数据结构+C 语言程序设计

数据结构

热 点 分 析

数据结构主要研究数据、数据之间的关系(逻辑结构),数据与关系在内存中如何表示(存储结构)以及操作。该部分内容在事业单位招聘考试中是重点考查内容,也是难点之一。在复习备考当中,考生要熟悉基本数据结构,如链表、栈、队列、树、图等的基本概念及其特点;掌握数据结构中常用的排序、查找算法的思想及应用;掌握使用 C 语言实现基本数据结构的定义及其操作的方法以及其他重要算法。

9.1 数据

9.1.1 基本概念和术语

1.数据

数据(Data)是信息的载体。它能够被计算机识别、存储和加工处理,是计算机程序加工的原料。随着计算机应用领域的扩大,数据的范畴包括整数、实数、字符串、图像和声音等。

2.数据元素

数据元素(Data Element)是数据的基本单位。数据元素也称元素、结点、顶点、记录。一个数据元素可以由若干个数据项(也可称为字段、域、属性)组成。数据项是具有独立含义的最小标识单位。

3.数据结构

数据结构(Data Structure)指的是数据之间的相互关系,即数据的组织形式。

(1)数据结构的内容

数据的逻辑结构(Logical Structure),即数据元素之间的逻辑关系。数据的逻辑结构是从逻辑关系上描述数据,与数据的存储无关,是独立于计算机的。数据的逻辑结构可以看作是从具体问题抽象出来的数学模型。

数据的存储结构(Storage Structure),即数据元素及其关系在计算机存储器内的表示。数据的存储结构是逻辑结构用计算机语言的实现(亦称为映象),它依赖于计算机语言。对机器语言而言,存储结构是具体的。一般只在高级语言的层次上讨论存储结构。

数据的运算,即对数据施加的操作。数据的运算定义在数据的逻辑结构上,每种逻辑结构都有一个运算的集合。最常用的检索、插入、删除、更新、排序等运算实际上只是在抽象的数据上所施加的一系列抽象的操作。所谓抽象的操作,是指我们只知道这些操作是“做什么”,而无需考虑“如何做”。只有确定了存储结构之后,才考虑如何具体实现这些运算。

(2)数据的逻辑结构分类

在不产生混淆的前提下,常将数据的逻辑结构简称为数据结构。数据的逻辑结构有线性结构和非线性结构两大类。

①线性结构的逻辑特征是:若结构是非空集,则有且仅有一个开始结点和一个终端结点,并且所

有结点都最多只有一个直接前趋和一个直接后继。线性表是一个典型的线性结构，栈、队列、串等都是线性结构。

②非线性结构的逻辑特征是：一个结点可能有多个直接前趋和直接后继。数组、广义表、树和图等数据结构都是非线性结构。

9.1.2 算法的描述和分析

数据的运算通过算法描述，讨论算法是数据结构课程的重要内容之一。

1.算法及其特性

算法是任意一个良定义的计算过程。它以一个或多个值作为输入，并产生一个或多个值作为输出。

(1)一个算法可以被认为是用来解决一个计算问题的工具。

(2)一个算法是一系列将输入转换为输出的计算步骤。

例如，有这样一个排序问题：将一个数字序列排序为非降序。该问题的形式定义由满足下述关系的输入输出序列构成。

输入：数字序列<a1,a2,…,an>。

输出：输出序列的一个枚举<a1′,a2′,…,an′>，使得 a1′≤a2′≤…≤a3′。

对于一个输入实例<31,41,59,26,41,58>，排序算法应返回输出序列<26,31,41,41,58,59>。

在以上例子中，一个问题的输入实例是满足问题陈述中所给出的限制、为计算该问题的解所需要的所有输入构成的。

若一个算法对于每个输入实例均能终止并给出正确的结果，则称该算法是正确的。正确的算法解决了给定的计算问题。一个不正确的算法是指对某些输入实例不终止，或者虽然终止但给出的结果不是所希望得到的答案，一般只考虑正确的算法。

由此可见，算法具有以下特性：

①有穷性：一个算法必须总是在执行有穷步之后结束，且每一步都可在有穷时间内完成。

②确定性：算法中每一条指令必须有确切的含义，不存在二义性，且算法只有一个入口和一个出口。

③可行性：一个算法是可行的，即算法中描述的操作都是可以通过已经实现的基本运算执行有限次来实现的。

④输入：一个算法有零个或多个的输入，这些输入取自于某个特定的对象集合。

⑤输出：一个算法有一个或多个的输出，这些输出是同输入有着某些特定关系的量。

2.算法的描述

一个算法可以用自然语言、计算机程序语言或其他语言来说明，唯一的要求是该说明必须精确地描述计算过程。一般而言，描述算法最合适的语言是介于自然语言和程序语言之间的伪语言。它的控制结构往往类似于 Pascal、C 语言等程序语言，但其中可使用任何表达能力强的方法使算法表达更加清晰和简洁，而不至于陷入具体程序语言的某些细节。

从易于上机验证算法和提高实际程序设计能力考虑，本章采用 C 语言描述算法。

3.算法设计的要求

(1)正确性：算法应满足具体问题的需求。

(2)可读性：算法应该好读，以有利于阅读者对程序的理解。

(3)健壮性：算法应具有容错处理。

(4)效率与低存储量需求：效率指的是算法执行时间。存储量需求指算法执行过程中所需要的最

大存储空间。两者都与问题的规模有关。

4.算法的复杂性

(1)时间复杂性

用算法中基本操作的执行次数随着问题规模的增长而增长的趋势作为算法时间效率的度量。记作:T(n)=O(f(n)),表示随着问题规模 n 的增大,算法执行时间的增长率和 f(n)的增长率相同,称作算法的渐进时间复杂度,简称时间复杂度。

基本操作执行次数和包含它的语句的频度相同,语句的频度指的是该语句重复执行的次数。

例如,求 n 阶矩阵的元素之和。

```
Sum(int num[][],int n)
{
    int i,j,r=0;
    for(i=0;i<n;i++)
    for(j=0;j<n;j++)
    r+=num[i][j];/*基本操作*/
    return r;
}
```

上述程序基本操作的频度为 n*n,则时间复杂度为 $O(n*n)=O(n^2)$。

若基本操作的执行次数不确定时,则可以考虑平均时间复杂度或最坏情况下的时间复杂度。

假设算法的两部分是“顺序”关系,则该算法的算法时间复杂度为:

$T(n)=T_1(n)+T_2(n)=O(Max(f_1(n),f_2(n)))$,即大 O 求和准则。

若算法的两部分是“嵌套”关系,则该算法的算法时间复杂度为:

$T(n)=T_1(n)*T_2(n)=O(f_1(n)*f_2(n))$,即大 O 乘法准则。

(2)空间复杂性

用算法中所需的额外空间随问题规模 n 的增长而增长的趋势作为算法空间效率的度量。类似于算法的时间复杂度,空间复杂度记作 S(n)=O(f(n))。

算法的时间复杂度与空间复杂度都与问题的规模有关。有的情况下,算法的时间复杂度还随问题的输入数据集不同而不同。

9.2 线性表

线性结构是最简单且最常用的数据结构。线性表是一种典型的线性结构。

1.线性表的逻辑定义

线性表是由 n(n≥0)个数据元素(结点)$a_1,a_2,\cdots,a_n$ 组成的有限序列。

(1)数据元素的个数 n 定义为表的长度(n=0 时称为空表)。

(2)将非空的线性表(n>0)记作:$(a_1,a_2,\cdots,a_n)$。

(3)数据元素 $a_i(1\leqslant i\leqslant n)$只是一个抽象符号,其具体含义在不同情况下可以不同。

例如,英文字母表(A,B,…,Z)是线性表,表中每个字母是一个数据元素(结点);一副扑克牌的点数(2,3,…,10,J,Q,K,A)也是一个线性表,其中的数据元素是每张牌的点数。

2.线性表的逻辑结构特征

对于非空的线性表,有且仅有一个开始结点 a_1,该结点没有直接前趋,有且仅有一个直接后继 a_2;有且仅有一个终结结点 a_n,该结点没有直接后继,有且仅有一个直接前趋 a_{n-1};其余的内部结点 a_i(2

$\leqslant i \leqslant n-1$)都有且仅有一个直接前趋 a_{i-1} 和一个直接后继 a_{i+1}。

3.常见的线性表的基本运算

(1)InitList(L)

构造一个空的线性表 L,即表的初始化。

(2)ListLength(L)

求线性表 L 中的结点个数,即求表长。

(3)GetNode(L,i)

取线性表 L 中的第 i 个结点,这里要求 1≤i≤ListLength(L)。

(4)LocateNode(L,x)

在 L 中查找值为 x 的结点,并返回该结点在 L 中的位置。若 L 中有多个结点的值和 x 相同,则返回首次找到的结点位置;若 L 中没有值为 x 的结点,则返回一个特殊值表示查找失败。

(5)InsertList(L,x,i)

在线性表 L 的第 i 个位置上插入一个值为 x 的新结点,使得原编号为 $i,i+1,\cdots,n$ 的结点变为编号为 $i+1,i+2,\cdots,n+1$ 的结点。这里 $1\leqslant i\leqslant n+1$,而 n 是原表 L 的长度。插入后,表 L 的长度加 1。

(6)DeleteList(L,i)

删除线性表 L 的第 i 个结点,使得原编号为 $i+1,i+2,\cdots,n$ 的结点变成编号为 $i,i+1,\cdots,n-1$ 的结点。这里 $1\leqslant i\leqslant n$,而 n 是原表 L 的长度。删除后表 L 的长度减 1。

9.3 顺序存储结构

9.3.1 顺序表

1.顺序表的定义

顺序存储方法:即把线性表的结点按逻辑次序依次存放在一组地址连续的存储单元里的方法。

顺序表:用顺序存储方法存储的线性表简称为顺序表。

2.结点 a_i 的存储地址

不失一般性,设线性表中所有结点的类型相同,则每个结点所占用存储空间大小亦相同。假设表中每个结点占用 c 个存储单元,其中第一个单元的存储地址则是该结点的存储地址,并设表中开始结点 a_1 的存储地址(简称为基地址)是 $LOC(a_1)$,那么结点 a_i 的存储地址 $LOC(a_i)$可通过下式计算:

$$LOC(a_i)=LOC(a_1)+(i-1)*c,1\leqslant i\leqslant n$$

在顺序表中,每个结点 a_i 的存储地址是该结点在表中的位置 i 的线性函数。只要知道基地址和每个结点所占存储空间的大小,就可在相同时间内求出任一结点的存储地址。顺序存储结构是一种随机存取结构。

3.顺序表类型定义

```
#define ListSize 100 //表空间的大小可根据实际需要而定,这里假设为 100
typedef int DataType; //DataType 的类型可根据实际情况而定,这里假设为 int
typedef struct {
    DataType data[ListSize];//向量 data 用于存放表结点
    int length;//当前的表长度
}SeqList;
```

顺序表的存储位置与下标的对应关系如图 9—1 所示。

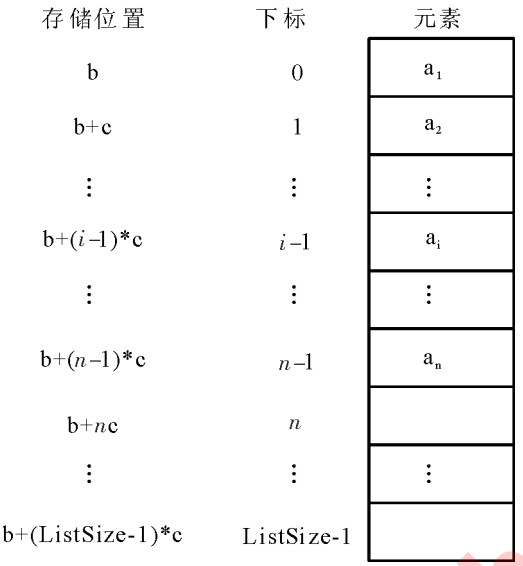


图 9—1　顺序表的存储位置与下标的对应关系

需要注意的是：

(1)用向量这种顺序存储的数组类型除存储线性表的元素外，顺序表还应该用一个变量来表示线性表的长度属性，因此用结构类型来定义顺序表类型。

(2)存放线性表结点的向量空间的大小 ListSize 应仔细选值，使其既能满足表结点的数目动态增加的需求，又不至于因预先定义过大而浪费存储空间。

(3)由于 C 语言中向量的下标从 0 开始，所以若 L 是 SeqList 类型的顺序表，则线性表的开始结点 a_1 和终端结点 a_n 分别存储在 L.data[0]和 L.data[L.length－1]中。

(4)若 L 是 SeqList 类型的指针变量，则 a_1 和 a_n 分别存储在 L→data[0]和 L→data[L→length－1]中。

4.顺序表的特点

顺序表是用向量实现的线性表，向量的下标可以看作结点的相对地址。因此，顺序表的特点是逻辑上相邻的结点，其物理位置亦相邻。

9.3.2　顺序表上的基本运算

1.表的初始化

```
void InitList(SeqList * L)
{//顺序表的初始化即将表的长度置为 0
    L→length=0;
}
```

2.求表长

```
int ListLength(SeqList * L)
{//求表长只需返回 L→length
    return L→length;
}
```

3.取表中第 i 个结点

```
DataType GetNode(SeqList *L,int i)
{//取表中第 i 个结点只需返回 L→data[i-1]即可;若不存在,提示 position error
    if(i<1||i> L→length-1)
    Error("position error");
    return L→data[i-1];
}
```

4.查找值为 x 的结点

```
int GetElem(SeqList *L,int i,dataType x)
{
    int j=0;
    LinkList *p=L;
    while(j<i&&p! =NULL)
        {
            j++;p=p→next;
        }
    if(p==NULL)
        return 0;
    else
        {
            x=p→data;
            return 1;
        }
}
```

5.插入

(1)插入运算的逻辑描述

线性表的插入运算是指在表的第 $i(1\leqslant i\leqslant n+1)$个位置上,插入一个新结点 x,使长度为 n 的线性表:

$$(a_1,\cdots,a_{i-1},a_i,\cdots,a_n)$$

变成长度为 n+1 的线性表:

$$(a_1,\cdots,a_{i-1},x,a_i,\cdots,a_n)$$

注意:

①由于向量空间大小在声明时确定,当 L→length≥ListSize 时,表空间已满,不可再做插入操作;

②当插入位置 i 的值为 $i>n$ 或 $i<1$ 时为非法位置,不可做正常插入操作。

(2)顺序表插入操作过程

在顺序表中,结点的物理顺序必须和结点的逻辑顺序保持一致,因此必须将表中位置为 $n,n-1,\cdots,i$ 上的结点,依次后移到位置 $n+1,n,\cdots,i+1$ 上,空出第 i 个位置,然后在该位置上插入新结点 x。仅当插入位置 $i=n+1$ 时,才无须移动结点,直接将 x 插入表的末尾。

(3)具体算法描述

```
void InsertList(SeqList *L,DataType x,int i)
{//将新结点 x 插入 L 所指的顺序表的第 i 个结点 ai 的位置上
```

```
int j;
if (i<1||i>L→length+1)
    Error("position error");//非法位置,退出运行
if (L→length>=ListSize)
    Error("overflow"); //表空间溢出,退出运行
for(j=L→length-1;j>=i-1;j--)
    L→data[j+1]=L→data[j];//结点后移
L→data[i-1]=x; //插入 x
L→Length++; //表长加 1
}
```

(4)算法分析

①问题的规模。表的长度 L→length(设值为 n)是问题的规模。

②移动结点的次数由表长 n 和插入位置 i 决定。算法的时间主要花费在 *for* 循环中的结点后移语句上。该语句的执行次数是 $n-i+1$。

a.当 $i=n+1$:移动结点次数为 0,即算法在最好情况下时间复杂度是 O(1)。

b.当 $i=1$:移动结点次数为 n,即算法在最坏情况下时间复杂度是 O(n)。

③移动结点的平均次数为 $E_{is}(n)$。

$$E_{is}(n)=P_i(n-i+1)$$

其中,在表中第 i 个位置插入一个结点的移动次数为 $n-i+1$;p_i 表示在表中第 i 个位置上插入一个结点的概率。不失一般性,假设在表中任何合法位置($1\leqslant i\leqslant n+1$)上的插入结点的机会是均等的,则:

$$p_1=p_2=\cdots=p_{n+1}=1/(n+1)$$

因此,在等概率插入的情况下,即在顺序表上进行插入运算,平均要移动一半结点。

6.删除

(1)删除运算的逻辑描述

线性表的删除运算是指将表的第 i($1\leqslant i\leqslant n$)个结点删去,使长度为 n 的线性表:

$$(a_1,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$$

变成长度为 n-1 的线性表:

$$(a_1,\cdots,a_{i-1},a_{i+1},\cdots,a_n)$$

注意,当要删除元素的位置 i 不在表长范围(即 $i<1$ 或 $i>$L→length)时,为非法位置,不能做正常的删除操作。

(2)顺序表删除操作过程

在顺序表上实现删除运算必须移动结点,才能反映出结点间的逻辑关系的变化。若 $i=n$,则只要简单地删除终端结点,无须移动结点;若 $1\leqslant i\leqslant n-1$,则必须将表中位置 $i+1,i+2,\cdots,n$ 的结点,依次前移到位置 $i,i+1,\cdots,n-1$ 上,以填补删除操作造成的空缺。

(3)具体算法描述

```
void DeleteList(SeqList *L,int i)
{//从 L 所指的顺序表中删除第 i 个结点 ai
int j;
if(i<1||i>L→length)
    Error("position error");        //非法位置
```

```
for(j=i;j<=L→length-1;j++)
    L→data[j-1]=L→data[j]; //结点前移
L→length--;              //表长减小
}
```

(4)算法分析

①结点的移动次数由表长 n 和位置 i 决定：

a.$i=n$ 时，结点的移动次数为 0，即为 O(1)。

b.$i=1$ 时，结点的移动次数为 $n-1$，算法时间复杂度是 O(n)。

②移动结点的平均次数 $E_{de}(n)$。

$$E_{de}(n)=P_i(n-i)$$

其中，删除表中第 i 个位置结点的移动次数为 $n-i$；p_i 表示删除表中第 i 个位置上结点的概率。不失一般性，假设在表中任何合法位置($1\leqslant i\leqslant n$)上的删除结点的机会是均等的，则：

$$p_1=p_2=\cdots=p_n=1/n$$

因此，在等概率插入的情况下，顺序表上做删除运算，平均要移动表中约一半的结点，平均时间复杂度也是 O(n)。

7.线性表顺序存储结构的特点

(1)线性表顺序存储结构的优点

①利用数据元素的存储位置表示线性表中相邻元素之间的前后关系，即线性表的逻辑结构与存储结构一致；

②在访问线性表时，可以利用上述的数学公式，快速计算出任何一个数据元素的存储地址。

(2)线性表顺序存储结构的缺点

①在做插入或删除元素的操作时，会产生大量的数据元素移动；

②对于长度变化较大的线性表，要一次性地分配足够的存储空间，但这些空间常常又得不到充分的利用；

③线性表的容量难以扩充。

9.4 链式存储结构

链式存储结构简称为链表，它是指用一组任意的存储单元(可以连续，也可以不连续)存储线性表中的数据元素。为了反映数据元素之间的逻辑关系，对于每个数据元素，不仅要表示它的具体内容，还要附加一个表示它的直接后继元素存储位置的信息，每个数据元素的两部分信息组合在一起被称为结点，如图 9—2 所示。其中，data 为数据域，用来存放结点数据元素的内容；next 为指针域(链域)，用来存放结点的直接后继的地址(或位置)。

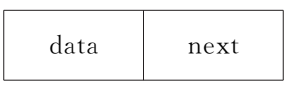


图 9—2 链表的结点结构

链表有线性链表(单链表)、循环链表、双向链表等。

链表的实现有动态链表和静态链表两种方式。

9.4.1 单链表

若链表的每一个结点只有一个链域，则将这种链表称为单链表，如图 9—3 所示。

1.单链表的图形描述

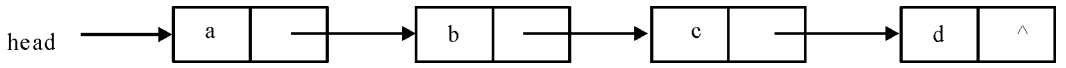


图 9—3　单链表的图形描述

(1)head 是头指针,它指向单链表中的第一个结点,这是单链表操作的入口点。

(2)最后一个结点没有直接后继结点,所以,它的指针域放入一个特殊的值 NULL。NULL 值在图中常用符号“∧”表示。

(3)单链表由表头唯一确定,因此单链表可以用头指针的名字来命名。若头指针名是 head,则把链表称为表 head。

为了简化对链表的操作,经常在链表的第一个结点之前附加一个结点,并称为头结点。这样可以免去对链表第一个结点的特殊处理。如图 9—4 所示。

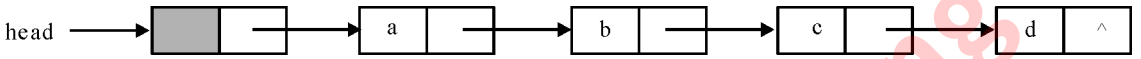


图 9—4　带头结点的单链表

头指针与头结点的区别为:头指针只相当于结点的指针域,头结点即整个线性链表的第一个结点,它的数据域可以放数据元素,也可以放线性表的长度等附加信息,也可以不存储任何信息。

2.动态单链表的类型定义

```
Typedef char DataType;//定义结点数据类型
Typedef struct LNode {
    DataType data;
    struct LNode  * next;
}LNode, * LinkList;
```

3.单链表基本操作的实现

以带头结点的单链表为例。

(1)初始化链表 L

```
int InitList(LinkList &L)
{
    //构造一个空链表 L,用 L 返回链表的头指针
    L→head=(LinkList)malloc(sizeof(LNode));
    //为头结点动态分配存储单元
    if (L→head)
        {
            L→head→next=NULL;
            return OK;
        }
    else
    return ERROR;
}
```

(2)插入数据元素

在链表 L 中第 i 个数据元素之前插入数据元素 e。

```
int ListInsert(LinkList &L,int i,DataType e)
```

```
{
    LNode *p,*s;
    int j;
    if(i<1||i>ListLength(L)+1)return ERROR;
    s=(LNode *)malloc(sizeof(LNode));
    if (s==NULL)return ERROR;
    s→data=e;
    for(p=L→head,j=0;p&&j<i-1;p=p→next,j++);
    //寻找第(i-1)个结点
    s→next=p→next;
    p→next=s;//将 s 结点插入
    return OK;
}
```

