树曾经是被删除节点的父节点的左子树的一部分，所以该右节点中的所有项在被删除节点的父节点项的前面。想像一下如何在树中从上到下查找该右子树的头所在的位置。它应该在被删除节点的父节点的前面，所以要沿着父节点的左子树向下找。但是，该右子树的所有项又在被删除节点左子树所有项的后面。因此要查看左子树的右支是否有新节点的空位。如果没有，就要沿着左子树的右支向下找，一直找到一个空位为止。图 17.15 演示了这种方法。

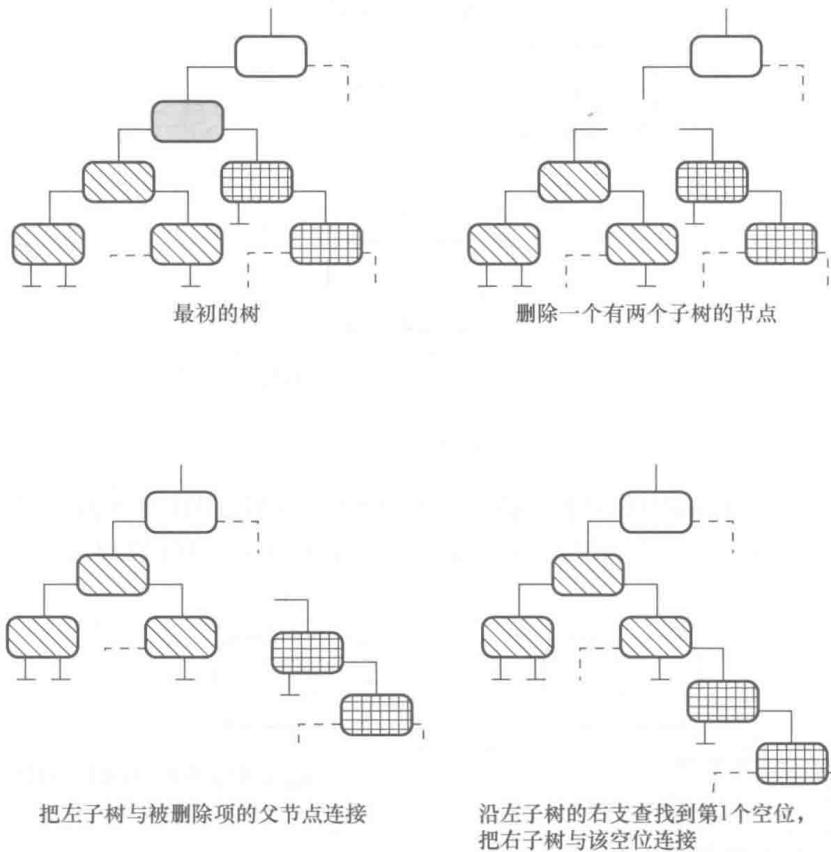


图 17.15 删除一个有两个子节点的项

### ① 删除一个节点

现在可以设计所需的函数了，可以分成两个任务：第一个任务是把特定项与待删除节点关联；第二个任务是删除节点。无论哪种情况都必须修改待删除项父节点的指针。因此，要注意以下两点。

- 该程序必须标识待删除节点的父节点。
- 为了修改指针，代码必须把该指针的地址传递给执行删除任务的函数。

第一点稍后讨论，下面先分析第二点。要修改的指针本身是 `Trnode **` 类型，即指向 `Trnode` 的指针。由于该函数的参数是该指针的地址，所以参数的类型是 `Trnode **`，即指向指针（该指针指向 `Trnode`）的指针。假设有合适的地址可用，可以这样编写执行删除任务的函数：

```
static void DeleteNode(Trnode **ptr)
/* ptr 是指向目标节点的父节点指针成员的地址 */
{
```

```

Trnode * temp;
if ((*ptr)->left == NULL)
{
    temp = *ptr;
    *ptr = (*ptr)->right;
    free(temp);
}
else if ((*ptr)->right == NULL)
{
    temp = *ptr;
    *ptr = (*ptr)->left;
    free(temp);
}
else /* 被删除的节点有两个子节点 */
{
    /* 找到重新连接右子树的位置 */
    for (temp = (*ptr)->left; temp->right != NULL;
         temp = temp->right)
        continue;
    temp->right = (*ptr)->right;
    temp = *ptr;
    *ptr = (*ptr)->left;
    free(temp);
}
}

```

该函数显式处理了 3 种情况：没有左子节点的节点、没有右子节点的节点和有两个子节点的节点。无子节点的节点可作为无左子节点的节点的特例。如果该节点没有左子节点，程序就将右子节点的地址赋给其父节点的指针。如果该节点也没有右子节点，则该指针为 NULL。这就是无子节点情况的值。

注意，代码中用临时指针记录被删除节点的地址。被删除节点的父节点指针 (\*ptr) 被重置后，程序会丢失被删除节点的地址，但是 free() 函数需要这个信息。所以，程序把 \*ptr 的原始值储存在 temp 中，然后用 free() 函数使用 temp 来释放被删除节点所占用的内存。

有两个子节点的情况，首先在 for 循环中通过 temp 指针从左子树的右半部分向下查找一个空位。找到空位后，把右子树连接于此。然后，再用 temp 保存被删除节点的位置。接下来，把左子树连接到被删除节点的父节点上，最后释放 temp 指向的节点。

注意，由于 ptr 的类型是 Trnode \*\*，所以 \*ptr 的类型是 Trnode \*，与 temp 的类型相同。

## ② 删除一个项

剩下的问题是把一个节点与特定项相关联。可以使用 SeekItem() 函数来完成。回忆一下，该函数返回一个结构（内含两个指针，一个指针指向父节点，一个指针指向包含特定项的节点）。然后就可以通过父节点的指针获得相应的地址传递给 DeleteNode() 函数。根据这个思路，DeleteNode() 函数的定义如下：

```

bool DeleteItem(const Item * pi, Tree * ptree)
{
    Pair look;
    look = SeekItem(pi, ptree);
    if (look.child == NULL)
        return false;
    if (look.parent == NULL) /* 删除根节点 */
        DeleteNode(&ptree->root);
}

```

```

    else if (look.parent->left == look.child)
        DeleteNode(&look.parent->left);
    else
        DeleteNode(&look.parent->right);
    ptree->size--;
}

return true;
}

```

首先, SeekItem() 函数的返回值被赋给 look 类型的结构变量。如果 look.child 是 NULL, 表明未找到指定项, DeleteItem() 函数退出, 并返回 false。如果找到了指定的 Item, 该函数分 3 种情况来处理。第一种情况是, look.parent 的值为 NULL, 这意味着该项在根节点中。在这情况下, 不用更新父节点, 但是要更新 Tree 结构中根节点的指针。因此, 函数该函数把该指针的地址传递给 DeleteNode() 函数。否则(即剩下两种情况), 程序判断待删除节点是其父节点的左子节点还是右子节点, 然后传递合适指针的地址。

注意, 公共接口函数 (DeleteItem()) 处理的是最终用户所关心的问题(项和树), 而隐藏的 DeleteNode() 函数处理的是与指针相关的实质性任务。

#### 4. 遍历树

遍历树比遍历链表更复杂, 因为每个节点都有两个分支。这种分支特性很适合使用分而制之的递归(详见第 9 章)来处理。对于每一个节点, 执行遍历任务的函数都要做如下的工作:

- 处理节点中的项;
- 处理左子树(递归调用);
- 处理右子树(递归调用)。

可以把遍历分成两个函数来完成: Traverse() 和 InOrder()。注意, InOrder() 函数处理左子树, 然后处理项, 最后处理右子树。这种遍历树的顺序是按字母排序进行。如果你有时间, 可以试试用不同的顺序, 比如, 项-左子树-右子树或者左子树-右子树-项, 看看会发生什么。

```

void Traverse(const Tree * ptree, void(*pfun)(Item item))
{
    if (ptree != NULL)
        InOrder(ptree->root, pfun);
}

static void InOrder(const Trnode * root, void(*pfun)(Item item))
{
    if (root != NULL)
    {
        InOrder(root->left, pfun);
        (*pfun)(root->item);
        InOrder(root->right, pfun);
    }
}

```

#### 5. 清空树

清空树基本上和遍历树的过程相同, 即清空树的代码也要访问每个节点, 而且要用 free() 函数释放内存。除此之外, 还要重置 Tree 类型结构的成员, 表明该树为空。DeleteAll() 函数负责处理 Tree 类型的结构, 把释放内存的任务交给 DeleteAllNode() 函数。DeleteAllNode() 与 InOrder() 函数的构造相同, 它储存了指针的值 root->right, 使其在释放根节点后仍然可用。下面是这两个函数的代码:

```

void DeleteAll(Tree * ptree)
{
    if (ptree != NULL)
        DeleteAllNodes(ptree->root);
    ptree->root = NULL;
    ptree->size = 0;
}
static void DeleteAllNodes(Trnode * root)
{
    Trnode * pright;
    if (root != NULL)
    {
        pright = root->right;
        DeleteAllNodes(root->left);
        free(root);
        DeleteAllNodes(pright);
    }
}

```

## 6. 完整的包

程序清单 17.11 演示了整个 tree.c 的代码。tree.h 和 tree.c 共同组成了树的程序包。

程序清单 17.11 tree.c 程序

---

```

/* tree.c -- 树的支持函数 */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

/* 局部数据类型 */
typedef struct pair {
    Trnode * parent;
    Trnode * child;
} Pair;

/* 局部函数的原型 */
static Trnode * MakeNode(const Item * pi);
static boolToLeft(const Item * i1, const Item * i2);
static bool ToRight(const Item * i1, const Item * i2);
static void AddNode(Trnode * new_node, Trnode * root);
static void InOrder(const Trnode * root, void(*pfun)(Item item));
static Pair SeekItem(const Item * pi, const Tree * ptree);
static void DeleteNode(Trnode **ptr);
static void DeleteAllNodes(Trnode * ptr);

/* 函数定义 */
void InitializeTree(Tree * ptree)
{
    ptree->root = NULL;
    ptree->size = 0;
}

bool TreeIsEmpty(const Tree * ptree)
{

```

```

if (ptree->root == NULL)
    return true;
else
    return false;
}

bool TreeIsFull(const Tree * ptree)
{
    if (ptree->size == MAXITEMS)
        return true;
    else
        return false;
}

int TreeItemCount(const Tree * ptree)
{
    return ptree->size;
}

bool AddItem(const Item * pi, Tree * ptree)
{
    Trnode * new_node;

    if (TreeIsFull(ptree))
    {
        fprintf(stderr, "Tree is full\n");
        return false; /* 提前返回 */
    }
    if (SeekItem(pi, ptree).child != NULL)
    {
        fprintf(stderr, "Attempted to add duplicate item\n");
        return false; /* 提前返回 */
    }
    new_node = MakeNode(pi); /* 指向新节点 */
    if (new_node == NULL)
    {
        fprintf(stderr, "Couldn't create node\n");
        return false; /* 提前返回 */
    }
    /* 成功创建了一个新节点 */
    ptree->size++;

    if (ptree->root == NULL) /* 情况 1：树为空 */
        ptree->root = new_node; /* 新节点为树的根节点 */
    else /* 情况 2：树不为空 */
        AddNode(new_node, ptree->root); /* 在树中添加新节点 */

    return true; /* 成功返回 */
}

bool InTree(const Item * pi, const Tree * ptree)
{
    return (SeekItem(pi, ptree).child == NULL) ? false : true;
}

```

```

bool DeleteItem(const Item * pi, Tree * ptree)
{
    Pair look;

    look = SeekItem(pi, ptree);
    if (look.child == NULL)
        return false;

    if (look.parent == NULL) /* 删除根节点项 */
        DeleteNode(&ptree->root);
    else if (look.parent->left == look.child)
        DeleteNode(&look.parent->left);
    else
        DeleteNode(&look.parent->right);
    ptree->size--;

    return true;
}

void Traverse(const Tree * ptree, void(*pfun)(Item item))
{
    if (ptree != NULL)
        InOrder(ptree->root, pfun);
}

void DeleteAll(Tree * ptree)
{
    if (ptree != NULL)
        DeleteAllNodes(ptree->root);
    ptree->root = NULL;
    ptree->size = 0;
}

/* 局部函数 */
static void InOrder(const Trnode * root, void(*pfun)(Item item))
{
    if (root != NULL)
    {
        InOrder(root->left, pfun);
        (*pfun)(root->item);
        InOrder(root->right, pfun);
    }
}

static void DeleteAllNodes(Trnode * root)
{
    Trnode * pright;

    if (root != NULL)
    {
        pright = root->right;
        DeleteAllNodes(root->left);
}

```

```

        free(root);
        DeleteAllNodes(pright);
    }

}

static void AddNode(Trnode * new_node, Trnode * root)
{
    if (ToLeft(&new_node->item, &root->item))
    {
        if (root->left == NULL) /* 空子树 */
            root->left = new_node; /* 把节点添加到此处 */
        else
            AddNode(new_node, root->left); /* 否则处理该子树 */
    }
    else if (ToRight(&new_node->item, &root->item))
    {
        if (root->right == NULL)
            root->right = new_node;
        else
            AddNode(new_node, root->right);
    }
    else /* 不允许有重复项 */
    {
        fprintf(stderr, "location error in AddNode()\n");
        exit(1);
    }
}

static bool ToLeft(const Item * i1, const Item * i2)
{
    int compl;

    if ((compl = strcmp(i1->petname, i2->petname)) < 0)
        return true;
    else if (compl == 0 && strcmp(i1->petkind, i2->petkind) < 0)
        return true;
    else
        return false;
}

static bool ToRight(const Item * i1, const Item * i2)
{
    int compl;

    if ((compl = strcmp(i1->petname, i2->petname)) > 0)
        return true;
    else if (compl == 0 &&
              strcmp(i1->petkind, i2->petkind) > 0)
        return true;
    else
        return false;
}

static Trnode * MakeNode(const Item * pi)
{

```

```

Trnode * new_node;

new_node = (Trnode *) malloc(sizeof(Trnode));
if (new_node != NULL)
{
    new_node->item = *pi;
    new_node->left = NULL;
    new_node->right = NULL;
}

return new_node;
}

static Pair SeekItem(const Item * pi, const Tree * ptree)
{
    Pair look;
    look.parent = NULL;
    look.child = ptree->root;

    if (look.child == NULL)
        return look; /* 提前返回 */

    while (look.child != NULL)
    {
        if (ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->left;
        }
        else if (ToRight(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->right;
        }
        else /* 如果前两种情况都不满足，则必定是相等的情况 */
            break; /* look.child 目标项的节点 */
    }

    return look; /* 成功返回 */
}

static void DeleteNode(Trnode **ptr)
/* ptr 是指向目标节点的父节点指针成员的地址 */
{
    Trnode * temp;

    if ((*ptr)->left == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->right;
        free(temp);
    }
    else if ((*ptr)->right == NULL)
    {

```

```

        temp = *ptr;
        *ptr = (*ptr)->left;
        free(temp);
    }
    else /* 被删除的节点有两个子节点 */
    {
        /* 找到重新连接右子树的位置 */
        for (temp = (*ptr)->left; temp->right != NULL; temp = temp->right)
            continue;
        temp->right = (*ptr)->right;
        temp = *ptr;
        *ptr = (*ptr)->left;
        free(temp);
    }
}

```

## 17.7.4 使用二叉树

现在，有了接口和函数的实现，就可以使用它们了。程序清单 17.12 中的程序以菜单的方式提供选择：向俱乐部成员花名册添加宠物、显示成员列表、报告成员数量、核实成员及退出。`main()` 函数很简单，主要提供程序的大纲。具体工作主要由支持函数来完成。

程序清单 17.12 petclub.c 程序

```

/* petclub.c -- 使用二叉查找数 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "tree.h"

char menu(void);
void addpet(Tree * pt);
void droppet(Tree * pt);
void showpets(const Tree * pt);
void findpet(const Tree * pt);
void printitem(Item item);
void uppercase(char * str);
char * s_gets(char * st, int n);

int main(void)
{
    Tree pets;
    char choice;

    InitializeTree(&pets);
    while ((choice = menu()) != 'q')
    {
        switch (choice)
        {
        case 'a': addpet(&pets);
                    break;
        case 'l': showpets(&pets);
                    break;
        }
    }
}

```

```

        case 'f': findpet(&pets);
            break;
        case 'n': printf("%d pets in club\n",
            TreeItemCount(&pets));
            break;
        case 'd': droppet(&pets);
            break;
        default: puts("Switching error");
    }
}

DeleteAll(&pets);
puts("Bye.");

return 0;
}

char menu(void)
{
    int ch;

    puts("Nerfville Pet Club Membership Program");
    puts("Enter the letter corresponding to your choice:");
    puts("a) add a pet      l) show list of pets");
    puts("n) number of pets  f) find pets");
    puts("d) delete a pet     q) quit");
    while ((ch = getchar()) != EOF)
    {
        while (getchar() != '\n') /* 处理输入行的剩余内容 */
            continue;
        ch = tolower(ch);
        if (strchr("alrfndq", ch) == NULL)
            puts("Please enter an a, l, f, n, d, or q:");
        else
            break;
    }
    if (ch == EOF) /* 使程序退出 */
        ch = 'q';
    return ch;
}

void addpet(Tree * pt)
{
    Item temp;

    if (TreeIsFull(pt))
        puts("No room in the club!");
    else
    {
        puts("Please enter name of pet:");
        s_gets(temp.petname, SLEN);
        puts("Please enter pet kind:");
        s_gets(temp.petkind, SLEN);
        uppercase(temp.petname);
        uppercase(temp.petkind);
        AddItem(&temp, pt);
    }
}

```

```

    }

}

void showpets(const Tree * pt)
{
    if (TreeIsEmpty(pt))
        puts("No entries!");
    else
        Traverse(pt, printitem);
}

void printitem(Item item)
{
    printf("Pet: %-19s Kind: %-19s\n", item.petname, item.petkind);
}

void findpet(const Tree * pt)
{
    Item temp;

    if (TreeIsEmpty(pt))
    {
        puts("No entries!");
        return;      /* 如果树为空, 则退出该函数 */
    }

    puts("Please enter name of pet you wish to find:");
    s_gets(temp.petname, SLEN);
    puts("Please enter pet kind:");
    s_gets(temp.petkind, SLEN);
    uppercase(temp.petname);
    uppercase(temp.petkind);
    printf("%s the %s ", temp.petname, temp.petkind);
    if (InTree(&temp, pt))
        printf("is a member.\n");
    else
        printf("is not a member.\n");
}

void droppet(Tree * pt)
{
    Item temp;

    if (TreeIsEmpty(pt))
    {
        puts("No entries!");
        return;      /* 如果树为空, 则退出该函数 */
    }

    puts("Please enter name of pet you wish to delete:");
    s_gets(temp.petname, SLEN);
    puts("Please enter pet kind:");
    s_gets(temp.petkind, SLEN);
    uppercase(temp.petname);
    uppercase(temp.petkind);
}

```

```

printf("%s the %s ", temp.petname, temp.petkind);
if (DeleteItem(&temp, pt))
    printf("is dropped from the club.\n");
else
    printf("is not a member.\n");
}

void uppercase(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
}
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; // 处理输入行的剩余内容
    }
    return ret_val;
}

```

该程序把所有字母都转换为大写字母，所以 SNUFFY、Snuffy 和 snuffy 都被视为相同。下面是该程序的一个运行示例：

```

Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet      1) show list of pets
n) number of pets   f) find pets
q) quit
a
Please enter name of pet:
Quincy
Please enter pet kind:
pig
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet      1) show list of pets
n) number of pets   f) find pets
q) quit
a
Please enter name of pet:
Bennie Haha
Please enter pet kind:

```

**parrot**

```

Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet      1) show list of pets
n) number of pets   f) find pets
q) quit
a
Please enter name of pet:
Hiram Jinx
Please enter pet kind:
domestic cat
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet      1) show list of pets
n) number of pets   f) find pets
q) quit
n
3 pets in club
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet      1) show list of pets
n) number of pets   f) find pets
q) quit
1
Pet: BENNIE HAHA      Kind: PARROT
Pet: HIRAM JINX       Kind: DOMESTIC CAT
Pet: QUINCY           Kind: PIG
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet      1) show list of pets
n) number of pets   f) find pets
q) quit
q
Bye.

```

## 17.7.5 树的思想

二叉查找树也有一些缺陷。例如，二叉查找树只有在满员（或平衡）时效率最高。假设要储存用户随机输入的单词。该树的外观应如图 17.12 所示。现在，假设用户按字母顺序输入数据，那么每个新节点应该被添加到右边，该树的外观应如图 17.16 所示。图 17.12 所示是平衡的树，图 17.16 所示是不平衡的树。查找这种树并不比查找链表要快。

避免串状树的方法之一是在创建树时多加注意。如果树或子树的一边或另一边太不平衡，就需要重新排列节点使之恢复平衡。与此类似，可能在进行删除操作后要重新排列树。俄国数学家 Adel'son-Vel'skii 和 Landis 发明了一种算法来解决这个问题。根据他们的算法创建的树称为 AVL 树。因为要重构，所以创建一个平衡的树所花费的时间更多，但是这样的树可以确保最大化搜索效率。

你可能需要一个能储存相同项的二叉查找树。例如，在分析一些文本时，统计某个单词在文本中出现的次数。一种方法是把 Item 定义成包含一个单词和一个数字的结构。第一次遇到一个单词时，将其添加到树中，并且该单词的数量加 1。下一次遇到同样的单词时，程序找到包含该单词的节点，并递增表示该单词数量的值。把基本二叉查找树修改成具有这一特性，不费多少工夫。

考虑 Nerfville 宠物俱乐部的示例，有另一种情况。示例中的树根据宠物的名字和种类进行排列，所以，可以把名为 Sam 的猫储存在一个节点中，把名为 Sam 的狗储存在另一节点中，把名为 Sam 的山羊储存在

第 3 个节点中。但是，不能储存两只名为 Sam 的猫。另一种方法是以名字来排序，但是这样做只能储存一个名为 Sam 的宠物。还需要把 Item 定义成多个结构，而不是一个结构。第一次出现 Sally 时，程序创建一个新的节点，并创建一个新的列表，然后把 Sally 及其种类添加到列表中。下一次出现 Sally 时，程序将定位到之前储存 Sally 的节点，并把新的数据添加到结构列表中。

### 提示 插件库

读者可能意识到实现一个像链表或树这样的 ADT 比较困难，很容易犯错。插件库提供了一种可选的方法：让其他人来完成这些工作和测试。在学完本章这两个相对简单的例子后，读者应该能很好地理解和认识这样的库。

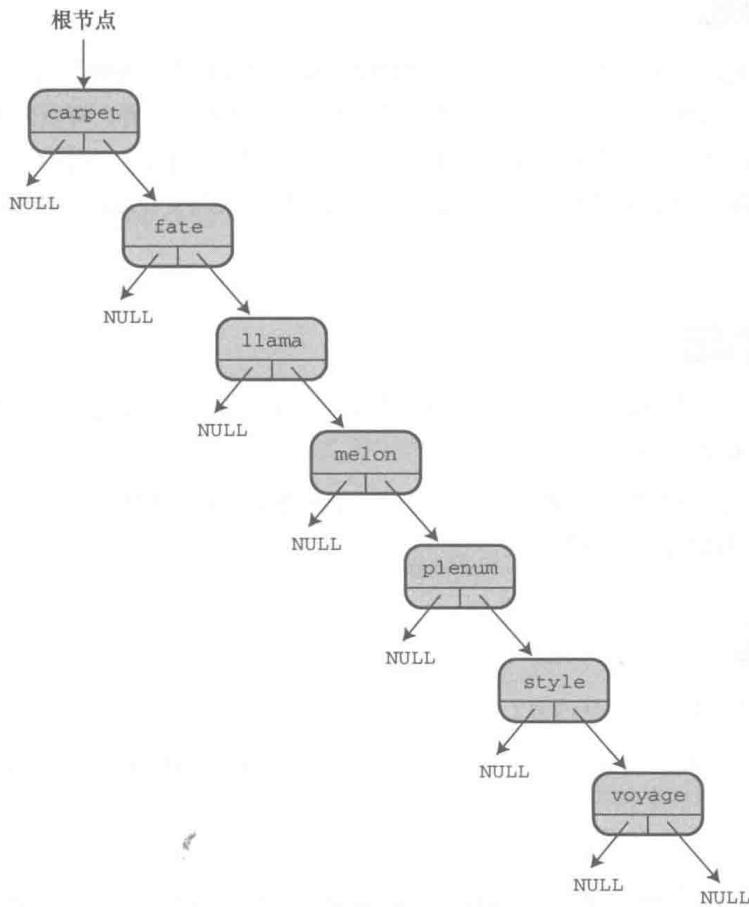


图 17.16 不平衡的二叉查找树

## 17.8 其他说明

本书中，我们涵盖了 C 语言的基本特性，但是只是简要介绍了库。ANSI C 库中包含多种有用的函数。绝大部分实现都针对特定的系统提供扩展库。基于 Windows 的编译器支持 Windows 图形接口。Macintosh C 编译器提供访问 Macintosh 工具箱的函数，以便编写具有标准 Macintosh 接口或 iOS 系统的程序产品，如 iPhone 或 iPad。与此类似，还有一些工具用于创建 Linux 程序的图形接口。花时间查看你的系统提供什么。如果没有你想要的工具，就自己编写函数。这是 C 的一部分。如果认为自己能编写一个更好的（如，输入函数），那就去做！随着你不断练习并提高自己的编程技术，会从一名新手成为经验丰富的资深程序员。

如果对链表、队列和树的相关概念感兴趣或觉得很有用，可以阅读其他相关的书籍，学习高级编程技巧。计算机科学家在开发和分析算法以及如何表示数据方面投入了大量的时间和精力。也许你会发现已经有人开发了你正需要的工具。

学会 C 语言后，你可能想研究 C++、Objective C 或 Java。这些都是以 C 为基础的面向对象(*object-oriented*)语言。C 已经涵盖了从简单的 `char` 类型变量到大型且复杂的结构在内的数据对象。面向对象语言更进一步发展了对象的观点。例如，对象的性质不仅包括它所储存的信息类型，而且还包括了对其进行的操作类型。本章介绍的 ADT 就遵循了这种模式。而且，对象可以继承其他对象的属性。OOP 提供比 C 更高级的抽象，很适合编写大型程序。

请参阅附录 B 中的参考资料 I “补充阅读” 中找到你感兴趣的书籍。

## 17.9 关键概念

一种数据类型通过以下几点来表征：如何构建数据、如何储存数据、有哪些可能的操作。抽象数据类型(ADT)以抽象的方式指定构成某种类型特征的属性和操作。从概念上看，可以分两步把 ADT 翻译成一种特定的编程语言。第 1 步是定义编程接口。在 C 中，通过使用头文件定义类型名，并提供与允许的操作相应的函数原型来实现。第 2 步是实现接口。在 C 中，可以用源代码文件提供与函数原型相应的函数定义来实现。

## 17.10 本章小结

链表、队列和二叉树是 ADT 在计算机程序设计中常用的示例。通常用动态内存分配和链式结构来实现它们，但有时用数组来实现会更好。

当使用一种特定类型（如队列或树）进行编程时，要根据类型接口来编写程序。这样，在修改或改进实现时就不用更改使用接口的那些程序。

## 17.11 复习题

1. 定义一种数据类型涉及哪些内容？
2. 为什么程序清单 17.2 只能沿一个方向遍历链表？如何修改 `struct film` 定义才能沿两个方向遍历链表？
3. 什么是 ADT？
4. `QueueIsEmpty()` 函数接受一个指向 `queue` 结构的指针作为参数，但是也可以将其编写成接受一个 `queue` 结构作为参数。这两种方式各有什么优缺点？
5. 栈(`stack`)是链表系列的另一种数据形式。在栈中，只能在链表的一端添加和删除项，项被“压入”栈和“弹出”栈。因此，栈是一种 LIFO(即后进先出 *last in, first out*)结构。
  - a. 设计一个栈 ADT
  - b. 为栈设计一个 C 编程接口，例如 `stack.h` 头文件
6. 在一个含有 3 个项的分类列表中，判断一个特定项是否在该列表中，用顺序查找和二叉查找方法分别需要最多多少次？当列表中有 1023 个项时分别是多少次？65535 个项是分别是多少次？
7. 假设一个程序用本章介绍的算法构造了一个储存单词的二叉查找树。假设根据下面所列的顺序输入

单词, 请画出每种情况的树:

- a. nice food roam dodge gate office wave
  - b. wave roam office nice gate food dodge
  - c. food dodge roam wave office gate nice
  - d. nice roam office food wave gate dodge
8. 考虑复习题 7 构造的二叉树, 根据本章的算法, 删除单词 food 之后, 各树是什么样子?

## 17.12 编程练习

1. 修改程序清单 17.2, 让该程序既能正序也能逆序显示电影列表。一种方法是修改链表的定义, 可以双向遍历链表。另一种方法是用递归。

2. 假设 list.h (程序清单 17.3) 使用下面的 list 定义:

```
typedef struct list
{
    Node * head; /* 指向 list 的开头 */
    Node * end; /* 指向 list 的末尾 */
} List;
```

重写 list.c (程序清单 17.5) 中的函数以适应新的定义, 并通过 films.c (程序清单 17.4) 测试最终的代码。

3. 假设 list.h (程序清单 17.3) 使用下面的 list 定义:

```
#define MAXSIZE 100
typedef struct list
{
    Item entries[MAXSIZE]; /* 内含项的数组 */
    int items; /* list 中的项数 */
} List;
```

重写 list.c (程序清单 17.5) 中的函数以适应新的定义, 并通过 films.c (程序清单 17.4) 测试最终的代码。

4. 重写 mall.c (程序清单 17.7), 用两个队列模拟两个摊位。

5. 编写一个程序, 提示用户输入一个字符串。然后该程序把该字符串的字符逐个压入一个栈 (参见复习题 5), 然后从栈中弹出这些字符, 并显示它们。结果显示为该字符串的逆序。

6. 编写一个函数接受 3 个参数: 一个数组名 (内含已排序的整数)、该数组的元素个数和待查找的整数。如果待查找的整数在数组中, 那么该函数返回 1; 如果该数不在数组中, 该函数则返回 0。用二分查找法实现。

7. 编写一个程序, 打开和读取一个文本文件, 并统计文件中每个单词出现的次数。用改进的二叉查找树储存单词及其出现的次数。程序在读入文件后, 会提供一个有 3 个选项的菜单。第 1 个选项是列出所有的单词和出现的次数。第 2 个选项是让用户输入一个单词, 程序报告该单词在文件中出现的次数。第 3 个选项是退出。

8. 修改宠物俱乐部程序, 把所有同名的宠物都储存在同一个节点中。当用户选择查找宠物时, 程序应询问用户该宠物的名字, 然后列出该名字的所有宠物 (及其种类)。



# 附录 A

## 复习题答案

### A.1 第1章复习题答案

1. 完美的可移植程序是，其源代码无需修改就能在不同计算机系统中成功编译的程序。
2. 源代码文件包含程序员使用的任何编程语言编写的代码。目标代码文件包含机器语言代码，它不必是完整的程序代码。可执行文件包含组成可执行程序的完整机器语言代码。
3. (1) 定义程序目标；(2) 设计程序；(3) 编写程序；(4) 编译程序；(5) 运行程序；  
(6) 测试和调试程序；(7) 维护和修改程序。
4. 编译器把源代码（如，用 C 语言编写的代码）翻译成等价的机器语言代码（也叫作目标代码）。
5. 链接器把编译器翻译好的源代码以及库代码和启动代码组合起来，生成一个可执行程序。

### A.2 第2章复习题答案

1. 它们都叫作函数。
2. 语法错误违反了组成语句或程序的规则。这是一个有语法错误的英文例子：Me speak English good..  
这是一个有语法错误的 C 语言例子：printf"Where are the parentheses?";。
3. 语义错误是指含义错误。这是一个有语义错误的英文例子：This sentence is excellent Czech.<sup>1</sup>。这是一个有语义错误的 C 语言例子： thrice\_n = 3 + n;<sup>2</sup>。
4. 第 1 行：以一个#开始；studio.h 应改成 stdio.h；然后用一对尖括号把 stdio.h 括起来。  
第 2 行：把{}改成()；注释末尾把/\*改成\*/。  
第 3 行：把(改成{  
第 4 行：int s 末尾加上一个分号。  
第 5 行没问题。  
第 6 行：把:=改成，赋值用=，而不是用:=（这说明 Indiana Sloth 了解 Pascal）。另外，用于赋值的值 56 也不对，一年有 52 周，不是 56 周。  
第 7 行应该是：printf("There are %d weeks in a year.\n", s);  
第 9 行：原程序中没有第 9 行，应该在该行加上一个右花括号}。  
修改后的程序如下：

```
#include <stdio.h>
int main(void) /* this prints the number of weeks in a year */
{
```

<sup>1</sup> 这句英文翻译成中文是“这句话是出色的捷克人”。显然不知所云，这就是语言中的语义错误。——译者注

<sup>2</sup> thrice\_n 本应表示 n 的 3 倍，但是 3 + n 表示的并不是 n 的 3 倍，应该用 3\*n 来表示。——译者注

```

int s;

s = 52;
printf("There are %d weeks in a year.\n", s);
return 0;
}

```

5. a. Baa Baa Black Sheep. Have you any wool? (注意, Sheep. 和 Have 之间没有空格)  
 b. Begone!  
     O creature of lard!  
 c. What?  
     No/nfish?  
     (注意斜杠/和反斜杠\的效果不同, /只是一个普通的字符, 原样打印)  
 d. 2 + 2 = 4  
     (注意, 每个%d 与列表中的值相对应。还要注意, +的意思是加法, 可以在 printf() 语句内部计算)
6. 关键字是 int 和 char (main 是一个函数名; function 是函数的意思; =是一个运算符)。
7. printf("There were %d words and %d lines.\n", words, lines);
8. 执行完第 7 行后, a 是 5, b 是 2。执行完第 8 行后, a 和 b 都是 5。执行完第 9 行后, a 和 b 仍然是 5 (注意, a 不会是 2, 因为在执行 a = b; 时, b 的值已经被改为 5)。
9. 执行完第 7 行后, x 是 10, b 是 5。执行完第 8 行后, x 是 10, y 是 15。执行完第 9 行后, x 是 150, y 是 15。

### A.3 第3章复习题答案

1. a. int 类型, 也可以是 short 类型或 unsigned short 类型。人口数是一个整数。  
 b. float 类型, 价格通常不是一个整数 (也可以使用 double 类型, 但实际上不需要那么高的精度)。  
 c. char 类型。  
 d. int 类型, 也可以是 unsigned 类型。
2. 原因之一: 在系统中要表示的数超过了 int 可表示的范围, 这时要使用 long 类型。原因之二: 如果要处理更大的值, 那么使用一种在所有系统上都保证至少是 32 位的类型, 可提高程序的可移植性。
3. 如果要正好获得 32 位的整数, 可以使用 int32\_t 类型。要获得可储存至少 32 位整数的最小类型, 可以使用 int\_least32\_t 类型。如果要为 32 位整数提供最快的计算速度, 可以选择 int\_fast32\_t 类型 (假设你的系统已定义了上述类型)。
4. a. char 类型常量 (但是储存为 int 类型)  
 b. int 类型常量  
 c. double 类型常量  
 d. unsigned int 类型常量, 十六进制格式  
 e. double 类型常量
5. 第 1 行: 应该是#include <stdio.h>  
 第 2 行: 应该是 int main(void)

第3行：把(改为 {

第4行：g 和 h 之间的;改成,

第5行：没问题

第6行：没问题

第7行：虽然这数字比较大，但在 e 前面应至少有一个数字，如 1e21 或 1.0e21 都可以。

第8行：没问题，至少没有语法问题。

第9行：把)改成}

除此之外，还缺少一些内容。首先，没有给 rate 变量赋值；其次未使用 h 变量；而且程序不会报告计算结果。虽然这些错误不会影响程序的运行（编译器可能给出变量未被使用的警告），但是它们确实与程序设计的初衷不符合。另外，在该程序的末尾应该有一个 return 语句。

下面是一个正确的版本，仅供参考：

```
#include <stdio.h>
int main(void)
{
    float g, h;
    float tax, rate;

    rate = 0.08;
    g = 1.0e5;
    tax = rate*g;
    h = g + tax;
    printf("You owe $%f plus $%f in taxes for a total of $%f.\n", g, tax, h);
    return 0;
}
```

6.

| 常量        | 类型              | 转换说明 (%转换字符) |
|-----------|-----------------|--------------|
| 12        | int             | %d           |
| 0X3       | unsigned int    | %#X          |
| 'C'       | char (实际上是 int) | %c           |
| 2.34E07   | double          | %e           |
| '\040'    | char (实际上是 int) | %c           |
| 7.0       | double          | %f           |
| 6L        | long            | %ld          |
| 6.0f      | float           | %f           |
| 0x5.b6p12 | float           | %a           |

7.

| 常量      | 类型              | 转换说明 (%转换字符) |
|---------|-----------------|--------------|
| 012     | unsigned int    | %#o          |
| 2.9e05L | long double     | %Le          |
| 's'     | char (实际上是 int) | %c           |
| 100000  | long            | %ld          |
| '\n'    | char (实际上是 int) | %c           |
| 20.0f   | float           | %f           |
| 0x44    | unsigned int    | %x           |
| -40     | int             | %d           |

8. 

```
printf("The odds against the %d were %ld to 1.\n", imate, shot);
printf("A score of %f is not an %c grade.\n", log, grade);
```
9. 

```
ch = '\r';
ch = 13;
ch = '\015';
ch = '\xd1'
```
10. 最前面缺少一行(第0行): #include <stdio.h>

第1行: 使用/\*和\*/把注释括起来, 或者在注释前面使用//。

第3行: int cows, legs;

第4行: country? \n");

第5行: 把%c改为%d, 把legs改为&legs。

第7行: 把%f改为%d。

另外, 在程序末尾还要加上return语句。

下面是修改后的版本:

```
#include <stdio.h>
int main(void) /* this program is perfect */
{
    int cows, legs;
    printf("How many cow legs did you count?\n");
    scanf("%d", &legs);
    cows = legs / 4;
    printf("That implies there are %d cows.\n", cows);
    return 0;
}
```

11. a. 换行字符
- b. 反斜杠字符
- c. 双引号字符
- d. 制表字符

## A.4 第4章复习题答案

1. 程序不能正常运行。第一个scanf()语句只读取用户输入的名, 而用户输入的姓仍留在输入缓冲区中(缓冲区是用于储存输入的临时存储区)。下一条scanf()语句在输入缓冲区查找重量时, 从上次读入结束的地方开始读取。这样就把留在缓冲区的姓作为体重来读取, 导致scanf()读取失败。另一方面, 如果在要求输入姓名时输入Lasha 144, 那么程序会把144作为用户的体重(虽然用户是在程序提示输入体重之前输入了144)。
2. a. He sold the painting for \$234.50.
- b. Hi! (注意, 第1个字符是字符常量; 第2个字符由十进制整数转换而来; 第3个字符是八进制字符常量的ASCII表示)
- c. His Hamlet was funny without being vulgar.  
has 42 characters.
- d. Is 1.20e+003 the same as 1201.00?

