21 世纪计算机专业重点课程辅导丛书

C 语言习题与解析

（第 2 版）

李 春 葆　　编著

清 华 大 学 出 版 社
北　京

第 10 章　结构体与共用体

本章学习要点

☑ 深入掌握结构体类型、共用体类型、枚举类型和自定义类型的概念和定义方法。
☑ 深入领会结构体类型和共用体类型之间的差异。
☑ 当结构体变量或指针作为函数参数时，理解各种形参实参传递方式的差异。
☑ 在掌握使用结构体实现链表的基本运算的基础上，能够实现复杂链表算法。
☑ 能够灵活地自定义结构体类型，节省程序代码。

10.1　基本知识点

10.1.1　结构体定义和变量说明

1. **结构体的定义**

结构体由不同数据类型的数据组成。组成结构体的每个数据称为该结构体的成员项，简称成员。在程序中使用结构体时，首先要对结构的组成进行描述，这称为结构体的定义。结构体定义的一般格式如下：

```
struct <结构体名>
{
    <数据类型>  <成员名 1>;
    <数据类型>  <成员名 2>;
     …
    <数据类型>  <成员名 n>;
};
```

其中，struct **是关键字。每个结构体都可以含有多个同类型的成员名，它们之间以逗号分隔。结构体中的成员可以和程序中的其他变量同名；不同结构体中的成员也可以同名。**

例如，以下语句定义了一个学生情况结构体 student**：**

```
struct student
{
   char name[12];              /*姓名*/
   char sex;                   /*性别*/
   int age;                    /*年龄*/
   int score;                  /*成绩*/
   char class[16];             /*班号*/
};
```

注意 结构体类型的说明只是列出了该结构的组成情况，标志着这种类型的结构“模式”已存在，编译程序并没有因此而分配任何存储空间。真正占用存储空间的是具有相应结构体类型的变量等。

2. 结构体类型变量的定义

可以使用如下几种方式定义结构体类型的变量。

（1）先定义结构体类型再定义变量

在已经定义好结构体后，再定义结构体变量的一般形式如下：

```
struct <结构体名> <结构体变量名表>;
```

其中，<结构体变量名表>由一个或多个结构体变量名组成，当多于一个结构体变量名时，它们之间用逗号分隔。

例如，以下语句定义两个结构体变量 student1 和 student2（student 结构体在前面已定义）：

```
struct student student1,stduent2;
```

（2）在定义结构体类型的同时定义变量

这种定义结构体变量的一般格式如下：

```
struct <结构体名>
{
    <结构体成员表>;
} <结构体变量名表>;
```

例如，以下语句采用这种定义方式定义两个结构体变量：

```
struct student
{
    char name[12];
    char sex;
    int age;
    int score;
    char class[16];
} student1,student2;
```

（3）直接定义结构体类型变量

这种方式在定义结构体变量时不出现结构体名，一般格式如下：

```
struct
{
    结构体成员表
} 结构体变量名表;
```

例如，以下语句采用这种定义方式定义两个结构体变量：

```
struct
{
    char name[12];
    char sex;
    int age;
    int score;
    char class[16];
} student1, student2;
```

10.1.2 结构体变量的引用和初始化

1. 结构体变量的引用

结构体是由不同数据类型的若干数据集合而成。在程序中使用结构体时，一般不允许把结构体作为一个整体参加操作处理，而应通过对结构体的各个成员项的引用来实现各种运算。

（1）引用结构体变量中的一个成员

先找到对应的结构体变量，然后再从中找出其中的一个成员。引用的一般方式如下：

<结构体变量名>.<成员名>
<指针变量名>-><成员名>

第二种方式是在结构体指针变量情况下使用。例如：

```
struct student stud,*ps;
```

那么引用 age 成员分别是：

```
stud.age
ps->age
```

注意 “.”运算符的优先级最高，因此 stud.age++是对 stud.age 进行自增运算，而不是先对 age 进行自增运算。

（2）结构体类型变量的整体引用

可以将一个结构体变量作为一个整体赋给另一个同类型的结构体变量。例如：

```
struct student student1, student2;
...
student1=student2;                          /*将 student2 复制给 student1*/
```