(3)删除数据元素

将链表 L 中第 i 个数据元素删除,并将其内容保存在 e 中。

```
int ListDelete(LinkList &L,int i,DataType &e)
{
    LNode *p,*s;
    int j;
    if(i<1||i>ListLength(L))return ERROR;
    //检查 i 值的合法性
    for(p=L→head,j=0;j<i-1;p=p→next,j++);
    //寻找第 i-1 个结点
    s=p→next;//用 s 指向将要删除的结点
    e=s→data;//将要删除数据的元素的值保存在 e 中
    p→next=s→next;//删除 s 指针所指向的结点
    free(s);
    return OK;
}
```

9.4.2 单循环链表

循环链表是一种头尾相接的链表。其特点是无需增加存储量,仅对表的链接方式稍作改变,即可使得表处理更加方便灵活。

在单链表中,将终端结点的指针域 NULL 改为指向表头结点或开始结点,就得到了单链形式的循环链表,简称为单循环链表。为了使空表和非空表的处理一致,循环链表中也可设置一个头结点,这样,空循环链表仅由一个自成循环的头结点表示,如图 9—5 所示。

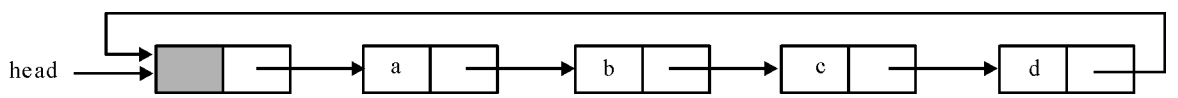


图 9—5 单循环链表

需要注意的是，很多情况下，表的操作在表的尾部位置上进行，此时头指针表示的单循环链表就显得不够方便，此时可以改用尾指针 rear 来表示单循环链表。

9.4.3 双向链表

以上讨论的链式存储结构的结点中只有一个指示直接后继的指针域，由此，从某个结点出发只能顺时针往后寻找其他结点。若要寻找结点的直接前趋，则需要从表头指针出发。为克服单链表这种单向性的缺点，可以利用双向链表。

顾名思义，在双向链表的结点中有两个指针域，其一指向直接后继，另一指向直接前趋。在 C 语言中可描述如下：

```
//线性表的双向链表存储结构
typedef struct DuLNode {
    ElemType data;
    struct DuLNode  * prior;
    struct DuLNode  * next;
}DuLNode, * DuLinkList;
```

和单链的循环表类似，双向链表也可以有循环表，如图 9—6 所示。

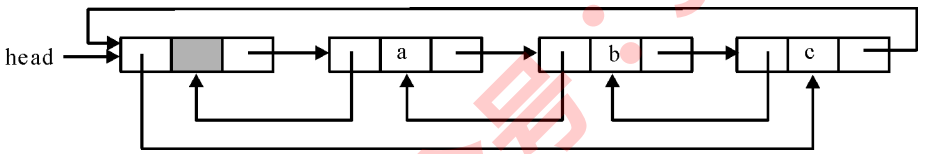


图 9—6　双向链表

设指针 p 指向某一结点，则双向链表结构的对称性可用下式描述：

$$(p\rightarrow prior)\rightarrow next = p = (p\rightarrow next)\rightarrow prior$$

即当前节点后继的前趋是自身，当前结点前趋的后继也是自身。

例如，删除双向链表的某个元素，关键操作如下：

```
p→next→prior=p→prior;
p→prior→next=p→next;
free(p);
```

9.5 栈和队列

栈和队列都是操作受限的线性表。栈的“后进先出”和队列的“先进先出”的特点，使它们成为程序设计中常用的数据结构。

9.5.1 栈

1.栈的定义

栈(stack)是限定仅在表尾进行插入和删除操作的线性表。表尾端称为栈顶(top)，表头端称为栈底(bottom)。不含元素的空表称为空栈。

假设栈 $S=(a_1,a_2,\cdots,a_n)$，则称 a_1 为栈底元素，a_n 为栈顶元素。栈中元素按 $a_1,a_2,\cdots,a_n$ 的次序进栈，退栈的第一个元素应为栈顶元素。换句话说，栈的修改是按“后进先出”的原则进行的，如图 9—7 所示。因此，栈又称为“后进先出”(Last In First Out)的线性表(简称 LIFO 结构)。

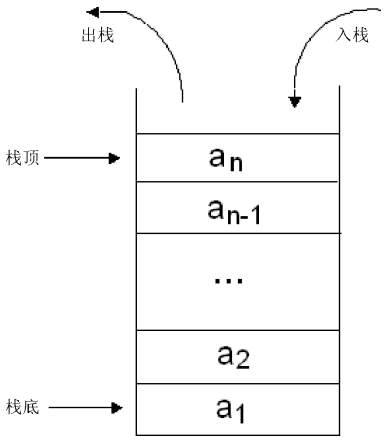


图 9—7　栈

2.栈的基本操作

栈上可以定义的基本操作如下所述。

InitStack(&S):构造一个空栈 S。

DestroyStack(&S):若栈 S 已存在,则将栈 S 销毁。

ClearStack(&S):若栈 S 已存在,将 S 清为空栈。

StackEmpty(S):若栈 S 已存在且为空栈,则返回 TRUE,否则返回 FALSE。

StackLength(S):若栈 S 已存在,返回 S 的元素个数,即栈的长度。

GetTop(S,&e):若栈 S 已存在且非空,用 e 返回 S 的栈顶元素。

Push(&S,e):若栈 S 已存在,插入元素 e 为新的栈顶元素。

Pop(&S,&e):若栈 S 已存在且非空,删除 S 的栈顶元素,并用 e 返回其值。

StackTraverse(S,visit()):若栈 S 已存在且非空,从栈底到栈顶依次对 S 的每个数据元素调用函数 visit()。一旦 visit()失败,则操作失效。

通常情况下,插入元素的操作称为入栈,删除栈顶元素的操作称为出栈。

3.栈的顺序存储

顺序栈,即栈的存储结构是顺序存储结构,利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素,同时附设指针 top 指示栈顶元素在顺序栈中的位置。C 语言中,顺序栈也可以采用静态分配和动态分配两种不同的方式。

由于栈在使用过程中所需最大空间的大小很难估计,因此,一般来说,在初始化设空栈时不应限定栈的最大容量。一个较合理的做法是:先为栈分配一个基本容量,然后在应用过程中,当栈的空间不够使用时再逐段扩大。为此,可设定两个常量,用于指定存储空间初始分配量和存储空间分配增量,并以下述类型说明作为顺序栈的定义。

```
typedef struct{
    SElemType * base
    SElemType * top;
    int stacksize;
}SqStack;
```

其中,stacksize 指示栈的当前可使用的最大容量。栈的初始化操作为:按设定的初始分配量进行第一次存储分配,base 可称为栈底指针,在顺序栈中,它始终指向栈底的位置,若 base 的值为 NULL,则表明栈结构不存在。top 为栈顶指针,其初值指向栈底,即 top==base 可作为栈空的标记,每当插入新

的栈顶元素时，指针 top 增 1；删除栈顶元素时，指针 top 减 1。因此，非空栈中的栈顶指针始终在栈顶元素的下一个位置上。如图 9—8 展示了顺序栈中数据元素和栈顶指针之间的对应关系。

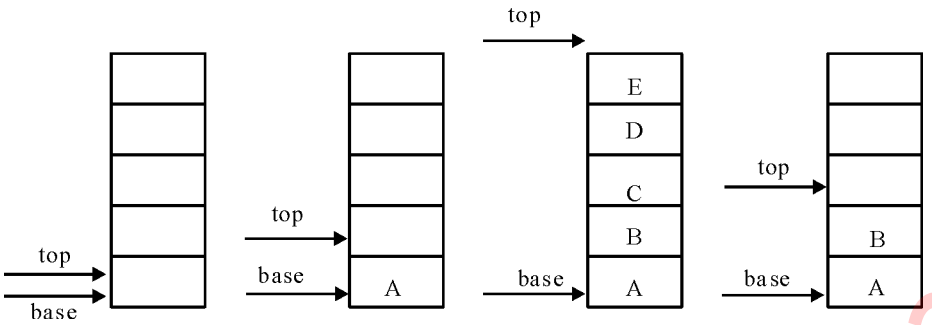


图 9—8　栈顶指针和栈中元素之间的关系

9.5.2　队列

1.队列的定义

和栈相反，队列(queue)是一种先进先出(First In First Out，FIFO)的线性表。它只允许在表的一端进行插入，而在另一端删除元素。这和我们日常生活中的排队是一致的，最早进入队列的元素最早离开。在队列中，允许插入的一端叫做队尾(rear)，允许删除的一端则称为队头(front)。假设队列为 $q=(a_1,a_2,\cdots,a_n)$，那么，a_1 就是队头元素，a_n 则是队尾元素。队列中的元素是按照 $a_1,a_2,\cdots,a_n$ 的顺序进入的，退出队列也只能按照这个次序依次退出，也就是说，只有在 $a_1,a_2,\cdots,a_{n-1}$ 都离开队列之后，a_n 才能退出队列。如图 9—9 所示。

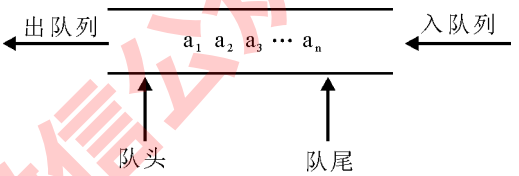


图 9—9　队列的示意图

2.队列的基本操作

队列上可以定义的基本操作如下。

InitQueue(&Q)：构造一个空队列 Q。

DestroyQueue(&Q)：若队列 Q 已存在，则销毁队列 Q。

ClearQueue(&Q)：若队列 Q 已存在，则将 Q 清为空队列。

QueueEmpty(Q)：若队列 Q 已存在，且 Q 为空队列，则返回 TRUE，否则返回 FALSE。

QueueLength(Q)：若队列 Q 已存在，则返回 Q 的元素个数，即队列的长度。

GetHead(Q,&e)：若 Q 为非空队列则用 e 返回 Q 的队头元素。

EnQueue(&Q,e)：若队列 Q 已存在，则为 Q 的队尾插入新元素 e。

DeQueue(&Q,&e)：若 Q 为非空队列，则删除 Q 的队头元素，并用 e 返回其值。

QueueTraverse(Q,visit())：若队列 Q 已存在且非空，visit()为元素的访问函数，则依次对 Q 的每个元素调用函数 visit()，一旦 visit()失败则操作失败。

3.队列的顺序存储

队列的顺序存储结构包括以下两部分：

(1)用于存放队列数据元素的一组连续存储单元。

(2)指示队头元素位置的队头指针 front,指示队尾元素位置的队尾指针 rear。

4.循环队列

循环队列将存储元素的存储空间看成是首尾相连的环。当队列空间中最后一个存储单元被占用后,只要队列前端还有可用空间,则新入队的元素将从头开始顺序向后存放。

```
#define QueueSize 100 //定义循环队列的最大容量
typedef struct {
    ElemType  * base;          //存储空间基址
    int front;                 //队头指针
    int rear;                  //队尾指针
    int queuesize;             //允许的最大存储空间
} CirQueue;
```

“少用一个存储单元”的方法判断队空或队满:设 Q 为 CirQueue 类型的变量,若 Q.rear＝Q.front,则表示该队列为空;若(Q.rear ＋ 1)% Q.queuesize＝＝Q.front,则表示队列已满。

5.循环队列的入队、出队算法

(1)入队

```
bool EnQueue(CirQueue &Q, ElemType e)
{
    //若队列未满,则将新元素 e 插入队尾
    if((Q.rear+1)% Q.queuesize==Q.front)
    return FLASE;                          //队满
    Q.base[Q.rear]=e;                      //队不满,将新元素 e 插入队尾
    Q.rear=(Q.rear+1)% Q.queuesize;        //修改队尾指针
    return TRUE;
}
```

(2)出队

```
bool DeQueue(Queue &Q, ElemType &e)
{
    if (Q.front == Q.rear)
    return FALSE;                          //队空
    e=Q.elem[Q.front];                     //队非空,将队头元素赋值给 e
    Q.front=(Q.front+1)% Q.queuesize;      //修改队头指针
    return TRUE;
}
```

9.6 数组

9.6.1 数组的定义

数组是由一组个数固定,类型相同的数据元素组成。每个元素在数组中的位置由它的下标决定。n 维数组的下标是 n 维空间的向量$(j_1,j_2,\cdots,j_n)$且具有上下界约束。数组一旦被定义,它的维数和维界就不再改变。因此,除了结构的初始化和销毁之外,数组只有存取元素和修改元素值的操作。

9.6.2 数组的基本操作

数组的基本操作有以下几种：

InitArray(&A,n,$bound_1$,…,$bound_n$)：若维数和各维长度合法，则构造相应的数组 A。

DestroyArray(&A)：若数组 A 存在，则将数组 A 销毁。

Value(A,&e,$index_1$,…,$index_n$)：若指定下标不越界，则用 e 返回该下标的数组元素。

Assign(&A,e,$index_1$,…,$index_n$)：若指定下标不越界，则将 e 的值赋给 A 指定下标的元素。

9.6.3 数组的顺序存储

一般采用顺序存储的方法来表示数组。

由于计算机的内存结构是一维的，因此用一维内存来表示多维数组，就必须按某种次序将数组元素排成一列序列，然后将这个线性序列存放在存储器中。

假设有一个 m 行 n 列的二维数组 A_{mn}，如图 9—10 所示，通常有以下两种顺序存储方式。

$$A_{mn}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

图 9—10　二维数组

(1)以行为主序的方式

将数组元素按行排列，第 i+1 个行向量紧接在第 i 个行向量后面。以二维数组为例，按行优先顺序存储的线性序列为：a_{11},a_{12},… ,a_{1n},a_{21},a_{22},…,a_{2n},…,a_{m1},a_{m2},…,a_{mn}。C 语言中，数组就是按以行为主序的方式存储的。

(2)以列为主序的方式

将数组元素按列向量排列，第 j+1 个列向量紧接在第 j 个列向量之后，A 的 m * n 个元素按列优先顺序存储的线性序列为：a_{11},a_{21},…,a_{m1},a_{12},a_{22},…,a_{m2},…,a_{1n},a_{2n},…,a_{mn}。FORTRAN 语言中，数组就是按以列为主序的方式存储的。

9.6.4 数组元素存储地址的计算

按以行为主序的方式和以列为主序的方式存储的数组，只要知道开始结点的存放地址(即基地址)、维数和每维的上、下界，每个数组元素所占用的单元数，就可以将数组元素的存放地址表示为其下标的线性函数。因此，数组中的任一元素可以在相同的时间内存取，即顺序存储的数组是一个随机存取结构。

例如，二维数组 A_{mn} 按“以行为主序的方式”存储在内存中，假设每个元素占用 k 个存储单元，则元素 a_{ij} 的存储地址应是数组的基地址加上排在 a_{ij} 前面的元素所占用的单元数。即：

$$LOC(a_{ij})=LOC(a_{11})+[(i-1)*n+j-1]*k$$

其中，$LOC(a_{11})$即是元素 a_{11} 的存储地址，也是二维数组 A_{mn} 的基址。

同样，三维数组以及多维数组的存储地址计算的原理与此类似。

注意：一般情况下，数组的下标从 0 开始。

9.7 树和二叉树

9.7.1 树的定义和基本术语

1.树的定义

树是 $n(n>=0)$个结点的有限集 T,它满足如下两个条件:

(1)有且仅有一个特定的称为根的结点;

(2)其他结点可分为 $m(m>=0)$个互不相交的子集 $T_1,T_2,T_3,\cdots,T_m$,其中每个子集又是一棵树,并称其为子树。T 为空时称为空树。

树中只有根结点没有前趋;其余结点都有且仅有一个前趋;树的所有结点,可以有零个或多个后继,如图 9—11 所示。

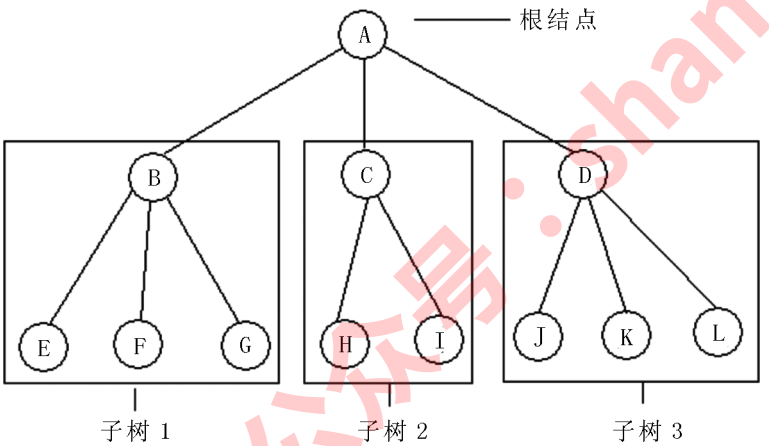


图 9—11 树的图形结构

2.基本术语

分支:根和子树根之间的连线。

结点的度:结点分支的个数定义。例如,结点 B 的度为 3,C 的度为 2,E 的度为 0。

树的度:树中所有结点度的最大值。

叶子结点:度为零的结点。

分支结点:度不为零的结点。

树的深度:树中叶子结点所在最大层次数。

森林:$m(m\geq 0)$棵互不相交的树的集合。

9.7.2 二叉树

1.二叉树的定义

二叉树是由 $n(n\geqslant 0)$个结点的有限集合构成,此集合或者为空集,或者由一个根结点及两棵互不相交的左右子树组成,并且左右子树都是二叉树。这是二叉树的递归定义。

二叉树可以是空集合,根可以有空的左子树或空的右子树。

二叉树与树的最主要的差别:二叉树结点的子树要区分左子树和右子树,即使只有一棵子树也要进行区分,必须说明它是左子树,还是右子树。

2.二叉树的形态

二叉树一共有 5 种形态，如图 9—12 所示。

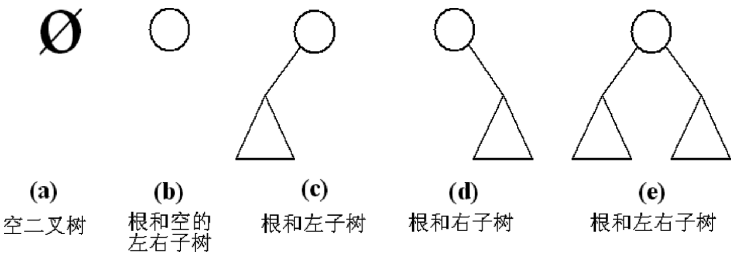


图 9—12　二叉树的形态

3.特殊的二叉树

满二叉树：如果深度为 k 的二叉树，有 2^k-1 个结点则称为满二叉树。满二叉树中所有的分支结点的度数都为 2，且叶子结点都在同一层次上。

完全二叉树：如果一棵二叉树只有最下一层结点数未达到最大，并且最下层结点都集中在该层的最左端，则称为完全二叉树。

满二叉树和完全二叉树如图 9—13 所示。

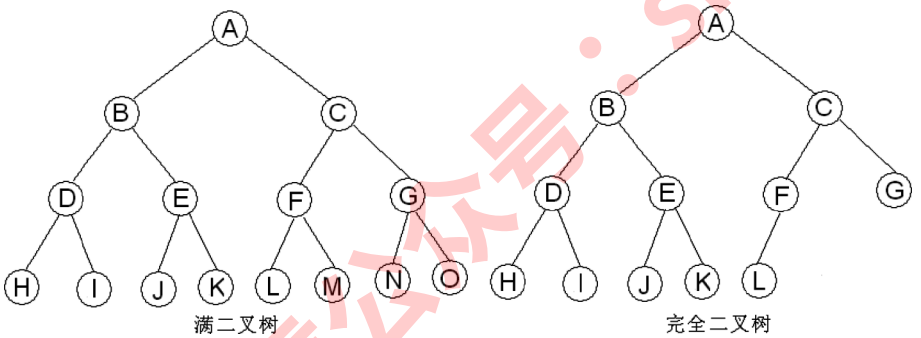


图 9—13　满二叉树和完全二叉树

4.二叉树的性质

假设根结点的层次为 1。

(1)在二叉树的第 i 层上至多有 2^{i-1} 个结点($i\geqslant 1$)。

(2)深度为 k 的二叉树中至多含有 2^k-1 个结点($k\geqslant 1$)。

(3)对任何一棵非空二叉树 T，如果其叶子结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。

(4)具有 n 个结点的完全二叉树的深度为$[\log_2 n]+1$。

(5)如果对一棵有 n 个结点的完全二叉树的结点按层序编号(从第一层到第$([\log_2 n]+1)$层，每层从左到右)，则对任一结点 $i(1\leqslant i\leqslant n)$，有：

①如果 $i=1$，则结点 i 无双亲，是二叉树的根；如果 $i>1$，则其双亲是结点$[i/2]$。

②如果 $2i>n$，则结点 i 为叶子结点，无左孩子；否则，其左孩子是结点 2i。

③如果 $2i+1>n$，则结点 i 无右孩子；否则，其右孩子是结点 $2i+1$。

5.二叉树的存储结构

(1)二叉树的顺序存储结构

顺序存储结构是指用一组连续的存储单元存储二叉树的数据元素。因此，必须把二叉树的所有结点安排成为一个恰当的序列，结点在这个序列中的相互位置能反映出结点之间的逻辑关系。则图 9—13 中完全二叉树的顺序存储结构表示如图 9—14 所示。

1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	D	E	F	G	H	I	J	K	L

图 9—14　完全二叉树的顺序存储结构表示

对于一般的二叉树，按完全二叉树形式补齐所缺少的结点，也可以用顺序存储结构存储，只是会浪费一些存储空间。

(2)二叉树的链式存储结构

对于二叉树来说，通常通过保存结点的左、右孩子或双亲结点的存储位置来表达元素之间的关系。存储二叉树经常用二叉链表法。

二叉链表法：二叉链表的每个结点包含一个数据域和两个指针域，数据域用于保存二叉树的数据元素，两个指针域分别保存左、右孩子结点的存储地址。若结点没有左右孩子，则其左、右指针域为空指针，如图 9—15 所示。

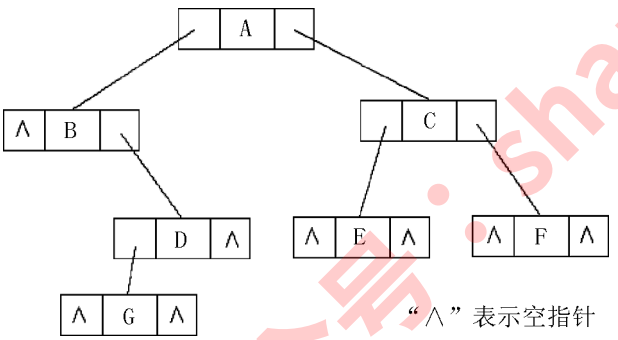


图 9—15　二叉链表

二叉树的链式存储结构一般需要给出根结点的指针，其存储结构表示如下：

```
Typedef struct BiTNode{
    ElemType data;
    struct BiTNode * lchild, * rchild;
} BiTNode, * BiTree;
```

