第九章 结构和其他数据机制

第六和第七章分别介绍了数组和指针机制，本章将介绍 C 语言的其他数据定义与描述机制，包括结构（struct）、联合（union）、枚举（enum）等。写处理复杂数据的程序往往需要定义复杂的数据类型和结构，这时常常需要使用这些机制。在计算机领域后续的“数据结构”等课程中也要大量使用这些机制。

本章将对这些数据机制的概念、意义和用途，以及使用它们的基本技术做一些介绍，举一些程序实例。后续课程中将会有更多应用这些机制的实例，读者也能在各种较深入的计算机书籍材料中看到大量的有关例子。

9.1 结构（struct）

客观世界里需要用计算机处理的数据千变万化，它们常常不是互相独立的，而是集合成组，若干数据元素形成一个逻辑整体，元素间存在着紧密的联系。这些情况使人不得不考虑复杂数据的处理问题。当逻辑数据体的各部分具有共同性质时，可以用数组作为组合手段（元素“类型”相同）。但情况未必如此，也存在许多组合体，其中各数据成分的类型并不统一，公民身份证的数据就是一个典型例子。一个身份证的数据成分包括姓名、性别、民族、出生日期、住址、身份证号码、发证日期、有效期限和发证单位，还有一张照片等等。显然，这样一组信息应看作一个逻辑整体，因为它们共同描述了一个居民的情况。但是，这些信息中有字符串，有数值，可能还有图像信息（照片）等，因此不适合用数组表示。这类情况在实际应用中非常普遍，这就要求程序语言提供相应的数据描述机制。

针对这类情况，许多高级语言提供了另一种数据机制，专门用于把多个类型可能不同的数据对象集合起来。C 语言将这种机制称为结构（structure）。结构是由若干（可不同类型的）数据项组合而成的复合数据对象，这些数据项称为结构的成分或成员。一个（或一类）结构中的每个成员都给定了一个名字，通过成员名实现对结构成员的访问。

9.1.1 结构说明与变量定义

要说明一个结构，就需要描述它的各个成员的情况，包括每个成员的类型及名字。结构描述用关键字 struct 引导，结构说明的最基本形式是：

```
struct { 成员说明序列 };
```

写在 struct 后面的部分是结构的成员说明表，它描述了被定义结构的组成，其中的每个成员说明在形式上与变量定义一样，可以用一个类型说明一个成员，也可以用一个类型说明几个（类型相同的）成员。下面是两个简单的结构说明：

```
struct {
    int n;
    double x, y;
};

struct {
    int n;
    double data[100];
};
```

这里的第一个结构包含三个结构成员：第一个是名字为 n 的整型成员；后面两个成员都为双精度类型，它们的名字分别是 x 和 y。第二个结构里包含两个成员，第一个是整型成员 n，第二个成员 data 是一个双精度数组。

在一个结构里可以有任意多个成员，这些成员可以是任何可用类型的，包括各种基本类型的成员，指针成员或者数组成员。成员本身也可以又是结构，也可以是后面介绍的联合等等。用 typedef 定义的类型名也可以用在这里。一个结构里的成员名字不能相互冲突。不同的结构里完全可以包含名字相同的成员，它们是相互无关的。在上面例子里就可以看到这方面的情况，实际程序里也常常可以看到这种情况。

定义结构变量

像上面那样单独写出的结构说明仅描述了一个结构的形式，在程序里写出这种说明并没有实际价值。如果要在程序里使用结构，那么就需要定义结构变量（以结构为值的变量）。

把结构说明当作类型描述，在这个说明之后给出希望定义的变量的名字（标识符），就构成了一个结构变量定义。这种描述使所定义变量成为具有相应结构的变量。例如，下面描述的定义了两个结构变量 st1 和 st2：

```
struct {
    int n;
    double x, y;
} st1, st2;
```

有关结构变量的使用方式在下面讨论。

结构标志及其使用

实际程序里的结构描述的形式可以很复杂。为避免在程序里反复写复杂的结构描述，减少编程者的负担，也减少多次描述出现不一致的可能性，C 语言提供了一种称为*结构标志*的机制。在说明结构时可以在 struct 后面加一个*标志*（标识符）。在这样做之后，程序里的其他地方就可以把这个标志放在关键字 struct 后面，用于代表相应的结构说明。例如，下面程序片段说明了三个结构，三个说明里都给出了结构标志：

```
struct point {
    double x, y;
};

struct circle {
    struct point center;
    double radius;
};

struct rectangle {
    struct point lu;
    struct point rd;
};
```

第一个结构的标志是 point，该结构有两个 double 成员。第二个结构的标志是 circle，它包含两个成员，其第一个成员是一个 point 结构（结构成员也可以是结构），第二个成员是一个双精度数。在结构 circle 的描述中还使用了刚定义的 point 的结构标志，这样就不必将整个 point 结构的描述重抄在这里了。上面定义的第三个结构是包含两个 point 结构成员的 rectangle 结构。

有了结构标志之后，定义结构变量就比较方便了。下面定义了几个结构变量：

```
struct point pt1, pt2;
struct circle circ1, circ2;
struct rectangle rec;
```

在程序里，struct point 这类描述形式也被当作类型看待，可以用在所有需要写类型的地方。例如，可以将它们用于函数定义和原型描述，定义指向结构的指针，做类型转换，计算结构类型或者结构变量大小。下面是两个与上面结构有关的函数原型说明：

```
double distance(stuct point p1, struct point p2);
int inrect(struct point p, struct rectangle rec);
```

其中的函数 distance 包含两个 struct point 结构的参数，返回 double 值；inrect 包含一个 struct point 结构参数和一个 struct rectangle 结构参数。

下面例子表明了如何通过动态分配建立存放结构数据对象的存储块：

```
struct circle *pp1, *pp2;
pp1 = (stuct circle *)malloc(sizeof(struct circle));
pp2 = pp1;
... ...
```

定义结构类型

如果需要在程序里反复使用某种结构，那么最好将它定义为一个结构类型。定义结构类型同样通过 typedef 完成。例如，下面是对应前面描述的结构的三个类型定义：

```
typedef struct {
    double x, y;
} POINT;

typedef struct {
    POINT center;
    double radius;
} CIRCLE;

typedef struct {
    POINT lu;
    POINT rd;
} RECTANGLE;
```

定义结构类型的方式与定义其他类型一样，也是将被定义类型名放在结构描述之后（放在原来写变量名的地方）。上面定义的三个类型的名字分别是 POINT、CIRCLE 和 RECTANGLE，它们可以用在任何需要用类型名的地方。

定义了结构类型之后，在程序里各处都可以直接用这个名字代表所定义类型，使用起来更加方便，写出的程序也更加简短清晰。这种方式很值得提倡。

下面是一个描述居民身份证信息的结构类型定义：

```
typedef struct {
    char name[20];
    int sex;
    char nationality[10];
    int born_in[3];
    char address[50];
    int date_of_issue[3];
    int valid_years;
    char issued_by[30];
    char id_number[18];
    char photo[100][64];
} IDCARD;
```

显然，在开发实际身份证处理程序时，为了定义好表示身份证的结构，有许多问题需要考虑和选择。例如：表示名字的字符串需要包含多少个字符？如果只考虑汉族人的名字，那么用 8 个字符可以表示 4 个汉字，应当够用了，做成字符串形式只需要 9 个字符。但中国是一个多民族国家，名字的写法千差万别，20 个字符也未必够用。有关地址的情况也类似。此外还要考虑用什么方式来表示身份证里的性别、民族、日期、照片等。

这里只是把身份证作为结构类型定义的例子。如果我们所面对的问题是实现一个真正的身份证信息管理系统，那么这许多问题都需要仔细研究和选择。这方面问题在软件设计过程中是非常重要的，一般被称作*数据表示*问题，要解决的就是用什么样的计算机内部形式（例如程序语言提供的类型、结构等）表示实际应用领域中的数据。

虽然在程序中所有与结构有关的描述都可以直接采用结构标志的方式写出，但从程序实践看，如果某个结构在程序里用得较多，最好还是定义为类型，用一个特殊的名字表示它。这样不但可以简化程序书写，也更加有利于程序的修改和维护。

上面定义的各结构类型将在下面作为例子使用，以后就不再重复说明了。

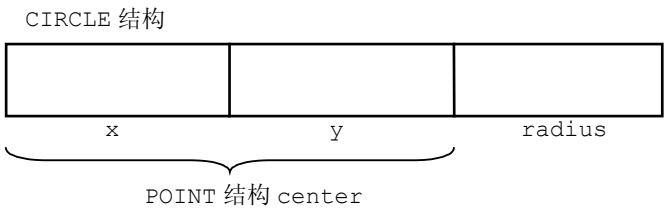

图 9.1　结构对象的存储形式

无效的结构描述

结构成员可以是各种基本数值类型，指针类型，也可以是数组、结构等复合类型。一个结构里可以有多个成员。对结构描述的一个重要限制是结构的成员不能是正在描述的这个结构本身。例如下面就是一个非法结构描述：

```
struct invalid {
    int n;
    struct invalid iv;
};
```

不允许这类结构的道理很简单。如果一个结构里包含了它自身，那么就会引起一种结构上的无穷嵌套，这个结构的基本成员（如果有的话）将会是无穷的。这是一种完全不合理的局面，也不可能在计算机里实现。

结构的实现

存储一个结构数据对象（例如结构变量）就需要存储其中的各个成员。编译系统将为每个结构变量分配一块足够大的存储区，把它的各个成员顺序存于其中。例如，类型为 CIRCLE 的对象的存储将具有图 9.1 所示的形式。其中 POINT 类型成员 center 存储在前，center 的两个成员 x 和 y 在 center 所占据的区域中顺序存放。center 之后存 CIRCLE 的另一个成员，double 类型的 radius。

结构成员类型可以不同，有时这会给实现带来一点麻烦。例如，下面描述的结构：

```
struct exam {
    char aa;
    int nn;
    double xx;
} ss;
```

实现时就会出现一种特殊情况。引起问题的原因是这个结构包含几个类型不同的成员，而这几个成员的大小又不同。成员 aa 是字符类型，一般需要一个字节；nn 的大小依赖于系统，如果是 16 位就需要 2 个字节；而双精度成员 xx 一般需要 8 个字节。