执行该赋值语句时，将 student2 变量中各成员项依次赋给 student1 中相应的各成员。这种赋值的前提条件是两个结构体变量必须具有相同的数据类型。

2. 结构体变量的初始化

在结构体变量说明的同时，可以给它的每个成员赋初值，这称为结构体变量的初始化。

结构体变量的初始化规则与数组相同，即只有当结构体变量为全局变量或静态变量时，才能进行初始化。

（1）对外部结构体变量进行初始化

在对这种结构体变量赋初值时，C 编译程序按每个成员在结构体中的顺序一一对应赋初值，不允许跳过前边的成员给后面的成员赋初值；但可以只给前面的若干个成员赋初值，对于后面未赋初值的成员，若为数值型，系统自动赋初值零，若为字符型，系统自动赋初值空。

例如，以下语句用于对 stud 变量的初始化：

```
struct student stud={"李明",'M',20,88,"98101"};
```

（2）对静态结构体变量进行初始化

例如，以下语句对 stud 静态变量进行初始化：

```
static struct student
{
    char name[12];
    char sex;
    int age;
    int score;
    char class[16];
} stud={"李明",'M',20,88,"98101"};
```

但以下赋初值的方式是错误的：

```
static struct student
{
    char name[12]="李明";
    char sex='M';
    int age=20;
    int score=88;
    char class[16]="98101";
} stud;
```

如果一个结构体类型内又嵌套另一个结构体类型，则对该结构体变量初始化时，仍按顺序写出各个初始值。

例如，以下代码定义了一个嵌套日期（date）类型的 teacher 类型，并对该类型的变量进行初始化：

```
struct date                                    /*定义 date 结构体*/
{
    int year, month, day;                      /*年，月，日*/
}
struct teacher
{
    char name[8];                              /*姓名*/
```

```
    struct date birthday;                    /*出生日期*/
    char depart[20];                         /*工作部门*/
} tech={"张强", 1958, 8, 20, "计算机系"};
```

10.1.3　结构体数组

1. 结构体数组的定义

有如下几种方法用于定义结构体数组。

（1）先进行结构体类型定义再定义结构体数组

例如，以下代码采用该方式定义了一个 student **结构体类型的数组** st**：**

```
struct stud
{
    char name[12];
    char sex;
    int age;
    int score;
    char class[16];
}
struct student st[40];
```

（2）同时进行结构体类型和结构体数组的定义

例如，以下代码采用该方式定义了一个 student **结构体类型的数组** st**：**

```
struct stud
{
    char name[12];
    char sex;
    int age;
    int score;
    char class[16];
} st[40];
```

（3）直接定义结构体数组而不需要定义结构体类型名

例如，以下代码采用该方式定义了一个 student **结构体类型的数组** st**：**

```
struct
{
    char name[12];
    char sex;
    int age;
    int score;
    char class[16];
} st[40];
```

2. 结构体数组的引用

对于结构体数组的引用就是指对结构体数组元素的引用。由于结构体数组元素相当于结构体变量，因此前面讨论的关于引用结构体变量的方法也同样适用于结构体数组元素。

（1）结构体数组元素中某一成员的引用

例如，在前面的 st 结构体数组定义后，st[2].age 表示 st 的第 3 个元素的 age 成员项；st[10].name 表示 st 的第 11 元素的 name 成员项。

（2）结构体数组元素的赋值

可以将一个结构体数组元素赋给同一结构体数组中的另一个元素，或者赋给同一类型的变量。例如，在前面的 st 结构体数组定义后，以下赋值语句都是合法的：

```
st[1]=st[5];
st[4]=st[39];
```

注意 结构体数组元素只能对单个成员项进行输入输出，而不能把结构体数组元素作为一个整体直接进行输入输出。

3. 结构体数组的初始化

C 语言规定只能对全局的或静态结构体数组进行初始化。给结构体数组赋初值的方式与数组赋初值的方式相同。只是由于数组中的每个元素都是一个结构体，因此要将其成员的值依次放在一对花括号中，以便区分各个元素。

例如，以下语句用于一个结构体数组的初始化：

```
struct depart
{
    int no;                          /*部门号*/
    char dname;                      /*部门*/
} dp[3]={ {3,"人事处"},{15,"财务处"},{8,"科技处"} };
```