6.二叉树的遍历

二叉树的遍历是指按某种搜索路径访问二叉树的每个结点，而且每个结点仅被访问一次。

二叉树两类遍历方式：深度优先遍历和按层遍历(即广度优先遍历)。

在深度优先遍历方法中，二叉树有 3 种遍历方式：先序遍历、中序遍历和后序遍历，如图 9—16 所示。

(1)先序遍历

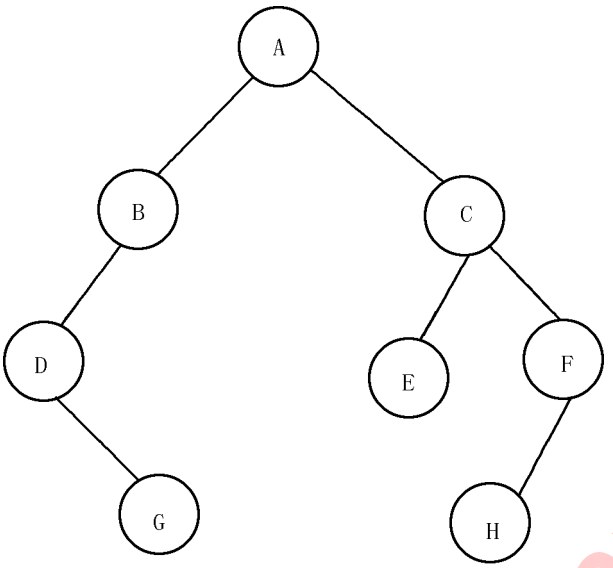
若二叉树为空，则空操作；否则，访问根结点，先序遍历左子树，先序遍历右子树。

(2)中序遍历

若二叉树为空，则空操作；否则，中序遍历左子树，访问根结点，中序遍历右子树。

(3)后序遍历

若二叉树为空，则空操作；否则，后序遍历左子树，后序遍历右子树，访问根结点。



先序遍历序列：ABDGCEFH
中序遍历序列：DGBAECHF
后序遍历序列：GDBEHFCA

图 9—16　二叉树的遍历

7.二叉树遍历的递归算法

二叉树的先序、中序、后序遍历递归算法类似，下面以先序遍历的递归算法为例。

```
void PreTraverse (BiTree BT)
{
    if (BT! =NULL)                    //若二叉树不为空
    {
        Visit(BT→data);
        PreTraverse(BT→Lchild);
        PreTraverse(BT→Rchild);
    }
}
```

8.线索二叉树

(1)线索二叉树

通过对二叉树进行先序、中序或后序遍历，即可得到相应的遍历序列。若能将序列中每个结点的前趋、后继结点的指针保存起来，则在下次再遍历二叉树时就可以根据所保存的前趋、后继结点指针对二叉树进行遍历。加上了前趋、后继指针(线索)的二叉树成为线索二叉树。

(2)线索二叉树的存储

为了能保存二叉树遍历过程中的线索信息，可增加标志域，如图 9—17 所示。

lchild	ltag	data	rtag	rchild

图 9—17　线索二叉树的存储结构

其中 ltag、rtag 为左、右标志域。当标志为 0 时，表示左、右指针域存储的是左、右孩子结点的指针；当标志为 1 时，表示左、右指针域存储的是前趋、后继结点的指针。

以上述结构构成的二叉链表，叫做线索链表。

二叉树的线索存储表示如下：

```
Typedef enum{Link, Thread}PointerTag;          //Link==0:指针;Thread==1:线索
Typedef struct BiThrNode
{
    ElemType data;
    struct BiTreeNode  * lchild, * rchild;
    PointerTag LTag, Rtag;
}BiTreeNode, * BiThrTree;
```

9.8 图

9.8.1 图的定义

图由顶点的有穷非空集合 V 和顶点的偶对(边)集合 E 组成，记为 G=(V,E)。其中，V 是结点(顶点)的有穷(非空)集合，E 是边的集合，是结点(顶点)的偶对。

9.8.2 图的基本概念

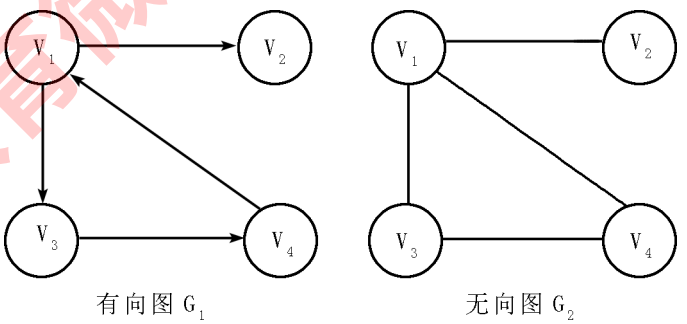
1.有向图与无向图

若图中的每条边都是有方向的，则称为有向图，如图 9—18 的左图所示。

有向图中，边是由两个顶点组成的有序对，有序对用尖括号表示。$<v_i,v_j>$表示一条有向边，v_i 是边的始点，v_j 是边的终点。顶点 v_i 邻接到 v_j，顶点 v_j 邻接到 v_i，边$<v_i,v_j>$与顶点 v_i，v_j 相关联。$<v_i,v_j>$和$<v_j,v_i>$代表两条不同的有向边。

若图中每条边都是无方向的，则称为无向图，如图 9—18 的右图所示。

无向图中，边是由两个顶点组成的无序对，无序对用圆括号表示。v_i 和 v_j 是相邻结点，(v_i,v_j)是与顶点 v_i 和 v_j 相关联的边。(v_i,v_j)和(v_j,v_i)代表同一条边。



有向图 G_1　　无向图 G_2

图 9—18　有向图与无向图

图 9—18 左边是有向图 G_1，其顶点集合和边集合分别如下：

$V(G_1)=\{v_1,v_2,v_3,v_4\}$

$E(G_1)=\{<v_1,v_2>,<v_1,v_3>,<v_3,v_4>,<v_4,v_1>\}$

图 9—18 右边是无向图 G_2，其顶点集合和边集合分别如下：

$V(G_2)=\{v_1,v_2,v_3,v_4\}$

$E(G_2)=\{(v_1,v_2),(v_1,v_3),(v_1,v_4),(v_3,v_4)\}$

若不考虑顶点到其自身的边，即若(v_i,v_j)或$<v_i,v_j>$是 E(G)中的边，且 $v_i \neq v_j$，且图中不允许一条边重复出现，则图中顶点数 n 和边数 e 满足以下关系：

①若 G 是有向图，则 $0 \leqslant e \leqslant n(n-1)$。

②若 G 是无向图，则 $0 \leqslant e \leqslant n(n-1)/2$。

有向完全图指的是有 $n(n-1)$条边的有向图。

无向完全图指的是有 $n(n-1)/2$ 条边的无向图。

在顶点个数相同的图中，完全图具有最多的边数。

2.度

度：与顶点 v 相关联的边数称为顶点的度，记为 D(v)。

入度：若 G 是一个有向图，则以顶点 v 为终点的边的数目称为 v 的入度，记为 ID(v)。

出度：若 G 是一个有向图，则以顶点 v 为始点的边的数目称为 v 的出度，记为 OD(v)。

对于有向图来说，其顶点 v 的度为其入度和出度之和，即 D(v)＝ID(v)＋OD(v)。

例如，图 9—18 中，有向图 G_1 的 V_1 顶点的入度为 1，出度为 2，度为 3。

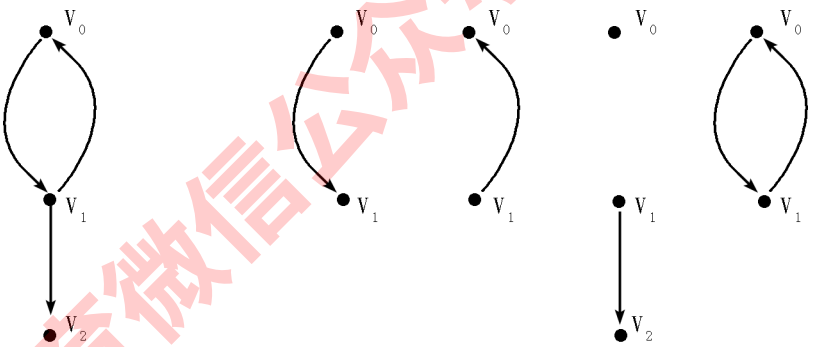
无论是有向图还是无向图，顶点数 n，边数 e 和度数 D(v)之间满足以下关系：

$$e=\sum_{i-1}^{n} D(v)/2$$

例如：图 9—18 中无向图 G_2，顶点数 n 为 4，边数 e 为 4，度数 D(v)之和为(3＋1＋2＋2)＝8。

3.子图

设有图 G＝(V,E)和 G′＝(V′,E′)，如果 V′是 V 的子集，E′是 E 的子集，则称 G′是 G 的子图。如图 9—19 所示。



有向图 G_3　　G_3 的子图

图 9—19　子图

4.路径

路径：图 G＝(V,E)中，若存在顶点序列 v_{i0}，v_{i1}，…，v_{in}，使得(v_{i0},v_{i1})，(v_{i1},v_{i2})，…，(v_{in-1},v_{in})都在 E 中(若是有向图，则使得$<v_{i0},v_{i1}>$，$<v_{i1},v_{i2}>$，…，$<v_{in-1},v_{in}>$都在 E 中)，则称从顶点 v_{i0} 到 v_{in} 存在一条路径。

路径长度：路径上的边数。

简单路径：路径上的顶点除 v_{i0} 和 v_{in} 可以相同外，其他顶点都不相同。

回路或环：起点和终点相同的简单路径。

例如，有向图 G_3 中顶点序列 v_0,v_1,v_0 是一长度为 2 的有向环。v_0,v_1,v_2 是一长度为 2 的简单路径。

5.根

有向图中 G，若存在一顶点 v，从该顶点可以有路径到图中其他所有顶点，则称此有向图为有根

图，顶点 v 称为图的根。有根图的根可能不唯一。

6.连通图

连通：无向图 G=(V,E)中，若从顶点 v_i 到 v_j 顶点有一条路径(从 v_j 到 v_i 也一定有一条路径)，则称 v_i 和 v_j 是连通的。

连通图：若无向图 G 中任意两个不同的顶点 v_i 和 v_j 都是连通的，则称 G 为连通图。

连通分量：无向图 G 中的最大连通子图称为 G 的连通分量。

强连通图：有向图 G=(V,E)中，若任意两个不同的顶点 v_i 和 v_j 都存在从 v_i 到 v_j 以及从 v_j 和 v_i 的路径，则称图 G 是强连通图。

强连通分量：有向图的最大强连通子图称为图的强连通分量。

强连通图只有一个强连通分量，就是其自身。非强连通的有向图有多个强连通分量。

7.带权图和网

带权图：若图的每条边都赋上一个权值，则该图被称为带权图。通常权是具有特定意义的数。

网：带权的连通图称为网。

8.生成树

生成树：对于连通的无向图或强连通的有向图，从任一顶点出发，或是对于有根的有向图，从图的根顶点出发，可以访问到所有的顶点。访问时经过的边加上所有顶点构成了图的一个连通子图，称为图的一个生成树。

最小生成树：是带权连通图的权最小的生成树。

生成树的特点如下：

①生成树是连通图的一个极小连通子图，它含有图中的全部 n 个顶点，但只有足以构成一棵树的 $n-1$ 条边；

②如在一棵生成树中增加一条边，则必定构成环；

③如在一棵生成树中去掉一条边，则连通图将变为不连通的；

④顶点数为 n，边数小于 $(n-1)$ 的无向图必定是不连通的；

⑤顶点数为 n，边数大于 $(n-1)$ 的无向图必定存在环；

⑥顶点数为 n，边数为 $(n-1)$ 的无向图不一定是生成树。

9.8.3 图的存储结构

图的结构较复杂，任意两个顶点间都可能存在联系，因而图的存储方法也很多，需要根据所求解问题的特点选择合适的存储结构。

常用的存储方法有：邻接矩阵表示法、邻接表表示法、邻接多重表表示法和图的十字链表等。

1.邻接矩阵表示法

邻接矩阵表示法也称数组表示法。它用一维数组存储图的顶点，通过图的邻接矩阵表示图中数据元素之间的关系。

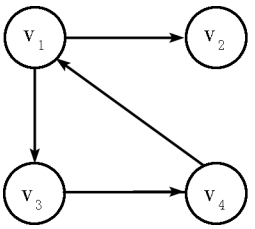
(1)图的邻接矩阵

邻接矩阵是指表示顶点间相邻关系的矩阵。

设 G=(V,E)为具有 n 个顶点的图，其邻接矩阵为具有以下性质的 n 阶矩阵：

$$A[i,j]=\begin{cases}1, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是图 G 的边}\\ 0, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是图 G 的边}\end{cases}$$

例如有向图 G_1 的邻接矩阵如图 9—20 所示。



$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

图 9—20　邻接矩阵

如果 G 是带权的图，w_{ij} 是边(v_i, v_j)或$<v_i, v_j>$的权，则其邻接矩阵定义如下：

$$A[i][j]=\begin{cases} w_{ij}, & \text{若}(v_i, v_j)\text{或}<v_i, v_j>\text{是 E[G]中的边} \\ 0\text{ 或}\infty, & \text{若}(v_i, v_j)\text{或}<v_i, v_j>\text{不是 E(G)中的边} \end{cases}$$

(2)邻接矩阵的存储结构

邻接矩阵存储结构需要存储一个包括 n 结点的顺序表来保存结点的数据或指向结点的数据指针，需要存储一个 n * n 的相邻矩阵来指示结点间的关系。

```
#define MAX_VERTEX_NUM 10
typedef char VexType;
typedef int VRType;
typedef struct
{
  VexType  vexs[MAX_VERTEX_NUM];                          //存储顶点数组
  VRType  arcs[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; //用二维数组存储邻接矩阵
  int  vexnum,arcnum;                                     //图的顶点数和弧数
}Graph;
```

对于有向图来说，需要 n * n 个单元来存储相邻矩阵，效率是 $O(n^2)$；对于无向图来说，由于相邻矩阵是对称的，所以只需存储相邻矩阵的下三角部分。

(3)邻接矩阵的特点

①无向图的邻接矩阵一定是一个对称方阵。

②无向图的邻接矩阵的第 i 行(或第 i 列)非零元素(或非∞元素)个数为第 i 个顶点的度 $D(v_i)$。

③有向图的邻接矩阵的第 i 行非零元素(或非∞元素)个数为第 i 个顶点的出度 $OD(v_i)$，第 i 列非零元素(或非∞元素)个数就是第 i 个顶点的入度 $ID(v_i)$。

④邻接矩阵表示图，很容易确定图中任意两个顶点之间是否有边相连。

2.邻接表表示法

邻接表是图的一种顺序存储与链式存储结合存储结构。在邻接表中，用一维数组存储图的顶点，用线性链表存储一个顶点的关联边，顶点 v 对应的链表存储了 v 所有的关联边。

(1)无向图的邻接表

每条边(v_i, v_j)在两个顶点 v_i、v_j 的链表中都占一个结点，因此，每条边在链表中存储两次。顶点 v_i 的度等于其对应链表中结点个数，等于其对应链表的长度。

无向图的邻接表如图 9—21 所示。

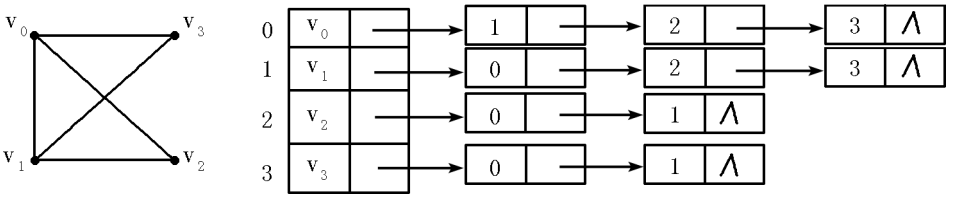


图 9—21　无向图的邻接表

(2)有向图的邻接表

顶点 v_i 的链表中每个结点对应的是以 v_i 为始点的一条边，因此，将有向图邻接表的边表称为出边表。顶点 v_i 的出度等于其对应链表中结点个数，等于其对应链表的长度。

有向图的邻接表如图 9—22 所示。

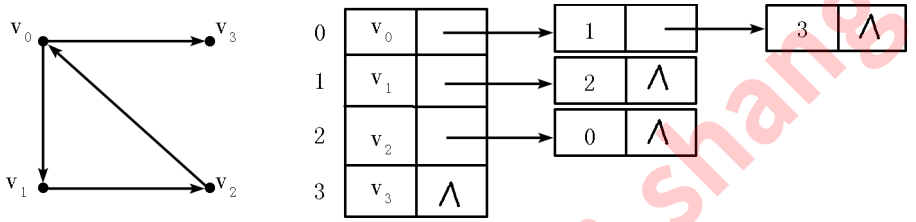


图 9—22　有向图的邻接表

(3)有向图的逆邻接表

逆邻接表也称入边表，表示方法与邻接表类似，只不过顶点 v_i 的链表中每个结点对应的是以 v_i 为终点的一条边。顶点 v_i 的入度等于其对应链表中结点个数，等于其对应链表的长度。

有向图的逆邻接表如图 9—23 所示。

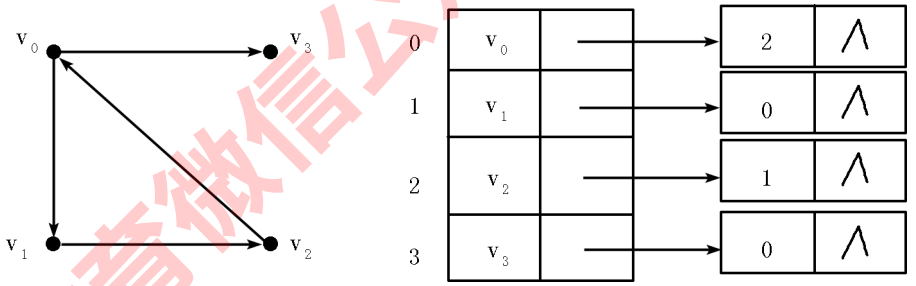


图 9—23　有向图的逆邻接表

9.8.4　图的遍历

按搜索路径的不同，图的遍历有深度优先遍历和广度优先遍历两种方法。

1.深度优先遍历

①访问图的某个指定出发顶点 v；

②若 v 有未被访问的邻接点，则任选一个顶点 w，从 w 出发继续对图进行深度优先遍历；否则返回到上一个出发点。

2.广度优先遍历

①访问图的某个指定出发顶点 v；

②访问 v 的所有未被访问的邻接点 $w_1, w_2, \cdots, w_k$；

③依次从这些邻接点出发，访问它们所有未被访问的邻接点；依此类推，直到访问图中所有已被

访问顶点的邻接点均被访问。

9.9 排序

9.9.1 排序的基本概念

1.基本概念

记录:由若干数据项组成。将唯一标识记录的数据项称为关键字。

排序:将记录按关键字递增(或递减)的次序排列起来。

稳定排序:在待排序记录中,任何两个关键字相同的记录,用某种排序方法排序后其相对位置保持不变,则称该排序方法是稳定排序,否则称为不稳定排序。

内排序:在排序过程中,文件放在内存中处理不涉及数据的内、外存交换的排序。常见的内排序方法有插入排序、交换排序、选择排序、归并排序和基数排序等。

外排序:待排序数据量大,无法一次全部读入内存,在排序时需要对外存进行访问的排序。

2.排序算法的基本操作

(1)比较关键字的大小。

(2)改变指向记录的指针或移动记录本身。

3.待排记录的存储方法

(1)顺序存储结构:用一维数组存储待排序记录。

(2)链式存储结构:用链表存储待排序记录。

(3)索引结构。

9.9.2 插入排序

插入排序的基本思想:依次将待排序记录按其关键字大小插入到前面已经排好序的有序子表中的适当位置,并使插入后的子表仍保持有序,直到全部记录插入完毕。

1.直接插入排序

(1)直接插入排序的思想

直接插入排序也称之为顺序插入排序,它使用顺序查找法确定插入位置。其核心思想是将一个记录插入到已经排好顺序的有序表中,从而得到一个新的有序表。

具体做法为:将待插入记录 R[i]的关键字从右向左依次与有序区中记录 R[j](j=i−1,i−2,…,1)的关键字进行比较。

①若 R[j]的关键字大于 R[i]的关键字,则将 R[j]后移一个位置;

②若 R[j]的关键字小于或等于 R[i]的关键字,则查找过程结束,j+1 即为 R[i]的插入位置。此时关键字比 R[i]的关键字大的记录均已后移,所以 j+1 的位置已经腾空,只要将 R[i]直接插入此位置即可完成一次直接插入排序。

例如,使用直接插入排序法对数据元素 49,38,65,97,76,13,27,$\overline{49}$进行排序,具体排序过程如下:

	R[0]	R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]	R[8]
		[49]	38	65	97	76	13	27	$\overline{49}$
j=2	(38)	[38	49]	65	97	76	13	27	$\overline{49}$

j=3	(65)	[38	49	65]	97	76	13	27	$\overline{49}$
j=4	(97)	[38	49	65	97]	76	13	27	$\overline{49}$
j=5	(76)	[38	49	65	76	97]	13	27	$\overline{49}$
j=6	(13)	[13	38	49	65	76	97]	27	$\overline{49}$
j=7	(27)	[13	27	38	49	65	76	97]	$\overline{49}$
j=8	(49)	[13	27	38	49	$\overline{49}$	65	76	97]

(2)直接插入排序的特点

①时间效率与初始序列有关。初始序列正向(递增)有序时,时间复杂度为 O(n);初始序列逆向(递减)有序时,时间复杂度为 O(n^2);无序时平均时间复杂度为 O(n^2)。

②空间复杂度为 O(1)。

③直接插入排序是稳定的排序方法。

④算法简单,适合记录基本正向有序或记录较少的情况。

2.希尔排序

(1)希尔排序基本思想

选取一个小于 n 的整数 d_1 作为第一个增量,把全部记录分成 d_1 个组。所有间隔为 d_1 的记录放在同一个组中,先在各组内进行直接插入排序;然后,取第二个增量 d_2($d_2<d_1$)重复上述的分组和排序,直至所取的增量 $d_k=1$($d_k<d_{t-1}<\cdots<d_2<d_1$),即所有记录放在同一组中进行直接插入排序为止。

例如,使用希尔排序法对数据元素 49,38,65,97,76,13,27,$\overline{49}$,55,10 进行排序,其增量序列的取值依次为:5,3,1。具体排序过程如下:

初始化状态:49,38,65,97,76,13,27,$\overline{49}$,55,10

当增量 $d_1=5$,分组情况为(49,13),(38,27),(65,$\overline{49}$),(97,55),(76,10)