3. 在这条语句中使用\": printf("\'%s\'\nhas %d characters.\n", Q, strlen(Q));

4. 下面是修改后的程序:

```
#include <stdio.h>      /* 别忘了要包含合适的头文件 */
#define B "booboo"        /* 添加#、双引号 */
#define X 10              /* 添加# */
int main(void)          /* 不是 main(int) */
{
    int age;
    int xp;             /* 声明所有的变量 */
    char name[40];       /* 把 name 声明为数组 */

    printf("Please enter your first name.\n"); /* 添加\n, 提高可读性 */
    scanf("%s", name);
    printf("All right, %s, what's your age?\n", name); /* %s 用于打印字符串 */
    scanf("%d", &age); /* 把%f改成%d, 把age改成&age */
    xp = age + X;
    printf("That's a %s! You must be at least %d.\n", B, xp);
    return 0; /* 不是 rerun */
}
```

5. 记住, 要打印%必须用%%:

```
printf("This copy of \"%s\" sells for $%0.2f.\n", BOOK, cost);
printf("That is %0.0f%% of list.\n", percent);
```

6. a. %d

b. %4X

c. %10.3f

d. %12.2e

e. %-30s

7. a. %15lu

b. %#4x

c. %-12.2E

d. %+10.3f

e. %8.8s

8. a. %6.4d

b. %\*o

c. %2c

d. %+0.2f

e. %-7.5s

9. a. int dalmations;

```
scanf("%d", &dalmations);
```

```
b. float kgs, share;
    scanf("%f%f", &kgs, &share);
```

(注意：对于本题的输入，可以使用转换字符 e、f 和 g。另外，除了%c之外，在%和转换字符之间加空格不会影响最终的结果)

```
c. char pasta[20];
    scanf("%s", pasta);
d. char action[20];
    int value;
    scanf("%s %d", action, &value);
e. int value;
    scanf("%*s %d", &value);
```

10. 空白包括空格、制表符和换行符。C 语言使用空白分隔记号。`scanf()` 使用空白分隔连续的输入项。

11. %z 中的 z 是修饰符，不是转换字符，所以要在修饰符后面加上一个它修饰的转换字符。可以使用%zd 打印十进制数，或用不同的说明符打印不同进制的数，例如，%zx 打印十六进制的数。

12. 可以分别把(和)替换成{和}。但是预处理器无法区分哪些圆括号应替换成花括号，哪些圆括号不能替换成花括号。因此，

```
#define ( {
#define ) }
int main(void)
{
    printf("Hello, O Great One!\n");
}
```

将变成：

```
int main(void)
{
    printf{"Hello, O Great One!\n"};
```

## A.5 第5章复习题答案

1. a. 30
  - b. 27 (不是 3)。 $(12+6)/(2*3)$  得 3。
  - c.  $x = 1, y = 1$  (整数除法)。
  - d.  $x = 3$  (整数除法),  $y = 9$ 。
2. a. 6 (由 3 + 3.3 截断而来)
  - b. 52
  - c. 0 (0 \* 22.0 的结果)
  - d. 13 (66.0 / 5 或 13.2, 然后把结果赋给 int 类型变量)
3. a. 37.5 (7.5 \* 5.0 的结果)
  - b. 1.5 (30.0 / 20.0 的结果)

- c. 35 (7 \* 5 的结果)
  - d. 37 (150 / 4 的结果)
  - e. 37.5 (7.5 \* 5 的结果)
  - f. 35.0 (7 \* 5.0 的结果)
4. 第0行：应增加一行`#include <stdio.h>`。

第3行：末尾用分号，而不是逗号。

第6行：while语句创建了一个无限循环。因为i的值始终为1，所以它总是小于30。推测一下，应该是想写`while(i++ < 30)`。

第6~8行：这样的缩进布局不能使第7行和第8行组成一个代码块。由于没有用花括号括起来，while循环只包括第7行，所以要添加花括号。

第7行：因为1和i都是整数，所以当i为1时，除法的结果是1；当i为更大的数时，除法结果为0。用`n = 1.0/i`，i在除法运算之前会被转换为浮点数，这样就能得到非零值。

第8行：在格式化字符串中没有换行符(\n)，这导致数字被打印成一行。

第10行：应该是`return 0;`

下面是正确的版本：

```
#include <stdio.h>
int main(void)
{
    int i = 1;
    float n;
    printf("Watch out! Here come a bunch of fractions!\n");
    while (i++ < 30)
    {
        n = 1.0/i;
        printf(" %f\n", n);
    }
    printf("That's all, folks!\n");
    return 0;
}
```

5. 这个版本最大的问题是测试条件(sec是否大于0?)和scanf()语句获取sec变量的值之间的关系。具体地说，第一次测试时，程序尚未获得sec的值，用来与0作比较的是正好在sec变量内存位置上的一个垃圾值。一个比较笨拙的方法是初始化sec(如，初始化为1)。这样就可通过第一次测试。不过，还有另一个问题。当最后输入0结束程序时，在循环结束之前不会检查sec，所以0也被打印了出来。因此，更好的方法是在while测试之前使用scanf()语句。可以这样修改：

```
scanf("%d", &sec);
while ( sec > 0 ) {
    min = sec/S_TO_M;
    left = sec % S_TO_M;
    printf("%d sec is %d min, %d sec. \n", sec, min, left);
    printf("Next input?\n");
    scanf("%d", &sec);
}
```

while循环第一轮迭代使用的是scanf()在循环外面获取的值。因此，在while循环的末尾还要使用一次scanf()语句。这是处理类似问题的常用方法。

## 6. 下面是该程序的输出:

```
%s! C is cool!
! C is cool!
11
11
12
11
```

解释一下。第 1 个 printf() 语句与下面的语句相同:

```
printf("%s! C is cool!\n", "%s! C is cool!\n");
```

第 2 个 printf() 语句首先把 num 递增为 11, 然后打印该值。第 3 个 printf() 语句打印 num 的值(值为 11)。第 4 个 printf() 语句打印 n 当前的值(仍为 12), 然后将其递减为 11。最后一个 printf() 语句打印 num 的当前值(值为 11)。

## 7. 下面是该程序的输出:

```
SOS:4 4.00
```

表达式 c1 -c2 的值和'S' - '0'的值相同(其对应的 ASCII 值是 83 - 79)。

## 8. 把 1~10 打印在一行, 每个数字占 5 列宽度, 然后开始新的一行:

```
1 2 3 4 5 6 7 8 9 10
```

## 9. 下面是一个参考程序, 假定字母连续编码, 与 ASCII 中的情况一样。

```
#include <stdio.h>

int main(void)
{
    char ch = 'a';
    while (ch <= 'g')
        printf("%5c", ch++);
    printf("\n");
    return 0;
}
```

## 10. 下面是每个部分的输出:

a. 1 2

注意, 先递增 x 的值再比较。光标仍留在同一行。

b. 101  
102  
103  
104

注意, 这次 x 先比较后递增。在示例 a 和 b 中, x 都是在先递增后打印。另外还要注意, 虽然第 2 个 printf() 语句缩进了, 但是这并不意味着它是 while 循环的一部分。因此, 在 while 循环结束后, 才会调用一次该 printf() 语句。

c. stuvw

该例中, 在第 1 次调用 printf() 语句后才会递增 ch。

## 11. 这个程序有点问题。while 循环没有用花括号把两个缩进的语句括起来, 只有 printf() 是循环的一部分, 所以该程序一直重复打印消息 COMPUTER BYTES DOG, 直到强行关闭程序为止。

12. a.  $x = x + 10;$ 

b.  $x++;$  or  $++x;$  or  $x = x + 1;$   
c.  $c = 2 * (a + b);$

- d.  $c = a + 2 * b;$   
 13. a.  $x--;$  or  $--x;$  or  $x = x - 1;$   
 b.  $m = n \% k;$   
 c.  $p = q / (b - a);$   
 d.  $x = (a + b) / (c * d);$

## A.6 第6章复习题答案

1. 2, 7, 70, 64, 8, 2。

2. 该循环的输出是：

36 18 9 4 2 1

如果 value 是 double 类型，即使 value 小于 1，循环的测试条件仍然为真。循环将一直执行，直到浮点数下溢生成 0 为止。另外，value 是 double 类型时，%3d 转换说明也不正确。

3. a.  $x > 5$

b. `scanf("%lf", &x) != 1`

c.  $x == 5$

4. a. `scanf("%d", &x) == 1`

b.  $x != 5$

c.  $x >= 20$

5. 第 4 行：应该是 `list[10]`。

第 6 行：逗号改为分号。i 的范围应该是 0~9，不是 1~10。

第 9 行：逗号改为分号。`>=` 改成 `<=`，否则，当 i 等于 1 时，该循环将成为无限循环。

第 10 行：在第 10 行和第 11 行之间少了一个右花括号。该右花括号与第 7 行的左花括号配对，形成一个 for 循环块。然后在这个右花括号与最后一个右花括号之间，少了一行 `return 0;`。

下面是一个正确的版本：

```
#include <stdio.h>
int main(void)
{
    /* 第 3 行 */
    int i, j, list(10); /* 第 4 行 */
    /* 第 5 行 */
    for (i = 1, i <= 10, i++) /* 第 6 行 */
    { /* 第 7 行 */
        list[i] = 2*i + 3; /* 第 8 行 */
        for (j = 1, j >= i, j++) /* 第 9 行 */
            printf(" %d", list[j]); /* 第 10 行 */
        printf("\n"); /* 第 11 行 */
    }
    return 0;
}
```

6. 下面是一种方法：

```
#include <stdio.h>
int main(void)
```

```

{
    int col, row;

    for (row = 1; row <= 4; row++)
    {
        for (col = 1; col <= 8; col++)
            printf("$");
        printf("\n");
    }
    return 0;
}

```

7. a. Hi! Hi! Hi! Bye! Bye! Bye! Bye!  
 b. ACMG (因为代码中把 int 类型值与 char 类型值相加, 编译器可能警告会损失有效数字)

8. a. Go west, youn  
 b. Hp!xftu-!zpvo  
 c. Go west, young  
 d. \$o west, youn

9. 其输入如下:

```

31|32|33|30|31|32|33|
***  

1  

5  

9  

13  

***  

2 6  

4 8  

8 10

```

```

***  

=====  

=====  

=====  

====  

==

```

10. a. mint  
 b. 10 个元素  
 c. double 类型的值  
 d. 第 ii 行正确, mint[2] 是 double 类型的值, &mingt[2] 是它在内存中的位置。

11. 因为第 1 个元素的索引是 0, 所以循环的范围应该是 0~SIZE - 1, 而不是 1~SIZE。但是, 如果只是这样更改会导致赋给第 1 个元素的值是 0, 不是 2。所以, 应重写这个循环:

```

for (index = 0; index < SIZE; index++)
    by_twos[index] = 2 * (index + 1);

```

与此类似, 第 2 个循环的范围也要更改。另外, 应该在数组名后面使用数组索引:

```

for( index = 0; index < SIZE; index++)
    printf("%d ", by_twos[index]);

```

错误的循环条件会成为程序的定时炸弹。程序可能开始运行良好，但是由于数据被放在错误的位置，可能在某一时刻导致程序不能正常工作。

12. 该函数应声明为返回类型为 long，并包含一个返回 long 类型值的 return 语句。
13. 把 num 的类型强制转换成 long 类型，确保计算使用 long 类型而不是 int 类型。在 int 为 16 位的系统中，两个 int 类型值的乘积在返回之前会被截断为一个 int 类型的值，这可能会丢失数据。

```
long square(int num)
{
    return ((long) num) * num;
}
```

14. 输出如下：

```
1: Hi!
k = 1
k is 1 in the loop
Now k is 3
k = 3
k is 3 in the loop
Now k is 5
k = 5
k is 5 in the loop
Now k is 7
k = 7
```

## A.7 第7章复习题答案

1. b 是 true。
2. a. number >= 90 && number < 100  
 b. ch != 'q' && ch != 'k'  
 c. (number >= 1 && number <= 9) && number != 5  
 d. 可以写成 !(number >= 1 && number <= 9)，但是 number < 1 || number > 9 更好理解。
3. 第 5 行：应该是 scanf("%d %d", &weight, &height);。不要忘记 scanf() 中要用&。另外，这一行前面应该有提示用户输入的语句。

第 9 行：测试条件中要表达的意思是 (height < 72 && height > 64)。根据前面第 7 行中的测试条件，能到第 9 行的 height 一定小于 72，所以，只需要用表达式 (height > 64) 即可。但是，第 6 行中已经包含了 height > 64 这个条件，所以这里完全不必再判断，if else 应改成 else。

第 11 行：条件冗余。第 2 个表达式 (weight 不小于或不等于 300) 和第 1 个表达式含义相同。只需用一个简单的表达式 (weight > 300) 即可。但是，问题不止于此。第 11 行是一个错误的 if，这行的 else if 与第 6 行的 if 匹配。但是，根据 if 的“最接近规则”，该 else if 应该与第 9 行的 else if 匹配。因此，在 weight 小于 100 且大于或等于 64 时到达第 11 行，而此时 weight 不可能超过 300。

第 7 行~第 10 行：应该用花括号括起来。这样第 11 行就确定与第 6 行匹配。但是，如果把第 9 行的 else if 替换成简单的 else，就不需要使用花括号。

第 13 行：应简化成 if (height > 48)。实际上，完全可以省略这一行。因为第 12 行已经测

试过该条件。

下面是修改后的版本：

```
#include <stdio.h>
int main(void)
{
    int weight, height; /* weight in lbs, height in inches */

    printf("Enter your weight in pounds and ");
    printf("your height in inches.\n");
    scanf("%d %d", &weight, &height);
    if (weight < 100 && height > 64)
        if (height >= 72)
            printf("You are very tall for your weight.\n");
        else
            printf("You are tall for your weight.\n");
    else if (weight > 300 && height < 48)
        printf(" You are quite short for your weight.\n");
    else
        printf("Your weight is ideal.\n");

    return 0;
}
```

4. a. 1。5 确实大于 2，表达式为真，即是 1。

b. 0。3 比 2 大，表达式为假，即是 0。

c. 1。如果第 1 个表达式为假，则第 2 个表达式为真，反之亦然。所以，只要一个表达式为真，整个表达式的结果即为真。

d. 6。因为  $6 > 2$  为真，所以  $(6 > 2)$  的值为 1。

e. 10。因为测试条件为真。

f. 0。如果  $x > y$  为真，表达式的值就是  $y > x$ ，这种情况下它为假或 0。如果  $x > y$  为假，那么表达式的值就是  $x > y$ ，这种情况下为假。

5. 该程序打印以下内容：

```
*#%*#%$#%*#%$#%*#%$#%*#%$#%*#%
```

无论怎样缩排，每次循环都会打印#，因为缩排并不能让 `putchar('#');` 成为 `if else` 复合语句的一部分。

6. 程序打印以下内容：

```
fat hat cat Oh no!
hat cat Oh no!
cat Oh no!
```

7. 第 5 行～第 7 行的注释要以 \*/ 结尾，或者把注释开头的 /\*换成 //。表达式 ' $a' \leq ch \geq 'z'$ ' 应替换成  $ch \geq 'a' \&& ch \leq 'z'$ '。

或者，包含 `ctype.h` 并使用 `islower()`，这种方法更简单，而且可移植性更高。顺带一提，虽然从 C 的语法方面看，' $a' \leq ch \geq 'z'$ ' 是有效的表达式，但是它的含义不明。因为关系运算符从左往右结合，该表达式被解释成  $('a' \leq ch) \geq 'z'$ 。圆括号中的表达式的值不是 1 就是 0（真或假），然后判断该值是否大于或等于 ' $z$ ' 的数值码。1 和 0 都不满足测试条件，所以整个表达式恒为 0（假）。在第 2 个测试表达式中，应该把 || 改成 &&。另外，虽然 `!(ch < 'A')` 是有

效的表达式，而且含义也正确，但是用 `ch >= 'A'` 更简单。这一行的'z'后面应该有两个圆括号。更简单的方法是使用 `isupper()`。在 `uc++;` 前面应该加一行 `else`。否则，每输入一个字符，`uc` 都会递增 1。另外，在 `printf()` 语句中的格式化字符串应该用双引号括起来。下面是修改后的版本：

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char ch;
    int lc = 0; /*统计小写字母*/
    int uc = 0; /*统计大写字母*/
    int oc = 0; /*统计其他字母*/

    while ((ch = getchar()) != '#')
    {
        if (islower(ch))
            lc++;
        else if (isupper(ch))
            uc++;
        else
            oc++;
    }
    printf("%d lowercase, %d uppercase, %d other", lc, uc, oc);
    return 0;
}
```

8. 该程序将不停重复打印下面一行：

```
You are 65. Here is your gold watch.
```

问题出在这一行：`if (age = 65)`

这行代码把 `age` 设置为 65，使得每次迭代的测试条件都为真。

9. 下面是根据给定输入的运行结果：

```
q
Step 1
Step 2
Step 3
c
Step 1
h
Step 1
Step 3
b
Step 1
Done
```

注意，`b` 和`#`都可以结束循环。但是输入 `b` 会使得程序打印 `step 1`，而输入`#`则不会。

10. 下面是一种解决方案：

```
#include <stdio.h>
int main(void)
{
    char ch;
    while ((ch = getchar()) != '#')
```

```

{
    if (ch != '\n')
    {
        printf("Step 1\n");
        if (ch == 'b')
            break;
        else if (ch != 'c')
        {
            if (ch != 'h')
                printf("Step 2\n");
            printf("Step 3\n");
        }
    }
    printf("Done\n");
    return 0;
}

```

## A.8 第 8 章复习题答案

- 表达式 `putchar(getchar())` 使程序读取下一个输入字符并打印出来。`getchar()` 的返回值是 `putchar()` 的参数。但 `getchar(putchar())` 是无效的表达式，因为 `getchar()` 不需要参数，而 `putchar()` 需要一个参数。
- a. 显示字符 H。  
b. 如果系统使用 ASCII，则发出一声警报。  
c. 把光标移至下一行的开始。  
d. 退后一格。
- `count <essay>essayct` 或者 `count >essayct <essay`
- 都不是有效的命令。
- `EOF` 是由 `getchar()` 和 `scanf()` 返回的信号（一个特殊值），表明函数检测到文件结尾。
- a. 输出是：If you qu

注意，字符 I 与字符 i 不同。还要注意，没有打印 i，因为循环在检测到 i 之后就退出了。

b. 如果系统使用 ASCII，输出是：HJacrthjacrt

`while` 的第 1 轮迭代中，为 `ch` 读取的值是 H。第 1 个 `putchar()` 语句使用的 `ch` 的值是 H，打印完毕后，`ch` 的值加 1（现在是 `ch` 的值是 I）。然后到第 2 个 `putchar()` 语句，因为是 `++ch`，所以先递增 `ch`（现在 `ch` 的值是 J）再打印它的值。然后进入下一轮迭代，读取输入序列中的下一个字符 (a)，重复以上步骤。需要注意的是，两个递增运算符只在 `ch` 被赋值后影响它的值，不会让程序在输入序列中移动。

- C 的标准 I/O 库把不同的文件映射为统一的流来统一处理。
- 数值输入会跳过空格和换行符，但是字符输入不会。假设有下面的代码：

```

int score;
char grade;
printf("Enter the score.\n");
scanf("%s", %score);

```

```
printf("Enter the letter grade.\n");
grade = getchar();
```

如果输入分数 98，然后按下 Enter 键把分数发送给程序，其实还发送了一个换行符。这个换行符会留在输入序列中，成为下一个读取的值（grade）。如果在字符输入之前输入了数字，就应该在处理字符输入之前添加删除换行符的代码。

## A.9 第9章复习题答案

1. 形式参数是定义在被调函数中的变量。实际参数是出现在函数调用中的值，该值被赋给形式参数。可以把实际参数视为在函数调用时初始化形式参数的值。

2. a. void donut(int n)  
 b. int gear(int t1, int t2)  
 c. int guess(void)  
 d. void stuff\_it(double d, double \*pd)
3. a. char n\_to\_char(int n)  
 b. int digits(double x, int n)  
 c. double \* which(double \* p1, double \* p2)  
 d. int random(void)

- 4.
- ```
int sum(int a, int b)
{
    return a + b;
}
```

5. 用 double 替换 int 即可：

```
double sum(double a, double b)
{
    return a + b;
}
```

6. 该函数要使用指针：

```
void alter(int * pa, int * pb)
{
    int temp;
    temp = *pa + *pb;
    *pb = *pa - *pb;
    *pa = temp;
}
```

或者：

```
void alter(int * pa, int * pb)
{
    *pa += *pb;
    *pb = *pa - 2 * *pb;
}
```

7. 不正确。num 应声明在 salami() 函数的参数列表中，而不是声明在函数体中。另外，把 count++ 改成 num++。

## 8. 下面是一种方案:

```
int largest(int a, int b, int c)
{
    int max = a;
    if (b > max)
        max = b;
    if (c > max)
        max = c;
    return max;
}
```

## 9. 下面是一个最小的程序, showmenu() 和 getchoice() 函数分别是 a 和 b 的答案。

```
#include <stdio.h>
/* 声明程序中要用到的函数 */
void showmenu(void);
int getchoice(int, int);
int main()
{
    int res;
    showmenu();
    while ((res = getchoice(1, 4)) != 4)
    {
        printf("I like choice %d.\n", res);
        showmenu();
    }
    printf("Bye!\n");
    return 0;
}
void showmenu(void)
{
    printf("Please choose one of the following:\n");
    printf("1) copy files      2) move files\n");
    printf("3) remove files     4) quit\n");
    printf("Enter the number of your choice:\n");
}
int getchoice(int low, int high)
{
    int ans;
    int good;
    good = scanf("%d", &ans);
    while (good == 1 && (ans < low || ans > high))
    {
        printf("%d is not a valid choice; try again\n", ans);
        showmenu();
        scanf("%d", &ans);
    }
    if (good != 1)
    {
        printf("Non-numeric input. ");
        ans = 4;
    }
    return ans;
}
```

## A.10 第 10 章复习题答案

1. 打印的内容如下：

```
8 8
4 4
0 0
2 2
```

2. 数组 ref 有 4 个元素，因为初始化列表中的值是 4 个。
3. 数组名 ref 指向该数组的首元素（整数 8）。表达式 ref + 1 指向该数组的第 2 个元素（整数 4）。  
++ref 不是有效的表达式，因为 ref 是一个常量，不是变量。
4. ptr 指向第 1 个元素，ptr + 2 指向第 3 个元素（即第 2 行的第一个元素）。
- a. 12 和 16。  
b. 12 和 14（初始化列表中，用花括号把 12 括起来，把 14 和 16 括起来，所以 12 初始化第 1 行的第一个元素，而 14 初始化第 2 行的第一个元素）。
5. ptr 指向第 1 行，ptr + 1 指向第 2 行。\*ptr 指向第 1 行的第一个元素，而\*(ptr + 1) 指向第 2 行的第一个元素。
- a. 12 和 16。  
b. 12 和 14（同第 4 题，12 初始化第 1 行的第一个元素，而 14 初始化第 2 行的第一个元素）。
6. a. &grid[22][56]  
b. &grid[22][0] 或 grid[22]  
(grid[22] 是一个内含 100 个元素的一维数组，因此它就是首元素 grid[22][0] 的地址。)  
c. &grid[0][0] 或 grid[0] 或 (int \*) grid  
(grid[0] 是 int 类型元素 grid[0][0] 的地址，grid 是内含 100 个元素的 grid[0] 数组的地址。  
这两个地址的数值相同，但是类型不同，可以用强制类型转换把它们转换成相同的类型。)
7. a. int digits[10];  
b. float rates[6];  
c. int mat[3][5];  
d. char \* psa[20];  
注意，[] 比 \* 的优先级高，所以在没有圆括号的情况下，psa 先与 [20] 结合，然后再与 \* 结合。因此该声明与 char \*(psa[20]); 相同。  
e. char (\*pstr)[20];

### 注意

对第 e 小题而言，char \*pstr[20]; 不正确。这会让 pstr 成为一个指针数组，而不是一个指向数组的指针。具体地说，如果使用该声明，pstr 就指向一个 char 类型的值（即数组的第一个成员），而 pstr + 1 则指向下一个字节。使用正确的声明，pstr 是一个变量，而不是一个数组名。而且 pstr + 1 指向起始字节后面的第 20 个字节。

8. a. int sextet[6] = {1, 2, 4, 8, 16, 32};  
b. sextet[2]  
c. int lots[100] = { [99] = -1};  
d. int pots[100] = { [5] = 101, [10] = 101, 101, 101, 101};  
9. 0~9  
10. a. rootbeer[2] = value; 有效。  
b. scanf("%f", &rootbeer); 无效, rootbeer 不是 float 类型。  
c. rootbeer = value; 无效, rootbeer 不是 float 类型。  
d. printf("%f", rootbeer); 无效, rootbeer 不是 float 类型。  
e. things[4][4] = rootbeer[3]; 有效。  
f. things[5] = rootbeer; 无效, 不能用数组赋值。  
g. pf = value; 无效, value 不是地址。  
h. pf = rootbeer; 有效。  
11. int screen[800][600] ;  
12. a.  

```
void process(double ar[], int n);
void processvla(int n, double ar[n]);
process(trots, 20);
processvla(20, trots);
```

b.  

```
void process2(short ar2[30], int n);
void process2vla(int n, int m, short ar2[n][m]);
process2(clops, 10);
process2vla(10, 30, clops);
```

c.  

```
void process3(long ar3[10][15], int n);
void process3vla(int n, int m, int k, long ar3[n][m][k]);
process3(shots, 5);
process3vla(5, 10, 15, shots);
```

13. a.  

```
show( (int [4]) {8,3,9,2}, 4);
```

b.  

```
show2( (int [][] [3]) {{8,3,9}, {5,4,1}}, 2);
```

## A.11 第11章复习题答案

1. 如果希望得到一个字符串, 初始化列表中应包含'\0'。当然, 也可以用另一种语法自动添加空字符:  

```
char name[] = "Fess";
```
2.  
See you at the snack bar.  
ee you at the snack bar.  
See you  
e you

3.

```
y
my
mmy
ummy
Yummy
```

4. I read part of it all the way through.

5. a. Ho Ho Ho!!oH oH oH

b. 指向 char 的指针（即，char \*）。

c. 第 1 个 H 的地址。

d. --pc 的意思是把指针递减 1，并使用储存在该位置上的值。--\*pc 的意思是解引用 pc 指向的值，然后把该值减 1（例如，H 变成 G）。

e. Ho Ho Ho!!oH oH o

### 注意

在两个! 之间有一个空字符，但是通常该字符不会产生任何打印的效果。

f. while (\*pc) 检查 pc 是否指向一个空字符（即，是否指向字符串的末尾）。while 的测试条件中使用储存在指针指向位置上的值。

while (pc == str) 检查 pc 是否与 str 指向相同的位置（即，字符串的开头）。while 的测试条件中使用储存在指针指向位置上的值。

g. 进入第 1 个 while 循环后，pc 指向空字符。进入第 2 个 while 循环后，它指向空字符前面的存储区（即，str 所指向位置前面的位置）。把该字节解释成一个字符，并打印这个字符。然后指针退回到前面的字节处。永远都不会满足结束条件 (pc == str)，所以这个过程会一直持续下去。

h. 必须在主调程序中声明 pr(): char \* pr(char \*);

6. 字符变量占用一个字节，所以 sign 占 1 字节。但是字符常量储存为 int 类型，意思是'\$'通常占用 2 或 4 字节。但是实际上只使用 int 的 1 字节储存'\$'的编码。字符串"\$"使用 2 字节：一个字节储存'\$'的编码，一个字节储存的'\0'编码。

7. 打印的内容如下：

```
How are ya, sweetie? How are ya, sweetie?
Beat the clock.
eat the clock.
Beat the clock. Win a toy.
Beat
chat
hat
at
t
t
at
How are ya, sweetie?
```

8. 打印的内容如下：

```
faavrhee
```

\*le\*on\*sm

9. 下面是一种方案：

```
#include <stdio.h> // 提供 fgets() 和 getchar() 的原型
char * s_gets(char * st, int n)
{
    char * ret_val;

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

10. 下面是一种方案：

```
int strlen(const char * s)
{
    int ct = 0;
    while (*s++) // 或者 while (*s++ != '\0')
        ct++;
    return(ct);
}
```

11. 下面是一种方案：

```
#include <stdio.h> // 提供 fgets() 和 getchar() 的原型
#include <string.h> // 提供 strchr() 的原型
char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;
    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n'); // 查找换行符
        if (find) // 如果地址不是 NULL,
            *find = '\0'; // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

12. 下面是一种方案：

```
#include <stdio.h> /* 提供 NULL 的定义 */
char * strblk(char * string)
{
```

```

while (*string != ' ' && *string != '\0')
    string++; /* 在第 1 个空白或空字符处停止 */
if (*string == '\0')
    return NULL; /* NULL 指空指针 */
else
    return string;
}

```

下面是第 2 种方案，可以防止函数修改字符串，但是允许使用返回值改变字符串。表达式 `(char *) string` 被称为“通过强制类型转换取消 const”。

```

#include <stdio.h> /* 提供 NULL 的定义 */
char * strblk(const char * string)
{
    while (*string != ' ' && *string != '\0')
        string++; /* 在第 1 个空白或空字符处停止 */
    if (*string == '\0')
        return NULL; /* NULL 指空指针 */
    else
        return (char *)string;
}

```

### 13. 下面是一种方案：

```

/* compare.c -- 可行方案 */
#include <stdio.h>
#include <string.h> // 提供 strcmp() 的原型
#include <ctype.h>
#define ANSWER "GRANT"
#define SIZE 40
char * s_gets(char * st, int n);
void ToUpper(char * str);

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

void ToUpper(char * str)
{
    while (*str != '\0')
    {
        *str = toupper(*str);
        str++;
    }
}

```

```

    }

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

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

```

## A.12 第 12 章复习题答案

1. 自动存储类别；寄存器存储类别；静态、无链接存储类别。
2. 静态、无链接存储类别；静态、内部链接存储类别；静态、外部链接存储类别。
3. 静态、外部链接存储类别可以被多个文件使用。静态、内部链接存储类别只能在一个文件中使用。
4. 无链接。
5. 关键字 `extern` 用于声明中，表明该变量或函数已定义在别处。
6. 两者都分配了一个内含 100 个 `int` 类型值的数组。第 2 行代码使用 `calloc()` 把数组中的每个元素都设置为 0。
7. 默认情况下，`daisy` 只对 `main()` 可见，以 `extern` 声明的 `daisy` 才对 `petal()`、`stem()` 和 `root()` 可见。文件 2 中的 `extern int daisy;` 声明使得 `daisy` 对文件 2 中的所有函数都可见。第 1 个 `lily` 是 `main()` 的局部变量。`petal()` 函数中引用的 `lily` 是错误的，因为两个文件中都没有外部链接的 `lily`。虽然文件 2 中有一个静态的 `lily`，但是它只对文件 2 可见。第 1 个外部 `rose` 对 `root()` 函数可见，但是 `stem()` 中的局部 `rose` 覆盖了外部的 `rose`。
8. 下面是程序的输出：

```

color in main() is B
color in first() is R
color in main() is B
color in second() is G
color in main() is G

```

`first()` 函数没有使用 `color` 变量，但是 `second()` 函数使用了。

9. a. 声明告诉我们，程序将使用一个变量 `plink`，该文件包含的函数都可以使用这个变量。`calu_ct()` 函数的第一个参数是指向一个整数的指针，并假定它指向内含 `n` 个元素的数组。这里关键是要理解该程序不允许使用指针 `arr` 修改原始数组中的值。
- b. 不会。`value` 和 `n` 已经是原始数据的备份，所以该函数无法更改主调函数中相应的值。这些声

明的作用是防止函数修改 value 和 n 的值。例如，如果用 const 限定 n，就不能使用 n++ 表达式。

## A.13 第13章复习题答案

- 根据文件定义，应包含 #include <stdio.h>。应该把 fp 声明为文件指针：FILE \*fp;。要给 fopen() 函数提供一种模式：fopen("gelatin", "w")，或者 "a" 模式。fputs() 函数的参数顺序应该反过来。输出字符串应该有一个换行符，提高可读性。fclose() 函数需要一个文件指针，而不是一个文件名：fclose(fp);。下面是修改后的版本：

