第一篇
基础知识

第1章
数据结构概述

近年来，随着计算机技术的快速发展，数据规模呈现几何级增长，数据类型也变得多样化，软件开发需要处理的数据日趋复杂，数据结构在人工智能、大数据技术飞速发展的今天显得尤为重要。要想编写出好的程序，不仅需要选择好的数据结构，还要有高效的算法。数据结构与算法往往是紧密联系在一起的。

本章重点和难点：

- 数据结构的相关概念。
- 数据的逻辑结构与存储结构。
- 抽象数据类型描述。
- 算法的时间复杂度和空间复杂度。

1.1 为什么要学习数据结构

1. 数据结构的前世今生

数据结构作为一门独立的课程是从 1968 年开始在美国设立的。1968 年，算法和程序设计技术的先驱，美国的唐·欧·克努特（Donald Ervin Knuth，中文名高德纳）教授开创了数据结构的最初体系，他所著的《计算机程序设计艺术》第一卷《基本算法》是第一本较系统地阐述数据的逻辑结构和存储结构及其操作的著作。从 20 世纪 60 年代末到 20 世纪 70 年代初，随着大型程序的出现，软件也相对独立，结构化程序设计成为程序设计方法学的主要内容，数据结构显得越来越重要。

从 20 世纪 70 年代中期到 20 世纪 80 年代，各种版本的数据结构著作相继出现。目前，数据结构的发展并未就此止步，随着大数据和人工智能时代的到来，数据结构开始在新的应用领域发挥重要作用。面对爆炸性增长的数据和计算机技术的发展，人工智能、大数据、机器学习等各应用领域中需要处理的大量多维数据就需要对数据进行组织和处理，数据结构的重要性不言而喻。

高德纳（Donald Ervin Knuth）写出了计算机科学理论与技术的经典巨著《计算机程序设计艺术》（*The Art of Computer Programming*）（共五卷），该著作被《美国科学家》杂志列为 20 世纪最重要的 12 本物理科学类专著之一，与爱因斯坦《相对论》、狄拉克《量子力学》、理查•费曼《量子电动力学》等经典比肩。高德纳因而在他 36 岁时就荣获 1974 年度的图灵奖。《计算机程序设计的艺术》推出之后，真正能读完读懂的人数并不多，据说比尔·盖茨花费了几个月才读完第一卷，然后说：“如果你觉得自己是一名优秀的程序员，那就去读《计算机程序设计

艺术》吧。对我来说，读完这本书不仅花了好几个月，而且还要求我有极高的自律性。如果你能读完这本书，不妨给我发个简历。”

2. 数据结构的作用与地位

数据结构是介于数学、计算机硬件和计算机软件三者之间的一门核心课程。数据结构已经不仅是计算机相关专业的核心课程，还是其他非计算机专业的主要选修课程之一，其重要性不言而喻。数据结构与计算机软件的研究有着更密切的关系，开发计算机系统软件和应用软件都会用到各种类型的数据结构。例如，算术表达式求值问题、迷宫求解、机器学习中的决策树分类等分别利用了数据结构中的栈、树进行解决，因此，要想更好地运用计算机来解决实际问题，使编写出的程序更高效、具有通用性，仅掌握计算机程序设计语言是难以应付众多复杂问题的，还必须学习和掌握好数据结构方面的有关知识。数据结构也是学习操作系统、软件工程、人工智能、算法设计与分析、机器学习、大数据等众多后继课程的重要基础。

1.2　基本概念和术语

在学习数据结构的过程中，有一些基本概念和专业术语会经常出现，下面先来了解一下这些基本概念和术语。

1. 数据

数据（data）是描述客观事物的符号，能输入到计算机中并能被计算机程序处理的符号集合。它是计算机程序加工的“原料”。例如，一个文字处理程序（如 Microsoft Word）的处理对象就是字符串，一个数值计算程序的处理对象就是整型和浮点型数据。因此，数据的含义非常广泛，如整型、浮点型等数值类型及字符、声音、图像、视频等非数值数据都属于数据范畴。

2. 数据元素

数据元素（data element）是数据的基本单位，在计算机程序中通常作为一个整体考虑和处理。一个数据元素可由若干个数据项（data item）组成，数据项是数据不可分割的最小单位。例如，一个学校的教职工基本情况表包括工号、姓名、性别、籍贯、所在院系、出生年月及职称等数据项。教职工基本情况如表 1-1 所示。表中的一行就是一个数据元素，也称为一条记录。

表 1-1　教职工基本情况

工　号	姓　名	性　别	籍　贯	所在院系	出生年月	职　称
2006002	孙冬平	男	河南	计算机学院	1970.10	教　授
2019056	朱　琳	女	北京	文学院	1985.08	讲　师
2015028	刘晓光	男	陕西	软件学院	1981.11	副教授

3. 数据对象

数据对象（data object）是性质相同的数据元素的集合，是数据的一个子集。例如，对于正整数来说，数据对象是集合 N={1，2，3，…}；对于字母字符数据来说，数据对象是集合 C={'A'，'B'，'C'，…}。

4. 数据结构

数据结构（data structure）即数据的组织形式，它是数据元素之间存在的一种或多种特定关

系的数据元素集合。在现实世界中，任何事物都是有内在联系的，而不是孤立存在的，同样在计算机中，数据元素不是孤立的、杂乱无序的，而是具有内在联系的数据集合。例如，表 1-1 的教职工基本情况表是一种表结构，学校的组织机构是一种层次结构，城市之间的交通路线属于图结构，如图 1-1 和图 1-2 所示。

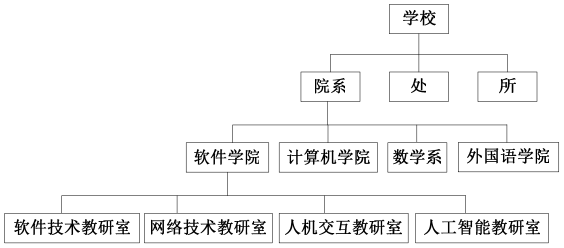


图 1-1　学校组织机构图

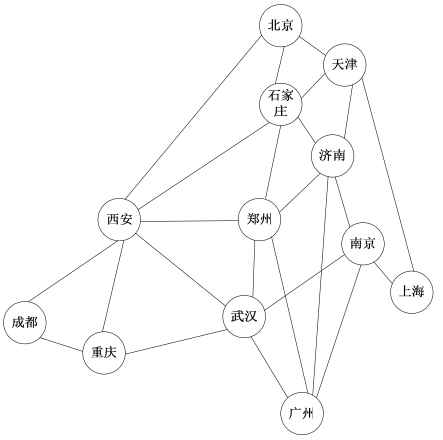


图 1-2　城市之间交通路线图

5. 数据类型

数据类型（data type）用来刻画一组性质相同的数据及其上的操作。数据类型是按照值的不同进行划分的。在高级语言中，每个变量、常量和表达式都有各自的取值范围，该类型就说明了变量或表达式的取值范围和所能进行的操作。例如，C 语言中的字符类型规定了所占空间是 8 位，也就决定了它的取值范围，同时也定义了在其范围内可以进行赋值运算、比较运算等。

在 C 语言中，按照取值的不同，数据类型还可以分为原子类型和结构类型两类。原子类型是不可以再分解的基本类型，包括整型、实型、字符型等。结构类型是由若干个类型组合而成，是可以再分解的。例如，整型数组是由若干整型数据组成的，结构类型的值也是由若干个类型范围的数据构成，它们的类型都是相同的。

随着计算机技术的飞速发展，计算机从最初仅能够处理数值信息，发展到现在能处理的对象包括数值、字符、文字、声音、图像及视频等信息。任何信息只要经过数字化处理，能够让计算机识别，都能够进行处理。当然，这需要对要处理的信息进行抽象描述，让计算机理解。

1.3　数据的逻辑结构与存储结构

数据结构的主要任务就是通过分析数据对象的结构特征，包括逻辑结构及数据对象之间的关系，并把逻辑结构表示成计算机可实现的物理结构，以便设计、实现算法。

1.3.1　逻辑结构

数据的逻辑结构（logical structure）是指在数据对象中数据元素之间的相互关系。数据元素之间存在不同的逻辑关系构成了以下 4 种结构类型。

（1）集合。结构中的数据元素除了同属于一个集合外，数据元素之间没有其他关系。这就

像数学中的自然数集合，集合中的所有元素都属于该集合，除此之外，没有其他特性。例如，数学中的正整数集合{5，67，978，20，123，18}，集合中的数除了属于正整数外，元素之间没有其他关系。数据结构中的集合关系就类似于数学中的集合。集合表示如图 1-3 所示。

（2）线性结构。结构中的数据元素之间是一对一的关系。线性结构如图 1-4 所示。数据元素之间有一种先后的次序关系，a、b、c……是一个线性表，其中，a 是 b 的前驱，b 是 a 的后继。

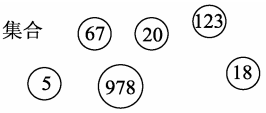


图 1-3　集合结构

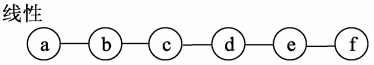


图 1-4　线性结构

（3）树状结构。结构中的数据元素之间存在一种一对多的层次关系，树状结构如图 1-5 所示。这就像学校的组织结构图，学校下面是教学的院系、行政机构及一些研究所。

（4）图结构。结构中的数据元素是多对多的关系，图 1-6 就是一个图结构。城市之间的交通路线图就是多对多的关系，a、b、c、d、e、f、g 是 7 个城市，城市 a 和城市 b、e、f 都存在一条直达路线，而城市 b 也和 a、c、f 存在一条直达路线。

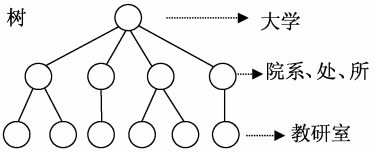


图 1-5　树状结构

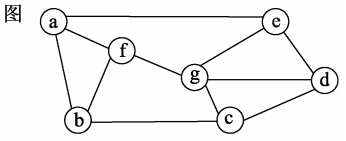


图 1-6　图结构

1.3.2　存储结构

存储结构（storage structure）也称为物理结构（physical structure），指的是数据的逻辑结构在计算机中的存储形式。数据的存储结构应能正确反映数据元素之间的逻辑关系。

数据元素的存储结构形式通常有顺序存储结构和链式存储结构两种。顺序存储是把数据元素存放在一组地址连续的存储单元里，其数据元素间的逻辑关系和物理关系是一致的。采用顺序存储的字符串“abcdef”的存储结构如图 1-7 所示。链式存储是把数据元素存放在任意的存储单元里，这组存储单元可以是连续的，也可以是不连续的，数据元素的存储关系并不能反映其逻辑关系，因此需要借助指针来表示数据元素之间的逻辑关系。字符串“abcdef”的链式存储结构如图 1-8 所示。

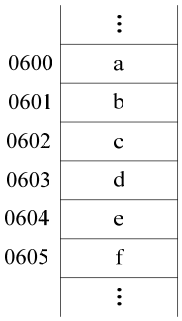


图 1-7　顺序存储结构

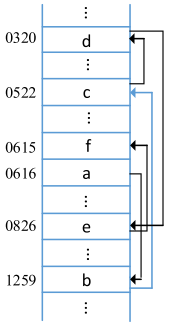


图 1-8　链式存储结构

数据的逻辑结构和物理结构是密切相关的，在学习数据结构的过程中，将会发现，任何一个算法的设计取决于选定的数据逻辑结构，而算法的实现则依赖于所采用的存储结构。

如何描述存储结构呢？通常是借助 C/C++/Java/Python 等高级程序设计语言中提供的数据类型进行描述。例如，对于数据结构中的顺序表可以用 C 语言中的一维数组表示；对于链表，可用 C 语言中的结构体描述，其中用指针来表示元素之间的逻辑关系。

1.4 抽象数据类型及其描述

在数据结构中，把一组包含数据类型、数据关系及在该数据上的一组基本操作统称为抽象数据类型。

1.4.1 什么是抽象数据类型

抽象数据类型（Abstract Data Type，ADT）是描述具有某种逻辑关系的数学模型，并在该数学模型上进行的一组操作。这个抽象数据类型有点类似于 C++和 Java 中的类，例如，Java 中的 Integer 类是基本类型 int 所对应的封装类，它包含 MAX_VALUE（整数最大值）、MIN_VALUE（整数最小值）等属性、toString(int i)、parseInt(String s)等方法。它们的区别在于，抽象数据类型描述的是一组逻辑上的特性，与在计算机内部如何表示无关；Java 中的 Integer 类是依赖具体实现的，是抽象数据类型的具体化表现形式。

抽象数据类型不仅包括在计算机中已经定义了的数据类型，如整型、浮点型等，还包括用户自己定义的数据类型，如结构体类型、类等。

一个抽象数据类型定义了一个数据对象、数据对象中数据元素之间的关系及对数据元素的操作。抽象数据类型通常是指用来解决应用问题的数据模型，包括数据的定义和操作。

抽象数据类型体现了程序设计中的问题分解、抽象和信息隐藏特性。抽象数据类型把实际生活中的问题分解为多个规模小且容易处理的问题，然后建立起一个计算机能处理的数据模型，并把每个功能模块的实现细节作为一个独立的单元，从而使具体实现过程隐藏起来。这就类似人们日常生活中盖房子，把盖房子分成若干个小任务：地皮审批、图纸设计、施工、装修等，工程管理人员负责地皮的审批，地皮审批下来之后，工程技术人员根据用户需求设计图纸，建筑工人根据设计好的图纸进行施工（包括打地基、砌墙、安装门窗等），盖好房子后请装修工人装修。

盖房子的过程与抽象数据类型中的问题分解类似，工程管理人员不需要了解图纸如何设计，工程技术人员不需要了解打地基和砌墙的具体过程，装修工人不需要知道怎么画图纸和怎样盖房子，这就是抽象数据类型中的信息隐藏。

1.4.2 抽象数据类型的描述

对于初学者来说，抽象数据类型不太容易理解，用一大堆公式会让不少读者迷茫，因此，本书采用通俗的语言去讲解抽象数据类型。本书把抽象数据类型分为两部分来描述，即数据对象集合和基本操作集合。其中，数据对象集合包括数据对象的定义及数据对象中元素之间关系的描述，基本操作集合是对数据对象的运算的描述。数据对象和数据关系的定义可采用数学符号和自然语言描述，基本操作的定义格式如下。

```
基本操作名（参数表）：初始条件和操作结果描述.
```

例如，集合 Set 的抽象数据类型描述如下。

1. 数据对象集合

集合 Set 的数据对象集合为{a_1，a_2，…，a_n}，每个元素的类型均为 DataType。

2. 基本操作集合

（1）InitSet (&*S*)：初始化操作，建立一个空的集合 *S*。

（2）SetEmpty(*S*)：若集合 *S* 为空，返回 1，否则返回 0。

（3）GetSetElem (*S*,*i*,&*e*)：返回集合 *S* 的第 *i* 个位置元素值给 *e*。

（4）LocateElem (*S*,*e*)：在集合 *S* 中查找与给定值 *e* 相等的元素，如果查找成功返回该元素在表中的序号，否则返回 0。

（5）InsertSet (&*S*,*e*)：在集合 *S* 中插入一个新元素 *e*。

（6）DelSet (&*S*,*i*,&*e*)：删除集合 *S* 中的第 *i* 个位置元素，并用 *e* 返回其值。

（7）SetLength(*S*)：返回集合 *S* 中的元素个数。

（8）ClearSet(&*L*)：将集合 *S* 清空。

（9）UnionSet(&*S*,*T*)：合并集合 *S* 和 *T*，即将 *T* 中的元素插入到 *S* 中，相同的元素只保留一个。

（10）DiffSet(&*S*,*T*)：求两个集合的差集，即 *S*-*T*，即删除 *S* 中与 *T* 中相同的元素。

（11）DispSet(*S*)：输出集合 *S* 中的元素。

基本操作实现如下。

```
typedef struct myset/*集合的类型定义*/
{
       DataType list[MAXSIZE];
       int length;
}Set;
void InitSet(Set *S)
/*集合 S 的初始化*/
{
       S->length=0;
}
int SetEmpty(Set S)
/*判断集合 S 是否为空，若为空，则返回 1；否则，返回 0*/
{
       if(S.length<=0)
              return 1;
       else
              return 0;
}
int SetLength(Set S)
/*返回集合 S 中元素个数*/
{
       return S.length;
}
void ClearSet(Set *S)
/*清空集合 S*/
{
       S->length=0;
}
int InsertSet(Set *S, DataType e)
/*在集合 S 中插入一个元素 e*/
{
       if(S->length>=MAXSIZE-1)
```

```
                return -1;
        else
        {
                S->list[S->length]=e;
                S->length++;
                return 1;
        }
}
int DelSet(Set *S, int pos)
/*删除集合 S 中的第 pos 个元素*/
{
        int i;
        if(S->length<=0)
                return -1;
        else
        {
                for(i=pos-1;i<S->length-1;i++)
                  S->list[i]=S->list[i+1];
                S->length--;
                return 1;
        }
}

int GetSetElem(Set S,int i,DataType *e)
/*获取集合 S 中第 i 个元素赋给 e*/
{
        if(S.length<=0)
                return -1;
        else if(i<1&&i>S.length)
                return -1;
        else
        {
                *e=S.list[i-1];
                return 1;
        }
}
int LocateElem(Set S, DataType e)
/*查找集合 S 中元素值为 e 的元素，返回其序号*/
{
        int i;
        for(i=1;i<=S.length;i++)
        {
                if(S.list[i-1]==e)
                  return i;
        }
        return 0;
}
int UnionSet(Set *S, Set T)
/*合并集合 S 和 T*/
{
        DataType e;
        int i;
        if(S->length+T.length>=MAXSIZE-1)
           return -1;
        else
        {
           for(i=1;i<=T.length;i++)
           {
                  GetSetElem(T,i,&e);
                  if(!LocateElem(*S,e))
```

```
                    InsertSet(S,e);
                }
        }
}
int DiffSet(Set *S, Set T)
/*求集合 S 和 T 的差集*/
{
        DataType e;
        int i,pos;
        if(S->length<=0)
                return -1;
        else
        {
                for(i=1;i<=T.length;i++)
                {
                  GetSetElem(T,i,&e);
                    if(pos=LocateElem(*S,e))
                      DelSet(S,pos);
                 }
                 return 1;
        }
}
void DispSet(Set S)
/*输出集合 S 中的元素*/
{
        int i;
        for(i=1;i<=S.length;i++)
                printf("%4c",S.list[i-1]);
        printf("\n");
}
```

1.5　算法

在定义好了数据类型之后，就要在此基础上设计实现算法，即程序。

1.5.1　数据结构与算法的关系

算法与数据结构关系密切，两者既有联系又有区别。数据结构与算法的联系可用一个公式描述：

程序=算法+数据结构

数据结构是算法实现的基础，算法依赖于某种数据结构才能实现。算法的操作对象是数据结构。算法的设计和选择要同时结合数据结构，只有确定了数据的存储方式和描述方式，即数据结构确定了之后，算法才能确定。例如，在数组和链表中查找元素值的算法实现是不同的。算法设计的实质就是对实际问题要处理的数据选择一种恰当的存储结构，并在选定的存储结构上设计一个好的算法。

数据结构是算法设计的基础。比如你要装修房子，装修房子的设计就相当于算法设计，而如何装修房子是要看房子的结构设计，不同的房间结构，其装修设计是不同的，只有确定了房间结构，才能进行房间的装修设计。房间的结构就像数据结构。算法设计必须要考虑到数据结构的构造，算法设计是不可能独立于数据结构存在的。数据结构的设计和选择需要为算法服务，根据数据结构及特点，才能设计出好的算法。

1.5.2 什么是算法

算法（algorithm）是解决特定问题求解步骤的描述，在计算机中表现为有限的操作序列。操作序列包括一组操作，每一个操作都完成特定的功能。例如，求 n 个数中最大者的问题，其算法描述如下。

（1）定义一个变量 max 和一个数组 a[]，分别用来存放最大数和数组的元素，并假定第一个数最大，赋给 max。

```
max=a[0];
```

（2）依次把数组 a 中其余的 n-1 个数与 max 进行比较，遇到较大的数时，将其赋给 max。

```
for(i=1;i<n;i++)                          /*for 循环处理*/
     if(max<a[i])                         /*判断是否满足 max 小于 a[i]的条件*/
        max=a[i];                         /*如果满足条件,将 a[i]赋值给 max*/
```

最后，max 中的数就是 n 个数中的最大者。

1.5.3 算法的五大特性

算法具有以下 5 个特性。

（1）有穷性（finiteness）。有穷性指的是算法在执行有限的步骤之后，自动结束而不会出现无限循环，并且每一个步骤在可接受的时间内完成。

（2）确定性（definiteness）。算法的每一步骤都具有确定的含义，不会出现二义性。算法在一定条件下只有一条执行路径，也就是相同的输入只能有一个唯一的输出结果。

（3）可行性（feasibility）。算法的每个操作都能够通过执行有限次基本运算完成。

（4）输入（input）。算法具有零个或多个输入。

（5）输出（output）。算法至少有一个或多个输出。输出的形式可以是打印输出也可以是返回一个或多个值。

1.5.4 算法的描述方式

算法的描述方式有多种，如自然语言、伪代码（或称为类语言）、程序流程图及程序设计语言（如 C 语言）等。其中，自然语言描述可以是汉语或英语等文字描述；伪代码形式类似于程序设计语言形式，但是不能直接运行；程序流程图的优点是直观，但是不易直接转换为可运行的程序；程序设计语言描述算法就是直接利用像 C、C++、Java 等语言表述，优点是可以直接在计算机上运行。为了方便读者学习和上机操作，本书所有算法均采用 C 语言描述，所有程序均可直接上机运行。

1.6 算法分析

一个好的算法往往可以使程序尽可能快地运行，衡量一个算法的好坏往往将算法效率和存储空间作为重要依据。算法的效率需要通过算法思想编写的程序在计算机上的运行时间来衡量，存储空间需要通过算法在执行过程中所占用的最大存储空间来衡量。

1.6.1　算法设计的 4 个目标

一个好的算法应该具备以下目标。

1. 算法的正确性

算法的正确性是指算法至少应该包括对于输入、输出和处理无歧义性的描述，能正确反映问题的需求，且能够得到问题的正确答案。

通常算法的正确性应包括以下 4 个层次。

（1）算法对应的程序没有语法错误。

（2）对于几组输入数据能得到满足规格要求的结果。

（3）对于精心选择的典型的、苛刻的、带有刁难性的几组输入数据能得到满足规格要求的结果。

（4）对于一切合法的输入都能得到满足要求的结果。

对于这 4 层算法正确性的含义，达到第 4 层意义上的正确是极为困难的，所有不同输入数据的数量大得惊人，逐一验证的方法是不现实的。一般情况下，把层次 3 作为衡量一个程序是否正确的标准。