排序结果为:13,27,$\overline{49}$,55,10,49,38,65,97,76

当增量 $d_2=3$,分组情况为:(13,55,38,76),(27,10,65),($\overline{49}$,49,97)

排序结果为:13,10,$\overline{49}$,38,27,49,55,65,97,76

当增量 $d_3=1$,所有元素属于同一个组,即(13,10,$\overline{49}$,38,27,49,55,65,97,76)

排序结果为:10,13,27,38,$\overline{49}$,49,55,65,76,97

(2)希尔排序的特点

①时间效率取决于增量序列的选择。若选择得当,效率高于直接插入排序,时间复杂度为O(nlogn)。

②空间复杂度为 O(1)。

③希尔排序是不稳定的排序方法。

④希尔排序适用于 n 较大的情况。

9.9.3 交换排序

交换排序的基本思想:两两比较待排序记录的关键字,发现两个记录的次序相反时即进行交换,直到没有反序的记录为止。

1.冒泡排序

(1)冒泡排序的基本思想

设想被排序的数组 R[1…n]垂直竖立,将每个数据元素看作有重量的气泡,根据轻气泡不能在重气泡之下的原则,从下往上扫描数组 R,凡扫描到违反本原则的轻气泡,就使其向上“漂浮”,如此反复进行,直至最后任何两个气泡都是轻者在上,重者在下为止。冒泡法排序最多做 $n-1$ 趟排序。

例如,使用冒泡排序法对数据元素 49,38,65,97,76,13,27,$\overline{49}$进行排序,具体排序过程如下:

初始状态	第一次	第二次	第三次	第四次	第五次	第六次	第七次
49	**13**	**13**	**13**	**13**	**13**	**13**	**13**
38	49	**27**	**27**	**27**	**27**	**27**	**27**
65	38	49	**38**	**38**	**38**	**38**	**38**
97	65	38	49	**49**	**49**	**49**	**49**
76	97	65	$\overline{49}$	$\overline{49}$	**$\overline{49}$**	**$\overline{49}$**	**$\overline{49}$**
13	76	97	65	65	65	**65**	**65**
27	27	76	97	76	76	76	**76**
$\overline{49}$	$\overline{49}$	$\overline{49}$	76	97	97	97	97

在以上排序过程中,可设置一个布尔变量,用于记录是否发生过记录交换,若没有记录交换,则表明记录已有序,退出循环,这样可提高算法的效率。

(2)冒泡法排序的特点

①时间效率与初始序列有关。初始序列正向(递增)有序时,时间复杂度为 O(n);初始序列逆向(递减)有序时,时间复杂度为 $O(n^2)$;无序时平均时间复杂度为 $O(n^2)$。

②空间复杂度为 O(1)。

③冒泡法排序是稳定的排序方法。

④算法简单,适合记录基本正向有序或记录较少的情况。

2.快速排序

(1)快速排序的基本思想

从要被排序的数组 R[1…n]中选定一个记录(一般为第一个记录)作为“基准”,从待排序序列的两端交替地向中间扫描,将其他记录的关键字 k′与基准的关键字 k 比较,若 k′<k,则将对应记录换至 R 之前,反之,若 k′>k,则将对应记录换到 R 之后。通过一趟排序将待排序记录分成两部分,排在 R 之前的记录的关键字均小于等于 k,排在 R 之后的记录的关键字均大于等于 k;然后继续对 R 前后两部分记录进行快速排序,直至排序范围为 1 为止。

例如,使用快速排序法对数据元素 49,38,65,97,76,13,27,$\overline{49}$进行排序,具体排序过程如下:

初始状态:　　[49　38　65　97　76　13　27　$\overline{49}$]

第一次排序之后:[27　38　13]　49　[76　97　65　$\overline{49}$]

第二次排序之后:[13]　27　[38]　49　[$\overline{49}$　65]　76　[97]

第三次排序之后:13　27　38　49　$\overline{49}$　[65]　76　97

最后的排序结果:13　27　38　49　$\overline{49}$　65　76　97

(2)快速排序的特点

①时间效率与初始序列有关。初始序列正向有序时,快速排序法与冒泡法相同,时间复杂度为

$O(n^2)$，无序状态下，时间复杂度为 $O(n\log_2 n)$。

②快速排序的过程是递归的。

③空间复杂度为 $O(n\log_2 n)$，最坏为 $O(n)$。

④快速排序法是不稳定的排序方法。

9.9.4 选择排序

选择排序的基本思想：在待排序记录中，依次选出关键字最小的记录，顺序放在已排好序的子表的最后，直到全部记录排序完毕。

1.简单选择排序

(1)简单选择排序基本思想

从待排序记录 $R[i\cdots n]$(初始状态下 $i=1$)中选择关键字最小的记录，设为 R_j；将 R_j 与 R_i 交换；将 i 向后移动 $i=i+1$；重复前述步骤直至 $i=n-1$。

例如，使用简单选择排序法对数据元素 49，38，65，97，76，13，27，$\overline{49}$进行排序，具体排序过程如下：

初始状态：　[49　38　65　97　76　13　27　$\overline{49}$]

第一次排序后：13　[38　65　97　76　49　27　$\overline{49}$]

第二次排序后：13　27　[65　97　76　49　38　$\overline{49}$]

第三次排序后：13　27　38　[97　76　49　65　$\overline{49}$]

第四次排序后：13　27　38　49　[76　97　65　$\overline{49}$]

第五次排序后：13　27　38　49　$\overline{49}$　[97　65　76]

第六次排序后：13　27　38　49　$\overline{49}$　65　[97　76]

第七次排序后：13　27　38　49　$\overline{49}$　65　76　[97]

最后排序结果：13　27　38　49　49　65　76　97

(2)简单选择排序的特点

①时间复杂度为 $O(n^2)$。

②空间复杂度为 $O(1)$。

③简单选择排序是不稳定的排序方法。

④简单选择排序适用于 n 较小的情况。

2.堆排序

(1)堆的定义

n 个关键字序列$(k_1,k_2,\cdots,k_n)$称为堆，当且仅当该序列满足如下条件：

①$k_i\leqslant k_{2i}$且 $k_i\leqslant k_{2i+1}$$(1\leqslant i\leqslant[n/2])$；

②或 $k_i\geqslant k_{2i}$且 $k_i\geqslant k_{2i+1}$$(1\leqslant i\leqslant[n/2])$。

若将序列$(k_1,k_2,\cdots,k_n)$看成是一棵完全二叉树的按层遍历序列，则堆实质上是满足如下性质的完全二叉树：

①树中任一非叶结点的关键字均不大于(或不小于)其左右孩子(若存在)结点的关键字；

②左、右子树也是堆。

小根堆：根结点(亦称为堆顶)的关键字是堆里所有结点关键字中最小者的堆称为小根堆。

大根堆：根结点的关键字是堆里所有结点关键字中最大者的堆称为大根堆。

例如，数据元素序列(9，16，23，18，20，36，49)为小根堆，数据元素序列(100，50，80，20，40，60，70)为大根堆，如图 9—24 所示。

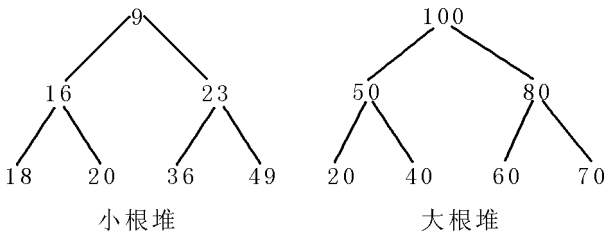


图 9—24　小根堆与大根堆

(2)建堆

采用筛选法。假设左、右子树已经是堆,若根关键字小于左、右孩子的关键字,则将根与左、右孩子中关键字较大的结点交换。若筛选后使左子树或右子树不满足堆的定义,则继续对左子树或右子树进行筛选直到满足堆的条件为止。

具体做法是:从完全二叉树的最后一个分支结点开始,自底向上筛选,把完全二叉树调整为堆。

(3)堆排序

堆排序利用小根堆(或大根堆)来选取当前无序区中关键字最小(或最大)的记录实现排序。下面以大根堆为例来说明堆排序。

堆排序基本步骤为:将待排序元素调整为一个大根堆;将关键字最大的堆顶元素与堆的最后一个元素交换;将剩下的元素(即除最后一个元素外的其余元素)调整为堆,然后将关键字最大的堆顶元素与该堆的最后一个元素交换;依此类推,直到全部元素排成有序序列。

例如,使用大根堆排序将数据元素序列(100,50,80,20,40,60,70)递增排序的过程如图 9—25 所示。

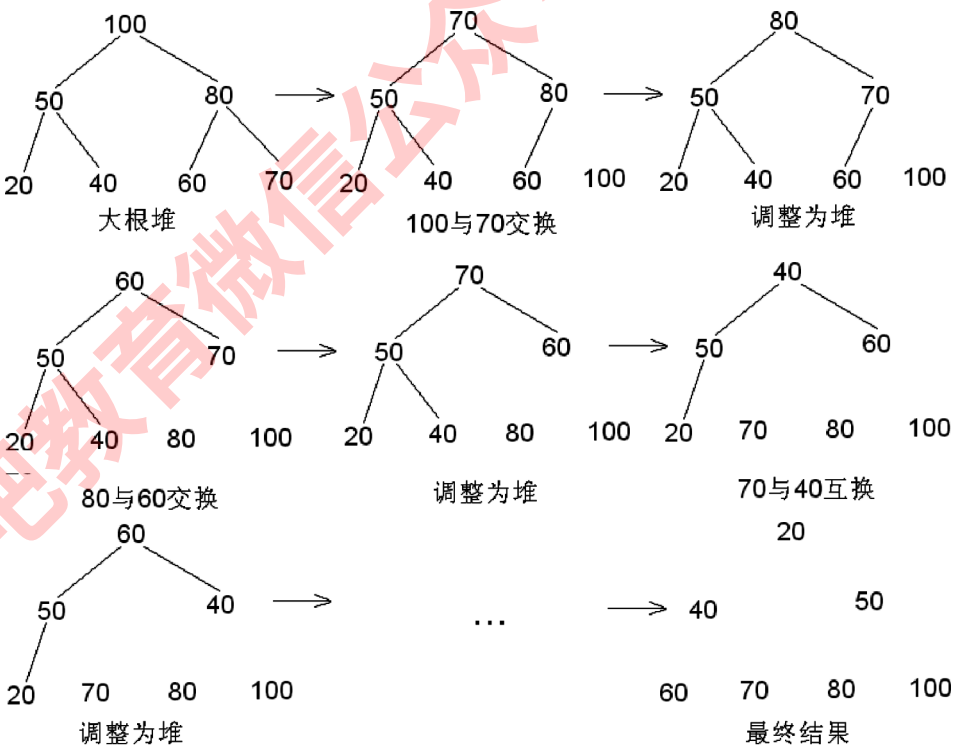


图 9—25　大根堆排序图

(4)堆排序的特点

①时间复杂度为 $O(n\log_2 n)$。时间由建堆和调整堆两部分组成。

②空间复杂度为 O(1)。

③堆排序是不稳定的排序方法。

④适用于 n 较大的情况。

9.9.5 各种排序方法的比较

1.影响排序效果的因素

①待排序的记录数目 n；

②记录的大小(规模)；

③关键字的结构及其分布情况；

④对稳定性的要求；

⑤存储结构；

⑥时间和空间复杂度等。

2.排序方法的选择

①若 n 较小,可采用直接插入排序、冒泡排序或简单选择排序。

②若 n 较大,可采用希尔排序,快速排序、堆排序或归并排序。

③若文件初始状态基本有序(正序),可采用直接插入排序或冒泡排序。

9.10 查找

9.10.1 查找的基本概念

查找表:一种以集合为逻辑结构、以查找为核心的数据结构。

关键字:数据元素的某个数据项,该数据项能唯一标识一个数据元素。

查找:给定一个值 K,在查找表中找出关键字等于给定值 K 的数据元素。若找到,则查找成功,返回该数据元素信息或该数据元素在表中的位置;否则查找失败,返回相关的指示信息。

动态查找表:查找的同时对表做修改操作(如插入和删除),则相应的表称之为动态查找表。

静态查找表:查找时没有对表进行修改操作,则相应的表称之为静态查找表。

内查找:整个查找过程都在内存进行。

外查找:查找过程中需要访问外存。

平均查找长度:查找算法中的主要操作是关键字与给定值的比较,所以通常把查找过程中对关键字需要执行的平均比较次数(即平均查找长度)作为衡量一个查找算法效率的度量。

平均查找长度 ASL 的计算公式如下:

$$ASL(n)=\sum_{i=2}^{n}P_iC_i$$

其中,n 是结点的个数;P_i 是查找第 i 个结点的概率;C_i 是找到第 i 个结点所需进行的比较次数。一般认为每个结点的查找概率相等,即 $P_1=P_2\cdots=P_n=1/n$。

9.10.2 顺序查找

1.顺序查找的基本思想

从表的一端开始,顺序扫描线性表,依次将表中各记录的关键字与给定值比较,若找到,则查找成功,反之,则查找失败。

2.顺序查找的特点

(1)查找表用线性表表示。如 L=(33,19,23,49,40)。

(2)查找表可以用顺序结构或链式结构存储。

(3)查找效率低,当 n 较大时不宜采用顺序查找。

9.10.3 折半查找

折半查找又称二分查找,它是一种效率较高的查找方法。

折半查找要求线性表是有序表,即表中数据元素按关键字有序(递增或递减),例如:

L=(10,20,30,40,50,60)。

1.折半查找的基本思想

假设查找表按关键字递增有序。若查找表为空,则查找失败;反之,将查找表中间位置数据元素的关键字与给定值进行比较:

(1)若该关键字等于给定值,则查找成功,返回数据元素的位置。

(2)若该关键字小于给定值,则继续在前半个表中进行折半查找。

(3)若该关键字大于给定值,则继续在后半个表中进行折半查找。

2.折半查找判定树

折半查找的过程可以用折半查找来描述。

具体做法为:把当前查找区间的中间位置上的结点作为根,左子表和右子表中的结点分别作为根的左子树和右子树,依此类推,得到的折半查找的判定树。

3.折半查找的特点

(1)查找表中的记录按关键字递增(递减)有序。

(2)查找表用顺序结构存储。折半查找不适合用链式结构存储。

9.10.4 查找树

1.二叉排序树

二叉排序树又称二叉查找树,常作为动态查找表的组织方式。

二叉排序树或者是一棵空树,或者是具有如下性质的二叉树:

(1)若左子树不为空,则左子树上的所有结点的值均小于根结点的值。

(2)若右子树不为空,则右子树上的所有结点的值均大于根结点的值。

(3)左、右子树均为二叉排序树。

如果按中序遍历树,则二叉排序树所得到的中序序列是一个递增有序序列。如图 9—26 所示,中序序列:2,3,4,5,6,7,8 为有序递增序列。

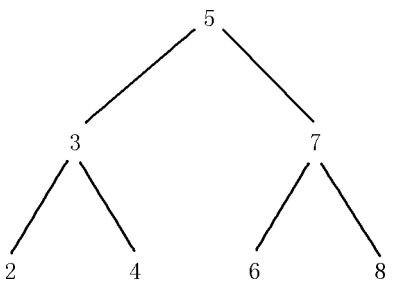


图 9—26 二叉排序树

2.平衡二叉树

平衡二叉排序树简称平衡二叉树，它除了具有二叉排序树的特点之外，还具有如下特点：

(1)左、右子树的高度之差不超过 1。

(2)左、右子树也是平衡二叉树。

在平衡二叉树中插入结点后，其调整原则如下：

①调整失去平衡的最小子树。

②根据要调整子树树型，进行左旋(或右旋)平衡处理。

③调整后必须保持平衡二叉树的特性。

图 9—26 既是一棵二叉排序树，也是一颗平衡二叉树。

3.B－树

B－树是一种多路平衡查找树，它适合在磁盘等直接存取设备上组织动态的查找表，常作为文件的索引。

真 题 精 选

1. 队列的入队序列为 1、2、3、4，则输出序列为(　　)。

A. 1、2、4、3　　B. 4、3、2、1

C. 1、4、2、3　　D. 1、2、3、4

【答案】D

【解析】队列为操作受限的线性表，其操作原则为“先入先出”。所以本题输出序列只能为 1、2、3、4。

2. 链表不具有的特点是(　　)。

A. 可随机访问任意元素

B. 不必事先估计存储空间

C. 插入数据元素时不需移动数据元素

D. 删除数据元素时不需移动数据元素

【答案】A

【解析】链表不具有随机访问任意元素的特点，这是顺序表的特点。

3. 以下数据结构中，属于非线性数据结构的是(　　)。

A. 树　　B. 队列　　C. 栈　　D. 字符串

【答案】A

【解析】根据数据结构中各数据元素之间前后件关系的复杂程度，一般将数据结构分为两大类型：线性结构与非线性结构。如果一个非空的数据结构满足下列两个条件：

(1)有且只有一个根结点；

(2)每一个结点最多有一个前件(直接前趋)，也最多有一个后件(直接后继)。

则称该数据结构为线性结构。线性结构又称线性表。在一个线性结构中插入或删除任何一个结点后还是线性结构。栈、队列、字符串等都是线性结构。

如果一个数据结构不是线性结构，则称之为非线性结构。数组、广义表、树和图等数据结构都是非线性结构。因此选择 A 选项。

4. 现有一个空栈，输入序列为 A，B，C，D，E，经过操作序列 push，pop，push，push，pop，pop，push，push 后，栈顶元素是(　　)。

A. B　　B. C　　C. D　　D. E

【答案】E

【解析】栈按“后进先出”的原则组织数据，pop 表示出栈，push 表示入栈，因此根据题干各元素入栈出栈的顺序是：A 入栈，A 出栈，B 入栈，C 入栈，C 出栈，B 出栈，D 入栈，E 入栈。此时栈顶元素为 E。

5. 在一棵二叉树的第五层上结点数最多是(　　)。

A. 8　　B. 16　　C. 32　　D. 15

【答案】B

【解析】在一棵二叉树的第 i 层上最多有 2^{i-1} 个结点，因此，第五层的结点数最多是 $2^{5-1}=16$。

强　化　训　练

一、单项选择题

1. 支持子程序调用的数据结构是(　　)。

A. 栈　　B. 树

C. 队列　　D. 二叉树

2. 一棵二叉树具有 n 个结点，用二叉链表存储时，其中有(　　)个指针用于指向孩子结点。

A. 2n　　B. n－1

C. n＋1　　D. n

3. 对长度为 n 的线性表排序，在最坏情况下，比较次数不是 n(n－1)/2 的排序方法是(　　)。

A. 快速排序　　B. 冒泡排序

C. 直接插入排序　　D. 堆排序

4. 一棵二叉树中共有 70 个叶子结点与 80 个度为 1 的结点，则该二叉树中的总结点数为(　　)。

A. 219　　B. 221

C. 229　　D. 231

5. 设有一个非空线性链表，在由 p 所指向的结点后插入一个由 q 所指向的结点，可依次执行操作(　　)。

A. p＝q→next；q＝p→next；　　B. q→next＝p→next；p→next＝q；

C. q→next＝p→next；q＝p；　　D. q→next＝p；p→next＝q；

二、多项选择题

1. 下列对循环队列的叙述，正确的是(　　)。

A. 顺序存储结构　　B. 不会产生下溢

C. 不会产生上溢　　D. 队满时 rear＝＝front

E. 不会产生假溢

2. 下列选项中，属于图的应用算法的是(　　)。

A. 克鲁斯卡尔算法　　B. 哈夫曼算法

C. 迪杰斯特拉算法　　D. 欧几里得算法

E. 拓扑排序算法

3. 一个输入序列 a,b,c,d 经过一个栈到输出序列,并且一旦离开输入序列后就不能再返回到输入序列,则下面(　　)为正确的输出序列。

A. b,c,a,d　　B. c,b,d,a

C. d,a,b,c　　D. a,c,b,d

E. d,c,b,a

三、判断题

1. 二分查找要求被查找的表是基本有序的。(　　)
2. 对一个算法的评价包括其正确性、时间复杂度和并行性等。(　　)
3. 希尔排序又称“缩小增量排序”,属于“交换”排序法。(　　)
4. 顺序存储结构中,插入、删除操作灵活方便,不必移动结点。(　　)
5. 算法的空间复杂度是指算法程序中指令(或语句)的条数。(　　)

四、简答题

数据元素之间的关系在计算机中有几种表示方法,各有什么特点?

