信息与计算科学教学丛书

算法与数据结构

江世宏　编著

科学出版社

信息与计算科学教学丛书

算法与数据结构

江世宏　编著

科学出版社

北京

内 容 简 介

本书将数据结构定义为一个抽象数据类型（即 ADT），它由一个数据集合以及与数据集合绑定在一起的一组运算所构成；采用面向过程的结构化程序设计方法，用 C/C++程序具体实现它，并列举了一些应用这种数据结构解决实际问题的实例。

本书内容包括数组、链表、算法与数据结构绪论、线性表、栈、递归、队列、树、图、排序、搜索、散列和模板。

本书突出实用性、条理清晰、可操作性强，可作为本科高等学校计算机类和信息类专业的教材，尤其适合作为信息与计算科学专业的教材。也可以作为进一步学习程序设计，提高编程水平的参考书。

图书在版编目（CIP）数据

算法与数据结构/江世宏编著. —北京：科学出版社，2016.5
（信息与计算科学教学丛书）
ISBN 978-7-03-048190-0

Ⅰ.①算… Ⅱ.①江… Ⅲ.①算法分析 ②数据结构 Ⅳ.①TP301.6
②TP311.12

中国版本图书馆 CIP 数据核字（2016）第 093993 号

责任编辑：王丽舸/责任校对：董艳辉
责任印制：彭 超/封面设计：蓝 正

科学出版社 出版
北京东黄城根北街 16 号
邮政编码：100717
http://www.sciencep.com

武汉市首壹印务有限公司印刷
科学出版社发行 各地新华书店经销
*
开本：787×1092 1/16
2016 年 5 月第 一 版 印张：16
2016 年 5 月第一次印刷 字数：378 000

定价：39.80 元

（如有印装质量问题，我社负责调换）

前　言

算法与数据结构主要研究数据的各种组织形式，以及建立在这些组织形式之上的各种运算的实现，它不仅为使用计算机语言进行程序设计提供了方法性的理论指导，还在一个更高的层次上总结了程序设计的常用方法与技巧，它是设计与开发大型应用程序的基础。

本书是专门为信息与计算科学专业（以下简称信息专业）学生学习算法与数据结构这门课程所编写的教材。

信息专业有着一些不同于计算机专业的特点。其一，所开设的计算机方向的课程门数和学时数，不如计算机专业多，在学习算法与数据结构这门课之前，只学习了 C/C++程序设计；其二，信息专业学生需要学习较多的数学课程，其专业方向主要有信息安全、人工智能、图像处理和应用数学等；其三，信息专业学生学习计算机方面课程的目的，主要是掌握计算机的编程方法，通过计算机编程去解决专业学习中所提出的问题。

为了编写一本适合于信息专业的算法与数据结构教材，编者在以下方面做了一些探索与尝试。

本书将数据结构定义为一个抽象数据类型（即 ADT），它由一个数据集合以及与该数据集合绑定在一起的一组运算所组成，类似于数学中的某种向量空间。采用面向过程的结构化程序设计方法，并用 C/C++程序来具体实现它。

借助实例讲授算法与数据结构的知识点，这种以任务驱动的知识学习法，既易于学生理解与掌握相关知识点，又锻炼了学生通过编程解决实际问题的能力。

对算法与数据结构中的许多结论，尽可能地给出证明，至少是计算机模拟证明，使计算机类课程的教学风格与数学类课程的教学风格保持一致。

对于一些复杂问题的求解，尽量利用图、表和文字进行详细分析，提出解决方案，然后进行数据结构与算法设计，最后给出问题的解。

通过一个实例，介绍了如何将抽象数据类型定义成类，将面向过程的程序转化成面向对象的程序。

对于信息专业的学生，仅靠计算机类课程的学习，要使编程能力得到实质性的提升，是有一定困难的。计算机编程源于数学中的数值计算，应该将计算机教学与数学教学融合在一起。即，让学生学会用计算机的方法去描述和解决数学问题，学会用数学方法去创建数学模型，构建适合计算机求解的算法或程序。

在本书及其配套的实验教材《算法与数据结构实验指导》中，配置了大量数学问题要求使用计算机编程求解的实例、习题、练习题和课程设计题。囿于教学计划和总学时限制，许多信息专业并未开设《算法设计与分析》这门课程。而一个高效的程序，最需要的恰恰是合理的数据结构和清晰高效的算法。考虑到《数据结构》与《算法设计与分析》具有同样的重要