2. 可读性

算法主要是为了人们方便阅读和交流，其次才是计算机执行。可读性好有助于人们对算法的理解，晦涩难懂的程序往往隐含错误不易被发现，难以调试和修改。

3. 健壮性

当输入数据不合法时，算法也应该能做出反应或进行处理，而不会产生异常或莫名其妙的输出结果。例如，求一元二次方程根 $ax^2+bx+c=0$ 的算法，需要考虑多种情况。先判断 b^2-4ac 的正负，如果为正数，则该方程有两个不同的实根；如果为负数，表明该方程无实根；如果为零，表明该方程只有一个实根；如果 $a=0$，则该方程又变成了一元一次方程，此时若 $b=0$，还要处理除数为零的情况。如果输入的 a、b、c 不是数值型，还要提示用户输入错误。

4. 高效率和低存储量

效率指的是算法的执行时间。对于同一个问题如果有多个算法能够解决，执行时间短的算法效率高，执行时间长的效率低。存储量需求指算法在执行过程中需要的最大存储空间。效率与低存储量需求都与问题的规模有关，求 100 个人的平均分与求 1000 个人的平均分所花的执行时间和运行空间显然有一定差别。设计算法时应尽量选择高效率和低存储量需求的算法。

1.6.2　算法效率评价

算法执行时间需通过依据该算法编制的程序在计算机上的运行时所耗费的时间来度量，而度量一个算法在计算机上的执行时间通常有如下两种方法。

1. 事后统计方法

目前计算机内部大都有计时功能，有的甚至可精确到毫秒级，不同算法的程序可通过一组或若干组相同的测试程序和数据以分辨算法的优劣。但是这种方法有两个缺陷：一是必须依据算法事先编制好程序，这通常需要花费大量的时间与精力；二是时间的长短依赖计算机硬件和软件等环境因素，有时会掩盖算法本身的优劣。因此，人们常常采用事前分析估算的方法评价算法的好坏。

2. 事前分析估算方法

这主要在计算机程序编制前，对算法依据数学中的统计方法进行估算。这主要是因为算法的程序在计算机上的运行时间取决于以下因素。

（1）算法采用的策略、方法。

（2）编译产生的代码质量。

（3）问题的规模。

（4）书写的程序语言，对于同一个算法，语言级别越高，执行效率越低。

（5）机器执行指令的速度。

在以上 5 个因素中，算法采用不同的策略，或不同的编译系统，或不同的语言实现，或在不同的机器上运行时，效率都不相同。抛开以上因素，算法效率则可以通过问题的规模来衡量。

一个算法由控制结构（顺序、分支和循环结构）和基本语句（赋值语句、声明语句和输入输出语句）构成，则算法的运行时间取决于两者执行时间的总和，所有语句的执行次数可以作为语句的执行时间的度量。语句的重复执行次数称为语句频度。

例如，斐波那契数列的算法和语句的频度如下。

```
                                                    每一条语句的频度
f0=0;                              /*赋值*/         1
f1=1;                              /*赋值*/         1
printf("%d,%d",f0,f1);             /*输出提示信息*/  1
for(i=2;i<=n;i++)                  /*for 循环处理*/  n
{
      fn=f0+f1;                    /*fn=f0+f1*/      n-1
      printf(",%d",fn);            /*输出提示信息*/  n-1
      f0=f1;                       /*赋值 f0=f1*/    n-1
      f1=fn;                                         n-1
}
```

每一条语句的右端是对应语句的**频度**（frequency count），即语句的执行次数。上面算法总的执行次数为 $f(n)=1+1+1+n+4(n-1)=5n-1$。

1.6.3 算法时间复杂度

算法分析的目的是看设计的算法是否具有可行性，并尽可能挑选运行效率高效的算法。

1. 什么是算法时间复杂度

在进行算法分析时，语句总的执行次数 $f(n)$是关于问题规模 n 的函数，进而分析 $f(n)$随 n 的变化情况并确定 $f(n)$的数量级。算法的时间复杂度，也就是算法的时间量度，记作 $T(n)=O(f(n))$。

它表示随问题规模 n 的增大，算法的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，简称为时间复杂度。其中，$f(n)$是问题规模 n 的某个函数。

一般情况下，随着 n 的增大，$T(n)$的增长较慢的算法为最优的算法。例如，在下列三段程序段中，给出基本操作 $x=x+1$ 的时间复杂度分析。

（1）x=x+1;

（2）for（i=1;i<=n;i++）

　　x=x+1;

（3）for（i=1;i<=n;i++）

```
for（j=1;j<=n;j++）
    x=x+1;
```

程序段（1）的时间复杂度为 $O(1)$，称为常量阶；程序段（2）的时间复杂度为 $O(n)$，称为线性阶；程序段（3）的时间复杂度为 $O(n^2)$，称为平方阶。此外，算法的时间复杂度还有对数阶 $O(\log_2 n)$、指数阶 $O(2^n)$等。

上面的斐波那契数列的时间复杂度 $T(n)=O(n)$。

常用的时间复杂度所耗费的时间从小到大依次是 $O(1)<O(\log_2 n)<O(n)<O(n^2)<O(n^3)<O(2^n)<O(n!)$。

算法的时间复杂度是衡量一个算法好坏的重要指标。一般情况下，具有指数级的时间复杂度算法只有当 n 足够小时才是可使用的算法。具有常量阶、线性阶、对数阶、平方阶和立方阶的时间复杂度算法是常用的算法。一些常见函数的增长率如图 1-9 所示。

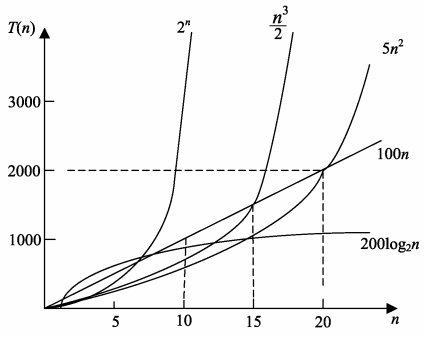


图 1-9　常见函数的增长率

一般情况下，算法的时间复杂度只需要考虑关于问题规模 n 的增长率或阶数。例如以下程序段。

```
for(i=2;i<=n;i++)                          /*for 外层循环*/
      for(j=2;j<=i-1;j++)                  /*for 内层循环*/
      {
           k++;                            /*k 自增*/
           a[i][j]=x;                      /*x 赋值给数组 a[i][j]*/
      }
```

语句 k++的执行次数关于 n 的增长率为 n^2，它是语句频度$(n-1)(n-2)/2$ 中增长最快的项。

在某些情况下，算法的基本操作的重复执行次数不仅依赖于输入数据集的规模，还依赖于数据集的初始状态。例如，在以下的冒泡排序算法中，其基本操作执行次数还取决于数据元素的初始排列状态。

```
void BubbleSort(int a[],int n)      /*冒泡排序算法函数]*/
{
     int i,j,t;                     /*定义三个整型变量*/
     change=TRUE;                   /*变量 change 赋值为 TRUE*/
     for(i=1;i<=n-1&&change;i++)    /*for 外层循环处理*/
     {
          change=FALSE;             /*变量 change 赋值为 FALSE*/
          for(j=1;j<=n-i;j++)       /*for 内层循环处理*/
               if(a[j]>a[j+1])      /*判断,冒泡排序算法实现*/
               {                    /*比较两个元素,如果它们的顺序错误就将它们交换过来*/
                    t=a[j];
                    a[j]=a[j+1];
                    a[j+1]=t;
                    change=TRUE;    /*变量 change 赋值为 TRUE*/
               }
      }
}
```

交换相邻两个整数为该算法中的基本操作。当数组 a 中的初始序列为从小到大有序排列时，

基本操作的执行次数为 0；当数组中初始序列从大到小排列时，基本操作的执行次数为 $n(n-1)/2$。对这类算法的分析，一种方法是计算所有情况的平均值，这种时间复杂的计算方法称为平均时间复杂度；另外一种方法是计算最坏情况下的时间复杂度，这种方法称为最坏时间复杂度。若数组 a 中初始输入数据可能出现 $n!$ 种的排列情况的概率相等，则冒泡排序的平均时间复杂度为 $T(n)=O(n^2)$。

然而，在很多情况下，各种输入数据集出现的概率难以确定，算法的平均复杂度也就难以确定。因此，另一种更可行也更为常用的办法是讨论算法在最坏情况下的时间复杂度，即分析最坏情况以估算算法执行时间的上界。例如，上面冒泡排序的最坏时间复杂度为数组 a 中初始序列为从大到小有序，则冒泡排序算法在最坏情况下的时间复杂度为 $T(n)=O(n^2)$。一般情况下，本书以后讨论的时间复杂度，在没有特殊说明的情况下，都指的是最坏情况下的时间复杂度。

2. 算法时间复杂度分析举例

一般情况下，算法的时间复杂度只需要考虑算法中的基本操作，即算法中最深层循环体内的操作。

【例 1.1】分析以下程序段的时间复杂度。

```
for(i=2;i<=n;i++)
    for(j=2;j<=i-1;j++)
    {
       x++;              //基本操作
       a[i][j]=x;        //基本操作
    }
```

该程序段中的基本操作是第二层 for 循环中的语句，即 x++和 $a[i][j]=x$，其语句频度为 $(n-1)(n-2)/2$。因此，其时间复杂度为 $O(n^2)$。

【例 1.2】分析以下算法的时间复杂度。

```
void Fun( )
{
    int i=1;
    while(i<=n)
    {
       i=i*2;            //基本操作
    }
}
```

该函数 Fun()的基本操作是 $i=i*2$，设循环次数为 $f(n)$，则 $2^{f(n)}\leqslant n$，则有 $f(n)\leqslant\log_2 n$。因此，时间复杂度为 $O(\log_2 n)$。

【例 1.3】分析以下算法的时间复杂度。

```
void Func( )
{
    i=s=0;
    while(s<n)
    {
       i++;          //基本操作
       s+=i;         //基本操作
    }
}
```

该算法中的基本操作是 while 循环中的语句，设 while 循环次数为 $f(n)$，则变量 i 从 0 到 $f(n)$，

因此循环次数为 $f(n)\times(f(n)+1)/2\leqslant n$，则 $f(n)\leqslant\sqrt{8n}$，故时间复杂度为 $O(\sqrt{n})$。

【例 1.4】一个算法所需时间由以下递归方程表示，分析算法的时间复杂度。

$$T(n)=\begin{cases}1, & n=1\\ 2T(n-1)+1, & n>1\end{cases}$$

根据以上递归方程，可得 $T(n)=2T(n-1)+1=2(2T(n-2)+1)+1=2^2T(n-2)+2+1$

$=2^2(2T(n-3)+1)+2+1$

$=\cdots$

$=2^{k-1}(2T(n-k)+1)+2^{k-2}+\cdots+2+1$

$=\cdots$

$=2^{n-2}(2T(1)+1)+2^{n-2}+\cdots+2+1$

$=2^{n-1}+\cdots+2+1$

$=2^n-1$

因此，该算法的时间复杂度为 $O(2^n)$。

1.6.4　算法空间复杂度

空间复杂度作为算法所需存储空间的量度，记作 $S(n)=O(f(n))$。其中，n 为问题的规模，$f(n)$ 为语句关于 n 的所占存储空间的函数。一般情况下，一个程序在机器上执行时，除了需要存储程序本身的指令、常数、变量和输入数据外，还需要存储对数据操作的存储单元。若输入数据所占空间只取决于问题本身，和算法无关，这样只需要分析该算法在实现时所需的辅助单元即可。若算法执行时所需的辅助空间相对于输入数据量而言是个常数，则称此算法为原地工作，空间复杂度为 $O(1)$。

【例 1.5】以下是一个简单插入排序算法，分析算法的空间复杂度。

```
for(i=0;i<n;i++)
{
   t=a[i+1];
   j=i;
   while(j>=0 && t<a[j])
   {
      a[j+1]=a[j];
      j--;
   }
   a[j+1]=t;
}
```

该算法借助了变量 t，与问题规模 n 的大小无关，空间复杂度为 $O(1)$。

【例 1.6】以下算法是求 n 个数中的最大者，分析算法的空间复杂度。

```
int FindMax(int a[], int n)
{
   int m;
   if(n<=1)
      return a[0];
   else
   {
```

```
        m=FindMax(a,n-1);
        return a[n-1]>=m?a[n-1]:m;
    }
}
```

设 FindMax(a,n)占用的临时空间为 $S(n)$，由以上算法可得到以下占用临时空间的递推式。

$$S(n)=\begin{cases}1, & n=1\\ S(n-1)+1, & n>1\end{cases}$$

则有 $S(n)=S(n-1)+1=S(n-2)+1+1=\cdots=S(1)+1+1+\cdots+1=O(n)$。因此，该算法的空间复杂度为 $O(n)$。

1.7 学好数据结构的秘诀

作为计算机专业的一名“老兵”，笔者从事数据结构和算法的研究已经有二十余年了，在学习的过程中，也会遇到一些问题，但在解决问题时，积累了一些经验，为了让读者在学习数据结构的过程中少走弯路，本节将分享一些笔者个人在学习数据结构与算法时的经验，希望对读者的学习有所帮助。

1. 明确数据结构的重要性，树立学好数据结构的信心

数据结构是计算机、软件工程等相关专业的核心课程，是操作系统、数据库原理、编译原理、人工智能、算法设计与分析等课程的重要基础。当今最流行的人工智能、机器学习中的所有算法无不蕴含着数据结构与算法知识，数据结构也是计算机专业硕士研究生入学考试，计算机软件水平考试、等级考试的必考内容之一，其重要性不言而喻。

一定要树立学好数据结构与算法的信心。万事开头难，学习任何一样新东西，都有一个适应过程，对于初学者来说，数据结构有些枯燥、乏味，但当你将数据结构中的知识与日常生活结合起来时，就不会觉得那么枯燥和乏味了，你会觉得它很有用。在学习数据结构与算法的过程中，主要困难可能是出于以下原因：一个是数据结构的概念比较抽象，不容易理解；另一个是没有熟练掌握一门程序设计语言。数据结构中的概念其实就是对日常生活中的具体问题进行了抽象，因此，只要与日常生活多联系，这些抽象的概念就变得好理解了。另外，一定要熟练掌握 C 语言/Java 语言工具，从代码中去领会算法思想。

2. 熟练掌握程序设计语言，变腐朽为神奇

程序语言是学习数据结构和算法设计的基础，很显然，没有良好的程序设计语言能力，就不能很好地把算法用程序设计语言描述出来，程序设计语言和数据结构、算法的关系就像是画笔和画家的思想关系一样，程序设计语言就是画笔，数据结构、算法就是画家的思想，即便画家的水平很高，如果不会使用画笔，再美的图画也无法展现出来。

可见，要想学好数据结构，必须至少熟练掌握一门程序设计语言，如 C 语言、C++语言、Java 语言等。

3. 结合生活实际，变抽象为具体

数据结构是一项把实际问题抽象化和进行复杂程序设计的工程。它要求学生不仅具备 C 语言等高级程序设计语言的基础，而且还要学会掌握把复杂问题抽象成计算机能够解决的离

散的数学模型的能力。在学习数据结构的过程中，要将各种结构与实际生活结合起来，把抽象的东西具体化，以便理解。例如，学到队列时，很自然就会联想到火车站售票窗口前面排起的长长的队伍，这支长长的队伍其实就是队列的具体化，这样就会很容易理解关于队列的概念，如队头、队尾、出队、入队等。

4. 多思考，多上机实践

数据结构既是一门理论性较强的学科，也是一门实践性很强的学科。特别是对于初学者而言，接触到的算法相对较少，编写算法还不够熟练，俗话说“熟能生巧，勤能补拙”，因此，只有多看有关算法和数据结构方面的图书，认真理解其中的算法思想。除了阅读算法之外，还要自己动手写算法，并在计算机上调试，这样才能知道编写的算法是否正确，存在哪些错误和缺陷，以避免今后再犯类似错误，长此以往，自己的算法和数据结构水平才能快速提高。

有的表面上看是正确的程序，在计算机上运行后才发现隐藏的错误，特别是很细微的错误，只有多试几组数据，才知道程序到底是不是正确。因此，对于一个程序或算法，除了仔细阅读程序或算法判断是否存在逻辑错误外，还需要上机调试，在可能出错的地方设置断点，单步跟踪调试程序，观察各变量的变化情况，才能找到具体哪个地方出了问题。有时，可能仅仅是误输入了一个符号或变量，就可能产生错误，这种错误往往不容易发现，只有上机调试才能发现错误。因此，在学习数据结构与算法的时候一定要多上机实践。

只要能做到以上几点，选择一本好的数据结构教材或参考书（最好算法完全用 C 语言实现，有完整代码），加上读者的勤奋，学好数据结构并不是什么难事。

第 2 章
数据结构与算法基础

“工欲善其事，必先利其器”。C 语言是数据结构与算法的主要描述语言，要想真正掌握好数据结构与算法，读懂并写出逻辑清晰、高效优雅的算法，必须首先对 C 语言了如指掌。本章旨在引领读者复习 C 语言中的一些重点和难点，为今后的数据结构与算法学习扫清语言障碍。本章主要内容包括 C 语言开发环境、函数与递归、指针、参数传递、动态内存分配及结构体、联合体。

本章重点和难点：

- 递归函数的实现和递归如何转换为非递归。
- 指针数组、数组指针、函数指针的定义及使用。
- 理解传地址调用中变量的变化情况。
- 链表的定义及其操作。

2.1 递归与非递归

递归是 C 语言学习过程中的重点和难点。在数据结构与算法实践过程中，经常会遇到利用递归实现算法的情况。递归是一种分而治之、将复杂问题转换为简单问题的求解方法。使用递归可以使编写出的程序简洁、结构清晰，程序的正确性很容易证明，不需要了解递归调用的具体细节。

2.1.1 函数的递归调用

简单来说，函数的递归调用就是自己调用自己，即一个函数在调用其他函数的过程中，又出现了对自身的调用，这种函数称为递归函数。函数的递归调用可分为直接递归调用和间接递归调用。其中，在函数中直接调用自己称为函数的直接递归调用，如图 2-1 所示；如果函数 f1 调用了函数 f2，函数 f2 又调用了 f1，这种调用方式称为间接递归调用，如图 2-2 所示。

函数的递归调用就是自己调用自己，可以直接调用自己也可以间接调用自己。

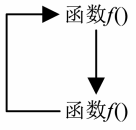


图 2-1　直接递归调用过程

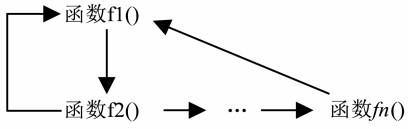


图 2-2　间接递归调用过程

在用递归解决实际问题时，递归函数只需知道最基本问题的解。在递归函数中，遇到基本问题时仅返回一个值，在解决较为复杂的问题时，通过将复杂的问题化解为比原有问题更简单、规模更小的问题，最后把复杂问题变成一个基本问题，而基本问题的答案是已知的，基本问题解决后，比基本问题大一点的问题也得到解决，直到原有问题得到解决。

2.1.2　递归应用举例

【例 2-1】利用递归求 $n!$。

【分析】n 的阶乘递归定义为 $n!=n\times(n-1)!$，当 $n=5$ 时，则有

5！=5×4！

4！=4×3！

3！=3×2！

2！=2×1！

1！=1×0！

0！=1

递归计算 5！的过程如图 2-3 所示。因为 5!＝5×(5−1)!，因此，如果能求出(5−1)!，也就求出了 5!；又因为(5−1)!＝(5−1)×(5−2)!，因此，如果能求出(5−2)!，则也就能求出(5−1)!；……最后一直递归到 1！＝1×0!。而 0！的值为 1 是已知条件，当得到了 0！的值后，就可以得到 1！的值，按上述分析过程逆向推回去，从而得到 2!、3!、4！和 5！的值。

这样就把求解问题 5！转换为 5 与 4!相乘，4！的值是未知的，接着继续把求解 4!转换为 4 与 3！相乘，这样将问题规模不断缩小，直到把原问题转换为求解 0！=1 这个最基本的已知问题为止。

根据上述分析可知，求解 5！可分成两个阶段：第一阶段是由未知逐步推得已知的过程，称为“回推”；第二阶段是与回推过程相反的过程，即由已知逐步推得最后结果的过程，称为“递推”。图中的左半部分是回推过程，回推过程在计算出 0！=1 时停止调用；右半部分是递推过程，直到计算出 5！=120 为止。

综上，递归求 $n!$的过程分以下两个过程。

（1）当 $n=0$（递归调用结束，即递归的出口）时，返回 1。

（2）当 $n\neq0$ 时，需要把复杂问题分解成较为简单的问题，直到分解成最简单的问题 0！=1 为止。

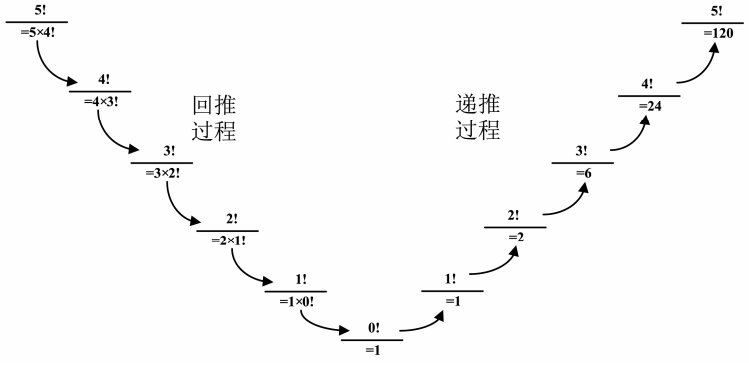


图 2-3　求 5！递归调用过程

递归求 *n*!的算法实现如下。

```
#include<stdio.h>
#include<stdlib.h>
long factorial(int n);                              /*求阶乘函数声明*/
void main()                                         /*主函数*/
{
    int num;                                        /*定义循环变量 num*/
    for(num=0;num<10;num++)                         /*for 循环处理*/
       printf("%d!=%ld\n",num,factorial(num));      /*输出阶乘计算结果*/
    system("pause");
}
long factorial(int n)
/*递归求 n! 函数实现*/
{
    if(n==0)                                        /*当 n=0 时,递归调用出口*/
       return 1;                                    /*0! =1 是最基本问题的解*/
    else                                            /*否则*/
       return n*factorial(n-1);                     /*递归调用将问题分解成较为简单的子问题*/
}
```