```
#include <stdio.h>
int main(void)
{
    FILE * fp;
    int k;
    fp = fopen("gelatin", "w");
    for (k = 0; k < 30; k++)
        fputs("Nanette eats gelatin.\n", fp);
    fclose(fp);
    return 0;
}
```

- 如果可以打开的话，会打开与命令行第 1 个参数名相同名称的文件，并在屏幕上显示文件中的每个数字字符。

- a. ch = getc(fp1);
- b. fprintf(fp2, "%c\n", ch);
- c. putc(ch, fp2);
- d. fclose(fp1); /\* 关闭 terky 文件 \*/

### 注意

fp1 用于输入操作，因为它识别以读模式打开的文件。与此类似，fp2 以写模式打开文件，所以常用于输出操作。

- 下面是一种方案：

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

int main(int argc, char * argv [])
{
    FILE * fp;
    double n;
    double sum = 0.0;
    int ct = 0;

    if (argc == 1)
        fp = stdin;
    else if (argc == 2)
    {
        if ((fp = fopen(argv[1], "r")) == NULL)
```

```

    {
        fprintf(stderr, "Can't open %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }
}
else
{
    fprintf(stderr, "Usage: %s [filename]\n", argv[0]);
    exit(EXIT_FAILURE);
}
while (fscanf(fp, "%lf", &n) == 1)
{
    sum += n;
    ++ct;
}
if (ct > 0)
    printf("Average of %d values = %f\n", ct, sum / ct);
else
    printf("No valid data.\n");

return 0;
}

```

5. 下面是一种方案：

```

#include <stdio.h>
#include <stdlib.h>
#define BUF 256
int has_ch(char ch, const char * line);
int main(int argc, char * argv [])
{
    FILE * fp;
    char ch;
    char line[BUF];

if (argc != 3)
{
    printf("Usage: %s character filename\n", argv[0]);
    exit(EXIT_FAILURE);
}
ch = argv[1][0];
if ((fp = fopen(argv[2], "r")) == NULL)
{
    printf("Can't open %s\n", argv[2]);
    exit(EXIT_FAILURE);
}
while (fgets(line, BUF, fp) != NULL)
{
    if (has_ch(ch, line))
        fputs(line, stdout);
}
fclose(fp);
return 0;
}

int has_ch(char ch, const char * line)
{
    while (*line)

```

```

    if (ch == *line++)
        return(1);
    return 0;
}

```

fgets()和fputs()函数要一起使用，因为fgets()会把按下Enter键的\n留在字符串中，fputs()与puts()不一样，不会添加一个换行符。

6. 二进制文件与文本文件的区别是，这两种文件格式对系统的依赖性不同。二进制流和文本流的区别包括是在读写流时程序执行的转换（二进制流不转换，而文本流可能要转换换行符和其他字符）。
7. a. 用fprintf()储存8238201时，将其视为7个字符，保存在7字节中。用fwrite()储存时，使用该数的二进制表示，将其储存为一个4字节的整数。  
b. 没有区别。两个函数都将其储存为一个单字节的二进制码。
8. 第1条语句是第2条语句的速记表示。第3条语句把消息写到标准错误上。通常，标准错误被定向到与标准输出相同的位置。但是标准错误不受标准输出重定向的影响。
9. 可以在以"r+"模式打开的文件中读写，所以该模式最合适。"a+"只允许在文件的末尾添加内容。"w+"模式提供一个空文件，丢弃文件原来的内容。

## A.14 第14章复习题答案

1. 正确的关键字是struct，不是structure。该结构模板要在左花括号前面有一个标记，或者在右花括号后面有一个结构变量名。另外，\*tags后面和模板结尾处都少一个分号。

2. 输出如下：

```

6 1
22 Spiffo Road
S p

```

- 3.

```

struct month {
    char name[10];
    char abbrev[4];
    int days;
    int monumb;
};

```

- 4.

```

struct month months[12] {
{
    { "January", "jan", 31, 1 },
    { "February", "feb", 28, 2 },
    { "March", "mar", 31, 3 },
    { "April", "apr", 30, 4 },
    { "May", "may", 31, 5 },
    { "June", "jun", 30, 6 },
    { "July", "jul", 31, 7 },
    { "August", "aug", 31, 8 },
    { "September", "sep", 30, 9 },
    { "October", "oct", 31, 10 },
    { "November", "nov", 30, 11 },
    { "December", "dec", 31, 12 }
};

```

5.

```

extern struct month months [];
int days(int month)
{
    int index, total;
    if (month < 1 || month > 12)
        return(-1); /* error signal */
    else
    {
        for (index = 0, total = 0; index < month; index++)
            total += months[index].days;
        return(total);
    }
}

```

注意, index 比月数小 1, 因为数组下标从 0 开始。然后, 用 `index < month` 代替 `index <= month`。

6. a. 要包含 `string.h` 头文件, 提供 `strcpy()` 的原型:

```

typedef struct lens { /* lens 描述 */
    float foclen;          /* 焦距长度, 单位: mm */
    float fstop;           /* 孔径 */
    char brand[30]; /* 品牌 */
} LENS;

LENS bigEye[10];
bigEye[2].foclen = 500;
bigEye[2].fstop = 2.0;
strcpy(bigEye[2].brand, "Remarkatar");
b. LENS bigEye[10] = { [2] = {500, 2, "Remarkatar"} };

```

7. a.

```

6
Arcturan
cturan

```

b. 使用结构名和指针:

```

deb.title.last
pb->title.last

```

c. 下面是一个版本:

```

#include <stdio.h>
#include "starfolk.h"      /* 让结构定义可用 */
void prbem (const struct bem * pbem )
{
    printf("%s %s is a %d-limbed %s.\n", pbem->title.first,
           pbem->title.last, pbem->limbs, pbem->type);
}

```

8. a. willie.born

```

b. pt->born
c. scanf("%d", &willie.born);
d. scanf("%d", &pt->born);
e. scanf("%s", willie.name.lname);
f. scanf("%s", pt->name.lname);

```

g. willie.name.fname[2]  
 h. strlen(willie.name.fname) + strlen(willie.name.lname)

9. 下面是一种方案:

```
struct car {
    char name[20];
    float hp;
    float epampg;
    float wbase;
    int year;
};
```

10. 应该这样建立函数:

```
struct gas {
    float distance;
    float gals;
    float mpg;
};

struct gas mpgs(struct gas trip)
{
    if (trip.gals > 0)
        trip.mpg = trip.distance / trip.gals;
    else
        trip.mpg = -1.0;
    return trip;
}

void set_mpgs(struct gas * ptrip)
{
    if (ptrip->gals > 0)
        ptrip->mpg = ptrip->distance / ptrip->gals;
    else
        ptrip->mpg = -1.0;
}
```

注意, 第 1 个函数不能直接改变其主调程序中的值, 所以必须用返回值才能传递信息。

```
struct gas idaho = {430.0, 14.8}; // 设置前两个成员
idaho = mpgs(idaho); // 重置数据结构
```

但是, 第 2 个函数可以直接访问最初的结构:

```
struct gas ohio = {583, 17.6}; // 设置前两个成员
set_mpgs(&ohio); // 设置第 3 个成员
```

11. enum choices {no, yes, maybe};

12. char \* (\*pfun)(char \*, char);

13.

```
double sum(double, double);
double diff(double, double);
double times(double, double);
double divide(double, double);
double (*pfl[4])(double, double) = {sum, diff, times, divide};
```

或者用更简单的形式, 把代码中最后一行替换成:

```
typedef double (*ptype) (double, double);
ptype pfl[4] = {sum, diff, times, divide};
```

调用 diff() 函数:

```
pf1[1](10.0, 2.5);      // 第 1 种表示法
(*pf1[1])(10.0, 2.5); // 等价表示法
```

## A.15 第 15 章复习题答案

1. a. 00000011  
b. 00001101  
c. 00111011  
d. 01110111
2. a. 21, 025, 0x15  
b. 85, 0125, 0x55  
c. 76, 0114, 0x4C  
d. 157, 0235, 0x9D
3. a. 252  
b. 2  
c. 7  
d. 7  
e. 5  
f. 3  
g. 28
4. a. 255  
b. 1 (not false is true)  
c. 0  
d. 1 (true and true is true)  
e. 6  
f. 1 (true or true is true)  
g. 40
5. 掩码的二进制是 1111111；十进制是 127；八进制是 0177；十六进制是 0x7F。
6. bitval \* 2 和 bitval << 1 都把 bitval 的当前值增加一倍，它们是等效的。但是 mask += bitval 和 mask |= bitval 只有在 bitval 和 mask 没有同时打开的位时效果才相同。例如，2 | 4 得 6，但是 3 | 6 也得 6。
7. a.

```
struct tb_drives {
    unsigned int diskdrives : 2;
    unsigned int          : 1;
    unsigned int cdromdrives : 2;
    unsigned int          : 1;
    unsigned int harddrives : 2;
};
```

b.

```
struct kb_drives {
    unsigned int harddrives : 2;
    unsigned int : 1;
    unsigned int cdromdrives : 2;
    unsigned int : 1;
    unsigned int diskdrives : 2;
};
```

## A.16 第 16 章复习题答案

1. a. `dist = 5280 * miles;` 有效。
- b. `plort = 4 * 4 + 4;` 有效。但是如果用户需要的是 `4 * (4 + 4)`，则应该使用 `#define POD (FEET + FEET)`。
- c. `nex == 6;` 无效（如果两个等号之间没有空格，则有效，但是没有意义）。显然，用户忘记了在编写预处理器代码时不用加=。
- d. `y = y + 5;` 有效。`berg = berg + 5 * lob;` 有效，但是可能得不到想要的结果。`est = berg + 5/y + 5;` 有效，但是可能得不到想要的结果。
2. `#define NEW(X) ((X) + 5)`
3. `#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))`
4. `#define EVEN_GT(X,Y) ((X) > (Y) && (X) % 2 == 0 ? 1 : 0)`
5. `#define PR(X,Y) printf(#X " is %d and " #Y " is %d\n", X, Y)`  
(因为该宏中没有运算符（如，乘法）作用于 X 和 Y，所以不需要使用圆括号。)
6. a. `#define QUARTERCENTURY 25`  
b. `#define SPACE ' '`  
c. `#define PS() putchar(' ') 或 #define PS() putchar(SPACE)`  
d. `#define BIG(X) ((X) + 3)`  
e. `#define SUMSQ(X,Y) ((X)*(X) + (Y)*(Y))`
7. 尝试这样：`#define P(X) printf("name: "#X"; value: %d; address: %p\n", X, &X)`  
(如果你的实现无法识别地址专用的%p 转换说明，可以用%u 或%lu 代替。)
8. 使用条件编译指令。一种方法是使用`#ifndef`：

```
#define _SKIP_ /* 如果不需要跳过代码，则删除这条指令 */
#ifndef _SKIP_
/* 需要跳过的代码 */
#endif
```
9.

```
#ifdef PR_DATE
    printf("Date = %s\n", __DATE__);
#endif
```
10. 第 1 个版本返回 `x*x`，这只是返回了 `square()` 的 `double` 类型值。例如，`square(1.3)` 会返回 1.69。第 2 个版本返回 `(int)(x*x)`，计算结果被截断后返回。但是，由于该函数的返回类型是 `double`，`int` 类型的值将被升级为 `double` 类型的值，所以 1.69 将先被转换成 1，然后

被转换成 1.00。第 3 个版本返回 (int) (x\*x+0.5)。加上 0.5 可以让函数把结果四舍五入至与原值最接近的值，而不是简单地截断。所以， $1.69+0.5$  得 2.19，然后被截断为 2，然后被转换成 2.00；而  $1.44+0.5$  得 1.94，被截断为 1，然后被转换成 1.00。

11. 这是一种方案：#define BOOL(X) \_Generic((X), \_Bool : "boolean", default : "not boolean")
12. 应该把 argv 参数声明为 char \*argv[] 类型。命令行参数被储存为字符串，所以该程序应该先把 argv[1] 中的字符串转换成 double 类型的值。例如，用 stdlib.h 库中的 atof() 函数。程序中使用了 sqrt() 函数，所以应包含 math.h 头文件。程序在求平方根之前应排除参数为负的情况（检查参数是否大于或等于 0）。
13. a. qsort( void \*scores, (size\_t) 1000, sizeof (double), comp);  
 b. 下面是一个比较使用的比较函数：  

```
int comp(const void * p1, const void * p2)
{
    /* 要用指向 int 的指针来访问值 */
    /* 在 C 中是否进行强制类型转换都可以，在 C++ 中必须进行强制类型转换 */
    const int * a1 = (const int *) p1; const int * a2 = (const int *)
p2;
    if (*a1 > *a2)
        return -1;
    else if (*a1 == *a2)
        return 0;
    else
        return 1;
}
```
14. a. 函数调用应该类似：memcpy(data1, data2, 100 \* sizeof(double));  
 b. 函数调用应该类似：memcpy(data1, data2 + 200, 100 \* sizeof(double));

## A.17 第 17 章复习题答案

1. 定义一种数据类型包括确定如何储存数据，以及设计管理该数据的一系列函数。
2. 因为每个结构包含下一个结构的地址，但是不包含上一个结构的地址，所以这个链表只能沿着一个方向遍历。可以修改结构，在结构中包含两个指针，一个指向下一个结构，一个指向下一个结构。当然，程序也要添加代码，在每次新增结构时为这些指针赋正确的地址。
3. ADT 是抽象数据类型，是对一种类型属性集和可以对该类型进行的操作的正式定义。ADT 应该用一般语言表示，而不是用某种特殊的计算机语言，而且不应该包含实现细节。
4. **直接传递变量的优点：**该函数查看一个队列，但是不改变其中的内容。直接传递队列变量，意味着该函数使用的是原始队列的副本，这保证了该函数不会更改原始的数据。直接传递变量时，不需要使用地址运算符或指针。

**直接传递变量的缺点：**程序必须分配足够的空间储存整个变量，然后拷贝原始数据的信息。如果变量是一个大型结构，用这种方法将花费大量的时间和内存空间。

**传递变量地址的优点：**如果待传递的变量是大型结构，那么传递变量的地址和访问原始数据会更快，所需的内存空间更少。

**传递变量地址的缺点：**必须记得使用地址运算符或指针。在 K&R C 中，函数可能会不小心改变原

始数据，但是用 ANSI C 中的 const 限定符可以解决这个问题。

5. a.

- |       |                                                            |
|-------|------------------------------------------------------------|
| 类型名:  | 栈                                                          |
| 类型属性: | 可以储存有序项                                                    |
| 类型操作: | 初始化栈为空<br>确定栈是否为空<br>确定栈是否已满<br>从栈顶添加项（压入项）<br>从栈顶删除项（弹出项） |

b. 下面以数组形式实现栈，但是这些信息只影响结构定义和函数定义的细节，不会影响函数原型的接口。

```
/* stack.h -- 栈的接口 */
#include <stdbool.h>
/* 在这里插入 Item 类型 */
/* 例如: typedef int Item; */

#define MAXSTACK 100

typedef struct stack
{
    Item items[MAXSTACK]; /* 储存信息 */
    int top; /* 第 1 个空位的索引 */
} Stack;

/* 操作:      初始化栈 */ /* */
/* 前提条件:  ps 指向一个栈 */ /* */
/* 后置条件:  该栈被初始化为空 */ /* */
void InitializeStack(Stack * ps);

/* 操作:      检查栈是否已满 */ /* */
/* 前提条件:  ps 指向之前已被初始化的栈 */ /* */
/* 后置条件:  如果栈已满, 该函数返回 true; 否则, 返回 false */ /* */
bool FullStack(const Stack * ps);

/* 操作:      检查栈是否为空 */ /* */
/* 前提条件:  ps 指向之前已被初始化的栈 */ /* */
/* 后置条件:  如果栈为空, 该函数返回 true; 否则, 返回 false */ /* */
bool EmptyStack(const Stack *ps);

/* 操作:      把项压入栈顶 */ /* */
/* 前提条件:  ps 指向之前已被初始化的栈 */ /* */
/*          item 是待压入栈顶的项 */ /* */
/* 后置条件:  如果栈不满, 把 item 放在栈顶, 该函数返回 true; */ /* */
/*          否则, 栈不变, 该函数返回 false */ /* */
bool Push(Item item, Stack * ps);

/* 操作:      从栈顶删除项 */ /* */
/* 前提条件:  ps 指向之前已被初始化的栈 */ /* */
/* 后置条件:  如果栈不为空, 把栈顶的 item 拷贝到*pitem, */ /* */
/*          栈变短, 该函数返回 true; 否则, 栈不变, 返回 false */ /* */
bool Pop(Item *pitem, Stack * ps);
```

```

/*
    删除栈顶的 item，该函数返回 true;
*/
/*
    如果该操作后栈中没有项，则重置该栈为空。
*/
/*
    如果删除操作之前栈为空，栈不变，该函数返回 false
*/
bool Pop(Item *pitem, Stack *ps);

```

6. 比较所需的最大次数如下：

| 项     | 顺序查找  | 二分查找 |
|-------|-------|------|
| 3     | 3     | 2    |
| 1023  | 1023  | 10   |
| 65535 | 65535 | 16   |

7. 见图 A.1。

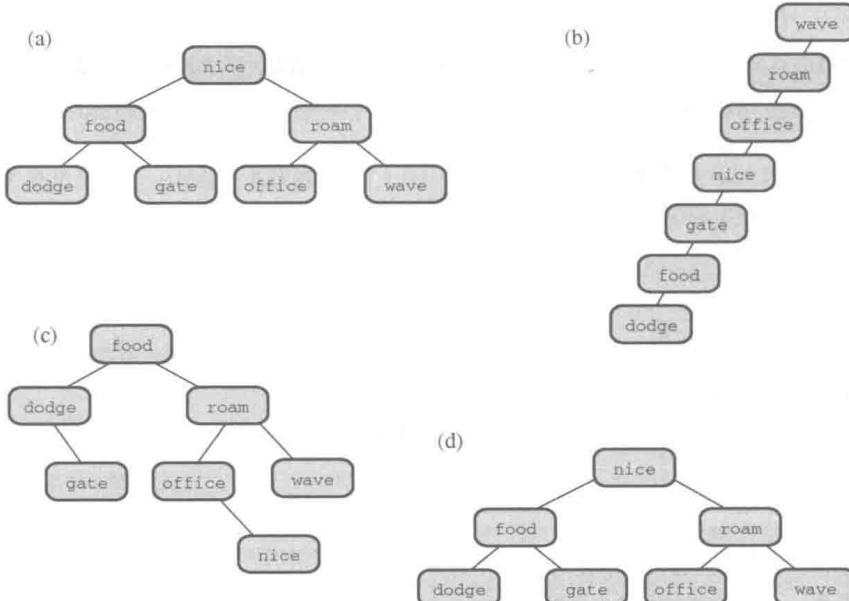


图 A.1 单词的二分查找树

8. 见图 A.2。

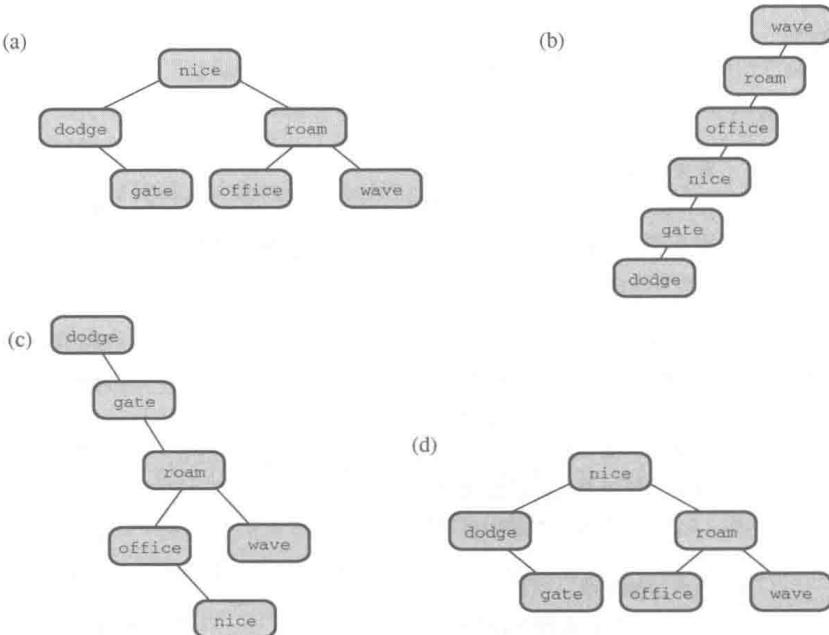


图 A.2 删除项后的单词二分查找树

# 附录 B

## 参考资料

本书这部分总结了 C 语言的基本特性和一些特定主题的详细内容，包括以下 9 个部分。

- 参考资料 I：补充阅读
- 参考资料 II：C 运算符
- 参考资料 III：基本类型和存储类别
- 参考资料 IV：表达式、语句和程序流
- 参考资料 V：新增了 C99 和 C11 的标准 ANSI C 库
- 参考资料 VI：扩展的整数类型
- 参考资料 VII：扩展的字符支持
- 参考资料 VIII：C99/C11 数值计算增强
- 参考资料 IX：C 与 C++的区别

### B.1 参考资料 I：补充阅读

如果想了解更多 C 语言和编程方面的知识，下面提供的资料会对你有所帮助。

#### B.1.1 在线资源

C 程序员帮助建立了互联网，而互联网可以帮助你学习 C。互联网时刻都在发展、变化，这里所列的资源只是在撰写本书时可用的资源。当然，你可以在互联网中找到其他资源。

如果有一些与 C 语言相关的问题或只是想扩展你的知识，可以浏览 C FAQ（常见问题解答）的站点：  
[c-faq.com](http://c-faq.com)

但是，这个站点的内容主要涵盖到 C89。

如果对 C 库有疑问，可以访问这个站点获得信息：[www.acm.uiuc.edu/webmonkeys/book/c\\_guide/index.html](http://www.acm.uiuc.edu/webmonkeys/book/c_guide/index.html)。  
这个站点全面讨论指针：[pweb.netcom.com/~tjensen/ptr/pointers.htm](http://pweb.netcom.com/~tjensen/ptr/pointers.htm)。

还可以使用谷歌和雅虎的搜索引擎，查找相关文章和站点：

[www.google.com](http://www.google.com)  
[search.yahoo.com](http://search.yahoo.com)  
[www.bing.com](http://www.bing.com)

可以使用这些站点中的高级搜索特性来优化你要搜索的内容。例如，尝试搜索 C 教程。

你可以通过新闻组（newsgroup）在网上提问。通常，新闻组阅读程序通过你的互联网服务提供商提供的账号访问新闻组。另一种访问方法是在网页浏览器中输入这个地址：<http://groups.google.com>。

你应该先花时间阅读新闻组，了解它涵盖了哪些主题。例如，如果你对如何使用 C 语言完成某事有疑问，可以试试这些新闻组：

comp.lang.c  
comp.lang.c.moderated

可以在这里找到愿意提供帮助的人。你所提的问题应该与标准 C 语言相关，不要在这里询问如何在 UNIX 系统中获得无缓冲输入之类的问题。特定平台都有专门的新闻组。最重要的是，不要询问他们如何解决家庭作业中的问题。

如果对 C 标准有疑问，试试这个新闻组：comp.std.c。但是，不要在这里询问如何声明一个指向三维数组的指针，这类问题应该到另一个新闻组：comp.lang.c。

最后，如果对 C 语言的历史感兴趣，可以浏览下 C 创始人 Dennis Ritchie 的站点，其中 1993 年中有一篇文章介绍了 C 的起源和发展：[cm.bell-labs.com/cm/cs/who/dmr/chist.html](http://cm.bell-labs.com/cm/cs/who/dmr/chist.html)。

### B.1.2 C 语言书籍

Feuer, Alan R. *The C Puzzle Book, Revised Printing*. Upper Saddle River, NJ: Addison-Wesley Professional, 1998。这本书包含了许多程序，可以用来学习，推测这些程序应输出的内容。预测输出对测试和扩展 C 的理解很有帮助。本书也附有答案和解释。

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*, Second Edition .Englewood Cliffs, NJ: Prentice Hall, 1988。第 1 本 C 语言书的第 2 版（注意，作者 Dennis Ritchie 是 C 的创始者）。本书的第 1 版给出了 K&R C 的定义，许多年来它都是非官方的标准。第 2 版基于当时的 ANSI 草案进行了修订，在编写本书时该草案已成为了标准。本书包含了许多有趣的例子，但是它假定读者已经熟悉了系统编程。

Koenig, Andrew. *C Traps and Pitfalls* . Reading, MA: Addison-Wesley, 1989。本书的中文版《C 陷阱与缺陷》已由人民邮电出版社出版。

Summit, Steve. *C Programming FAQs* . Reading, MA: Addison-Wesley, 1995。这本书是互联网 FAQ 的延伸阅读版本。

### B.1.3 编程书籍

Kernighan, Brian W. and P.J. Plauger. *The Elements of Programming Style*, Second Edition . NewYork: McGraw-Hill, 1978。这本短小精悍的绝版书籍，历经岁月却无法掩盖其真知灼见。书中介绍了要编写高效的程序，什么该做，什么不该做。

Knuth, Donald E. *The Art of Computer Programming*, 第 1 卷（基本算法）, Third Edition . Reading, MA: Addison-Wesley, 1997。这本经典的标准参考书非常详尽地介绍了数据表示和算法分析。第 2 卷（半数学算法, 1997) 探讨了伪随机数。第 3 卷（排序和搜索, 1998) 介绍了排序和搜索，以伪代码和汇编语言的形式给出示例。

Sedgewick, Robert. *Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*, Third Edition. Reading, MA: Addison-Wesley Professional, 1997。顾名思义，这本书介绍了数据结构、排序和搜索。本书中文版《C 算法（第 1 卷）基础、数据结构、排序和搜索（第 3 版）》已由人民邮电出版社出版。

### B.1.4 参考书籍

Harbison, Samuel P. and Steele, Guy L. C: *A Reference Manual*, Fifth Edition. Englewood Cliffs,NJ: Prentice Hall, 2002。这本参考手册介绍了 C 语言的规则和大多数标准库函数。它结合了 C99，提供了许多例子。《C 语言参考手册（第 5 版）（英文版）》已由人民邮电出版社出版。

Plauger, P.J. *The Standard C Library* . Englewood Cliffs, NJ: Prentice Hall, 1992。这本大型的参考手册介

绍了标准库函数，比一般的编译器手册更详尽。

**The International C Standard, ISO/IEC 9899:2011**。在撰写本书时，可以花 285 美元从 [www.ansi.org](http://www.ansi.org) 下载该标准的电子版，或者花 238 欧元从 IEC 下载。别指望通过这本书学习 C 语言，因为它并不是一本学习教程。这是一句有代表性的话，可见一斑：“如果在一个翻译单元中声明一个特定标识符多次，在该翻译单元中都可见，那么语法可根据上下文无歧义地引用不同的实体”。

## B.1.5 C++书籍

Prata, Stephen. **C++ Primer Plus**, Sixth Edition. Upper Saddle River, NJ: Addison-Wesley, 2012。本书介绍了 C++ 语言（C++11 标准）和面向对象编程的原则。

Stroustrup, Bjarne. **The C++ Programming Language**, Fourth Edition. Reading, MA: Addison-Wesley, 2013。本书由 C++ 的创始人撰写，介绍了 C++11 标准。

## B.2 参考资料 II: C 运算符

C 语言有大量的运算符。表 B.2.1 按优先级从高至低的顺序列出了 C 运算符，并给出了其结合性。除非特别指明，否则所有运算符都是二元运算符（需要两个运算对象）。注意，一些二元运算符和一元运算符的表示符号相同，但是其优先级不同。例如，\*（乘法运算符）和\*（间接运算符）。表后面总结了每个运算符的用法。

表 B.2.1 C 运算符

| 运算符（优先级从高至低）                                | 结合律  |
|---------------------------------------------|------|
| ++（后缀） --（后缀） ()（函数调用）<br>[] {}（复合字面量） . -> | 从左往右 |
| ++（前缀） --（前缀） - + ~ !<br>*（解引用） &（取址）       | 从右往左 |
| sizeof _Alignof(类型名)（本栏都是一元运算符）<br>(类型名)    | 从右往左 |
| * / %                                       | 从左往右 |
| + -（都是二元运算符）                                | 从左往右 |
| <<>>                                        | 从左往右 |
| <><= >=                                     | 从左往右 |
| == !=                                       | 从左往右 |
| &                                           | 从左往右 |
| ^                                           | 从左往右 |
|                                             | 从左往右 |
| &&                                          | 从左往右 |
|                                             | 从左往右 |
| ? : (条件表达式)                                 | 从右往左 |
| = *= /= += -= <<= >>= &=  = ^=              | 从右往左 |
| , (逗号运算符)                                   | 从左往右 |

## B.2.1 算术运算符

- + 把右边的值加到左边的值上。
- + 作为一元运算符，生成一个大小和符号都与右边值相同的值。
- 从左边的值中减去右边的值。
- 作为一元运算符，生成一个与右边值大小相等符号相反的值。
- \* 把左边的值乘以右边的值。
- / 把左边的值除以右边的值；如果两个运算对象都是整数，其结果要被截断。
- % 得左边值除以右边值时的余数
- ++ 把右边变量的值加 1（前缀模式），或把左边变量的值加 1（后缀模式）。
- 把右边变量的值减 1（前缀模式），或把左边变量的值减 1（后缀模式）。

## B.2.2 关系运算符

下面的每个运算符都把左边的值与右边的值相比较。

- < 小于
- <= 小于或等于
- == 等于
- >= 大于或等于
- > 大于
- != 不等于

### 关系表达式

简单的关系表达式由关系运算符及其两侧的运算对象组成。如果关系为真，则关系表达式的值为 1；如果关系为假，则关系表达式的值为 0。下面是两个例子：

`5 > 2` 关系为真，整个表达式的值为 1。  
`(2 + a) == a` 关系为假，整个表达式的值为 0。

## B.2.3 赋值运算符

C 语言有一个基本赋值运算符和多个复合赋值运算符。`=` 运算符是基本的形式：

- = 把它右边的值赋给其左边的左值。

下面的每个赋值运算符都根据它右边的值更新其左边的左值。我们使用 `R-H` 表示右边，`L-R` 表示左边。

- `+=` 把左边的变量加上右边的量，并把结果储存在左边的变量中。
- `-=` 从左边的变量中减去右边的量，并把结果储存在左边的变量中。
- `*=` 把左边的变量乘以右边的量，并把结果储存在左边的变量中。
- `/=` 把左边的变量除以右边的量，并把结果储存在左边的变量中。
- `%=` 得到左边量除以右边量的余数，并把结果储存在左边的变量中。
- `&=` 把 `L-R & R-H` 的值赋给左边的量，并把结果储存在左边的变量中。

**| =** 把 L-H | R-H 的值赋给左边的量，并把结果储存在左边的变量中。  
**^ =** 把 L-H ^ R-H 的值赋给左边的量，并把结果储存在左边的变量中。  
**>>=** 把 L-H >> R-H 的值赋给左边的量，并把结果储存在左边的变量中。  
**<<=** 把 L-H << R-H 的值赋给左边的量，并把结果储存在左边的变量中。

#### 示例

`rabbits *= 1.6;` 与 `rabbits = rabbits * 1.6` 效果相同。

## B.2.4 逻辑运算符

逻辑运算符通常以关系表达式作为运算对象。**!** 运算符只需要一个运算对象，其他运算符需要两个运算对象，运算符左边一个，右边一个。

**&&** 逻辑与  
**||** 逻辑或  
**!** 逻辑非

### 1. 逻辑表达式

当且仅当两个表达式都为真时，`expression1 && expression2` 的值才为真。

两个表达式中至少有一个为真时，`expression1 || expression2` 的值就为真。

如果 `expression` 的值为假，则 `!expression` 为真，反之亦然。

### 2. 逻辑表达式的求值顺序

逻辑表达式的求值顺序是从左往右。当发现可以使整个表达式为假的条件时立即停止求值。

### 3. 示例

`6 > 2 && 3 == 3` 为真。

`!(6 > 2 && 3 == 3)` 为假。

`x != 0 && 20/x < 5` 只有在 `x` 是非零时才会对第 2 个表达式求值。

## B.2.5 条件运算符

**?:** 有 3 个运算对象，每个运算对象都是一个表达式：`expression1 ? expression2 : expression3`

如果 `expression1` 为真，则整个表达式的值等于 `expression2` 的值；否则，等于 `expression3` 的值。

#### 示例

`(5 > 3) ? 1 : 2` 的值为 1。

`(3 > 5) ? 1 : 2` 的值为 2。

`(a > b) ? a : b` 的值是 `a` 和 `b` 中较大者

## B.2.6 与指针有关的运算符

**&** 是地址运算符。当它后面是一个变量名时，**&** 给出该变量的地址。

**\*** 是间接或解引用运算符。当它后面是一个指针时，**\*** 给出储存在指针指向地址中的值。

**示例**

`&nurse` 是变量 `nurse` 的地址:

```
nurse = 22;
ptr = &nurse; /* 指向 nurse 的指针 */
val = *ptr;
```

以上代码的效果是把 22 赋给 `val`。

## B.2.7 符号运算符

- 是负号，反转运算对象的符号。

+ 是正号，不改变运算对象的符号。

## B.2.8 结构和联合运算符

结构和联合使用一些运算符标识成员。成员运算符与结构和联合一起使用，间接成员运算符与指向结构或联合的指针一起使用。

### 1. 成员运算符

成员运算符（.）与结构名或联合名一起使用，指定结构或联合中的一个成员。如果 `name` 是一个结构名，`member` 是该结构模板指定的成员名，那么 `name.member` 标识该结构中的这个成员。`name.member` 的类型就是被指定 `member` 的类型。在联合中也可以用相同的方式使用成员运算符。

**示例**

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

上面这条语句把 1265 赋给结构变量 `item` 的成员 `code`。

### 2. 间接成员运算符（或结构指针运算符）

间接成员运算符（->）与一个指向结构或联合的指针一起使用，标识该结构或联合的一个成员。假设 `ptrstr` 是一个指向结构的指针，`member` 是该结构模板指定的成员，那么 `ptrstr->member` 标识了指针所指向结构的这个成员。在联合中也可以用相同的方式使用间接成员运算符。

**示例**

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

以上程序段把 3451 赋给结构 `item` 的成员 `code`。下面 3 种写法是等效的：

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

## B.2.9 按位运算符

下面所列除了～，都是按位运算符。

～是一元运算符，它通过翻转运算对象的每一位得到一个值。

- & 是逻辑与运算符，只有当两个运算对象中对应的位都为 1 时，它生成的值中对应的位才为 1。
- | 是逻辑或运算符，只要两个运算对象中对应的位有一位为 1，它生成的值中对应的位就为 1。
- ^ 是按位异或运算符，只有两个运算对象中对应的位中只有一位为 1（不能全为 1），它生成的值中对应的位才为 1。
- << 是左移运算符，把左边运算对象中的位向左移动得到一个值。移动的位数由该运算符右边的运算对象确定，空出的位用 0 填充。
- >> 是右移运算符，把左边运算对象中的位向右移动得到一个值。移动的位数由该运算符右边的运算对象确定，空出的位用 0 填充。

#### 示例

假设有下面的代码：

```
int x = 2;
int y = 3;
```

`x & y` 的值为 2，因为 `x` 和 `y` 的位组合中，只有第 1 位均为 1。而 `y << x` 的值为 12，因为在 `y` 的位组合中，3 的位组合向左移动两位，得到 12。

## B.2.10 混合运算符

`sizeof` 给出它右边运算对象的大小，单位是 `char` 的大小。通常，`char` 类型的大小是 1 字节。运算对象可以圆括号中的类型说明符，如 `sizeof(float)`，也可以是特定的变量名、数组名等，如 `sizeof foo`。`sizeof` 表达式的类型是 `size_t`。

`_Alignof` (C11) 给出它的运算对象指定类型的对齐要求。一些系统要求以特定值的倍数在地址上储存特定类型，如 4 的倍数。这个整数就是对齐要求。

`(类型名)` 是强制类型转换运算符，它把后面的值转换成圆括号中关键字指定的类型。例如，`(float) 9` 把整数 9 转换成浮点数 9.0。

`,` 是逗号运算符，它把两个表达式链接成一个表达式，并保证先对最左端的表达式求值。整个表达式的值是最右边表达式的值。该运算符通常在 `for` 循环头中用于包含更多的信息。