五、应用题

对下列数据表(100,13,40,56,78,4,34,25,98,48,23,92,84,69,8,72),设增量序列为 d={4,2,1},写出采用希尔排序算法的每一趟结果。

参考答案

一、单项选择题

1. A [解析] 栈是一种限定在一端进行插入与删除的线性表。在主函数调用子函数时,要首先保存主函数当前的状态,然后转去执行子函数,把子函数的运行结果返回到主函数调用子函数时的位置,主函数接着向下执行,这种过程符合栈的特点,所以一般采用栈式存储方式。

2. B [解析] 一棵二叉树具有 n 个结点,用二叉链表存储时,其中有 $n-1$ 个指针用于指向孩子结点。

3. D [解析] 对长度为 n 的线性表进行堆排序,最坏情况下比较次数为 $2n\log_2 n$。

4. A [解析] 在任意一棵二叉树中,度为 2 的结点总比度为 0 的结点少一个,根据此关系可以计算出本题中度为 2 的结点为 69 个,所以总结点数为 69+70+80=219。

5. B [解析] 在线性链表中插入结点后,仍然是线性链表,除了首结点和尾结点外,每个结点均有一个前趋和一个后继。本题中,要在 p 结点之后插入 q 结点,可先将 p 的后继结点地址存入 q 的指针域 next 中,然后将 q 结点的地址存入 p 的 next 域中即可。正确答案为 B。

二、多项选择题

1. AE [解析] 循环队列是顺序存储结构,并且不会产生假溢,因此选择 A、E 选项。

2. ACE [解析] 克鲁斯卡尔算法用于求解图的最小生成树,迪杰斯特拉算法用于求解图的最短路径,拓扑排序算法用于求解图的拓扑序列,因此选择 A、C、E 选项。

3. ABDE [解析] C 选项中,当 i=1,j=2,k=3 时,有 $p_i=d$,$p_j=a$,$p_k=b$,即有 $p_i<p_k<p_j$,因此是不正确的。其他选项均是正确的输出序列。

三、判断题

1. √ [解析]二分查找又称折半查找，它是一种效率较高的查找方法，适用于基本有序的表。

2. × [解析]对一个算法的评价包括其健壮性、可读性、正确性和时空复杂度。

3. × [解析]希尔排序法是对直接插入排序进行改进而得到的一种插入排序法。

4. × [解析]顺序存储结构的特点是逻辑关系上相邻的两个元素在物理位置上也相邻，其缺点是在做插入或删除操作时，需移动大量的元素。

5. × [解析]算法的空间复杂度是对算法在执行时所需的辅助存储空间的度量。

四、简答题

[专家点拨]数据元素之间的关系在计算机中有顺序存储、链式存储、索引存储和散列存储4种表示方式，各种方式的特点分别如下：

(1)顺序存储方式。数据元素按顺序存放，每个存储结点只包含一个元素。存储位置反映数据元素间的逻辑关系。这种方式具有存储密度大的优点，但在进行插入、删除操作时效率较低。

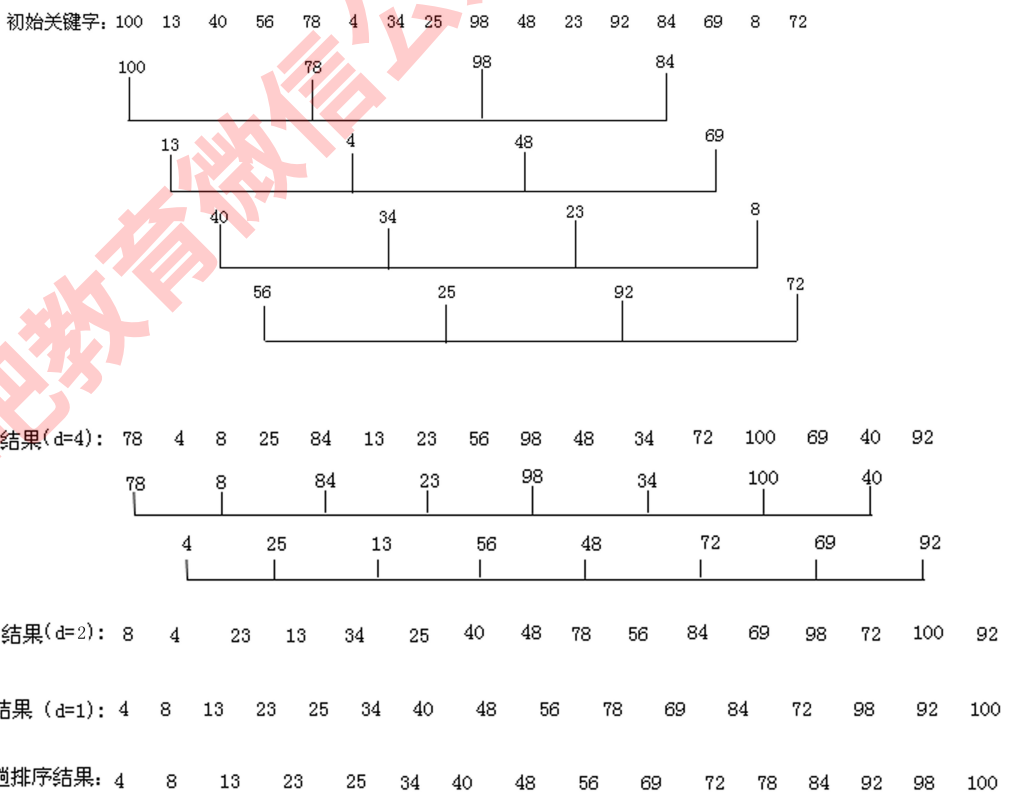
(2)链式存储方式。每个存储结点包含数据元素信息外，还包含一组(至少一个)指针。指针反映数据元素间的逻辑关系。这种方式不要求存储空间连续，便于动态操作(如插入、删除等)，但存储空间开销大，不能进行折半查找。

(3)索引存储方式。除数据元素存储在一片地址连续的内存空间外，尚需建立一个索引表，索引表中的索引指示存储结点的存储位置或存储区间端点，兼有动态和静态特性。

(4)散列存储方式。通过散列函数和解决冲突的方法将关键字散列在连续、有限的地址空间内，并将散列函数的值解释成关键字所在元素的存储地址，这种存储方式称为散列存储。其特点是存取速度快，只能按关键字随机存取，不能顺序存取，也不能进行折半查找。

五、应用题

[专家点拨]



C 语言程序设计

热 点 分 析

C 语言是一种计算机程序设计语言，是计算机编程初学者的首选入门语言。在事业单位招聘考试中，该部分中常作为独立考查内容的是基本语法、数据类型等基础知识。另外，由于 C 语言常被作为数据结构的实现语言，故多数试题将 C 语言与数据结构结合起来考查考生的应用能力。因此，考生在学习数据结构之前，应对 C 语言知识做全面透彻的了解及学习，尤其是函数定义、函数调用、变量与函数生存期以及指针等知识。

10.1 C 语言基础

10.1.1 C 语言程序基础

任何一种计算机程序语言，都具有特定的语法规则和一定的表现形式。程序的书写格式和构成规则是程序语言表现形式的一个重要方面。按照规定的格式和构成规则来书写程序，不仅可以使程序设计人员和使用程序的人容易理解，更重要的是把程序输入计算机时，计算机能够充分识别，从而能够正确执行它。

C 语言程序格式的具有以下特点。

第一，C 语言程序是由函数构成的。一个完整的 C 语言程序可以只包含一个主函数（main 函数），也可以包含一个 main 函数和若干个其他函数（库函数和自编函数）。利用这一特点很容易实现 C 语言程序的模块化。

第二，一个 C 语言程序总是从 main 函数开始执行，而不论 main 函数在整个程序中处于何位置。

第三，从格式上看，每个函数是由函数名和花括号“{}”包围的若干语句组成的。但组成 C 语言程序的函数中，必须有一个且只有一个名字为 main 的函数，叫做主函数。除主函数之外的函数由用户命名。C 语言程序的执行是从主函数开始的，主函数中的所有语句执行完毕则程序执行结束。但程序执行当中可调用函数，因此，C 语言程序中，main 函数之外的其他函数都是在执行 main 函数时，通过嵌套调用的方式得以执行的。在程序中除了可以调用用户自己编制的函数外，还可以调用由系统提供的标准函数。

10.1.2 C 语言程序的结构

程序开始带有 # 号的词句是编译预处理语句，又可称为头文件。

一个函数由两部分组成：

第一，函数体的说明部分，包括函数名、函数类型、函数参数名、形式参数类型；

第二，函数体，即函数说明部分下面的花括号“{}”内的部分。函数体一般包括变量定义部分和执行语句部分。

C 语言函数的一般格式如下：

函数名(参数列表)
{
变量定义部分;
执行语句部分;
}

有的函数没有参数,则参数列表部分也就不存在了,但函数名后的圆括号不能省略。

10.1.3 C语言程序的风格

C语言程序的风格可以归纳为以下几点。

(1)C语言程序习惯上使用小写英文字母,也可用大写字母,但大写字母常常是作为常量的宏定义和其他特殊用途使用。

(2)C语言程序由一个个语句组成。每个语句都具有规定的语法格式和特定的功能。

(3)C语言程序不使用行序号。

(4)C语言程序使用分号“;”作为语句的终止符或分隔符。

(5)C语言程序不存在程序行的概念。一个程序可以自由使用任意的书写行,即一行中可以有多个语句,一个语句也可以占用任意多行,但注意语句之间必须用“;”分隔。

(6)C语言程序中用花括号“{}”表示程序的结构层次范围。

(7)C语言程序中,为了增强可读性,可以使用适量的空格和空行。但是,变量名、函数名以及C语言中的保留字中间不能插入空格。除此之外的空格和空行是可以任意设置的,C语言编译系统将忽略这些空格和空行。

综上所述,C语言程序的书写格式自由度较高、灵活性很强,有较大的随意性。但是,为了避免程序书写时层次混乱不清,更便于人们阅读和理解,一般都采用一定格式的书写方式。这样的书写方式并不是计算机要求的,而是为了给人们在阅读和分析程序时提供便利。

为了更直观地了解C语言程序的结构特点,下面给出一个程序例子。

```
    /*找出两个数中的大数*/
main()                          /*主函数*/
    {
      int a,b,c;                /*定义3个整型变量*/
      scanf("%d,%d",&a,&b);     /*输入变量a和b的值*/
      c=max(a,b);               /*调用max函数,找出大数并赋给c*/
      printf("max=%d\n",c);     /*输出c即为大数*/
    }
    int max(int x,int y)        /*定义max函数,函数返回值为整型,x,y为形式参数*/
    {
      int z;                    /*定义max函数中用到的变量z*/
      if(x>y)
        z=x;
      else
        z=y;
      return(z);                /*将z的值,即找出的最大值返回*/
    }                           /*返回调用处*/
```

C 语言程序的函数模块结构特点，使用程序整体结构清楚、层次分明，它为模块化软件设计方法提供了有力的支持。

将 C 语言程序输入计算机时，首先要使用系统提供的编译程序（又称为编辑器）建立 C 语言程序的源文件，建立后的源文件以文本文件的形式存储在磁盘上，源文件的名字由用户给出，扩展名一般为.c。

和其他程序一样，C 语言程序中也可以使用注释。注释的格式如下：

/ * 注释内容 * /

注释可以加在程序的任意位置，它可以占用一行以上的位置，也可以写在语句的后面。注释作为源程序的一个部分存在于源程序中，但对源程序进行编译时，系统将忽略注释。

10.2 C 语言的语法基础

10.2.1 基本语法

C 语言的基本词法由 3 部分组成：符号集、标识符、关键字（又称保留字）。

符号集就是一门语言中允许出现的字符的集合，C 语言的符号集就是 ASCII 码表中的一些字符，在键盘上不能直接得到（比如说响铃字符），C 语言引入了转义字符的概念，利用反斜杠符号“\”后加上字母的一个字符组合来表示这些字符，当在源程序中遇到这类字符组合时，虽然这个字符组合是一个字符串的形式，但 C 语言仍会自动将之理解成某一特定的符号。比如“\r”，C 语言在处理这个字符组合时，会自动理解成回车换行符号。转义字符经过进一步引申应用，形成了另外两种形式：“\ddd”和“\xnn”。这时“\”后的 ddd 和 xnn 分别代表 3 位八进制和两位十六进制数（打头的 x 只是表明后面跟着的是十六进制数），这两种形式不再局限于表示不可打印的字符，它们可以表示 ASCII 码表中的任意字符，只要把所需表示的字符的 ASCII 码转换成八进制数和十六进制数即可。比如说字母 A，ASCII 码为 65，65 的八进制和十六进制分别为 101 和 x41，字母 A 可表示为“\101”或“\x41”。

标识符就是用以标识的符号。正如现实生活中给每一个人都取一个名字一样，C 语言中的每一个对象（如函数、变量等）都必须取一个标识符以便和其他对象区别开。在 C 语言中，这个标识符是一个字符串，这个字符串的选定有一定的规则：必须是以字母或下划线开头的字母与数字的序列。除了这个基本的规则外，C 语言对标识符的命名还有几个限制需加以注意：

（1）长度最好不要超过 8 个字符。因 C 语言中对标识符只处理前 8 个字符，超过部分将被 C 语言自动忽略掉。在 C 语言中，“ABCDEFGH1”和“ABCDEFGH2”是同一个标识符；

（2）标识符不要与保留字同名，最好也不要与 C 语言提供的标准标识符，如库函数重名；

（3）应注意，C 语言区分大小字母，ABcd 和 abcd 是两个不同的标识符。

关键字实际上就是一些特殊的标识符，又称保留字，这些保留字不允许用户对它重新定义。

10.2.2 头文件、数据说明、函数的开始和结束标志

头文件：也称为包含文件或标题文件，一般放在一个 C 语言程序的开头，用 # include"文件名"或 # include〈文件名〉的格式，其中文件名是头文件名，一般用.h 作为扩展名。

数据说明：C 语言中的数据分为常量和变量两种。

常量：有数值常量和符号常量两种。数值常量：可以分为整型常量、实型常量、浮点型常量和字符型常量。符号常量：用一个标识符代表的一个常量，又称标识符形式的常量。

变量：其值可以改变的量，变量名习惯上用小写字母表示。

10.2.3 数据类型

C 语言的数据类型可以分为 3 类，如表 10—1 所示。

表 10—1　　C 语言的数据类型

基本类型	构造类型	派生类型
整型 int	结构体 struct	用户定义类型(使用关键字
字符型 char	共用(联合)体 union	typedef)
双精度型 double	数组类型	指针类型
空类型 void	单精度型 float	

1.整型常量

C 语言中的整型常量有 3 种形式：十进制整型常量、八进制整型常量和十六进制整型常量。十进制整型常量可以用一串连续的十进制数字来表示；八进制整型常量用数字 0 开头(注意：不是字母 O)，后面可以跟一串合法的八进制数字；十六进制整型常量用 0x 或 0X 开头，后面可以跟一串合法的十六进制数字。

整型常量又有短整型(short int)、基本整型(int)、长整型(long int)和无符号型(unsigned)之分。

2.整型变量

整型变量也可以分为基本型、短整型、长整型和无符号型 4 种，分别用 int，short int(或 short)，long int(或 long)，unsigned int(unsigned short，unsigned long)对它们进行定义。

不同的计算机对上述几种型数据所占用的内存字数和数值范围有不同的规定，以 IBM-PC 微机为例，以上各种数据所分配的存储空间和数值范围如表 10—2 所示。

表 10—2　　数据类型与存储空间

类型名	所占字节数	数值范围
int	2	−32768～+32767
short[int]	2	−32768～+32767
long[int]	4	−2147483648～+2147483647
unsigned[int]	2	0～65535
unsigned short	2	0～65535
unsigned long	4	0～4294967295

3.实型常量

C 语言中的实型常量有两种表示形式：十进制数形式和指数形式。在用指数形式表示实型数据时，字母 E 可以用小写 e 代替，指数部分必须是整数(若为正整数时，可以省略“+”号)。

4.实型变量

C 语言中的实型变量分为两种：单精度类型和双精度类型，分别用关键字 float 和 double 进行定义。在一般系统中，一个 float 型数据在内存中占 4 个字节；一个 double 型数据占 8 个字节(一个 long double 型数据占 16 个字节)。

5.字符常量

C 语言的字符常量代表 ASCII 码字符集里的一个字符，在程序中用单引号括起来。C 语言规定字符常量可以作为整数常量来处理(注：这里的整数常量指的是相应字符的 ASCII 代码，因此字符常量可以参与算术运算)。

在C语言中还有一类特殊形式的字符常量，称为“转义字符”。这类字符常量是以一个反斜杠开头的字符序列，但它们只代表某个特定的ASCII码字符，在程序中使用这种常量时要括在一对单引号中。

6.字符变量

C语言中的字符变量用关键字char来定义，每个字符变量中只能存放一个字符。在一般系统中，一个字符变量在计算机内存中占一个字节。与字符常量一样，字符变量也可以出现在任何允许整型变量参与的运算中。

7.字符串常量

C语言中的字符串常量是由一对双引号括起来的字符序列。注意不要将字符常量和字符串常量混淆。C语言对字符串常量的长度不加限制，C语言编译程序总是自动地在字符串的结尾加一个转义字符‘\0’，作为字符串常量的结束标志。C语言中没有专门的字符串变量，如果要把字符串存放在变量中，则要用一个字符型数组来实现。

10.2.4 运算符

C语言中的运算符可以归纳为下列5类：算术运算符、关系运算符、逻辑运算符、赋值运算符和条件运算符。

1.算术运算符

算术运算符有＋、－、＊、/、％，分别表示算术加、减、乘、除和取余运算。

这些运算符需要两个运算对象，称双目运算符。这些运算符中，除取余(％)运算符外，其他运算符的运算对象既可以是整型，也可以是实型数据。取余运算的运算对象只能是整型。取余运算的结果是两数相除后所得的余数。

“＋”和“－”也可以用做单目运算符，但作为单目运算符时必须出现在运算量的左边，运算量可为整型，也可以为实型。

C语言中还提供两个特殊的单目运算符：＋＋和－－，这两个运算符既可以放在运算对象之前，又可以放在运算对象之后。

在C语言中，凡是按C语言语法规则用常量、变量、函数调用以及运算符把运算数连接起来的式子都是合法的表达式。凡表达式都有一个值，即运算结果。

算术运算符和一对圆括号组成的算术表达式的运算优先级如图10—1所示。

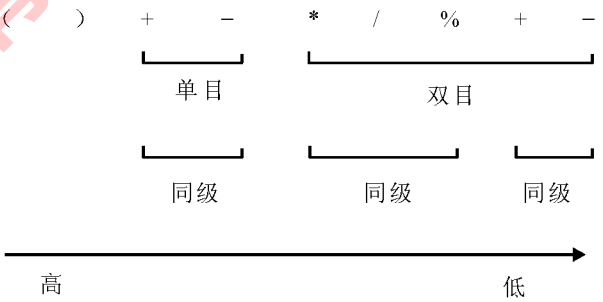


图10—1 算术运算符优先级

以上所列的运算符中，只有单目运算符“＋”和“－”的结合性是从右到左，其余运算符的结合性都是从左到右。

算术表达式的求值规律与数学中的四则运算规律类似，其运算规律和要求为：

①在算术表达式中，可使用多层括号，但左右括号必须配对。运算时从内层圆括号开始，由内向

外依次计算表达式的值。

②在算术表达式中，若包含不同优先级的运算符，则按运算符的优先级别由高到低进行，若表达式中运算符的级别相同，则按运算符的结合方向进行。

2.关系运算符

在算术表达式中，C 语言提供了 6 种关系运算符：<，>，<=，>=，==，!=。前 4 种运算符（<，>，<=，>=）的优先级相同，后两种的优先级也相同，并且前 4 种的优先级高于后两种。

关系运算符属于双目运算符，其结合方向为自左至右。

用关系运算符可以将两个表达式（包括算术表达式、关系表达式、逻辑表达式、赋值表达式和字符表达式）连接起来构成关系表达式。

关系运算的结果是 1 或 0。在 C 语言中没有逻辑值，用 0 代表“假”，用 1 代表“真”。

3.逻辑运算符

C 语言提供 3 种逻辑运算符：&&（逻辑与）、||（逻辑或）、!（逻辑非）。其中前两种为双目运算符，第三种是单目运算符。

关系运算符中的 && 和 || 运算符的优先级相同，! 运算符的优先级高于前两个。

算术运算符、逻辑运算符和关系运算符三者间的优先级关系如图 10—2 所示。

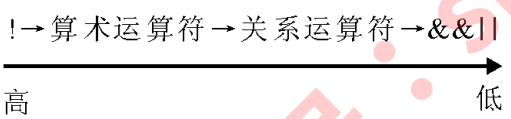


图 10—2　不同运算符的优先级

用逻辑运算符将关系表达式或任意数据类型（除 void 外）的数据连接起来就构成了逻辑表达式。逻辑表达式的值是 0 或 1。

4.赋值运算符

在 C 语言中，“=”称为赋值运算符。由赋值运算符组成的表达式称为赋值表达式。赋值表达式的形式为：

变量=表达式；

赋值符号左边必须是一个代表某一存储单元的变量名，赋值符号的右边必须是 C 语言中合法的表达式。

赋值运算的功能是先计算右边表达式的值，然后再把此值赋给赋值符号左边的变量，确切地说，是把数据放入以该变量为标识的存储单元中。

5.条件运算符

C 语言中把“? :”称作条件运算符。条件运算符要求有 3 个运算对象，它是 C 语言中唯一的一个三目运算符。由条件运算符构成的条件表达式的一般形式如下：

表达式 1? 表达式 2：表达式 3

当表达式 1 的值为真时，取表达式 2 的值为此条件表达式的值；当表达式 1 的值为假时，取表达式 3的值为此条件表达式的值。

条件运算符具有自右向左的结合性，其优先级别比关系运算符和算术运算符都低。

10.3 程序控制

10.3.1 顺序结构

1.表达式语句、函数调用语句和空语句

C 语言的语句共分 5 大类:表达式语句、控制语句、函数调用语句、空语句和复合语句。

①表达式语句的一般形式为:

表达式;

最典型的表达式语句是由一个赋值表达式加一个分号构成的赋值语句。

②控制语句是 C 语言程序设计中用来构成分支结构和循环结构的语句。此类语句有 if 语句、for 语句、while 语句、do…while 语句、switch 语句等。

③函数调用语句的一般形式如下:

函数名(实参表);

④空语句的一般形式如下:

;(只含有一个分号的语句)

这条语句的含义是什么也不做。

⑤复合语句的一般形式如下:

{语句 1;语句 2;…;}

复合语句在功能上相当于一条语句。

2.输入输出函数的调用

C 语言本身没有提供输入、输出操作语句。C 语言程序的输入和输出完全依靠调用 C 语言的标准输入、输出函数来完成。4 个常用的输入、输出函数是:printf 函数、scanf 函数、putchar 函数、getchar 函数。

(1)printf 函数

printf 函数是 C 语言提供的标准输出函数,它的作用是在终端设备(或系统隐含指定的输出设备)上按指定格式进行输出。printf 函数的一般调用形式如下:

printf(格式控制,输出项表);

格式控制参数以字符串的形式描述,由以下两部分组成。

①普通字符:将被简单地显示;

②格式字符:将引起一个输出参数项的转换和显示,由“%”引出并以一个类型描述符结束的字符串,中间可加一些可选的附加说明项,如表 10—3 所示。

表 10—3　　附加说明项

附加说明项	说明
−或+	用于指定是否对齐输出,具有“−”符号表示左对齐,缺省或有“+”表示右对齐
0	用于指定是否填写 0,有此项表示空位用 0 补充,无此项表示以空格补充
m.n	用于指定输出域宽及精度,m 是指域宽,n 为精度,当不指定 n 时,隐含的精度为 6 位
l 或 h	用于输出长度修正,其中,l 对于整型是指 long,对于实型是 double;h 应只用于整型的格式字符,并修正为 short 型

格式字符用于指定输出项的数据类型及输入格式，如表 10—4 所示。

注意：编译程序只是在检查了 printf 函数中的格式参数后，才能确定有几个输出项，是什么类型、以什么格式输出。在编程序时，应使输出格式与输出项对应。

表 10—4　　格式字符

格式字符	说明
CcD	输出一个字符
d 或 i	输出带符号的十进制整型数
OoO	以八进制无符号形式输出整型数(不带前导 0)
x 或 X	以十六进制无符号形式输出整型数(不带前导 0x 或 0X)，对于 x，用 abcdef 输出十六进制数码；对于 X，用 ABCDEF 输出十六进制数码
UuU	按无符号的八进制形式输出整型数
FfF	以小数形式输出单精度或双精度数，小数位由精度指定，隐含的精度为 6；如指定精度为 0，则小数部分(包含小数点)都不输出
e 或 E	以指数形式输出单精度及双精度数，小数位数由精度指定，隐含的精度为 6；如指定精度为 0，则小数部分(包含小数点)都不输出
g 或 G	由系统决定是采用%f 还是采用%e 格式，以便使输出的宽度最小
SsS	输出字符串中的字符，直到遇到“\0”时为止，或输出指定的字符数
PpP	输出变量的内存地址
%	打印一个%