程序运行结果如图 2-4 所示。

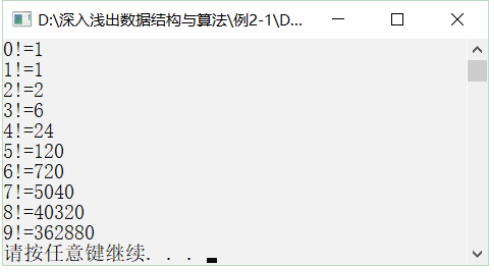


图 2-4　程序运行结果

【例 2-2】要求利用递归实现求 *n* 个数中的最大者。

【分析】假设元素序列存放在数组 *a*[]中，数组 *a*[]中 *n* 个元素的最大者可以通过将 *a*[*n*−1]与前 *n*−1 个元素最大者比较之后得到，而前 *n*−1 个元素的最大者可通过将 *a*[*n*−2]与前 *n*−2 个元素的最大者比较之后得到，依次类推。

也就是说，数组 *a*[]中只有一个元素时，最大者是 *a*[0]；超过一个元素时，则要比较最后一个元素 *a*[*n*−1]和前 *n*−1 个元素中的最大者，其中较大的一个即所求。而求前 *n*−1 个元素的最大者需要继续调用 findmax()函数得到。

求 *n* 个数中的最大者的递归算法实现如下。

```
#include<stdio.h>
#include<stdlib.h>
#define N 200                                   /*宏定义 N=200*/
int findmax(int a[],int n);                     /*求数组中最大者的函数声明*/
void main()
{
    int a[N],n,i;                               /*定义变量*/
    printf("请输入 n 的值:");                    /*输出提示信息*/
    scanf("%d",&n);                             /*从键盘输入 n 的值*/
```

```
    printf("请依次输入%d 个数: \n",n);              /*输出提示信息*/
    for(i=0;i<n;i++)                               /*输入 n 个整数,存入数组 a 中*/
       scanf("%d",&a[i]);
    printf("在这%d 个数中,最大的元素是:%d\n",n,findmax(a,n)); /*输出 n 个数中最大的一个*/
    system("pause");
}
int findmax(int a[],int n)                         /*求 n 个数中最大者的函数实现*/
{
    int m;                                         /*定义变量*/
    if(n<=1)                                       /*如果只有一个数*/
       return a[0];                                /*则数组中第一个数就是最大的数*/
    else                                           /*否则*/
    {
       m=findmax(a,n-1);                           /*通过递归求前 n-1 个数中的最大者,将其赋给 m*/
       return a[n-1]>=m?a[n-1]:m;
       /*若第 n 个数大于或等于 m,则第 n 个数就是最大者; 否则,m 为最大者*/
    }
}
```

程序的运行结果如图 2-5 所示。

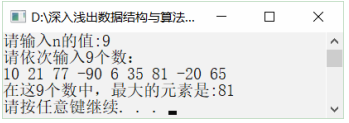


图 2-5　递归实现求 *n* 个数的最大者程序运行结果

2.1.3　迭代与递归

大量的递归调用会耗费大量的时间和内存。每次递归调用都会建立函数的一个备份，会占用大量的内存空间。迭代则不需要反复调用函数和占用额外的内存。通过分析递归求 *n* 的阶乘 *n*!的计算过程，可以转换为非递归实现，其非递归实现如下。

```
int NonRecFact(int n)
/*非递归求前 n 的阶乘*/
{
    int i,s=1;
    for(i=1;i<=n;i++)              /*通过迭代求 n 的阶乘*/
       s*=i;
    return s;                      /*返回计算结果*/
}
```

对于大整数问题，考虑到 *n* 值非常大的情况，运算结果超出一般整数的位数，可以用一维数组存储长整数，数组中的每个元素只存储长整数的一位数字。如有 *m* 位长整数 *N* 用数组 a[]存储，N=a[m]*10^{m-1}+a[m-1]*10^{m-2}+…+a[2]*10^1+a[1]*10^0，并用 *a*[0]存储长整数 *N* 的位数 *m*，即 *a*[0]=*m*。按上述约定，数组的每个元素存储 *k* 的阶乘 *k*!的每一位数字，并从低位到高位依次存储于数组的第 2 个元素、第 3 个元素……例如，6!=720 在数组中的存储形式如图 2-6 所示。

3	0	2	7	…	…	…	…

图 2-6　*k*!在数组中的存储情况

其中，第 1 个元素 3 表示长整数是一个 3 位数，接着是低位到高位的 0、2、7，表示长整数 720。

在计算阶乘 $k!$时，可以采用对已求得的阶乘$(k-1)!$连续累加 $k-1$ 次(即得到 $k\times(k-1)!$)后得到。例如，已知 5!=120，计算 6!，可对原来的 120 再累加 5 次 120(即得到 6×5!)得到 720。具体程序实现如下。

```
#include<stdio.h>
#include<malloc.h>                          /*包含该头文件的目的是使用了函数 malloc*/
#include<stdlib.h>
#define N 100                               /*宏定义 N=100*/
void fact(int a[],int k)                    /*求阶乘的非递归实现*/
{
   int *b,m,i,j,r,carry;                    /*定义变量*/
   m=a[0];                                  /*将正整数的位数赋给 m*/
   b=(int*)malloc(sizeof(int)*(m+1));           /*申请分配指定字节的内存空间并赋值给 b*/
   for(i=1;i<=m;i++)
      b[i]=a[i];                            /*将数组 a 的数据保存到数组 b 中*/
   for(j=1;j<k;j++)                         /*for 外层循环*/
   {
      for(carry=0,i=1;i<=m;i++)             /*for 内层循环*/
      {
         r=(i<=a[0]?a[i]+b[i]:a[i])+carry;      /*阶乘计算和存储*/
         a[i]=r%10;                         /*数组的每个元素存储 k 的阶乘 k!的每一位数字*/
         carry=r/10;
      }
      if(carry)                             /*是否满足条件*/
         a[++m]=carry;                      /*赋值*/
   }
   free(b);                                 /*释放资源*/
   a[0]=m;                                  /*将求得的整数位数存入 a[0]*/
}
void write(int *a,int k)                    /*write 函数实现*/
{
   int i;                                   /*定义变量*/
   printf("%4d!=",k);                       /*输出提示信息*/
   for(i=a[0];i>0;i--)
         printf("%d",a[i]);                 /*依次输出数组中的元素,即阶乘*/
   printf("\n");}
void main()
{
   int a[N],n,k;
   printf("请输入正整数 n 的值:"); scanf("%d",&n);
   a[0]=1;a[1]=1;                           /*将 1 的阶乘存入数组 a*/
   write(a,1);                              /*调用 write 函数输出 n 的阶乘*/
   for(k=2;k<=n;k++)                        /*依次求 2～n 的阶乘*/
   {
      fact(a,k);                            /*调用 fact 函数求 k 的阶乘*/
      write(a,k);                           /*调用 write 函数输出 k 的阶乘*/
   }
   system("pause");
}
```

程序运行结果如图 2-7 所示。

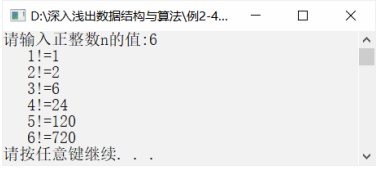


图 2-7　使用非递归求 n 的阶乘

对于较为简单的递归问题，可以利用迭代将其转换为非递归。而对于较为复杂的递归问题，需要使用数据结构中的栈来消除递归。

2.2　指针

指针是 C 语言中的一个重要概念，也是最不容易掌握的内容。指针常常用在函数的参数传递和动态内存分配中。指针与数组相结合，使引用数组元素的形式更加多样化，访问数组元素的手段更加灵活；指针与结构体相结合，利用系统提供的动态存储手段，能构造出各种复杂的动态数据结构；利用指针形参，使函数能实现传递地址形参和函数形参的要求。在“数据结构”课程中，指针的使用非常频繁，因此，要想真正掌握数据结构，就需要灵活、正确地使用指针。

2.2.1　什么是指针

指针是一种变量，也称指针变量，它的值不是整数、浮点数和字符，而是内存地址。指针的值就是变量的地址，而变量又拥有一个具体值。因此，可以理解为变量名直接引用了一个值，指针间接地引用了一个值。

在理解指针之前，先来了解下地址的概念。图 2-8 展示了变量在内存中的存储情况。假设 *a*、*b*、*c*、*d*、bPtr 分别是 5 个变量，其中，*a*、*b*、*c*、*d* 是整型变量，bPtr 是指针变量。整型变量在内存中占用 4B，变量 *a* 的存放地址是 2000～2003 四个内存单元，变量 *b* 存放在 2004～2007 内存单元中，变量 bPtr 存放在 4600～4603 四个内存单元中。整型变量 *a*、*b*、*c*、*d* 的内容分别是 25、12、78、5，而指针变量 bPtr 的内容是一个地址，为 2004 开始的内存地址，即 bPtr 存放的是变量 *b* 的地址，换句话说，就是 bPtr 指向变量 *b* 的存储位置，可以用一个箭头表示从地址是 4600 的位置指向变量地址为 2004 的位置。

一个存放变量地址的类型称为该变量的“指针”。如果有一个变量用来存放另一个变量的地址，则称这个变量为指针变量。在图 2-9 中，qPtr 用来存放变量 *q* 的地址，qPtr 就是一个指针变量。

在 C 语言中，所有变量在使用前都需要声明。例如，声明一个指针变量的语句如下。

```
int *qPtr,q;
```

q 是整型变量，表示要存放一个整数类型的值；qPtr 是一个整型指针变量，表示要存放一个变量的地址，而这个变量是整数类型。qPtr 叫作一个指向整型的指针。

说明：在声明指针变量时，“*”只是一个指针类型标识符，指针变量的声明也可以写成 int* qPtr。

指针变量可以在声明时赋值，也可以在声明后赋值。例如，在声明时为指针变量赋值的语句如下。

```
int q=12;                                   /*声明整型变量 q 并赋值*/
int *qPtr=&q;                               /*声明指针变量 qPtr 并赋值*/
```

或在声明后为指针变量赋值，语句如下。

```
int q=12,*qPtr;                             /*声明一个整型变量和一个指针变量*/
qPtr=&q;                                    /*为指针变量赋值*/
```

这两种赋值方法都是把变量 *q* 的地址赋值给指针变量 qPtr。qPtr=&q 叫作指向变量 *q*，其中，

&是取地址运算符，表示返回变量 *q* 的地址。指针变量 qPtr 与变量 *q* 的关系如图 2-9 所示。

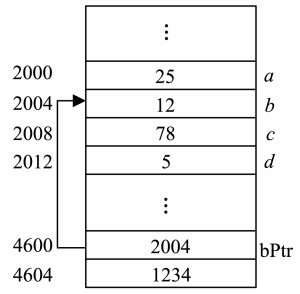


图 2-8　指针变量在内存中的表示

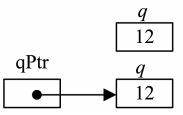


图 2-9　*q* 直接引用一个值和 qPtr 间接引用一个变量 *q*

直接引用和间接引用可以用日常生活中的两个抽屉来形象说明。有两个抽屉 A 和 B，抽屉 A 有一把钥匙，抽屉 B 也有一把钥匙。为了方便，可以把两把钥匙都带在身上，需要取抽屉 A 中的东西时直接用钥匙 A 打开抽屉；也可以为了安全考虑，把钥匙 A 放到抽屉 B 中，把抽屉 B 的钥匙带在身上，需要取抽屉 A 中的东西时，先打开抽屉 B，再取出抽屉 A 的钥匙，然后打开抽屉 A，取出需要的东西。前一种方法就相当于通过变量直接引用，后一种方法相当于通过指针间接引用。其中，抽屉 B 的钥匙相当于指针变量，抽屉 A 的钥匙相当于一般的变量。

2.2.2　指针变量的间接引用

与普通变量一样，指针变量也可以对数据进行操作。指针变量主要通过取地址运算符&和指针运算符*来存取数据。例如，&*a* 指的是变量 *a* 的地址，*ptr 表示变量 ptr 所指向的内存单元存放的内容。下面通过具体例子说明&和*运算符及指针变量的使用。

【例 2-3】利用变量和指针变量存取数据。

【分析】主要考查如何利用&和*运算符来存取变量中的数据，取地址运算符&和指针运算符*是互逆的操作，应灵活掌握两个运算符的使用技巧。

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int q=12;
    int *qPtr;                    /*声明指针变量 qPtr*/
    qPtr=&q;                      /*指针变量 qPtr 指向变量 q*/
    /*打印变量 q 的地址和 qPtr 的内容*/
    printf("q 的地址是: %p\nqPtr 中的内容是: %p\n",&q,qPtr);
    /*打印 q 的值和 qPtr 指向变量的内容*/
    printf("q 的值是: %d\n*qPtr 的值是: %d\n",q,*qPtr);
    /*运算符'&'和'*'是互逆的*/
    printf("&*qPtr=%p,*&qPtr=%p\n 因此有&*qPtr=*&qPtr\n",&*qPtr,*&qPtr);
    system("pause");
}
```

程序运行结果如图 2-10 所示。

&和*作为单目运算符，结合性是从右到左，优先级别相同，因此对于表达式&*qPtr 来说，先进行*运算，后进行&运算。因为 qPtr 是指向变量 *q* 的，所以*qPtr 的值为 *q*，&*qPtr 就是对 *q* 取地址，即&q，q 的地址。*&qPtr 是先进行取地址运算即&qPtr，即 qPtr 的地址，然后进行*运算，那么*&qPtr 就是 qPtr 本身，即 *q* 的地址。因此，&*qPtr 和*&qPtr 是等价的。

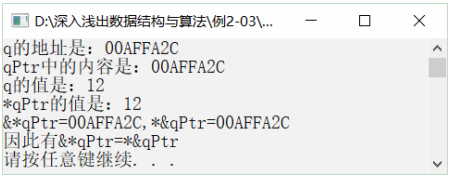


图 2-10　利用指针变量进行存取操作的程序运行结果

注意：指针变量也是一种数据类型，用来存放变量的地址。指针变量的类型应和所指向的变量的类型一致。例如，整型指针只能指向整型变量，不能指向浮点型变量。指针变量只能用来存放地址，不能将一个整型值赋给一个指针变量。一般所说的变量指针指的是变量的地址。指针是指的地址，指针变量就是存储地址的变量。

2.2.3　指针与数组

指针可以与变量结合，也可以与数组结合使用。指针数组和数组指针是两个截然不同的概念，指针数组是一种数组，该数组存放的是一组变量的地址。数组指针是一个指针，表示该指针是指向数组的指针。

1. 指向数组元素的指针

指针可以指向变量，也可以指向数组及数组中的元素。

例如，定义一个整型数组和一个指针变量，语句如下。

```
int a[5]={10,20,30,40,50};      /*定义数组并赋值*/
int *aPtr;                      /*定义指针变量*/
```

这里的 *a* 是一个数组，它包含 5 个整型数据。变量名 *a* 就是数组 *a* 的首地址，它与&*a*[0]等价。如果令 aPtr=&*a*[0]或者 aPtr=*a*，则 aPtr 也指向了数组 *a* 的首地址，如图 2-11 所示。

也可以在定义指针变量时直接赋值，以下语句是等价的。

```
int *aPtr=&a[0];                /*定义并同时初始化指针变量,将数组 a 的首地址赋给 aPtr*/

int *aPtr;                      /*先定义指针变量 aPtr*/
aPtr =&a[0];                    /*然后初始化,将数组 a 的首地址赋给 aPtr*/
```

与整型、浮点型数据一样，指针也可以进行算术运算，但含义却不同。当一个指针加（或减）1 并不是指针值增加（或减少）1，而是使指针指向的位置向后（或向前）移动了一个位置，即加上（或减去）该整数与指针指向对象的大小的乘积。例如，对于 aPtr+=3，如果一个整数占用 4B，则相加后 aPtr=2000+4×3=2012（这里假设指针的初值是 2000）。同样指针也可以进行自增（++）运算和自减（--）运算。

也可以用一个指针变量减去另一个指针变量。例如，指向数组元素的指针 aPtr 的地址是 2008，另一个指向数组元素的指针 bPtr 的地址是 2000，则 *a*=aPtr-bPtr 的运算结果就是把从 aPtr 到 bPtr 之间的元素个数赋给 *a*，元素个数为（2008−2000）/4=2（假设整数占用 4B）。

也可以通过指针来引用数组元素。例如：

```
*(aPtr+2);
```

如果 aPtr 是指向 *a*[0]，即数组 *a* 的首地址，则 aPtr+2 就是数组 *a*[2]的地址，*(aPtr+2)就是 30。

注意：指向数组的指针可以进行自增或自减运算，但是数组名则不能进行自增或自减运算，这是因为数组名是一个常量指针，常量值是不能改变的。

【例 2-4】用指针引用数组元素并打印输出。

【分析】主要考查指针与数组结合的运算，有指针对数组的引用及指针的加、减运算。

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int a[5]={10,20,30,40,50};          /*定义数组并赋值*/
    int *aPtr,i;                        /*指针变量声明*/
    aPtr=&a[0];                         /*指针变量指向变量 a*/
    for(i=0;i<5;i++)                    /*通过数组下标引用元素的方式输出数组元素*/
       printf("a[%d]=%d\n",i,a[i]);
    for(i=0;i<5;i++)                    /*通过数组名引用元素的方式输出数组元素*/
       printf("*(a+%d)=%d\n",i,*(a+i));
    for(i=0;i<5;i++)                    /*通过指针变量下标引用元素的方式输出数组元素*/
       printf("aPtr[%d]=%d\n",i,aPtr[i]);
    for(aPtr=a,i=0;aPtr<a+5;aPtr++,i++) /*通过指针变量偏移引用元素的方式输出数组元素*/
       printf("*(aPtr+%d)=%d\n",i,*aPtr);
    system("pause");
}
```

程序中共有 4 个 for 循环，其中：第 1 个 for 循环是利用数组的下标访问数组的元素；第 2 个 for 循环是利用数组名访问数组的元素，在 C 语言中，地址也可以像一般的变量一样进行加、减运算，但是指针的加 1 和减 1 表示的是一个元素单元；第 3 个 for 循环是利用指针访问数组中的元素；第 4 个 for 循环则是先将指针偏移，然后访问该指针所指向的内容。

程序运行结果如图 2-12 所示。

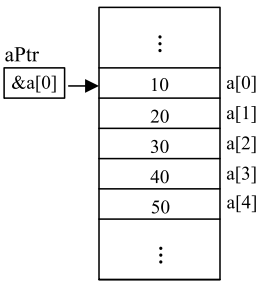


图 2-11　数组的指针与数组在内存中的关系

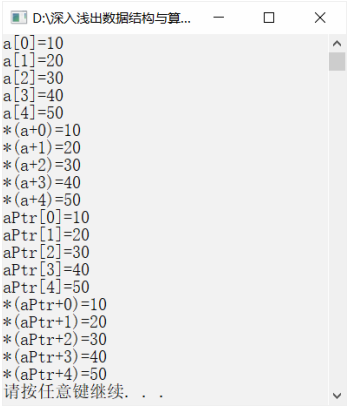


图 2-12　指针引用数组元素的运行结果

2. 指针数组

指针数组其实也是一个数组，只是数组中的元素是指针类型的数据。换句话说，指针数组中的每一个元素都是一个指针变量。

定义指针数组的方式如下。

```
int *p[4];                          /*定义指针数组*/
```

由于[]运算符优先级比*高，*p* 优先与[]结合，形成 *p*[]数组形式，然后与*结合，表示该数组是指针类型的，每个数组元素是一个指向整型的变量。从字面上理解，指针数组首先是一个数组，这个数组存放的是指针类型的变量。

指针数组常常用于存储一些长度不等的字符串数据。有的读者可能会问，为什么不存放在二维数组中？这是因为字符串长度不等，若将这些字符串存放在二维数组中，就需要定义一个

能容纳最长字符串的二维数组，这样就会出现一部分存储空间不能得到有效利用。

字符串常用于存储一些长度不等的字符串数据，字符串“C Programming Language”“Python Programming”“Data Structure”和“Machine Learning”在二维数组中的存储情况如图 2-13 所示。

C		P	r	o	g	r	a	m	m	i	n	g		L	a	n	g	u	a	g	e	\0
P	y	t	h	o	n		P	r	o	g	r	a	m	m	i	n	g	\0				
D	a	t	a		S	t	r	u	c	t	u	r	e	\0								
M	a	c	h	i	n	e		L	e	a	r	n	i	n	g	\0						

图 2-13　字符串在二维数组中的存储情况

不难看出，利用二维数组保存多个字符串时，为了保证能存储所有的字符串，必须按最长的字符串长度来定义二维数组的列数。为了节省存储单元，可以采用指针数组保存字符串，定义如下。

```
    char *book[4]={"C Programming Language","Python Programming","Data Structure","Machine
Learning"};
```

以上字符串在指针数组中的存储情况如图 2-14 所示。

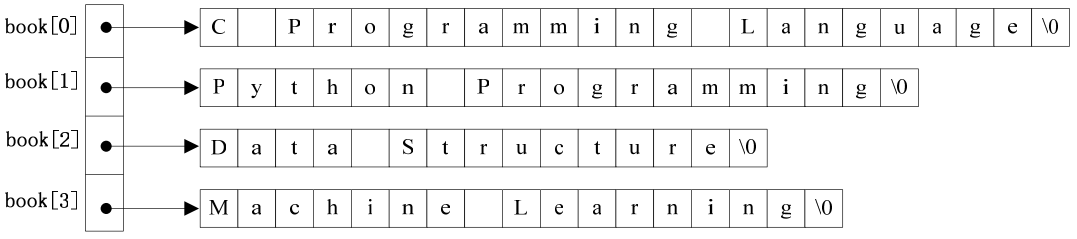


图 2-14　字符串在指针数组中的存储情况

在字符串比较多且长度不一时，利用指针数组存储就可以大大地节省内存空间。

【例 2-5】用指针数组保存字符串并将字符串打印输出。

【分析】主要考查指针的应用及对指针数组概念的理解，其实 book[4]就是一个特殊的数组，book[0]、book[1]、book[2]、book[3]分别存放指向 4 个字符串的指针，即数组保存的是各个字符串的首地址。

```
    #include<stdio.h>
    #include<stdlib.h>
    void main()
    {
        /*定义指针数组*/
        char *book[4]={"C Programming Language","Python Language"," Data Structure ","Machine
Learning"};
        int n=4;                                          /*指针数组元素的个数*/
        int i;
        char *aPtr;
        /*第 1 种方法输出：通过数组名输出字符串*/
        printf("第 1 种方法输出：通过指针数组的数组名输出字符串:\n");
        for(i=0;i<n;i++)
           printf("第%d 个字符串：%s\n",i+1,book[i]);
        /*第 2 种方法输出：通过指向数组的指针输出字符串*/
        printf("第 2 种方法输出：通过指向数组的指针输出字符串:\n");
        for(aPtr=book[0],i=0;i<n;aPtr=book[i])
        {
           printf("第%d 个字符串：%s\n",i+1,aPtr);       /*输出字符串*/
```

```
        i++;                                                    /*自增 1*/
    }
    system("pause");
}
```