10.1.4 结构体指针

结构体指针是一个指针变量，用来指向一个结构体变量，即指向该变量所分配的存储区域的首地址。结构体指针变量还可以用来指向结构体数组中的元素。结构体指针的一般定义方式如下：

```
struct <结构体类型名>  *<结构体指针名>;
```

结构体指针是一个指针变量，它与以前介绍的各种指针在特性和使用方法上完全相同。结构体指针的运算也是按照 C 语言的地址计算规则进行的。例如，结构体指针加一将指向内存中下一个结构体，结构体指针自身地址值的增加量取决于它所指向的结构体的数据长度（可用 sizeof()函数获取）。

1. 结构体变量指针

结构体变量指针是指向一个结构体变量的一个指针。
例如，有如下定义：

```
struct student stud, *p=&stud;  /*这里的 student 是前面定义的结构体类型*/
```

其中，p 是 student 结构体指针，通过 p=&stud 赋值语句使 p 指向 stud 结构体变量。

注意　使用结构体变量指针应考虑以下几点：

（1）p 不是结构体变量，因此不能写成 p.age，必须加上圆括号写成(*p).age。为此，C 语言中引入了一个指向运算符“ -> ”连接指针变量与其指向的结构体变量的成员，如：

```
(*p).age
```

改写为

```
p->age
```

（2）p 只能指向一个结构体变量，如：

```
p=&stud;
```

而不能指向结构体变量中的一个成员，所以 p=&stud.age 是错误的。

（3）指向运算符“ -> ”的优先级最高，如：
p->age+1 相当于(p->age)+1，即返回 p->age 之值加 1 的结果；
p->age++相当于(p->age)++，即将 p 所指向的结构体的 age 成员值自增 1。

2. 结构体数组指针

一个指针变量可以指向结构体数组，即将该数组的起始地址赋值给该指针变量。
例如，有如下定义：

```
struct student stud[40],*p;       /*这里的 student 是前面定义的结构体类型*/
p=stud;
```

其中，p 是 student 结构体指针，通过 p=stud 赋值语句使 p 指向 stud 结构体数组的第 1 个元素，此时 p->name 等价于 stud[0].name。

注意　使用结构体数组指针应考虑以下几点：

（1）当执行 p=stud 语句后，指针 p 指向 stud 数组的第一个元素；当进行 p++后，表示指针 p 指向下一个元素的起始地址。但要注意下面两种操作的不同之处：

- (++p)->age 先将 p 自增 1，然后取得它指向的元素中的成员 age 的值，即 stud[0].age。

- (p++)->age 先取得 p->age 的值，然后再进行 p 自增 1，指向下一个元素 stud[1]。

（2）p 只能指向该结构体数组的一个元素，然后用指向运算符“ -> ”取其成员之值，而不能直接指向一个成员。例如，p->age 是错误的。

10.1.5 函数之间结构体变量的数据传递

函数之间结构体变量的数据传递有以下几种方式。

1. 结构体复制方式

在函数之间以数据复制方式（即传数据）传递结构体变量时，在调用函数中实参使用结构体变量名，被调用函数的相应形参也是具有相同结构体类型的结构体变量。在函数调用的程序控制转移的同时，实参的结构体变量赋给形参的结构体变量，所以作为形参的结构体变量的数据变化不会回传给实参的结构体变量。

例如，有以下程序：

```
#include <stdio.h>
#include <string.h>
struct stud
{
    int no;
    char name[10];
};
void fun(struct stud st)
{
    st.no=10;strcpy(st.name,"李明");
}
main()
{
    struct stud st1;
    st1.no=1;strcpy(st1.name,"张华");
    fun(st1);
    printf("%d,%s\n",st1.no,st1.name);
}
```

其中，main()调用 fun()函数时采用结构体复制方式，所以 st1 仍保留原来的值，即程序运行时输出：1,张华。

2. 结构体地址传递方式

结构体地址传递方式是把结构体变量的存储地址作为实参传递给形参，而形参接收该地址值。然后在函数中通过这个结构体指针来处理结构体中的各项数据。采用这种函数间的传递方式，由于同一地址的数据相同，所以作为形参的结构体变量的数据变化会回传给实参的结构体变量。