为了程序执行效率，计算机硬件通常规定了各种基本类型数据的摆放方式。例如，通常要求两字节表示的整数从偶数地址的单元开始存放；对于需要 8 个字节表示的双精度数，可能要求从 4 的倍数（或者 8 的倍数）地址的单元开始存放*。此外，系统对整个结构对象的存放也可能有起始位置要求，这样，存储 exam 结构的区域中就可能出现空位。这类问题一般统称为对齐问题。由此引起的问题包括：（1）结构里的后续成员未必紧接着前一个成员存放，成员间可能出现空位；（2）结构的大小未必是其各个成员的大小之和。

这一情况告诉我们，在程序里不能想当然地去确定成员位置，不能认为一个成员之后就是另一个成员。另一方面，如果在程序中需要用结构的大小（例如做动态存储分配），就应当用 sizeof 运算符计算，而不要自己主观认定。

9.1.2 结构变量的初始化和使用

与简单类型变量和数组一样，结构变量也可以在定义时直接初始化。为结构提供初始值的形式与数组一样，例如，下面结构变量定义中就包含了几个初始化描述：

* 不同机器、不同系统在这些方面的性质可能不同，但带来的问题是类似的。

```
POINT pt1 = {2.34, 3.28}, pt2;
CIRCLE circ1 = {{3.5, 2.07}, 1.25},
       circ2 = {12.35, 10.6, 2.56};
```

初始化描述中的各个值将顺序地提供给结构变量的各个基本成员，初值表达式必须是可静态求值的表达式。从两个 CIRCLE 变量定义的例子可以看到，对于嵌套结构，初始化表示中可以加嵌套括号，也可以不加。初始化描述中数据项的个数不得多于结构变量所需，如果提供的数据项不够，其余结构成员自动用 0 初始化。这些都与数组的规定一样。无论是全局结构变量，还是函数内的局部结构变量，都可以用这种形式进行初始化。

如果在定义结构变量时未提供初始值，系统的处理方式也与对其他变量一样。对于全局变量和静态局部变量，结构的所有成员用 0 初始化；对自动变量不进行初始化，各成员将处在没有明确定义的状态。应特别指出，结构的初始值描述形式只能用在变量定义的初始化时，不能用在程序中其他地方（例如语句里）。这一点也与对数组的规定相同。

对结构变量的操作主要是整体赋值和结构成员访问。

整体赋值

结构变量可以整体赋值，显然，赋值时只能用同样类型的“结构值”。这种赋值的效果就是结构中各个成员的分别赋值。例如，我们可以用一个已经有值的结构给另一个同样类型的结构复制。此时位于赋值号右边的结构的各成员值将分别复制到赋值运算符左边的结构中对应的成员里。在有了上面的定义之后，就可以写出下面赋值：

```
pt2 = pt1;
```

这个赋值之后，变量 pt2 各成员的值将与 pt1 对应成员的值完全一样。

成员访问

访问结构成员的操作用圆点运算符（.）描述，这个运算符也具有最高的优先级，并采用自左向右的结合方式。下面是使用这个运算符的几个例子：

```
pt2.y = pt1.y + 2.4;
circ1.center.x = 2.07;
circ1.center.y = pt1.y;
```

如果需要将结构 circ2 所表示的圆做一个平移，可以写：

```
circ2.center.x += 2.8;
circ2.center.y += 0.24;
```

访问结构变量的成员相当于访问一个具有相应类型的变量，对这个成员能做什么操作，完全由该成员的类型决定。

此外，与其他数据对象一样，程序里也可以用&取得结构变量的地址。C 语言规定不能对结构做相等与不等比较，也不能做其他运算。

简单程序实例

下面是一个展示结构说明，结构变量定义和使用的简单程序例子。它简单地计算出的两个点之间的距离并输出：

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

typedef struct {
    double x, y;
} POINT;
typedef struct {
    POINT center;
    double radius;
} CIRCLE;

int main () {
    POINT pt1 = {2.34, 3.28}, pt2;
    CIRCLE circ1 = {{3.5, 2.07}, 1.25};
```

```
    double x, y, dist;

    pt2.x = 8.0;
    pt2.y = circ1.center.y + 3.44;
    x = pt2.x - pt1.x;
    y = pt2.y - pt1.y;
    printf("distance: %f\n", sqrt(x*x + y*y));
    return 0;
}
```

9.1.3 结构、数组与指针

结构里可以包含数组成员，前面已有这样的例子，身份证结构里就有这种情况。另一方面，也可以定义以结构作为元素的数组。下面是一个熟悉的例子：

例：用结构的数组重新构造第五章中统计 C 程序中各关键字出现次数的程序。

前面章节里已经讨论过两种实现方式，第一种方式中用了一个两维字符数组和一个计数器数组；第二种方式中用了一个字符指针数组和一个计数器数组。在这个程序里，每个关键字都有一个对应的计数器，关键字与计数器间的联系更密切一些。但在前面的两种实现里，这两种东西被割裂开，分别存放在不同的数组里，它们之间的关系没有显式表现出来。这就是说，另一种可能更合理的方式是采用一种结构来表示与一个关键字有关的所有信息。为此可以定义下面的结构类型：

```
typedef struct {
    char * key;
    int count;
} KEYC;
```

这个结构类型在逻辑上也是意义的，可以将它看作一种专门的“关键字计数器”类型。这种类型的变量表示的就是与一个关键字有关的所有信息。

以这种类型为基础改写关键字统计程序时，程序里需要的主要数据结构就是一个 KEYC 类型的数组。下面同样在定义时对它做初始化：

```
KEYC keytable[32] = {
    "auto",    0,  "break",    0,  "case",     0,
    "char",    0,  "const",    0,  "continue",0,
    "defualt", 0,  "do",       0,  "double",   0,
    "else",    0,  "enum",     0,  "extern",   0,
    "float",   0,  "for",      0,  "goto",     0,
    "if",      0,  "int",      0,  "long",     0,
    "register",0,  "return",   0,  "short",    0,
    "signed",  0,  "sizeof",   0,  "static",   0,
    "struct",  0,  "switch",   0,  "typedef",  0,
    "union",   0,  "unsigned",0,  "void",     0,
    "volatile",0,  "while",    0
};
```

如前所述，在这种初始化表示中也可以写嵌套的括号，区分对各个成员的初始化描述。

采用这种数据结构，程序的主循环可以写为：

```
int i;
...
while (getline(s, ...) > 0)
    for (i = 0; I < 32; ++i)
        if (strcmp(s, keytable[i].key) == 0) {
            keytable[i].count++;
            break;
        }
```

我们也可以采用指针方式描述这一循环，可将上述循环改写为下面形式：

```
KEYC *p;
...
while (getline(s, ...) > 0)
    for (p = keytable; p < keytable + 32; ++p)
        if (strcmp(s, (*p).key) == 0) {
```

```
                (*p).count++;
                break;
            }
```

请注意程序里 (*p).key 的写法，这表示先由 p 间接得到被指结构，而后取得这个结构的相应成员 key。由于运算符优先级的规定（取结构成员的运算符 . 优先级高），在这一描述中必须写括号。不写括号的描述 *p.key 是错误的，它实际上表示的是 *(p.key)。这显然不是我们希望的东西，编译时也会报错。

在 C 程序里，指向结构的指针有的应用非常广，人们常要由这种指针出发去访问结构成员（就像上面程序段中所做的那样）。为了描述方便，C 语言为这种操作专门提供了一个运算符号“->”，p->key 就相当于(*p).key。运算符->也具有最高优先级（与圆点运算符、函数调用()及数组元素访问[]一样），它也遵循从左向右的结合方式。

到现在为止，我们已经介绍完了 C 语言的所有运算符号，附录 A 列出了 C 语言里所有的运算符符号，它们的意义、优先级和结合方式方面的规定。请读者自己看看附录 A 的表格，检查一下自己对 C 中各种运算符的认识，必要时重新查阅本书正文中的介绍。

9.1.4 字段*

C 语言的结构还提供了一种定义字段的机制，使人在需要时能把几个结构成员压缩到一个基本数据类型成员里存放，这可以看作是一种数据压缩表示方式。

在讨论字段问题之前先要理解“字”的概念。每种计算机都有一种硬件确定的基本处理单位，通常也是内存的基本数据存储单位，这就是 CPU 的机器字（简称字）。常见计算机中一个字的长度（字中所包含的二进制位数）有 16 位、32 位、64 位等，大于一个字的数据元素需用连续的几个字表示。在一般 C 系统里，int 类型的数据用一个字表示。

在另一方面，实际应用中确实也有些简单数据，其信息量很小，完全没有必要用一个字表示。以表示身份证的结构为例。其中的性别成员仅有两种不同值，用一个二进制位就够了。中国的不同民族有五十几个，每个民族给定一个编码，6 位二进制数足以表示这一成员。表示日期的成员也可以压缩，不需要将年月日中的每项用一个 int，如此等等。

读者可能说，这样节省有什么意义呢？在实现真正的应用系统时，有效利用存储是需要特别关注的大问题。以身份证表示为例，虽然对于一个身份证结构变量，采用任何压缩方式也节省不了多少存储，但是对全国身份证的管理系统而言，节省下来的存储将非常可观。系统的数据容量减少了，不仅节约了硬件存储设备，而且各种处理操作也可能节省时间。

此外，外部硬件设备的操作通常也通过命令字的方式表示，这个字里不同的二进制位或者位段表示对于硬件各部分状态的设置或指令。如果某个 C 程序需要与硬件打交道，那么就需要针对一个字里的各个部分进行操作。前面已经讨论了按位运算，C 语言里解决这类问题的另一种机制是字段。

结构成员可以定义为字段，其定义方式与一般的结构成员类似，但这里有两个附加规定：第一，这种成员的类型只能是 int 或 unsigned int；第二，在每个字段说明的最后需要给出该字段的二进制位数，任一字段的位数不能超过一个字（一个 int）的长度。也就是说，在结构说明里带有位数描述的成员就是字段成员。

下面是一个包含三个字段成员的记录的描述：