程序运行结果如图 2-15 所示。

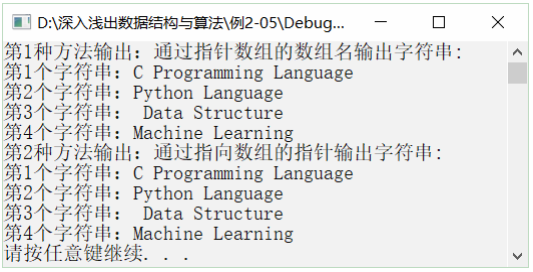


图 2-15　字符数组输出结果

3. 数组指针

数组指针是指向数组的一个指针，如下定义。

```
int (*p)[4];
```

其中，p 是指向一个拥有 4 个元素的数组指针，数组中每个元素都为整型。与指针数组定义相比，数组指针的定义中多了一对圆括号，这里*p 两边的圆括号不可以省略。在这个定义中，p 仅仅是一个指针，不过这个指针有点特殊，这里的 p 指向的是包含 4 个元素的一维数组。数组指针 p 与它指向的数组之间的关系可以用图 2-16 来表示。

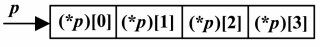


图 2-16　数组指针 p 的表示

如果按以下方式使用指针变量：

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

数组指针 p 与数组 a 中元素之间的关系如图 2-17 所示。其中，(*p)[0]、(*p)[1]、(*p)[2]、(*p)[3]分别保存的是元素值为 1、2、3、4 的地址。p、p+1 和 p+2 分别指向二维数组的第 1 行、第 2 行和第 3 行，p+1 表示将指针 p 移动到下一行。

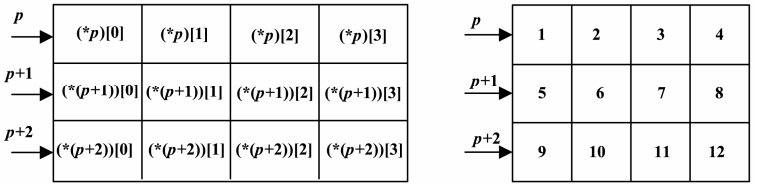


图 2-17　数组指针 p 与二维数组的对应关系

(p+1)+2 表示数组 a 第 1 行第 2 列的元素的地址，即&a[1][2]，(*(p+1)+2)表示 a[1][2]的值即 7，其中，1 表示行，2 表示列。

【例 2-6】在屏幕上打印图 2-17 中数组指针 p 及数组 a 中的元素。

【分析】主要考查利用数组指针引用数组中的元素的方法。数组指针 p 与数组 a 中元素的对应关系如图 2-17 所示。通过利用数组指针 p 引用数组 a 中的元素并输出 p 的值，以验证对指针

引用的正确性，加深对数组指针的理解。实现代码如下。

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}}; /*定义数组 a 并赋值*/
    int (*p)[4];                        /*声明数组指针变量 p*/
    int row,col;                        /*定义变量*/
    p=a;                                /*指针 p 指向数组元素为 4 的数组*/
    /*打印输出数组指针 p 指向的数组元素值*/
    for(row=0;row<3;row++)
    {
        for(col=0;col<4;col++)
            printf("a[%d,%d]=%-4d",row,col,*(*(p+row)+col));
                                        /*通过数组指针 p 逐个输出数组元素值*/
        printf("\n");
    }
    /*通过改变指针 p 指向 a 的行地址输出数组 a 中每个元素的地址*/
    for(p=a,row=0;p<a+3;p++,row++)   /*修改 p 指向数组 a 的行地址访问每一行元素*/
    {
        for(col=0;col<4;col++)          /*控制数组中每一行的元素*/
            printf("(*p[%d])[%d]=%p",row,col,((*p)+col));    /*输出每个元素的地址*/
        printf("\n");
    }
system("pause");
}
```

程序运行结果如图 2-18 所示。

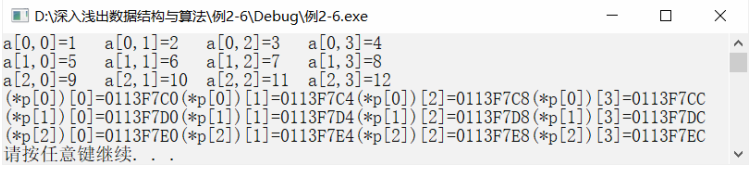


图 2-18　打印输出数组指针和数组元素

注意：区别数组指针和指针数组。数组指针首先是一个指针，并且它是一个指向数组的指针。指针数组首先是一个数组，并且它是保存指针变量的数组。

2.2.4　指针函数与函数指针

与指针数组、数组指针一样，指针函数与函数指针也是一对孪生兄弟，也是常常容易混淆的概念。顾名思义，指针函数是一种函数，它表示函数的返回值是指针类型；函数指针是指针的一种，它表示该指针指向一个函数。

1. 指针函数

指针函数是指函数的返回值是指针类型的函数。在 C 语言中，一个函数的返回值可以是整型、实型和字符型，也可以是指针类型。例如，以下语句是一个指针函数的声明。

```
float *func(int a,int b);
```

其中，func 是函数名，前面的*表明返回值的类型是指针类型，因为前面的类型标识符是 float，所以返回的指针是指向浮点型的。该函数有两个参数，参数类型都是整型。下面通过一个具体实例来介绍指针函数的用法。

【例 2-7】假设若干个学生的成绩在二维数组中存放，要求输入学生编号，利用指针函数实现其成绩的输出。

【分析】主要考查指针函数的使用。学生成绩存放在二维数组中，每一行存放一个学生的成绩，通过输入学生编号，返回该学生存放成绩的地址，然后利用指针输出每一门的学生成绩。

```
#include<stdio.h>
#include<stdlib.h>
int *FindAddress(int (*ptr)[4],int n);       /*声明查找成绩地址函数*/
void Display(int a[][4],int n,int *p);       /*声明输出成绩函数*/
void main()
{
    int row,n=4;
    int *p;                                  /*定义指针变量*/
    int score[3][4]={{83,78,79,88},{71,88,92,63},{99,92,87,80}};/*定义数组并赋值*/
    printf("请输入学生编号(1 或 2 或 3).输入 0 退出程序.\n");
    scanf("%d",&row);                        /*输入要输出学生成绩的编号*/
    while(row)                               /*若学生编号不为 0*/
    {
       if(row==1||row==2||row==3)
       {
          printf("第%d 个学生的成绩 4 门课的成绩是: \n",row);
          p=FindAddress(score,row-1);        /*调用指针函数*/
          Display(score,n,p);                /*调用输出成绩函数*/
          printf("请输入学生编号(1 或 2 或 3).输入 0 退出程序.\n");
          scanf("%d",&row);
       }
       else
       {
          printf("输入不合法,请重新输入(1 或 2 或 3),输入 0 退出程序.\n");
          scanf("%d",&row);
       }
    }
    system("pause");
}
int *FindAddress(int (*ptrScore)[4],int n)
/*查找某条学生成绩记录地址函数.通过传递的行地址找到要查找学生成绩的地址,并返回行地址*/
{
    int *ptr;
    ptr=*(ptrScore+n);                   /*修改行地址,即找到学生的第一门课成绩的地址*/
    return ptr;                          /*返回学生第一门课成绩的地址*/
}
void Display(int a[][4],int n,int *p)
/*输出学生成绩的实现函数.利用传递过来的指针输出每门课的成绩*/
{
    int col;
    for(col=0;col<n;col++)
       printf("%4d",*(p+col));           /*输出查找学生的每门课成绩*/
    printf("\n");
}
```

程序运行结果如图 2-19 所示。

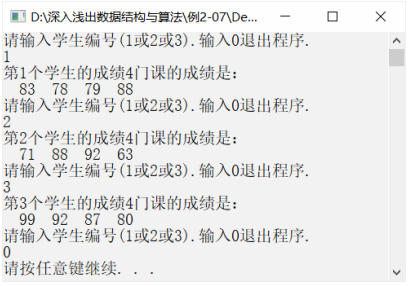


图 2-19　通过指针函数返回指针并输出成绩的运行结果

主函数通过语句 *p*=FindAddress(score,row-1)；调用指针函数*FindAddress(int(*ptrScore)[4],int *n*)，并把二维数组的行地址传递给形式参数 ptrScore，在*FindAddress(int (*ptrScore)[4],int *n*)中，执行语句 ptr=*(ptrScore+*n*)，返回行指针 ptr，然后调用 Display(score,*n*,*p*)输出成绩。在 Display(int *a*[][4],int *n*,int *p*)中，通过 *p*+col 改变列地址，即找到该学生的每门课成绩的位置，依次输出每门课的成绩。

2. 函数指针

指针可以指向变量、数组，也可以指向函数，指向函数的指针就是函数指针。与数组名类似，函数名就是程序在内存中的起始地址。指向函数的指针可以把地址传递给函数，也可以从函数返回给指向函数的指针。

【例 2-8】利用函数指针作为函数参数，实现选择排序算法的升序排列和降序排列。

【分析】主要考查函数指针作为函数参数的使用。

```
#include<stdio.h>
#include<stdlib.h>
#define N 10
int Ascending(int a,int b);                    /*声明升序排列函数*/
int Descending(int a,int b);                   /*声明降序排列函数*/
void swap(int *,int *);                        /*声明交换数据函数*/
void SelectSort(int a[],int n,int (*compare)(int,int));/*选择排序,函数指针作为参数调用*/
void Display(int a[],int n);                   /*声明输出数组元素函数*/
void main()
{
    int a[N]={22,55,12,7,19,65,81,3,30,52};
    int flag;
    while(1)
    {
       printf("1:从小到大排序.\n2:从大到小排序.\n0:结束!\n");
       printf("请输入: ");
       scanf("%d",&flag);
       switch(flag)
       {
       case 1:
          printf("排序前的数据为:");
          Display(a,N);
          SelectSort(a,N,Ascending);           /*从小到大排序,将函数作为参数传递*/
          printf("从小到大排列后的元素序列为:");
          Display(a,N);
          break;
       case 2:
          printf("排序前的数据为:");
          Display(a,N);
          SelectSort(a,N,Descending);          /*从大到小排序,将函数作为参数传递*/
```

```
            printf("从大到小排列后的元素序列为:");
            Display(a,N);
            break;
        case 0:
            printf("程序结束!\n");
            break;
        default:
            printf("输入数据不合法,请重新输入.\n");
            break;
        }
    }
    system("pause");
}
/*选择排序,将函数作为参数传递,判断是从小到大还是从大到小排序*/
void SelectSort(int a[],int n,int(*compare)(int,int))
{
    int i,j,k;
    for(i=0;i<n;i++)
    {
        j=i;
        for(k=i+1;k<n;k++)
            if((*compare)(a[k],a[j]))          /*调用 compare 函数,比较 a[k]和 a[j]大小*/
                j=k;
        swap(&a[i],&a[j]);                     /*交换 a[i]和 a[j]*/
    }
}
/*交换数组的元素*/
void swap(int *a,int *b)
{
    int t;
    t=*a;
    *a=*b;
    *b=t;
}
/*判断相邻数据大小,如果前者大,升序排列需要交换*/
int Ascending(int a,int b)
{
    if(a>b)                                    /*若 a>b*/
        return 1;                              /*返回 1*/
    else                                       /*否则*/
        return 0;                              /*返回 0*/
}
/*判断相邻数据大小,如果前者小,降序排列需要交换*/
int Descending(int a,int b)
{
    if(a<b)                                    /*若 a<b*/
        return 1;                              /*返回 1*/
    else                                       /*否则*/
        return 0;                              /*返回 0*/
}
/*输出数组元素*/
void Display(int a[],int n)
{
    int i;
    for(i=0;i<n;i++)
        printf("%4d",a[i]);
    printf("\n");
}
```

程序运行结果如图 2-20 所示。

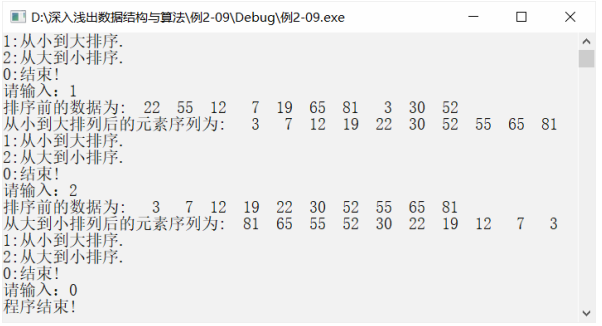


图 2-20　函数指针作为函数参数传递的排序运行结果

其中，函数 SelectSort(*a*,*N*,Ascending)中的参数 Asscending 是一个函数名，传递给函数定义 void SelectSort (int *a*[],int *n*,int(*compare)(int,int))中的函数指针 compare，这样指针就指向了 Asscending。从而可以在执行语句(*compare)(*a*[j],*a*[j+1])时调用函数 Ascending(int *a*,int *b*)判断是否需要交换数组中两个相邻的元素，然后调用 swap(&*a*[j],&*a*[j+1])进行交换。

注意：函数指针不能执行像 *f*+1、*f*++、*f*--等运算。

2.3　参数传递

在程序设计过程中，参数传递是经常会遇到的情况。在 C 语言中，函数的参数传递的方式通常有两种，一种是传值的方式，另一种是传地址的方式。

2.3.1　传值调用

在函数调用时，一般情况下，调用函数和被调用函数之间会有参数传递。调用函数后面括号里面的参数是实际参数，被调用函数中的参数是形式参数。传值调用是建立参数的一个副本并把值传递给形式参数，在被调用函数中修改形式参数的值，并不会影响到调用函数实际参数的值。

【例 2-9】编写一个函数，求两个整数的最大公约数。

【分析】通过传值调用的方式，把实际参数的值传递给形式参数，其实形式参数是实际参数的一个副本（拷贝）。

```
#include <stdio.h>                  /*包含输入输出函数*/
int GCD(int m,int n);               /*求两个整数的最大公约数的函数声明*/
void main()                         /*主函数*/
{
    int a,b,v;                      /*定义变量*/
    printf("请输入两个整数:");      /*输出提示信息*/
    scanf("%d,%d",&a,&b);           /*键盘输入两个数*/
    v=GCD(a,b);                     /*调用求两个数中的较大者的函数*/
    printf("%d 和%d 的最大公约数为:%d\n",a,b,v); /*输出提示信息*/
}
int GCD(int m,int n)
/*求两个整数的最大公约数,并返回公约数*/
{
    int r;                          /*定义变量*/
```

```
    r=m;                          /*将参数 m 赋值给 r*/
    do
    {
       m=n;                       /*赋值*/
       n=r;
       r=m%n;                     /*r 是 m 除以 n 的模*/
    }while(r);
    return n;                     /*返回最大公约数 n*/
}
```

程序的输出结果如图 2-21 所示。

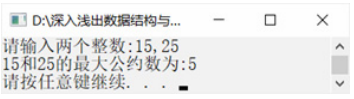


图 2-21　求两个整数的最大公约数运行结果

假设输入两个数 15 和 25，在主函数中，将 15 和 25 分别赋值给实际参数 *a* 和 *b*，通过语句 v=GCD(a,b) 调用实现函数 GCD(int *m*,int *n*)，也就是所谓的被调用函数，将 15 和 25 分别传递给被调用函数的形式参数 *m* 和 *n*。然后求 *m* 和 *n* 的最大公约数，通过语句 return *n*；将最大公约数 5 返回给主函数，即被调用函数，因此输出结果为 5。

上述函数参数传递属于参数的单向传递，即 *a* 和 *b* 可以把值分别传递给 *m* 和 *n*，而不可以把 *m* 和 *n* 传递给 *a* 和 *b*。在传值调用中，实际参数和形式参数分别占用不同的内存单元，形式参数是实际参数的一个副本，实际参数和形式参数的值的改变都不会相互受到影响，如图 2-22 所示。这就像有一张身份证原件，它的复印件就是个副本，复印件的丢失不会影响到身份证原件的存在，身份证原件的丢失也不会影响到复印件的存在。

在调用函数时，形式参数被分配存储单元，并把 15 和 25 传递给形式参数，在函数调用结束，形式参数被分配的存储单元被释放，形式参数不复存在，而主函数中的实际参数仍然存在，并且其值不会受到影响。在被调用函数中，如果改变形式参数的值，假设把 *m* 和 *n* 的值分别改变为 20 和 35，*a* 和 *b* 的值不会改变，如图 2-23 所示。

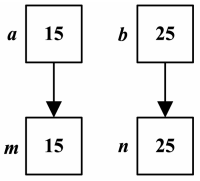


图 2-22　参数传递过程

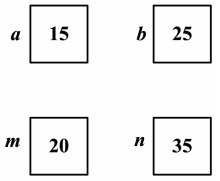


图 2-23　形式参数改变后的情况

2.3.2　传地址调用

C 语言通过指针（地址）实现传地址调用。在函数调用过程中，如果需要在被调用函数中修改参数值，则需要把实际参数的地址传递给形式参数，通过修改该地址的内容改变形式参数的值，以达到修改调用函数中实际参数的目的。

【例 2-10】编写一个求两个整数较大者和较小者的函数，要求用传地址方式实现。

【分析】通过传地址调用的方式，把两个实际参数传递给形式参数。在被调用函数中，先比较两个形式参数值的大小，如果前者小于后者，则交换两个参数值，其中，前者为大，后者为小。传地址调用时，在调用函数和被调用函数中，对参数的操作其实都是在对同一块内存操作，实际参数和形式参数共用同一块内存。

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

```
void Swap(int *x,int *y);  /*函数声明*/
void main()
{
    int a,b;
    printf("请输入两个整数: \n");
    scanf("%d,%d",&a,&b);
    if(a<b)
       Swap(&a,&b);     /*两个数中如果前者小,则交换两个值,使其较大的保存在 a 中,较小的保存在 b 中*/
    printf("在两个整数%d 和%d 中,较大者为:%d,较小者为:%d\n",a,b,a,b);
    system("pause");
}
void Swap(int *x,int *y)    /*交换函数实现,参数 x 和 y 分别指向实参中的 a 和 b/*
/*交换两个数,较大的保存在*x 中,较小的保存在*y 中*/
{
    int z;
    z=*x;                   /*交换 x 和 y 指向的值,实际上就是交换 a 和 b 的值*/
    *x=*y;
    *y=z;
}
```

程序的运行结果如图 2-24 所示。

在主函数中，如果 $a<b$，则调用 Swap(&*a*,&*b*)函数交换两个数。其中，实际参数是变量的地址，就是把地址传递给被调用函数 Swap(int **x*,int **y*)中的形式参数 *x* 和 *y*，*x* 和 *y* 是指针变量，即指针 *x* 和 *y* 指向变量 *a* 和 *b*。这样，交换**x* 和**y* 的值就是交换 *a* 和 *b* 的值。函数调用时，实际参数和形式参数的变化情况如图 2-25 所示。

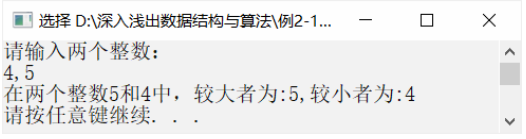
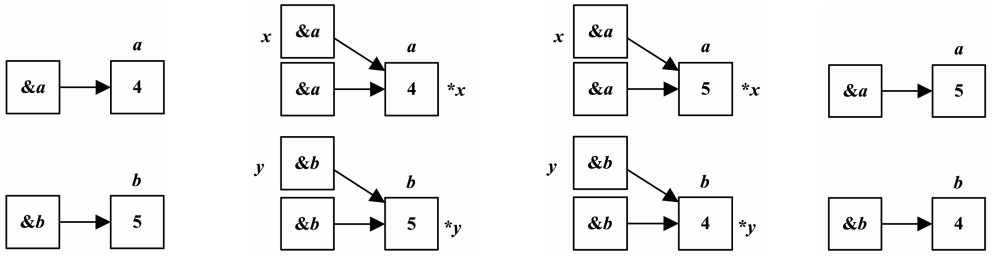


图 2-24　传地址方式求两个整数的较大者和较小者的程序运行结果



（a）调用函数前　（b）实际参数传递给形式参数后　（c）两个数交换后　（d）函数调用结束后

图 2-25　实际参数和形式参数的变化情况

如果要修改多个参数的值并返回给调用函数，该怎么呢？这就需要将数组名作为参数传递给被调用函数。数组名作为参数传递时，传递的是整个数组。数组名是数组的首地址，如果把数组名作为实际参数，在函数调用时，会把数组的首地址传递给形式参数。这样形式参数就可以根据数组的首地址访问整个数组并对其操作，这是因为整个数组元素的地址是连续的。

注意：在传值调用中，参数传递是单向传递，只能由实际参数传递给形式参数，而不能把形式参数反向传递给实际参数。而在传地址调用中，对形式参数的操作，即是对实际参数的操作，它们拥有同一块内存单元，属于双向传递。

2.4 结构体

结构体是自定义的数据类型，用于构造非数值数据类型，在处理实际问题中应用非常广泛。数据结构中的链表、队列、树、图等结构都需要用到结构体。

2.4.1 结构体的定义

一个教职工基本情况表包括工号、姓名、性别、职称、学位和联系电话等信息，每个数据信息的类型并不相同，使用前面学过的数据类型不能将这些信息有效组织起来。每一个教职工都包含工号、姓名、性别、职称、学位和联系电话等数据项，这些数据项放在一起构成的信息称为一个记录。例如，一个教师基本情况表如表 2-1 所示。

表 2-1 教师基本情况表

工　号	姓　名	性　别	职　称	学　位	联 系 电 话
2019011	刘　云	男	教授	硕士	88308523
2016035	吴　起	男	副教授	博士	88308233
2018020	赵小曼	女	副教授	硕士	88308758

要用 C 语言描述表中的某一条记录，需要定义一种特殊的类型，这种类型就是结构体类型。它的定义如下。