例如，有以下程序：

```
#include <stdio.h>
#include <string.h>
struct stud
{
    int no;
    char name[10];
};
void fun(struct stud *st)
{
    st->no=10;strcpy(st->name,"李明");
}
main()
{
    struct stud st1;
    st1.no=1;strcpy(st1.name,"张华");
    fun(&st1);
    printf("%d,%s\n",st1.no,st1.name);
}
```

其中，main()**调用** fun()**函数时采用结构体地址传递方式，所以** st1 **变成了新的值，即程序运行时输出：**10,**李明。**

3. **结构体数组在函数间传递**

当需要把多个结构体作为一个参数向函数传递时，应该把它们组织成结构体数组。函数间传递结构体数组时，一般采用地址传递方式，即把结构体数组的存储首地址作为实参。在调用的函数中，用同样结构体类型的结构体指针作为形参接收传递的地址值。

10.1.6 结构体嵌套

当一个结构体成员项是结构体时，就形成了结构体嵌套。在数据处理中有时要使用结构体嵌套处理组织结构比较复杂的数据集合。

例如，有以下程序：

```
#include <stdio.h>
#include <string.h>
struct date
{
    int year,month,day;
};
struct worker
{
    int no;
    char name[8];
```

```
    struct date birthday;
}
main()
{
    struct worker work,*p;
    work.no=100;
    strcpy(work.name,"王华");
    work.birthday.year=1965;
    work.birthday.month=10;
    work.birthday.day=1;
    p=&work;
    printf("编号:%d    姓名:%s\n",p->no,p->name);
    printf("出生日期:%4d 年%d 月%d 日\n",
           p->birthday.year,p->birthday.month,p->birthday.day);
}
```

该程序定义的 worker 结构体中包含一个 date 结构体成员，从而构成结构体嵌套。程序执行结果如下：

```
编号:100    姓名:王华
出生日期:1965 年 10 月 1 日
```

10.1.7 链表

链表是指将若干个数据项按一定的原则连接起来的表。链表中每一个数据（可能包含多个成员项）称为结点。链表的连接原则是：前一个结点指向下一个结点；只有通过前一个结点才能找到下一个结点。本节都是讨论单链表的运算，有关双链表和循环链表参见数据结构相关教程。

为了实现链表操作，需要动态地分配和释放内存空间。为此 C 语言提供了两个动态分配和释放内存空间的函数：malloc()和 free()。它们的说明在 stdlib.h（Turbo C 系统）或 malloc.h（VC++系统）中。

malloc()函数的一般使用格式如下：

```
void *malloc(size_t size);
```

该函数用来分配 size 个字节的存储区域，返回一个指向存储区域首地址的基类型为 void 的指针。若没有足够的内存单元供分配，函数返回空指针（NULL）。

由于该函数返回的指针为 void *(无值型)，故在调用函数时，必须使用强制类型转换将其转换成所需的类型。例如，要使 p 指向一个 float 类型的存储单元：

```
float *p;
p=(float *)malloc(sizeof(float));
```

free()函数的一般使用格式如下：

```
void free(void *p);
```

其中，指针变量 p 必须指向由动态分配函数 malloc()分配的地址。free()函数将指针 p 所指的存储空间释放，使这部分空间可以由系统重新分配。

本节以学生链表为例讨论单链表的一般运算算法的实现过程。学生链表的结构如下：

```
struct Stud                    /*定义链表结构*/
{
    int no;                    /*学号*/
    int score;                 /*分数*/
    struct Stud *next;         /*指针域*/
};
```

为了方便，链表带有一个头结点（实际上，对于不带头结点的链表，可以为它临时创建一个头结点），如图 10.1 所示是一个带头结点*head 的单链表，真正存放数据是*head 之后的结点开始的，所以通常只对存放实际数据的结点进行编号，在图 10.1 中，（1，90）的结点为第 1 个结点，（2，72）的结点为第 2 个结点，（3，85）的结点为第 3 个结点，也是最后结点，其 next 域置为“ ∧ ”，表示空。