```
struct pack {
    unsigned a:2;
    unsigned b:8;
    unsigned c:6;
} pk1, pk2;
```

这说明了一种包含字段成员的结构 struct pack，还定义了两个这种结构的变量 pk1 和 pk2。结构变量 pk1 或者 pk2 的三个成员将总共占用 16 位存储，其中 a 占用 2 位，b 占用

8 位，c 占用 6 位。

使用字段成员的形式与使用普通结构成员一样，下面是两个使用字段的语句：

```
n = pk1.a;
pk2.c = 13;
```

虽然上述结构的各个成员只占用了很少几个位，我们还是将它们看作 unsigned 性质，采用类似的计算规则。如果结构里有连续的多个字段，其排列就可能跨越字的边界。对于这种情况，不同的编译系统可能采用不同的安排方式。C 语言对这一点没做具体规定。

人们发现，采用字段压缩方式，通常会使访问结构成员的操作效率下降很多。因此，除非必要，现在人们不太提倡在程序里使用字段成员。此外，如果确实需要将许多信息项压缩存储在一个字里，我们还可以采用前面介绍的二进制位运算的方式处理。采用位运算的方式比较灵活，可以根据需要自己设计，在效率方面常常能优于采用字段。

9.2 结构与函数

由于结构可以整体赋值，所以可以将结构作为值参数传递给函数，也可以定义返回结构值的函数。这样，要函数处理存储在结构中的数据，我们至少有三种不同方法：

1. 个别地将结构成员的值传递给函数处理。
2. 将整个结构作为参数值传递给函数，一般将这种参数称作*结构参数*。
3. 将结构的地址传给函数，也就是说传递指向结构的指针值。这称为*结构指针参数*。

后两种方式都是把结构作为整体来看待和处理，但正如针对其他参数的值传递和指针传递一样，这两种参数的作用方式和效果不同。

如果函数 f 有一个结构参数 r，在用结构变量 s 作为实参调用时（假定 s 的类型匹配），s 的值将首先赋给 r，而后进入函数内部的处理。无论在函数 f 内部对 r 做什么操作，都不会改变实际参数 s。作为值的结构参数具有清晰的语义，是一种很常用的方式。

在另一方面，如果函数 g 有一个结构指针参数 p，调用 g 时将 s 的地址传给 p（假定类型匹配），函数体里就可以通过 p 对 s 做任何操作，包括赋值，修改其成员等。采用指针的另一优点是可以避免复制整个结构。如果被处理的结构很大，多次复制将耗费很多时间，也可以考虑用指针方式传递。

9.2.1 处理结构的函数

通过函数处理结构时，不同的传递方法各有其特点，有其适用与方便之处，下面通过一些简单的例子来说明这些问题。

例：由参数值出发构造一个 POINT 类型的结构值，写出的函数定义如下：

```
POINT mkpoint1(double x, double y) {
    POINT temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

函数 mkpoint1 返回一个 POINT 结构类型的值，这个值可以赋给任何 POINT 结构类型的变量。这种函数的特点是从结构成员的值出发构造出结构值，可称作“结构值的构造函数”，在复杂的程序里很常见。下面是使用该函数的两个例子：

```
pt1 = mkpoint1(3.825, 20.7);
pt2 = mkpoint1(pt1.x, 0.0);
```

注意，函数 mkpoint1 里的 temp 是局部的结构变量，它将随着 mkpoint1 结束而撤消。但在 mkpoint1 结束时 temp 的值被作为函数值返回，与变量的撤消无关。我们可以把这

种情况与简单类型局部变量的值作为返回值的情况做个对比，除了结构变量可能占用较大存储空间外，两者在其他方面的情况完全一样。当然，前面提出的问题在这里也出现了，对于很大的结构，返回结构值就要做较多的复制工作。

例：由参数值出发构造一个 CIRCLE 类型的结构值，函数的定义如下：

```
CIRCLE mkcircle(POINT c, double r) {
    CIRCLE temp;
    temp.center = c;
    temp.radius = r;
    return temp;
}
```

这里不但以结构作为返回值，函数还有一个结构参数。下面是使用 mkcircle 的例子：

```
circ1 = mkcircle(pt1, 5.254);
circ2 = mkcircle(circ1.center, 11.7);
circ3 = mkcircle(mkpoint1(2.05, 3.7), 3.245);
```

其中第三个例子比较特殊。由于 mkpoint 返回 POINT 类型的结构值，把这个值传递给函数 mkcircle 的 POINT 类型的参数是合理的。这种函数嵌套调用也是程序设计的一种技术，当然它要求外层函数的参数与内层函数的返回值类型匹配。

现在考虑定义一个计算两个点之间距离的函数，它也采用普通的结构参数：

```
double distance(POINT p1, POINT p2) {
    double x = p1.x - p2.x, y = p1.y - p2.y;
    return sqrt(x*x + y*y);
}
```

上面这些例子采用结构参数或者结构返回值。函数调用时，实参结构的值被整个赋给函数内的形参；而作为函数计算结果的结构值建立副本，在函数退出之后再赋给指定变量。这种定义方式的优点是语义非常清楚，函数内外的计算互不干扰，即使是上面有个例子里那样写了函数嵌套调用，计算的结果也很容易弄明白。这种方法的一个缺点是常常需要反复复制整个结构，可能增大计算中的时间开销。

例：写一个打印输出身份证结构中身份证号码和姓名的函数。函数的一种写法是：

```
void prtIDCard0(IDCARD ic) {
    printf("%s\n", ic.id_number);
    printf("%s\n\n", ic.name);
}
```

这个函数可以完成工作，但每次打印一个身份证记录，都需要复制整个结构的值，而函数里实际需要的信息非常少，大部分复制工作完全没有价值。这个例子说明，编程不当可能降低程序效率。对这个实例就应该考虑传递结构指针。如前所述，传递结构指针至少有两种用途，一是使函数能够改变执行环境中的结构；另一个作用就是避免无谓的结构复制。

下面是传递结构指针的同样函数，其中用了从指针访问被指结构成员的运算符：

```
void prtIDCard(IDCARD *icp) {
    printf("%s\n", icp->id_number);
    printf("%s\n\n", icp->name);
}
```

用这个新函数打印一个身份证的有关信息，调用时只需复制一个指针，这当然有效得多，也更合理。如果 idc 是一个 IDCARD 结构变量，打印其中信息用下面调用形式：

```
prtIDCard(&idc);
```

下面是函数使用的另一个例子，程序中打印的是一个身份证结构数组中的各个身份证的信息，其中使用了一个指向结构的指针：

```
IDCARD idcs[1000], *p;
... ... /* 假设idcs的元素都在这里给了值 */
for (p = idcs; p < idcs + 1000; p++)
    prtIDCard(p);
```

在许多计算中需要建立新的结构。我们可能不希望破坏已有的东西，而希望在函数执行中建立起新的结构，并希望所创建的结构可以很方便地传递，而不需要反复作为结构值复制。这种情况下可以使用动态存储管理机制，建立动态分配的结构。下面的函数建立一个动态分配的 POINT 结构，并把它的地址作为指针值返回：

```
POINT *mkpoint2(double x, double y) {
    POINT *p;
    p = (POINT *)malloc(sizeof(POINT));
    p->x = x;
    p->y = y;
    return p;
}
```

在使用这个函数时，应当用指向 POINT 类型的指针接收返回值。例如：

```
POINT *pp1, *pp2;
... ...
pp1 = mkpoint2(2.57, 3.86);
pp2 = mkpoint2(pp1->y + 2.6, pp1->x - 0.7);
... ...  /* 使用建立的两个结构 */
free(pp1);  /* 使用完毕，释放存储 */
free(pp2);
```

采用动态建立与管理的方式有一些特点。一方面，这样建立的结构的存在期不受建立操作所在位置的约束，通过指针传递也不必做整个结构的复制。这种实现方式在复杂的软件系统里使用广泛。今后读者还会看到许多这方面的例子。

9.2.2 程序实例

例，基于结构设计第 8 章的账目管理系统。

管理资金来往账目的系统里需要记录每笔收支的日期、项目简记和相应金额。其中用负数表示支出，正数表示收入。假设程序输入按行进行，每行输入一笔账目。这里考虑相关的数据结构和输入问题。显然，有关一笔账的信息是一个逻辑整体，应考虑用一种结构表示。为此设计下面的结构类型：

```
typedef struct {
    int year, month, day;  /* 年月日 */
    char outline[LINELEN]; /* 项目简记 */
    double amout;          /* 金额 */
} ACCITEM;
```

以此作为程序里的基本数据类型。这样，账目管理程序的输入就是填充这种结构，账本就是

与结构有关的一个错误

前面已经看到，我们有时需要定义能创建结构的构造函数，如果被建立的结构很大，把结构作为值返回也会带来大的复制负担。于是可能有人会提出下面方案，令函数返回指向结构的指针。以简单的 POINT 结构为例，构造函数的定义如下：

```
POINT *mkpoint0(double x, double y) {
    POINT temp;
    temp.x = x;
    temp.y = y;
    return &temp;
}
```

这个函数定义中有一个重要错误，请读者注意分析（请考虑变量的存在期问题）。

ACCITEM 的一个数组，而各种查询统计数据的工作就是在这种数组中查询和计算。例如，我们可以定义：

```
enum { NACCOUNT = 400 };
ACCITEM accbook[NACCOUNT];
```

作为程序里所用的基本数据表示。

下面考虑程序输入的实现。如前，我们应该将这个部分定义为一个函数，其工作就是填充一个 ACCITEM 的数组。这种工作前面已经做得很多了。为了使同一个函数可以适应各种输入源的需要，可以为函数引进一个文件指针参数，采用下面原型：

```
int readall(FILE *fp, int limit, ACCITEM accbook[]);
```

为使这个输入更加方便，可以设计实现一个基本账目项输入函数：

```
int readitem(FILE *fp, ACCITEM *ip);
```

这个函数从流 fp 读入并填充好 ip 所指的结构。这里显然需要采用结构指针参数，将实际结构的地址传递给函数。readitem 输入中可能出现几种情况：1）正常完成一个账目项的读入；2）读入数据出错；3）已经读完了所有数据。现考虑让函数在这三种情况下分别返回值 1，0 和-1。这样，函数 readall 里读入循环就可以写成下面形式：

```
while (1) {
    switch(readitem(fp, &accbook[i])) {
    case 1:
        ++i;
        continue;
    case 0:
        ... /* 处理输入出错 */
        continue;
    case -1:
    }
    break;
}
```