(2)scanf 函数

scanf 函数是 C 语言提供的标准输入函数，它的作用是在终端设备如键盘(或系统隐含指定的输入设备)上输入数据。scanf 函数的一般调用形式如下：

scanf(格式控制，输入项表)；

格式控制是用双引号括起来的字符串，称为格式控制串。格式控制串的作用是指定输入时的数据转换格式，即格式转换说明。格式转换说明也是由“%”符号开始，其后是格式描述符。

输入项表中的各输入项用逗号隔开，各输入项只能是合法的地址表达式，即在变量之前加一个地址符号“&”。

在 scanf 函数中每个格式说明都必须用%开头，以一个“格式字符”结束。

scanf 函数中的格式控制字符与 printf 函数中的相似，由格式说明项与输入格式符组成。格式说明项如表 10—5 所示。

表 10—5　　格式说明项

格式说明项	说明
%	起始符
*	赋值抑制符，用于按格式说明读入数据，但不赋给任何变量
MmM	域宽说明
l 或 h	长度修正说明符

scanf 中的格式字符如表 10—6 所示。

表 10—6　　scanf 中的格式字符

格式说明项	说明
CcC	输入一个字符
DdD	输入十进制整型数
IiI	输入整型数，整数可以是带前导 0 的八进制数，也可以是带前导 0x(0X)的十六进制数
OoO	以八进制形式输入整型数(可以带前导 0，也可不带前导 0)
XxX	以十六进制形式输入整型数(可带前导 0x 或 0X，也可不带)
UuU	无符号十进制整数
FfF	以带小数点形式或指数形式输入实型数
EeE	与 f 的作用相同
SsS	输入字符串

(3)putchar 函数

putchar 函数的作用是把一个字符输出到标准输出设备(通常指显示或打印机)上，一般调用形式如下：

```
putchar(ch);
```

其中，ch 代表一个字符变量或一个整型变量，ch 也可以代表一个字符常量(包括转义字符常量)。

(4)getchar 函数

getchar 函数的作用是从标准输入设备(通常指键盘)上读入一个字符。一般调用形式如下：

```
getchar();
```

getchar 函数本身没有参数，其函数值就是从输入设备输入的字符。

3.复合语句

在 C 语言中，一对花括号“{}”不仅可以用作函数体的开头和结尾标志，也可以用作复合语句的开头和结尾标志。复合语句的形式如下：

```
{
语句 1;
语句 2;
…
语句 n;
}
```

4.goto 语句及语句标号的使用

goto 语句称为无条件转向语句，一般形式如下：

```
goto 语句标号;
```

goto 语句的作用是把程序执行转向语句标号所在的位置，这个语句标号必须和此 goto 语句同在一个函数内。

语句标号在 C 语言中不必加以定义，这一点与变量的使用方法不同。标号可以是任意合法的标识符，当在标识符后面加一个冒号，该标识符就成了一个语句标号。

10.3.2 选择结构

1.用 if 语句实现选择结构

在 C 语言中,if 语句有两种形式,这两种形式分别如所述。

①形式 1:

```
if(表达式)
语句;
```

②形式 2:

```
if(表达式)
语句 1;
else
语句 2;
```

if 语句执行时,首先计算紧跟在 if 后面一对圆括号中的表达式的值,如果表达式的值为“真”,则执行 if 后的“语句”,然后去执行 if 语句的下一条语句。如果表达式的值为“假”,则不执行 if 后的“语句”,直接执行 if 语句后的下一条语句。

if 语句后面的表达式并不限于是关系表达式或逻辑表达式,而可以是任意表达式。

2.用 switch 语句实现多分支选择结构

switch 语句是用来处理多分支选择的一种语句。它的一般形式如下:

```
switch(表达式)
{   case 常量表达式 1:语句 1;
    case 常量表达式 2:语句 2;
        …
case 常量表达式 n:语句 n;
default:语句 n+1;
}
```

switch 语句的执行过程是:首先计算紧跟在 switch 后面的一对圆括号中表达式的值,当表达式的值与某一个 case 后面的常量表达式的值相等时,就执行此 case 后面的语句块并将控制流程转移到下一个 case 语句继续执行,直至 switch 语句的结束;若所有的 case 中的常量表达式的值都没有与表达式值匹配,又存在 default,则执行 default 后面的语句,直至 switch 语句结束;如果不存在 default,则跳过 switch 语句,什么也不做。

3.选择结构的嵌套

if 语句和 switch 语句都可以嵌套使用,特别要注意,对于构成嵌套的 if 语句,else 子句总是和离它最近的且不带 else 的 if 子句相匹配,不能弄混;在一个 switch 语句中的 case 后面又嵌套了一个 switch 语句,在执行内嵌的 switch 语句后还要执行一条 break 语句才跳出外层的 switch 语句。

10.3.3 循环结构

循环结构是在给定条件成立时,反复执行某个程序段。反复执行的程序段称为循环体。C 语言有 3 种循环流程控制:while 循环、for 循环和 do…while 循环。它们的循环可以是复合语句,也可以是单一语句或空语句。

1.for 循环结构

for 循环是功能上比 while 循环更强的一种循环结构,它的程序格式如下:

```
for(表达式 1;表达式 2;表达式 3)
{
    循环体语句;
}
```

for 循环的执行顺序是,首先进行表达式 1 的运算,然后计算表达式 2 的值,若结果为真或非零值,则执行循环体的语句,最后进行表达式 3 的运算。至此完成一次循环,然后再从计算表达式 2 的值开始下一次循环。如此下去,直至表达式 2 的结果为假或零时,循环结束。

for 循环的 3 个表达式起着不同的作用。表达式 1 用于进入循环之前给变量赋初值。表达式 2 表明循环的条件,它与 while 循环的表达式起相同作用;它一般是关系或逻辑表达式。表达式 3 用于循环一次后对某些变量的值进行修正。

for 结构中的表达式仅仅是表达式,其后的分号只是一个分隔符,并不表示这是一个语句,3 个表达式可以省略一个或多个,且表达式 1 和表达式 3 可以是逗号表达式。

2.while 和 do…while 循环结构

while 循环的格式如下:

```
while(表达式)
{
    循环体语句;
}
```

while 循环的执行过程:首先计算表达式的值,若表达式的结果是真或非零值,则执行循环体语句。然后再计算表达式的值,重复上述过程,直至计算表达式的值为零时,退出循环,流程控制转至循环下面的语句。C 语言中另外一种循环结构是 do…while 循环,它的格式如下:

```
do
  {
    循环体语句;
  }while(表达式);
```

do…while 循环的执行过程:首先执行循环体中的语句,然后计算表达式的值。若表达式的结果为非零值,则再次执行循环体,再计算表达式的值。如此重复,直至计算表达式的值为零时,结束循环。

do…while 循环类似于 while 循环,不同之处是,它们执行循环体与计算表达式的先后顺序不同。此外,从格式中可以看出,do…while 循环至少要执行一次循环体。而 while 和 for 循环中,在进行循环体之前,判断循环条件表达式,若条件不成立,则循环体一次也不执行就结束循环流程。所以这两种循环的循环体可能执行零次或若干次。

注意:在 do…while 循环中,while(表达式)后面的分号“;”不能缺少。

3.continue 语句、break 语句

除了正常结束循环的方式以外,还可以从循环中途退出而结束循环。实现该功能的语句是 break 和 continue。

在循环体使用中 continue 语句可以使程序控制退出循环。但是它的作用不是结束循环流程,是退出本次循环之后继续下一次循环。使用 continue 语句时需要注意,它使本次循环终止后,程序控制将转移至什么位置。在 while 和 do…while 循环情况下,continue 语句把程序控制转至表达式的计算处,即从计算表达式开始下一次循环。而 for 循环中是转至表达式 3 的计算。

在 switch 分支中,break 语句的功能是使程序控制退出到该结束的大括号之外。此外,break 语

句还可以用于循环体中，其功能是使程序控制从循环体中退出，从而结束循环过程。break 语句一般用在条件分支中，其作用是，在循环过程中，当某个条件成立时，就用 break 语句退出循环体，从而结束循环过程。使用循环结构时，当循环次数不能预先确定的情况下，一般采用无限循环方式。这时必须在循环体中使用 break 语句，在指定条件成立后结束循环。需要进一步强调的是，在多重循环中使用 break 语句时，控制仅仅退出包围该 break 语句的那层循环体，即一个 break 语句不能使程序控制退出一层以上的循环。

10.4 数组与字符串

10.4.1 数组的定义

1.一维数组

数组是具有一定顺序关系的若干数据的集合体，组成数组的变量称为该数组的元素变量，简称元素。C 语言中数组的元素变量用数组名后带方括号的下标表示。如：

data[5],name[18],nlist[2][3],xyx[3][3][9]

其中，带有一个方括号的称为一维数组；带有两个括号的称为二维数组；依此类推，二维和二维以上的数组统称多维数组。数组名由用户命名，其命名规则与变量名相同。方括号中的下标表示元素在数组中的位置。C 语言数组的下标从 0 开始。下标必须是整数型的常量或已赋值的变量。

由于数组是变量的集合体，所以数组也具有与变量相同的数据类型和存储类型，数组的类型就是它所有的元素变量的类型。数组在使用之前，也必须在数据说明或参数说明部分予以说明，一般格式如下：

存储类型　数据类型　数组[元素个数]

一维数组各元素的值可以通过赋值语句或输入语句得到，还可以在数组定义时对其初始化。

static int a[10]={0,1,2,3,4,5,6,7,8,9};

这是对静态数组进行初始化，static 表明是静态存储属性。C 语言规定，只能对静态数组或外部数组进行初始化。也可只给数组中部分元素初始化。在对全部元素进行初始化时，可以不指定数组的长度。

数组经过定义之后就可以使用，编译程序会为其分配一片连续的存储空间进行连续存储。C 语言中只能单个引用数组元素而不能一次引用整个数组。引用的形式如下：

数组名[下标]

其中，下标可以是整型常量或有确定值的整型表达式。

2.二维数组

二维数组定义的一般形式如下：

类型说明符　数组名[常量表达式][常量表达式]

C 语言采用上述定义方法定义一个二维数组，可以把二维数组看作是一种特殊的一维数组：该二维数组的元素又是一维数组。在 C 语言中，二维数组中元素的排列顺序是：先按行存放，再按列存放，即在内存中先顺序存放第一行的元素，再存放第二行的元素。

二维数组可以用下面几种方法进行初始化：

(1)分行为二维数组赋初值。如：

static int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};

以上赋值把第一个花括号内的数据赋给第一行元素，把第二个花括号内的数据赋给第二行元素，

依此类推，即按行赋值。

(2)可以将所有的数据写在一个花括号内，按数组排列的顺序对各元素赋值。

(3)可以对数组的部分元素赋初值。如：

```
static int a[3][4]={{1},{5},{9}};
```

以上赋值的结果是：数组第一列的元素分别赋了初值1,5,9，其余元素的值都是0。

(4)如果对二维数组的全部元素都赋初值，则定义数组时对第一维的长度可以不指定，但第二维的长度不能省。

二维数组的元素可以表示为以下形式：

数组[下标][下标]

在引用二维数组时，必须是单个元素，不能是整个数组名。下标可以是一个表达式，但不能是变量。如果下标是一个表达式，注意表达式的值不能超出数组定义的上、下限。

3.多维数组

C语言中数组的维数没有限制，多维数组的一般形式如下：

类型说明符　数组名[常量表达式][常量表达式]…

与一维数组相同，多维数组在内存中存储时，是按照其元素下标的顺序依次存储在内存的连续递增的空间中，即从第一个元素直至最后一个元素连续存储。多维数组的存储顺序是以最右边的下标最先变化为规律的。

多维数组也可以在数据说明部分进行初始化。这时要特别注意向多维数组的各个元素变量赋值的数据排列顺序。这个排列顺序必须与数组各元素在内存的存储顺序完全一致。对多维数组进行初始化，还可以采取另一种更直观的表达形式。它的出发点是根据上述的逐步分解、降低维数的方法，将一个多维数组分解成若干个一维数组，然后依次向这些一维数组赋值。为了区分各个一维数组的初值数据，可以用大括号嵌套，即每一组一维数组的初值数据再用一对大括号括起。使用这种大括号嵌套的方法，还可以像一维数组初始化那样，对于赋0值的元素，可以在对应初值数据位置上缺省该数据。

多维数组的引用形式如下：

数组名[下标][下标]…

注意：在C语言中，数组的下标都是从0开始的。下标可以是整型表达式。在对数组中元素进行引用时，特别要注意不要超出数组的界限。

10.4.2　字符串与字符数组

1.字符数组的定义

定义方法与前面介绍的类似，只是将数组定义为字符型即可。例如：

```
char c[10];
```

这里定义了一个包括10个元素的字符数组c。

2.字符数组的初始化

对字符数组初始化，可以采用以下方法：

(1)将字符逐个赋给数组中的元素；

(2)用字符串常量对字符数组进行初始化。

3.字符串

在C语言中，字符串是作为字符数组来处理的，字符串可以存放在字符型一维数组中，故可以把字符型一维数组作为字符串变量。

字符串常量是用双引号括起来的一串字符。

C 语言中约定用“\0”作为字符串的结束标志，它占内存空间，但不计入串的长度，“\0”的代码值为 0。

系统对字符串常量也自动加一个“\0”作为结束符，例如“c language”共有 10 个字符，但在内存中占 11 个字节，最后一个字节存放“\0”。

4.字符数组的输入输出

(1)逐个字符的输入输出

在字符数组中，逐个字符的输入输出，通常采用循环语句来实现，不同之处在于格式说明符为“%c”。

例如，下面的程序段可输出字符数组 str 中的字符。

```
int i;
char str[20];
scanf("%s",str);
for(i=0;i<20;i++)
printf("%c",str[i]);
```

使用说明：

①格式化输入是缓冲读，必须在接收到“回车”时，scanf 才开始读取数据。

②读字符数据时，空格、回车符都被保存进字符数组。

③当按 Enter 键时，若输入的字符少于 scanf 循环读取的字符，scanf 继续等待用户将剩下的字符输入；若输入的字符多于 scanf 循环读取的字符时，scanf 循环只读入前面的字符。

④逐个读入字符结束后，不会自动在末尾加“\0”。所以输出时，最好也使用逐个字符输出。

(2)整个字符串的输入输出

对于字符数组中，整串字符的输入输出，可采用“%s”格式符来实现。例如，下面的程序段也可一次输出字符数组 str 中的字符。

```
char str[20]="I am s student!";
printf("%s",str);
```

使用说明如下：

①格式化输入输出字符串，参数要求字符数组的首地址，即字符数组名。

②按照“%s”格式化输入字符串时，输入的字符串中不能有空格(空格、Tab)，否则空格后面的字符不能读入，scanf 函数认为输入的是两个字符串。如果输入含有空格的字符串可以使用 gets 函数。

③按照“%s”格式化输入字符串时，并不检查字符数组的空间是否够用。如果输入长字符串，可能导致数组越界，应当保证字符数组分配了足够的空间。

④按照“%s”格式化输入字符串时，自动在最后加字符串结束标志。

⑤按照“%s”格式化输入字长符串时，可以用“%c”或“%s”格式逐个输出。

⑥对于不是按照“%s”格式化输入的字符串在输出时，应该确保末尾有字符串结束标志。

10.5 函数

10.5.1 库函数的调用

C 语言提供了丰富的库函数，包括常用的数学函数、对字符和字符串处理函数、输入处理函数等。

在调用库函数时要注意以下几点：

(1)调用C语言标准库函数时，必须在源程序中用include命令，该命令的格式如下：

#include "头文件名"

include命令必须以#号开头，系统提供的头文件名都以“.h”作为后缀，头文件名用一对双引号""或一对尖括号〈〉括起来。

(2)标准库函数的调用形式如下：

函数名(参数表)

在C语言中库函数的调用可以以两种形式出现在表达式中，作为独立的语句完成某种操作。

10.5.2 自定义函数

函数的一般形式如下：

```
函数返回值的类型名  函数名(类型名  形参1,类型名  形参2,…)
{
    说明部分
    语句部分
}
```

定义的第一行是函数的首部，{}中的是函数体。

在旧的C语言版本中，函数的首部形式如下：

```
函数返回值的类型名 函数名(形参1,形参2,…)
                    形参类型说明;
```

新的ANSI标准C兼容这种形式的函数首部说明。

函数名和形参名都是由用户命名的标识符。在同一程序中，函数名必须唯一。形式参数名只要在同一函数中唯一即可，可以与函数中的变量同名。

C语言规定不能在一个函数内部再定义函数。

若在函数的首部省略了函数返回值的类型名，则函数的首部形式如下：

```
函数名(类型名 形参1,类型名 形参2,…)
```

则C默认函数返回值的类型为int类型。

当没有形参时，函数名后面的一对圆括号中的内容可以为空，但括号不能省略。

函数的类型由函数定义中的函数返回值的类型名确定，函数的类型可以是任何简单类型，如整型、字符型、指针型、双精度型等，它指出了函数返回值的具体类型。当函数返回的是整型值时，可以省略函数类型名。当函数只完成特定的操作而没有或不需要返回值时，可用类型名void(空类型)。

函数返回值就是return语句中表达式的值。当程序执行到return语句时，程序的流程就返回到调用该函数的地方(通常称为退出调用函数)，并带回函数值。

10.5.3 参数传递

在函数定义中，出现的参数名称称为形参(形式参数)，在调用函数时，使用的参数值称为实参(实际参数)。

调用函数和被调用函数之间的参数值的传递是“按值”进行的，即数据只能从实参单向传递给形参。也就是说，当简单变量作为实参时，用户不能在函数中改变对应实参的值。

10.6 变量与函数的生存期

10.6.1 局部变量与全局变量

1.局部变量

在一个函数内部所定义的变量称为局部变量,局部变量只在本函数范围内有效。

需要注意以下几点:

(1)不同函数中可以使用相同局部变量名,它们将代表不同的对象,互不干扰;

(2)一个函数的形参也是局部变量;

(3)在函数内部,复合语句也可定义变量,这些变量也为局部变量,只在此复合语句中有效。

2.全局变量

在C语言中,程序的编译单位是源程序文件,一个源程序文件中包含一个或多个函数。在函数之外所定义的变量称为外部变量,也称为全局变量。全局变量可以被包含它的源程序文件中的各个函数共用,作用域为从定义变量的位置开始到源程序文件结束,全局变量可以增加函数之间的数据的联系。

需要注意的是,当在同一个源程序文件中,全局变量与局部变量可以同名,当二者同名时,在局部变量的作用范围内,全局变量不起作用,局部变量起作用。

10.6.2 变量的存储类别、作用域及生存期

1.变量的存储类别

在C语言中,有两类存储类别:自动类别及静态类别。

与两种存储类别有关的说明符有4个,分别为auto(自动)、register(寄存器)、static(静态)和extern(外部),这些说明符一般与类型说明一起出现,一般放在类型名的左边,例如:

```
auto long I,j;
```

也可写成:

```
long auto I,j;
```

(1)自动变量

自动变量是C程序中使用最多的一种变量;这种变量的建立和撤销都是在系统中自动进行的。定义格式如下。

```
[auto] 数据类型 变量名[=初始化表达式];
```

上面的说明格式中,方括号中是可省略的部分,auto为自动类别标识符,若省略auto,系统缺省的存储类别也为自动类别。

需要注意的是,函数的形参也为自动类别,在定义时不必加存储类别标识符。

(2)寄存器变量

寄存器变量与自动变量的性质相同,其区别只在于存储的位置不同,寄存器变量存储在CPU的寄存器中,而自动变量存储在内存中的动态存储区中,寄存器变量的存取速度要快些,其定义格式如下。

```
register 数据类型 变量名[=初始化表达式];
```

上面的定义格式中,register为寄存器变量的存储类别标识符。

关于寄存器有几点说明如下:

①CPU 中寄存器的数目是有限的，因此只能把少数的变量说明为寄存器变量；

②寄存器变量是存放在寄存器中的，而不是存放于内存中，所以，寄存器变量无地址；

③寄存器变量的说明应尽量放在靠近要使用的地方，用完后尽快释放，这样可提高使用效率。

(3)静态变量

静态类别变量的存储空间在程序的整个运行期间是固定的。定义格式如下。

static 数据类型 变量名[=初始化表达式]；

在上面的说明格式中，static 为静态变量的存储类别标识符。

静态变量的初始化在编译时进行，定义时可采用常量或表达式进行显式初始化。对于没有初始化的静态变量，自动初始化为 0(整型)或 0.0(实型)。

需要注意的是，静态变量具有可继承性，这与自动变量有所不同。

(4)外部变量

使用 extern 可使外部变量使用范围扩充到需要使用它的函数。外部变量可作显式的初始化，若不作初始化，系统将自动地初始化为 0 或 0.0。定义格式如下。

[extern] 数据类型 变量名[=初始化表达式]；

上面的说明格式中，extern 使外部变量的作用范围扩大到其他源程序文件中。

需要注意的是，局部变量既可以为自动类别，也可以说明为静态类别；全局变量只能说明为静态类别。

2.变量的作用域及生存期

在 C 语言中，变量必须先声明后使用，在程序中一个已定义的变量的使用范围就是此变量的作用域。经过赋值的变量在程序运行期间能保持其值的时间范围为该变量的生存期。

(1)局部变量的使用域及生存期

①自动变量的使用域及生存期。自动变量的存储单元被分配在内存的动态存储区，每当进入函数体(或复合语句)时，系统自动为自动变量分配存储单元，退出时自动释放这些存储单元。自动变量的作用域为从定义的位置起，到函数体(或复合语句)结束为止。

自动变量在进入到定义它们的函数体(或复合语句)时生成，在退出所有的函数体(或复合语句)时消失，这就是自动变量的生存期。

使用自动变量的优点是使各函数之间形成信息分隔，不同函数中使用同名变量时不会相互影响。

②寄存器变量的使用域及生存期，寄存器变量的使用域及生存期与自动变量相同。

③静态存储类别的局部变量。在函数体(或复合语句)内部，用 static 说明的变量为静态存储类别的局部变量，这种变量的作用域与自动(或寄存器)变量的作用域相同，但是生存期有所不同。

在整个程序运行期间，静态局部变量在内存的静态存储区中占据着永久的存储单元，甚至在退出函数后下次再进入函数时，静态局部变量仍使用原来的存储单元。由于不释放存储单元，所以这些存储单元中的值将会被保留下来。静态局部变量的生存期将一直延长到程序运行结束。

(2)全局变量的作用域及生存期

全局变量的作用域为从变量定义的位置开始，到整个源程序文件结束为止。生存期为整个程序的运行期间。

注意：全局变量在整个程序运行期间占用内存空间。全局变量必须在函数以外定义，因而降低了函数的通用性，影响函数的独立性。使用全局变量时，当全局变量的值意外改变时，会引起副作用，这种错误一般难以查找。