```
struct teacher          /*结构体类型*/
{
    int no;             /*工号*/
    char name[20];      /*姓名*/
    char sex[4];        /*性别*/
    char headship[8];   /*职称*/
    char degree[6];     /*学位*/
    char tel[15];       /*联系电话*/
};
```

其中，struct teacher 就是新的数据类型——结构体类型，no、name、sex、headship、degree 和 tel 为结构体类型的成员，表示记录中的数据项。这样，结构体类型 struct teacher 就可以完整地表示一个教师信息了。

定义结构体类型的一般格式如下。

```
struct 结构体名
{
    成员列表;
};
```

struct 与结构体名合在一起构成结构体类型，结构体名与变量名的命名规则一样。teacher 就是结构体名。使用一对花括号将成员列表括起来，在右花括号外使用一个分号作为定义结构体类型的结束。前面的 no、name、sex 等都是结构体类型的成员，每个成员需要说明其类型，就像定义变量一样。

struct teacher 是一个类型名，如果要定义一个结构体变量，可使用如下语句。

```
struct teacher t1;
```

t1 就是类型为结构体 struct teacher 类型的变量。如果给结构体变量 t1 的成员分别赋值，语句如下。

```
t1.no=19001;
strcpy(t1.name,"Zhu Tong");
strcpy(t1.sex,"m");
strcpy(t1.headship,"Professor");
strcpy(t1.degree,"doctor");
strcpy(t1.tel,"15639038813");
```

则 t1 的结构如图 2-26 所示。

结构体的变量定义也可以和定义结构体类型同时定义。例如：

```
struct teacher              /*结构体类型*/
{
   int no;                  /*工号*/
   char name[20];           /*姓名*/
   char sex[4];             /*性别*/
   char headship[8];        /*职称*/
   char degree[20];         /*学位*/
   char tel[15];            /*联系电话*/
}t1;
```

同样，也可以定义结构体数组类型。结构体变量的定义与初始化可以分开进行，也可以在结构体数组定义的时候初始化。例如：

```
struct teacher              /*结构体类型*/
{
   int no;                  /*工号*/
   char name[20];           /*姓名*/
   char sex[4];             /*性别*/
   char headship[8];        /*职称*/
   char degree[20];         /*学位*/
   char tel[15];            /*联系电话*/
}t1[2]={{19001,"Zhu Tong","m","教授","博士","88301234"},
   {19002,"Guo Jing","f","讲师","硕士","88125630"}};
```

数组中各个元素在内存中的情况如图 2-27 所示。

19001	Zhu Tong	m	Professor	doctor	15639038813

图 2-26　t1 的结构

t1[0]	19001
	Zhu Tong
	m
	教授
	博士
	88301234
t2[1]	19002
	Guo Jing
	f
	讲师
	硕士
	88125630
	…

图 2-27　结构体数组 t1 在内存中的结构

2.4.2 指向结构体的指针

指针可以指向整型、浮点型、字符等基本类型变量，同样也可以指向结构体变量。指向结构体变量的指针的值是结构体变量的起始地址。指针可以指向结构体，也可以指向结构体数组。

【例 2-11】利用指向结构体数组的指针输出学生基本信息。

【说明】指向结构体的指针与指向数组的指针一样，结构体中的成员变量地址是连续的，将指针指向结构体数组，就可直接访问结构体中的所有成员。程序实现如下。

```
#include <stdio.h>
   #include<stdlib.h>
#define N 10
/*结构体类型及变量定义、初始化*/
struct student
{
   char *no;
   char *name;
   char sex;
   int age;
   float score;
}stu[3]={{"19001","Zhu Tong",'m',22,90.0},
      {"19002","Li Hua",'f',21,82.0},
      {"19003","Yang Yang",'m',22,95.0}};
void main()
{
   struct student *p;                    /*定义结构体指针*/
   printf("学生基本情况表:\n");
   printf("学号         姓名       性别    年龄    成绩\n");/*输出表头*/
   for(p=stu;p<stu+3;p++)                /*通过指向结构体的指针输出学生信息*/
         printf("%-8s%12s%8c%8d%8.1f\n",p->no,p->name,p->sex,p->age,p->score);
   system("pause");
}
```

程序运行结果如图 2-28 所示。

首先定义了一个指向结构体的指针变量 *p*，在循环体中，指针指向结构体数组 *p*=stu，即指针指向了结构体变量的起始地址。通过 *p*->no、*p*->name 等访问各个成员。如果 *p*+1，表示数组中第 2 个元素 stu[1]的起始地址，*p*+2 表示数组中的第 3 个元素地址，如图 2-29 所示。

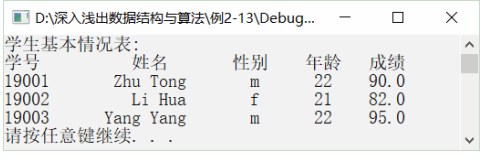


图 2-28 通过结构体指针输出学生信息

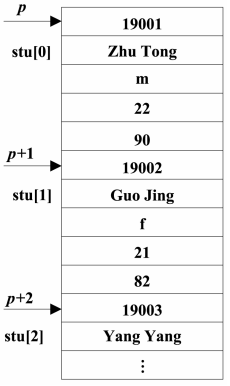


图 2-29 指向结构体数组的指针在内存的情况

2.4.3　用 typedef 定义数据类型

通常情况下，在定义结构体类型时，使用关键字 typedef 为新的数据类型起一个好记的名字。typedef 是 C 语言中的关键字，它的主要作用是为类型重新命名，一般形式如下。

```
typedef 类型名 1 类型名 2
```

其中，类型名 1 是已经存在的类型，如 int、float、char、long 等；也可以是结构体类型，如 struct student。类型名 2 是重新起的名字，命名规则与变量名的命名规则类似，必须是一个合法的标识符。

1. 使用 typedef 为基本数据类型重新命名

例如：

```
typedef int COUNT;              /*将 int 型重新命名为 COUNT*/
typedef float SCORE;            /*将 float 型重新命名为 SCORE*/
```

经过以上重新定义变量，COUNT 就代表了 int，SCORE 就表示了 float。这样，如下两条语句等价。

```
int a,b,c;                      /*定义 int 型变量 a、b、c*/
COUNT a,b,c;                    /*定义 COUNT 型变量 a、b、c*/
```

2. 使用 typedef 为数组类型重新命名

例如，以下代码是将 NUM 定义为数组类型：

```
typedef int NUM[20];            /*NUM 被定义为新的数组类型*/
```

NUM 被定义为数组类型，该数组的长度为 20，类型为 int。可以使用 NUM 定义 int 型数组，代码如下。

```
NUM a;                          /*使用 NUM 定义 int 型数组*/
```

a 表示长度为 20 的 int 型数组，它与如下代码等价。

```
int a[20];                      /*使用 int 定义数组*/
```

3. 使用 typedef 为指针类型重新命名

使用 typedef 为指针类型变量重新命名与重新命名数组类型的方法是类似的。例如：

```
typedef float *POINTER;         /*POINT 被定义为指针类型*/
```

POINTER 表示指向 float 类型的指针类型。如果要定义一个 float 类型的指针变量 *p*，代码如下。

```
POINTER p;                      /*使用 POINTER 定义指针变量*/
```

p 被定义为指向 float 类型的指针变量。同样，也可以使用 typedef 重新为指向函数的指针类型命名，例如，定义一个函数指针类型，代码如下。

```
typedef int (*PTR)(int,int);    /*PTR 被定义为函数指针类型*/
```

PTR 被定义为函数指针类型，PTR 是指向返回值为 int 且有两个 int 型参数的函数指针。以下语句使用 PTR 定义变量。

```
PTR pm;                         /*使用 PTR 定义一个函数指针变量 pm*/
```

pm 被定义为一个函数指针变量。

4. 使用 typedef 为用户自定义数据类型重新命名

用户自己定义的数据类型主要包括结构体、联合体、枚举类型，最为常用的是为结构体类型重新命名，联合体和枚举类型的命名方法与结构体的重新命名方法类似。例如，将一个结构体命名为 DATE，代码如下。

```
typedef struct          /*为结构体类型重新命名*/
{
    int year;           /*年*/
    int month;          /*月*/
    int day;            /*日*/
}DATE;
```

从类型名 DATE 可以很容易看出，DATE 是表示日期的类型。上面的类型重新定义是在定义结构体类型的同时为结构体命名；也可以先定义结构体类型，然后重新为结构体命名，代码如下。

```
struct date             /*定义结构体类型*/
{
    int year;
    int month;
    int day;
};
typedef date DATE;      /*为结构体类型重新命名*/
```

以上两段代码是等价的。注意，date 和 DATE 是两个不同的名字，C 语言是区分大小写的。接下来，就可以使用 DATE 定义变量了，代码如下。

```
DATE d;                 /*定义变量 d*/
```

上面的变量定义与如下变量定义等价。

```
struct date d;
```

2.5 小结

本章主要介绍了 C 语言的重点和难点部分，目的是为今后学习数据结构扫清障碍。首先围绕着 C 语言中的重点和难点——递归、指针、参数传递、结构体，结合典型案例进行了详细分析、讲解。

递归是 C 语言及算法设计中常常使用的技术，递归可以把复杂的问题变成与原问题类似且规模小的问题加以解决，使用递归使程序的结构很清晰，更具有层次性，写出的程序简洁易懂。使用递归只需要少量的程序就可以描述解决问题需要的重复计算过程，大大减少了程序的代码量。任何使用递归解决的问题都能使用迭代的方法解决。

指针是 C 语言的精髓所在。指针不仅可以与变量结合起来使用，还可以与数组、函数相结合，使用指针能很方便地操作字符串、动态分配内存。指针使用不当，也常常会出现一些致命错误，这种错误十分隐蔽，难以发现，这就需要读者能熟练使用指针操作，以避免或减少错误的发生，并能掌握程序调试技巧，以快速找出原因并解决问题。

在 C 语言中，函数的参数传递有两种：传值调用和传地址调用。其中，前者是一种单向值传递方式，实际参数和形式参数分别占用不同的内存空间。后者是一种双向的值传递方式，实际参数和形式参数占用同一块内存单元。

结构体属于用户自己定义的类型，它常常用于非数值程序设计中，特别是在今后学习数据结构的过程中，链表、栈、队列、树及图等都会用到结构体类型。

第二篇
线性数据结构

第 3 章
线性表

线性表是一种最简单的线性结构。线性结构的特点是在非空的有限集合中存在唯一的一个被称为“第一个”的数据元素，存在唯一的一个被称为“最后一个”的数据元素。第一个元素没有直接前驱元素，最后一个元素没有直接后继元素，其他元素都有唯一的前驱元素和唯一的后继元素。线性表有两种存储结构，即顺序存储结构和链式存储结构。

本章重点和难点：

- 顺序表的基本操作实现。
- 单链表与双向链表的存储表示与基本操作实现。

3.1 线性表的定义及抽象数据类型

线性表（Linear_List）是最简单且最常用的一种线性结构。

3.1.1 线性表的逻辑结构

线性表是由 n 个类型相同的数据元素组成的有限序列，记为（a_1，a_2，…，a_{i-1}，a_i，a_{i+1}，…，a_n）。其中，这里的数据元素可以是原子类型，也可以是结构类型。线性表的数据元素存在着序偶关系，即数据元素之间具有一定的次序。在线性表中，数据元素 a_{i-1} 在 a_i 的前面，a_i 又在 a_{i+1} 的前面，可以把 a_{i-1} 称为 a_i 的直接前驱元素，a_i 称为 a_{i+1} 的直接前驱元素。a_i 称为 a_{i-1} 的直接后继元素，a_{i+1} 称为 a_i 的直接后继元素。

线性表的逻辑结构如图 3-1 所示。

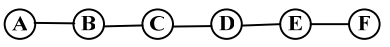


图 3-1　线性表的逻辑结构

英文单词就可看作是简单的线性表，例如 China、Science、Structure。其中每一个英文字母就是一个数据元素，每个数据元素之间存在着唯一的顺序关系。如“China”中字母 C 后面是字母 h，字母 h 后面是字母 i。

在较为复杂的线性表中，一个数据元素可以由若干个数据项组成，在如表 3-1 所示的一所学校的教职工情况表中，一个数据元素由姓名、性别、出生年月、籍贯、学历、职称及任职时间 7 个数据项组成。数据元素也称为记录。

知识点：在线性表中，除了第一个元素 a_1，每个元素有且仅有一个直接前驱元素；除了最后一个元素 a_n，每个元素有且只有一个直接后继元素。

表 3-1　教职工情况表

姓　名	性　别	出生年月	籍　贯	学　历	职　称	任职时间
王　欢	女	1958 年 10 月	河南	本科	教授	2000 年 10 月
周启泰	男	1969 年 5 月	陕西	研究生	副教授	2002 年 10 月
刘　娜	女	1978 年 12 月	四川	研究生	讲师	2006 年 11 月
⋮	⋮	⋮	⋮	⋮	⋮	⋮

3.1.2　线性表的抽象数据类型

线性表的抽象数据类型包括数据对象集合和基本操作集合。

1. 数据对象集合

线性表的数据对象集合为{a_1，a_2，…，a_n}，元素类型为 DataType。

数据元素之间的关系是一对一的关系。除了第一个元素 a_1 外，每个元素有且只有一个直接前驱元素，除了最后一个元素 a_n 外，每个元素有且只有一个直接后继元素。

2. 基本操作集合

（1）InitList(&*L*)：初始化操作，建立一个空的线性表 *L*。这就像是在日常生活中，一所院校为了方便管理，建立一个教职工基本情况表，准备登记教职工信息。

（2）ListEmpty(*L*)：若线性表 *L* 为空，返回 1，否则返回 0。这就像是刚刚建立了教职工基本情况表，还没有登记教职工信息。

（3）GetElem(*L*,*i*,&*e*)：返回线性表 *L* 的第 *i* 个位置元素值给 *e*。这就像在教职工基本情况表中，根据给定序号查找某个教师信息。

（4）LocateElem(*L*,*e*)：在线性表 *L* 中查找与给定值 *e* 相等的元素，如果查找成功返回该元素在表中的序号表示成功，否则返回 0 表示失败。这就像在教职工基本情况表中，根据给定的姓名查找教师信息。

（5）InsertList(&*L*,*i*,*e*)：在线性表 *L* 中的第 *i* 个位置插入新元素 *e*。这就类似于经过招聘考试，引进了一名教师，这个教师信息登记到教职工基本情况表中。

（6）DeleteList(&*L*,*i*,&*e*)：删除线性表 *L* 中的第 *i* 个位置元素，并用 *e* 返回其值。这就像某个教职工到了退休年龄或者调入其他学校，需要将该教职工从教职工基本情况表中删除。

（7）ListLength(*L*)：返回线性表 *L* 的元素个数。这就像查看教职工基本情况表中有多少个教职工。

（8）ClearList(&*L*)：将线性表 *L* 清空。这就像学校被撤销，不需要再保留教职工基本信息，将这些教职工信息全部清空。

3.2　线性表的顺序表示与实现

在了解了线性表的基本概念和逻辑结构之后，接下来就需要将线性表的逻辑结构转换为计算机能识别的存储结构，以便实现线性表的操作。线性表的存储结构主要有顺序存储结构和链式存储结构两种。本节主要介绍线性表的顺序存储结构及操作实现。

3.2.1 线性表的顺序存储结构

线性表的顺序存储指的是将线性表中的各个元素依次存放在一组地址连续的存储单元中。

假设线性表的每个元素需占用 m 个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置。则线性表中第 i+1 个元素的存储位置 LOC(a_{i+1})和第 i 个元素的存储位置 LOC(a_i)之间满足关系 LOC(a_{i+1})=LOC(a_i)+m。

线性表中第 i 个元素的存储位置与第一个元素 a_1 的存储位置满足以下关系。

LOC(a_i)=LOC(a_1)+(i−1)×m

其中，第一个元素的位置 LOC(a_1)称为起始地址或基地址。

线性表的这种机内表示称为线性表的顺序存储结构或顺序映像，通常将这种方法存储的线性表称为顺序表。顺序表逻辑上相邻的元素在物理上也是相邻的。每一个数据元素的存储位置都和线性表的起始位置相差一个和数据元素在线性表中的位序成正比的常数（见图 3-2）。只要确定了第一个元素的起始位置，线性表中的任一元素都可以随机存取，因此，线性表的顺序存储结构是一种随机存取的存储结构。

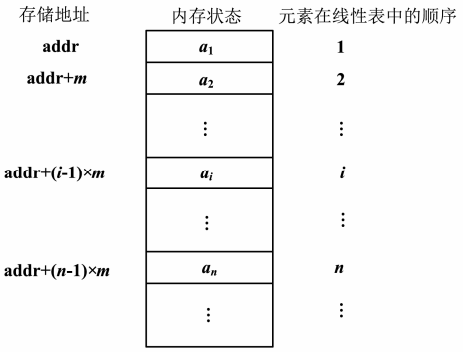


图 3-2 线性表存储结构

由于 C 语言的数组具有随机存取特点，因此可采用数组来描述顺序表。顺序表的存储结构描述如下。

```
#define LISTSIZE 100                    /*宏定义 LISTSIZE 表示 100*/
typedef struct                          /*定义结构体 SeqList*/
{
   DataType list[LISTSIZE];             /*定义线性表*/
   int length;                          /*定义变量 length*/
}SeqList;
```

其中，DataType 表示数据元素类型，list 用于存储线性表中的数据元素，length 用来表示线性表中数据元素的个数，SeqList 是结构体类型名。

如果要定义一个顺序表，代码如下。

```
SeqList L;
```

如果要定义一个指向顺序表的指针，代码如下。

```
SeqList *L;
```

3.2.2　顺序表的基本运算

在顺序存储结构中，线性表的基本运算如下（以下算法的实现保存在文件 SeqList.h 中）。

（1）初始化线性表。

```
void InitList(SeqList *L)
/*初始化线性表*/
{
    L->length=0;                    /*把线性表的长度置为 0*/
}
```

（2）判断线性表是否为空。

```
int ListEmpty(SeqList L)
/*判断线性表是否为空,线性表为空返回 1,否则返回 0*/
{
   if(L.length==0)                  /*若线性表的长度为 0*/
      return 1;                     /*返回 1*/
   else                             /*否则*/
      return 0;                     /*返回 0*/
}
```

（3）按序号查找。先判断序号是否合法，如果合法，把对应位置的元素赋给 e，并返回 1 表示查找成功；否则返回-1 表示查找失败。按序号查找的算法实现如下。

```
int GetElem(SeqList L,int i,DataType *e)
/*查找线性表中第 i 个元素。查找成功将该值返回给 e,并返回 1 表示成功；否则返回-1 表示失败*/
{
  if(i<1||i>L.length)               /*在查找第 i 个元素之前,判断该序号是否合法*/
      return -1;                    /*返回-1*/
   *e=L.list[i-1];                  /*将第 i 个元素的值赋值给 e*/
   return 1;                        /*返回 1*/
}
```

（4）按内容查找。从线性表中的第一个元素开始，依次与 e 比较，如果相等，返回该序号表示成功；否则返回 0 表示查找失败。按内容查找的算法实现如下。

```
int LocateElem(SeqList L,DataType e)
/*查找线性表中元素值为 e 的元素*/
{
   int i;
   for(i=0;i<L.length;i++)          /*从第一个元素开始与 e 进行比较*/
      if(L.list[i]==e)              /*若存在与 e 值相等的元素*/
           return i+1;              /*则返回该元素在线性表中的序号*/
   return 0;                        /*否则,返回 0*/
}
```

（5）插入操作。插入操作就是在线性表 L 中的第 i 个位置插入新元素 e，使线性表$\{a_1, a_2, \cdots, a_{i-1}, a_i, \cdots, a_n\}$变为$\{a_1, a_2, \cdots, a_{i-1}, e, a_i, \cdots, a_n\}$，线性表的长度也由 n 变成 n+1。

要在顺序表中的第 i 个位置上插入元素 e，首先将第 i 个位置以后的元素依次向后移动 1 个位置，然后把元素 e 插入到第 i 个位置。移动元素时要从后往前移动元素，先移动最后 1 个元素，再移动倒数第 2 个元素，以此类推。

例如，在线性表{9, 12, 6, 15, 20, 10, 4, 22}中，要在第 5 个元素之前插入 1 个元素 28，需要将序号为 8、7、6、5 的元素依次向后移动 1 个位置，然后在第 5 号位置插入元素 28，这样，线性表就变成了{9, 12, 6, 15, 28, 20, 10, 4, 22}，如图 3-3 所示。

图 3-3　在顺序表中插入元素 28 的过程

插入元素之前要判断插入的位置是否合法，顺序表是否已满，在插入元素后要将表长增加 1。插入元素的算法实现如下。

```
int InsertList(SeqList *L,int i,DataType e) /*在顺序表中第 i 个位置插入元素的算法实现*/
/*在顺序表的第 i 个位置插入元素 e,插入成功返回 1,如果插入位置不合法返回-1,顺序表满返回 0*/
{
    int j;
    if(i<1||i>L->length+1)                  /*在插入元素前,判断插入位置是否合法*/
    {
        printf("插入位置 i 不合法! \n");     /*输出错误提示信息*/
        return -1;                          /*返回-1*/
    }
    else if(L->length>=LISTSIZE)            /*在插入元素前,判断顺序表是否已经满*/
    {
        printf("顺序表已满,不能插入元素.\n"); /*输出错误提示信息*/
        return 0;                           /*返回 0*/
    }
    else
    {
        for(j=L->length;j>=i;j--)           /*将第 i 个位置以后的元素依次后移*/
            L->list[j]=L->list[j-1];
        L->list[i-1]=e;                     /*插入元素到第 i 个位置*/
        L->length=L->length+1;              /*将顺序表长增 1*/
        return 1;                           /*返回 1*/
    }
}
```

插入元素的位置 i 的合法范围应该是 $1 \leqslant i \leqslant L \rightarrow \text{length}+1$。当 $i=1$ 时，插入位置是在第一个元素之前，对应 C 语言数组中的第 0 个元素；当 $i=L \rightarrow \text{length}+1$ 时，插入位置是最后一个元素之后，对应 C 语言数组中的最后一个元素之后的位置。当插入位置是 $i=L \rightarrow \text{length}+1$ 时，不需要移动元素；当插入位置是 $i=0$ 时，则需要移动所有元素。

（6）删除第 i 个元素。删除第 i 个元素之后，线性表$\{a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n\}$变为$\{a_1, a_2, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n\}$，线性表的长度由 n 变成 $n-1$。

为了删除第 i 个元素，需要将第 $i+1$ 后面的元素依次向前移动一个位置，将前面的元素覆盖。移动元素时要先将第 $i+1$ 个元素移动到第 i 个位置，再将第 $i+2$ 个元素移动到第 $i+1$ 个位置，以此类推，直到最后一个元素移动到倒数第二个位置。最后将顺序表的长度减 1。

例如，要删除线性表{9, 12, 6, 15, 28, 20, 10, 4, 22}的第 4 个元素，需要依次将序号为 5、6、7、8、9 的元素向前移动一个位置，并将表长减 1，如图 3-4 所示。

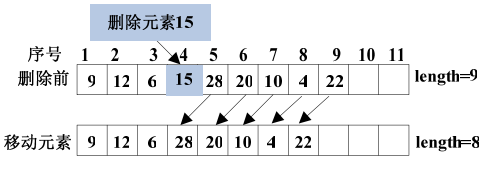


图 3-4　删除元素 15 的过程

在进行删除操作时，先判断顺序表是否为空，若不空，接着判断序号是否合法，若不空且合法，则将要删除的元素赋给 e，并把该元素删除，将表长减 1。删除第 i 个元素的算法实现如下。