循环中用一个开关语句分别处理各种情况。在读入一项数据出错时，readitem 将有关情况报告给 readall，可便于生成更合适的出错信息。函数 readall 还应该负责输入出错后产生错误消息，清理出错之后的输入流（以便继续下一数据项的输入），这些都可以根据需要，在上面程序段里注释着“处理输入出错”的部分完成。最后一种情况是输入结束，这种情况下 switch 语句完成，while 最后的 break 语句导致循环结束。这一循环也可以采用其他写法，例如引进一个 int 变量保存 readitem 的返回值，而后用 if 分情况处理，在数据输入完成时退出循环等。

readitem 的最简单实现是：

```
int readitem(FILE *fp, ACCITEM *ip) {
    int n = fscanf(fp, "%d%d%d %s %lf", &ip->year, &ip->month, &ip->day,
                   ip->outline, &ip->amount);
    return n == 5 ? 1 : n == EOF ? -1 : 0;
}
```

我们可以根据需要考虑在这个层次设法处理输入数据的格式问题等，有关的技术在前面章节里都讨论过了，这里就不再讨论了。注意，我们假设每个帐目条目为一个输入行，readitem 里的输入错误处理也应该反映输入行的概念。不应企图在这个函数里给出错误信息，或者设法重新输入（按照设计，这些事项的处理都是函数 readall 的责任）。

假设我们需要考虑的是一个统计账目信息的系统，现在考虑开发一个驱动交互活动的主函数。这里要为用户提供一系列命令，并能根据命令完成装入账目文件，完成各种统计的工作。函数 main 的主要部分是一个交互循环，其中提示用户输入命令并完成各种工作。这个循环可以写为下面的样子：

```
int main() {
    int n, inum = 0;
```

```
    ACCITEM accbook[NACCOUNT];

    initialization();
    while ((n = getcommand()) >= 0) {
        switch (n) {
        case 0: /* 由用户得到账目文件名并读入 */
            inum = readfile(NACCOUNT, accbook);
            break;
        case 1: /* 计算最终余额 */
            balance(inum, accbook);
            break;
        case 2: /* 统计所有收入 */
            receipts(inum, accbook);
            break;
        case 3: /* 统计所有支出 */
            expenditures(inum, accbook);
            break;
        case 4: /* 打印大额支出，向用户要求额度限 */
            printsome(inum, accbook);
            break;
        default: /* 错误命令。输出错误信息 */
            errmessage();
            break;
        }
    }

    finalization();
    return 0;
}
```

这个主函数调用了若干函数，完成各种具体工作。其中的大部分函数都以 inum 和 accbook 为实际参数完成自己的工作。下面不准备继续给出有关的程序代码了，只是对其中一些函数的实现提出一些想法。

许多程序启动后，开始正式工作之前都要做一些准备工作。对于交互式的程序，启动后可能需要输出一些信息等，有时还可能需要设置一些全局变量（本程序没有全局变量）。上面 main 的最前面调用函数 initialization 完成这方面的工作，其中可以输出一些信息，例如介绍本程序的使用。main 最后的 finalization 完成程序退出前的最后处理。举例说，可以给本程序增加一些有关一次执行中所处理的文件的总的统计信息，这时就可能为程序增加一些全局变量，由上述两个函数完成它们的初始化和最后统计输出工作。

函数 getcommand 应输出提示信息并读入用户命令（一个数）。它输出的提示信息里可以包含有关命令与编码的对照信息。实现这个函数时要考虑对用户错误输入的处理，这里想采用的方式是命令错误时返回某个大的数，当用户要求结束时返回负数。

函数 readfile 应完成所有与输入有关的工作，保证这个函数结束时输入已正常完成。它返回实际输入的账目条目数。函数的工作包括：向用户要求账目文件名，处理文件无法正常打开的情况，调用 readall 实际读入等等。

函数 errmessage()输出错误信息，并应对本程序的使用方式给出一些提示。

其他函数完成的都是很基本的计算，这里就不讨论了。写出各函数原型，完成它们的工作都留给读者作为练习。读者还可以根据自己的考虑为这个系统扩充一些函数。本章后面的练习提出了许多扩充改进的建议。

在上面所采用结构里的日期用三个整数表示，这种做法也有些缺点。例如，完成不同帐目项的日期比较就不方便。此外，采用 3 个整数表示日期也可能浪费了空间。这方面的改进是采用如下结构作为基本表示：

```
typedef struct {
    unsigned long date;  /* 年月日 */
    char outline[LINELEN]; /* 项目简记 */
```

```
    double amout;           /* 金额 */
} ACCITEM;
```

将帐目项读入函数修改为：

```
int readitem(FILE *fp, ACCITEM *ip) {
    int y, m, d;
    int n = fscanf(fp, "%d%d%d %s %lf", &y, &m, &d,
                   ip->outline, &ip->amount);
    if (n != 5) return n == EOF ? -1 : 0;
    ip->date = y*10000 + m*100 + d;
    return 1;
}
```

这就将年月日的信息都编码在一个无符号的长整数里。在需要使用其中的各部分信息时，可以通过算术运算提取。例如，下面语句将帐目项 item 里的月份取出赋值给 m：

```
m = (item.date / 100) % 100;
```

与日期表示有关的最值得一提的事件就是“2000 年问题”。早期计算机应用开发中因为存储紧张，也因为人们的日常习惯，许多（软件或者硬件）系统里的年份表示只用了 2 位（十进制数，或者两个字符）。2000 年世纪之交，这就变成了一个潜在的影响非常广泛的程序错误。例如，日期的比较，计算日期之间的差等等都无法正常完成了。为了解决这一问题，千千万万计算机工作者在 2000 年之前花费了很多人年的工作时间，公司和各国政府部门花费了数以亿美元计的金钱。

9.3 联合（union）

本节介绍 C 语言中的联合机制，这里只介绍它的基本情况与基本性质。联合也是一种数据构造机制，只用在比较复杂的程序中，其用途比较有限。

一个联合是几个类型不同（也可以相同）的成员的组合，其中每个成员各有一个名字。从这些方面看，联合与结构很相似。联合与结构的差异在于它们的表示方式不同。在一个结构（变量）里，结构的各成员顺序排列存储，每个成员都有自己独立的存储位置。联合的情况不是这样，一个联合变量的所有成员共享从同一片存储区。因此一个联合变量在每个时刻里只能保存它的某一个成员的值。

联合说明在形式上与结构相同，但由关键字 union 引导。下面是一个说明，在说明这个联合的形式的同时还定义了两个联合变量：

```
union {
    int n;
    double x;
    char c;
} u1, u2;
```

这样变量 u1 和 u2 就被定义为一种联合变量，这种变量里或者可以存储一个整数，或者可以存储一个双精度数，或者可以存储一个字符（但不能同时存储任何两种成分，更不能同时存储三种成分）。

联合说明也可以像结构说明那样引进一个标志。例如上面联合可以用下面形式描述：

```
union data {
    int n;
    double x;
    char c;
};
```

在此之后就可以用 union data 在代表这个联合说明，用在各种需要的地方了。例如：

```
union data fun10(int m, union data u);
union data u1, u2, u3, u4, *up;
up = (union data *)malloc(sizeof(union data));
```

在复杂程序里人们也采用联合类型定义，定义的方式与结构相同。

联合变量的初始化和使用

联合变量也在可以定义时直接进行初始化，但这个初始化只能对第一个成员做。例如下面的描述定义了两个联合变量，并对它们进行了初始化：

```
union data u1 = {3}, u2 = {5};
```

联合变量的使用方式与结构变量一样，它也可以做整体赋值、成员访问、取地址。对联合成员访问的形式与对结构成员访问的形式也一样。下面是使用联合成员的几个例子：

```
n = u1.n;
u1.c = '\n';
m = u2.n + 167;
```

程序里也可以定义指向联合的指针，同样可以从这种指针出发，通过->运算符访问被指联合对象的成员。例如：

```
int n;
union data dt0, *dp;
dt0.n = 3;
dp = &dt0;
n = dp->n;
dp->n++;
```

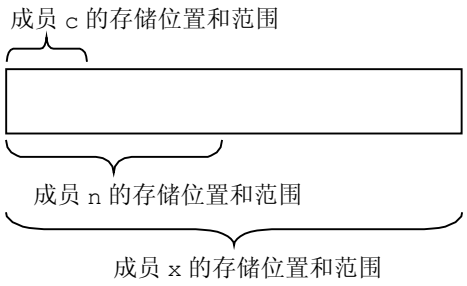

图 9.2　联合 union data 的表示

联合变量的存储实现

前面说过，一个联合变量的各成员共用同一存储位置。由于类型不同，联合中各成员所需要的存储单元也可能不同。这样，一个联合变量要求的存储区大小就将由它最大的成员决定。例如上面说明的 union data，其成员 n 是整数，而成员 c 是一个字符，成员 x 是 double。那么属于这种联合类型的所有变量，例如上面定义的 u1、u2，都需要占用能存放下一个双精度数的存储区域。它们的三个成员在这个区域里的安排见图 9.2（图中没有考虑实际比例）。

联合变量使用的基本规则

联合变量可以看成是一种能够改变自身面目的变量，就像孙悟空有七十二变一样，一个联合变量在不同的时刻可能以不同实质出现，表示着不同的东西。例如，上面 union data 的变量就有三种不同面目，可以当作三种不同的东西使用。以上面定义的变量 u1 为例，我们可以通过 u1.c 将这个变量当作一个 char 变量使用，或者通过 u1.n 将这个变量当作一个 int 变量使用，又可以通过 u1.x 将它当作 double 变量使用。当然，这种使用也不是无原则的，正确的使用必须有道理。例如，下面的程序就完全无理了：

```
u1.n = 5;
u2.x = u1.x;
```