#### 示例

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
    fargo += step;
```

# B.3 参考资料 III：基本类型和存储类别

## B.3.1 总结：基本数据类型

C 语言的基本数据类型分为两大类：整数类型和浮点数类型。不同的种类提供了不同的范围和精度。

### 1. 关键字

创建基本数据类型要用到 8 个关键字：`int`、`long`、`short`、`unsigned`、`char`、`float`、`double`、`signed` (ANSI C)。

### 2. 有符号整数

有符号整数可以具有正值或负值。

`int` 是所有系统中基本整数类型。

`long` 或 `long int` 可储存的整数应大于或等于 `int` 可储存的最大数；`long` 至少是 32 位。

`short` 或 `short int` 整数应小于或等于 `int` 可储存的最大数；`short` 至少是 16 位。通常，`long` 比 `short` 大。例如，在 PC 中的 C DOS 编译器提供 16 位的 `short` 和 `int`、32 位的 `long`。这完全取决于系统。

C99 标准提供了 `long long` 类型，至少和 `long` 一样大，至少是 64 位。

### 3. 无符号整数

无符号整数只有 0 和正值，这使得该类型能表示的正数范围更大。在所需的类型前面加上关键字 `unsigned`: `unsigned int`、`unsigned long`、`unsigned short`、`unsigned long long`。单独的 `unsigned` 相当于 `unsigned int`。

### 4. 字符

字符是如 A、&、+ 这样的印刷符号。根据定义，`char` 类型的变量占用 1 字节的内存。过去，`char` 类型的大小通常是 8 位。然而，C 在处理更大的字符集时，`char` 类型可以是 16 位，或者甚至是 32 位。

这种类型的关键字是 `char`。一些实现使用有符号的 `char`，但是其他实现使用无符号的 `char`。ANSI C 允许使用关键字 `signed` 和 `unsigned` 指定所需类型。从技术层面上看，`char`、`unsigned char` 和 `signed char` 是 3 种不同的类型，但是 `char` 类型与其他两种类型的表示方法相同。

### 5. 布尔类型 (C99)

`_Bool` 是 C99 新增的布尔类型。它一个无符号整数类型，只能储存 0（表示假）或 1（表示真）。包含 `stdbool.h` 头文件后，可以用 `bool` 表示 `_Bool`、`true` 表示 1、`false` 表示 0，让代码与 C++ 兼容。

### 6. 实浮点数和复浮点数类型

C99 识别两种浮点数类型：实浮点数和复浮点数。浮点类型由这两种类型构成。

实浮点数可以是正值或负值。C 识别 3 种实浮点类型。

`float` 是系统中的基本浮点类型。它至少可以精确表示 6 位有效数字，通常 `float` 为 32 位。

`double`（可能）表示更大的浮点数。它能表示比 `float` 更多的有效数字和更大的指数。它至少能精确表示 10 位有效数字。通常，`double` 为 64 位。

`long double`（可能）表示更大的浮点数。它能表示比 `double` 更多的有效数字和更大的指数。

复数由两部分组成：实部和虚部。C99 规定一个复数在内部用一个有两个元素的数组表示，第 1 个元素表示实部，第 2 个元素表示虚部。有 3 种复浮点数类型。

`float _Complex` 表示实部和虚部都是 `float` 类型的值。

`double _Complex` 表示实部虚部都是 `double` 类型的值。

`long double _Complex` 表示实部和虚部都是 `long double` 类型的值。

每种情况，前缀部分的类型都称为相应的实数类型 (*corresponding real type*)。例如，`double` 是 `double _Complex` 相应的实数类型。

C99 中，复数类型在独立环境中是可选的，这样的环境中不需要操作系统也可运行 C 程序。在 C11 中，复数类型在独立环境和主机环境都是可选的。

有 3 种虚数类型。它们在独立环境中和主机环境中（C 程序在一种操作系统下运行的环境）都是可选

的。虚数只有虚部。这 3 种类型如下。

`float _Imaginary` 表示虚部是 `float` 类型的值。

`double _Imaginary` 表示虚部是 `double` 类型的值。

`long double _Imaginary` 表示虚部是 `long double` 类型的值。

可以用实数和 `I` 值来初始化复数。`I` 定义在 `complex.h` 头文件中，表示  $i$ （即  $-1$  的平方根）。

```
#include <complex.h>           // I 定义在该头文件中
double _Complex z = 3.0;        // 实部 = 3.0, 虚部 = 0
double _Complex w = 4.0 * I;    // 实部 = 0.0, 虚部 = 4.0
double Complex u = 6.0 - 8.0 * I; // 实部 = 6.0, 虚部 = -8.0
```

前面章节讨论过，`complex.h` 库包含一些返回复数实部和虚部的函数。

## B.3.2 总结：如何声明一个简单变量

1. 选择所需的类型。
2. 选择一个合适的变量名。
3. 使用这种声明格式：`type-specifier variable-name;`

`type-specifier` 由一个或多个类型关键字组成，下面是一些例子：

```
int erest;
unsigned short cash;
```

4. 声明多个同类型变量时，使用逗号分隔符隔开各变量名：

```
char ch, init, ans;
```

5. 可以在声明的同时初始化变量：

```
float mass = 6.0E24;
```

### 总结：存储类别

关键字：auto、extern、static、register、`_Thread_local` (C11)

一般注解：

变量的存储类别取决于它的作用域、链接和存储期。存储类别由声明变量的位置和与之关联的关键字决定。定义在所有函数外部的变量具有文件作用域、外部链接、静态存储期。声明在函数中的变量是自动变量，除非该变量前面使用了其他关键字。它们具有块作用域、无链接、自动存储期。以 `static` 关键字声明在函数中的变量具有块作用域、无链接、静态存储期。以 `static` 关键字声明在函数外部的变量具有文件作用域、内部链接、静态存储期。

C11 新增了一个存储类别说明符：`_Thread_local`。以该关键字声明的对象具有线程存储期，意思是在线程中声明的对象在该线程运行期间一直存在，且在线程开始时被初始化。因此，这种对象属于线程私有。

属性：

下面总结了这些存储类别的属性：

| 存储类别 | 存储期 | 作用域 | 链接 | 如何声明                            |
|------|-----|-----|----|---------------------------------|
| 自动   | 自动  | 块   | 无  | 在块中                             |
| 寄存器  | 自动  | 块   | 无  | 在块中，使用关键字 <code>register</code> |

续表

| 存储类别    | 存储期 | 作用域 | 链接 | 如何声明                                 |
|---------|-----|-----|----|--------------------------------------|
| 静态、外部链接 | 静态  | 文件  | 外部 | 在所有函数外部                              |
| 静态、内部链接 | 静态  | 文件  | 内部 | 在所有函数外部，使用关键字 static                 |
| 静态、无链接  | 静态  | 块   | 无  | 在块中，使用关键字 static                     |
| 线程、外部链接 | 线程  | 文件  | 外部 | 在所有块的外部，使用关键字 _Thread_local          |
| 线程、内部链接 | 线程  | 文件  | 内部 | 在所有块的外部，使用关键字 static 和 _Thread_local |
| 线程、无链接  | 线程  | 块   | 无  | 在块中，使用关键字 static 和 _Thread_local     |

注意，关键字 `extern` 只能用来再次声明在别处已定义过的变量。在函数外部定义变量，该变量具有外部链接属性。

除了以上介绍的存储类别，C 还提供了动态分配内存。这种内存通过调用 `malloc()` 函数系列中的一个函数来分配。这种函数返回一个可用于访问内存的指针。调用 `free()` 函数或结束程序可以释放动态分配的内存。任何可以访问指向该内存指针的函数均可访问这块内存。例如，一个函数可以把这个指针的值返回给另一个函数，那么另一个函数也可以访问该指针所指向的内存。

### B.3.3 总结：限定符

#### 关键字

使用下面关键字限定变量：

`const`、`volatile`、`restrict`

#### 一般注释

限定符用于限制变量的使用方式。不能改变初始化以后的 `const` 变量。编译器不会假设 `volatile` 变量不被某些外部代理（如，一个硬件更新）改变。`restrict` 限定的指针是访问它所指向内存的唯一方式（在特定作用域中）。

#### 属性

`const int joy = 101;` 声明创建了变量 `joy`，它的值被初始化为 101。

`volatile unsigned int incoming;` 声明创建了变量 `incoming`，该变量在程序中两次出现之间，其值可能会发生改变。

`const int * ptr = &joy;` 声明创建了指针 `ptr`，该指针不能用来改变变量 `joy` 的值，但是它可以指向其他位置。

`int * const ptr = &joy;` 声明创建了指针 `ptr`，不能改变该指针的值，即 `ptr` 只能指向 `joy`，但是可以用它来改变 `joy` 的值。

`void simple (const char * s);` 声明表明形式参数 `s` 被传递给 `simple()` 的值初始化后，`simple()` 不能改变 `s` 指向的值。

`void supple(int * const pi);` 与 `void supple(int pi[const]);` 等价。这两个声明都表明 `supple()` 函数不会改变形参 `pi`。

`void interleave(int * restrict p1, int * restrict p2, int n);` 声明表明 `p1` 和 `p2` 是访问它们所指向内存的唯一方法，这意味着这两个块不能重叠。

## B.4 参考资料 IV：表达式、语句和程序流

### B.4.1 总结：表达式和语句

在 C 语言中，对表达式可以求值，通过语句可以执行某些行为。

#### 表达式

表达式由运算符和运算对象组成。最简单的表达式是一个常量或一个不带运算符的变量，如 22 或 beepop。稍复杂些的例子是 55 + 22 和 vap = 2 \* (vip + (vup = 4))。

#### 语句

大部分语句都以分号结尾。以分号结尾的表达式都是语句，但这样的语句不一定有意义。语句分为简单语句和复合语句。简单语句以分号结尾，如下所示：

```
toes = 12;           // 赋值表达式语句
printf("%d\n", toes); // 函数调用表达式语句
;
// 空语句，什么也不做
(注意，在 C 语言中，声明不是语句。)
```

用花括号括起来的一条或多条语句是复合语句或块。如下面的 while 语句所示：

```
while (years < 100)
{
    wisdom = wisdom + 1;
    printf("%d %d\n", years, wisdom);
    years = years + 1;
}
```

### B.4.2 总结：while 语句

#### 关键字

while 语句的关键字是 while。

#### 一般注释

while 语句创建了一个循环，在 expression 为假之前重复执行。while 语句是一个入口条件循环，在下一轮迭代之前先确定是否要再次循环。因此可能一次循环也不执行。statement 可以是一个简单语句或复合语句。

#### 形式

```
while (expression)
statement
```

当 expression 为假（或 0）之前，重复执行 statement 部分。

#### 示例

```
while (n++ < 100)
    printf(" %d %d\n", n, 2*n+1);

while (fargo < 1000)
{
    fargo = fargo + step;
    step = 2 * step;
}
```

### B.4.3 总结: for 语句

#### 关键字

for 语句的关键字是 for。

#### 一般注释

for 语句使用 3 个控制表达式控制循环过程，分别用分号隔开。initialize 表达式在执行 for 语句之前只执行一次；然后对 test 表达式求值，如果表达式为真（或非零），执行循环一次；接着对 update 表达式求值，并再次检查 test 表达式。for 语句是一种入口条件循环，即在执行循环之前就决定了是否执行循环。因此，for 循环可能一次都不执行。statement 部分可以是一条简单语句或复合语句。

#### 形式:

```
for ( initialize; test; update )
    statement
```

在 test 为假或 0 之前，重复执行 statement 部分。

C99 允许在 for 循环头中包含声明。变量的作用域和生命周期被限制在 for 循环中。

#### 示例:

```
for (n = 0; n < 10 ; n++)
    printf(" %d %d\n", n, 2 * n + 1);
for (int k = 0; k < 10 ; ++k) // C99
    printf("%d %d\n", k, 2 * k+1);
```

### B.4.4 总结: do while 语句

#### 关键字

do while 语句的关键字是 do 和 while。

#### 一般注解:

do while 语句创建一个循环，在 expression 为假或 0 之前重复执行循环体中的内容。do while 语句是一种出口条件循环，即在执行完循环体后才根据测试条件决定是否再次执行循环。因此，该循环至少必须执行一次。statement 部分可是一条简单语句或复合语句。

#### 形式:

```
do
    statement
    while ( expression );
```

在 test 为假或 0 之前，重复执行 statement 部分。

#### 示例:

```
do
    scanf("%d", &number);
    while ( number != 20 );
```

### B.4.5 总结: if 语句

#### 小结: 用 if 语句进行选择

#### 关键字: if、else

#### 一般注解:

下面各形式中，statement 可以是一条简单语句或复合语句。表达式为真说明其值是非零值。

**形式 1：**

```
if (expression)
statement
```

如果 *expression* 为真，则执行 *statement* 部分。

**形式 2：**

```
if (expression)
statement1
else
statement2
```

如果 *expression* 为真，执行 *statement1* 部分；否则，执行 *statement2* 部分。

**形式 3：**

```
if (expression1)
statement1
else if (expression2)
statement2
else
statement3
```

如果 *expression1* 为真，执行 *statement1* 部分；如果 *expression2* 为真，执行 *statement2* 部分；否则，执行 *statement3* 部分。

**示例：**

```
if (legs == 4)
    printf("It might be a horse.\n");
else if (legs > 4)
    printf("It is not a horse.\n");
else /* 如果 legs < 4 */
{
    legs++;
printf("Now it has one more leg.\n");
}
```

## B.4.6 带多重选择的 switch 语句

**关键字：switch****一般注解：**

程序控制根据 *expression* 的值跳转至相应的 *case* 标签处。然后，程序流执行剩下的所有语句，除非执行到 *break* 语句进行重定向。*expression* 和 *case* 标签都必须是整数值（包括 *char* 类型），标签必须是常量或完全由常量组成的表达式。如果没有 *case* 标签与 *expression* 的值匹配，控制则转至标有 *default* 的语句（如果有的话）；否则，控制将转至紧跟在 *switch* 语句后面的语句。控制转至特定标签后，将执行 *switch* 语句中其后的所有语句，除非到达 *switch* 末尾，或执行到 *break* 语句。

**形式：**

```
switch ( expression )
{
    case label1 : statement1 // 使用 break 跳出 switch
    case label2 : statement2
    default      : statement3
}
```

可以有多个标签语句，*default* 语句可选。

**示例:**

```

switch (value)
{
    case 1 : find_sum(ar, n);
               break;
    case 2 : show_array(ar, n);
               break;
    case 3 : puts("Goodbye!");
               break;
    default : puts("Invalid choice, try again.");
               break;
}

switch (letter)
{
    case 'a' :
    case 'e' : printf("%d is a vowel\n", letter);
    case 'c' :
    case 'n' : printf("%d is in \"cane\"\n", letter);
    default : printf("Have a nice day.\n");
}

```

如果 letter 的值是'a'或'e'，就打印这 3 条消息；如果 letter 的值是'c'或'n'，则只打印后两条消息；letter 是其他值时，值打印最后一条消息。

## B.4.7 总结：程序跳转

**关键字:** break、continue、goto

**一般注解:**

这 3 种语句都能使程序流从程序的一处跳转至另一处。

**break 语句:**

所有的循环和 switch 语句都可以使用 break 语句。它使程序控制跳出当前循环或 switch 语句的剩余部分，并继续执行跟在循环或 switch 后面的语句。

**示例:**

```

while ((ch = getchar()) != EOF)
{
    putchar(ch);
    if (ch == ' ')
        break;           // 结束循环
    chcount++;
}

```

**continue 语句:**

所有的循环都可以使用 continue 语句，但是 switch 语句不行。continue 语句使程序控制跳出循环的剩余部分。对于 while 或 for 循环，程序执行到 continue 语句后会开始进入下一轮迭代。对于 do while 循环，对出口条件求值后，如有必要会进入下一轮迭代。

**示例:**

```

while ((ch = getchar()) != EOF)
{
    if (ch == ' ')
        continue; // 跳转至测试条件
    putchar(ch);
}

```

```

    chcount++;
}

```

以上程序段打印用户输入的内容并统计非空格字符

goto 语句：

goto 语句使程序控制跳转至相应标签语句。冒号用于分隔标签和标签语句。标签名遵循变量命名规则。标签语句可以出现在 goto 的前面或后面。

形式：

```
goto label ;
```

```
.
```

```
.
```

```
label : statement
```

示例：

```
top : ch = getchar();
```

```
.
```

```
.
```

```
if (ch != 'y')
```

```
goto top;
```

## B.5 参考资料 V：新增 C99 和 C11 的 ANSI C 库

ANSI C 库把函数分成不同的组，每个组都有相关联的头文件。本节将概括地介绍库函数，列出头文件并简要描述相关的函数。文中会较详细地介绍某些函数（例如，一些 I/O 函数）。欲了解完整的函数说明，请参考具体实现的文档或参考手册，或者试试这个在线参考：[http://www.acm.uiuc.edu/webmonkeys/book/c\\_guide/](http://www.acm.uiuc.edu/webmonkeys/book/c_guide/)。

### B.5.1 断言：assert.h

assert.h 头文件中把 assert() 定义为一个宏。在包含 assert.h 头文件之前定义宏标识符 NDEBUG，可以禁用 assert() 宏。通常用一个关系表达式或逻辑表达式作为 assert() 的参数，如果运行正常，那么程序在执行到该点时，作为参数的表达式应该为真。表 B.5.1 描述了 assert() 宏。

表 B.5.1 断言宏

| 原型                      | 描述                                                                                            |
|-------------------------|-----------------------------------------------------------------------------------------------|
| void assert(int exprs); | 如果 exprs 为 0 (或真)，宏什么也不做。如果 exprs 为 0 (或假)，assert() 就显示该表达式及其所在的行号和文件名。然后，assert() 调用 abort() |

C11 新增了 static\_assert 宏，展开为 \_Static\_assert。\_Static\_assert 是一个关键字，被认为是一种声明形式。它以这种方式提供一个编译时检查：

```
_Static_assert(常量表达式,字符串字面量);
```

如果对常量表达式求值为 0，编译器会给出一条包含字符串字面量的错误消息；否则，没有任何效果。

### B.5.2 复数：complex.h (C99)

C99 标准支持复数计算，C11 进一步支持了这个功能。实现除提供 \_Complex 类型外还可以选择是否提供 \_Imaginary 类型。在 C11 中，可以选择是否提供这两种类型。C99 规定，实现必须提供 \_Complex 类型，但是 \_Imaginary 类型为可选，可以提供或不提供。附录 B 的参考资料 VIII 中进一步讨论了 C 如何

支持复数。`complex.h` 头文件中定义了表 B.5.2 所列的宏。

表 B.5.2 `complex.h` 宏

| 宏                         | 描述                                                                  |
|---------------------------|---------------------------------------------------------------------|
| <code>complex</code>      | 展开为类型关键字 <code>_Complex</code>                                      |
| <code>_Complex_I</code>   | 展开为 <code>const float _Complex</code> 类型的表达式, 其值的平方是 -1             |
| <code>imaginary</code>    | 如果支持虚数类型, 展开为类型关键字 <code>_Imaginary</code>                          |
| <code>_Imaginary_I</code> | 如果支持虚数类型, 展开为 <code>const float _Imaginary</code> 类型的表达式, 其值的平方是 -1 |
| <code>I</code>            | 展开为 <code>_Complex_I</code> 或 <code>_Imaginary_I</code>             |

对于实现复数方面, C 和 C++ 不同。C 通过 `complex.h` 头文件支持, 而 C++ 通过 `complex` 头文件支持。而且, C++ 使用类来定义复数类型。

可以使用 `STDC CX_LIMITED_RANGE` 编译指令来表明是使用普通的数学公式 (设置为 `on` 时), 还是要特别注意极值 (设置为 `off` 时):

```
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on
```

库函数分为 3 种: `double`、`float`、`long double`。表 B.5.3 列出了 `double` 版本的函数。`float` 和 `long double` 版本只需要在函数名后面分别加上 `f` 和 `l`。即 `csinf()` 就是 `csin()` 的 `float` 版本, 而 `csinl()` 是 `csin()` 的 `long double` 版本。另外要注意, 角度的单位是弧度。

表 B.5.3 复数函数

| 原型                                                                                     | 描述                                      |
|----------------------------------------------------------------------------------------|-----------------------------------------|
| <code>double complex cacos(double complex z);</code>                                   | 返回 <code>z</code> 的复数反余弦                |
| <code>double complex casin(double complex z);</code>                                   | 返回 <code>z</code> 的复数反正弦                |
| <code>double complex catan(double complex z);</code>                                   | 返回 <code>z</code> 的复数反正切                |
| <code>double complex ccos(double complex z);</code>                                    | 返回 <code>z</code> 的复数余弦                 |
| <code>double complex csin(double complex z);</code>                                    | 返回 <code>z</code> 的复数正弦                 |
| <code>double complex ctan(double complex z);</code>                                    | 返回 <code>z</code> 的复数正切                 |
| <code>double complex cacosh(double complex z);</code>                                  | 返回 <code>z</code> 的复数反双曲余弦              |
| <code>double complex casinh(double complex z);</code>                                  | 返回 <code>z</code> 的复数反双曲正弦              |
| <code>double complex catanh(double complex z);</code>                                  | 返回 <code>z</code> 的复数反双曲正切              |
| <code>double complex ccosh(double complex z);</code>                                   | 返回 <code>z</code> 的复数双曲余弦               |
| <code>double complex csinh(double complex z);</code>                                   | 返回 <code>z</code> 的复数双曲正弦               |
| <code>double complex ctanh(double complex z);</code>                                   | 返回 <code>z</code> 的复数双曲正切               |
| <code>double complex cexp(double complex z);</code>                                    | 返回 $e$ 的 <code>z</code> 次幂复数值           |
| <code>double complex clog(double complex z);</code>                                    | 返回 <code>z</code> 的自然对数 (以 $e$ 为底) 的复数值 |
| <code>double cabs(double complex z);</code>                                            | 返回 <code>z</code> 的绝对值 (或大小)            |
| <code>double complex cpows(double complex z,</code><br><code>double complex y);</code> | 返回 <code>z</code> 的 <code>y</code> 次幂   |
| <code>double complex csqrt(double complex z);</code>                                   | 返回 <code>z</code> 的复数平方根                |

续表

| 原型                                       | 描述                      |
|------------------------------------------|-------------------------|
| double carg(double complex z);           | 以弧度为单位返回 z 的相位角（或幅角）    |
| double cimag(double complex z);          | 以实数形式返回 z 的虚部           |
| double complex conj(double complex z);   | 返回 z 的共轭复数              |
| double complex cproj(double complex z);  | 返回 z 在黎曼球面上的投影          |
| double complex CMPLX(double x,double y); | 返回实部为 x、虚部为 y 的复数 (C11) |
| double creal(double complex z);          | 以实数形式返回 z 的实部           |

### B.5.3 字符处理: ctype.h

这些函数都接受 int 类型的参数，这些参数可以表示为 unsigned char 类型的值或 EOF。使用其他值的效果是未定义的。在表 B.5.4 中，“真”表示“非 0 值”。对一些定义的解释取决于当前的本地设置，这些由 locale.h 中的函数来控制。该表显示了在解释本地化的“C”时要用到的一些函数。

表 B.5.4 字符处理函数

| 原型                   | 描述                                                    |
|----------------------|-------------------------------------------------------|
| int isalnum(int c);  | 如果 c 是字母或数字，则返回真                                      |
| int isalpha(int c);  | 如果 c 是字母，则返回真                                         |
| int isblank(int c);  | 如果 c 是空格或水平制表符，则返回真 (C99)                             |
| int iscntrl(int c);  | 如果 c 是控制字符 (如 Ctrl+B)，则返回真                            |
| int isdigit(int c);  | 如果 c 是数字，则返回真                                         |
| int isgraph(int c);  | 如果 c 是非空格打印字符，则返回真                                    |
| int islower(int c);  | 如果 c 是小写字符，则返回真                                       |
| int isprint(int c);  | 如果 c 是打印字符，则返回真                                       |
| int ispunct(int c);  | 如果 c 是标点字符 (除了空格、字母、数字以外的字符)，则返回真                     |
| int isspace(int c);  | 如果 c 是空白字符 (空格、换行符、换页符、回车符、垂直或水平制表符，或者其他实现定义的字符)，则返回真 |
| int isupper(int c);  | 如果 c 是大写字符，则返回真                                       |
| int isxdigit(int c); | 如果 c 是十六进制数字字符，则返回真                                   |
| int tolower(int c);  | 如果 c 是大写字符，则返回其小写字符；否则返回 c                            |
| int toupper(int c);  | 如果 c 是小写字符，则返回其大写字符；否则返回 c                            |

### B.5.4 错误报告: errno.h

errno.h 头文件支持较老式的错误报告机制。该机制提供一个标识符（或有时称为宏）ERRNO 可访问的外部静态内存位置。一些库函数把一个值放进这个位置用于报告错误，然后包含该头文件的程序就可以通过查看 ERRNO 的值检查是否报告了一个特定的错误。ERRNO 机制被认为不够艺术，而且设置 ERRNO 值也不需要数学函数了。标准提供了 3 个宏值表示特殊的错误，但是有些实现会提供更多。表 B.5.5 列出了这些标准宏。

表 B.5.5 `errno.h` 宏

| 宏      | 含义                |
|--------|-------------------|
| EDOM   | 函数调用中的域错误（参数越界）   |
| ERANGE | 函数返回值的范围错误（返回值越界） |
| EILSEQ | 宽字符转换错误           |

## B.5.5 浮点环境: `fenv.h` (C99)

C99 标准通过 `fenv.h` 头文件提供访问和控制浮点环境。

浮点环境 (*floating-point environment*) 由一组状态标志 (*status flag*) 和控制模式 (*control mode*) 组成。在浮点计算中发生异常情况时 (如, 被零除), 可以“抛出一个异常”。这意味着该异常情况设置了一个浮点环境标志。控制模式值可以进行一些控制, 例如控制舍入的方向。`fenv.h` 头文件定义了一组宏表示多种异常情况和控制模式, 并提供了与环境交互的函数原型。头文件还提供了一个编译指令来启用或禁用访问浮点环境的功能。

下面的指令开启访问浮点环境:

```
#pragma STDC FENV_ACCESS on
```

下面的指令关闭访问浮点环境:

```
#pragma STDC FENV_ACCESS off
```

应该把该编译指示放在所有外部声明之前或者复合块的开始处。在遇到下一个编译指示之前、或到达文件末尾 (外部指令)、或到达复合语句的末尾 (块指令), 当前编译指示一直有效。

头文件定义了两种类型, 如表 B.5.6 所示。

表 B.5.6 `fenv.h` 类型

| 类型                     | 表示       |
|------------------------|----------|
| <code>fenv_t</code>    | 整个浮点环境   |
| <code>fexcept_t</code> | 浮点状态标志集合 |

头文件定义了一些宏, 表示一些可能发生的浮点异常情况控制状态。其他实现可能定义更多的宏, 但是必须以 `FE_` 开头, 后面跟大写字母。表 B.5.7 列出了一些标准异常宏。

表 B.5.7 `fenv.h` 中的标准异常宏

| 宏                          | 含义                                      |
|----------------------------|-----------------------------------------|
| <code>FE_DIVBYZERO</code>  | 抛出被零除异常                                 |
| <code>FE_INEXACT</code>    | 抛出不精确值异常                                |
| <code>FE_INVALID</code>    | 抛出无效值异常                                 |
| <code>FE_OVERFLOW</code>   | 抛出上溢异常                                  |
| <code>FE_UNDERFLOW</code>  | 抛出下溢异常                                  |
| <code>FE_ALL_EXCEPT</code> | 实现支持的所有浮点异常的按位或                         |
| <code>FE_DOWNWARD</code>   | 向下舍入                                    |
| <code>FE_TONEAREST</code>  | 向最近的舍入                                  |
| <code>FE_TOWARDZERO</code> | 趋 0 舍入                                  |
| <code>FE_UPWARD</code>     | 向上舍入                                    |
| <code>FE_DFL_ENV</code>    | 表示默认环境, 类型是 <code>const fenv_t *</code> |

表 B.5.8 中列出了 `fenv.h` 头文件中的标准函数原型。注意，常用的参数值和返回值与表 B.5.7 中的宏相对应。例如，`FE_UPWARD` 是 `fesetround()` 的一个合适参数。

表 B.5.8 `fenv.h` 中的标准函数原型

| 原型                                                                      | 描述                                                                                                                                                                   |
|-------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void feclearexcept(int excepts);</code>                           | 清理 <code>excepts</code> 表示的异常                                                                                                                                        |
| <code>void fegetexceptflag(fexcept_t *flagp, int excepts);</code>       | 把 <code>excepts</code> 指明的浮点状态标志储存在 <code>flagp</code> 指向的对象中                                                                                                        |
| <code>void feraiseexcept(int excepts);</code>                           | 抛出 <code>excepts</code> 指定的异常                                                                                                                                        |
| <code>void fesetexceptflag(const fexcept_t *flagp, int excepts);</code> | 把 <code>excepts</code> 指明的浮点状态标志设置为 <code>flagp</code> 的值；在此之前， <code>fegetexceptflag()</code> 调用应该设置 <code>flagp</code> 的值                                          |
| <code>int fetestexcept(int excepts);</code>                             | 测试 <code>excepts</code> 指定的状态标志；该函数返回指定状态标志的按位或                                                                                                                      |
| <code>int fegetround(void);</code>                                      | 返回当前的舍入方向                                                                                                                                                            |
| <code>int fesetround(int round);</code>                                 | 把舍入方向设置为 <code>round</code> 的值；当且仅当设置成功时，函数返回 0                                                                                                                      |
| <code>void fegetenv(fenv_t *envp);</code>                               | 把当前环境储存至 <code>envp</code> 指向的位置中                                                                                                                                    |
| <code>int feholdexcept(fenv_t *envp);</code>                            | 把当前浮点环境储存至 <code>envp</code> 指向的位置中，清除浮点状态标志，然后如果可能的话就设置非停模式（ <i>nonstop mode</i> ），在这种模式中即使发生异常也继续执行。当且仅当执行成功时，函数返回 0                                               |
| <code>void fesetenv(const fenv_t *envp);</code>                         | 建立 <code>envp</code> 表示的浮点环境； <code>envp</code> 应指向一个之前通过调用 <code>fegetenv()</code> 、 <code>feholdexcept()</code> 或浮点环境宏设置的数据对象                                      |
| <code>void feupdateenv(const fenv_t *envp);</code>                      | 函数在自动存储区中储存当前抛出的异常，建立 <code>envp</code> 指向的对象表示的浮点环境，然后抛出已储存的浮点异常； <code>envp</code> 应指向一个之前通过调用 <code>fegetenv()</code> 、 <code>feholdexcept()</code> 或浮点环境宏设置的数据对象 |

## B.5.6 浮点特性: `float.h`

`float.h` 头文件中定义了一些表示各种限制和形参的宏。表 B.5.9 列出了这些宏，C11 新增的宏以斜体并缩进标出。许多宏都涉及下面的浮点表示模型：

$$x = sb^e \sum_{k=1}^p f_k b^{-k}$$

如果第 1 个数  $f_1$  是非 0（且  $x$  是非 0），该数字被称为标准化浮点数。附录 B 的参考资料 VIII 中将更详细地解释一些宏。

表 B.5.9 `float.h` 宏

| 宏                                  | 含义                                    |
|------------------------------------|---------------------------------------|
| <code>FLT_ROUNDS</code>            | 默认舍入方案                                |
| <code>FLT_EVAL_METHOD</code>       | 浮点表达式求值的默认方案                          |
| <code>FLT_HAS_SUBNORM</code>       | 存在或缺少 <code>float</code> 类型的反常值       |
| <code>DBL_HAS_SUBNORM</code>       | 存在或缺少 <code>double</code> 类型的反常值      |
| <code>LDBL_HAS_SUBNORM</code>      | 存在或缺少 <code>long double</code> 类型的反常值 |
| <code>FLT_RADIX<sup>1</sup></code> | 指数表示法中使用的进制数 (b)，最小值为 2               |

<sup>1</sup> `FLT_RADIX` 用于表示 3 种浮点数类型的基数。——译者注

续表

| 宏                | 含义                                                       |
|------------------|----------------------------------------------------------|
| FLT_MANT_DIG     | 以 FLT_RADIX 进制表示的 float 类型数的位数 (模型中的 p)                  |
| DBL_MANT_DIG     | 以 FLT_RADIX 进制表示的 double 类型数的位数 (模型中的 p)                 |
| LDBL_MANT_DIG    | 以 FLT_RADIX 进制表示的 long double 类型数的位数 (模型中的 p)            |
| FLT_DECIMAL_DIG  | 在 b 进制和十进制相互转换不损失精度的前提下, float 类型的十进制数的位数 (最小值是 6)       |
| DBL_DECIMAL_DIG  | 在 b 进制和十进制相互转换不损失精度的前提下, double 类型的十进制数的位数 (最小值是 10)     |
| LDBL_DECIMAL_DIG | 在 b 进制和十进制相互转换不损失精度的前提下, long double 类型的十进制数的位数(最小值是 10) |
| DECIMAL_DIG      | 在 b 进制与十进制相互转换不损失精度的前提下, 浮点类型十进制数的最大个数 (最小值为 10)         |
| FLT_DIG          | 在不损失精度的前提下, float 类型可表示的十进制数位数 (最小值为 6)                  |
| DBL_DIG          | 在不损失精度的前提下, double 类型可表示的十进制数位数 (最小值为 10)                |
| LDBL_DIG         | 在不损失精度的前提下, long double 类型可表示的十进制数位数 (最小值为 10)           |
| FLT_MIN_EXP      | float 类型 e 表示法, 指数的最小负正整数值                               |
| DBL_MIN_EXP      | double 类型 e 表示法, 指数的最小负正整数值                              |
| LDBL_MIN_EXP     | long double 类型 e 表示法, 指数的最小负正整数值                         |
| FLT_MIN_10_EXP   | 用 10 的 x 次幂表示规范化 float 类型数时, x 的最小负整数值 (不超过 -37)         |
| DBL_MIN_10_EXP   | 用 10 的 x 次幂表示规范化 double 类型数时, x 的最小负整数值 (不超过 -37)        |
| LDBL_MIN_10_EXP  | 用 10 的 x 次幂表示规范化 long double 类型数时, x 的最小负整数值 (不超过 -37)   |
| FLT_MAX_EXP      | float 类型 e 表示法, 指数的最大正整数值                                |
| DBL_MAX_EXP      | double 类型 e 表示法, 指数的最大正整数值                               |
| LDBL_MAX_EXP     | long double 类型 e 表示法, 指数的最大正整数值                          |
| FLT_MAX_10_EXP   | 用 10 的 x 次幂表示规范化 float 类型数时, x 的最大正整数值 (至少 +37)          |
| DBL_MAX_10_EXP   | 用 10 的 x 次幂表示规范化 double 类型数时, x 的最大正整数值 (至少 +37)         |
| LDBL_MAX_10_EXP  | 用 10 的 x 次幂表示规范化 long double 类型数时, x 的最大正整数值 (至少 +37)    |
| FLT_MAX          | float 类型的最大有限值 (至少 1E+37)                                |
| DBL_MAX          | double 类型的最大有限值 (至少 1E+37)                               |
| LDBL_MAX         | long double 类型的最大有限值 (至少 1E+37)                          |
| FLT_EPSILON      | float 类型比 1 大的最小值与 1 的差值 (不超过 1E-9)                      |
| DBL_EPSILON      | double 类型比 1 大的最小值与 1 的差值 (不超过 1E-9)                     |
| LDBL_EPSILON     | long double 类型比 1 大的最小值与 1 的差值 (不超过 1E-9)                |
| FLT_MIN          | 标准化 float 类型的最小正值 (不超过 1E-37)                            |
| DBL_MIN          | 标准化 double 类型的最小正值 (不超过 1E-37)                           |
| LDBL_MIN         | 标准化 long double 类型的最小正值 (不超过 1E-37)                      |
| FLT_TRUE_MIN     | float 类型的最小正值 (不超过 1E-37)                                |
| DBL_TRUE_MIN     | double 类型的最小正值 (不超过 1E-37)                               |
| LDBL_TRUE_MIN    | long double 类型的最小正值 (不超过 1E-37)                          |

## B.5.7 整数类型的格式转换：inttypes.h

该头文件定义了一些宏可用作转换说明来扩展整数类型。参考资料 VI “扩展的整数类型”将进一步讨论。该头文件还声明了这个类型：`imaxdiv_t`。这是一个结构类型，表示 `idivmax()` 函数的返回值。

该头文件中还包含 `stdint.h`，并声明了一些使用最大长度整数类型的函数，这种整数类型在 `stdint.h` 中声明为 `intmax`。表 B.5.10 列出了这些函数。

表 B.5.10 使用最大长度整数的函数

| 原型                                                                                                             | 描述                                                                       |
|----------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| <code>intmax_t imaxabs(intmax_t j);</code>                                                                     | 返回 <code>j</code> 的绝对值                                                   |
| <code>imaxdiv_t imaxdiv(intmax_t numer,<br/>intmax_t denom);</code>                                            | 单独计算 <code>numer/denom</code> 的商和余数，并把两个计算结果储存在返回的结构中                    |
| <code>intmax_t strtoimax(const char *<br/>restrict nptr, char ** restrict endptr,<br/>int base);</code>        | 相当于 <code>strtol()</code> 函数，但是该函数把字符串转换成 <code>intmax_t</code> 类型并返回该值  |
| <code>uintmax_t strtoumax(const char *<br/>restrict nptr, char ** restrict endptr,<br/>int base);</code>       | 相当于 <code>strtoul()</code> 函数，但是该函数把字符串转换成 <code>intmax_t</code> 类型并返回该值 |
| <code>intmax_t wcstoimax(const wchar_t *<br/>restrict nptr, wchar_t ** restrict<br/>endptr, int base);</code>  | <code>strtoimax()</code> 函数的 <code>wchar_t</code> 版本                     |
| <code>uintmax_t wcstoumax(const wchar_t *<br/>restrict nptr, wchar_t ** restrict<br/>endptr, int base);</code> | <code>strtoumax()</code> 函数的 <code>wchar_t</code> 版本                     |

## B.5.8 可选拼写：iso646.h

该头文件提供了 11 个宏，扩展了指定的运算符，如表 B.5.11 所列。

表 B.5.11 可选拼写

| 宏                  | 运算符                     | 宏                   | 运算符                 | 宏                   | 运算符                |
|--------------------|-------------------------|---------------------|---------------------|---------------------|--------------------|
| <code>and</code>   | <code>&amp;&amp;</code> | <code>and_eq</code> | <code>&amp;=</code> | <code>bitand</code> | <code>&amp;</code> |
| <code>bitor</code> | <code> </code>          | <code>compl</code>  | <code>~</code>      | <code>not</code>    | <code>!</code>     |
| <code>not</code>   | <code>!=</code>         | <code>or</code>     | <code>  </code>     | <code>or_eq</code>  | <code> =</code>    |
| <code>xor</code>   | <code>^</code>          | <code>xor_eq</code> | <code>^=</code>     |                     |                    |

## B.5.9 本地化：locale.h

本地化是一组设置，用于控制一些特定的设置项，如表示小数点的符号。本地值储存在 `struct lconv` 类型的结构中，定义在 `locale.h` 头文件中。可以用一个字符串来指定本地化，该字符串指定了一个结构成员的特殊值。默认的本地化由字符串 "C" 指定。表 B.5.12 列出了本地化函数，后面做了简要说明。

表 B.5.12 本地化函数

| 原型                                                      | 描述                                                                                                                |
|---------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| char * setlocale(int category,<br>const char * locale); | 该函数把某些值设置为本地和 locale 指定的值。category 的值决定要设置哪些本地值（参见 B.5.13）。如果成功设置本地化，该函数将返回一个在新本地化中与指定类别相关联的指针；如果不能完成本地化请求，则返回空指针 |
| struct lconv<br>*localeconv(void);                      | 返回一个指向 struct lconv 类型结构的指针，该结构中储存着当前的本地值                                                                         |

setlocale() 函数的 locale 形参所需的值可能是默认值"C"，也可能是""，表示实现定义的本地环境。实现可以定义更多的本地化设置。category 形参的值可能由表 B.5.13 中所列的宏表示。

表 B.5.13 category 宏

| 原型          | 描述                                   |
|-------------|--------------------------------------|
| NULL        | 本地化设置不变，返回指向当前本地化的指针                 |
| LC_ALL      | 改变所有的本地值                             |
| LC_COLLATE  | 改变 strcoll() 和 strxfrm() 所用的排列顺序的本地值 |
| LC_CTYPE    | 改变字符处理函数和多字节函数的本地值                   |
| LC_MONETARY | 改变货币格式信息的本地值                         |
| LC_NUMERIC  | 改变十进制小数点符号和格式化 I/O 使用的非货币格式本地值       |
| LC_TIME     | 改变 strftime() 所用的时间格式本地值             |

表 B.5.14 列出了 struct lconv 结构所需的成员。

表 B.5.14 struct lconv 所需的成员

| 成员变量                    | 描述                                                                                |
|-------------------------|-----------------------------------------------------------------------------------|
| char *decimal_point     | 非货币值的小数点字符                                                                        |
| char *thousands_sep     | 非货币值中小数点前面的千位分隔符                                                                  |
| char *grouping          | 一个字符串，表示非货币量中每组数字的大小                                                              |
| char *int_curr_symbol   | 国际货币符号                                                                            |
| char *currency_symbol   | 本地货币符号                                                                            |
| char *mon_decimal_point | 货币值的小数点符号                                                                         |
| char *mon_thousands_sep | 货币值的千位分隔符                                                                         |
| char *mon_grouping      | 一个字符串，表示货币量中每组数字的大小                                                               |
| char *positive_sign     | 指明非负格式化货币值的字符串                                                                    |
| char *negative_sign     | 指明负格式化货币值的字符串                                                                     |
| char int_frac_digits    | 国际格式化货币值中，小数点后面的数字个数                                                              |
| char frac_digits        | 本地格式化货币值中，小数点后面的数字个数                                                              |
| char p_cs_precedes      | 如果该值为 1，则 currency_symbol 在非负格式化货币值的前面；<br>如果该值为 0，则 currency_symbol 在非负格式化货币值的后面 |

续表

| 成员变量                    | 描述                                                                                                                                  |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| char p_sep_by_space     | 如果该值为 1，则用空格把 currency_symbol 和非负格式化货币值隔开；如果该值为 0，则不用空格分隔 currency_symbol 和非负格式化货币值                                                 |
| char n_cs_precedes      | 如果该值为 1，则 currency_symbol 在负格式化货币值的前面；如果该值为 0，则 currency_symbol 在负格式化货币值的后面                                                         |
| char n_sep_by_space     | 如果该值为 1，则用空格把 currency_symbol 和负格式化货币值隔开；如果该值为 0，则不用空格分隔 currency_symbol 和负格式化货币值                                                   |
| char p_sign_posn        | 其值表示 positive_sign 字符串的位置：<br>0 表示用圆括号把数值和货币符号括起来<br>1 表示字符串在数值和货币符号前面<br>2 表示字符串在数值和货币符号后面<br>3 表示直接把字符串放在货币前面<br>4 表示字符串紧跟在货币符号后面 |
| char n_sign_posn        | 其值表示 negative_sign 字符串的位置，含义与 p_sign_posn 相同                                                                                        |
| char int_p_cs_precedes  | 如果该值为 1，则 int_currency_symbol 在非负格式化货币值的前面；如果该值为 0，则 int_currency_symbol 在非负格式化货币值的后面                                               |
| char int_p_sep_by_space | 如果该值为 1，则用空格把 int_currency_symbol 和非负格式化货币值隔开；如果该值为 0，则不用空格分隔 int_currency_symbol 和非负格式化货币值                                         |
| char int_n_cs_precedes  | 如果该值为 1，则 int_currency_symbol 在负格式化货币值的前面；如果该值为 0，则 int_currency_symbol 在负格式化货币值的后面                                                 |
| char int_n_sep_by_space | 如果该值为 1，则用空格把 int_currency_symbol 和负格式化货币值隔开；如果该值为 0，则不用空格分隔 int_currency_symbol 和负格式化货币值                                           |
| char int_p_sign_posn    | 其值表示 positive_sign 相对于非负国际格式化货币值的位置                                                                                                 |
| char int_n_sign_posn    | 其值表示 negative_sign 相对于负国际格式化货币值的位置                                                                                                  |

### B.5.10 数学库: math.h

C99 为 math.h 头文件定义了两种类型: float\_t 和 double\_t。这两种类型分别与 float 和 double 类型至少等宽，是计算 float 和 double 时效率最高的类型。

该头文件还定义了一些宏，如表 B.5.15 所列。该表中除了 HUGE\_VAL 外，都是 C99 新增的。在参考资料 VIII：“C99 数值计算增强”中会进一步详细介绍。

表 B.5.15 math.h 宏

| 宏         | 描述                                                       |
|-----------|----------------------------------------------------------|
| HUGE_VAL  | 正双精度常量，不一定能用浮点数表示；在过去，函数的计算结果超过了可表示的最大值时，就用它作为函数的返回值     |
| HUGE_VALF | 与 HUGE_VAL 类似，适用于 float 类型                               |
| HUGE_VALL | 与 HUGE_VAL 类似，适用于 long double 类型                         |
| INFINITY  | 如果允许的话，展开为一个表示无符号或正无穷大的常量 float 表达式；否则，展开为一个在编译时溢出的正浮点常量 |

续表

| 宏                | 描述                                                                                            |
|------------------|-----------------------------------------------------------------------------------------------|
| NAN              | 当且仅当实现支持 float 类型的 NaN 时才被定义 (NaN 是 Not-a-Number 的缩写, 表示“非数”, 用于处理计算中的错误情况, 如除以 0.0 或求负数的平方根) |
| FP_INFINITE      | 分类数, 表示一个无穷大的浮点值                                                                              |
| FP_NAN           | 分类数, 表示一个不是数的浮点值                                                                              |
| FP_NORMAL        | 分类数, 表示一个正常的浮点值                                                                               |
| FP_SUBNORMAL     | 分类数, 表示一个低于正常浮点值的值 (精度被降低)                                                                    |
| FP_ZERO          | 分类数, 表示 0 的浮点值                                                                                |
| FP_FAST_FMA      | (可选) 如果已定义, 对于 double 类型的运算对象, 该宏表明 fma() 函数与先乘法运算后加法运算的速度相当或更快                               |
| FP_FAST_FMAF     | (可选) 如果已定义, 对于 float 类型的运算对象, 该宏表明 fmaf() 函数与先乘法运算后加法运算的速度相当或更快                               |
| FP_FAST_FMAL     | (可选) 如果已定义, 对于 long double 类型的运算对象, 该宏表明 fmwl() 函数与先乘法运算后加法运算的速度相当或更快                         |
| FP_ILOGB0        | 整型常量表达式, 表示 ilogn(0) 的返回值                                                                     |
| FP_ILOGBNAN      | 整型常量表达式, 表示 ilogn(NaN) 的返回值                                                                   |
| MATH_ERRNO       | 展开为整型常量 1                                                                                     |
| MATH_ERREXCEPT   | 展开为整型常量 2                                                                                     |
| math_errhandling | 值为 MATH_ERRNO、MATH_ERREXCEPT 或这两个值的按位或                                                        |

数学函数通常使用 double 类型的值。C99 新增了这些函数的 float 和 long double 版本, 其函数名为分别在原函数名后添加 f 后缀和 l 后缀。例如, C 语言现在提供这些函数原型:

```
double sin(double);
float sinf(float);
long double sinl(long double);
```

篇幅有限, 表 B.5.16 仅列出了数学库中这些函数的 double 版本。该表引用了 FLT\_RADIX, 该常量定义在 float.h 中, 代表内部浮点表示法中幂的底数。最常用的值是 2。

表 B.5.16 ANSI C 标准数学函数

| 原型                             | 描述                             |
|--------------------------------|--------------------------------|
| int classify(real-floating x); | C99 宏, 返回适合 x 的浮点分类值           |
| int isfinite(real-floating x); | C99 宏, 当且仅当 x 为有穷时返回一个非 0 值    |
| int isfin(real-floating x);    | C99 宏, 当且仅当 x 为无穷时返回一个非 0 值    |
| int isnan(real-floating x);    | C99 宏, 当且仅当 x 为 NaN 时返回一个非 0 值 |
| int isnormal(real-floating x); | C99 宏, 当且仅当 x 为正常数时返回一个非 0 值   |
| int signbit(real-floating x);  | C99 宏, 当且仅当 x 的符号为负时返回一个非 0 值  |
| double acos(double x);         | 返回余弦为 x 的角度 (0~π 弧度)           |
| double asin(double x);         | 返回正弦为 x 的角度 (-π/2~π/2 弧度)      |
| double atan(double x);         | 返回正切为 x 的角度 (-π/2~π/2 弧度)      |

续表

| 原型                                 | 描述                                                         |
|------------------------------------|------------------------------------------------------------|
| double atan2(double y, double x);  | 返回正切为 $y/x$ 的角度 ( $-\pi \sim \pi$ 弧度)                      |
| double cos(double x);              | 返回 $x$ (弧度) 的余弦值                                           |
| double sin(double x);              | 返回 $x$ (弧度) 的正弦值                                           |
| double tan(double x);              | 返回 $x$ (弧度) 的正切值                                           |
| double cosh(double x);             | 返回 $x$ 的双曲余弦值                                              |
| double sinh(double x);             | 返回 $x$ 的双曲正弦值                                              |
| double tanh(double x);             | 返回 $x$ 的双曲切值                                               |
| double exp(double x);              | 返回 $e$ 的 $x$ 次幂 ( $e^x$ )                                  |
| double exp2(double x);             | 返回 $2$ 的 $x$ 次幂 ( $2^x$ )                                  |
| double expm1(double x);            | 返回 $e^x - 1$ (C99)                                         |
| double frexp(double v, int *pt_e); | 把 $v$ 的值分成两部分, 一个是返回的规范化小数; 一个是 $2$ 的幂, 储存在 $pt\_e$ 指向的位置上 |
| int ilogb(double x);               | 以 signed int 类型返回 $x$ 的指数 (C99)                            |
| double ldexp(double x, int p);     | 返回 $x$ 乘以 $2$ 的 $p$ 次幂 (即 $x * 2^p$ )                      |
| double log(double x);              | 返回 $x$ 的自然对数                                               |
| double log10(double x);            | 返回以 $10$ 为底 $x$ 的对数                                        |
| double log1p(double x);            | 返回 $\log(1 + x)$ (C99)                                     |
| double log2(double x);             | 返回以 $2$ 为底 $x$ 的对数 (C99)                                   |
| double logb(double x);             | 返回 FLT_RADIX (系统内部浮点表示法中幂的底数) 为底 $x$ 的有符号对数 (C99)          |
| double modf(double x, double *p);  | 把 $x$ 分成整数部分和小数部分, 两部分的符号相同, 返回小数部分, 并把整数部分储存在 $p$ 所指向的位置上 |
| double scalbn(double x, int n);    | 返回 $x * \text{FLT\_RADIX}^n$ (C99)                         |
| double scalbln(double x, long n);  | 返回 $x * \text{FLT\_RADIX}^n$ (C99)                         |
| double cbrt(double x);             | 返回 $x$ 的立方根 (C99)                                          |
| double hypot(double x, double y);  | 返回 $x$ 平方与 $y$ 平方之和的平方根 (C99)                              |
| double pow(double x, double y);    | 返回 $x$ 的 $y$ 次幂                                            |
| double sqrt(double x);             | 返回 $x$ 的平方根                                                |
| double erf(double x);              | 返回 $x$ 的误差函数 (C99)                                         |
| double lgamma(double x);           | 返回 $x$ 的伽马函数绝对值的自然对数 (C99)                                 |
| double tgamma(double x);           | 返回 $x$ 的伽马函数 (C99)                                         |
| double ceil(double x);             | 返回不小于 $x$ 的最小整数值                                           |
| double fabs(double x);             | 返回 $x$ 的绝对值                                                |
| double floor(double x);            | 返回不大于 $x$ 的最大值                                             |
| double nearbyint(double x);        | 以浮点格式把 $x$ 四舍五入为最接近的整数; 使用浮点环境指定的舍入规则 (C99)                |

续表

| 原型                                                    | 描述                                                                                                                                                                            |
|-------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| double rint(double x);                                | 与 nearbyint() 类似，但是该函数会抛出“不精确”异常                                                                                                                                              |
| long int lrint(double x);                             | 以 long int 格式把 x 舍入为最接近的整数；使用浮点环境指定的舍入规则 (C99)                                                                                                                                |
| long long int llrint(double x);                       | 以 long long int 格式把 x 舍入为最接近的整数；使用浮点环境指定的舍入规则 (C99)                                                                                                                           |
| double round(double x);                               | 以浮点格式把 x 舍入为最接近的整数，总是四舍五入                                                                                                                                                     |
| long int lround(double x);                            | 与 round() 类似，但是该函数返回值的类型是 long int                                                                                                                                            |
| long long int llround(double x);                      | 与 round() 类似，但是该函数返回值的类型是 long long int                                                                                                                                       |
| double trunc(double x);                               | 以浮点格式把 x 舍入为最接近的整数，其结果的绝对值不大于 x 的绝对值 (C99)                                                                                                                                    |
| int fmod(double x, double y);                         | 返回 x/y 的小数部分，如果 y 不是 0，则其计算结果的符号与 x 相同，而且该结果的绝对值要小于 y 的绝对值                                                                                                                    |
| double remainder(double x, double y);                 | 返回 x 除以 y 的余数，IEC 60559 定义为 $x - n*y$ , n 取与 $x/y$ 最接近的整数；如果 $(n - x/y)$ 的绝对值是 $1/2$ , n 取偶数                                                                                  |
| double remquo(double x, double y, int *quo);          | 返回与 remainder() 相同的值；把 $x/y$ 的整数大小求模 $2^k$ 的值储存在 quo 所指向的位置中，符号与 $x/y$ 的符号相同，其中 k 为整数，至少是 3，具体值因实现而异 (C99)                                                                    |
| double copysign(double x, double y);                  | 返回 x 的大小和 y 的符号 (C99)                                                                                                                                                         |
| double nan(const char *tagp);                         | 返回以 double 类型表示的 quiet NaN <sup>1</sup> ；<br>nan("n-char-seq") 与 strtod("NAN(n-char-seq)", (char **)NULL) 等价；nan("") 与 strtod("NAN()", (char**)NULL) 等价。如果不支持 quiet NaN，则返回 0 |
| double nextafter(double x, double y);                 | 返回 x 在 y 方向上可表示的最接近的 double 类型值；如果 x 等于 y，则返回 x (C99)                                                                                                                         |
| double nexttoward(double x, long double y);           | 与 nextafter() 类似，但该函数的第 2 个参数是 long double 类型；如果 x 等于 y，则返回转换为 double 类型的 y (C99)                                                                                             |
| double fdim(double x, double y);                      | 如果 x 大于 y，则返回 $x - y$ 的值；如果 x 小于或等于 y，则返回 0 (C99)                                                                                                                             |
| double fmax(double x, double y);                      | 返回参数的最大值，如果一个参数是 NaN、另一个参数是数值，则返回数值 (C99)                                                                                                                                     |
| double fmin(double x, double y);                      | 返回参数的最小值，如果一个参数是 NaN、另一个参数是数值，则返回数值 (C99)                                                                                                                                     |
| double fma(double x, double y, double z);             | 返回三元运算 $(x*y)+z$ 的大小，只在最后舍入一次 (C99)                                                                                                                                           |
| int isgreater(real-floating x, real-floating y);      | C99 宏，返回 $(x) > (y)$ 的值，如果有参数是 NaN，不会抛出“无效”浮点异常                                                                                                                               |
| int isgreaterequal(real-floating x, real-floating y); | C99 宏，返回 $(x) \geq (y)$ 的值，如果有参数是 NaN，不会抛出无效浮点异常                                                                                                                              |

<sup>1</sup> NaN 分为两类：quiet NaN 和 signaling NaN。两者的区别是：quiet NaN 的尾数部分最高位定义为 1，而 signaling NaN 最高位定义为 0。——译者注

续表

| 原型                                                      | 描述                                                               |
|---------------------------------------------------------|------------------------------------------------------------------|
| int isless(real-floating x,<br>real-floating y);        | C99 宏，返回 $(x) < (y)$ 的值，如果有参数是 NaN，不会抛出无效浮点异常                    |
| int islessequal(real-floating x,<br>real-floating y);   | C99 宏，返回 $(x) \leq (y)$ 的值，如果有参数是 NaN，不会抛出无效浮点异常                 |
| int islessgreater(real-floating x,<br>real-floating y); | C99 宏，返回 $(x) < (y) \mid\mid (x) > (y)$ 的值，如果有参数是 NaN，不会抛出无效浮点异常 |
| int isunordered(real-floating x,<br>real-floating y);   | 如果参数不按顺序排列（至少有一个参数是 NaN），函数返回 1；否则，返回 0                          |

### B.5.11 非本地跳转: setjmp.h

setjmp.h 头文件可以让你不遵循通常的函数调用、函数返回顺序。setjmp() 函数把当前执行环境的信息（例如，指向当前指令的指针）储存在 jmp\_buf 类型（定义在 setjmp.h 头文件中的数组类型）的变量中，然后 longjmp() 函数把执行转至这个环境中。这些函数主要是用来处理错误条件，并不是通常程序流控制的一部分。表 B.5.17 列出了这些函数。

表 B.5.17 setjmp.h 中的函数

| 原型                                     | 描述                                                                                           |
|----------------------------------------|----------------------------------------------------------------------------------------------|
| int setjmp(jmp_buf env);               | 把调用环境储存在数组 env 中，如果是直接调用，则返回 0；如果是通过 longjmp() 调用，则返回非 0                                     |
| void longjmp(jmp_buf env,<br>int val); | 恢复最近的 setjmp() 调用（设置 env 数组）储存的环境；完成后，程序继续像调用 setjmp() 那样执行该函数，返回 val（但是该函数不允许返回 0，会将其转换成 1） |

### B.5.12 信号处理: signal.h

信号（signal）是在程序执行期间可以报告的一种情况，可以用正整数表示。raise() 函数发送（或抛出）一个信号，signal() 函数设置特定信号的响应。

标准定义了一个整数类型：sig\_atomic\_t，专门用于在处理信号时指定原子对象。也就是说，更新原子类型是不可分割的过程。

标准提供的宏列于表 B.5.18 中，它们表示可能的信号，可用作 raise() 和 signal() 的参数。当然，实现也可以添加更多的值。

表 B.5.18 信 号 宏

| 宏       | 描述                      |
|---------|-------------------------|
| SIGABRT | 异常终止，例如 abort() 调用发出的信号 |
| SIGFPE  | 错误的算术运算                 |
| SIGILL  | 检测到无效功能（例如，非法指令）        |
| SIGINT  | 接收到交互注意信号（如，DOS 中断）     |
| SIGSEGV | 非法访问内存                  |
| SIGTERM | 向程序发送终止请求               |

`signal()` 函数的第 2 个参数接受一个指向 `void` 函数的指针，该函数有一个 `int` 类型的参数，也返回相同类型的指针。为响应一个信号而被调用的函数称为信号处理器（*signal handler*）。标准定义了 3 个满足下面原型的宏：

```
void (*funct) (int);
```

表 B.5.19 列出了这 3 种宏。

表 B.5.19 void (\*f) (int) 宏

| 宏                    | 描述                                                 |
|----------------------|----------------------------------------------------|
| <code>SIG_DFL</code> | 当该宏与一个信号值一起作为 <code>signal()</code> 的参数时，表示默认处理信号  |
| <code>SIG_ERR</code> | 如果 <code>signal()</code> 不能返回它的第 2 个参数，就用该宏作为它的返回值 |
| <code>SIG_IGN</code> | 当该宏与一个信号值一起作为 <code>signal()</code> 的参数时，表示忽略信号    |

如果产生了信号 `sig`，而且 `func` 指向一个函数（参见表 B.5.20 中 `signal()` 原型），那么大多数情况下先调用 `signal(sig, SIG_DFL)` 把信号重置为默认设置，然后调用 `(*func)(sig)`。可以执行返回语句或调用 `abort()`、`exit()` 或 `longjmp()` 来结束 `func` 指向的信号处理函数。

表 B.5.20 信号函数

| 宏                                                             | 描述                                                                                                         |
|---------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| <code>void (*signal(int sig, void (*func)(int)))(int);</code> | 如果产生信号 <code>sig</code> ，则执行 <code>func</code> 指向的函数；如果能执行则返回 <code>func</code> ，否则返回 <code>SIG_ERR</code> |
| <code>int raise(int sig);</code>                              | 向执行程序发送信号 <code>sig</code> ；如果成功发送则返回 0，否则返回非 0                                                            |

### B.5.13 对齐: stdalign.h (C11)

`stdalign.h` 头文件定义了 4 个宏，用于确定和指定数据对象的对齐属性。表 B.5.21 中列出了这些宏，其中前两个创建的别名与 C++ 的用法兼容。

表 B.5.21 void (\*f) (int) 宏

| 宏                                | 描述                             |
|----------------------------------|--------------------------------|
| <code>alignas</code>             | 展开为关键字 <code>_Alignas</code>   |
| <code>alignof</code>             | 展开为关键字 <code>_Alignof</code>   |
| <code>_alignas_is_defined</code> | 展开为整型常量 1，适用于 <code>#if</code> |
| <code>_alignof_is_defined</code> | 展开为整型常量 1，适用于 <code>#if</code> |

### B.5.14 可变参数: stdarg.h

`stdarg.h` 头文件提供一种方法定义参数数量可变的函数。这种函数的原型有一个形参列表，列表中至少有一个形参后面跟有省略号：

```
void f1(int n, ...); /* 有效 */
int f2(int n, float x, int k, ...); /* 有效 */
double f3(...); /* 无效 */
```

在下面的表中，`parmN` 是省略号前面的最后一个形参的标识符。在上面的例子中，第 1 种情况的 `parmN` 为 `n`，第 2 种情况的 `parmN` 为 `k`。

头文件中声明了 `va_list` 类型表示储存形参列表中省略号部分的形参数据对象。表 B.5.22 中列出了 3 个带可变参数列表的函数中用到的宏。在使用这些宏之前要声明一个 `va_list` 类型的对象。

表 B.5.22 可变参数列表宏

| 宏                                                     | 描述                                                                                                                                |
|-------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <code>void va_start(va_list ap, parmN);</code>        | 该宏在 <code>va_arg()</code> 和 <code>va_end()</code> 使用 <code>ap</code> 之前初始化 <code>ap</code> , <code>parmN</code> 是形参列表中最后一个形参名的标识符 |
| <code>void va_copy(va_list dest, va_list src);</code> | 该宏把 <code>dest</code> 初始化为 <code>src</code> 当前状态的备份 (C99)                                                                         |
| <code>type va_arg(va_list ap, type );</code>          | 该宏展开为一个表达式, 其值和类型都与 <code>ap</code> 表示的形参列表的下一项相同, <code>type</code> 是该项的类型。每次调用该宏都前进到 <code>ap</code> 中的下一项                      |
| <code>void va_end(va_list ap);</code>                 | 该宏关闭以上过程, 可能导致 <code>ap</code> 在再次调用 <code>va_start()</code> 之前不可用                                                                |
| <code>void va_copy(va_list dest, va_list src);</code> | 该宏把 <code>dest</code> 初始化为 <code>src</code> 当前状态的备份 (C99)                                                                         |

### B.5.15 原子支持: stdatomic.h (C11)

`stdatomic.h` 和 `threads.h` 头文件支持并发编程。并发编程的内容超过了本书讨论的范围, 简单地说, `stdatomic.h` 头文件提供了创建原子操作的宏。编程社区使用原子这个术语是为了强调不可分割的特性。一个操作 (如, 把一个结构赋给另一个结构) 从编程层面上看是原子操作, 但是从机器语言层面上看是由多个步骤组成。如果程序被分成多个线程, 那么其中的线程可能读或修改另一个线程正在使用的数据。例如, 可以想象给一个结构的多个成员赋值, 不同线程给不同成员赋值。有了 `stdatomic.h` 头文件, 就能创建这些可以看作是不可分割的操作, 这样就能保证线程之间互不干扰。

### B.5.16 布尔支持: stdbool.h (C99)

`stdbool.h` 头文件定义了 4 个宏, 如表 B.5.23 所列。

表 B.5.23 stdbool.h 宏

| 宏                                         | 描述                     |
|-------------------------------------------|------------------------|
| <code>bool</code>                         | 展开为 <code>_Bool</code> |
| <code>false</code>                        | 展开为整型常量 0              |
| <code>true</code>                         | 展开为整型常量 1              |
| <code>_bool_true_false_are_defined</code> | 展开为整型常量 1              |

### B.5.17 通用定义: stddef.h

该头文件定义了一些类型和宏, 如表 B.5.24 和表 B.5.25 所列。

表 B.5.24 stddef.h 类型

| 类型                     | 描述                                     |
|------------------------|----------------------------------------|
| <code>ptrdiff_t</code> | 有符号整数类型, 表示两个指针之差                      |
| <code>size_t</code>    | 无符号整数类型, 表示 <code>sizeof</code> 运算符的结果 |
| <code>wchar_t</code>   | 整数类型, 表示支持的本地化所指定的最大扩展字符集              |

表 B.5.25 stddef.h 宏

| 类型                                | 描述                                                                      |
|-----------------------------------|-------------------------------------------------------------------------|
| NULL                              | 实现定义的常量，表示空指针                                                           |
| offsetof(type, member-designator) | 展开为 size_t 类型的值，表示 type 类型结构的指定成员在该结构中的偏移量，以字节为单位。如果成员是一个位字段，该宏的行为是未定义的 |

**示例**

```
#include <stddef.h>

struct car
{
    char brand[30];
    char model[30];
    double hp;
    double price;
};