```
int DeleteList(SeqList *L,int i,DataType *e)      /*删除第 i 个元素的算法实现*/
{
    int j;
    if(L->length<=0)                              /*若顺序表的长度小于或等于 0*/
    {
        printf("顺序表已空不能进行删除!\n");          /*表示不能进行删除操作,输出提示信息*/
        return 0;                                 /*返回 0*/
    }
    else if(i<1||i>L->length)                     /*若删除位置不合法*/
    {
        printf("删除位置不合适!\n");                 /*则输出提示信息*/
        return -1;                                /*返回-1*/
    }
    else                                          /*否则*/
    {
        *e=L->list[i-1];                          /*将要删除的元素赋给 e*/
        for(j=i;j<=L->length-1;j++)
            L->list[j-1]=L->list[j];
        L->length=L->length-1;                    /*将表长减 1*/
        return 1;                                 /*返回 1*/
    }
}
```

删除元素的位置 i 的合法范围应该是 $1 \leqslant i \leqslant L \rightarrow$length。当 i=1 时，表示要删除第一个元素，对应 C 语言数组中的第 0 个元素；当 $i=L \rightarrow$length 时，要删除的是最后一个元素。

（7）求线性表的长度，代码如下。

```
int ListLength(SeqList L)                         /*求线性表的长度实现函数*/
{
    return L.length;                              /*返回线性表的长度*/
}
```

（8）清空顺序表，代码如下。

```
void ClearList(SeqList *L)                        /*清空顺序表实现函数*/
{
    L->length=0;                                  /*清空顺序表*/
}
```

3.2.3　顺序表的实现算法分析

在顺序表的实现算法中，除了按内容查找、插入和删除操作外，算法的时间复杂度均为 O（1）。

在按内容查找的算法中，若要查找的是第一个元素，则仅需要进行一次比较；若要查找的是最后一个元素，则需要比较 n 次才能找到该元素（设线性表的长度为 n）。

设 p_i 表示在第 i 个位置上找到与 e 相等的元素的概率，若在任何位置上找到元素的概率相等，即 $p_i=1/n$。则查找元素需要的平均比较次数为：

$$E_{\text{loc}}=\sum_{i=1}^{n} p_i \times i=\frac{1}{n}\sum_{i=1}^{n} i=\frac{n+1}{2}$$

因此，按内容查找的平均时间复杂度为 O（n）。

在顺序表中插入元素时，主要时间耗费在元素的移动上。如果要将元素插入到第一个位置，则需要移动元素的次数为 n 次；如果要在最后一个元素之前插入，则仅需把最后一个元素向后移动即可；如果要在最后一个元素之后插入，即第 n+1 个位置，则不需要移动元素。设 p_i 表示在第 i 个位置上插入元素的概率，假设在任何位置上找到元素的概率相等，即 p_i=1/（n+1）。则在顺序表的第 i 个位置插入元素时，需要移动元素的平均次数为：

$$E_{\text{ins}}=\sum_{i=1}^{n+1}p_i\times(n-i+1)=\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2}$$

因此，插入操作的平均时间复杂度为 O（n）。

在顺序表的删除算法中，时间主要耗费仍在元素的移动上。如果要删除的是第一个元素，则需要移动元素次数为 n−1 次；如果要删除的是最后一个元素，则需要移动 0 次。设 p_i 表示删除第 i 个位置上的元素的概率，假设在任何位置上找到元素的概率相等，即 p_i=1/n。则在顺序表中删除第 i 个元素时，需要移动元素的平均次数为：

$$E_{\text{del}}=\sum_{i=1}^{n}p_i\times(n-i)=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$$

因此，删除操作的平均时间复杂度为 O（n）。

3.2.4 顺序表的优缺点

线性表的顺序存储结构的优缺点如下。

1. 优点

（1）无须为表示表中元素之间的关系而增加额外的存储空间。

（2）可以快速地存取表中任一位置的元素。

2. 缺点

（1）插入和删除操作需要移动大量的元素。

（2）使用前须事先分配好存储空间，当线性表长度变化较大时，难以确定存储空间的容量。分配空间过大会造成存储空间的巨大浪费；分配的空间过小，难以适应问题的需要。

3.2.5 顺序表应用举例

在掌握了顺序表的基本操作之后，通过几个具体实例来加强对顺序表知识点的掌握。

【例 3-1】假设线性表 LA 和 LB 分别表示两个集合 A 和 B，利用线性表的基本运算实现集合运算：A=A-B，即如果在顺序表 LA 中出现的元素，在顺序表 LB 中也出现，则删除 A 中该元素。

【分析】只有依次从线性表 LB 中取出每个数据元素，并依次在线性表 LA 中查找该元素，如果 LA 中也存在该元素，则将该元素从 LA 中删除。其实这是求两个表的差集，即 A-B。依次检查顺序表 LB 中的每一元素，如果在顺序表 LA 中也出现，则在 A 中删除该元素。核心代码如下。

```
void DelElem(SeqList *LA,SeqList LB)
/*从 LA 中删除 LB 也出现的元素*/
{
  int i,flag,pos;
```

```
    DataType e;
    for(i=0;i<=LB.length;i++)
    {
        flag=GetElem(LB,i,&e);              /*依次把 LB 中每个元素取出给 e*/
        if(flag==1)
        {
          pos=LocateElem(*LA,e);            /*在 LA 中查找和 LB 中取出的元素 e 相等的元素*/
          if(pos>0)
             DeleteList(LA,pos,&e);         /*如果找到该元素,将其从 LA 中删除*/
        }
     }
}
```

程序运行结果如图 3-5 所示。

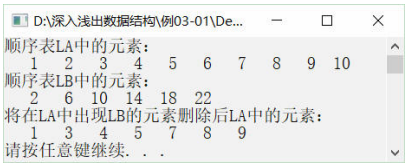


图 3-5　集合 *A*-*B* 运算的程序运行结果

说明：在设计程序时需要用到头文件“SeqList.h”，而在顺序表的类型定义中包含 DataType 数据类型和顺序表长度，所以在包含#include"SeqList.h"之前首先进行宏定义。宏定义、类型定义和包含文件语句的次序如下。

```
#define LISTSIZE 100
typedef int DataType;
#include"SeqList.h"
```

【例 3-2】编写一个算法，把一个顺序表分拆成两个部分，使顺序表中小于或等于 0 的元素位于左端，大于 0 的元素位于右端。要求不占用额外的存储空间。例如，顺序表（−21,8,−9,25,−31,3,−2, −36）经过分拆调整后变为（−21,−36,−9,−2,−31,3,25,8）。

【分析】设置两个指示器 *i* 和 *j*，分别扫描顺序表中的元素，*i* 和 *j* 分别从顺序表的左端和右端开始扫描。如果 *i* 遇到小于或等于 0 的元素，略过不处理，继续向前扫描；如果遇到大于 0 的元素，暂停扫描。如果 *j* 遇到大于 0 的元素，略过不处理，继续向前扫描；如果遇到小于或等于 0 的元素，暂停扫描。如果 *i* 和 *j* 都停下来，则交换 *i* 和 *j* 指向的元素。重复执行直到 $i \geqslant j$ 为止。

算法描述如下。

```
void SplitSeqList(SeqList *L)
/*将顺序表 L 分成两个部分: 左边是小于或等于 0 的元素,右边是大于 0 的元素*/
{
    int i,j;                                /*定义两个指示器 i 和 j*/
    DataType e;
    i=0,j=(*L).length-1;                    /*指示器 i 和 j 分别指示顺序表的左端和右端元素*/
    while(i<j)                              /*若未扫描完毕所有元素*/
    {
       while(L->list[i]<=0)                 /*i 遇到小于或等于 0 的元素*/
          i++;                              /*略过*/
       while(L->list[j]>0)                  /*j 遇到大于 0 的元素*/
          j--;                              /*略过*/
       if(i<j)                              /*交换 i 和 j 指向的元素*/
       {
          e=L->list[i];
          L->list[i]=L->list[j];
          L->list[j]=e;
       }
```

```
        }
    }
```

程序运行结果如图 3-6 所示。

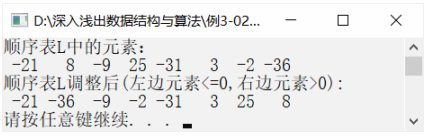


图 3-6　程序运行结果

【考研真题】设将 n（n>1）个整数存放到一维数组 R 中，试设计一个在时间和空间两方面都尽可能高效的算法，将 R 中保存的序列循环左移 p（0<p<n）个位置，即把 R 中的数据序列由$(x_0,x_1,\cdots,x_{n-1})$变换为$(x_p,x_{p+1},\cdots,x_{n-1},x_0,x_1,\cdots,x_{p-1})$。要求如下。

（1）给出算法的基本设计思想。

（2）根据设计思想，采用 C 语言描述算法。

（3）说明所设计算法的时间复杂度和空间复杂度。

【分析】该题目主要考查对顺序表的掌握情况，具有一定的灵活性。

（1）先将这 n 个元素序列$(x_0,x_1,\cdots,x_p,x_{p+1},\cdots,x_{n-1})$就地逆置，得到$(x_{n-1},x_{n-2},\cdots,x_p,x_{p-1},\cdots,x_0)$，然后再将前 $n-p$ 个元素$(x_{n-1},x_{n-2},\cdots,x_p)$和后 p 个元素$(x_{p-1},x_{p-2},\cdots,x_0)$分别就地逆置，得到最终结果$(x_p,x_{p+1},\cdots,x_{n-1},x_0,x_1,\cdots,x_{p-1})$。

（2）算法实现，可用 Reverse 和 LeftShift 两个函数实现。

```
void Reverse(int R[],int left,int right)     /*将 n 个元素序列逆置的算法实现*/
{
    int k=left,j=right,t;                    /*定义变量*/
    while(k<j)                               /*若未完成逆置*/
    {
        t=R[k];                              /*将这 n 个元素序列逆置处理过程*/
        R[k]=R[j];
        R[j]=t;
        k++;
        j--;
    }
}
void LeftShift(int R[],int n,int p) /*将 R 中保存的序列循环左移 p（0<p<n）个位置的算法实现*
{
    If(p>0 && p<n)                       /*若循环左移的位置合法*/
    {
        Reverse(R,0,n-1);                /*将全部元素逆置*/
        Reverse(R,0,n-p-1);              /*逆置前 n-p 个元素*/
        Reverse(R,n-p,n-1);              /*逆置后 n 个元素*/
    }
}
```

（3）上述算法的时间复杂度为 O（n），空间复杂度为 O（1）。

3.3　线性表的链式表示与实现

在解决实际问题时，有时并不适合采用线性表的顺序存储结构，例如，两个一元多项式相加、相乘，这就需要采用线性表另一种存储结构——链式存储。

3.3.1　单链表的存储结构

线性表的链式存储是采用一组任意的存储单元存放线性表的元素。这组存储单元可以是连续的，也可以是不连续的。因此，为了表示每个元素 a_i 与其直接后继元素 a_{i+1} 的逻辑关系，除了存储元素本身的信息外，还需要存储一个指示其直接后继元素的信息（即直接后继元素的地址）。这两部分构成的存储结构称为结点（node）。结点包括数据域和指针域两个域，数据域存放数据元素的信息，指针域存放元素的直接后继元素的存储地址。指针域中存储的信息称为指针。结点结构如图 3-7 所示。

通过指针域将线性表中 n 个结点元素按照逻辑顺序链在一起就构成了链表，如图 3-8 所示。由于链表中每个结点只有一个指针域，所以将这样的链表称为线性链表或者单链表。

图 3-7　结点结构

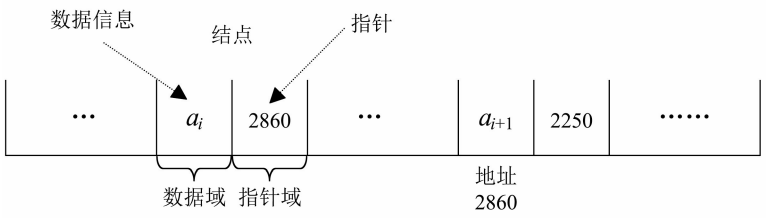


图 3-8　单链表

例如，线性表（Yang,Zheng,Feng,Xu,Wu,Wang,Geng）采用链式存储结构，链表的存取必须从头指针 head 开始，头指针指向链表的第一个结点，从头指针可以找到链表中的每一个元素。线性表的链式存储结构如图 3-9 所示。

如图 3-10 所示的通过结点的指针域表示线性表中的前后逻辑关系，叫作链式存储。链式存储结构中逻辑上相邻的元素，在物理位置上不一定相邻。

头指针 head：25

存储地址	数据域	指针域
6	Xu	36
19	Feng	6
25	Yang	51
36	Wu	47
43	Geng	NULL
47	Wang	43
51	Zheng	19

图 3-9　线性表的链式存储结构

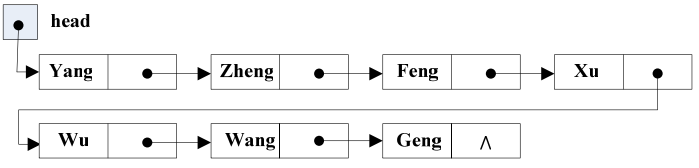


图 3-10　单链表的逻辑状态

一般情况下，只关心链表的逻辑顺序，而不关心链表的物理位置。通常把链表表示成通过箭头链接起来的序列，箭头表示指针域中的指针。如图 3-9 所示的线性表可以表示成如图 3-10 的序列。

为了操作上的方便，在单链表的第一个结点之前增加一个结点，称为头结点。头结点的数据域可以存放如线性表的长度等信息，头结点的指针域存放第一个结点的地址信息，指向第一个结点。此时的头指针 head 就指向了头结点，不再指向链表的第一个结点。带头结点的单链表如图 3-11 所示。

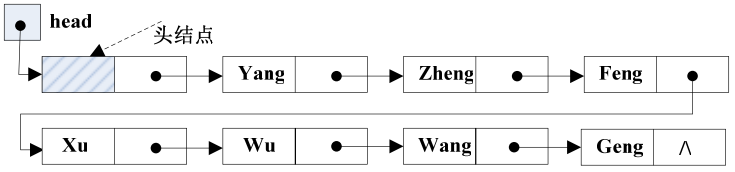


图 3-11　带头结点的单链表的逻辑状态

若带头结点的链表为空链表，则头结点的指针域为“空”，如图 3-12 所示。

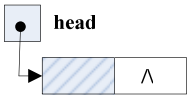


图 3-12　带头结点的单链表

单链表的存储结构用 C 语言描述如下。

```
typedef struct Node
{
   DataType data;
   struct Node *next;
}ListNode,*LinkList;
```

其中，ListNode 是链表的结点类型，LinkList 是指向链表结点的指针类型。假设有如下定义。

```
LinkList L;                                  /*定义 LinkList 类型变量*/
```

则 *L* 被定义为指向单链表的指针类型，相当于如下定义。

```
ListNode *L;                                 /*定义指向单链表的指针类型*/
```

注意：初学者需要区分头指针和头结点的区别。头指针是指向链表第一个结点的指针，若链表有头结点，则是指向头结点的指针。头指针链表的必要元素具有标识作用，所以常用头指针冠以链表的名字。头结点是为了操作的统一和方便而设立的，放在第一个元素结点之前，不是链表的必要元素。有了头结点，对在第一个元素结点前插入结点和删除第一个结点，其操作与其他结点的操作就统一了。

3.3.2　单链表上的基本运算

单链表上的基本运算有链表的创建、单链表的插入、单链表的删除、求单链表的长度等，以下是带头结点的单链表的基本运算的具体实现（保存在文件 LinkList.h 中）。

（1）初始化单链表。

```
void InitList(LinkList *head)
/*初始化单链表*/
{
    if((*head=(LinkList)malloc(sizeof(ListNode)))==NULL)   /*为头结点分配一个存储空间*/
          exit(-1);
    (*head)->next=NULL;          /*将单链表的头结点指针域置为空*/
}
```

（2）判断单链表是否为空。若单链表为空，返回 1；否则返回 0。算法实现如下。

```
int ListEmpty(LinkList head)
/*判断单链表是否为空*/
{
    if(head->next==NULL)          /*如果单链表头结点的指针域为空*/
       return 1;                  /*返回 1*/
    else                          /*否则*/
       return 0;                  /*返回 0*/
}
```

（3）按序号查找操作。从单链表的头指针 head 出发，利用结点的指针域依次扫描链表的结点，并进行计数，直到计数为 i，就找到了第 i 个结点。如果查找成功，返回该结点的指针，否则返回 NULL 表示查找失败。按序号查找的算法实现如下。

```
ListNode *Get(LinkList head,int i)
/*按序号查找单链表中第 i 个结点。查找成功返回该结点的指针表示成功；否则返回 NULL 表示失败*/
{
     ListNode *p;                 /*定义指向单链表的指针*/
     int j;                       /*定义计数器*/
      if(ListEmpty(head))         /*如果链表为空*/
          return NULL;            /*返回 NULL*/
      if(i<1)                     /*如果序号不合法*/
          return NULL;            /*则返回 NULL*/
      j=0;                        /*将计数器初始化为 0*/
      p=head;                     /*head 指针赋值给 p*/
      while(p->next!=NULL&&j<i)   /*如果在遍历完链表前且还未找到第 i 个结点*/
      {
           p=p->next;             /*则令 p 指向下一个结点继续查找*/
           j++;
      }
      if(j==i)                    /*找到第 i 个结点*/
           return p;              /*返回指针 p*/
      else                        /*否则*/
           return NULL;           /*返回 NULL*/
}
```

查找元素时，要注意判断条件 p→next!=NULL，保证 p 的下一个结点不为空，如果没有这个条件，就无法保证执行循环体中的 p=p→next 语句。

（4）按内容查找，查找元素值为 e 的结点。从单链表中的头指针开始，依次与 e 比较，如果找到返回该元素结点的指针；否则返回 NULL。查找元素值为 e 的结点的算法实现如下。

```
ListNode *LocateElem(LinkList head,DataType e)
/*按内容查找单链表中元素值为 e 的元素,若查找成功则返回对应元素的结点指针,否则返回 NULL 表示失败*/
{
    ListNode *p;                  /*定义指向单链表的指针*/
    p=head->next;                 /*指针 p 指向第一个结点*/
    while(p)
    {
        if(p->data!=e)            /*没有找到与 e 相等的元素*/
           p=p->next;             /*继续找下一个元素*/
        else                      /*找到与 e 相等的元素*/
           break;                 /*退出循环*/
    }
    return p;                     /*返回元素值为 e 的结点指针*/
}
```

（5）定位操作。定位操作与按内容查找类似，只是返回的是该结点的序号。从单链表的头指针出发，依次访问每个结点，并将结点的值与 e 比较，如果相等，返回该序号表示成功；如

果没有与 *e* 值相等的元素，返回 0 表示失败。定位操作的算法实现如下。

```
int LocatePos(LinkList head,DataType e)
/*查找线性表中元素值为 e 的元素,查找成功将对应元素的序号返回,否则返回 0 表示失败*/
{
    ListNode *p;                  /*定义指向单链表的指针*/
    int i;                        /*定义指示器变量*/
    if(ListEmpty(head))           /*在查找第 i 个元素之前,判断链表是否为空*/
          return 0;
    p=head->next;                 /*指针 p 指向第一个结点*/
    i=1;                          /*将指示器置为 1*/
    while(p)
    {
        if(p->data==e)            /*若找到与 e 相等的元素*/
           return i;              /*返回该序号*/
        else                      /*否则*/
        {
            p=p->next;            /*令 p 指向下一个结点继续查找*/
            i++;                  /*指示器加 1*/
         }
    }
    if(!p)                        /*如果没有找到与 e 相等的元素*/
         return 0;                /*返回 0*/
}
```

（6）在第 *i* 个位置插入元素 *e*。插入成功返回 1，否则返回 0；如果没有与 *e* 值相等的元素，返回 0 表示失败。

假设存储元素 *e* 的结点为 *p*，要将 *p* 指向的结点插入 pre 和 pre→next 之间，根本不需要移动其他结点，只需要让 *p* 指向结点的指针和 pre 指向结点的指针做一点改变即可。即先把*pre 的直接后继结点变成*p* 的直接后继结点，然后把*p* 变成*pre 的直接后继结点，如图 3-13 所示，代码如下。

```
p->next=pre->next;
pre->next=p;
```

图 3-13　在*pre 结点之后插入新结点*p*

注意：插入结点的两行代码不能颠倒顺序。如果先进行 pre→next=*p*，后进行 *p*→next=pre→next 操作，则第一条代码就会覆盖 pre→next 的地址，pre→next 的地址就变成了 *p* 的地址，执行 *p*→next=pre→next 就等于执行 *p*→next=*p*，这样 pre→next 就与上级断开了链接，造成尴尬的局面，如图 3-14 所示。

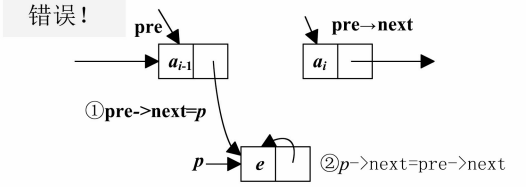


图 3-14　插入结点代码顺序颠倒后，*(pre→next)结点与上级断开链接

如果要在单链表的第 i 个位置插入一个新元素 e，首先需要在链表中找到其直接前驱结点，即第 i-1 个结点，并由指针 pre 指向该结点，如图 3-15 所示。然后申请一个新结点空间，由 p 指向该结点，将值 e 赋值给 p 指向结点的数据域，最后修改*p 和*pre 结点的指针域，如图 3-16 所示。这样就完成了结点的插入操作。

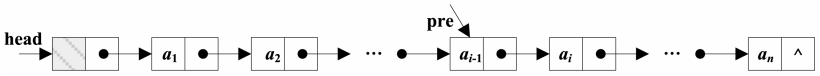


图 3-15　找到第 i 个结点的直接前驱结点

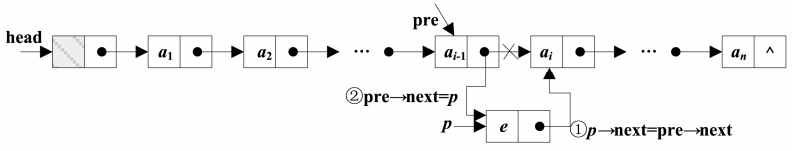


图 3-16　将新结点插入第 i 个位置

在单链表的第 i 个位置插入新数据元素 e 的算法实现如下。