这里刚刚将 u1 当作 int 变量赋值，而后立刻又把它当作 double 变量取值。这样做的结果将没有任何保证。

对联合变量使用的基本原则是：当时它所保存的是什么东西（将它看作哪个成员），就应该按照那种东西（那个成员）的方式去使用。说得更详细点，这就是说：在前面的最近一次对这个联合变量进行赋值采用的是什么方式（把它当作哪个成员来赋值），现在取值时也应该采用同样的方式（通过同样的成员进行访问，取值使用）。C 语言只保证所有遵循这一规则的使用不会出错，对于不符合这个规则的使用，具体结果完全依赖于系统实现。

这种取值与前面赋值的一致性要靠写程序的人来保证，C 语言不检查这个访问规则是否被遵守，实际上它也无法做严格准确的检查。如果取值方式与前面赋值的方式不一致，语言定义没有规定将得到什么结果。

由于联合机制只用在一些很特殊的地方，这里就不打算给出具体的程序实例了。读者在目前学习阶段，只需要了解这个概念及其基本性质。下面讨论枚举之后有一个小例子。

9.4 枚举（enum）

枚举是一种用于定义一组命名常量的机制，以这种方式定义的常量一般称为枚举常量。枚举说明的基本形式是：

```
enum 枚举标志 {枚举常量名, ...};
```

例如，下面是一个枚举说明：

```
enum color {RED, GREEN, BLUE};
```

这就说明了一个具有标志 color 的枚举集合，其中定义出的枚举常量分别是 RED、GREEN 和 BLUE。

一个枚举说明不但在程序引进了一组常量名，同时也为每个常量确定了一个整数值。对于上面这种最基本形式，其第一个常量自动给值 0，随后的常量值顺序递增，这几个常量的值互不相同。对于枚举说明的基本要求是所用的枚举常量名不能相同。假如一个程序里有多个枚举说明，它们定义的枚举常量名也必须互不相同。

枚举标志的用途与结构标志一样，可以用于在程序里写变量定义等。例如写：

```
enum color cr1, cr2;
```

下面是程序里使用这种变量的几个例子：

```
cr1 = RED;
cr2 = BLUE;
... ...
if (cr2 == GREEN) ...;
... ...
switch (cr1) {
    case RED:   ... ...; break;
    case GREEN: ... ...; break;
    case BLUE:  ... ...; break;
}
... ...
```

人们也常常定义枚举类型（用关键字 typedef），其书写形式与定义结构类型的形式类似。我们同样可以定义枚举类型：

```
typedef enum {WHITE, BLACK} COLOR1;
COLOR1 c3, c4;
```

应特别指出， C 语言实际上并不区分程序里定义的各种枚举类型与整数类型，也就是说，不区分用枚举变量以及整型变量。所有枚举类型的变量都被当作整型变量看待。引进枚举描述的主要作用是为了提高程序的可读性。

通过枚举方式定义常量名，在效果上与用 #define 定义符号常量类似。不同之处在于由#define 定义的符号常量通过预处理过程中的宏替换实现，在预处理之后程序编译时，程序里已经没有这种符号常量的任何信息了。而枚举常量是在编译过程中处理的，因此编译系统能够得到与它们有关的信息，这可能对程序调试有用。另一方面，用枚举方式可以方便地一次定义一组常量，使用起来也非常方便。

在枚举说明中还可以给枚举常量指定特定值，所采用的形式与给变量指定初始值的形式类似。如果给某个枚举量指定了值，跟随其后的没有指定值的枚举常量也将跟着顺序递增取值，直到下一个指定特殊值的常量处为止。例如写出下面枚举说明：

```
enum color {RED = 1, GREEN, BLUE, WHITE = 11,
            GREY, BLACK, PINK = 15};
```

这时，RED、GREEN,、BLUE 的值将分别是 1、2、3，WHITE、GREY、BLACK 的值将分别是 11、12、13，而 PINK 的值是 15。

简单实例

人们需要使用联合，常常是为了定义一些函数，使几种不同类型的数据都可以传递给它们处理。例如我们的角度数据可以表示为弧度（0 到 2π）或者角度（度分）。现在需要定义

一些函数等等处理角度数据，这时可以定义类型 ANGLE，所有功能都是基于这个类型定义的。ANGLE 的数据有两种表示形式，我们可以将它定义为一种联合类型。但在函数里怎么能知道得到的一个 ANGLE 变量究竟是用那种形式表示的呢？

解决这一问题的自然方式就是在 ANGLE 类型里加一个标签，也就是说，增加一个成员，专门用来表示具体 ANGLE 变量里数据的情况。标签当然可以用整数或者其他类型，但采用枚举类型就特别合适，因为枚举符是标识符。我们可以有如下定义：

```
enum ANGLETAG {DEGREE, RADIAN};

typedef struct {
    enum ANGLETAG tag;
    union {
        struct { int deg, mnt } degree;
        double radian;
    } data;
} ANGLE;
```

这里将 ANGLE 定义为包含两个成员的结构，其第一个成员是表示标签的枚举成员，第二个成员是实际数据成员 data。data 是一个联合，它或者是结构 degree，或者是双精度数 radian。有了这个类型后就可以定义相关函数了。例如可以定义如下输出角度的函数：

```
void prtdegree(FILE* fp, ANGLE a) {
    switch (a.tag) {
    case DEGREE:
        fprintf(fp, "%d degrees %d minutes",
                a.data.degree.deg, a.data.degree.mnt);
        break;
    case RADIAN:
        fprintf(fp, "%f radian", a.data.radian);
        break;
    }
}
```

这个函数向流 fp 输出。其他函数可以类似地定义。

9.5 编程实例

本节讨论两个与结构等数据机制有关的编程实例。

9.5.1 数据组的排序

现在考虑用结构重新实现前面的学生成绩实例。除了作为结构的编程实例外，我们还想通过这一实例介绍排序的概念，以及标准库排序函数的使用方法。

首先，原来的学生成绩记录文件格式并不合适。文件里应该包含学生姓名和成绩。下面采用如下设计的学生成绩记录文件：

```
02001014 zhangshan 86
02001016 lisi 77
... ...
```

为在程序里保存这样一组学生成绩，应该定义一种结构。我们采用下面定义：

```
enum {
    NUM = 400,
    HISTOHIGH = 60,
    SEGLEN = 5,
    HISTONUM = (100/SEGLEN)+1
};
```

```
typedef struct {
   unsigned long num;
   char name[20];
   double score;
} StuRec;
```

定义下面外部的学生记录数组作为程序里的基本变量（也可以采用动态分配的结构数组，这里不讨论了）：

```
StuRec students[MAXNUM];
```

输入函数具有如下原型，返回实际读入的学生记录数：

```
int readrecs(FILE *fp, int limit, StuRec ss[]);
```

将函数的实际输入流也定义为参数。程序的其他部分可以参考第 6 章的程序实例，除了这里使用的是结构的数组外，其他方面并没有什么差异，这里就不多讨论了。

我们想增加的一项功能是希望程序能按照成绩排列输出学生记录。为了按照成绩输出，确实可以采用多次扫描整个结构数组的方法实现，逐步从高到低或从低到高输出（想想可能怎么做，要什么假定）。但这种实现方式的效率低，也过于特殊。另一种可行方式是先把数组里的学生记录按照成绩重新排列，此后就可以直接顺序打印了。在实际计算机应用中，将一组数据按照某种顺序排列是处理数据中最常见的一种操作，这种操作就称为排序（sorting），人们为此开发了许多有趣的算法。排序的技术问题是《数据结构》课程的重要内容，这里不打算涉及其中的技术问题，只想结合在本例中的使用介绍标准库排序函数 qsort，这个函数能将各种数组里的元素搬来搬去，按照指定顺序排列好。

标准库的排序函数 qsort 在<stdlib.h>里定义，其原型比较复杂：

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *));
```

函数 qsort 有 4 个参数，其中的第一个参数是 void 指针，用于表示被排序的数据项组的起始位置。由于定义库函数 qsort 时无法确定被排序数组的元素类型，因此只能采用这种方式。随后的两个参数的类型都是 size_t，这是标准库定义的一个无符号整数类型名，由具体系统确定它到底是 unsigned 还是其他无符号整型。参数 n 表示被排序的数据项数，参数 size 表示被排序的数组元素的大小。假设我们要对数组 students 里的 snum 个学生记录排序，调用这一函数的形式是（参数 scrcmp 在下面讨论）：

```
qsort(students, snum, sizeof(StuRec), scrcmp);
```

要用 qsort 对一组数据排序，必须告诉它所需的排列方式。qsort 要求提供比较两个数据元素大小的准则，它将按有关准则将“较小”元素排在前面，大元素排到后面。qsort 的最后参数（函数指针参数）cmp 就是所需的比较准则。要对一组数据排序，必须先定义一个表示比较准则的函数，该函数必须符合 qsort 对比较函数的类型要求和功能要求。

比较函数的类型要求在 qsort 原型的参数中描述得很明确。假设我们比较学生分数的函数取名 scrcmp，其原型就必须是：

```
int scrcmp(const void *vp1, const void *vp2);
```

qsort 将对被排序的数据元素调用有关的比较函数。为正确写出 qsort 能用的比较函数，必须弄清楚：1）qsort 通过参数传给函数的是什么；2）qsort 要求函数以什么方式表示元素比较结果。

qsort 传递给函数的是两个指向被排序元素的指针。由于 qsort 是通用排序函数，写函数的人当然不可能知道被排序元素的类型，因此只能采用(void*)指针。我们需要在所提供的比较函数里将这个指针转换为元素类型的指针后使用。其次，qsort 要求比较函数在第一个元素“大于”第二个元素时返回正值，两元素相等时返回 0，第一个元素较小时返回负值。当然，这里的“大小”完全根据我们的排序需要确定。

scrcmp 的定义实际上很简单。为清晰起见，在函数里先定义两个 StuRec 类型的指针，将参数值（转换后）给它们，而后比较两个结构中 score 成员的大小：

```
int scrcmp(const void *vp1, const void *vp2) {
    StuRec *p1 = (StuRec*)vp1, *p2 = (StuRec*)vp2;
    return p1->score > p2->score ? 1 :
           p1->score == p2->score ? 0 : -1;
}
```

这样就可以完成对学生成绩记录的排序工作了。

qsort 是一个“通用的”排序函数，按照所提供的排序准则工作。如果我们提供另一个不同的排序准则，qsort 就能完成对同一组数据的另一种排序。例如，我们可能希望将学生记录按照 5 分一段，分段排列，那么就可以采用下面比较函数：