int main(void)
{
    size_t into = offsetof(struct car, hp); /* hp 成员的偏移量 */
    ...
}
```

**B.5.18 整数类型: stdint.h**

stdint.h 头文件中使用 `typedef` 工具创建整数类型名，指定整数的属性。stdint.h 头文件包含在 inttypes.h 中，后者提供输入/输出函数调用的宏。参考资料 VI 的“扩展的整数类型”中介绍了这些类型的用法。

**1. 精确宽度类型**

stdint.h 头文件中用一组 `typedef` 标识精确宽度的类型。表 B.5.26 列出了它们的类型名和大小。然而，注意，并不是所有的系统都支持其中的所有类型。

表 B.5.26 确切宽度类型

| typedef 名 | 属性       |
|-----------|----------|
| int8_t    | 8 位，有符号  |
| int16_t   | 16 位，有符号 |
| int32_t   | 32 位，有符号 |
| int64_t   | 64 位，有符号 |
| uint8_t   | 8 位，无符号  |
| uint16_t  | 16 位，无符号 |
| uint32_t  | 32 位，无符号 |
| uint64_t  | 64 位，无符号 |

**2. 最小宽度类型**

最小宽度类型保证其类型的大小至少是某数量位。表 B.5.27 列出了最小宽度类型，系统中一定会有这些类型。

表 B.5.27 最小宽度类型

| typedef 名      | 属性          |
|----------------|-------------|
| int_least8_t   | 至少 8 位，有符号  |
| int_least16_t  | 至少 16 位，有符号 |
| int_least32_t  | 至少 32 位，有符号 |
| int_least64_t  | 至少 64 位，有符号 |
| uint_least8_t  | 至少 8 位，无符号  |
| uint_least16_t | 至少 16 位，无符号 |
| uint_least32_t | 至少 32 位，无符号 |
| uint_least64_t | 至少 64 位，无符号 |

### 3. 最快最小宽度类型

在特定系统中，使用某些整数类型比其他整数类型更快。为此，`stdint.h` 也定义了最快最小宽度类型，如表 B.5.28 所列，系统中一定会有这些类型。

表 B.5.28 最快最小宽度类型

| typedef 名     | 属性         |
|---------------|------------|
| int_fast8_t   | 至少 8 位有符号  |
| int_fast16_t  | 至少 16 位有符号 |
| int_fast32_t  | 至少 32 位有符号 |
| int_fast64_t  | 至少 64 位有符号 |
| uint_fast8_t  | 至少 8 位无符号  |
| uint_fast16_t | 至少 16 位无符号 |
| uint_fast32_t | 至少 32 位无符号 |
| uint_fast64_t | 至少 64 位无符号 |

### 4. 最大宽度类型

`stdint.h` 头文件还定义了最大宽度类型。这种类型的变量可以储存系统中的任意整数值，还要考虑符号。表 B.5.29 列出了这些类型。

表 B.5.29 最大宽度类型

| typedef 名 | 描述         |
|-----------|------------|
| intmax_t  | 最大宽度的有符号类型 |
| uintmax_t | 最大宽度的无符号类型 |

### 5. 可储存指针值的整数类型

`stdint.h` 头文件中还包括表 B.5.30 中所列的两种整数类型，它们可以精确地储存指针值。也就是说，如果把一个 `void *` 类型的值赋给这种类型的变量，然后再把该类型的值赋回给指针，不会丢失任何信息。系统可能不支持这类型。

表 B.5.30 可储存指针值的整数类型

| typedef 名 | 描述           |
|-----------|--------------|
| intptr_t  | 可储存指针值的有符号类型 |
| uintptr_t | 可储存指针值的无符号类型 |

## 6. 已定义的常量

stdint.h 头文件定义了一些常量，用于表示该头文件中所定义类型的限定值。常量都根据类型命名，即用 \_MIN 或 \_MAX 代替类型名中的 \_t，然后把所有字母大写即得到表示该类型最小值或最大值的常量名。例如，int32\_t 类型的最小值是 INT32\_MIN、unit\_fast16\_t 的最大值是 UNIT\_FAST16\_MAX。表 B.5.31 总结了这些常量以及与之相关的 intptr\_t、unitptr\_t、intmax\_t 和 uintmax\_t 类型，其中的 N 表示位数。这些常量的值应等于或大于（除非指明了一定要等于）所列的值。

表 B.5.31 整型常量

| 常量标识符           | 最小值               |
|-----------------|-------------------|
| INTN_MIN        | 等于 $-(2^{N-1}-1)$ |
| INTN_MAX        | 等于 $2^{N-1}-1$    |
| UINTN_MAX       | 等于 $2^N-1$        |
| INT_LEASTN_MIN  | $-(2^{N-1}-1)$    |
| INT_LEASTN_MAX  | $2^{N-1}-1$       |
| UINT_LEASTN_MAX | $2^N-1$           |
| INT_FASTN_MIN   | $-(2^{N-1}-1)$    |
| INT_FASTN_MAX   | $2^{N-1}-1$       |
| UINT_FASTN_MAX  | $2^N-1$           |
| INTPTR_MIN      | $-(2^{15}-1)$     |
| INTPTR_MAX      | $2^{15}-1$        |
| UINTPTR_MAX     | $2^{16}-1$        |
| INTMAX_MIN      | $-(2^{15}-1)$     |
| INTMAX_MAX      | $2^{63}-1$        |
| UINTMAX_MAX     | $2^{64}-1$        |

该头文件还定义了一些别处定义的类型使用的常量，如表 B.5.32 所示。

表 B.5.32 其他整型常量

| 常量标识符          | 含义                  |
|----------------|---------------------|
| PTRDIFF_MIN    | ptrdiff_t 类型的最小值    |
| PTRDIFF_MAX    | ptrdiff_t 类型的最大值    |
| SIG_ATOMIC_MIN | sig_atomic_t 类型的最小值 |
| SIG_ATOMIC_MAX | sig_atomic_t 类型的最大值 |
| WCHAR_MIN      | wchar_t 类型的最小值      |
| WCHAR_MAX      | wchar_t 类型的最大值      |
| WINT_MIN       | wint_t 类型的最小值       |
| WINT_MAX       | wint_t 类型的最大值       |
| SIZE_MAX       | size_t 类型的最大值       |

## 7. 扩展的整型常量

stdin.h 头文件定义了一些宏用于指定各种扩展整数类型。从本质上讲，这种宏是底层类型（即在特定实现中表示扩展类型的基本类型）的强制转换。

把类型名后面的 `_t` 替换成 `_C`，然后大写所有的字母就构成了一个宏名。例如，使用表达式 `UNIT_LEAST64_C(1000)` 后，1000 就是 `unit_least64_t` 类型的常量。

## B.5.19 标准 I/O 库: stdio.h

ANSI C 标准库包含一些与流相关联的标准 I/O 函数和 stdio.h 头文件。表 B.5.33 列出了 ANSI 中这些函数的原型和简介（第 13 章详细介绍过其中的一些函数）。stdio.h 头文件定义了 FILE 类型、EOF 和 NULL 的值、标准 I/O 流（stdin、stdout 和 stderr）以及标准 I/O 库函数要用到的一些常量。

表 B.5.33 C 标准 I/O 函数

| 原型                                                                                         | 描述                                           |
|--------------------------------------------------------------------------------------------|----------------------------------------------|
| <code>void clearerr(FILE *);</code>                                                        | 清除文件结尾和错误指示符                                 |
| <code>int fclose(FILE *);</code>                                                           | 关闭指定的文件                                      |
| <code>int feof(FILE *);</code>                                                             | 测试文件结尾                                       |
| <code>int ferror(FILE *);</code>                                                           | 测试错误指示符                                      |
| <code>int fflush(FILE *);</code>                                                           | 刷新指定的文件                                      |
| <code>int fgetc(FILE *);</code>                                                            | 获得指定输入流的下一个字符                                |
| <code>int fgetpos(FILE *restrict, restrict);</code>                                        | 储存文件位置指示符的 <code>fpos_t *当前值</code>          |
| <code>char * fgets(char *restrict, restrict);</code>                                       | 从指定流中获取下一行（或 <code>int、FILE *</code> 指定的字符数） |
| <code>FILE * fopen(const char*restrict, const char*restrict);</code>                       | 打开指定的文件                                      |
| <code>int fprintf(FILE *restrict, const char *restrict, ...);</code>                       | 把格式化输出写入指定流                                  |
| <code>int fputc(int, FILE *);</code>                                                       | 把指定字符写入指定流                                   |
| <code>int fputs(const char* restrict, FILE * restrict);</code>                             | 把第 1 个参数指向的字符串写入指定流                          |
| <code>size_t fread(void *restrict, size_t, size_t, FILE * restrict);</code>                | 读取指定流中的二进制数据                                 |
| <code>FILE * freopen(const char * restrict, const char * restrict, FILE *restrict);</code> | 打开指定文件，并将其与指定流相关联                            |
| <code>int fscanf(FILE *restrict, const char * restrict, ...);</code>                       | 读取指定流中的格式化输入                                 |
| <code>int fsetpos(FILE *,const fpos_t *);</code>                                           | 设置文件位置指针指向指定的值                               |
| <code>int fseek(FILE *, long,int);</code>                                                  | 设置文件位置指针指向指定的值                               |
| <code>long ftell(FILE *);</code>                                                           | 获取当前文件位置                                     |
| <code>size_t fwrite(const void* restrict, size_t,size_t, FILE * restrict);</code>          | 把二进制数据写入指定流                                  |
| <code>int getc(FILE *);</code>                                                             | 读取指定输入的下一个字符                                 |
| <code>int getchar();</code>                                                                | 读取标准输入的下一个字符                                 |
| <code>char * gets(char *);</code>                                                          | 获取标准输入的下一行（C11 库已删除）                         |
| <code>void perror(const char*);</code>                                                     | 把系统错误信息写入标准错误中                               |
| <code>int printf(const char *restrict, ...);</code>                                        | 把格式化输出写入标准输出中                                |

续表

| 原型                                                                       | 描述                                                                |
|--------------------------------------------------------------------------|-------------------------------------------------------------------|
| int putc(int, FILE *);                                                   | 把指定字符写入指定输出中                                                      |
| int putchar(int);                                                        | 把指定字符写入指定输出中                                                      |
| int puts(const char *);                                                  | 把字符串写入标准输出中                                                       |
| int remove(const char *);                                                | 移除已命名文件                                                           |
| int rename(const char *, const char *);                                  | 重命名文件                                                             |
| void rewind(FILE *);                                                     | 设置文件位置指针指向文件开始处                                                   |
| int scanf(const char *restrict, ...);                                    | 读取标准输入中的格式化输入                                                     |
| void setbuf(FILE *restrict, char *restrict);                             | 设置缓冲区大小和位置                                                        |
| int setvbuf(FILE *restrict, char *restrict, int, size_t);                | 设置缓冲区大小、位置和模式                                                     |
| int snprintf(char *restrict, size_t n, const char * restrict, ...);      | 把格式化输出中的前 n 个字符写入指定字符串中                                           |
| int sprintf(char *restrict, const char *restrict, ...);                  | 把格式化输出写入指定字符串中                                                    |
| int sscanf(const char *restrict, const char *restrict, ...);             | 把格式化输入写入指定字符串中                                                    |
| FILE * tmpfile(void);                                                    | 创建一个临时文件                                                          |
| char * tmpnam(char *);                                                   | 为临时文件生成一个唯一的文件名                                                   |
| int ungetc(int, FILE *);                                                 | 把指定字符放回输入流中                                                       |
| int vfprintf(FILE *restrict, const char *restrict, va_list);             | 与 fprintf() 类似, 但该函数用一个 va_list 类型形参列表 (由 va_start 初始化) 代替变量参数列表  |
| int vprintf(const char *restrict, va_list);                              | 与 printf() 类似, 但该函数用一个 va_list 类型形参列表 (由 va_start 初始化) 代替变量参数列表   |
| int vsnprintf(char *restrict, size_t n, const char * restrict, va_list); | 与 snprintf() 类似, 但该函数用一个 va_list 类型形参列表 (由 va_start 初始化) 代替变量参数列表 |
| int vsprintf(char *restrict, const char *restrict, va_list);             | 与 sprintf() 类似, 但该函数用一个 va_list 类型形参列表 (由 va_start 初始化) 代替变量参数列表  |
| int vscanf(const char *restrict, va_list);                               | 与 scanf() 类似, 但该函数用一个 va_list 类型形参列表 (由 va_start 初始化) 代替变量参数列表    |
| int vsscanf(const char *restrict, va_list);                              | 与 sscanf() 类似, 但该函数用一个 va_list 类型形参列表 (由 va_start 初始化) 代替变量参数列表   |

## B.5.20 通用工具: stdlib.h

ANSI C 标准库在 stdlib.h 头文件中定义了一些实用函数。该头文件定义了一些类型, 如表 B.5.34 所示。

表 B.5.34 stdlib.h 中声明的类型

| 类型      | 描述                                                        |
|---------|-----------------------------------------------------------|
| size_t  | sizeof 运算符返回的整数类型                                         |
| wchar_t | 用于表示宽字符的整数类型                                              |
| div_t   | div() 返回的结构类型, 该类型中的 quot 和 rem 成员都是 int 类型               |
| ldiv_t  | ldiv() 返回的结构类型, 该类型中的 quot 和 rem 成员都是 long 类型             |
| lldiv_t | lldiv() 返回的结构类型, 该类型中的 quot 和 rem 成员都是 long long 类型 (C99) |

stdlib.h 头文件定义的常量列于表 B.5.35 中。

表 B.5.35 stdlib.h 中定义的常量

| 类型           | 描述                      |
|--------------|-------------------------|
| NULL         | 空指针 (相当于 0)             |
| EXIT_FAILURE | 可用作 exit() 的参数，表示执行程序失败 |
| EXIT_SUCCESS | 可用作 exit() 的参数，表示成功执行程序 |
| RAND_MAX     | rand() 返回的最大值 (一个整数)    |
| MB_CUR_MAX   | 当前本地化的扩展字符集中多字节字符的最大字节数 |

表 B.5.36 列出了 stdlib.h 中的函数原型。

表 B.5.36 通用工具

| 原型                                                                               | 描述                                                                                                                                                                           |
|----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| double atof(const char * nptr);                                                  | 返回把字符串 npt 开始部分的数字(和符号)字符转换为 double 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0                                                                                              |
| int atoi(const char* npt);                                                       | 返回把字符串 npt 开始部分的数字(和符号)字符转换为 int 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0                                                                                                 |
| int atol(const char* npt);                                                       | 返回把字符串 npt 开始部分的数字(和符号)字符转换为 long 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0                                                                                                |
| double strtod(const char* restrict npt, char ** restrict ept);                   | 返回把字符串 npt 开始部分的数字(和符号)字符转换为 double 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0；如果转换成功，则把数字后第 1 个字符的地址赋给 ept 指向的位置；如果转换失败，则把 npt 赋给 ept 指向的位置                                |
| float strtof(const char * restrict npt, char ** restrict ept);                   | 与 strtod() 类似，但是该函数把 npt 指向的字符串转换为 float 类型的值 (C99)                                                                                                                          |
| long double strtols(const char * restrict npt, char **restrict ept);             | 与 strtod() 类似，但是该函数把 npt 指向的字符串转换成 long double 类型的值 (C99)                                                                                                                    |
| long strtol(const char * restrict npt, char ** restrict ept, int base);          | 返回把字符串 npt 开始部分的数字(和符号)字符转换成 long 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0；如果转换成功，则把数字后第 1 个字符的地址赋给 ept 指向的位置；如果转换失败，则把 npt 赋给 ept 指向的位置；假定字符串中的数字以 base 指定的数为基数          |
| long long strtoll(const char * restrict npt, char** restrict ept, int base);     | 与 strtol() 类似，但是该函数把 npt 指向的字符串转换为 long long 类型的值 (C99)                                                                                                                      |
| unsigned long strtoul(const char * restrict npt, char** restrict ept, int base); | 返回把字符串 npt 开始部分的数字(和符号)字符转换为 unsigned long 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0；如果转换成功，则把数字后第 1 个字符的地址赋给 ept 指向的位置；如果转换失败，则把 npt 赋给 ept 指向的位置；假定字符串中的数字以 base 指定的数为基数 |

续表

| 原型                                                                                     | 描述                                                                                                                                                                                   |
|----------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| unsigned long long strtoull(const char* restrict npt, char ** restrict ept, int base); | 与 strtoul() 类似，但是该函数把 npt 指向的字符串转换为 unsigned long long 类型的值 (C99)                                                                                                                    |
| int rand(void);                                                                        | 返回 0~RAND_MAX 范围内的一个伪随机整数                                                                                                                                                            |
| void srand(unsigned int seed);                                                         | 把随机数生成器种子设置为 seed，如果在调用 rand() 之前调用 srand()，则种子为 1                                                                                                                                   |
| void *aligned_alloc(size_t align, size_t size);                                        | 为对齐对象 align 分配 size 字节的空间，应支持 align 对齐值，size 应该是 align 的倍数 (C11)                                                                                                                     |
| void *calloc(size_t nmemb, size_t size);                                               | 为内含 nmemb 个成员的数组分配空间，每个元素占 size 字节大；空间中的所有位都初始化为 0；如果操作成功，该函数返回数组的地址，否则返回 NULL                                                                                                       |
| void free(void*ptr);                                                                   | 释放 ptr 指向的空间，ptr 应该是之前调用 calloc()、malloc() 或 realloc() 返回的值，或者 ptr 也可以是空指针，出现这种情况时什么也不做。如果 ptr 是其他值，其行为是未定义的                                                                         |
| void *malloc(size_t size);                                                             | 分配 size 字节的未初始化内存块；如果成功分配，该函数返回数组的地址，否则返回 NULL                                                                                                                                       |
| void *realloc(void*ptr, size_t size);                                                  | 把 ptr 指向的内存块大小改为 size 字节，size 字节内的内存块内容不变。该函数返回块的位置，它可能被移动。如果不能重新分配空间，函数返回 NULL，原始块不变；如果 ptr 为 NULL，其行为与调用带 size 参数的 malloc() 相同；如果 size 是 0，且 ptr 不是 NULL，其行为与调用带 ptr 参数的 free() 相同 |
| void abort(void);                                                                      | 除非捕获信号 SIGABRT，且相应的信号处理器没有返回，否则该函数将导致程序异常结束。是否关闭 I/O 流和临时文件，因实现而异。该函数执行 raise(SIGABRT)                                                                                               |
| int atexit(void(*func) (void));                                                        | 注册 func 指向的函数，使其在程序正常结束时被调用。实现应支持注册至少 32 个函数，并根据它们注册顺序的逆序调用。如果注册成功，函数返回 0；否则返回非 0                                                                                                    |
| int at_quick_exit(void (*func) (void));                                                | 注册 func 指向的函数，如果调用 quick_exit() 则调用被注册的函数。实现应支持注册至少 32 个函数，并根据它们注册顺序的逆序调用。如果注册成功，函数返回 0；否则返回非 0 (C11)                                                                                |
| void exit(int status);                                                                 | 该函数将正常结束程序。首先调用由 atexit() 注册的函数，然后刷新所有打开的输出流、关闭所有的 I/O 流、关闭 tmpfile() 创建的所有文件，并把控制权返回主机环境中；如果 status 是 0 或 EXIT_SUCCESS，则返回一个实现定义的值，表明未成功结束程序                                        |
| void _Exit(int status);                                                                | 与 exit() 类似，但是该函数不调用 atexit() 注册的函数和 signal() 注册的信号处理器，其处理打开流的方式依实现而异                                                                                                                |
| char *getenv(const char * name);                                                       | 返回一个指向字符串的指针，该字符串表示 name 指向的环境变量的值。如果无法匹配指定的 name，则返回 NULL                                                                                                                           |
| _Noreturn void quick_exit(int status);                                                 | 该函数将正常结束程序。不调用 atexit() 注册的函数和 signal() 注册的信号处理器。根据 at_quick_exit() 注册函数的顺序，逆序调用这些函数。如果程序多次调用 quick_exit() 或者同时调用 quick_exit() 和 exit()，其行为是未定义的。通过调用 _Exit(status) 将控制权返回主机环境 (C11) |

续表

| 原型                                                                                                                   | 描述                                                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| int system(const char *str);                                                                                         | 把 str 指向的字符串传递给命令处理器(如 DOS 或 UNIX)执行的主机环境。如果 str 是 NULL 指针, 且命令处理器可用, 则该函数返回非 0, 否则返回; 如果 str 不是 NULL, 返回值依实现而异                                                                                                         |
| void *bsearch(const void *key, const void *base, size_t nmem, size_t size, int (*comp)(const void *, const void *)); | 查找 base 指向的一个数组(有 nmem 个元素, 每个元素的大小为 size)中是否有元素匹配 key 指向的对象。通过 comp 指向的函数比较各项, 如果 key 指向的对象小于数组元素, 那么比较函数将返回小于 0 的值; 如果两者相等, 则返回 0; 如果 key 指向的对象大于数组元素, 则返回大于 0 的值。该函数返回指向匹配元素的指针或 NULL(如果无匹配元素)。如果有多个元素匹配, 未定义返回哪一个元素 |
| void qsort(void *base, size_t nmem, size_t size, int (*comp)(const void *, const void *));                           | 根据 comp 指向的函数所提供的顺序排列 base 指向的数组。该数组有 nmem 个元素, 每个元素的大小是 size。如果第 1 个参数指向的对象小于数组元素, 那么比较函数将返回小于 0 的值; 如果两者相等, 则返回 0; 如果第 1 个参数指向的对象大于数组元素, 则返回大于 0 的值                                                                   |
| int abs(int n);                                                                                                      | 返回 n 的绝对值。如果 n 是负数但没有与之对应的正数, 那么返回值是未定义的(当 n 是以二进制补码表示的 INT_MIN 时, 会出现这种情况)                                                                                                                                             |
| div_t div(int numer, int denom);                                                                                     | 计算 number 除以 denom 的商和余, 把商和余数分别储存在 div_t 结构的 quot 成员和 rem 成员中。对于无法整除的除法, 商要趋零截断(即直接截去小数部分)                                                                                                                             |
| long labs(int n);                                                                                                    | 返回 n 的绝对值, 如果 n 是负数但没有与之对应的正数, 那么返回值是未定义的(当 n 是以二进制补码表示的 LONG_MIN 时, 会出现这种情况)                                                                                                                                           |
| ldiv_t ldiv(long numer, long denom);                                                                                 | 计算 number 除以 denom 的商和余, 把商和余数分别储存在 ldiv_t 结构的 quot 成员和 rem 成员中。对于无法整除的除法, 商要趋零截断(即直接截去小数部分)                                                                                                                            |
| long long llabs(int n);                                                                                              | 返回 n 的绝对值, 如果 n 是负数但没有与之对应的正数, 那么返回值是未定义的(当 n 是以二进制补码表示的 LONG_LONG_MIN 时, 会出现这种情况)                                                                                                                                      |
| lldiv_t lldiv(long numer, long denom);                                                                               | 计算 number 除以 denom 的商和余, 把商和余数分别储存在 lldiv_t 结构的 quot 成员和 rem 成员中。对于无法整除的除法, 商要趋零截断(即直接截去小数部分)(C99)                                                                                                                      |
| int mblen(const char *s, size_t n);                                                                                  | 返回组成 s 指向的多字节字符的字节数(最大为 n)。如果 s 指向空字符, 该函数则返回 0; 如果 s 未指向多字节字符, 则返回 -1; 如果 s 是 NULL, 且多字节根据状态进行编码, 该函数则返回非 0, 否则返回 0                                                                                                    |
| int mbtowc(wchar_t *pw, const char *s, size_t n);                                                                    | 如果 s 不是 NULL, 该函数确定了组成 s 指向的多字节字符的字节数(最大为 n), 并确定字符的 wchar_t 类型编码。如果 pw 不是 NULL, 则把类型编码赋给 pw 指向的位置。返回值与 mbolen(s, n) 相同                                                                                                 |
| int wctomb(char *s, wchar_t wc);                                                                                     | 把 wc 中的字符代码转换成相应的多字节字符表示, 并将其储存在 s 指向的数组中, 除非 s 是 NULL。如果 s 不是 NULL, 且如果 wc 无法转换成相应有效多字节字符, 该函数返回 -1; 如果 wc 有效, 该函数返回组成多字节的字节数; 如果 s 是 NULL, 且如果多字节字符根据状态进行编码, 该函数则返回非 0, 否则返回 0                                        |

续表

| 原型                                                                                        | 描述                                                                                                                                       |
|-------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <code>size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s, size_t n);</code>   | 把 s 指向的多字节字符数组转换成储存在 pwcs 开始位置的宽字符编码数组中，转换 pwcs 数组中的 n 个字符或转换到 s 数组的空字节停止。如果遇到无效的多字节字符，该函数返回 (size_t) (-1)，否则返回已填充的数组元素个数（如果有空字符，不包含空字符） |
| <code>size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);</code> | 把储存在 pwcs 指向数组中的宽字符编码序列转换成一个多字节字符序列，并把它拷贝到 s 指向的位置上，储存 n 个字节或遇到空字符时停止转换。如果遇到无效的宽字符编码，该函数返回 (size_t) (-1)，否则返回已填充数组的字节数（如果有空字符，不包含空字符）    |

### B.5.21 \_Noreturn: stdnoreturn.h

`stdnoreturn.h` 定义了 `noreturn` 宏，该宏展开为 `_Noreturn`。

### B.5.22 处理字符串: string.h

`string.h` 库定义了 `size_t` 类型和空指针要使用的 `NULL` 宏。`string.h` 头文件提供了一些分析和操控字符串的函数，其中一些函数以更通用的方式处理内存。表 B.5.37 列出了这些函数。

表 B.5.37 字符串函数

| 原型                                                                                 | 描述                                                                                                                                                                                                         |
|------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void *memchr(const void *s, int c, size_t n);</code>                         | 在 s 指向对象的前 n 个字符中查找是否有 c。如果找到，则返回首次出现 c 处的指针，如果未找到则返回 <code>NULL</code>                                                                                                                                    |
| <code>int memcmp(const void*s1, const void *s2, size_t n);</code>                  | 比较 s1 指向对象中的前 n 个字符和 s2 指向对象的前 n 个字符，每个值都解释为 <code>unsigned char</code> 类型。如果 n 个字符都匹配，则两个对象完全相同；否则，比较两个对象中首次不匹配的字符对。如果两个对象相同，函数返回 0；如果在数值上第 1 个对象小于第 2 个对象，函数返回小于 0 的值；如果在数值上第 1 个对象大于第 2 个对象，函数返回大于 0 的值 |
| <code>void *memcpy(void *restrict s1, const void * restrict s2, size_t n);</code>  | 把 s2 所指向位置上的 n 字节拷贝到 s1 指向的位置上，函数返回 s1 的值。如果两个位置出现重叠，其行为是未定义的                                                                                                                                              |
| <code>void *memmove(void*s1, const void *s2, size_t n);</code>                     | 把 s2 所指向位置上的 n 字节拷贝到 s1 指向的位置上，其行为与拷贝类似，返回 s1 的值。但是，如果出现局部重叠情况，该函数会先把重叠的内容拷贝至临时位置                                                                                                                          |
| <code>void *memset(void *s, int v, size_t n);</code>                               | 把 v 的值（转换为 <code>unsigned char</code> ）拷贝至 s 指向的前 n 字节中，函数返回 s                                                                                                                                             |
| <code>char *strcat(char *restrict s1, const char * restrict s2);</code>            | 把 s2 指向的字符串拷贝到 s1 指向字符串后面，s2 字符串的第一个字符覆盖 s1 字符串的空字符。该函数返回 s1                                                                                                                                               |
| <code>char *strncat(char *restrict s1, const char * restrict s2, size_t n);</code> | 把 s2 指向字符串的 n 个字符拷贝到 s1 指向的字符串后面（或拷贝到 s2 的空字符为止）。s2 字符串的第一个字符覆盖 s1 字符串的空字符。函数返回 s1                                                                                                                         |
| <code>char *strcpy(char *restrict s1, const char * restrict s2);</code>            | 把 s2 指向的字符串拷贝到 s1 指向的位置。函数返回 s1                                                                                                                                                                            |

续表

| 原型                                                                       | 描述                                                                                                                                                                                                                      |
|--------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| char *strncpy(char *restrict s1,<br>const char * restrict s2,size_t n);  | 把 s2 指向字符串的 n 个字符拷贝到 s1 指向的位置 (或拷贝到 s2 的空字符为止)。如果在拷贝 n 个字符之前遇到空字符，则在拷贝字符后面添加若干个空字符，使其长度为 n；如果拷贝 n 个字符没有遇到空字符，则不添加空字符。函数返回 s1                                                                                            |
| int strcmp(const char*s1, const char<br>*s2);                            | 比较 s1 和 s2 指向的两个字符串。如果完全匹配，则两字符串相同，否则比较首次出现不匹配的字符对。通过字符编码值比较字符。如果两个字符串相同，函数返回 0；如果第 1 个字符串小于第 2 个字符串，函数返回小于 0 的值；如果第 1 个字符串大于第 2 个字符串，函数返回大于 0 的值                                                                       |
| int strcoll(const char *s1, const char<br>*s2);                          | 与 strcmp() 类似，但是该函数使用当前本地化的 LC_COLLATE 类别(由 setlocale() 函数设置)所指定的排序方式进行比较                                                                                                                                               |
| int strncmp(const char *s1, const char<br>*s2, size_t n);                | 比较 s1 和 s2 指向数组中的前 n 个字符，或比较到第 1 个空字符位置。如果所有的字符对都匹配，则两个数组相同否则比较两个数组中首次不匹配的字符对。通过字符编码值比较字符。如果两个数组相同，函数返回 0；如果第 1 个数组小于第 2 个数组，函数返回小于 0 的值；如果第 1 个数组大于第 2 个数组，函数返回大于 0 的值                                                 |
| size_t strxfrm(char* restrict s1,<br>const char * restrict s2,size_t n); | 转换 s2 中的字符串，并把转换后的前 n 个字符(包括空字符)拷贝到 s1 指向的数组中。用 strcmp() 比较转换后的两个字符串的结果和用 strcoll() 比较两个未转换字符串的结果相同。函数返回转换后的字符串长度(不包括末尾的空字符)                                                                                            |
| char *strchr(const char *s, int c);                                      | 查找 s 指向的字符串中首次出现 c 的位置。空字符是字符串的一部分。函数返回一个指针，指向首次出现 c 的位置。如果没有找到指定的 c 则返回 NULL                                                                                                                                           |
| size_t strcspn(const char *s1, const<br>char*s2);                        | 返回 s1 中未出现 s2 中任何字符的最大起始段长度                                                                                                                                                                                             |
| char *strupbrk(const char *s1, const<br>char*s2);                        | 返回一个指针，指向 s1 中与 s2 任意字符匹配的第一个字符的位置。如果未发现任何匹配的字符，函数返回 NULL                                                                                                                                                               |
| char *strrchr(const char *s, int c);                                     | 在 s 指向的字符串中查找末次出现 c 的位置(即从 s2 右侧开始查找字符 c 首次出现的位置)。空字符是字符串的一部分。如果找到，函数返回指向该位置的指针；如果未找到，则返回 NULL                                                                                                                          |
| size_t strspn(const char *s1, const<br>char*s2);                         | 返回 s1 中包含 s2 所有字符的最大起始段长度                                                                                                                                                                                               |
| char *strrstr(const char *s1, const<br>char*s2);                         | 返回一个指针，指向 s1 中首次出现 s2 中字符序列(不包括结束的空字符)的位置。如果未找到，函数返回 NULL                                                                                                                                                               |
| char *strtok(char *restrict s1, const<br>char * restrict s2);            | 该函数把 s1 字符串分解为单独的记号。s2 字符串包含了作为记号分隔符的字符。按顺序调用该函数。第 1 次调用时，s1 应指向待分解的字符串。函数定位到非分隔符后的第 1 个记号分隔符，并用空字符替换它。函数返回一个指针，指向储存第 1 个记号的字符串。如果未找到记号，函数返回 NULL。在此次调用 strtok() 查找字符串中的更多记号。每次调用都返回指向下一个记号的指针，如果未找到则返回 NULL(请参看表后面的示例) |
| char * strerror(int errnum);                                             | 返回一个指针，指向与储存在 errnum 中的错误号相对应的错误信息字符串(依实现而异)                                                                                                                                                                            |
| int strlen(const char* s);                                               | 返回字符串 s 中的字符数(末尾的空字除外)                                                                                                                                                                                                  |

`strtok()` 函数的用法有点不寻常，下面演示一个简短的示例。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char data[] = " C is\t too#much\nfun!";
    const char tokseps[] = "\t\n#"; /* 分隔符 */
    char * pt;

    puts(data);
    pt = strtok(data, tokseps);      /* 首次调用 */
    while (pt)                      /* 如果 pt 是 NULL，则退出 */
    {
        puts(pt);                  /* 显示记号 */
        pt = strtok(NULL, tokseps); /* 下一个记号 */
    }
    return 0;
}
```

下面是该示例的输出：

```
C is too#much
fun!
C
is
too
much
fun!
```

### B.5.23 通用类型数学：`tgmath.h` (C99)

`math.h` 和 `complex.h` 库中有许多类型不同但功能相似的函数。例如，下面 6 个都是计算正弦的函数：

```
double sin(double);
float sinf(float);
long double sinl(long double);
double complex csin(double complex);
float csinf(float complex);
long double csinl(long double complex);
```

`tgmath.h` 头文件定义了展开为通用调用的宏，即根据指定的参数类型调用合适的函数。下面的代码演示了使用 `sin()` 宏时，展开为正弦函数的不同形式：

```
#include <tgmath.h>
...
double dx, dy;
float fx, fy;
long double complex clx, cly;
dy = sin(dx);           // 展开为 dy = sin(dx) (函数)
fy = sin(fx);           // 展开为 fy = sinf(fx)
cly = sin(clx);         // 展开为 cly = csinl(cly)
```

`tgmath.h` 头文件为 3 类函数定义了通用宏。第 1 类由 `math.h` 和 `complex.h` 中定义的 6 个函数的变式组成，用 `l` 和 `f` 后缀和 `c` 前缀，如前面的 `sin()` 函数所示。在这种情况下，通用宏名与该函数 `double` 类型版本的函数名相同。

第 2 类由 `math.h` 头文件中定义的 3 个函数变式组成，使用 `l` 和 `f` 后缀，没有对应的复数函数（如，`erf()`）。

在这种情况下，宏名与没有后缀的函数名相同，如 `erf()`。使用带复数参数的这种宏的效果是未定义的。

第 3 类由 `complex.h` 头文件中定义的 3 个函数变式组成，使用 `l` 和 `f` 后缀，没有对应的实数函数，如 `cimag()`。使用带实数参数的这种宏的效果是未定义的。

表 B.5.38 列出了一些通用宏函数。

表 B.5.38 通用数学函数

|                     |                      |                        |                        |                         |                        |
|---------------------|----------------------|------------------------|------------------------|-------------------------|------------------------|
| <code>acos</code>   | <code>asin</code>    | <code>atanb</code>     | <code>acosh</code>     | <code>asinh</code>      | <code>atanh</code>     |
| <code>cos</code>    | <code>sin</code>     | <code>tan</code>       | <code>cosh</code>      | <code>sinh</code>       | <code>tanh</code>      |
| <code>exp</code>    | <code>log</code>     | <code>pow</code>       | <code>sqrt</code>      | <code>fabs</code>       | <code>atan2</code>     |
| <code>cbrt</code>   | <code>ceil</code>    | <code>copysign</code>  | <code>erf</code>       | <code>erfc</code>       | <code>exp2</code>      |
| <code>expm1</code>  | <code>fdim</code>    | <code>floor</code>     | <code>fma</code>       | <code>fmax</code>       | <code>fmin</code>      |
| <code>fmod</code>   | <code>frexp</code>   | <code>hypot</code>     | <code>ilogb</code>     | <code>ldexp</code>      | <code>lgamma</code>    |
| <code>llrint</code> | <code>llround</code> | <code>log10</code>     | <code>log1p</code>     | <code>log2</code>       | <code>logb</code>      |
| <code>lrint</code>  | <code>lround</code>  | <code>nearbyint</code> | <code>nextafter</code> | <code>nexttoward</code> | <code>remainder</code> |
| <code>remquo</code> | <code>rint</code>    | <code>round</code>     | <code>scalbn</code>    | <code>scalbln</code>    | <code>tgamma</code>    |
| <code>trunc</code>  | <code>carg</code>    | <code>cimag</code>     | <code>conj</code>      | <code>cproj</code>      | <code>creal</code>     |

在 C11 以前，编写实现必须依赖扩展标准才能实现通用宏。但是使用 C11 新增的 `_Generic` 表达式可以直接实现。

## B.5.24 线程: threads.h (C11)

`threads.h` 和 `stdatomic.h` 头文件支持并发编程。这方面的内容超出了本书讨论的范围，简而言之，该头文件支持程序执行多线程，原则上可以把多个线程分配给多个处理器处理。

## B.5.25 日期和时间: time.h

`time.h` 定义了 3 个宏。第 1 个宏是表示空指针的 `NULL`，许多其他头文件中也定义了这个宏。第 2 个宏是 `CLOCKS_PER_SEC`，该宏除以 `clock()` 的返回值得以秒为单位的时间值。第 3 个宏 (C11) 是 `TIME_UTC`，这是一个正整型常量，用于指定协调世界时<sup>1</sup> (即 UTC)。该宏是 `timespec_get()` 函数的一个可选参数。

UTC 是目前主要世界时间标准，作为互联网和万维网的普通标准，广泛应用于航空、天气预报、同步计算机时钟等各领域。

`time.h` 头文件中定义的类型列在表 B.5.39 中。

表 B.5.39 time.h 中定义的类型

| 类型                           | 描述                             |
|------------------------------|--------------------------------|
| <code>size_t</code>          | <code>sizeof</code> 运算符返回的整数类型 |
| <code>clock_t</code>         | 适用于表示时间的算术类型                   |
| <code>time_t</code>          | 适用于表示时间的算术类型                   |
| <code>struct timespec</code> | 以秒和纳秒为单位储存指定时间间隔的结构 (C11)      |
| <code>struct tm</code>       | 储存日历时间的各部分                     |

<sup>1</sup> 也称为世界标准时间，简称 UTC，从英文“Coordinated Universal Time”/法文“Temps Universel Coordonné”而来。中国内地的时间与 UTC 的时差为 +8，也就是 UTC+8。——译者注

`timespec` 结构中至少有两个成员，如表 B.5.40 所列。

表 B.5.40 `timespec` 结构中的成员

| 成员                         | 描述                  |
|----------------------------|---------------------|
| <code>time_t tv_sec</code> | 秒 ( $>= 0$ )        |
| <code>long tv_nsec</code>  | 纳秒 ([0, 999999999]) |

日历类型的各组成部分被称为分解时间 (*broken-down time*)。表 B.5.41 列出了 `struct tm` 结构中所需的成员。

表 B.5.41 `struct tm` 结构中的成员

| 成员                        | 描述                                          |
|---------------------------|---------------------------------------------|
| <code>int tm_sec</code>   | 分后的秒 (0-61)                                 |
| <code>int tm_min</code>   | 小时后的分 (0-59)                                |
| <code>int tm_hour</code>  | 小时 (0-23)                                   |
| <code>int tm_mday</code>  | 一个月的天数 (0-31)                               |
| <code>int tm_mon</code>   | 一月后的月数 (0-11)                               |
| <code>int tm_year</code>  | 1900 年后的年数                                  |
| <code>int tm_wday</code>  | 星期日开始的天数 (0-6)                              |
| <code>int tm_yday</code>  | 从 1 月 1 日开始的天数 (0-365)                      |
| <code>int tm_isdst</code> | 夏令时标志 (大于 0 说明夏令时有效，等于 0 说明无效，小于 0 说明信息不可用) |

日历时间 (*calendar time*) 表示当前的日期和时间，例如，可以是从 1900 年的第 1 秒开始经过的秒数。本地时间 (*local time*) 指的是本地时区的日历时间。表 B.5.42 列出了一些时间函数。

表 B.5.42 时间 函数

| 成员                                                           | 描述                                                                                                                                                                                                                                                                   |
|--------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>clock_t clock(void);</code>                            | 该函数返回实现从开始执行程序到调用该函数时，处理器经过的最接近的时间。该函数的返回值除以 <code>CLOCK_PER_SEC</code> 得到以秒为单位的时间。如果时间不可用或无法表示，函数返回 ( <code>clock_t</code> ) (-1)                                                                                                                                   |
| <code>double difftime(time_t t1, time_t t0);</code>          | 返回两个日历时间 ( <code>t1 - t0</code> ) 的差值。该函数返回计算结果，单位是秒                                                                                                                                                                                                                 |
| <code>time_t mktime(struct tm *tmptr);</code>                | 把 <code>tmptr</code> 指向的结构中的分解时间转换为日历时间。其编码与 <code>time()</code> 函数相同，但是结构改变了，以便对结构中超出范围的值进行调整（例如，2 分 100 秒会调整为 4 分 40 秒），而且把 <code>tm_wday</code> 和 <code>tm_yday</code> 设置为其他成员指定的值。如果无法表示日历时间，该函数返回 ( <code>time_t</code> ) (-1)；否则以 <code>time_t</code> 格式返回日历时间 |
| <code>time_t time(time_t *ptm)</code>                        | 返回当前日历时间，并将其储存在 <code>ptm</code> 指向的位置，假设 <code>ptm</code> 不是空指针。如果日期时间不可用，该函数返回 ( <code>time_t</code> ) (-1)                                                                                                                                                        |
| <code>int timespec_get(struct timespec *ts, int base)</code> | 根据指定的时基，把 <code>ts</code> 指向的结构设置为当前日历时间。如果成功，返回 <code>base</code> (非 0 值)，否则返回 0 (C11)                                                                                                                                                                              |
| <code>char *asctime(const struct tm *tmptr);</code>          | 把 <code>tmptr</code> 指向的结构中的分解时间转换成 <code>Thu Feb 26 13:14:33 1998\n\0</code> 格式的字符串，并返回指向该字符串的指针                                                                                                                                                                    |