①在同一编译单位内用 extern 标识符来扩展全局变量的作用域。当全局变量定义后，在引用函数前，应在引用它的函数中用 extern 对此全局变量进行说明，以便使编译程序能确定此外部变量已被

定义，不必再为它分配存储单元，此时的作用域从 extern 说明开始，到此函数结束。

全局变量的定义只能出现一次，这时不可使用 extern 标识符。如果多次引用全局变量，这时必须用 extern 标识符。

②在不同编译单位内用 extern 标识符来扩展全局变量的作用域。C 语言中的不同函数可以存放在不同的源程序文件中，每个源程序文件可以单独进行编译，进行语法检查，再生成目标文件，最后用系统提供的连接程序把多个目标文件连接成一个可执行程序。

当程序由多个源程序文件组成时，若每个文件中都要引用一个全局变量，这时如在每个源程序文件中都定义一个所需的同名全局变量，在文件连接时将产生重复定义错误。解决办法是在其中一个源程序文件中定义所有的全局变量，而在其他用到全局变量的源程序文件中用 extern 对这些变量进行说明，以表明它们已在其他编译单元中被定义。

③静态全局变量。当用 static 标识符说明全局变量时，全局变量为静态全局变量。静态全局变量只能在定义它的源程序文件中使用，不能被其他源程序文件使用。

10.6.3 内部函数与外部函数

根据函数能否被其他源文件调用，可以将函数分为内部函数与外部函数。

1.内部函数

如果限定在一个源文件中定义的函数只能被本文件中的函数调用，则称这种函数为内部函数。其定义的一般格式如下：

static 类型说明符　函数名(形参表){函数体}

内部函数也称为静态函数。但此处静态 static 的含义已不是指存储方式，而是指对函数的调用范围只限于本文件。因此在不同的源文件中定义同名的静态函数不会引起混淆。

2.外部函数

在定义函数时，如果在函数名前加上关键字 extern，则表示该函数是外部函数。其定义的一般格式如下：

extern 类型说明符　函数名(形参表){函数体}

外部函数在整个源程序中都有效，它可以被其他文件调用。如果在函数定义中没有说明 extern 或 static，则默认为 extern。

在一个源文件的函数中调用其他源文件中定义的外部函数时，应用 extern 说明被调函数为外部函数。

10.7 指针

10.7.1 指针的基本概念

在 C 语言中，指针是指一个变量的地址，通过变量的地址“指向”的位置找到变量的值，这种“指向”变量地址可形象地看作“指针”。用来存放指针的变量称为指针变量，它是一种特殊的变量，它存放的是地址值。

定义指针变量的一般形式如下：

类型名　*指针变量 1,*指针变量 2,…;

“类型名”称为“基类型”，它规定了后面的指针变量中存放的数据类型；“*”号表明后面的“指针变量 1”、“指针变量 2”等是指针变量，“*”号在定义时不能省略，否则就会变成一般变量的定义了；

“指针变量 1”、“指针变量 2”等称为指针变量名。

一个指针变量只能指向同一类型的变量。

以下是与指针和指针变量有关的两个运算符。

*：指针运算符(或称“间接访问”运算符)。

&：取地址运算符。

通过 * 号可以引用一个存储单元，如有如下定义：

```
int i=123, *p,k;
```

则 p=&i;k=*p;或 k=*&i;都将变量 i 中的值赋给 k。

*p=10;或 *&i=10;都能把整数 10 赋给变量 i。这里，等号左边的表达式 *p 和 *&i 都代表变量 i 的存储单元。

10.7.2 指针的使用

1.变量、数组、字符串、函数、结构体的指针以及指向它们的指针变量

(1)变量的指针和指向变量的指针变量

变量的指针即变量的地址。指向变量的指针变量，即声明一个指针变量，用以存放其他变量的地址，使该指针变量指向某个变量。

(2)数组的指针和指向数组的指针变量

所谓数组的指针是指数组的起始地址，数组元素的指针是数组元素的地址。C 语言规定数组名代表数组的首地址，也就是第一个元素的地址。

(3)字符串的指针和指向字符串的指针变量

可以通过定义声明一个指针指向一个字符串。C 语言将字符串隐含处理成一维字符数组，但数组的每个元素没有具体的名称，这一点跟字符数组不一样。要引用字符串中的某个字符，只能通过指针来引用，例如：*(s+0)，*(s+1)，…，*(s+n)。

(4)函数的指针和指向函数的指针变量

指向函数的指针变量的一般形式如下：

```
数据类型标识符 (*指针变量名)();
```

这里的“数据类型标识符”是指函数返回值的类型。

函数的调用可以通过函数名调用，也可以通过函数指针调用(即用指向函数的指针变量调用)。

指向函数的指针变量表示定义了一个指向函数的指针变量，它不是固定指向哪一个函数，而只是定义了这样的一个类型变量，专门用来存放函数的入口地址。在程序中把哪一个函数的地址赋给它，它就指向哪一个函数。在一个程序中，一个指针变量可以先后指向不同的函数。

在给函数指针变量赋值时，只需给出函数名而不必给出参数。因为函数指针的值仅是函数的入口地址，而不涉及实参与形参的结合问题。

对指向函数的指针变量，表达式 p+n，p++，p--等都无意义。

(5)结构体的指针是指向结构体的指针变量

一个结构体变量的指针就是该变量所占据的内存段的起始地址。可以定义一个指针变量，用来指向一个结构体变量，此时该指针变量的值是结构体变量的起始地址。指针变量也可以用来指向结构体数组中的元素。

2.用指针作函数参数

函数的参数不仅可以是整型、实型、字符型等数据，还可以是指针类型，它的作用是将一个变量的地址传送到另一个函数中。

3.返回指针的指针函数

一个函数可以返回一个整型值、字符值、实型值等，也可以返回指针型的数据，即地址这种带回指针值的函数，一般的定义形式如下：

类型标识符　*函数名(形参表)

4.指针数组、指向指针的指针

指针数组指的是一个数组，其元素均为指针类型数据，也就是说，指针数组中的每一个元素都是指针变量。指针数组的定义形式为：

类型标识　*数组名[数组长度说明]

指针数组可以使字符串处理更加方便。

指向指针的指针是指指向指针的指针变量，一个指向指针数据的指针变量的一般形式如下：

类型标识 * *p;

10.7.3　main 函数的命令参数

指针数组的一个重要应用是作为 main 函数的形参，一般来说，main 函数后的括号中是空的，即没有参数。实际上 main 可以有参数，如：

main(argc,argv)

其中，argc 和 argv 就是 main 函数的形参。其他函数形参的值可以通过函数调用语句的实参得到，由于 main 函数是由系统调用的，因而 main 函数的形参值不能从程序中得到，但可以在操作系统状态下，将实参与命令一起给出，从而使 main 函数的形参得到值。命令行的一般形式如下：

命令名　参数 1　参数 2…参数 n

命令名和各参数之间用空格分隔开。

10.7.4　动态存储分配

在 C 语言中有一种称为“动态存储分配”的内存空间分配方式：程序在执行期间需要存储空间时，通过“申请”分配指定的内存空间；当闲置不用时，可随时将其释放，由系统另作他用。下面介绍 C 语言中动态分配系统的主要函数 malloc()、calloc()、free()及 realloc()。使用这些函数时，必须在程序开始包含文件 stdlib.h。

1.主内存分配函数 malloc()

函数格式如下：

void * malloc(unsigned size);

函数功能：从内存中分配一大小为 size 字节的块。

参数说明：size 为无符号整型，用于指定需要分配的内存空间的字节数。

返回值：新分配内存的地址，如无足够的内存可分配，则返回 NULL。

说明：当 size 为 0 时，返回 NULL。

2.主内存分配函数 calloc()

函数格式如下：

void * calloc(unsigned n,unsigned size);

函数功能：从内存中分配 n 个同一类型数据项的连续存储空间，每个数据项的大小为 size 字节。

参数说明：n 为无符号整型，用于指定分配的数据项的个数；size 为无符号整型，用于指定需要分配的数据项所占内存空间的字节数。

返回值：新分配内存的地址，如无足够的内存可分配，则返回 NULL。

3.重新分配内存空间函数 realloc()

函数格式如下：

```
void * realloc(void * block,unsigned size);
```

函数功能：将 block 所指内存区的大小改为 size 字节。

参数说明：block 为 void 类型的指针，指向内存中某个块；size 为无符号整型，用于指定需要分配的内存空间的字节数。

返回值：新分配内存的地址，如无足够的内存可分配，则返回 NULL。

4.释放内存函数 free()

函数格式如下：

```
void free(void * block);
```

函数功能：将 calloc()、malloc()及 realloc()函数所分配的内存空间释放为自由空间。

参数说明：block 为 void 类型的指针，指向要释放的内存空间。

返回值：无。

10.7.5 字符串指针

1.用指针方法实现一个字符串的存储和运算

例如：

```
char * strp="china";
```

此处定义了一个字符指针变量 strp，变量中存放的是字符串第一个字符的地址。

C 语言对字符串常量是按字符数组处理的，它实际上在内存开辟了一个字符数组用来存放字符串变量，并把字符串首地址赋给字符指针变量 strp。

在输出时用 printf 函数，其形式如下：

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

通过字符数组名或字符指针变量可以输出一个字符串。而对一个数值型数组，是不能企图用数组名输出它的全部元素的。

2.字符指针变量与字符数组

虽然用字符数组和字符指针变量都能实现字符串的存储和运算，但它们二者之间是有区别的，不应混为一谈，主要有以下几点：

(1)字符数组由若干个元素组成，每个元素中放一个字符，而字符指针变量中存放的是地址(字符串的首地址)，绝不是将字符串放到字符指针变量中。

(2)对字符数组只能对各个元素赋值，不能用以下办法对字符数组赋值。

```
char str[14];
str="I love China!";
```

而对字符指针变量，可以采用下面方法赋值：

```
char  * a;
a="I love China!";
```

但要注意，赋值给 a 的不是字符，而是字符串的首地址。

(3)赋初值时，对以下的变量定义和赋初值。

```
char * a="I love China!";
```

等价于：

```
char * a;
```

a="I love China!";

而对数组初始化时：

static char str[14]={"I love China!"};

不能等价于：

char str[14];

str[]={"I love China!"};

即数组可以在变量定义时整体赋初值，但不能在赋值语句中整体赋值。

(4)在定义一个数组时，在编译时即已分配内存单元，有确定的地址。而定义一个字符指针变量时，给指针变量分配内存单元，在其中可以放一个地址值，也就是说，该指针变量可以指向一个字符型数据，但如果未对它赋予一个地址值，这时该指针变量并未具体指向某一个字符数据。

(5)指针变量的值是可以改变的。

3.字符串处理函数

C语言中没有对字符串进行合并、比较和赋值的运算符，但几乎所有版本的C语言中都提供了有关的库函数。常用的字符串处理函数如下。

strcat函数：连接两个字符数组中的字符串。

strcpy函数：字符拷贝函数。

strcmp函数：字符比较函数。

strlen函数：求字符串长度的函数。

strlwr函数：将字符串中大写字母转换成小写字母。

strupr函数：将字符串中小写字母转换成大写字母。

10.8 宏定义与条件编译

10.8.1 宏定义

宏定义就是用标识符来代表一个字符串，即给字符串取个名字。C语言用“#define”进行宏定义。C语言编译系统在编译前将这些标识符替换成所定义的字符串。

C语言的宏定义有两种形式：不带参数的宏定义和带参数的宏定义。

1.不带参数的宏定义

它的一般形式如下：

#define　标识符　字符串

其中“标识符”称之为“宏名”。宏名通常可用大写字母表示，以便与程序中的其他变量名相区别。字符串也称宏体，外面不加双引号。各部分之间用空格分开，最后以换行结束。

通常在程序中使用宏定义，即把常量用有意义的符号代替，不但可以使程序更加清晰，容易理解，而且当常量值改变量，不需在整个程序中查找、修改，只要改变宏定义就可以。

2.带参数的宏定义

带参数的宏定义不只是进行简单的字符串替换，还要进行参数替换。它的一般格式如下：

#define　宏名(参数表)　字符串

带参数宏定义的格式类似于函数头，不同之处在于它没有类型说明，参数也不需要进行类型说明。

10.8.2 条件编译

利用条件编译命令可以按不同的条件去编译不同的程序部分，因而产生不同的目标代码文件，这对于程序的移植和调试是很有用的。条件编译有以下 3 种形式。

1.第一种形式

```
#ifdef  标识符
    程序段 1
#else
    程序段 2
#endif
```

它的作用是：如果标识符已被 #define 命令定义过，则对“程序段 1”进行编译，否则对“程序段 2”进行编译。

2.第二种形式

```
#ifndef  标识符
    程序段 1
#else
    程序段 2
#endif
```

与第一种形式的区别是将 ifdef 改为 ifndef。

它的作用是：如果标识符未被 #define 命令定义过，则对“程序段 1”进行编译，否则对“程序段 2”进行编译。这与第一种形式的功能正好相反。

3.第三种形式

```
#if  常量表达式
    程序段 1
#else
    程序段 2
#endif
```

它的作用是：如果常量表达式的值为真(非 0)，则对“程序段 1”进行编译，否则对“程序段 2”进行编译。它可以使程序在不同条件下，完成不同的功能。

条件编译允许只编译源程序中满足条件的程序段，使生成的目标程序较短，从而减少了内存的开销，提高了程序的执行效率。

10.9 结构体与共用体

10.9.1 结构体和共用体

结构体和共用体(又称联合体，共同体)是在 C 语言中使用的另一类数据的构造类型，它们是由不同数据类型数据组成的集合体。在数据处理中，经常需要处理不同数据类型的集合，这种集合称为结构体，简称结构。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，首先要对结构体的组成进行描述，这称为结构的定义。结构的定义是声明该结构由几个成员项组成以及每个成员项的数据类型。结构定义的一般形式如下：

```
struct  结构名
  {
    数据类型  成员名 1;
    数据类型  成员名 2;
        ……
    数据类型  成员名 n;
  };
```

结构的定义以保留字 struct 作为标识符,其后是定义结构名,它们两者形成了特定结构的类型标识符。结构名由用户命名、命名原则与变量名等组成。在结构名下面的大括号中包围的是组成结构的各个成员项。每个成员项由其数据类型和成员名组成。每个成员项后用分号作为结束符。整个结构的定义也用分号作为结束符,注意不要忘记这个分号。

在程序中,结构的定义可以在函数内部也可以在所有函数的外部。在函数内部定义的结构,其可见性仅限于该函数内部,而函数外部定义的结构,在所有函数中都是可见的。

1.关于结构体类型与结构体变量

(1)类型与变量是不同的概念,对结构体变量来说,在定义时一般先定义一个结构体类型,然后定义变量为该类型。只能对变量赋值、存取或运算,而不能对一个类型赋值、存取或运算。在编译时,对类型是不分配空间的,只对变量分配空间。

(2)对结构体成员,可以单独使用,它的作用与地位相当于普通变量。

(3)成员也可以是一个结构体变量。

某个结构体一经定义之后,就可以定义该结构体的变量,这称为结构变量的声明,它的一般形式如下:

```
struct  结构名  结构变量名;
```

结构体的说明在程序的数据说明部分给出。由于结构体变量使用内存空间,所以它们也具有一定的存储类型。结构体变量的存储类型概念、寿命、可见性及使用范围与普通变量、数组等完全一致。在函数内部说明的结构体变量,可以是 auto 型(一般缺省)和 static 型,由于 register 型占用数目有限的寄存器,所以一般不用于存储结构体数据。auto 型结构变量是局部变量,它的寿命和可见性仅限于被说明的函数内部。内部 static 型结构变量具有局部可见性和全局寿命。在所有函数外部说明的结构变量可以是缺省存储类型,它具有全局寿命,在同一编译单位的所有函数中都是可见的。

在引用结构体变量时应注意以下几个方面的问题:

①结构体变量不能作为整体进行赋值和输出(除了将一个结构体变量直接赋给另一个具有相同结构的结构体变量),只能对结构体变量中各个成员分别访问。

②如果成员本身又属于一个结构体类型,则要用若干个成员运算符,一级一级地找到最低一级的成员。只能对最低级成员进行赋值或存取及运算。

③对成员变量可以施加该成员类型所允许的所有运算。

④可以引用结构体变量的地址,也可以引用结构体变量中成员的地址。引用它们的地址主要是用于函数参数、传递结构体的地址或对成员进行输入。

在程序中,结构的说明必须在该结构定义之后,对于尚未定义的结构,不能用它对任何结构变量进行说明。结构的定义和说明也可以同时进行。这时,被说明的结构变量直接在结构定义的大括号后给出。

结构体在程序中有着独特的使用形式。此外,在结构说明的同时可以给各个成员项赋值,即结构体的初始化。

结构体是不同数据类型的若干数据的集合体。在程序中使用结构体时，一般情况下不能把结构体作为一个整体参加数据处理，而参加各种运算和操作的是结构体的各个成员项数据。

在结构体说明的同时，可以给它的每个成员项赋初值，这称为结构体的初始化。结构体初始化的一般形式如下：

```
struct  结构名  结构变量
{初始数据};
```

结构体变量只能对存储属性是外部变量或静态变量进行初始化。不能对动态变量进行初始化。在初始化时，按照所定义的结构体类型的数据结构，依次写出各初始值，在编译时就将它们赋值给此变量中的各成员。

2.共用体

在C语言中，不同数据类型的数据可以使用共同的存储区域，这种数据构造类型称为共用体，共用体在定义、说明和使用形式上与结构体相似。两者本质上的不同仅在于使用内存的方式上。共用体的定义形式一般如下：

```
union  共用体名
  {
    数据类型  成员名1;
    数据类型  成员名2;
    ……
    数据类型  成员名n;
  };
```

共用体的定义制定了共用体的组成形式，同时指出了组成共用体的成员具有的数据类型。与结构的定义相同，共用体的定义并不为共用体分配具体的内存空间，它仅仅说明了共用体使用内存的模式。

共用体的特点如下：

(1)同一个内存段可以用来存放几种不同类型的成员，但在每一时刻只存放其中的一种，而不能同时存放几种。

(2)共用体变量中起作用的成员是最后一次存放的成员，在存放一个新成员后原有的成员就失去作用。

(3)共用体变量的地址和它的各级成员的地址是同一地址。

(4)不能在定义共用体时对它进行初始化。

(5)不能把共用体变量作为函数的参数，也不能带回共用体变量，但可以使用指向共用体变量的指针。

(6)共用体变量可以出现在结构类型定义中，也可以定义共用体数组，结构体中也可以出现共用体。

3.结构数组

在C语言中，具有相同数据类型的数据可以组成数组，指向相同类型的指针可以组成指针数组。根据同样的原则，具有相同结构的结构体也可以组成数组，称它们为结构数组。结构数组的每一个元素都是结构变量。

结构数组的说明形式如下：

```
struct  结构名  结构数组中[元素个数];
```

结构数组在说明的同时也可以进行初始化。它的一般形式如下：

```
struct  结构名  结构数组名[]
```

={初始数据};

在对结构数组进行初始化时,方括号中元素个数的指定可以缺省。由于结构体是由若干不同类型的数据组成的,而结构数组又由若干结构体组成。所以要特别注意包围在大括号中的初始数据的顺序以及它们与各个成员项间的对应关系。

与结构的初始化规定相同,在函数内部对结构数组进行初始化时,必须指明为 static 型。在函数外部进行初始化的结构数组可以是缺省存储类型或 static 型的。

4.自定义数据类型

在 C 语言程序中,用户可以定义自己命名的数据类型,这称为类型定义。类型定义语句形式如下:

typedef type name;

其中,typedef 是类型定义的保留字,type 是系统提供的标准类型名或已经定义的其他类型名,而 name 就是用户定义的类型名。typedef 语句的功能是把 name 定义为与 type 相同的类型。

类型定义在形式上类似于编译预处理的宏定义,但它们在本质上是不同的。类型定义不是编译预处理语句,所以对它的处理不是在编译之前,而是在编译过程中。用户定义的类型增加了程序中使用的类型名称,而不是像宏定义那样存在着字符串置换功能。此外,在表现形式上,类型定义语句前面不能有#符号,并且语句必须用分号作为结束符。

通过类型定义,用户可以使用赋予更明确意义的类型名代替系统提供的类型名。需要清楚地了解,类型定义并没有创造任何一种新的数据类型。

10.9.2 链表操作

1.结构指针

在 C 语言中有各种不同用途的指针,如指向数组数据的指针、指向指针数据的指针和指向函数的指针等。指针在数据处理和函数数据传递中起着十分重要的作用。在 C 语言中对结构体的数据也可以使用指针进行处理。我们把指向结构体的指针称为结构指针。

结构指针是一个指针变量,它保持着结构体的存储地址。结构指针与前面介绍的各种指针在特性和使用方法上完全相同。例如结构指针的存储类型以及它的寿命和可见性等与前面介绍的相同。此外,结构指针的运算也是按照 C 语言的地址计算原则进行的。例如,结构指针加一将指向内存中下一个结构体,结构指针自身地址值的增加量取决于它所指向结构的数据长度。结构指针指向的结构体称为它的目标结构体。与前面相同,在程序中结构指针也是通过访问目标运算符“*”访问它的目标结构体。

结构指针在程序中说明的一般形式如下:

struct 结构名 *结构指针名

其中,结构名必须是已经定义过的结构。使用结构指针时,该指针指向结构体的成员项一般表示形式如下:

(*结构指针名).成员名

在这种表示形式中总是需要使用圆括号,显得很不简练。因此,C 语言中对于结构指针指向的结构成员项,给出了另外一种简洁的表示方法,这种方法如下:

结构指针名→成员名

它与前一种表示方法在意义上是完全等价的。

2.结构体在函数间的传递

在调用函数时,可以把结构体作为参数传递给函数。由于结构体是多个数据的集合体,当把它们

传递给函数时，早期的编译系统都不能把结构体整体作为一个参数复制到函数中去。因此，使用这种编译系统时，采用数据复制方式，只能把每个成员数据作为一个个的参数传递到函数中。一般不采用这种方式。

与数组在函数间传递一样，结构体传递给函数时，一般采用地址传递方式，即把结构体的存储地址作为参数向函数传递，函数中用指向相同结构类型的指针接收该地址值。然后，在函数中通过这个结构指针来处理结构体中的各项数据。

需要进一步说明的是，当前较新的C语言编译系统，如美国国家标准C语言(87ANSI标准C)，把结构变量作为普通变量一样处理。

与普通变量以数据传递方式在函数间传递的特性相同，在调用的函数中，作为形式参数的结构体中任何成员项的数值变动，不会影响用它的函数中作为实参数的相应结构体的任何数值。