```
int scr5cmp(const void *vp1, const void *vp2) {
    int n1 = ((StuRec*)vp1)->score,
        n2 = ((StuRec*)vp2)->score;
    return n1/5 - n2/5;
}
```

请注意表达式中运算符的优先级问题。

如果要对这些记录按照学生名排序，可以借用标准库的 strcmp 函数。但我们不能直接使用（参数类型不对），qsort 调用函数时提供的是指向被比较对象的指针。这里被比较的是两个字符串，用 char* 类型的指针代表它们。我们可以写出下面转接函数：

```
int str1cmp(const void *vp1, const void *vp2) {
    char *p1 = (StuRec*)vp1->name, *p2 = (StuRec*)vp2->name;
    return strcmp(p1, p2);
}
```

这个函数也可以简写为：

```
int str1cmp(const void *vp1, const void *vp2) {
    return strcmp((StuRec*)vp1->name, (StuRec*)vp2->name);
}
```

有关这个程序其他部分的修改完善，都留给读者作为练习。

对于一组数据，完全可能有许多不同的排序要求。例如对一个超级市场的商品食品记录，我们可能需要将这些记录按照商品名称排序，按照食品出厂日期排序，按照食品的过期日期排序，按照商品的单价排序，按照一段时间的销售额排序，按照出品厂商名称排序等等。只要提供了合适的比较函数，这些工作都可以借助于函数 qsort 完成。我们还可以用 qsort 对某种类型的数组中一段元素排序。

9.5.2 复数的表示和处理

C 语言提供了许多数值类型，可供人们在处理数据时使用。但这里的数值类型也不完全，例如这里就缺乏一个复数类型。假设我们要写一个处理复数数据的程序，那应该怎么办呢？我们当然可以直接用两个 double 表示一个复数去完成这种程序。例如定义函数：

```
addcomplex(double r1, double i1, double r2, double i2)
```

这样不但很难返回结果，调用时也需要时时记住各个参数的位置。程序写起来会很麻烦。

应注意到，这里的一个复数就是一个逻辑上的数据体，应该定义为一个类型，而后再定义一批以复数类型为操作对象的函数。在完成了这些工作之后，再写程序的其他部分就会变得清晰简单了。人们在程序设计实践中逐步认识到，在设计实现一个比较复杂的程序时，最重要的一个步骤就是找出程序里所需的一批数据类型。将它们的结构和有关功能分析清楚，设计并予以实现。而后在这些类型的基础上实现整个程序。这样做，得到的程序将更清晰，其中各个部分的功能划分比较明确，更容易理解，也更容易修改。

现在我们就具体考虑复数类型的实现。复数可以有多种表示方式。一种常见表示方式采用平面坐标，将一个复数表示为一个实部和一个虚部；另一种常见方式是采用极坐标表示，将一个复数表示为幅角和模。对于一种特定应用，某种表示方式可能更适合一些，因此需要仔细斟酌。这里作为例子，我们选择平面坐标表示，将一个复数表示为两个实数，分别表示其实部和虚部。针对通常的复数运算，可以考虑用两个 double 类型的值表示一个复数。

应该采用什么机制将这两部分结合起来呢？由于这两部分的类型相同，我们可以考虑用具有两个 double 元素的数组，或者用具有两个 double 成员的结构。由于需要定义许多运算，用结构表示有利于将复数作为参数传递或者作为结果返回。因此有下面定义：

```
typedef struct {
    double re, im;
} Complex;
```

下面就可以考虑基于这个类型的各种运算了。

由于复数类型数据对象里的数据项很少，我们可以考虑直接传递 Complex 类型的值和结果，这样可以避免复杂的存储管理问题。考虑最基本的算术函数，它们的原型应是：

```
Complex addComplex(Complex x, Complex y);
Complex subComplex(Complex x, Complex y);
Complex tmsComplex(Complex x, Complex y);
Complex divComplex(Complex x, Complex y);
```

还需要考虑如何构造复数。我们不希望在使用复数的程序里直接访问 Complex 类型的成分，如果人们经常需要那样做，程序里的错误将很难控制和查找。如果对复数的所有使用都是经过我们定义的函数，只要这些函数的定义正确，程序里对复数的正确使用也就有保证了。下面是几个构造函数：

```
Complex mkComplex(double re, double im);
Complex d2Complex(double d);
Complex n2Complex(int n);
```

后两个函数可以看作由 double 和 int 到复数的数值转换函数，定义它们是为了使用方便：

```
Complex mkComplex(double r, double i) {
    Complex c;
    c.re = r;
    c.im = i;
    return c;
}

Complex d2Complex(double d)  {
    Complex c;
    c.re = r;
    c.im = 0;
    return c;
}

Complex n2Complex(int n)  {
    Complex;
    c.re = r;
    c.im = 0;
    return c;
}
```

下面是加法函数的定义：

```
Complex addComplex(Complex x, Complex y) {
    Complex c;
    c.re = x.re + y.re;
    c.im = x.im + y.im;
    return c;
}
```

减法函数与此类似。乘法函数的算法复杂一点，根据数学定义也不难给出。定义除法函数时却遇到了一个新问题：如果除数为 0 时该怎么办？复数除法的定义是：

$$\frac{a+bi}{c+di}=\frac{ac+bd}{c^2+d^2}+\frac{bc-ad}{c^2+d^2}i$$

当除数为 0 时两个分式的分母为 0。C 语言对内部类型除 0 的规定是“其行为没有定义”，要求编程序的人保证不出现这种情况。我们在实现复数操作时也有两种选择：1）直接沿用 C 语言的方式，要求使用复数类的人保证不出现除 0。2）检查除 0 的情况，提供动态信息并返回某个特殊值。采用第一种方式就可以直接按上面公式定义函数。下面函数定义里检查了除 0 情况，输出错误信息并返回 1 对应的复数：

```
Complex divComplex(Complex x, Complex y) {
    Complex c;
    double den = y.re * y.re + y.im * y.im;

    if (den == 0.0) {
        fprintf(stderr, "Complex error: divid 0.\n");
        c.re = 1; c.im = 0;
    }
    else {
        c.re = (x.re * y.re + x.im * y.im) / den;
        c.im = (x.im * y.re - x.re * y.im) / den;
    }
    return c;
}
```

有了这些函数之后，就可以很方便地写各种复数计算了。

为了使用方便，我们还可以定义复数的输出和函数：

```
void prtComplex(FILE *fp, Complex x);
```

它向某个流（标准流或者打开文件创建的流）输出一个复数。实现这个函数之前，先需要为复数确定一种输出形式，函数的定义很简单。还可以定义输入函数：

```
int readComplex(FILE *fp, Complex *xp);
```

参数 xp 对应的实际参数应该是一个 Complex 结构的地址。这个函数的设计应该参考标准库的输入函数：在实际完成复数的输入时返回 1，转换失败时返回 0，遇到文件结束或者其他错误时返回 EOF 值。具体实现依赖于对输入形式的设计。这一对输入输出函数最好能采用统一的设计，使通过函数 prtComplex 产生的输出能通过 readComplex 输入。举例说，我们可以将实部为 *a* 虚部为 *b* 的复数输出为 (*a*, *b*)。这时输出函数可以如下定义：

```
void prtComplex(FILE *fp, Complex x) {
    fprintf(fp, "(%f, %f)", x.re, x.im);
}
```

我们不应该让自己类型的输出函数随便输出空格或者换行。实际程序的输出形式由使用这些基本函数的用户确定。输入函数的定义留给读者完成。

这样就完成了一个基本的复数计算程序包，基于它已经可以做许多有用的事情了。读者还可以扩充这个程序包的功能，增加其他有用的函数。第 10 章还将进一步讨论如何利用 C 语言预处理的功能，将这样一个程序包组织为更加易用的结构，使其他需要使用复数功能的程序都能很方便地使用它。

9.6 链接结构（自引用结构）

随着需要用计算机处理的问题变得更丰富复杂，程序的实践也不断地提出新的问题，促使人们去思考，设法创造出新的解决问题的方法。这节里我们从一个问题出发，讨论一类复

杂的数据表示方式，自引用结构。

9.6.1 链接结构

假设我们现在需要对一些正文文件里出现的所有的词做一个统计，统计其中每个词在文件里出现的次数。这是一类很典型的计算机应用的一个例子。举例说，在语言学研究中人们常要研究各种文本中各种词汇的出现频率。在分析各类文献，分析计算机程序等许多研究工作中都需要做类似的词频统计工作。

首先可以看到，由于统计前不知道文件里到底有多少不同的词，我们将无法在编程序时确定在统计中需用的完整数据结构（如像前面 C 关键字统计程序那样，事先定义一个计数器数组）。一个可能解决方案是使用动态分配的计数器数组，必要时调整数组的大小（用标准库函数 realloc，前面已经有这种实例）。关于这种方案，第 7 章已经详细的讨论。这种方式的缺点是需要一个大存储块保留所有数据。如果文件里的单词非常多，能否找到足够大的存储块有时也会成为问题。

下面介绍另一种结构方式，采用该种方式表示的数据结构可以方便地在执行中动态增长或者变化调整，很容易满足被存储数据项个数在执行中动态增加或减少，或者相互关联关系变化的需要。这种结构称为链接结构，一般是利用程序语言中的指针、结构或记录（C 语言里的 struct）和动态存储管理实现的。

在 C 语言里实现链接结构的基本构件是*自引用结构*，这种结构的每个元素（数据对象，简单情况下用一个 struct 表示）可以分为两部分，其形式可示意为：

结构中的各种实际数据成员
一个或几个指向本类结构的指针

这种数据对象中的第一个部分是它们实际需要保存的数据内容；第二个部分是一个或者几个指针，指向同类型的结构（继续学习可以知道，在更复杂的数据结构中，这种指针也可以指向其他类型的数据对象）。这种指针本身并不保存具体数据，只是用于建立不同数据对象之间的关系。这种形式就是将它们称为“自引用结构”的原因，结构中的一个 struct 可以通过指针引用另一个 struct，多个自引用结构可以通过指针部分连接起来。这种引用其他结构对象的指针被称为*链接*，通过链接形成的复杂数据结构被称为*链接结构*。

最简单的链接结构是通过线性链接形成的*表*，或称*链接表*。这里每个自引用结构只有一个链接指针，多个 struct 一个链接到另一个，形成一个链接起来的序列，如图 9.3 所示。一个表就像一条链条，其中的每个自引用结构像一个链节，通常称为表中的*结点*。人们通常把表最后结点的链接指针设置成空指针，表示这个表的结束。

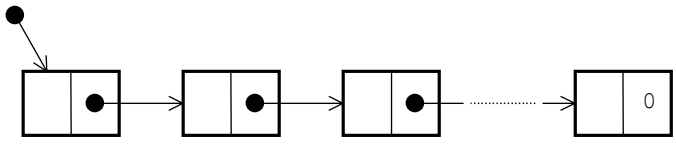

图 9.3 线性的链接表结构

一般说，链接表中所有的数据对象都通过动态存储分配的方式得到。我们用一个指针指向表的第一个结点，这个指针代表整个的表，这种指针称为*表头指针*。在需要处理这种表里的数据时，我们需要用一个指向表结点的工作指针。首先利用表头指针是工作指针指向表的第一个结点，而后就可以沿着表中的链接（指针）顺序访问表里的各个结点，对其中的数据实施所需的各种操作。

以各种不同的链接对象作为基本构件，还可以构造出许多更加复杂的数据结构。另一种非常典型的结构是二叉树结构，在这种结构

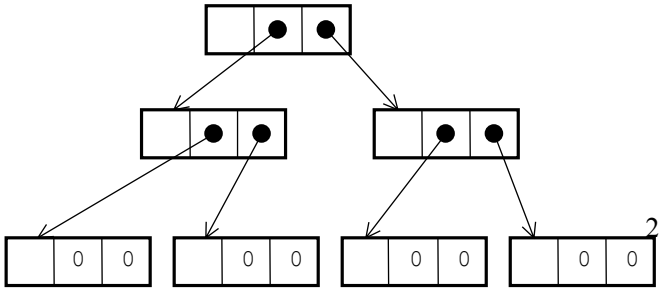

图 9.4 一种复杂的链接结构——二叉树

里的每个结点有两个链接指针。例如图 9.4 画出的是一个二叉树结构，每个结点的两个指针指向其他结点。在后续的数据结构课程里，有关于这些方面的进一步讨论。

9.6.2 自引用结构的定义

本节将以词频统计问题为例讨论有关的问题。现在假设已经确定采用链接表作为程序中使用的基本数据结构，以完成上面提出的任务。首先，我们仍然需要对有关的情况进行仔细分析。显然，在程序处理的过程中，对文本里出现的每个词，需要保存的相关信息包括了这个词本身，以及对这个词出现次数的计数值。为简单起见，我们假设文件中遇到的所有词的长度都不超过 19 个字符*，并假设整数的表示范围足以应付词的统计。这样，作为统计用的结构里的两个基本数据成员可以定义为：