续表

| 成员                                                                                                      | 描述                                                                                                                                             |
|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| char *ctime(const time_t *ptm);                                                                         | 把 ptm 指向的结构中的分解时间转换成 Wed Aug 11 10:48:24 1999\n\0 格式的字符串，并返回指向该字符串的指针                                                                          |
| struct tm *gmtime(const time_t *ptm);                                                                   | 把 ptm 指向的日历时间转换成协调世界时 (UTC) 表示的分解时间，返回一个指向结构的指针，该结构中储时间信息。如果 UTC 不可用，则返回 NULL                                                                  |
| struct tm *localtime(const time_t *ptm);                                                                | 把 ptm 指向的日历时间转换成本地时间表示的分解时间，储存 tm 结构并返回指向该结构的指针                                                                                                |
| size_t strftime(char *restrict s, size_t max const char *restrict fmt, const struct tm *restrict tmpt); | 把字符串 fmt 拷贝到字符串 s 中，用 tmp 指向的分解时间结构中的合适数据替换 fmt 中的转换说明（见表 B.5.43）。最多在 s 中放入 max 个字符。该函数返回放入 s 中的字符数（不包括空格）；如果字符串中的字符数大于 max，函数返回 0，且 s 中的内容不确定 |

表 B.5.43 列出了 strftime() 函数中使用的转换说明。其中许多替换的值（如，月份名）都取决于当前的本地化设置。

表 B.5.43 strftime() 函数中使用的转换说明

| 转换说明 | 被替换为                              |
|------|-----------------------------------|
| %a   | 本地化的星期名称缩写                        |
| %A   | 本地化的星期名称全名                        |
| %b   | 本地化的月份名称缩写                        |
| %B   | 本地化的月份名称全名                        |
| %c   | 本地化指定的日期和时间                       |
| %C   | 年份的后两位数字（年份除以 100，取小数部分的数）(00-99) |
| %d   | 十进制数表示的月份中的某天（01-31）              |
| %D   | 月/日/年，等价于“%m/%d/%y”               |
| %e   | 十进制数表示的月份中的某天，在仅一位的数字前有一个空格（1-31） |
| %F   | 年-月-日，等价于“%Y-%m-%d”               |
| %g   | 基于周的年份的最后两位数字（00-99）              |
| %G   | 十进制数表示的基于周的年份                     |
| %h   | 等价于“%b”                           |
| %H   | 十进制数（00-23）表示的小时（24 小时制）          |
| %I   | 十进制数（01-12）表示的小时（12 小时制）          |
| %j   | 十进制数表示的一年中的某天（001-366）            |
| %m   | 十进制数表示的月份（01-12）                  |
| %n   | 换行符                               |
| %M   | 十进制数表示的分钟（00-59）                  |
| %p   | 等价于本地 12 小时制中的 am/pm              |
| %r   | 本地的 12 小时制                        |

续表

| 转换说明 | 被替换为                                                                      |
|------|---------------------------------------------------------------------------|
| %R   | 小时:分钟, 等价于“%H:%M”                                                         |
| %S   | 十进制数表示的秒(00~61)                                                           |
| %t   | 水平制表符                                                                     |
| %T   | 小时:分钟:秒, 等价于“%H:%M:%S”                                                    |
| %u   | ISO 8601 的星期数(1~7), 星期一为1                                                 |
| %U   | 一年中的周数(00~53), 把星期天作为一周的第一天                                               |
| %V   | ISO 8601 的一年周数(00~53), 把星期天作为一周的第一天                                       |
| %w   | 十进制表示的星期数(0~6), 从星期天开始                                                    |
| %W   | 一年的周数(00~53), 把星期一作为一周的第一天                                                |
| %x   | 本地化日期表示                                                                   |
| %X   | 本地化时间表示                                                                   |
| %y   | 不带世纪的十进制年份(00~99)                                                         |
| %Y   | 带世纪的十进制年份                                                                 |
| %z   | 按照 ISO 8601 格式的相对 UTC 偏移(“-800”表示格林威治时间后的 8 小时, 即是向西 8 小时), 如果无可用信息则无替换字符 |
| %Z   | 时区名, 如果无可用信息则无替换字符                                                        |
| %%   | % (即百分号)                                                                  |

### B.5.26 统一码工具: uchar.h (C11)

C99 的 wchar.h 头文件提供两种途径支持大型字符集。C11 专门针对统一码(Unicode)新增了适用于 UTF-16 和 UTF-32 编码的类型(见表 B.5.44)。

表 B.5.44 uchar.h 中声明的类型

| 类型        | 描述                                                 |
|-----------|----------------------------------------------------|
| char16_t  | 使用 16 位字符的无符号整数类型(与 stdint.h 中的 unit_least16_t 相同) |
| char32_t  | 使用 32 位字符的无符号整数类型(与 stdint.h 中的 unit_least32_t 相同) |
| size_t    | sizeof 运算符(stddef.h)返回的整数类型                        |
| mbstate_t | 非数组类型, 可储存多字节字符序列和宽字符相互转换的转换状态信息                   |

该头文件中还声明了一些多字节字符串与 char16\_t、char32\_t 格式相互转换的函数(见表 B.5.45)。

表 B.5.45 宽字符与多字节转换函数

| 类型                                                                                                    | 描述                                                                |
|-------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
| size_t mbrtol6(char16_t* restrict pwc, const char * restrict s, size_t n, mbstate_t* restrict ps);    | 与 mbrtowc() 函数相同(wchar.h), 但该函数是把字符转换为 char_16 类型, 而不是 wchar_t 类型 |
| size_t mbrto32( char32_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps); | 与 mbrtol6() 函数相同, 但该函数是把字符转换为 char32_t 类型                         |

续表

| 类型                                                                          | 描述                                                               |
|-----------------------------------------------------------------------------|------------------------------------------------------------------|
| size_t c16rtomb(char * restrict s,<br>wchar_t wc, mbstate_t * restrict ps); | 与 wcrtobm() 函数相同 (wchar.h)，但该函数转换的是 char16_t 类型字符，而不是 wchar_t 类型 |
| size_t c32rtomb(char * restrict s,<br>wchar_t wc, mbstate_t * restrict ps); | 与 wcrtobm() 函数相同 (wchar.h)，但该函数转换的是 char32_t 类型字符，而不是 wchar_t 类型 |

### B.5.27 扩展的多字节字符和宽字符工具: wchar.h (C99)

每种实现都有一个基本字符集，要求 C 的 char 类型足够宽，以便能处理这个字符集。实现还要支持扩展的字符集，这些字符集中的字符可能需要多字节来表示。可以把多字节字符与单字节字符一起储存在普通的 char 类型数组，用特定的字节值指定多字节字符本身及其大小。如何解释多字节字符取决于移位状态 (*shift state*)。在最初的移位状态中，单字节字符保留其通常的解释。特殊的多字节字符可以改变移位状态。除非显式改变特定的移位状态，否则移位状态一直保持有效。

wchar\_t 类型提供另一种表示扩展字符的方法，该类型足够宽，可以表示扩展字符集中任何成员的编码。用这种宽字符类型来表示字符时，可以把单字符储存在 wchar\_t 类型的变量中，把宽字符的字符串储存在 wchar\_t 类型的数组中。字符的宽字符表示和多字节字符表示不必相同，因为后者可能使用前者并不使用的移位状态。

wchar.h 头文件提供了一些工具用于处理扩展字符的两种表示法。该头文件中定义的类型列在表 B.5.46 中（其中有些类型也定义在其他的头文件中）。

表 B.5.46 wchar.h 中定义的类型

| 类型        | 描述                                |
|-----------|-----------------------------------|
| wchar_t   | 整数类型，可表示本地化支持的最大扩展字符集             |
| wint_t    | 整数类型，可储存扩展字符集的任意值和至少一个不是扩展字符集成员的值 |
| size_t    | sizeof 运算符返回的整数类型                 |
| mbstate_t | 非数组类型，可储存多字节字符序列和宽字符之间转换所需的转换状态信息 |
| struct tm | 结构类型，用于储存日历时间的组成部分                |

wchar.h 头文件中还定义了一些宏，如表 B.5.47 所列。

表 B.5.47 wchar.h 中定义的宏

| 宏         | 描述                                                          |
|-----------|-------------------------------------------------------------|
| NULL      | 空指针                                                         |
| WCHAR_MAX | wchar_t 类型可储存的最大值                                           |
| WCHAR_MIN | wchar_t 类型可储存的最小值                                           |
| WEOF      | wint_t 类型的常量表达式，不与扩展字符集的任何成员对；相当于 EOF 的宽字符表示，用于指定宽字符输入的文件结尾 |

该库提供的输入/输出函数类似于 stdio.h 中的标准输入/输出函数。在标准 I/O 函数返回 EOF 的情况下，对应的宽字符函数返回 WEOF。表 B.5.48 中列出了这些函数。

表 B.5.48 宽字符 I/O 函数

## 函数原型

```

int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);
int fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);
int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, ...);
int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);
int vfwprintf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
int vfwscanf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list arg);
int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);
int vwprintf(const wchar_t * restrict format, va_list arg);
int vwscanf(const wchar_t * restrict format, va_list arg);
int wprintf(const wchar_t * restrict format, ...);
int wsprintf(const wchar_t * restrict format, ...);
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t * restrict s, FILE * restrict stream);
int fwide(FILE *stream, int mode);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
wint_t ungetwc(wint_t c, FILE *stream);

```

有一个宽字符 I/O 函数没有对应的标准 I/O 函数：

```
int fwide(FILE *stream, int mode)1;
```

如果 mode 为正，函数先尝试把形参表示的流指定为宽字符定向 (*wide-character oriented*)；如果 mode 为负，函数先尝试把流指定为字节定向 (*byte oriented*)；如果 mode 为 0，函数则不改变流的定向。该函数只有在流最初无定向时才改变其定向。在以上所有的情况下，如果流是宽字符定向，函数返回正值；如果流是字节定向，函数返回负值；如果流没有定向，函数则返回 0。

wchar.h 头文件参照 string.h，也提供了一些转换和控制字符串的函数。一般而言，用 wcs 代替 string.h 中的 str 标识符，这样 wcstod() 就是 strtod() 函数的宽字符版本。表 B.5.49 列出了这些函数。

表 B.5.49 宽字符字符串工具

## 函数原型

```

double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);

```

<sup>1</sup> fwide() 函数用于设置流的定向，根据 mode 的不同值来执行不同的工作。——译者注

续表

## 函数原型

```

long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
unsigned long long int wcstoull( const wchar_t * restrict nptr, wchar_t **restrict endptr,
int base);

wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
size_t wcsxfrm(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wcschr(const wchar_t *s, wchar_t c);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
size_t wcslen(const wchar_t *s);
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
size_t wcspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcstr(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t ** restrict ptr);
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
int wmemcmp(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);

```

该头文件还参照 time.h 头文件中的 strftime() 函数，声明了一个时间函数：

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize, const wchar_t * restrict format,
const struct tm * restrict timeptr);
```

除此之外，该头文件还声明了一些用于宽字符字符串和多字节字符相互转换的函数，如表 B.5.50 所列。

表 B.5.50 宽字节和多字节字符转换函数

| 函数原型                              | 描述                                                                                           |
|-----------------------------------|----------------------------------------------------------------------------------------------|
| wint_t btowc(int c);              | 如果在初始移位状态中 c ( unsigned char ) 是有效的单字节字符，那么该函数返回宽字节表示；否则，返回 WEOF                             |
| int wctob(wint_t c);              | 如果 c 是一个扩展字符集的成员，它在初始移位状态中的多字节字符表示的是单字节，该函数就返回一个转换为 int 类型的 unsigned char 的单字节表示；否则，函数返回 EOF |
| int mbsinit(const mbstate_t *ps); | 如果 ps 是空指针或指向一个指定为初始转换状态的数据对象，函数就返回非零值；否则，函数返回 0                                             |

续表

| 函数原型                                                                                                       | 描述                                                                                                                                                                                                                                                                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);                                 | mbrlen() 函数相当于调用 mbrtowc(NULL, s, n, ps != NULL ? ps : &internal)，其中 internal 是 mbrlen() 函数的 mbstate_t 对象，除非 ps 指定的表达式只计算一次                                                                                                                                                                                                                                                                                 |
| size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);        | 如果 s 是空指针，调用该函数相当于把 pwc 设置为空指针、把 n 设置为 1。如果 s 不是空指针，该函数最多检查 n 字节以确定下一个完整的多字节字符所需的字节数（包括所有的移位序列）。如果该函数确定了下一个多字节字符的结束处且合法，它就确定了对应宽字符的值。然后，如果 pwc 不为空，则把值储存在 pwc 指向的对象中。如果对应的宽字符是空的宽字符，描述的最终状态就是初始转换状态。如果检测到空的宽字符，函数返回 0；如果检测到另一个有效宽字符，函数返回完整字符所需的字节数。如果 n 字节不足以表示一个有效的宽字符，但是能表示其中的一部分，函数返回 -2。如果出现编码错误，函数返回 -1，并把 EILSEQ 储存在 errno 中，且不储存任何值                                                                  |
| size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);                                    | 如果 s 是空指针，那么调用该函数相当于把 wc 设置为第 1 个参数使用内部缓冲区。如果 s 不是空指针，wcrtomb() 函数则确定表示 wc 指定宽字符对应的多字节字符表示所需的字节数（包括所有移位序列），并把多字节字符表示储存在一个数组中（s 指向该数组的第一个元素），最多储存 MB_CUR_MAX 字节。如果 wc 是空的宽字符，就在初始移位状态所需的移位序列后储存一个空字节。描述的结果状态就是初始转换状态。如果 wc 是有效的宽字符，该函数返回储存多字节字符所需的字节数（包括指定移位状态的字节）。如果 wc 无效，函数则把 EILSEQ 储存在 errno 中，并返回 -1                                                                                                   |
| size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len, mbstate_t * restrict ps); | mbstrtows() 函数把 src 间接指向的数组中的多字节字符序列转换成对应的宽字符序列，从 ps 指向的对象所描述的转换状态开始，一直转换到结尾的空字符（包括该字符并储存）或转换了 len 个宽字符。如果 dst 不是空指针，则待转换的字符将储存在 dst 指向的数组中。出现这两种情况时停止转换：如果字节序列无法构成一个有效的多字节字符，或者（如果 dst 不是空指针）len 个宽字符已储存在 dst 指向的数组中。每转换一次都相当于调用一次 mbrtowc() 函数。如果 dst 不是空指针，就把空指针（如果因到达结尾的空字符而停止转换）或最后一个待转换多字节字符的地址赋给 src 指向的指针对象。如果由于到达结尾的空字符而停止转换，且 dst 不是空指针，那么描述的结果状态就是初始转状态。如果执行成功，函数返回成功转换的多字节字符数（不包括空字符）；否则函数返回 -1 |
| size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src, size_t len, mbstate_t * restrict ps); | wcsrtombs() 函数把 src 间接指向的数组中的宽字符序列转换成对应的多字节字符序列（从 ps 指向的对象描述的转换状态开始）。如果 dst 不是空指针，待转换的字符将被储存在 dst 指向的数组中。一直转换到结尾的空字符（包括该字符并储存）或换了 len 个多字节字符。出现这两种情况时停止转换：如果宽字符没有对应的有效多字节字符，或者（如果 dst 不是空指针）下一个多字节字超过了储存在 dst 指向的数组中的总字节数 len 的限制。每转换一次都相当于调用一次 wcrtomb() 函数。如果 dst 不是空指针，就把空指针（如果因到达结尾的空字符而停止转换）或最后一个待转换多字节字符的地址赋给 src 指向的指针对象。如果由于到达结尾的空字符而停止转换，描述的结果状态就是初始转状态。如果执行成功，函数返回成功转换的多字节字符数（不包括空字符）；否则函数返回 -1   |

## B.5.28 宽字符分类和映射工具: wctype.h (C99)

wctype.h 库提供了一些与 ctype.h 中的字符函数类似的宽字符函数，以及其他函数。wctype.h

还定义了表 B.5.51 中列出的 3 种类型和宏。

表 B.5.51 wctpe.h 中定义的类型和宏

| 类型/宏      | 描述                                                         |
|-----------|------------------------------------------------------------|
| wint_t    | 整数类型，用于储存扩展字符集中的任意值，还可以储存至少一个不是扩展字符成员的值                    |
| wctrans_t | 标量类型，可以表示本地化指定的字符映射                                        |
| wctype_t  | 标量类型，可以表示本地化指定的字符分类                                        |
| WEOF      | wint_t 类型的常量表达式，不对应扩展字符集中的任何成员，相当于宽字符中的 EOF，用于表示宽字符输入的文件结尾 |

在该库中，如果宽字符参数满足字符分类函数的条件时，函数返回真（非 0）。一般而言，因为单字节字符对应宽字符，所以如果 ctype.h 中对应的函数返回真，宽字符函数也返回真。表 B.5.52 列出了这些函数。

表 B.5.52 宽字节分类函数

| 函数原型                       | 描述                                         |
|----------------------------|--------------------------------------------|
| int iswalnum(wint_t wc);   | 如果 wc 表示一个字母数字字符（字母或数字），函数返回真              |
| int iswalpha(wint_t wc);   | 如果 wc 表示一个字母字符，函数返回真                       |
| int iswblank(wint_t wc);   | 如果 wc 表示一个空格，函数返回真                         |
| int iswcntrl(wint_t wc);   | 如果 wc 表示一个控制字符，函数返回真                       |
| int iswdigit(wint_t wc);   | 如果 wc 表示一个数字，函数返回真                         |
| int iswgraph(wint_t wc);   | 如果 iswprint(wc) 为真，且 iswspace(wc) 为假，函数返回真 |
| int iswlower(wint_t wc);   | 如果 wc 表示一个小写字符，函数返回真                       |
| int iswprint(wint_t wc);   | 如果 wc 表示一个可打印字符，函数返回真                      |
| int iswpunct(wint_t wc);   | 如果 wc 表示一个标点字符，函数返回真                       |
| int iswspace(wint_t wc);   | 如果 wc 表示一个制表符、空格或换行符，函数返回真                 |
| int iswupper(wint_t wc);   | 如果 wc 表示一个大写字符，函数返回真                       |
| int iswdxdigit(wint_t wc); | 如果 wc 表示一个十六进制数字，函数返回真                     |

该库还包含两个可扩展的分类函数，因为它们使用当前本地化的 LC\_CTYPE 值进行分类。表 B.5.53 列出了这些函数。

表 B.5.53 可扩展的宽字符分类函数

| 原型                                      | 描述                                                                                                                                                           |
|-----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| int iswctype(wint_t wc, wctype_t desc); | 如果 wc 具有 desc 描述的属性，函数返回真                                                                                                                                    |
| wctype_t wctype(const char *property);  | wctype() 函数构建了一个 wctype_t 类型的值，它描述了由字符串参数 property 指定的宽字符分类。如果根据当前本地化的 LC_CTYPE 类别，property 识别宽字符分类有效，wctype() 函数则返回非零值（可作为 iswctype() 函数的第 2 个参数）；否则，函数返回 0 |

wctype() 函数的有效参数名即是宽字符分类函数名去掉 isw 前缀。例如，wctype("alpha") 表示

的是 `iswalph()` 函数判断的字符类别。因此，调用 `iswctype(wc, wctype("alpha"))` 相当于调用 `iswalph(wc)`，唯一的区别是前者使用 `LC_CTYPE` 类别进行分类。

该库还有 4 个与转换相关的函数。其中有两个函数分别与 `ctype.h` 库中 `toupper()` 和 `tolower()` 对应。第 3 个函数是一个可扩展的版本，通过本地化的 `LC_CTYPE` 设置确定字符是大写还是小写。第 4 个函数为第 3 个函数提供合适的分类参数。表 B.5.54 列出了这些函数。

表 B.5.54 宽字符转换函数

| 原型                                                        | 描述                                                                                                                                                                                                                                      |
|-----------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>wint_t towlower(wint_t wc);</code>                  | 如果 <code>wc</code> 是大写字符，返回其小写形式；否则返回 <code>wc</code>                                                                                                                                                                                   |
| <code>wint_t towupper(wint_t wc);</code>                  | 如果 <code>wc</code> 是小写字符，返回其大写形式；否则返回 <code>wc</code>                                                                                                                                                                                   |
| <code>wint_t towctrans(wint_t wc, wctrans_t desc);</code> | 如果 <code>desc</code> 等于 <code>wctrans("lower")</code> 的返回值，函数返回 <code>wc</code> 的小写形式（由 <code>LC_CTYPE</code> 设置确定）；如果 <code>dest</code> 等于 <code>wctrans("upper")</code> 的返回值，函数返回 <code>wc</code> 的大写形式（由 <code>LC_CTYPE</code> 设置确定） |
| <code>wctrans_t wctrans(const char*property);</code>      | 如果参数是 "lower" 或 "upper"，函数返回一个 <code>wctrans_t</code> 类型的值，可用作 <code>towctrans()</code> 的参数并反映 <code>LC_CTYPE</code> 设置，否则函数返回 0                                                                                                        |

## B.6 参考资料 VI：扩展的整数类型

第 3 章介绍过，C99 的 `inttypes.h` 头文件为不同的整数类型提供一套系统的别名。这些名称与标准名称相比，能更清楚地描述类型的性质。例如，`int` 类型可能是 16 位、32 位或 64 位，但是 `int32_t` 类型一定是 32 位。

更精确地说，`inttypes.h` 头文件定义的一些宏可用于 `scanf()` 和 `printf()` 函数中读写这些类型的整数。`inttypes.h` 头文件包含的 `stdlib.h` 头文件提供实际的类型定义。格式化宏可以与其他字符串拼接起来形成合适格式化的字符串。

该头文件中的类型都使用 `typedef` 定义。例如，32 位系统的 `int` 可能使用这样的定义：

```
typedef int int32_t;
```

用 `#define` 指令定义转换说明。例如，使用之前定义的 `int32_t` 的系统可以这样定义：

```
#define PRIId32 "d" // 输出说明符
#define SCNd32 "d" // 输入说明符
```

使用这些定义，可以声明扩展的整型变量、输入一个值和显示该值：

```
int32_t cd_sales; // 32 位整数类型
scanf("%" SCNd32, &cd_sales);
printf("CD sales = %10" PRIId32 " units\n", cd_sales);
```

如果需要，可以把字符串拼接起得到最终的格式字符串。因此，上面的代码可以这样写：

```
int cd_sales; // 32 位整数类型
scanf("%d", &cd_sales);
printf("CD sales = %10d units\n", cd_sales);
```

如果把原始代码移植到 16 位 `int` 的系统中，该系统可能把 `int32_t` 定义为 `long`，把 `PRIId32` 定义为 "`ld`"。但是，仍可以使用相同的代码，只要知道系统使用的是 32 位整型即可。

该参考资料的其余部分列出了扩展类型、转换说明以及表示类型限制的宏。

## B.6.1 精确宽度类型

`typedef` 标识了一组精确宽度的类型，通用形式是 `intN_t`（有符号类型）和 `uintN_t`（无符号类型），其中 `N` 表示位数（即类型的宽度）。但是要注意，不是所有的系统都支持所有的这些类型。例如，最小可用内存大小是 16 位的系统就不支持 `int8_t` 和 `uint8_t` 类型。格式宏可以使用 `d` 或 `i` 表示有符号类型，所以 `PRIi8` 和 `SCNi8` 都有效。对于无符号类型，可以使用 `o`、`x` 或 `u` 以获得 `%o`、`%x` 或 `%X` 转换说明来代替 `%u`。例如，可以使用 `PRIX32` 以十六进制格式打印 `uint32_t` 类型的值。表 B.6.1 列出了精确宽度类型、格式说明符和最小值、最大值。

表 B.6.1 精确宽度类型

| 类型名                   | printf() 说明符        | scanf() 说明符         | 最小值                    | 最大值                     |
|-----------------------|---------------------|---------------------|------------------------|-------------------------|
| <code>int8_t</code>   | <code>PRId8</code>  | <code>SCNd8</code>  | <code>INT8_MIN</code>  | <code>INT8_MAX</code>   |
| <code>int16_t</code>  | <code>PRId16</code> | <code>SCNd16</code> | <code>INT16_MIN</code> | <code>INT16_MAX</code>  |
| <code>int32_t</code>  | <code>PRId32</code> | <code>SCNd32</code> | <code>INT32_MIN</code> | <code>INT32_MAX</code>  |
| <code>int64_t</code>  | <code>PRId64</code> | <code>SCNd64</code> | <code>INT64_MIN</code> | <code>INT64_MAX</code>  |
| <code>uint8_t</code>  | <code>PRIu8</code>  | <code>SCNu8</code>  | 0                      | <code>UINT8_MAX</code>  |
| <code>uint16_t</code> | <code>PRIu16</code> | <code>SCNu16</code> | 0                      | <code>UINT16_MAX</code> |
| <code>uint32_t</code> | <code>PRIu32</code> | <code>SCNu32</code> | 0                      | <code>UINT32_MAX</code> |
| <code>uint64_t</code> | <code>PRIu64</code> | <code>SCNu64</code> | 0                      | <code>UINT64_MAX</code> |

## B.6.2 最小宽度类型

最小宽度类型保证一种类型的大小至少是某位。这些类型一定存在。例如，不支持 8 位单元的系统可以把 `int_least_8` 定义为 16 位类型。表 B.6.2 列出了最小宽度类型、格式说明符和最小值、最大值。

表 B.6.2 最小宽度类型

| 类型名                         | printf() 说明符             | scanf() 说明符              | 最小值                          | 最大值                           |
|-----------------------------|--------------------------|--------------------------|------------------------------|-------------------------------|
| <code>int_least8_t</code>   | <code>PRILEASTd8</code>  | <code>SCNLEASTd8</code>  | <code>INT_LEAST8_MIN</code>  | <code>INT_LEAST8_MAX</code>   |
| <code>int_least16_t</code>  | <code>PRILEASTd16</code> | <code>SCNLEASTd16</code> | <code>INT_LEAST16_MIN</code> | <code>INT_LEAST16_MAX</code>  |
| <code>int_least32_t</code>  | <code>PRILEASTd32</code> | <code>SCNLEASTd32</code> | <code>INT_LEAST32_MIN</code> | <code>INT_LEAST32_MAX</code>  |
| <code>int_least64_t</code>  | <code>PRILEASTd64</code> | <code>SCNLEASTd64</code> | <code>INT_LEAST64_MIN</code> | <code>INT_LEAST64_MAX</code>  |
| <code>uint_least8_t</code>  | <code>PRILEASTu8</code>  | <code>SCNLEASTu8</code>  | 0                            | <code>UINT_LEAST8_MAX</code>  |
| <code>uint_least16_t</code> | <code>PRILEASTu16</code> | <code>SCNLEASTu16</code> | 0                            | <code>UINT_LEAST16_MAX</code> |
| <code>uint_least32_t</code> | <code>PRILEASTu32</code> | <code>SCNLEASTu32</code> | 0                            | <code>UINT_LEAST32_MAX</code> |
| <code>uint_least64_t</code> | <code>PRILEASTu64</code> | <code>SCNLEASTu64</code> | 0                            | <code>UINT_LEAST64_MAX</code> |

## B.6.3 最快最小宽度类型

对于特定的系统，用特定的整型更快。例如，在某些实现中 `int_least16_t` 可能是 `short`，但是系统在进行算术运算时用 `int` 类型会更快些。因此，`inttypes.h` 还定义了表示为某位数的最快类型。这些类型一定存在。在某些情况下，可能并未明确指定哪种类型最快，此时系统会简单地选择其中的一种。表 B.6.3 列出了最快最小宽度类型、格式说明符和最小值、最大值。

表 B.6.3 最快最小宽度类型

| 类型名           | printf()说明符 | scanf()说明符 | 最小值            | 最大值             |
|---------------|-------------|------------|----------------|-----------------|
| int_fast8_t   | PRIFASTd8   | SCNFASTd8  | INT_FAST8_MIN  | INT_FAST8_MAX   |
| int_fast16_t  | PRIFASTd16  | SCNFASTd16 | INT_FAST16_MIN | INT_FAST16_MAX  |
| int_fast32_t  | PRIFASTd32  | SCNFASTd32 | INT_FAST32_MIN | INT_FAST32_MAX  |
| int_fast64_t  | PRIFASTd64  | SCNFASTd64 | INT_FAST64_MIN | INT_FAST64_MAX  |
| uint_fast8_t  | PRIFASTu8   | SCNFASTu8  | 0              | UINT_FAST8_MAX  |
| uint_fast16_t | PRIFASTu16  | SCNFASTu16 | 0              | UINT_FAST16_MAX |
| uint_fast32_t | PRIFASTu32  | SCNFASTu32 | 0              | UINT_FAST32_MAX |
| uint_fast64_t | PRIFASTu64  | SCNFASTu64 | 0              | UINT_FAST64_MAX |

## B.6.4 最大宽度类型

有些情况下要使用最大整数类型，表 B.6.4 列出了这些类型。实际上，由于系统可能会提供比所需类型更大宽度的类型，因此这些类型的宽度可能比 long long 或 unsigned long long 更大。

表 B.6.4 最大宽度类型

| 类型名       | printf()说明符 | scanf()说明符 | 最小值        | 最大值         |
|-----------|-------------|------------|------------|-------------|
| intmax_t  | PRIIdMAX    | SCNdMAX    | INTMAX_MIN | INTMAX_MAX  |
| uintmax_t | PRIuMAX     | SCBuMAX    | 0          | UINTMAX_MAX |

## B.6.5 可储存指针值的整型

inttypes.h 头文件（通过包含 stdint.h 即可包含该头文件）定义了两种整数类型，可精确地储存指针值，见表 B.6.5。

表 B.6.5 可储存指针值的整数类型

| 类型名       | printf()说明符 | scanf()说明符 | 最小值        | 最大值         |
|-----------|-------------|------------|------------|-------------|
| intptr_t  | PRIIdPTR    | SCNdPTR    | INTPTR_MIN | INTPTR_MAX  |
| uintptr_t | PRIuPTR     | SCBuPTR    | 0          | UINTPTR_MAX |

## B.6.6 扩展的整型常量

在整数后面加上 L 后缀可表示 long 类型的常量，如 445566L。如何表示 int32\_t 类型的常量？要使用 inttypes.h 头文件中定义的宏。例如，表达式 INT32\_C(445566) 展开为一个 int32\_t 类型的常量。从本质上讲，这种宏相当于把当前类型强制转换成底层类型，即特殊实现中表示 int32\_t 类型的基本类型。

宏名是把相应类型名中的\_c 用\_t 替换，再把名称中所有的字母大写。例如，要把 1000 设置为 unit\_least64\_t 类型的常量，可以使用表达式 UNIT\_LEAST64\_C(1000)。

## B.7 参考资料 VII：扩展字符支持

C 语言最初并不是作为国际编程语言设计的，其字符的选择或多或少是基于标准的美国键盘。但是，随着后来 C 在世界范围内越来越流行，不得不扩展来支持不同且更大的字符集。这部分参考资料概括介绍了一些相关内容。

## B.7.1 三字符序列

有些键盘没有 C 中使用的所有符号，因此 C 提供了一些由三个字符组成的序列（即三字符序列）作为这些符号的替换表示。如表 B.7.1 所示。

表 B.7.1 三字符序列

| 三字符序列 | 符号 | 三字符序列 | 符号 | 三字符序列 | 符号 |
|-------|----|-------|----|-------|----|
| ??=   | #  | ??(   | [  | ??/   | \  |
| ??)   | ]  | ??'   | ^  | ??<   | {  |
| ??!   |    | ??>   | }  | ??-   | ~  |

C 替换了源代码文件中的这些三字符序列，即使它们在双引号中也是如此。因此，下面的代码：

```
??=include <stdio.h>
??=define LIM 100
int main()
??<
    int q??(LIM??);
    printf("More to come.??/n");
    ...
??>
```

会变成这样：

```
#include <stdio.h>
#define LIM 100
int main()
{
    int q[LIM];
    printf("More to come.\n");
    ...
}
```

当然，要在编译器中设置相关选项才能激活这个特性。

## B.7.2 双字符

意识到三字符系统很笨拙，C99 提供了双字符 (*digraph*)，可以使用它们来替换某些标准 C 标点符号。

表 B.7.2 双字符

| 双字符 | 符号 | 双字符 | 符号 | 双字符  | 符号 |
|-----|----|-----|----|------|----|
| <:  | [  | :>  | ]  | <%   | {  |
| %>  | }  | %:  | #  | %:%: | ## |

与三字符不同的是，不会替换双引号中的双字符。因此，下面的代码：

```
%:include <stdio.h>
%:define LIM 100
int main()
<%
    int q<:LIM:>;
    printf("More to come.:>");
    ...
%>
```

会变成这样：

```
#include <stdio.h>
#define LIM 100
int main()
{
    int q[LIM];
    printf("More to come.:>"); // :>是字符串的一部分
    ...
}
```

// :>与 } 相同

### B.7.3 可选拼写：iso646.h

使用三字符序列可以把 || 运算符写成 ??! ??!, 这看上去比较混乱。C99 通过 iso646.h 头文件（参考资料 V 中的表 B.5.11）提供了可展开为运算符的宏。C 标准把这些宏称为可选拼写（*alternative spelling*）。

如果包含了 iso646.h 头文件，以下代码：

```
if(x == M1 or x == M2)
    x and_eq 0xFF;
```

可展开为下面的代码：

```
if(x == M1 || x == M2)
    x &= 0xFF;
```

### B.7.4 多字节字符

C 标准把多字节字符描述为一个或多个字节的序列，表示源环境或执行环境中的扩展字符集成员。源环境指的是编写源代码的环境，执行环境指的是用户运行已编译程序的环境。这两个环境不同。例如，可以在一个环境中开发程序，在另一个环境中运行该程序。扩展字符集是 C 语言所需的基本字符集的超集。

有些实现会提供扩展字符集，方便用户通过键盘输入与基本字符集不对应的字符。这些字符可用于字符串字面量和字符常量中，也可出现在文件中。有些实现会提供与基本字符集等效的多字节字符，可替换三字符和双字符。

例如，德国的一个实现也许会允许用户在字符串中使用日耳曼元音变音字符：

```
puts("eins zwei drei vier fünf");
```

一般而言，程序可使用的扩展字符集因本地化设置而异。

### B.7.5 通用字符名（UCN）

多字节字符可以用在字符串中，但是不能用在标识符中。C99 新增了通用字符名（UCN），允许用户在标识名中使用扩展字符集中的字符。系统扩展了转义序列的概念，允许编码 ISO/IEC 10646 标准中的字符。该标准由国际标准化组织（ISO）和国际电工技术委员会（IEC）共同制定，为大量的字符提供数值码。10646 标准和统一码（*Unicode*）关系密切。

有两种形式的 UCN 序列。第 1 种形式是 \u hexguard，其中 hexguard 是一个 4 位的十六进制数序列（如，\u00F6）。第 2 种形式是 \U hexguardhexguard，如 \U0000AC01。因为十六进制每一位上的数对应 4 位，\u 形式可用于 16 位整数表示的编码，\U 形式可用于 32 位整数表示的编码。

如果系统实现了 UCN，而且包含了扩展字符集中所需的字符，就可以在字符串、字符常量和标识符中使用 UCN：