3.结构型函数和结构指针型函数

函数具有不同的数据类型，它们是由函数返回值的数据类型决定的。结构也是一种数据构造类型。当函数的返回值是结构变量时，该函数就是结构型函数。而函数的返回值是结构体地址时，它就是结构指针型函数。

(1)结构型函数

在C语言中，可以把具有给定结构的结构变量作为函数的返回值。这时该函数的数据类型就是给定的结构型。

定义结构型函数的一般形式如下：

```
struct  结构名  函数名()
```

程序中调用结构型函数时，应该在调用函数的数据说明部分对此函数进行说明。并且，用于接收函数返回值的量，必须是具有相同结构类型的结构变量。

(2)结构指针型函数

C语言中，结构体的存储首地址可以作为函数的返回值传递给调用它的函数。返回值为结构体地址的函数就是结构指针型函数。

定义结构指针型函数的一般形式如下：

```
struct  结构名  * 函数名();
```

在程序中使用结构指针型函数时，应该对它进行说明。并且用于接收函数返回值的量必须是指向相同结构的结构指针。

结构指针型函数是指用地址传递方式调用它的函数返回结构体的数据。采用这种方式，不仅可以返回某个结构体的地址，还可以返回结构数组的地址，从而把函数中处理的若干结构体的数据返回给调用它的函数。作为结构指针型函数，其返回值也可以是结构指针。

当前使用的C语言编译系统中，大多数都支持结构指针型函数的使用。

4.结构嵌套

当一个结构体的成员项是结构体时，就形成了结构嵌套。在数据处理中有时要使用结构嵌套处理组织结构比较复杂的数据集合。结构嵌套中最内层的成员项一般表现形式如下：

```
结构变量名.外层成员名.内层成员名
```

其中，外层成员名是具有内层结构的结构变量名。C语言对结构嵌套层次没有限制，但是层次过多的嵌套使数据组织变得十分复杂，在程序中并不多见。此外，在使用结构嵌套时，内层结构的定义必须在外层结构定义之前，否则将发生错误。

在C语言中，结构体可以自己嵌套自己，即结构体的成员项仍是具有该结构类型的结构体，这种结构体称为自嵌套结构体，或称为递归结构体。

5.位字段结构体

计算机应用于过程控制、参数检测和数据通信等领域时，要求应用程序具有对外部设备接口硬件进行控制和管理的功能。它们经常使用的控制方式是向接口发送方式字或命令字以及从接口读取状态字等。与接口有关的命令字、方式字和状态字是以二进制位(bit)为单位的字段组成的数据，一般将它们称为位字段数据。

位字段结构定义中的每个成员项一般表示形式如下：

unsigned　成员名:位数;

需要指出的是，位字段结构体在存储时使用的内存空间大小与 int 型数据相同，即使位字段结构体中各成员项的位数总和小于 int 型的位长，它也占用一个 int 型位长的内存空间。当成员项总位长超过 int 型时，它将占用下一个连续的 int 位长空间。但是不允许任何一个成员项跨越两个 int 空间的边界。此外，按这种组织方式定义的结构中，多余的字段一般也把它作为一个不使用的成员项。所以，使用位字段结构体的 C 语言，在可移植性上有一定的局限性。

接收位字段结构数据时，应该使用指针变量，而把变量的值赋给位字段结构体时，应该使用结构指针。

10.10　位运算

10.10.1　位运算的相关概念

1.位的概念

大多数计算机系统的内存储器是由众多的存储单元构成的。在微机中，每个存储单元是 1 个字节，它由 8 位二进制数构成，可以表示 $2^8=256$ 种信息，各位的编号从 0～7，最左边的位(第 7 位)是最高位，最右边的位(第 0 位)是最低位。由于二进制本身的特点，各位上的数字不是 1，就是 0。

2.数的编码

数在计算机中是以二进制表示的，但是它并不简单地以它本身的数值的二进制形式来直接表示，而要进行一定的编码，以方便计算机进行处理。常用的编码有原码、反码、补码 3 种。

3.真值与原码

将一个十进制数的二进制表示称为这个十进制的真值，它代表了这个十进制数本身的数值。如表10－7所示列出了一些数的真值。

表 10—7　　真值举例

数	二进制表示	真值(16 位)
0	0	0000000000000000
1	1	0000000000000001
⋮	⋮	⋮
7	111	0000000000000111
⋮	⋮	⋮
15	1111	0000000000001111
⋮	⋮	⋮
255	11111111	0000000011111111
⋮	⋮	⋮
4095	111111111111	0000111111111111
65535	1111111111111111	1111111111111111

用真值表示的数只能是正数，对于负数，要用“－”号标明，例如：

－7 的真值为－0000000000000111

－65535 的真值为－1111111111111111

这势必造成用计算机表示数时的不便，故引入了原码表示法。

在原码表示法中，最高位代表符号位，用 1 表示负数，0 表示正数；其他数位用来表示真值的绝对值。

数字零存在着两种表示方法：＋0 与－0。

4.反码

若采用反码表示，则对应的原码应按照以下方法进行转换：

(1)如果真值为正，则它的反码与原码相同；

(2)如果真值为负，则反码的符号位为 1，其余各位就是对原码取反(即原码的 1 变为 0，原码的 0 变为 1)。

5.补码

(1)补码具有的优点：首先它可以变减法运算为加法运算，使得计算时步骤统一，速度提高；其次，在这种系统下的 0 只有唯一的一种表示方法，这就是现代的计算机系统中大多采用补码的原因。

(2)补码的规定如下：

①正数的原码、补码、反码均相同；

②计算负数的补码时，先置符号位为 1，再对剩余原码的位数逐位取反，最后对整个数加 1。

在微机上以 8 位二进制数为一字节的存储单元中采用补码系统，它可以存放的最小整数为－128，最大整数为＋127。若采用两个字节来表示一个整数，则可表示最小整数为－32768，最大整数为＋32767。

10.10.2 简单位运算

C 语言提供了位运算的功能，这使它能像汇编语言一样用来编写系统程序。位运算符共有 6 种，如表 10—8 所示。

表 10—8　　位运算符

位运算符	含义
&	位与
|	位或
∧	位异或
～	位取反
<<	位左移
>>	位右移

下面介绍前 4 种运算，即位的逻辑运算。

1.按位与运算

(1)概念

按位与运算符“&”是双目运算符，其功能是将参与运算的两数各对应二进位相与。只有对应的两个二进位均为 1 时，结果位才是 1，否则为 0。

(2)按位与运算的特殊用途

①清零。如果想将一个存储单元清零，即使其全部二进位为 0，可按如下的方法计算。

找一个数，它的补码形式中各位的值符合如下条件：原来的数中为 1 的位，新数中相应位为 0(注

意，并不要求原数为 0 的位上，新数相应位为 1，新数相应位可以是 0 或 1)；对二者进行位与运算。

②取一个数中某些字节。对于一个整数 a(占 2 个字节)，如果想要得到其中的低字节，只需将 a 与特定的一个数按位与即可。

③要想将一个数的某一位保留下来，可将该数与一个特定的数进行位与处理。

2.按位或运算

按位或运算符“|”是双目运算符，其功能是将参与运算的两数各对应的二进位相或。只要对应的两个二进位有一个为 1 时，结果位就为 1。参与运算的整数均以补码形式出现。

3.按位异或运算

(1)概念

按位异或运算符“∧”是双目运算符，其功能是将参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为 1，否则为 0。参与运算整数仍以补码形式出现。

(2)异或运算的特殊应用

①使特定位翻转。

②与 0 相“异或”，保留原值。

③交换两个值。

4.取反运算

取反运算符“～”是一个一元运算符，即参与运算的数只有一个，用来对一个二进制数按位取反，即将 0 变 1，1 变 0。比如说，～025 就是对八进制数 25(即二进制数 000000000010101)按位求反的结果。

10.10.3 移位运算

1.移位运算符

移位运算符对操作数以二进制为单位进行左移或右移，如表 10—9 所示。

表 10—9　　移位操作

运算符	名称	例子	运算功能
>>	右移位	b>>3	b 右移 3 位
<<	左移位	c<<2	c 左移 2 位

2.左移运算

左移运算符“<<”是双目运算符，其功能把“<<”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补 0。例如：

a<<4

就把 a 的各二进位向左移动 4 位。如 a=00000011(十进制数 3)，左移 4 位后 00110000(十进制数 48)。

3.右移运算

右移运算符“>>”是双目运算符，其功能是把“>>”左边的运算数的各二进位全部右移若干位，由“>>”右边的数指定移动的位数。例如：a=15，a>>2；表示把 0000001111 右移为 00000011(十进制数 3)。应该说明的是，对于有符号数，在右移时，符号位将随同移动。当为正数时，最高位补 0；而为负数时，符号位为 1，最高位是补 0 或 1 取决于编译系统的规定。Turbo C 规定补 1。

右移运算相当于运算对象除 2。

4.位赋值运算

位赋值运算符如表 10—10 所示。

表 10—10　　位赋值运算符

运算符	名称	例子	等价于
&=	位与赋值	a&=b	a=a&b
|=	位或赋值	a|=b	a=a|b
∧=	位异或赋值	a∧=b	a=a∧b
>>=	右移赋值	a>>=b	a=a>>b
<<=	左移赋值	a<<=b	a=a<<b

位赋值运算的过程如下：

(1)先对两个操作数进行位操作；

(2)然后把结果赋予第一个操作数，因此第一个操作必须是变量。

位赋值运算与算术赋值运算相似，它们都统称复合赋值运算。

10.11　文件操作

10.11.1　文件类型指针

C 语言中的文件分缓冲型文件和非缓冲型文件两种，此处只讨论缓冲型文件。对于缓冲型文件，每个被使用的文件都在内存中开辟一个区，用来存放文件的有关信息(如文件名字、文件状态及文件当前位置等)。这些信息保存在有关结构体类型的变量中。该结构体类型由系统定义，取名为 FILE。

10.11.2　文件的打开与关闭

与其他高级语言一样，C 语言对文件读写前应该“打开”该文件，在使用结束之后应“关闭”该文件。

C 语言中打开文件用 fopen()函数，该函数的调用方式通常为：

```
FILE　*fp;
fp=fopen(文件中,使用文件方式);
```

文件的操作方式如表 10—11 所示。

表 10—11　　文件操作方式

操作方式	属性	操作方式的功能
rrr	只读	为输入打开一个字符文件
www	只写	为输出打开一个字符文件
AaA	追加	向字符文件尾增补数据
rb	只读	为输入打开一个二进制文件
wb	只写	为输出打开一个二进制文件
ab	追加	向二进制文件尾增补数据
r+	读写	为读/写打开一个字符文件
w+	读写	为读/写打开一个新的字符文件
a+	读写	为读/写打开一个字符文件

续表

操作方式	属性	操作方式的功能
rb+	读写	为读/写打开一个二进制文件
wb+	读写	为读/写打开一个新的二进制文件
ab+	读写	为读/写打开一个二进制文件

一般使用如下方法打开一个文件：

```
if((fp=fopen("file","r"))==NULL)
{   printf("Cannot open the file! \n")};
  exit(0);
}
```

也就是当执行 fopen 函数时，如能顺利打开，则将此文件信息区的起始地址赋给指针变量 fp；如打开失败，则 fp 的值为 NULL，在屏幕上显示“Cannot open the file!”，然后执行 exit(0)。

对打开的文件使用完后用 fclose 函数关闭它，调用该函数的一般形式如下：

```
fclose(文件指针);
```

10.11.3 文件的读写

文件打开以后，就可以对它进行读写操作了，对文件的读写可以用以下几个函数实现。

1.fputc 函数和 fgetc 函数

fputc 函数把一个字符写到磁盘文件去，其一般形式如下：

```
fputc(ch,fp);
```

其中，ch 是要输出的字符，它可以是一个字符常量，也可以是一个字符变量；fp 是文件指针变量，它从 fopen 函数得到返回值。

fgetc 函数从指定的文件读入一个字符，该文件必须是以读或写方式打开的，其一般调用形式如下：

```
ch=fgetc(fp);
```

其中，fp 为文件型指针变量，ch 为字符变量。fgetc 函数返回一个字符，赋给 ch。

2.fread 函数和 fwrite 函数

fread 函数可以用来一次读入一组数据。fwrite 函数可以用来一次向文件中写一组数据。这两个函数的调用形式分别如下：

```
fread(buffer,size,count,fp);
fwrite(buffer,size,count,fp);
```

其中，buffer 是一个指针，size 为要读写的字节数，count 为要进行读写多少 size 字节的数据项，fp 为文件型指针。

3.fprintf 函数和 fscanf 函数

fprintf 函数、fscanf 函数与 printf 函数、scanf 函数作用相似，都是格式化读写函数，不同的是，fprintf 函数和 fscanf 函数的读写对象不是终端而是磁盘文件。这两个函数的调用形式分别如下：

```
fprintf(文件指针,格式字符串,输出列表);
fscanf(文件指针,格式字符串,输出列表);
```

4.rewind 函数和 fseek 函数

文件中有一个位置指针，指向当前的读写位置。如果顺序读写一个文件，每次读写一个字符，则读写完字符后，该位置指针自动指向下一个字符位置。如果想改变这样的规律，强制使位置指针指向

其他指定的位置，可以用调用形式 rewind 和 fseek 函数。

rewind 函数的作用是使位置指针重新返回文件的开头，此函数没有返回值。

fseek 函数可以改变文件的位置指针，其调用形式如下：

fseek(文件类型指针，位移量，起始点)；

10.11.4 文件的定位

文件位置指针是一个形象化的概念，用于表示当前读或写的数据在文件中的位置。

1.rewind 函数

rewind 函数又称为“反绕”函数，其调用形式如下：

rewind(fp)；

此函数无返回值，其功能为使文件的位置指针回到 fp 所指的文件开头处。

2.fseek 函数

fseek 函数用来移动文件位置指针到指定的位置上，使其以后的读写操作从此位置开始，其调用形式如下：

fseek(fp,offset,origin)；

其中，各参数的说明如下：

fp：文件指针。

offset：以字节为单位的位移量，表示以起始点为基点向前移动的字节数，如位移量为负，则表示向后移，这里的向前移是指从文件开头向文件末尾移动的方向。

origin：起始点，用于指定是以某个位置为基准，起始点可用标识符来表示，也可用数字来表示。

真 题 精 选

1. 下列有关 if 语句的形式，错误的是(　　)。

A.
```
if(表达式)
    语句；
```

B.
```
if(表达式 1)
    语句 1；
  else(表达式 2)
    语句 2；
```

C.
```
if(表达式)
    语句 1；
  else
    语句 2；
```

D.
```
if(表达式 1)
    语句 1；
  if(表达式 2)
    语句 2；
  else
    语句 3；
  else
    语句 4；
```

【答案】 B

【解析】 if 语句有两种形式，形式 1 为：

```
if(表达式)
  语句；
```

形式 2 为：

```
if(表达式 1)
  语句 1;
else
  语句 2;
```

此外，if 语句还可以嵌套。题目中，选项 A 为形式 1，正确；选项 B，else 后不能再跟表达式，所以错误；选项 C 为形式 2，正确；选项 D 为 if 语句嵌套，else 与最近的 if 语句配对，正确。

2. 对于编译系统来说，下列在 C 语言程序中属于无意义信息的是(　　)。

A. 注释　　B. 头函数　　C. ；　　D. {

【答案】A

【解析】和其他程序一样，C 语言程序中也可以使用注释，注释的格式一般为：/ * 注释内容 * /。注释可以加在程序的任意位置，并且可以占用一行以上的位置。编译系统对其既不编译也不执行，属于无意义信息。

3. 在 C 语言中做了如下定义：char a [3][4]，下列表述正确的是(　　)。

A. 定义了一个数列　　B. 可以在该数组中存入字符串“Abcd”

C. 定义了一个二维数组　　D. a[3][4]＝'E';

【答案】C

【解析】题干中的语句定义了一个二维数组，可以在其中存入字符或者字符串。系统在字符串结尾处自动加上“\0”作为结束标志，所以存入字符串“Abcd”会造成溢出。D 选项数组下标越界。

4. C 语言中，已知 int j，k＝6；，则计算表达式 j＝k＋＋后，正确的是(　　)。

A. j＝6，k＝6　　B. j＝6，k＝7　　C. j＝7，k＝7　　D. j＝7，k＝6

【答案】B

【解析】在使用 C 语言的自增运算符＋＋时，若将自增运算符放在变量之前，表示先使变量的值加 1 后使用；若将自增运算符放在变量之后，表示先使用变量的值然后加 1。因此，j＝k＋＋中，系统将 k＝6 的值赋给 j，j＝6，之后 k＋1＝7，因此选择 B 选项。

5.
```
#include <stdio.h>
#include <stdlib.h>
struct List
{
    int data;
    struct List * next;
};
typedef struct List node;
typedef node * link;

void main()
{
    link ptr,head,tail;
    int num,i;
    tail=(link)malloc(sizeof(node));
      tail→next=    (1)    ;
```

```
        ptr=tail;
        printf("\nplease input data:\n");
        for(i=0;i<=4;i++)
        {
            scanf("%d",    (2)    ;
            ptr→data=num;
            head=(link)malloc(sizeof(node));
            head→next=    (3)    ;
            ptr=head;
        }
    ptr=    (4)    ;
    while(ptr! =NULL)
    {
        printf("the value is %d.\n",    (5)    );
        ptr=ptr→next;
}
}
```

【答案】(1)NULL　(2)&num　(3)ptr　(4)head→next　(5)ptr→data

【解析】本程序实现的功能是:利用尾插法创建一个包含5个数据结点的单链表,然后将这5个结点的数据输出。

程序一开始定义了一个结构体,即:

```
struct List
{
    int data;
    struct List * next;
};
```

由此可以看出,单链表的每个结点包含了一个数据域和一个指向后继结点的指针域。

在main函数中,首先创建一个结点,令尾指针指向该结点即“tail=(link)malloc(sizeof(node));”,由于后面第二句ptr=tail令工作指针指向尾结点,可知第一个空填NULL,即将尾结点的指针域置为NULL,“tail→next= NULL”;。

for循环的作用是依次创建5个结点,并为这5个结点的数据域赋值。由“ptr→data=num;”可知其前一句是用num变量保存输入的整数,则第二个空填&num,即“scanf("%d",&num)”;。

由于头指针是指向第一个结点的,因此第三个空填入ptr,即“head→next=ptr;”,令头指针一直指向第一个结点的地址。

最后用while循环输出链表中每个结点的数据值,因此在进入while循环之前令工作指针ptr指向第一个结点,即“ptr=head→next;”,在输出结点数据值时,利用ptr→data取当前指针指向的数据域即可,即“printf("the value is %d. \n",ptr→data);”。

强 化 训 练

一、单项选择题

1. 设有条件表达式:(EXP)? i++;j--,则以下表达式中与(EXP)完全等价的是(　　)。

A. (EXP==0)　　B. (EXP! =0)

C. (EXP==1)　　D. (EXP! =1)

2. 若有定义语句:int a[3][6];,按元素在内存中的存放顺序,a 数组的第十个元素是(　　)。

A. a[0][4]　　B. a[1][3]

C. a[0][3]　　D. a[1][4]

3. 若变量均已正确定义并赋值,以下合法的 C 语言赋值语句是(　　)。

A. x=y==5;　　B. x=n%2.5;

C. x+n=1;　　D. x=5=4+1;

4. 以下选项中不属于字符常量的是(　　)。

A. 'C'　　B. "C"

C. '\xCC0'　　D. '\072'

5. 以下 C 语言用户标识符中,不合法的是(　　)。

A. _1　　B. AaBc

C. a_b　　D. a-b

二、判断题

1. C 语言中,一个自定义函数中可以根据不同的情况设置多条 return 语句。(　　)
2. C 语言中的注释会被编译,但不会被执行。(　　)
3. 在一个 C 语言源程序文件中定义的全局变量,其作用域为其所在函数的全部范围。(　　)
4. C 语言源程序名的后缀是.c。(　　)
5. 在 C 语言中,用户可以在程序中指定任意一个函数作为主函数,程序将从此开始执行。(　　)

三、程序设计题

输入 8 个正整数,然后输出其中的"对称数",并统计"对称数"的个数。所谓"对称数",是指将该数的各位数反过来以后与原来的数一样,如 101、151、8118、1991 等。

参 考 答 案

一、单项选择题

1. B　[解析]题干中的表达式用于判断 EXP 是否为真,在 C 语言中用 0 值代表假,用非零值代表真,因此在判断 EXP 是否为真时应当使用 EXP! =0。

2. B　[解析]本题考查二维数组的存储方式。二维数组中的元素是按行顺序存放的,第一个元素为 a[0][0],第二个元素为 a[0][1],因此 a 数组的第十个元素是 a[1][3]。

3. A　[解析]运算符%只能对 int 型数据进行求余,B 中对浮点数使用%求余,因此不正确。C 中 x+n 表示运算操作,不能对操作的结果进行赋值,因此不正确。D 中 5=4+1 表示对常量 5 赋值,因此也不正确。

4. B [解析]在C语言中,单引号用来表示字符常量,双引号用来表示字符串常量,因此B选项不是字符常量,而是一个字符串常量。

5. D [解析]在C语言中,用户标识符由下划线、字母或数字组成,并且必须由字母或下划线开头,因此D选项中含有减号(—)字符不合法。

二、判断题

1. √ [解析]在C语言中,一个函数可以有一条以上的return语句,当其中的一条return语句得以执行后立即跳出该函数,不再执行其他return语句。

2. × [解析]注释在程序中仅仅起到对程序的注解和说明的作用,在程序编译的词法分析阶段,注释将被从程序中删除。

3. × [解析]全局变量是指作用域在程序级和文件级的变量,在C语言中,其作用域由其定义位置来决定。

4. √ [解析]C语言中,源程序名的后缀是".c"。

5. × [解析]C语言中,以main函数作为入口函数,程序从此开始。

三、程序设计题

[专家点拨]

```
#include <stdio.h>
int main()
{
    int num[8];//num 数组用于保存输入的 8 个正整数
    int i;
    int count=0;
    for(i=0;i<8;i++)//输入 8 个正整数
        scanf("%d",num[i]);
    for(i=0;i<8;i++)
    {
        int isRegular = isRegularNum(num[i]); //判断第 i 个数是否是对称数
        if(isRegular == 1) //如果是对称数,则输出该数
        {
            printf("%d\n",num[i]);
            count++;
        }
    }
    printf("对称数的个数为:%d\n",count);//输出对称数的个数
    return 0;
}
int isRegularNum(int num)
{
    /*流程:把一个数 num 从个位开始取到最高位,以此顺序构造一个 N 位数,
    如果这个数和 num 一样,则它是对称数*/
    int regular=0; //存这个反序数
    int temp=num;
```

```
    while(temp! =0)
    {
        regular = regular * 10+temp%10;
        temp = temp/10;
    }
    if(regular == num)
        return 1;
return 0;
}
```