```
char word[20];
int count;
```

现在的问题是自引用结构本身的说明。

说明自引用结构有一点麻烦，因为在结构说明里需要描述一个指向本类结构的指针。下面介绍两种说明方法，它们在简洁性或清晰性方面各有长短。由于程序里对这种数据的操作很多，我们理所当然地应该把所需要的自引用结构定义为类型。这里不但要定义结构类型本身，还应该定义一个指向结构的指针类型。第一种定义方式如下：

```
typedef struct node NODE, *LINK;

struct node {
    char word[20];
    int count;
    LINK next;
};
```

这个定义利用了 C 语言允许写不完全的结构说明的规定。描述中的第一行定义了两个类型，NODE 是被定义的结点类型（这是一个 struct node 自引用结构类型），而 LINK 是指向这种结构的指针类型。这行中没给出结构本身的描述，所以说这是一个不完全的结构说明，C 语言允许这种方式。随后给出结构本身的描述，很容易理解。在这里，结构标志起了联系两个定义的作用。在描述的结构里有三个成员，前面两个是实际数据成员，形成了一种单词计数器。最后的是一个链接指针成员，用于构造单词计数器的链接表。至此，与自引用结构有关的类型定义就完成了。

现在介绍第二种方式。下面的描述定义了同样的两个类型：

```
typedef struct node {
    char word[20];
    int count;
    struct node *next;
} NODE, *LINK;
```

这种方式比较紧凑，因此也被许多人采用。但从这个定义描述里，不容易看清楚在结构里定义的 next 成员本身就是一个 LINK 类型的指针成员。在这个方面，第一种定义方式有优越性。在需要定义这类结构时，读者可以根据自己的喜好选用其中任何一种。

9.6.3 程序实现

有了这些与结构有关的类型定义之后，程序中的主函数部分可以写为：

```
#include <stdio.h>
#include <ctype.h>  /* 因为在读入过程中需要判断字符类型 */
#include <string.h> /* 程序中需要做字符串复制和比较 */
```

* 如果没有这个限制，程序就需要解决任意长度的词的保存和处理问题。采用已知技术完全可以解决这种问题，但处理它将将使程序进一步复杂化，而且与这里希望讨论的重点无关。我们把这个问题留给读者，作为学习本章后的一个大的程序练习。

```
#include <stdlib.h> /* 程序中要做动态存储分配 */

enum { MAXLEN = 20 };
typedef struct node NODE, *LINK; /* 这一段是类型定义 */
struct node {
    char word[MAXLEN];
    int count;
    LINK next;
};

int getword(int limit, char w[]); /* 有关函数的原型说明 */
LINK addword(LINK l, char w[]);
void printwords(LINK l);

LINK list = NULL;   /* 全局变量，作为表的头指针 */
char word[MAXLEN];  /* 读入用的临时字符数组 */

int main (void) {
    while (getword(MAXLEN, word) != 0)
        list = addword(list, word);
    printwords(list);
    return 0;
}
```

程序头部是一些必要的预处理命令，它们之后给出了程序所需要的类型定义和几个函数原型说明。这里用枚举常量定义表结点结构 word 成员的大小，以方便程序的修改和调整。

在几个函数里，getword 不是新东西，这里也要求它把读入的单词存入参数数组里，最后返回词的长度，长度为 0 表示再也没有新词了，程序的工作可以结束。在 getword 返回时，数组 word 里的有效字符一定不超过 19 个，函数还应在有效字符后面放一个空字符 '\0'。这个函数的定义留给读者完成，应根据具体需要定义。

函数 printwords 非常简单，可以用循环方式实现。它借助一个指针（参数，也是局部变量）顺序访问表中各结点，每次打印一个结点中的数据。这个函数可以写为：

```
void printwords(LINK p) {
    for ( ; p != NULL; p = p->next)
        printf("%d  %s\n", p->count, p->word);
}
```

这个函数定义表现出链接结构处理的基本方式：用一个指针，从表头结点开始，沿着表的链接指针逐步前进，一个一个结点地顺序进行处理，一直进行到表的结束。人们常把链接表处理中使用的这种指针称作扫描指针，把这种处理过程看作对整个链接表的一次“扫描”。上面函数里所用的扫描指针是函数的形式参数 p，它也是一个局部变量，调用这个函数时它将得到表中第一个结点的地址。从这个函数定义里也可以看到链接表最后的空指针的作用，它帮助程序确定处理过程的结束（循环的终止条件）。

函数 addword 是程序中最关键的部分，其第一个参数是统计表（或者其中一部分），第二个参数是当前处理的词。addword 完成一个词的统计，更新当时的统计表，返回修改过的表。主函数得到一个词后就调用这个函数，使统计表得到更新。

在 addword 处理一个词时，如果这个词是第一次遇到，那么就需要为它建立一个新结点，把词本身和统计值 1 记录进去；如果这个词已经遇到过，表里有它的计数器结点，就可以将找到的相应结点里的统计值加一。下面用递归方式定义 addword。在处理一个可能修改的表时这样做特别方便。我们也可以用循环解决问题，但写出的函数复杂一些。

为了使程序简洁易读，首先定义一个分配结点的辅助函数 mknode，它申请一个实现结点的存储块，并在其中存入有关的信息：

```
LINK mknode(char w[]) {
```

```
    LINK p = (LINK)malloc(sizeof(NODE));
    if (p != NULL) {
        strncpy(p->word, w, MAXLEN);
        p->count = 1;
        p->next = NULL;
    }
    return p;
}
```

这里需要注意两点：一是检查分配的成功与否，不能盲目地通过指针去赋值。二是应当把新分配结点的链接指针域置为空，使它处于一个确定状态。函数里还特别注意将 next 指针置空，以保证安全（如果不做这个赋值，next 的值不确定），也是下面程序的需要。

有了这些之后准备，函数 addword 就很简单了：

```
LINK addword(LINK p, char w[]) {
    if (p != NULL) {
        if(strcmp(p->word, w) == 0)
            p->count++;
        else
            p->next = addword(p->next, w);
        return p;
    }
    else
        return mknode(w);
}
```

如果被加入词的表不空，根据指针当时所指结点里的词与当前词的关系，分为两种不同情况。如果这两个词相同，那么只要把结点计数器加一，统计表的更新就完成了。如果这两个词不同，那么就应该用考虑将当前词的信息“加入”表的后面部分，因为它可能出现在后面部分。递归更新后的后面部分仍然链接在原处。两种情况的最后都返回更新后的表。

另一种情况是被处理的表已经为空。这种情况说明当时的统计表里没有被处理的这个词，这时应该以这个词作为数据，建立一个新结点返回。出现这种情况可能是因为表中还没有结点（在程序初始时）；也可能是因为在对表的递归处理中一直没有找到有关结点，函数顺着链接递归达到了表的末尾。无论如何函数都返回创建的新结点（作为只包含一个结点的表），它会被链接到整个表的最后。

我们也可以用循环方式重新写函数 addword。函数原型不变，调用方式也不变。与 printwords 类似，在这里也需要一个在表中扫描的指针。函数首先处理空表的特殊问题（程序初始时），因为一般情况是将新结点连在已有结点的 next（更改的是某个结点的 next 部分）。程序开始时直接需要直接返回第一个结点。函数定义如下：