```
wchar_t value\u00F6\u00F8 = L'\u00f6';
```

## 统一码和 ISO 10646

统一码为表示不同的字符集提供了一种解决方案，可以根据类型为大量字符和符号制定标准的编号系统。例如，ASCII 码被合并为统一码的子集，因此美国拉丁字符（如 A~Z）在这两个系统中的编码相同。但是，统一码还合并了其他拉丁字符（如，欧洲语言中使用的一些字符）和其他语言中的字符，包括希腊文、西里尔字母、希伯来文、切罗基文、阿拉伯文、泰文、孟加拉文和形意文字（如中文和日文）。到目前为止，统一码表示的符号超过了 110000 个，而且仍在发展中。欲了解更多细节，请查阅统一码联合站点：[www.unicode.org](http://www.unicode.org)。

统一码为每个字符分配一个数字，这个数字称为代码点（*code point*）。典型的统一码代码点类似：U-222B。U 表示该字符是统一字符，222B 是表示该字符的一个十六进制数，在这种情况下，表示积分号。

国际标准化组织（ISO）组建了一个团队开发 ISO 10646 和标准编码的多语言文本。ISO 10646 团队和统一码团队从 1991 年开始合作，一直保持两个标准的相互协调。

## B.7.6 宽字符

C99 为使用宽字符提供更多支持，通过 `wchar.h` 和 `wctype.h` 库包含了更多大型字符集。这两个头文件把 `wchar_t` 定义为一种整型类型，其确切的类型依赖实现。该类型用于储存扩展字符集中的字符，扩展字符集是基本字符集的超集。根据定义，`char` 类型足够处理基本字符集，而 `wchar_t` 类型则需要更多位才能储存更大范围的编码值。例如，`char` 可能是 8 位字节，`wchar_t` 可能是 16 位的 `unsigned short`。

用 `L` 前缀标识宽字符常量和字符串字面量，用 `%lc` 和 `%ls` 显示宽字符数据：

```
wchar_t wch = L'I';
wchar_t w_arr[20] = L"am wide!";
printf("%lc %ls\n", wch, w_arr);
```

例如，如果把 `wchar_t` 实现为 2 字节单元，'I' 的 1 字节编码应储存在 `wch` 的低位字节。不是标准字符集中的字符可能需要两个字节储存字符编码。例如，可以使用通用字符编码表示超出 `char` 类型范围的字符编码：

```
wchar_t w = L'\u00E2'; /* 16 位编码值 */
```

内含 `wchar_t` 类型值的数组可用于储存宽字符串，每个元素储存一个宽字符编码。编码值为 0 的 `wchar_t` 值是空字符的 `wchar_t` 类型等价字符。该字符被称为空宽字符（*null wide character*），用于表示宽字符串的结尾。

可以使用 `%lc` 和 `%ls` 读取宽字符：

```
wchar_t wchl;
wchar_t w_arr[20];
puts("Enter your grade:");
scanf("%lc", &wchl);
puts("Enter your first name:");
scanf("%ls", w_arr);
```

`wchar_t` 头文件为宽字符提供更多支持，特别是提供了宽字符 I/O 函数、宽字符转换函数和宽字符串控制函数。例如，可以用 `fwprintf()` 和 `wprintf()` 函数输出，用 `fwscanf()` 和 `wscanf()` 函数输入。与一般输入/输出函数的主要区别是，这些函数需要宽字符格式字符串，处理的是宽字符输入/输出流。例如，下面的代码把信息作为宽字符显示：

```
wchar_t * pw = L"Points to a wide-character string";
int dozen = 12;
wprintf(L"Item %d: %ls\n", dozen, pw);
```

类似地，还有 `getwchar()`、`putwchar()`、`fgetws()` 和 `fputws()` 函数。`wchar_t` 头文件定义了一个 `WEOF` 宏，与 `EOF` 在面向字节的 I/O 中起的作用相同。该宏要求其值是一个与任何有效字符都不对应的值。因为 `wchar_t` 类型的值都有可能是有效字符，所以 `wchar_t` 库定义了一个 `wint_t` 类型，包含了所有 `wchar_t` 类型的值和 `WEOF` 的值。

该库中还有与 `string.h` 库等价的函数。例如，`wcscpy(ws1, ws2)` 把 `ws1` 指定的宽字符串拷贝到 `ws2` 指向的宽字符数组中。类似地，`wcscmp()` 函数比较宽字符串，等等。

`wctype.h` 头文件新增了字符分类函数，例如，如果 `iswdigit()` 函数的宽字符参数是数字，则返回真；如果 `iswblank()` 函数的参数是空白，则返回真。空白的标准值是空格和水平制表符，分别写作 `L' '` 和 `L'\t'`。

C11 标准通过 `uchar.h` 头文件为宽字符提供更多支持，为匹配两种常用的统一码格式，定义了两个新类型。第 1 种类型是 `char16_t`，可储存一个 16 位编码，是可用的最小无符号整数类型，用于 `hexguard` UCN 形式和统一码 UTF-16 编码方案。

```
char16_t = '\u00F6';
```

第 2 种类型是 `char32_t`，可储存一个 32 位编码，最小的可用无符号整数类型，可用于 `hexguard` UCN 形式和统一码 UTF-32 编码方案

```
char32_t = '\u0000AC01';
```

前缀 `u` 和 `U` 分别表示 `char16_t` 和 `char32_t` 字符串。

```
char16_t ws16[11] = u"Tannh\u00E4user";
char32_t ws32[13] = U"caf\U000000E9 au lait";
```

注意，这两种类型比 `wchar_t` 类型更具体。例如，在一个系统中，`wchar_t` 可以储存 32 位编码，但是在另一个系统中也许只能储存 16 位的编码。另外，这两种新类型都与 C++ 兼容。

## B.7.7 宽字符和多字节字符

宽字符和多字节字符是处理扩展字符集的两种不同的方法。例如，多字节字符可能是一个字节、两个字节、三个字节或更多字节，而所有的宽字符都只有一个宽度。多字节字符可能使用移位状态（移位状态是一个字节，确定如何解释后续字节）；而宽字符没有移位状态。可以把多字节字符的文件读入使用标准输入函数的普通 `char` 类型数组，把宽字符的文件读入使用宽字符输入函数的宽字节数组。

C99 在 `wchar.h` 库中提供了一些函数，用于多字节和宽字节之间的转换。`mbrtowc()` 函数把多字节字符转换为宽字符，`wcrtombs()` 函数把宽字符转换为多字节字符。类似地，`mbstrtowcs()` 函数把多字节字符串转换为宽字节字符串，`wcstrtombs()` 函数把宽字节字符串转换为多字节字符串。

C11 在 `uchar.h` 库中提供了一些函数，用于多字节和 `char16_t` 之间的转换，以及多字节和 `char32_t` 之间的转换。

## B.8 参考资料 VIII：C99/C11 数值计算增强

过去，FORTRAN 是数值科学计算和工程计算的首选语言。C90 使 C 的计算方法更接近于 FORTRAN。例如，`float.h` 中使用的浮点特性规范都是基于 FORTRAN 标准委员会开发的模型。C99 和 C11 标准继续增强了 C 的计算能力。例如，C99 新增的变长数组（C11 成为可选的特性），比传统的 C 数组更符合 FORTRAN 的用法（如果实现不支持变长数组，C11 指定了 `_STDC_NO_VLA_` 宏的值为 1）。

## B.8.1 IEC 浮点标准

国际电工技术委员会 (IEC) 已经发布了一套浮点计算的标准 (IEC 60559)。该标准包括了浮点数的格式、精度、NaN、无穷值、舍入规则、转换、异常以及推荐的函数和算法等。C99 纳入了该标准，将其作为 C 实现浮点计算的指导标准。C99 新增的大部分浮点工具 (如, `fenv.h` 头文件和一些新的数学函数) 都基于此。另外, `float.h` 头文件定义了一些与 IEC 浮点模型相关的宏。

### 1. 浮点模型

下面简要介绍一下浮点模型。标准把浮点数  $x$  看作是一个基数的某次幂乘以一个分数，而不是 C 语言的 E 记数法 (例如，可以把 876.54 写成 0.87654E3)。正式的浮点表示更为复杂：

$$x = sb^e \sum_{k=1}^p f_k b^{-k}$$

简单地说，这种表示法把一个数表示为有效数 (*significand*) 与  $b$  的  $e$  次幂的乘积。

下面是各部分的含义。

$s$  代表符号 ( $\pm 1$ )。

$b$  代表基数。最常见的值是 2，因为浮点处理器通常使用二进制数学。

$e$  代表整数指数 (不要与自然对数中使用的数值常量  $e$  混淆)，限制最小值和最大值。这些值依赖于留出储存指数的位数。

$f_k$  代表基数为  $b$  时可能的数字。例如，基数为 2 时，可能的数字是 0 和 1；在十六进制中，可能的数字是 0~F。

$p$  代表精度，基数为  $b$  时，表示有效数的位数。其值受限于预留储存有效数字的位数。

明白这种表示法的关键是理解 `float.h` 和 `fenv.h` 的内容。下面，举两个例子解释内部如何表示浮点数。

首先，假设一个浮点数的基数  $b$  为 10，精度  $p$  为 5。那么，根据上面的表示法，24.51 应写成：

$$(+1) 10^3 (2/10 + 4/100 + 5/1000 + 1/10000 + 0/100000)$$

假设计算机可储存十进制数 (0~9)，那么可以储存符号、指数 3 和 5 个  $f_k$  值：2、4、5、1、0 (这里,  $f_1$  是 2,  $f_2$  是 4, 等等)。因此，有效数是 0.24510，乘以  $10^3$  得 24.51。

接下来，假设符号为正，基数  $b$  是 2,  $p$  是 7 (即，用 7 位二进制数表示)，指数是 5，待储存的有效数是 1011001。下面，根据上面的公式构造该数：

$$\begin{aligned} x &= (+1) 2^5 (1/2 + 0/4 + 1/8 + 1/16 + 0/32 + 0/64 + 1/128) \\ &= 32(1/2 + 0/4 + 1/8 + 1/16 + 0/32 + 0/64 + 1/128) \\ &= 16 + 0 + 4 + 2 + 0 + 0 + 1/4 = 22.25 \end{aligned}$$

`float.h` 中的许多宏都与该浮点表示相关。例如，对于一个 `float` 类型的值，表示基数的 `FLT_RADIX` 是  $b$ ，表示有效数位数 (基数为  $b$  时) 的 `FLT_MANT_DIG` 是  $p$ 。

### 2. 正常值和低于正常的值

正常浮点值 (*normalized floating-point value*) 的概念非常重要，下面简要介绍一下。为简单起见，先假设系统使用十进制 ( $b = \text{FLT\_RADIX} = 10$ ) 和浮点值的精度为 5 ( $p = \text{FLT\_MANT\_DIG} = 5$ ) (标准要求的精度更高)。考虑下面表示 31.841 的方式：

指数 = 3，有效数 = .31841 (.31841E3)

指数 = 4, 有效数 = .03184 (.03184E4)

指数 = 5, 有效数 = .00318 (.00318E5)

显而易见, 第 1 种方法精度最高, 因为在有效数中使用了所有的 5 位可用位。规范化浮点非零值是第 1 位有效位为非零的值, 这也是通常储存浮点数的方式。

现在, 假设最小指数 (FLT\_MIN\_EXP) 是 -10, 那么最小的规范值是:

指数 = -10, 有效数 = .10000 (.10000E-10)

通常, 乘以或除以 10 意味着使指数增大或减小, 但是在这种情况下, 如果除以 10, 却无法再减小指数。但是, 可以改变有效数获得这种表示:

指数 = -10, 有效数 = .01000 (.01000E-10)

这个数被称为低于正常的 (*subnormal*), 因为该数并未使用有效数的全精度。例如, 0.12343E-10 除以 10 得 .01234E-10, 损失了一位的信息。

对于这个特例, 0.1000E-10 是最小的非零正常值 (FLT\_MIN), 最小的非零低于正常值是 0.00001E-10 (FLT\_TRUE\_MIN)。

float.h 中的宏 FLT\_HAS\_SUBNORM、DBL\_HAS\_SUBNORM 和 LDBL\_HAS\_SUBNORM 表征实现如何处理低于正常的值。下面是这些宏可能会用到的值及其含义:

-1 不确定 (尚未统一)

0 不存在 (例如, 实现可能会用 0 替换低于正常的值)

1 存在

math.h 库提供一些方法, 包括 fpclassify() 和 isnormal() 宏, 可以识别程序何时生成低于正常的值, 这样会损失一些精度。

### 3. 求值方案

float.h 中的宏 FLT\_EVAL\_METHOD 确定了实现采用何种浮点表达式的求值方案, 如下所示 (有些实现还会提供其他负值选项)。

-1 不确定

0 对在所有浮点类型范围和精度内的操作、常量求值

1 对在 double 类型的精度内和 float、double 类型的范围内的操作、常量求值, 对 longdouble 范围内的 long double 类型的操作、常量求值

2 对所有浮点类型范围内和 long double 类型精度内的操作和常量求值

例如, 假设程序中要把两个 float 类型的值相乘, 并把乘积赋给第 3 个 float 类型变量。对于选项 1 (即 K&R C 采用的方案), 这两个 float 类型的值将被扩展为 double 类型, 使用 double 类型完成乘法计算, 然后在赋值计算结果时再把乘积转为 float 类型。

如果选择 0 (即 ANSI C 采用的方案), 实现将直接使用这两个 float 类型的值相乘, 然后赋值乘积。这样做比选项 1 快, 但是会稍微损失一点精度。

### 4. 舍入

float.h 中的宏 FLT\_ROUNDS 确定了系统如何处理舍入, 其指定值所对应的舍入方案如下所示。

-1 不确定

0 趋零截断

1 舍入到最接近的值

- 2 趋向正无穷
- 3 趋向负无穷

系统可以定义其他值，对应其他舍入方案。

一些系统提供控制舍入的方案，在这种情况下，`fenv.h` 中的 `feastround()` 函数提供编程控制。

如果只是计算制作 37 个蛋糕需要多少面粉，这些不同的舍入方案可能并不重要，但是对于金融和科学计算而言，这很重要。显然，把较高精度的浮点值转换成较低精度值时需要使用舍入方案。例如，把 `double` 类型的计算结果赋给 `float` 类型的变量。另外，在改变进制时，也会用到舍入方案。不同进制下精确表示的分数不同。例如，考虑下面的代码：

```
float x = 0.8;
```

在十进制下， $8/10$  或  $4/5$  都可以精确表示 0.8。但是大部分计算机系统都以二进制储存结果，在二进制下， $4/5$  表示为一个无限循环小数：

```
0.1100110011001100...
```

因此，在把 0.8 储存在 `x` 中时，将其舍入为一个近似值，其具体值取决于使用的舍入方案。

尽管如此，有些实现可能不满足 IEC 60559 的要求。例如，底层硬件可能无法满足要求。因此，C99 定义了两个可用作预处理器指令的宏，检查实现是否符合规范。第 1 个宏是 `_STDC_IEC_559_`，如果实现遵循 IEC 60559 浮点规范，该宏被定义为常量 1。第 2 个宏是 `_STDC_IEC_559_COMPLEX_`，如果实现遵循 IEC 60559 兼容复数运算，该宏被定义为常量 1。

如果实现中未定义这两个宏，则不能保证遵循 IEC 60559。

## B.8.2 `fenv.h` 头文件

`fenv.h` 头文件提供一些与浮点环境交互的方法。也就是说，允许用户设置浮点控制模式值（该值管理如何执行浮点运算）并确定浮点状态标志（或异常）的值（报告运算效果的信息）。例如，控制模式设置可指定舍入的方案；如果运算出现浮点溢出则设置一个状态标志。设置状态标志的操作叫作抛出异常。

状态标志和控制模式只有在硬件支持的前提下才能发挥作用。例如，如果硬件没有这些选项，则无法更改舍入方案。

使用下面的编译指示开启支持：

```
#pragma STDC FENV_ACCESS ON
```

这意味着程序到包含该编译指示的块末尾一直支持，或者如果该编译指示是外部的，则支持到该文件或翻译单元的末尾。使用下面的编译指示关闭支持：

```
#pragma STDC FENV_ACCESS OFF
```

使用下面的编译指示可恢复编译器的默认设置，具体设置取决于实现：

```
#pragma STDC FENV_ACCESS DEFAULT
```

如果涉及关键的浮点运算，这个功能非常重要。但是，一般用户使用的程度有限，所以本附录不再深入讨论。

## B.8.3 `STDC FP_CONTRACT` 编译指示

一些浮点数处理器可以把多个运算符的浮点表达式合并成一个运算。例如，处理器只需一步就求出下面表达式的值：

```
x*y - z
```

这加快了运算速度，但是减少了运算的可预测性。`STDC FP_CONTRACT` 编译指示允许用户开启或关

闭这个特性。默认状态取决于实现。

为特定运算关闭合并特性，然后再开启，可以这样做：

```
#pragma STDC FP_CONTRACT OFF
val = x * y - z;
#pragma STDC FP_CONTRACT ON
```

## B.8.4 math.h 库增补

大部分 C90 数学库中都声明了 double 类型参数和 double 类型返回值的函数，例如：

```
double sin(double);
double sqrt(double);
```

C99 和 C11 库为所有这些函数都提供了 float 类型和 long double 类型的函数。这些函数的名称由原来函数名加上 f 或 l 后缀构成，例如：

```
float sinf(float);           /* sin() 的 float 版本 */
long double sinl(long double); /* sin() 的 long double 版本 */
```

有了这些不同精度的函数系列，用户可以根据具体情况选择最效率的类型和函数组合。

C99 还新增了一些科学、工程和数学运算中常用的函数。表 B.5.16 列出了所有数学函数的 double 版本。在许多情况下，这些函数的返回值都可以使用现有的函数计算得出，但是新函数计算得更快更精确。例如， $\log_10(x)$  表示的值与  $\log(1 + x)$  相同，但是  $\log_10(x)$  使用了不同的算法，对于较小的 x 值而言计算更精确。因此，可以使用  $\log()$  函数作普通运算，但是对于精确要求较高且 x 值较小时，用  $\log_10()$  函数更好。

除这些函数以外，数学库中还定义了一些常量和与数字分类、舍入相关的函数。例如，可以把值分为无穷值、非数 (NaN)、正常值、低于正常的值、真零。[NaN 是一个特别的值，用于表示一个不是数的值。例如， $\sin(2.0)$  返回 NaN，因为定义了  $\sin()$  函数的参数必须是  $-1 \sim 1$  范围内的值。低于正常的值是比使用全精度表示的最小值还要小的数。]还有一些专用的比较函数，如果一个或多个参数是非正常值时，函数的行为与标准的关系运算符不同。

使用 C99 的分类方案可以检测计算的规律性。例如，math.h 中的 `isnormal()` 宏，如果其参数是一个正常的数，则返回真。下面的代码使用该宏在 num 不正常时结束循环：

```
#include <math.h>      // 为了使用 isnormal()
...
float num = 1.7e-19;
float numprev = num;

while (isnormal(num)) // 当 num 为全精度的 float 类型值
{
    numprev = num;
    num /= 13.7f;
}
```

简而言之，数学库为更好地控制如何计算浮点数，提供了扩展支持。

## B.8.5 对复数的支持

复数是有实部和虚部的数。实部是普通的实数，如浮点类型表示的数。虚部表示一个虚数。虚数是  $-1$  的平方根的倍数。在数学中，复数通常写作类似  $4.2 + 2.0i$  的形式，其中  $i$  表示  $-1$  的平方根。

C99 支持 3 种复数类型（在 C11 中为可选）：

- float \_Complex
- double \_Complex
- long double \_Complex

例如，储存 float \_Complex 类型的值时，使用与两个 float 类型元素的数组相同的内存布局，实部值储存在第 1 个元素中，虚部值储存在第 2 个元素中。

C99 和 C11 还支持下面 3 种虚类型：

- float \_Imaginary
- double \_Imaginary
- long double \_Imaginary

包含了 complex.h 头文件，就可以用 complex 代替 \_Complex，用 imaginary 代替 \_Imaginary。

为复数类型定义的算术运算遵循一般的数学规则。例如， $(a+b*I)*(c+d*I)$  即是  $(a*c-b*d)+(b*c+a*d)*I$ 。

complex.h 头文件定义了一些宏和接受复数参数并返回复数的函数。特别是，宏 I 表示 -1 的平方根。因此，可以编写这样的代码：

```
double complex c1 = 4.2 + 2.0 * I;
float imaginary c2= -3.0 * I;
```

C11 提供了另一种方法，通过 CMPLX() 宏给复数赋值。例如，如果 re 和 im 都是 double 类型的值，可以这样做：

```
double complex c3 = CMPLX(re, im);
```

这种方法的目的是，宏在处理不常见的情况（如，im 是无穷大或非数）时比直接赋值好。

complex.h 头文件提供了一些复数函数的原型，其中许多复数函数都有对应 math.h 中的函数，其函数名即是对应函数名前加上 c 前缀。例如，csin() 返回其复数参数的复正弦。其他函数与特定的复数特性相关。例如，creal() 函数返回一个复数的实部，cimag() 函数返回一个复数的虚部。也就是说，给定一个 double complex 类型的 z，下面的代码为真：

```
z = creal(z) + cimag(z) * I;
```

如果熟悉复数，需要使用复数，请详细阅读 complex.h 中的内容。

下面的示例演示了对复数的一些支持：

```
// complex.c -- 复数
#include <stdio.h>
#include <complex.h>
void show_cmlx(complex double cv);
int main(void)
{
    complex double v1 = 4.0 + 3.0*I;
    double re, im;
    complex double v2;
    complex double sum, prod, conjug;

    printf("Enter the real part of a complex number: ");
    scanf("%lf", &re);
    printf("Enter the imaginary part of a complex number: ");
    scanf("%lf", &im);
    // CMPLX() 是 C11 中的一个特性
    // v2 = CMPLX(re, im);
    v2 = re + im * I;
```

```

printf("v1: ");
show_cmlx(v1);
putchar('\n');
printf("v2: ");
show_cmlx(v2);
putchar('\n');
sum = v1 + v2;
prod = v1 * v2;
conjug =conj(v1);
printf("sum: ");
show_cmlx(sum);
putchar('\n');
printf("product: ");
show_cmlx(prod);
putchar('\n');

printf("complex conjugate of v1: ");
show_cmlx(conjug);
putchar('\n');

return 0;
}
void show_cmlx(complex double cv)
{
    printf("(%.2f, %.2fi)", creal(cv), cimag(cv));
    return;
}

```

如果使用 C++, 会发现 C++ 的 `complex` 头文件提供一种基于类的方式处理复数, 这与 C 的 `complex.h` 头文件使用的方法不同。

## B.9 参考资料 IX: C 和 C++ 的区别

在很大程度上, C++ 是 C 的超集, 这意味着一个有效的 C 程序也是一个有效的 C++ 程序。C 和 C++ 的主要区别是, C++ 支持许多附加特性。但是, C++ 中有许多规则与 C 稍有不同。这些不同使得 C 程序作为 C++ 程序编译时可能以不同的方式运行或根本不能运行。本节着重讨论这些区别。如果使用 C++ 的编译器编译 C 程序, 就知道这些不同之处。虽然 C 和 C++ 的区别对本书的示例影响很小, 但如果把 C 代码作为 C++ 程序编译的话, 会导致产生错误的消息。

C99 标准的发布使得问题更加复杂, 因为有些情况下使得 C 更接近 C++。例如, C99 标准允许在代码中的任意处进行声明, 而且可以识别 // 注释指示符。在其他方面, C99 使其与 C++ 的差异变大。例如, 新增了变长数组和关键字 `restrict`。C11 缩小了与 C++ 的差异。例如, 引进了 `char16_t` 类型, 新增了关键字 `_Alignas`, 新增了 `alignas` 宏与 C++ 的关键字匹配。C11 仍处于起步阶段, 许多编译器开发商甚至都没有完全支持 C99。我们要了解 C90、C99、C11 之间的区别, 还要了解 C++11 与这些标准之间的区别, 以及每个标准与 C 标准之间的区别。这部分主要讨论 C99、C11 和 C++ 之间的区别。当然, C++ 也正在发展, 因此, C 和 C++ 的异同也在不断变化。

### B.9.1 函数原型

在 C++ 中, 函数原型必不可少, 但是在 C 中是可选的。这一区别在声明一个函数时让函数名后面的圆括号为空, 就可以看出来。在 C 中, 空圆括号说明这是前置原型, 但是在 C++ 中则说明该函数没有参数。也就是说, 在 C++ 中, `int slice();` 和 `int slice(void);` 相同。例如, 下面旧风格的代码在 C 中可

以接受，但是在 C++ 中会产生错误：

```
int slice();
int main()
{
    ...
    slice(20, 50);
    ...
}
int slice(int a, int b)
{
    ...
}
```

在 C 中，编译器假定用户使用旧风格声明函数。在 C++ 中，编译器假定 `slice()` 与 `slice(void)` 相同，且未声明 `slice(int, int)` 函数。

另外，C++ 允许用户声明多个同名函数，只要它们的参数列表不同即可。

## B.9.2 char 常量

C 把 `char` 常量视为 `int` 类型，而 C++ 将其视为 `char` 类型。例如，考虑下面的语句：

```
char ch = 'A';
```

在 C 中，常量 '`A`' 被储存在 `int` 大小的内存块中，更精确地说，字符编码被储存为一个 `int` 类型的值。相同的数值也储存在变量 `ch` 中，但是在 `ch` 中该值只占内存的 1 字节。

在 C++ 中，'`A`' 和 `ch` 都占用 1 字节。它们的区别不会影响本书中的示例。但是，有些 C 程序利用 `char` 常量被视为 `int` 类型这一特性，用字符来表示整数值。例如，如果一个系统中的 `int` 是 4 字节，就可以这样编写 C 代码：

```
int x = 'ABCD'; /* 对于 int 是 4 字节的系统，该语句出现在 C 程序中没问题，但是出现在 C++ 程序中会出错 */
```

'`ABCD`' 表示一个 4 字节的 `int` 类型值，其中第 1 个字节储存 `A` 的字符编码，第 2 个字节储存 `B` 的字符编码，以此类推。注意，'`ABCD`' 和 "ABCD" 不同。前者只是书写 `int` 类型值的一种方式，而后者是一个字符串，它对应一个 5 字节内存块的地址。

考虑下面的代码：

```
int x = 'ABCD';
char c = 'ABCD';
printf("%d %d %c %c\n", x, 'ABCD', c, 'ABCD');
```

在我们的系统中，得到的输出如下：

```
1094861636 1094861636 D D
```

该例说明，如果把 '`ABCD`' 视为 `int` 类型，它是一个 4 字节的整数值。但是，如果将其视为 `char` 类型，程序只使用最后一个字节。在我们的系统中，尝试用 `%s` 转换说明打印 '`ABCD`' 会导致程序奔溃，因为 '`ABCD`' 的数值（1094861636）已超出该类型可表示的范围。

可以这样使用的原因是 C 提供了一种方法可单独设置 `int` 类型中的每个字节，因为每个字符都对应一个字节。但是，由于要依赖特定的字符编码，所以更好的方法是使用十六进制的整型常量，因为每两位十六进制数对应一个字节。第 15 章详细介绍过相关内容（C 的早期版本不提供十六进制记法，这也许是多字符常量技术首先得到发展的原因）。

## B.9.3 const 限定符

在 C 中，全局的 `const` 具有外部链接，但是在 C++ 中，具有内部链接。也就是说，下面 C++ 的声明：

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

相当于下面 C 中的声明:

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

假设这两条声明都在所有函数的外部。C++规则的意图是为了在头文件更加方便地使用 `const`。如果 `const` 变量是内部链接，每个包含该头文件的文件都会获得一份 `const` 变量的备份。如果 `const` 变量是外部链接，就必须在一个文件中进行定义式声明，然后在其他文件中使用关键字 `extern` 进行引用式声明。

顺带一提，C++可以使用关键字 `extern` 使一个 `const` 值具有外部链接。所以两种语言都可以创建内部链接和外部链接的 `const` 变量。它们的区别在于默认使用哪种链接。

另外，在 C++ 中，可以用 `const` 来声明普通数组的大小:

```
const int ARSIZE = 100;
double loons[ARSIZE]; /* 在 C++ 中，与 double loons[100]; 相同 */
```

当然，也可以在 C99 中使用相同的声明，不过这样的声明会创建一个变长数组。

在 C++ 中，可以使用 `const` 值来初始化其他 `const` 变量，但是在 C 中不能这样做:

```
const double RATE = 0.06;           // C++ 和 C 都可以
const double STEP = 24.5;           // C++ 和 C 都可以
const double LEVEL = RATE * STEP;   // C++ 可以，C 不可以
```

## B.9.4 结构和联合

声明一个有标记的结构或联合后，就可以在 C++ 中使用这个标记作为类型名:

```
struct duo
{
    int a;
    int b;
};

struct duo m; /* C 和 C++ 都可以 */
duo n; /* C 不可以，C++ 可以 */
```

结果是结构名会与变量名冲突。例如，下面的程序可作为 C 程序编译，但是作为 C++ 程序编译时会失败。因为 C++ 把 `printf()` 语句中的 `duo` 解释成结构类型而不是外部变量:

```
#include <stdio.h>
float duo = 100.3;
int main(void)
{
    struct duo { int a; int b; };
    struct duo y = { 2, 4 };
    printf ("%f\n", duo); /* 在 C 中没问题，但是在 C++ 不行 */
    return 0;
}
```

在 C 和 C++ 中，都可以在一个结构的内部声明另一个结构:

```
struct box
{
    struct point { int x; int y; } upperleft;
    struct point lowerright;
};
```

在 C 中，随后可以使用任意使用这些结构，但是在 C++ 中使用嵌套结构时要使用一个特殊的符号:

```
struct box ad;           /* C 和 C++ 都可以 */
struct point dot;        /* C 可以, C++ 不行 */
box::point dot;          /* C 不行, C++ 可以 */
```

## B.9.5 枚举

C++ 使用枚举比 C 严格。特别是，只能把 enum 常量赋给 enum 变量，然后把变量与其他值作比较。不经过显式强制类型转换，不能把 int 类型值赋给 enum 变量，而且也不能递增一个 enum 变量。下面的代码说明了这些问题：

```
enum sample {sage, thyme, salt, pepper};
enum sample season;
season = sage;           /* C 和 C++ 都可以 */
season = 2;               /* 在 C 中会发出警告，在 C++ 中是一个错误 */
season = (enum sample) 3; /* C 和 C++ 都可以 */
season++;                /* C 可以，在 C++ 中是一个错误 */
```

另外，在 C++ 中，不使用关键字 enum 也可以声明枚举变量：

```
enum sample {sage, thyme, salt, pepper};
sample season;           /* C++ 可以，在 C 中不可以 */
```

与结构和联合的情况类似，如果一个变量和 enum 类型的同名会导致名称冲突。

## B.9.6 指向 void 的指针

C++ 可以把任意类型的指针赋给指向 void 的指针，这点与 C 相同。但是不同的是，只有使用显式强制类型转换才能把指向 void 的指针赋给其他类型的指针。下面的代码说明了这一点：

```
int ar[5] = {4, 5, 6, 7, 8};
int * pi;
void * pv;
pv = ar;                 /* C 和 C++ 都可以 */
pi = pv;                 /* C 可以，C++ 不可以 */
pi = (int *) pv;          /* C 和 C++ 都可以 */
```

C++ 与 C 的另一个区别是，C++ 可以把派生类对象的地址赋给基类指针，但是在 C 中没有这里涉及的特性。

## B.9.7 布尔类型

在 C++ 中，布尔类型是 bool，而且 true 和 false 都是关键字。在 C 中，布尔类型是 \_Bool，但是要包含 stdbool.h 头文件才可以使用 bool、true 和 false。

## B.9.8 可选拼写

在 C++ 中，可以用 or 来代替 ||，还有一些其他的可选拼写，它们都是关键字。在 C99 和 C11 中，这些可选拼写都被定义为宏，要包含 iso646.h 才能使用它们。

## B.9.9 宽字符支持

在 C++ 中，wchar\_t 是内置类型，而且 wchar\_t 是关键字。在 C99 和 C11 中，wchar\_t 类型被定义在多个头文件中 (stddef.h、stdlib.h、wchar.h、wctype.h)。与此类似，char16\_t 和 char32\_t

都是 C++11 的关键字，但是在 C11 中它们都定义在 uchar.h 头文件中。

C++通过 iostream 头文件提供宽字符 I/O 支持 (wchar\_t、char16\_t 和 char32\_t)，而 C99 通过 wchar.h 头文件提供一种完全不同的 I/O 支持包。

## B.9.10 复数类型

C++在 complex 头文件中提供一个复数类来支持复数类型。C 有内置的复数类型，并通过 complex.h 头文件来支持。这两种方法区别很大，不兼容。C 更关心数值计算社区提出的需求。

## B.9.11 内联函数

C99 支持了 C++的内联函数特性。但是，C99 的实现更加灵活。在 C++中，内联函数默认是内部链接。在 C++中，如果一个内联函数多次出现在多个文件中，该函数的定义必须相同，而且要使用相同的语言记号。例如，不允许在一个文件的定义中使用 int 类型形参，而在另一个文件的定义中使用 int32\_t 类型形参。即使用 typedef 把 int32\_t 定义为 int 也不能这样做。但是在 C 中可以这样做。另外，在第 15 章中介绍过，C 允许混合使用内联定义和外部定义，而 C++不允许。

## B.9.12 C++11 中没有的 C99/C11 特性

虽然在过去 C 或多或少可以看作是 C++的子集，但是 C99 标准增加了一些 C++没有的新特性。下面列出了一些只有 C99/C11 中才有的特性：

- 指定初始化器；
- 复合初始化器 (Compound initializer)；
- 受限指针 (*Restricted pointer*) (即，restrict 指针)；
- 变长数组；
- 伸缩型数组成员；
- 带可变数量参数的宏。

### 注意

以上所列只是在特定时期内的情况，随着时间的推移和 C、C++的不断发展，列表中的项会有所增减。例如，C++14 新增的一个特性就与 C99 的变长数组类似。

# C Primer Plus

## (第6版) 中文版

本书是一本经过仔细测试、精心设计的完整C语言教程，它涵盖了C语言编程中的核心内容。本书作为计算机科学的经典著作，讲解了包含结构化代码和自顶向下设计在内的程序设计原则。

与以前的版本一样，作者的目标仍旧是为读者提供一本入门型、条理清晰、见解深刻的C语言教程。作者把基础的编程概念与C语言的细节很好地融合在一起，并通过大量短小精悍的示例同时演示一两个概念，通过学以致用的方式鼓励读者掌握新的主题。

每章末尾的复习题和编程练习题进一步强化了最重要的信息，有助于读者消化那些难以理解的概念。本书采用了友好、易于使用的编排方式，不仅适合打算认真学习C语言编程的学生阅读，也适合那些精通其他编程语言，但希望更好地掌握C语言这门核心语言的开发人员阅读。

本书在之前版本的基础之上进行了全新升级，它涵盖了C语言最新的进展以及C11标准的详细内容。本书还提供了大量深度与广度齐备的教学技术和工具，来提高你的学习。



- 详细完整地讨论了C语言的基础特性和附加特性；
- 清晰解释了使用C语言不同部分的时机，以及原因；
- 通过简洁、简单的示例加强读者的动手练习，以帮助一次理解一两个概念；
- 囊括了数百个实用的代码示例；
- 每章末尾的复习题和编程练习可以检测你的理解情况；
- 涵盖了C泛型编程，以提供最大的灵活性。



读者可通过<http://www.epubit.com.cn/book/details/1848>下载该书的源代码。

异步社区 [www.epubit.com.cn](http://www.epubit.com.cn)  
新浪微博 @人邮异步社区  
投稿/反馈邮箱 contact@epubit.com.cn



ISBN 978-7-115-39059-2



ISBN 978-7-115-39059-2

定价:89.00 元

封面设计: 董志桢

分类建议: 计算机 / 程序设计 / C  
人民邮电出版社网址: [www.ptpress.com.cn](http://www.ptpress.com.cn)