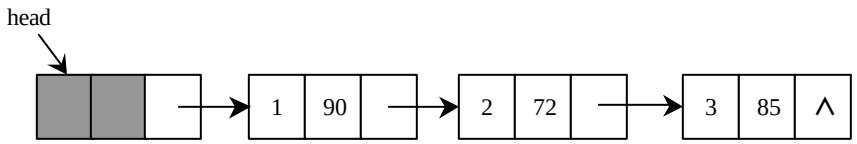

图 10.1　带头结点的单链表

1. 建立链表

以下函数 create()用于建立一个学生链表。用户输入学号和分数，采用后插法建立结点，直到输入 0,0 为止，该函数返回所建链表的头结点指针。

```
struct Stud *create()
{
    struct Stud *head,*p,*q;
    int n,s;
    head=(struct Stud *)malloc(sizeof(struct Stud));
    head->next=NULL;
    p=head;
    while (1)
    {
        printf("学号 分数:");
        scanf("%d%d",&n,&s);
        if (n==0 && s==0)                    /*输入 0,0 时退出循环*/
            break;
        else
        {
            q=(struct Stud *)malloc(sizeof(struct Stud));
```

```
            q->no=n;q->score=s;q->next=NULL;
            p->next=q;p=q;
        }
    }
    return head;                                    /*返回头结点指针*/
}
```

2. **显示链表**

以下函数 disp()**用于显示头结点为***h **的链表的所有结点数据域。**

```
void disp(struct Stud *h)
{
    struct Stud *p=h->next;                         /*p 指向链表的第一个结点*/
    printf("输出链表:");
    if (p==NULL)
        printf("空表\n");
    else
    {
        while (p->next!=NULL)
        {
            printf("%d %d,",p->no,p->score);
            p=p->next;
        }
        printf("%d %d\n",p->no,p->score);           /*输出最后一个结点*/
    }
}
```

3. **查找结点**

以下函数 locate()**用于在头结点为***h **的链表中查找学号等于** n **的结点，并返回该结点的指针。**

```
struct Stud *locate(struct Stud *h,int n)
{
    struct Stud *p=h->next;
    while (p!=NULL && p->no!=n)
        p=p->next;
    return p;
}
```

4. **插入结点**

以下函数用于在头结点为*h **的链表中的第** i **个结点之后插入一个结点（其** no **域值为** n**，**score **域值为** s**）。**

```
void insnode(struct Stud *h,int i,int n,int s)
{
```

```
    int j;
    struct Stud *p,*q;
    p=(struct Stud *)malloc(sizeof(struct Stud));        /*创建新结点*/
    p->no=n;p->score=s;p->next=NULL;
    if (i==0)                    /*i=0表示插入的结点作为该链表的第1个结点*/
    {
        p->next=h->next;
        h->next=p;
    }
    else
    {
        q=h->next;
        for (j=1;j<i && q!=NULL;j++)              /*找到第 i 个结点*q*/
            q=q->next;
        if (q!=NULL)                              /*将*p 插到*q 之后*/
        {
            p->next=q->next;
            q->next=p;
        }
        else printf("i 值太大\n");
    }
}
```

5. **删除结点**

以下函数用于在头结点为*h 的链表中删除第 i 个结点。

```
void delnode(struct Stud *h,int i)
{
    struct Stud *p,*q;
    int j;
    q=h;
    for (j=1;j<=i && q!=NULL;j++)                 /*找到第 i-1 个结点*q*/
        q=q->next;
    if (q!=NULL)                                  /*删除*q 之后的结点*/
    {
        p=q->next;                                /*p 指向要删除的结点*/
        q->next=p->next;                          /*从链表中删除结点*p*/
        free(p);
    }
    else printf("i 值太大\n");
}
```

6. **释放链表**

以下函数用于释放头结点为*h 的链表。

```
void freelist(struct Stud *h)
{
```

```
    struct Stud *p=h,*q=h->next;
    while (q!=NULL)
    {
        free(p);
        p=q;q=q->next;
    }
    free(q);                          /*释放最后一个结点*/
}
```

有关采用结构体类型构造二叉树的内容请参见数据结构相关教程。

10.1.8 共用体

在 C 语言中，可以定义不同类型的数据使用共用的存储区域，这种形式的数据构造类型称为共用体。共用体类型和结构体类型在定义、说明和引用形式很相似，但它们在存储空间的占用分配上有本质区别。