```
LINK addword(LINK list, char w[]) {
    LINK p;
    if (list == NULL) return mknode(w); /* 表为空，开始情况 */

    for (p = list; ; p = p->next) {
        if (strcmp(p->word, w) == 0) {
            p->count++;
            break;
        }
        if (p->next == NULL) {
            p->next = mknode(w);
            break;
        }
    }
    return list;
}
```

这个函数比前一个复杂一些，遇到新词时建立的新计数器结点也是总连接在表的最后。

与此类似的另一种实现方式是将 list 的地址作为 addword 函数的参数，让 addword 函数根据需要修改 list（注意，只有 list 的值为空时才需要修改它）。函数定义如下：

```
int addword(LINK *lp, char w[]) {
    LINK p;
    if (*lp == NULL) /* 表为空，开始情况 */
        return (*lp = mknode(w)) != NULL ? 1 : 0;
    for (p = *lp; ; p = p->next) {
        if (strcmp(p->word, w) == 0) {
            p->count++;
            return 1;
        }
        if (p->next == NULL)
            return (p->next = mknode(w)) != NULL ? 1 : 0;
    }
}
```

这种定义方式的优点是可以通过函数返回值报告工作情况。如果采用这一定义，main 里的函数调用就需要改为：

```
addword(&list, word);
```

更合理的方式是将 main 里的循环改写为：

```
while (getword(MAXLEN, word) != 0)
    if (addword1(&list, word) == 0) {
        printf("No enough memory. Partial result.");
        break;
    }
```

这个函数还有许多实现方式，例如可以在函数里直接访问和修改证据变量 list。作为全局变量处理时，还可以采用将把新结点插在表最前面的方式，函数定义能有些简化，空表情况也不必单独处理了。这些实现方式请读者自己考虑。

很容易看到这个程序定义的缺陷和问题。由于限制了词的最大字符数，对于那些特别长的词，这个程序肯定无法正确统计。当然，由于这里没有给出函数 getword 的定义，具体会出现什么错误我们无法考虑。请读者结合自己实现的 getword 函数，对这方面问题做一个分析。我们还可以发现，存储空间的限制也可能引起统计错误。请读者考虑，对于上面的各种实现，如果到某个时候存储空间申请不到，会出现什么问题：会不会出现程序运行的严重错误？会不会导致非法的指针访问？统计结果将会出现怎样的错误？

9.6.4 数据与查找

上面的程序实现基本上能够完成要求的工作，但这种实现方法也有很大的缺点。其中一个非常重要的问题是实现效率。首先应当注意到，随着处理中遇到新词的增加，统计数据所用的计数器表会不断增长，而且这个表的长度正好等于处理中已经遇到的不同词的个数。在处理小文件时，由于文件里的词很少，不同的词当然也不多，统计表也就不会很长。如果被处理的文件非常大，其中不同的词很多，处理效率问题就会越来越明显。

假设一个文件中有 1000 万个词，其中不同的词共有 1 万个，让我们来做一个大略的分析，看看完成统计需要做多少工作。显然，统计表的长度最终将达到 1 万个结点，假设这个表的长度均匀增长，那么工作过程中表的平均长度可以看作是 5000 个结点。为了找到一个词所对应的统计项（或者确定它不在表里），我们假设平均需要检查半个表，这样，为完成一个词的统计，平均要做的字符串比较大约为 2500 次。由此可以估算出，完成整个工作大概要做 250 亿次字符串比较，当然还要做一些其他工作。假设所用的计算机每秒能完成 100 万个字符串比较，完成整个工作也需要 25000 秒，也就是大约 7 小时。

在实际工作中，1000 万个词并不是一个很大的数目，要处理更大的问题，对于所采用的方法就需要做进一步的研究。由于这种存储和查找是实际计算机应用中非常典型的问题，

人们对它进行了许多研究，提出了各种提高效率的方法。一个常用的做法就是采用前面简单介绍过的树型结构，人们还提出了许多其他处理这种问题的数据表示方式。关于这方面的问题，本章的练习里提出了几种改进的想法，后续的数据结构课程里有许多进一步讨论。读者也可以自己开动脑筋，想一想有什么好主意。

问题解释

1. (8.2) `mkpoint2`：考虑变量的存在期。当函数退出的时候，局部自动变量的存在期结束，而这里把它的地址作为指针值返回。在函数结束后，这个指针值已经不是合法的了，因为它不指向一个合法的变量。所以这个函数的定义是错误的。

本章讨论的重要概念

结构，记录（其他语言使用的术语），结构标志，结构类型，对齐问题，结构成员访问，字段，联合，联合标志，枚举，枚举常量，链接结构，自引用结构

练习

1. 请定义如下计算平面点与其他图形之间关系的函数：
   1） 计算一个点是否在一个圆之内；
   2） 计算一个点与一个圆之间的距离（该点与圆周上最近一点之间的距离）；
   3） 判断一个点是否在一个矩形内部；
   4） 计算一个点与一个矩形之间的距离（该点与矩形上最近一点之间的距离）；
   5） 计算两个圆之间的距离（如果相切或者互相有重叠则认为距离是 0）；
   6） 计算两个矩形之间的距离；
   7） 计算一个矩形与一个圆之间的距离。
2. 定义一个表示时间的结构（包括时、分、秒成员）类型，定义由这种类型的参数计算总秒数的函数。
3. 定义一个表示日期的结构类型，然后定义下面函数：
   1）由这种类型计算一天是该年的第几天的函数（注意闰年问题）；
   2）比较两个日期的大小的函数；
   3）计算两个日期的天数差的函数；
   4）定义以一个日期和一个天数为参数的函数，它能算出某日期若干天之后的日期；
   5）定义计算某日期之前若干天的日期的函数；
   6）其他你认为重要的日期函数。
   用这些函数计算：到本世纪末还有多少天，今天是本世纪的第多少天，你自己已经生活了多少天，离你的下一个生日还有多少天，你最近的生日与今天的天数差，从孔子出生到今天已经有多少天，等等。
4. 定义几个比较两个身份证大小的函数：以出生年月日作为标准；以身份证号作为标准，以姓名的编码作为标准。考虑应该使用结构参数，还是使用结构指针参数。
5. 完成正文中设计实现的复数结构及其相关函数，再为它增加一些有用的函数。设计一个交互式的复数计算系统界面主函数。
6. 以两个整数为成员的结构可以表示有理数。定义计算这种有理数加、减、乘、除运算的函数。函数的返回值如何定义比较合理？
7. 开发一个小型信息查询系统。当这个系统运行时，人输入一个城市的名字（拼音），它就能给出由你所在城市到所输入城市的铁路距离，火车票价。（如果要求的是对输入的一对城市给出有关信息，你能够完成这个小系统吗？）

8. 考虑定义一个计算台球位置的函数，这个函数的一个参数应当是一个表示台球状态的结构，其成员至少包括：台球直径、当前位置，速度向量。把台球桌的大小作为程序常量，定义计算t时间后台球状态的函数。进一步考虑桌面阻力因素和台球桌边沿的弹性系数，修改前面定义的函数，使其能够更好地反映现实世界的情况。考虑利用这个函数做一个有趣的程序实例。
9. 完成正文中基于结构的学生成绩处理程序。提供按照成绩排序输出的功能。
10. 完成正文中的账目管理系统。
11. 对你所完成的账目管理系统做一些扩充，例如：1）增加用直方图形式显示账目变动曲线的情况，可以考虑按时间周期的收入情况、支出情况和余额情况等。2）增加对本次处理的一系列账目的总的统计。3）提供对一段时期的各种情况的统计功能。4）采用动态存储管理的方式分配账目“数组”，保证能够处理任意大小的账目。
12. 对你所完成的账目管理系统做一些修改，使用户在启动程序时可以通过命令行参数提供一批文件名，程序在启动后自动装入这些文件里的账目记录，而后接受用户命令完成各种统计工作。增加一个命令，使程序可以要求新文件，并将新文件的内容附加到程序已经装入的账目条目之后。再增加一个命令，使程序可以将程序里当前保存的账目全部输出到用户指定的文件里。
13. 修改账目管理系统，使它在每次启动后首先去查看名字为 `account.sav` 的文件。如果这个文件存在，本系统就将列在该文件里的账目装入。用户提供的其他文件的内容都附加在程序里已有账目内容之后。在程序结束时将当时数组里保存的账目重新存入上述文件，以备本程序下次再用。注意，请在保存之前将整个数组按照日期排序。
14. 参考上面的账目管理系统，设计一个学生成绩管理系统。它应该能：
    1） 启动时自动读入一个学生成绩文件。
    2） 读入新的学生成绩文件，并将其中的成绩记录在有关学生的原有成绩记录之后（可以假定每个学生总共的课程数不超过某个事先定义好的常量，因此一个学生的所有成绩可以存在一个数组里，作为学生记录的一部分）。这里可能出现新的学生和新的课程；
    3） 每次运行完成后能将程序中所有学生成绩记录保存到将来能自动读入的文件。
    4） 增加各种有用的统计输出功能。（在完成这一工作时，可以考虑或者不考虑课程名。如果考虑课程名，可以维持一个课程名与成绩记录位置的对照表）。
15. 扩充正文中的单词统计程序，使之能处理任意长度的单词。（提示：请考虑用动态分配的存储块保存单词）
16. 如果文件中的单词很多，记录它们的链接表就会非常长，查找确定一个单词所需的时间也会大大增加。考虑下面的改进技术：
    1） 定义一个指针数组 `LINK table[21]`，数组里的每个指针指向一个结点表；
    2） 将长度为 `i` 的单词都存入指针 `table[i]` 所指向的表里（`table[0]` 闲置）；
    3） 在 `words[20]` 里保存长度大于等于 20 的所有单词。

    这种方式可以减少程序处理每个单词所需的时间。请采用这一技术重新实现单词计数程序。通过处理很大的文本，比较这一实现与本章正文中的实现的执行情况。
17. 在实现单词计数时（假设为英文单词）考虑下面的改进技术：
    1） 定义一个指针数组 `LINK table[26]`，数组里的每个指针指向一个结点表；
    2） 按照单词的首字母将读入的单词分别存入指针数组 `table` 的不同元素所指向的链接表里，在读入单词后查找存入表之前将单词里所有大写字母改为小写字母。

    通过处理很大的文本，比较这一实现与本章正文中所提出的实现的性能。