```
int InsertList(LinkList head,int i,DataType e)
/*在单链表中第 i 个位置插入一个结点,结点的元素值为 e。插入成功返回 1,失败返回 0*/
{
     ListNode *pre,*p;      /*定义第 i 个元素的前驱结点指针 pre,指针 p 指向新生成的结点*/
     int j;                 /*定义计数器变量*/
     pre=head;              /*指针 p 指向头结点*/
     j=0;                                /*将计数器置为 0*/
     while(pre->next!=NULL&&j<i-1)  /*在未到最后一个结点前,找到第 i 个结点的前驱结点*/
     {
           pre=pre->next;                /*令 pre 指向下一个结点*/
           j++;                          /*计数器加 1*/
     }
     if(j!=i-1)                          /*若不存在第 i 个结点的前驱结点,说明插入位置错误*/
     {
           printf("插入位置错误! ");  /*输出错误提示信息*/
           return 0;                     /*返回 0*/
     }
     /*新生成一个结点,并将 e 赋值给该结点的数据域*/
     if((p=(ListNode*)malloc(sizeof(ListNode)))==NULL)  /*动态分配一个结点的内存空间*/
         exit(-1);
     p->data=e;
                                         /*插入结点操作*/
     p->next=pre->next;
     pre->next=p;
     return 1;                           /*返回 1*/
}
```

（7）删除第 i 个结点。

假设 p 指向第 i 个结点，要将*p 结点删除，只需要绕过它的直接前驱结点的指针，使其直接指向它的直接后继结点即可删除链表的第 i 个结点，如图 3-17 所示。

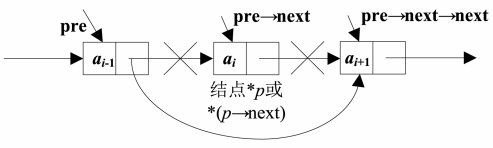


图 3-17　删除*pre 的直接后继结点

将单链表中第 i 个结点删除可分为 3 步：第一步找到第 i 个结点的直接前驱结点，即第 i-1 个结点，并用 pre 指向该结点，p 指向其直接后继结点，即第 i 个结点，如图 3-18 所示；第二步将*p 结点的数据域赋值给 e；第三步删除第 i 个结点，即 pre→next=p→next，并释放*p 结点的内存空间。删除过程如图 3-19 所示。

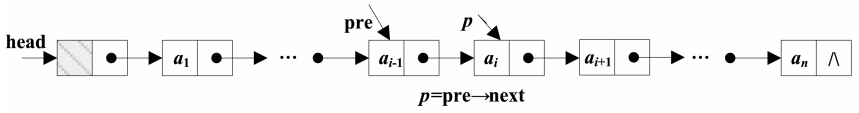


图 3-18　找到第 i-1 个结点和第 i 个结点

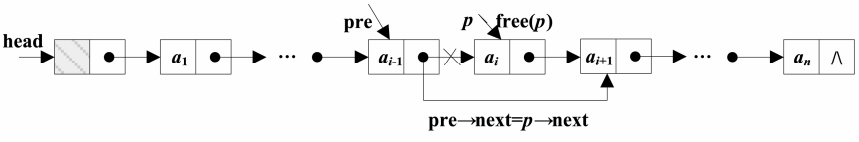


图 3-19　删除第 i 个结点

删除第 i 个结点的算法实现如下。

```
int DeleteList(LinkList head,int i,DataType *e)
/*删除单链表中的第 i 个位置的结点。删除成功返回 1,失败返回 0*/
{
    ListNode *pre,*p;              /*定义第 i 个元素的前驱结点指针 pre 和指向新结点的指针 p*/
    int j;                         /*定义计数器变量*/
    pre=head;                      /*指针 p 指向头结点*/
    j=0;                           /*将计数器置为 0*/
    while(pre->next!=NULL&&pre->next->next!=NULL&&j<i-1)/*判断是否找到前驱结点*/
    {
        pre=pre->next;             /*指向下一个结点*/
        j++;                       /*计数器加 1*/
    }
    if(j!=i-1)                     /*如果没找到要删除的结点位置,说明删除位置有误*/
    {
        printf("删除位置有误");    /*输出错误提示信息*/
        return 0;                  /*返回 0*/
    }
    /*指针 p 指向单链表中的第 i 个结点,并将该结点的数据域值赋值给 e*/
    p=pre->next;
    *e=p->data;
    /*将前驱结点的指针域指向要删除结点的下一个结点,也就是将 p 指向的结点与单链表断开*/
    pre->next=p->next;             /*令 pre 指向 p 的下一个结点*/
    free(p);                       /*释放 p 指向的结点*/
    return 1;                      /*返回 1*/
}
```

注意：在查找第 i-1 个结点时，要注意不可遗漏判断条件 pre→next→next!=NULL，确保第 i 个结点非空。如果没有此判断条件，而 pre 指针指向了单链表的最后一个结点，在执行循环后的 p=pre→next，*e=p→data 操作时，p 指针指向的就是 NULL 指针域，会产生致命错误。

（8）求表长操作。求表长操作即返回单链表的元素个数，求单链表的表长算法实现代码如下。

```
int ListLength(LinkList head)
/*求表长操作*/
{
    ListNode *p;                   /*定义指向新生成的结点指针变量*/
    int count=0;                   /*定义计数器变量 count 并初始化*/
    p=head;                        /*指针 p 指向头结点*/
```

```
    while(p->next!=NULL)          /*如果指针 p 没有到达链表末尾*/
    {
        p=p->next;                /*令 p 指向下一个结点*/
        count++;                  /*计数器加 1*/
    }
    return count;                 /*返回元素个数*/
}
```

（9）销毁链表操作，实现代码如下。

```
void DestroyList(LinkList head)
/*销毁链表*/
{
    ListNode *p,*q;               /*定义指向新生成的结点的指针变量*/
    p=head;                       /*指针 p 指向头结点*/
    while(p!=NULL)                /*如果链表不为空*/
    {
        q=p;                      /*q 指向待销毁的结点*/
        p=p->next;                /*p 指向下一个结点*/
        free(q);                  /*释放 q 指向的结点空间*/
    }
}
```

3.3.3　单链表存储结构与顺序存储结构的优缺点

下面简单对单链表存储结构和顺序存储结构进行对比。

1. 存储分配方式

顺序存储结构用一组连续的存储单元依次存储线性表的数据元素。单链表采用链式存储结构，用一组任意的存储单元存放线性表的数据元素。

2. 时间性能

采用顺序存储结构时，查找操作时间复杂度为 O（1），插入和删除操作需要移动平均一半的数据元素，时间复杂度为 O（n）。采用单链表存储结构时，查找操作时间复杂度为 O（n），插入和删除操作不需要大量移动元素，时间复杂度仅为 O（1）。

3. 空间性能

采用顺序存储结构时，需要预先分配存储空间，分配的空间过大会造成浪费，分配的空间过小不能满足问题需要。采用单链表存储结构时，可根据需要临时分配，不需要估计问题的规模大小，只要内存够就可以分配，还可以用于一些特殊情况，如一元多项式的表示。

3.3.4　单链表应用举例

【例 3-3】已知两个单链表 A 和 B，其中的元素都是非递减排列，编写算法将单链表 A 和 B 合并得到一个递减有序的单链表 C（值相同的元素只保留一个），并要求利用原链表结点空间。

【分析】此题为单链表合并问题。利用头插法建立单链表，使先插入元素值小的结点在链表末尾，后插入元素值大的结点在链表表头。初始时，单链表 C 为空（插入的是 C 的第一个结点），将单链表 A 和 B 中较小的元素值结点插入 C 中；单链表 C 不为空时，比较 C 和将插入结点的元素值大小，值不同时插入到 C 中，值相同时，释放该结点。当 A 和 B 中有一个链表为空时，将剩下的结点依次插入 C 中。核心算法实现代码如下。

```
void MergeList(LinkList A,LinkList B,LinkList *C)
```

```
/*将非递减排列的单链表 A 和 B 中的元素合并到 C 中,使 C 中的元素按递减排列,相同值的元素只保留一个*/
{
    ListNode *pa,*pb,*qa,*qb;          /*定义指向单链表 A,B 的指针*/
    pa=A->next;                        /*pa 指向单链表 A*/
    pb=B->next;                        /*pb 指向单链表 B*/
    free(B);                           /*释放单链表 B 的头结点*/
    *C=A;                              /*初始化单链表 C,利用单链表 A 的头结点作为 C 的头结点*/
    (*C)->next=NULL;                   /*单链表 C 初始时为空*/
    /*利用头插法将单链表 A 和 B 中的结点插入到单链表 C 中（先插入元素值较小的结点）*/
    while(pa&&pb)                      /*单链表 A 和 B 均不空时*/
    {
       if(pa->data<pb->data)           /*pa 指向结点元素值较小时,将 pa 指向的结点插入到 C 中*/
       {
          qa=pa;                       /*qa 指向待插入结点*/
          pa=pa->next;                 /*pa 指向下一个结点*/
          if((*C)->next==NULL)              /*单链表 C 为空时,直接将结点插入到 C 中*/
          {
             qa->next=(*C)->next;
             (*C)->next=qa;
          }
          else if((*C)->next->data<qa->data)  /*pa 指向的结点元素值不同于已有结点元素值时,才插入结点*/
          {
             qa->next=(*C)->next;
             (*C)->next=qa;
          }
          else                             /*否则,释放元素值相同的结点*/
             free(qa);
       }
       else                                /*pb 指向结点元素值较小,将 pb 指向的结点插入到 C 中*/
       {
          qb=pb;                           /*qb 指向待插入结点*/
          pb=pb->next;                     /*pb 指向下一个结点*/
          if((*C)->next==NULL)             /*单链表 C 为空时,直接将结点插入到 C 中*/
          {
             qb->next=(*C)->next;
             (*C)->next=qb;
          }
          else if((*C)->next->data<qb->data)  /*pb 指向的结点元素值不同于已有结点元素时,才将结点插入*/
          {
             qb->next=(*C)->next;
             (*C)->next=qb;
          }
          else                             /*否则,释放元素值相同的结点*/
             free(qb);
       }
    }
    while(pa)                       /*如果 pb 为空、pa 不为空,则将 pa 指向的后继结点插入到 C 中*/
    {
       qa=pa;                       /*qa 指向待插入结点*/
       pa=pa->next;                 /*pa 指向下一个结点*/
       if((*C)->next&&(*C)->next->data<qa->data)
       {                            /*pa 指向的结点元素值不同于已有结点元素时,才将结点插入*/
          qa->next=(*C)->next;
          (*C)->next=qa;
       }
       else                         /*否则,释放元素值相同的结点*/
          free(qa);
    }
    while(pb)                       /*如果 pa 为空、pb 不为空,则将 pb 指向的后继结点插入到 C 中*/
    {
```

```
            qb=pb;                          /*qb 指向待插入结点*/
            pb=pb->next;                    /*pb 指向下一个结点*/
            if((*C)->next&&(*C)->next->data<qb->data)
            {                               /*pb 指向的结点元素值不同于已有结点元素时,才将结点插入*/
               qb->next=(*C)->next;
               (*C)->next=qb;
            }
            else                            /*否则,释放元素值相同的结点*/
               free(qb);
        }
    }
```

程序的运行结果如图 3-20 所示。

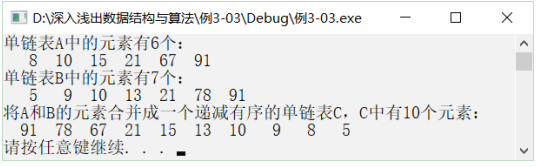


图 3-20　合并单链表的程序运行结果

在将两个单链表 *A* 和 *B* 的合并算法 MergeList 中，需要特别注意的是，不要遗漏单链表为空时的处理。当单链表为空时，将结点插入 *C* 中，代码如下。

```
    if((*C)->next==NULL)                    /*单链表 C 为空时,直接将结点插入 C 中*/
    {
        qa->next=(*C)->next;
        (*C)->next=qa;
    }
```

针对这个题目，经常会遗漏单链表为空的情况，以下代码遗漏了单链表为空的情况。

```
    if((*C)->next&&(*C)->next->data<qb->next)   /*错误代码：遗漏了单链表为空的情况*/
    {
        qb->next=(*C)->next;
        (*C)->next=qb;
    }
```

所以，对于初学者而言，写完算法后，一定要上机调试下算法的正确性。

【例 3-4】利用单链表的基本运算，求两个集合的交集。

【分析】假设 *A* 和 *B* 是两个带头结点的单链表，分别表示两个给定的集合 *A* 和 *B*，求 $C=A\cap B$。先将单链表 *A* 和 *B* 分别从小到大排序，然后依次比较两个单链表中的元素值大小，pa 指向 *A* 中当前比较的结点，pb 指向 *B* 中当前比较的结点，如果 pa→data<pb→data，则 pa 指向 *A* 中下一个结点；如果 pa→data>pb→data，则 pb 指向 *B* 中下一个结点；如果 pa→data==pb→data，则将当前结点插入 *C* 中。

```
    void Interction(LinkList A,LinkList B,LinkList *C)
    /*求 A 和 B 的交集*/
    {
        ListNode *pa,*pb,*pc;                   /*定义 3 个结点指针*/
        Sort(A);                                /*对数组 A 进行排序*/
        printf("排序后 A 中的元素:\n");          /*输出提示信息*/
        DispList(A);                            /*输出排序后 A 中的元素*/
        Sort(B);                                /*对数组 B 进行排序*/
        printf("排序后 B 中的元素:\n");          /*输出提示信息*/
        DispList(B);                            /*输出排序后 B 中的元素*/
        pa=A->next;                             /*pa 指向 A 的第一个结点*/
        pb=B->next;                             /*pb 指向 B 的第一个结点*/
        *C=(LinkList)malloc(sizeof(ListNode));  /*为指针*C 指向的新链表动态分配内存空间*/
```

```
        (*C)->next=NULL;
        while(pa&&pb)                          /*若 pa 和 pb 指向的结点都不为空*/
        {
           if(pa->data<pb->data)               /*如果pa指向的结点元素值小于pb指向的结点元素值*/
              pa=pa->next;                     /*则略过该结点*/
           else if(pa->data>pb->data)          /*如果pa指向的结点元素值大于pb指向的结点元素值*/
              pb=pb->next;                     /*则略过该结点*/
           else                                /*否则*/
           {                                   /*即 pa->data==pb->data,则将当前结点插入C中.*/
              pc=(ListNode*)malloc(sizeof(ListNode));
              pc->data=pa->data;
              pc->next=(*C)->next;
              (*C)->next=pc;
              pa=pa->next;                     /*则 pa 指向 A 中下一个结点*/
              pb=pb->next;                     /*则 pb 指向 B 中下一个结点*/
           }
        }
    }
```

程序的运行结果如图 3-21 所示。

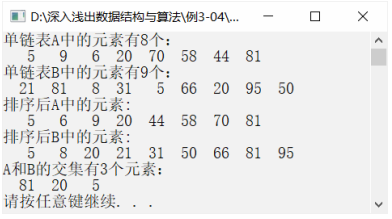


图 3-21　求 *A* 和 *B* 交集的程序运行结果

【考研真题】假设一个带有表头结点的单链表，结点结构如下。

data	link

假设该链表只给出了头指针 list，在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第 *k* 个位置上的结点（*k* 为正整数）。若查找成功，算法输出该结点数据域的值，并返回 1；否则返回 0。要求如下。

（1）描述算法的基本设计思想。

（2）描述算法的详细实现步骤。

（3）根据设计思想和实现步骤，采用程序设计语言描述算法。

【分析】这是一道考研试题，主要考查对链表的掌握程度，这个题目比较灵活，利用一般的思维方式不容易实现。

（1）算法的基本思想：定义两个指针 *p* 和 *q*，初始时均指向头结点的下一个结点。*p* 指针沿着链表移动，当 *p* 指针移动到第 *k* 个结点时，*q* 指针与 *p* 指针同步移动，当 *p* 指针移动到链表表尾结点时，*q* 指针所指向的结点即为倒数第 *k* 个结点。

（2）算法的详细步骤如下。

① 令 count=0，*p* 和 *q* 指向链表的第一个结点。

② 若 *p* 为空，则转向⑤执行。

③ 若 count 等于 *k*，则 *q* 指向下一个结点；否则令 count++。

④ 令 *p* 指向下一个结点，转向②执行。

⑤ 若 count 等于 *k*，则查找成功，输出结点的数据域的值，并返回 1；否则，查找失败，

返回 0。

（3）算法实现代码如下。

```
typedef struct LNode                    /*定义结点*/
{
   int data;
   struct Lnode *link;
}*LinkList;                             /*定义结点指针变量*/
int SearchNode(LinkList list,int k)     /*查找结点*/
{
    LinkList p,q;                       /*定义两个指针 p 和 q*/
    int count=0;                        /*定义计数器变量并赋初值为 0*/
    p=q=list->link;                     /*p 和 q 指向链表的第一个结点*/
    while(p!=NULL)
    {
        if(count<k)                     /*若 p 未移动到第 k 个结点*/
           count++;                     /*则计数器加 1*/
        else
           q=q->link;                   /*当 p 移到第 k 个结点后,q 开始与 p 同步移动下一个结点*/
        p=p->link;                      /*p 移动到下一个结点*/
    }
    if(count<k)                         /*如果满足小于 k*/
       return 0;                        /*返回 0*/
    else
    {
       printf("倒数第%d 个结点元素值为%d\n",k,q->data); /*输出倒数第 k 个结点的元素值*/
       return 1;                        /*返回 1*/
     }
}
```

3.4　循环单链表

循环单链表是首尾相连的单链表，是另一种形式的单链表。将单链表的最后一个结点的指针域由空指针改为指向头结点或第一个结点，整个链表就形成一个环，这样的单链表称为循环单链表。从表中任何一个结点出发均可找到表中其他结点。

与单链表类似，循环单链表也可分为带头结点结构和不带头结点结构两种。对于不带头结点的循环单链表，当表不为空时，最后一个结点的指针域指向头结点，如图 3-22 所示。对于带头结点的循环单链表，当表为空时，头结点的指针域指向头结点本身，如图 3-23 所示。

图 3-22　循环单链表

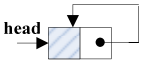


图 3-23　结点为空的循环单链表

循环单链表与单链表在结构、类型定义及实现方法上都是一样的，唯一的区别仅在于判断链表是否为空的条件上。判断单链表为空的条件是 head→next==NULL，判断循环单链表为空的条件是 head->next==head。

在单链表中，访问第一个结点的时间复杂度为 $O(1)$，而访问最后一个结点则需要将整个单链表扫描一遍，故时间复杂度为 $O(n)$。对于循环单链表，只需设置一个尾指针（利用 rear 指向循环单链表的最后一个结点）而不设置头指针，就可以直接访问最后一个结点，时间复杂度为 $O(1)$。访问第一个结点即 rear->next->next，时间复杂度也为 $O(1)$，如图 3-24 所示。

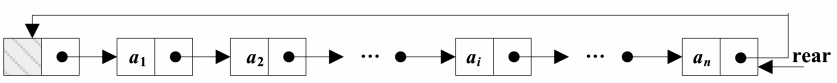


图 3-24　仅设置尾指针的循环单链表

在循环单链表中设置尾指针，还可以使有些操作变得简单，例如，要将如图 3-25 所示的两个循环单链表（尾指针分别为 LA 和 LB）合并成一个链表，只需要将一个表的表尾和另一个表的表头连接即可，如图 3-26 所示。

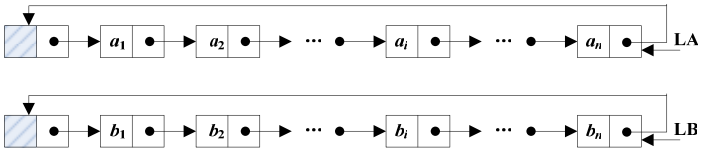


图 3-25　两个设置尾指针的循环单链表

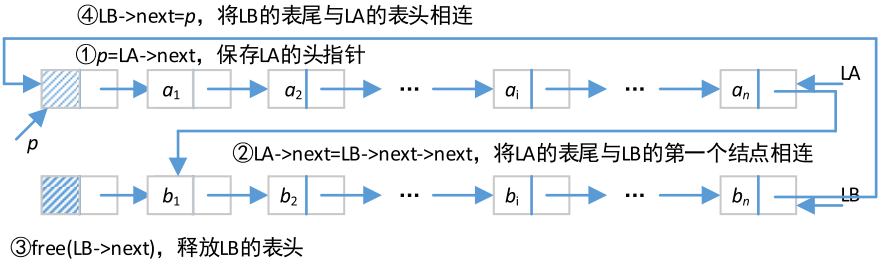


图 3-26　合并两个设置尾指针的循环单链表

将循环单链表合并为一个循环单链表只需要 4 步操作：①保存 LA 的头指针，即 p=LA->next。②将 LA 的表尾与 LB 的第一个结点相连，即 LA->next=LB->next->next；③释放 LB 的头结点，即 free(LB->next)；④将 LB 的表尾与 LA 的表头相连，即 LB->next=p。

对于设置了头指针的两个循环单链表（头指针分别是 head1 和 head2），要将其合并成一个循环单链表，需要先找到两个链表的最后一个结点，分别增加一个尾指针，分别使其指向最后一个结点。然后将第一个链表的尾指针与第二个链表的第一个结点连接起来，第二个链表的尾指针与第一个链表的第一个结点连接起来，就形成了一个循环链表。

合并两个循环单链表的算法实现如下。