1. 共用体类型的定义

共用体类型的定义的一般格式如下：

```
union <共用体类型名>
{
    <数据类型>  <成员名 1>;
    <数据类型>  <成员名 2>;
     …
    <数据类型>  <成员名 n>;
};
```

该定义确定了共用体的组织形式，同时指出了组成共用体的成员具有的数据类型。例如，以下语句定义了一个共用体 st：

```
union st
{
    int n;
    char c;
    float f;
}
```

这表示整型成员 n、字符型成员 c 和单精度型 f 共享一段存储空间。

2. 共用体变量的定义

共用体变量的定义和结构体变量的定义相似，常用的格式是：

```
union <共用体类型名> <共用体变量表>;
```

例如，以下语句定义了上述共用体类型 st 的两个变量：

```
union st s1,s2;
```

注意 共用体变量和结构体变量的区别如下：

- 不能对共用体变量赋值，不能通过引用变量名来得到一个值，也不能在定义共用体变量时对它进行初始化。
- 共用体变量中的所有成员共享一段公共存储区，所以共用体变量所占内存字节数与其成员中占字节数最多的那个成员相等；而结构体变量中的每个成员分别占有独立的存储空间，所以结构体变量所占内存字节数是其成员所占字节数的总和。
- 由于共用体变量中的所有成员共享存储空间，因此变量中的所有成员的首地址相同，而且变量的地址也就是该变量成员的地址。
- 共用体变量中起作用的成员是最后一次存放的成员，在存入一个新的成员后原有的成员就失去作用。
- 不能把共用体变量作为函数参数，但可以使用指向共用体变量的指针作为函数参数。

3. **共用体变量的引用**

共用体变量的引用方式如下：

<共用体变量名>.<成员名>

例如，以下引用语句都是合法的：

```
union st s1;                    /*st 是前面定义的共用体类型*/
s1.n=100;
s1.c='A';
s2.f=1.2345;
```

共用体指针变量的引用方式如下：

<共用体指针变量名>-><成员名>

例如，以下引用语句都是合法的：

```
union st s1,*ps=&s1;            /*st 是前面定义的共用体类型*/
ps->n=100;
ps->c='A';
ps->f=1.2345;
```

10.1.9 枚举类型

枚举是一个命名为整型常量的集合。
枚举的定义格式和结构体定义格式相似：

```
enum <枚举类型名> { <枚举元素表> } <枚举变量表>;
```

其中，<枚举元素表>由一系列符号组成，每个符号都表示一个整数值，且可用在任意的整型表达式中。

例如，以下语句定义了一个 week_day 的枚举类型和一个具有该类型的枚举变量 week：

```
enum week_day {Mon, Tue, Wed, Thu, Fri, Sat, Sun} week;
```

枚举元素是按常量处理的，如果没有进行初始化，第一个枚举元素的值为 0，第二个枚举元素的值 1，依此类推。对上述枚举变量 week 来说，如下语句：

```
week=Fri;
printf("%d", week);
```

显示结果是 4。

通过枚举类型中的枚举元素表，使得相应枚举变量只能取枚举元素表中的某个元素，而不能是其他值。如不能把整数直接赋给枚举变量。

例如，以下程序的功能是假设今天是星期天，求四天之后是星期几。

```
#include <stdio.h>
main()
{
    enum {sun,mon,tue,wed,thu,fri,sat} day;
    char weekday[7][7]={"星期天", "星期一", "星期二", "星期三",
                        "星期四", "星期五", "星期六"};
    day=sun;
    printf("今天是%s,四天后是%s\n", weekday[day], weekday[(day+4)%7]);
}
```

本程序的执行结果如下：

```
今天是星期天，四天后是星期四
```

10.1.10 用户定义类型

C 语言允许用 typedef 说明一种新的类型名，其一般格式为：

```
typedef  <类型名 1>  <类型名 2>;
```

其中，<类型名 1>是系统提供的标准类型名或已经定义过的其他类型名，<类型名 2>就是用户定义的类型名。例如：

```
typedef int INTEGER;                /*将 INTEGER 定义成整型 int*/
typedef struct
{
    int no;
    char *name;
} PERSON;                           /*将 PERSON 定义成该结构体类型*/
typedef char NAME[10];              /*定义 NAME 为字符型数组类型名*/
```