```
LinkList Link(LinkList head1,LinkList head2)
/*将两个链表 head1 和 head2 连接在一起形成一个循环链表*/
{
    ListNode *p,*q;                 /*定义两个指针变量 p 和 q*/
    p=head1;                        /*p 指向第一个链表*/
    while(p->next!=head1)           /*指针 p 指向链表的最后一个结点*/
         p=p->next;
    q=head2;
    while(q->next!=head2)           /*指针 q 指向链表的最后一个结点*/
```

```
        q=q->next;                  /*指向下一个结点*/
    p->next=head2->next;            /*将第一个链表的尾端连接到第二个链表的第一个结点*/
    q->next=head1;                  /*将第二个链表的尾端连接到第一个链表的第一个结点*/
    return head1;                   /*返回第一个链表的头指针*/
}
```

3.5　双向链表

在单链表和循环单链表中，每个结点只有一个指向其后继结点的指针域，只能根据指针域查找后继结点，要查找指针 p 指向结点的直接前驱结点，必须从 p 指针出发，顺着指针域把整个链表访问一遍，才能找到该结点，其时间复杂度是 $O(n)$。因此，要访问某个结点的前驱结点，效率太低，为了便于操作，可将单链表设计成双向链表。

3.5.1　双向链表的存储结构

顾名思义，双向链表就是链表中的每个结点有两个指针域：一个指向直接前驱结点，另一个指向直接后继结点。双向链表的每个结点有 data 域、prior 域和 next 域 3 个域。双向链表的结点结构如图 3-27 所示。

图 3-27　双向链表的结点结构

其中，data 域为数据域，存放数据元素；prior 域为前驱结点指针域，指向直接前驱结点；next 域为后继结点指针域，指向直接后继结点。

与单链表类似，也可以为双向链表增加一个头结点，这样使某些操作更加方便。双向链表也有循环结构，称为双向循环链表。带头结点的双向循环链表如图 3-28 所示。双向循环链表为空的情况如图 3-29 所示，判断带头结点的双向循环链表为空的条件是 head->prior==head 或 head->next==head。

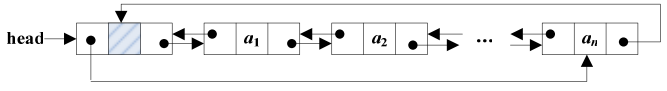


图 3-28　带头结点的双向循环链表

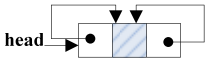


图 3-29　带头结点的空双向循环链表

在双向链表中，因为每个结点既有前驱结点的指针域又有后继结点的指针域，所以查找结点非常方便。对于带头结点的双向链表，如果链表为空，则有 p=p->prior->next=p->next->prior。

双向链表的结点存储结构描述如下。

```
typedef struct Node                 /*定义双向链表的结点存储结构*/
{
```

```
    DataType data;                  /*数据域*/
    struct Node *prior;             /*指向前驱结点的指针域*/
    struct Node *next;              /*指向后继结点的指针域*/
}DListNode,*DLinkList;
```

3.5.2 双向链表的插入和删除操作

在双向链表中，有些操作如求链表的长度、查找链表的第 i 个结点等，仅涉及一个方向的指针，与单链表中的算法实现基本没什么区别。但是对于双向循环链表的插入和删除操作，因为涉及前驱结点和后继结点的指针，所以需要修改两个方向上的指针。

1. 在第 i 个位置插入元素值为 e 的结点

首先找到第 i 个结点，用 p 指向该结点；再申请一个新结点，由 s 指向该结点，将 e 放入数据域；然后修改 p 和 s 指向的结点的指针域，修改 s 的 prior 域，使其指向 p 的直接前驱结点，即 s->prior=p->prior；修改 p 的直接前驱结点的 next 域，使其指向 s 指向的结点，即 p->prior->next=s；修改 s 的 next 域，使其指向 p 指向的结点，即 s->next=p；修改 p 的 prior 域，使其指向 s 指向的结点，即 p->prior=s。插入操作指针修改情况如图 3-30 所示。

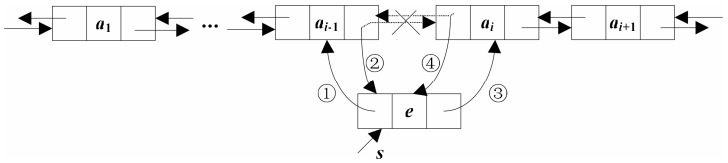


图 3-30 双向循环链表的插入结点操作过程

插入操作算法实现如下。

```
int InsertDList(DListLink head,int i,DataType e)  /*双向链表插入操作的算法实现*/
{
    DListNode *p,*s;                            /*定义双向链表的结点指针 p 和 s*/
    int j;
    p=head->next;                               /*p 指向链表的第一个结点*/
    j=0;                                        /*计数器初始化为 0*/
    while(p!=head&&j<i)                         /*若还未到第 i 个结点*/
    {
        p=p->next;                              /*则继续查找下一个结点*/
        j++;                                    /*计数器加 1*/
    }
    if(j!=i)                                    /*若不存在第 i 个结点*/
    {
       printf("插入位置不正确");                /*则输出错误提示信息*/
       return 0;                                /*返回 0*/
     }
     s=(DListNode*)malloc(sizeof(DListNode));/*动态分配一个结点内存空间,由 s 指向该结点*/
     if(!s)
         return -1;
     s->data=e;                         /*将参数 e 存入数据域*/
     s->prior=p->prior;                 /*修改 s 的 prior 域,使其指向 p 的直接前驱结点*/
     p->prior->next=s;                  /*修改 p 的前驱结点的 next 域,使其指向 s 指向的结点*/
     s->next =p;                        /*修改 s 的 next 域,使其指向 p 指向的结点*/
     p->prior=s;                        /*修改 p 的 prior 域,使其指向 s 指向的结点*/
     return 1;                          /*插入成功,返回 1*/
```

```
}
```

2. 删除第 i 个结点

首先找到第 i 个结点，用 p 指向该结点；然后修改 p 指向的结点的直接前驱结点和直接后继结点的指针域，从而将 p 与链表断开。将 p 指向的结点与链表断开需要两步：第一步，修改 p 的前驱结点的 next 域，使其指向 p 的直接后继结点，即 p->prior->next=p->next；第二步，修改 p 的直接后继结点的 prior 域，使其指向 p 的直接前驱结点，即 p->next->prior=p->prior。删除操作指针修改情况如图 3-31 所示。

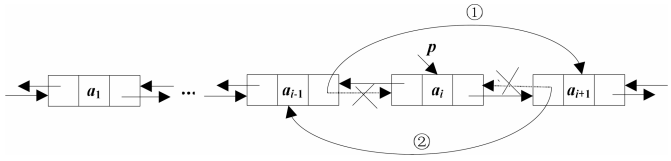


图 3-31　双向循环链表的删除结点操作过程

删除操作算法实现如下。

```
int DeleteDList(DListLink head,int i,DataType *e)     /*双向链表删除操作的算法实现*/
{
    DListNode *p;
    int j;
    p=head->next;                                     /*p 指向双向链表的第一个结点*/
    j=0;                                              /*计数器初始化为 0*/
    while(p!=head&&j<i)                               /*若还未找到待删除的结点*/
    {
       p=p->next;                                     /*则令 p 指向下一个结点继续查找*/
       j++;                                           /*计数器加 1*/
    }
    if(j!=i)                                          /*若不存在待删除的结点位置*/
    {
       printf("删除位置不正确");                       /*则输出错误提示信息*/
       return 0;                                      /*返回 0*/
    }
    p->prior->next=p->next;    /*修改 p 的前驱结点的 next 域,使其指向 p 的直接后继结点*/
    p->next->prior =p->prior;  /*修改 p 的直接后继结点的 prior 域,使其指向 p 的直接前驱结点*/
    free(p);                   /*释放 p 指向结点的空间*/
    return 1;                  /*返回 1*/
}
```

插入和删除操作的时间耗费主要在查找结点上，两者的时间复杂度都为 $O(n)$。

3.5.3　双向链表应用举例

【例 3-5】约瑟夫环问题。有 n 个小朋友，编号分别为 1，2，…，n，按编号围成一个圆圈，他们按顺时针方向从编号为 k 的人由 1 开始报数，报数为 m 的人出列，他的下一个人重新从 1 开始报数，数到 m 的人出列，照这样重复下去，直到所有人都出列。编写一个算法，输入 n、k 和 m，按照出列顺序输出编号。

【分析】解决约瑟夫环问题可以分为 3 个步骤：第一步创建一个具有 n 个结点的不带头结点的双向循环链表（模拟编号从 1～n 的圆圈可以利用循环单链表实现，这里采用双向循环链表实现），编号从 1 到 n，代表 n 个小朋友；第二步找到第 k 个结点，即第一个开始报数的人；第三步，编号为 k 的人从 1 开始报数，并开始计数，报到 m 的人出列即将该结点删除。继续从下一

个结点开始报数，直到最后一个结点被删除。

```
void Josephus(DLinkList head,int n,int m,int k)
/*在长度为 n 的双向循环链表中,从第 k 个人开始报数,数到 m 的人出列*/
{
    DListNode *p,*q;                        /*定义结点指针变量*/
    int i;
    p=head;                                 /*p 指向双向循环链表的第一个结点*/
    for(i=1;i<k;i++)                        /*从第 k 个人开始报数*/
    {
       q=p;
       p=p->next;                           /*p 指向下一个结点*/
    }
    while(p->next!=p)
    {
       for(i=1;i<m;i++)                     /*数到 m 的人出列*/
       {
          q=p;
          p=p->next;                        /*p 指向下一个结点*/
       }
       q->next=p->next;                     /*将 p 指向的结点删除,即报数为 m 的人出列*/
       p->next->prior=q;
       printf("%4d",p->data);               /*输出被删除的结点*/
       free(p);                             /*释放 p 指向的结点空间*/
       p=q->next;                           /*p 指向下一个结点,重新开始报数*/
    }
    printf("%4d\n",p->data);                /*输出最后出列的人*/
}
```

程序运行结果如图 3-32 所示。

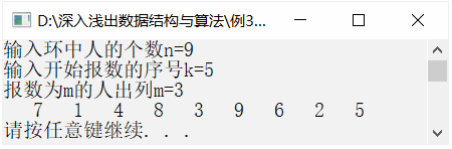


图 3-32　约瑟夫问题程序运行结果

在创建双向循环链表 CreateDCList 函数中，根据创建的是否为第一个结点分为两种情况处理。如果是第一个结点，则让该结点的前驱结点指针域和后继结点指针域都指向该结点，并让头指针指向该结点，代码如下。

```
head=s;
s->prior=head;
s->next=head;
```

切记不要漏掉 *s*->next=head 或 *s*->prior=head，否则在程序运行时会出现错误。

如果不是第一个结点，则将新结点插入双向链表的尾部，代码如下。

```
s->next=q->next;
q->next=s;
s->prior=q;
head->prior=s;
```

注意：语句 *s*->next=*q*->next 和 *q*->next=*s* 的顺序不能颠倒，另外不要忘记让头结点的 prior 域指向 *s*。

3.6　综合案例：一元多项式的表示与相加

一元多项式的相加是线性表在生活中的一个实际应用，它涵盖了本节所学到的链表的各种操作。通过使用链表实现一元多项式的相加，巩固读者对链表基本操作的理解与掌握。

3.6.1　一元多项式的表示

假设一元多项式为 $P_n(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$，一元多项式的每一项由系数和指数构成，因此要表示一元多项式，需要定义一个结构体。结构体由两个部分构成，分别为 coef 和 exp，分别表示系数和指数。定义结构体的代码如下。

```
struct node          /*定义结构体 struct node*/
{
    float coef;      /*系数*/
    int exp;         /*指数*/
};
```

如果用结构体数组表示多项式的每一项，则需要 $n+1$ 个数组元素存放多项式（假设 n 为最高次数）。遇到指数不连续且指数之间跨越非常大时，例如，多项式 $2x^{500}+1$，则需要数组的长度为 501。这显然会浪费很多内存单元。

为了有效利用内存空间，可以使用链表表示多项式，多项式的每一项使用结点表示。结点由系数、指数和指针域 3 个部分构成，结构如图 3-33 所示。

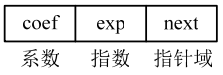


图 3-33　多项式每一项的结点结构

结点用 C 语言描述如下。

```
struct node                 /*定义结构体 struct node*/
{
    float coef;             /*系数*/
    int exp;                /*指数*/
    struct node *next;      /*指针域*/
};
```

3.6.2　一元多项式相加

为了操作方便，将链表按照指数从高到低进行排列，即降幂排列。一个最高次数为 n 的多项式构成的链表如图 3-34 所示。

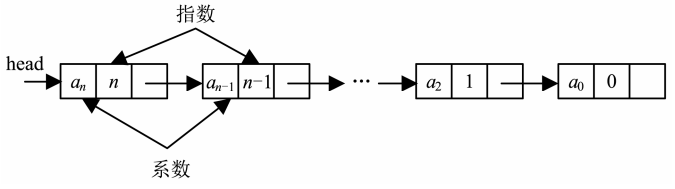


图 3-34　一元多项式的链表结构

例如，有两个一元多项式 $p(x)=3x^2+2x+1$ 和 $q(x)=5x^3+3x+2$，链表表示如图 3-35 所示。

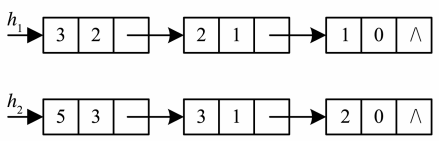


图 3-35　一元多项式的链表表示

如果要将两个多项式相加，需要比较两个多项式的指数项后决定。当两个多项式的两项中指数相同时，才将系数相加。如果两个多项式的指数不相等，则多项式该项和的系数是其中一个多项式的系数。实现代码如下。

```
if(s1->exp==s2->exp)            /*如果两个指数相等,则将系数相加*/
{
   c=s1->coef+s2->coef;
   e=s1->exp;
   s1=s1->next;
   s2=s2->next;
}
else if(s1->exp>s2->exp)        /*如果 s1 的指数大于 s2 的指数,则将 s1 的指数作为结果*/
{
   c=s1->coef;
   e=s1->exp;
   s1=s1->next;
}
else                            /*如果 s1 的指数小于或等于 s2,则将 s2 的指数作为结果*/
{
   c=s2->coef;
   e=s2->exp;
   s2=s2->next;
}
```

其中，s1 和 s2 分别指向两个链表表示的表达式。因为表达式是按照指数从大到小排列的，所以在指数不等时，将指数大的作为结果。指数小的还要继续进行比较。例如，如果当前 s1 指向系数为 3，指数为 2 的结点即(3,2)，s2 指向(3,1)的结点，因为 s1->exp>s2->exp，所以将 s1 的结点作为结果。在 s1 指向(2,1)时，还要与 s2 的(3,1)相加，得到(5,1)。

如果相加后的系数不为 0，则需要生成一个结点存放到链表中，代码如下。

```
if(c!=0)                                        /*如果相加后的系数不为 0*/
{
   p=(ListNode*)malloc(sizeof(ListNode));       /*动态生成一个结点 p*/
   p->coef=c;                                   /*将系数存入 coef 域*/
   p->exp=e;                                    /*将指数存入 exp 域*/
   p->next=NULL;                                /*结点的指针域为空*/
   if(s==NULL)                                  /*若新生成的链表为空*/
      s=p;                                      /*则 s 指向新生成的结点*/
   else                                         /*否则*/
      r->next=p;                                /*p 指向的结点成为 r 的下一个结点*/
   r=p;                                         /*使 r 指向新链表的最后一个结点*
}
```

如果在一个链表已经到达末尾，而另一个链表还有结点时，需要将剩下的结点插入新链表中，代码如下。

```
while(s1!=NULL)                     /*如果 s1 还有结点*/
{
   c=s1->coef;                      /*s1 结点的系数赋给 c*/
   e=s1->exp;                       /*s1 结点的指数赋给 e*/
   s1=s1->next;                     /*s1 结点指向下一个结点*/
```

```
        if(c!=0)                          /*如果相加后的系数不为 0,则生成一个结点存放到链表*/
        {
           p=(ListNode*)malloc(sizeof(ListNode)); /*动态生成一个结点 p*/
           p->coef=c;
           p->exp=e;
           p->next=NULL;
           if(s==NULL)                    /*若新生成的链表为空*/
              s=p;                        /*则将 p 指向的结点作为第一个结点*/
           else                           /*否则*/
              r->next=p;                  /*将新结点插入 r 指向的结点之后*/
           r=p;                           /*使 r 指向链表的最后一个结点*/
        }
    }
    while(s2!=NULL)                       /*如果 s2 还有剩余结点*/
    {
        c=s2->coef;                       /*s2 结点的系数赋给 c*/
        e=s2->exp;                        /*s2 结点的指数赋给 e*/
        s2=s2->next;                      /*s2 结点指向下一个结点*/
        if(c!=0)                          /*如果相加后的系数不为 0,则生成一个结点存放到链表*/
        {
           p=(ListNode*)malloc(sizeof(ListNode)); /*动态生成一个结点 p*/
           p->coef=c;
           p->exp=e;
           p->next=NULL;
           if(s==NULL)                    /*若新链表为空*/
              s=p;                        /*则将 p 指向的结点作为第一个结点*/
           else                           /*否则*/
              r->next=p;                  /*将新结点插入到 r 指向的结点之后*/
           r=p;                           /*使 r 指向链表的最后一个结点*/
        }
    }
```

最后，*s* 指向的链表就是两个多项式的和。

【例 3-6】依次输入两个多项式，编写程序求两个多项式的和。

```
ListNode *addpoly(ListNode *h1,ListNode *h2)
/*将两个多项式相加*/
{
    ListNode *p,*r=NULL,*s1,*s2,*s=NULL;
    float c;                              /*定义系数变量 c*/
    int e;                                /*定义指数变量 e*/
    s1=h1;                                /*使 s1 指向第一个多项式*/
    s2=h2;                                /*使 s2 指向第二个多项式*/
    while(s1!=NULL&&s2!=NULL)             /*如果两个多项式都不为空*/
    {
        if(s1->exp==s2->exp)              /*如果两个指数相等*/
        {
           c=s1->coef+s2->coef;           /*则对应系数相加后,将和赋给 c*/
           e=s1->exp;                     /*将指数赋给 e*/
           s1=s1->next;                   /*使 s1 指向下一个待处理结点*/
           s2=s2->next;                   /*使 s2 指向下一个待处理结点*/
        }
        else if(s1->exp>s2->exp)          /*如果第一个多项式结点的指数大于第二个多项式结点的指数*/
        {
           c=s1->coef;                    /*将第一个多项式结点的系数赋给 c*/
           e=s1->exp;                     /*将第一个多项式结点的指数赋给 e*/
           s1=s1->next;                   /*使 s1 指向下一个待处理结点*/
        }
        else                              /*否则*/
```

```
        {
            c=s2->coef;                 /*将第二个多项式结点的系数赋给 c*/
            e=s2->exp;                  /*将第二个多项式结点的指数赋给 e*/
            s2=s2->next;                /*使 s2 指向下一个待处理结点*/
        }
        if(c!=0)                        /*如果相加后的系数不为 0,则生成一个结点存放到链表*/
        {
            p=(ListNode*)malloc(sizeof(ListNode)); /*动态生成一个结点 p*/
            p->coef=c;                              /*将 c 赋给结点的系数*/
            p->exp=e;                               /*将 e 赋给结点的指数*/
            p->next=NULL;                           /*将结点的指针域置为空*/
            if(s==NULL)                             /*如果 s 为空链表*/
                s=p;                                /*则使新结点成为 s 的第一个结点*/
            else                                    /*否则*/
                r->next=p;                          /*使新结点*p 成为 r 的下一个结点*/
            r=p;                                    /*使 r 指向链表的最后一个结点*/
        }
    }
    while(s1!=NULL)                                 /*如果第一个多项式还有其他结点*/
    {
        c=s1->coef;                                 /*第一个多项式结点的系数赋给 c*/
        e=s1->exp;                                  /*第一个多项式结点的指数赋给 e*/
        s1=s1->next;                                /*将 s1 指向下一个结点*/
        if(c!=0)                        /*如果相加后的系数不为 0,则生成一个结点存放到链表*/
        {
            p=(ListNode*)malloc(sizeof(ListNode));
            p->coef=c;
            p->exp=e;
            p->next=NULL;
            if(s==NULL)
                s=p;
            else
                r->next=p;
            r=p;
        }
}
while(s2!=NULL)                                     /*如果第二个多项式还有其他结点*/
{
        c=s2->coef;                 /*第二个多项式结点的系数赋给 c*/
        e=s2->exp;                  /*第二个多项式结点的指数赋给 e*/
        s2=s2->next;                /*将 s2 指向下一个结点*/
        if(c!=0)                    /*如果相加后的系数不为 0,则生成一个结点存放到链表*/
        {
            p=(ListNode*)malloc(sizeof(ListNode));
            p->coef=c;
            p->exp=e;
            p->next=NULL;
            if(s==NULL)
                s=p;
            else
                r->next=p;
            r=p;
        }
    }
    return s;                       /*返回新生成的链表指针 s*/
}
```

程序运行结果如图 3-36 所示。

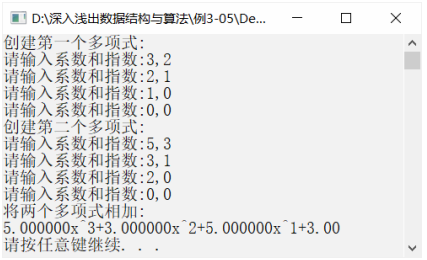


图 3-36　程序运行结果

3.7　小结

线性表中的元素之间是一对一的关系，除了第一个元素外，其他元素只有唯一的直接前驱，除了最后一个元素外，其他元素只有唯一的直接后继。

线性表有顺序存储和链式存储两种存储方式。采用顺序存储结构的线性表称为顺序表，采用链式存储结构的线性表称为链表。

顺序表中数据元素的逻辑顺序与物理顺序一致，因此可以随机存取。链表是靠指针域表示元素之间的逻辑关系。

链表又分为单链表和双向链表，这两种链表又可构成单循环链表、双向循环链表。单链表只有一个指针域，指针域指向直接后继结点。双向链表的一个指针域指向直接前驱结点，另一个指针域指向直接后继结点。

顺序表的优点是可以随机存取任意一个元素，算法实现较为简单，存储空间利用率高；缺点是需要预先分配存储空间，存储规模不好确定，插入和删除操作需要移动大量元素。链表的优点是不需要事先确定存储空间的大小，插入和删除元素不需要移动大量元素；缺点是只能从第一个结点开始顺序存取元素，存储单元利用率不高，算法实现较为复杂，因涉及指针操作，操作不当，会产生无法预料的内存错误。