性，虽有交叉但也有区别。因此，本书在主要介绍数据结构知识的同时，也兼顾了对一些典型算法的介绍，这一点在与其配套的实验教材中表现得尤为突出。

算法与数据结构的教学重点应该是面向设计的，学生学习这门课程的难点，并非是对数据结构知识的理解，而是怎样用程序代码去实现它，并将它应用于实际问题的求解。因此，本书中所有的数据结构均用完整的 C/C++程序来描述，程序代码注释详细，结构清晰一致，它们既是理解算法与数据结构的示例，也是学习 C/C++程序设计的范例。

本书内容深入浅出，配有大量的实例、图示、表格和文字注释，易教易学。书中习题及其配套的实验教材中的练习题、课程设计题，均与信息专业联系紧密。全书共分 13 章。

第 1～2 章数组和链表。考虑到信息专业学生对数组与链表这两种数据结构的掌握程度，特别增写了这两章。在这两章中，对数组与链表的相关知识，做了更深入的介绍。

第 3 章算法与数据结构绪论。这一章是全书的基础，它对于理解算法与数据结构的研究内容，相互关系以及对程序设计所起的基础作用，会有很大帮助。

第 4 章线性表。给出了线性表的定义，顺序表和链表的实现，还介绍了集合并交运算的实现方法。

第 5 章栈。给出了栈定义，顺序栈与链栈的实现以及栈应用实例。

第 6 章递归。递归是程序设计的一个常用方法，也是学生学习中的一个难点，本章从数据结构的角度对递归做了更深入全面的介绍。

第 7 章队列。给出了队列定义，顺序队列与链队列的实现以及队列应用实例。

第 8 章树。给出了一般树、二叉树的定义，给出了二叉树的相关结论、二叉树的表示法、二叉树的遍历，介绍了排序二叉树、堆和哈夫曼树，还介绍了多元树与森林的二叉树表示法，二叉树的应用实例。

第 9 章图。给出了图的定义，介绍了图的表示法、图的遍历、最小生成树、图的最短路径、AOV 和 AOE 网络、拓扑排序和关键路径。

第 10～11 章排序与搜索。排序与搜索是数据处理中最常见的运算，介绍了一些常用的排序算法与搜索算法，讨论了一些排序算法的平均时间复杂度，搜索算法的平均搜索长度。

第 12 章散列。给出了散列表的定义，介绍了散列技术、散列函数、冲突处理方法和哈希表的搜索。

第 13 章模板。增写这一章的目的，是为了探索面向过程的程序设计与面向对象的程序设计之间的关系。在本章中，通过一个实例，将一个抽象数据类型(即 ADT)，通过模板化、封装化，改造成了一个类，将一个面向过程的应用程序转化成了一个面向对象的应用程序。

本书在编写过程中的主要参考资料一并在参考文献中列出，对这些专家教授给予本书的帮助，作者表示衷心的感谢。鉴于作者的水平，疏漏与不妥之处在所难免，恳请专家和读者批评指正。

编　者

2015 年 8 月

目　录

第1章 数　　组

1.1 数组概念

数组(array)是用一片连续的内存空间,存储有限个数据类型相同的有序元素的集合。在不同的程序语言中,数组的声明会有所差异,但通常包含以下6种属性。

起始地址:数组名(或数组的首个元素)所在内存的地址。

维数:决定数组是一维的、二维的或者三维的。

数组类型:决定数组元素占有内存空间的字节数。

下标的上下界:数组元素在内存中所存放位置的上界与下界。

数组的元素个数:数组占用字节数除以数据类型字节数。

数组元素的随机访问:可通过下标随机访问数组中的任一元素。

1.1.1 一维数组

在C/C++语言中一维数组的定义方式如下:

数据类型 数组名称[元素个数];

例如,定义 int A[6],它的逻辑结构如图1.1所示。

元素值	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
下标	0	1	2	3	4	5

图1.1　一维数组的逻辑结构

由于整型数据在存储时,需占用4个字节,故在内存中分配了6×4=24个字节的存储空间,用来存放数组A[6],它的存储结构如图1.2所示。

图1.2　一维数组的存储结构

例1.1　对于整型数组A[4]={0,1,2,3},用程序展示它的6种属性。

解

```
#include<iostream>
using namespace std;
void main()
```

```
{   int A[4]={0,1,2,3};                          //定义含4个整型元素的一维数组并赋值
    cout<<"数组名的地址为:"<<A<<endl;
    cout<<"数据类型占字节数为:"<<sizeof(int)<<endl;
    cout<<"数组含元素个数为:"<<sizeof(A)/sizeof(int)<<endl;
    for(int i=0;i<4;i++)
        cout<<A[i]<<"["<<&A[i]<<"]"<<" ";//输出数组元素的值以及存储地址
    cout<<endl;
}
```

程序运行的结果如图 1.3 所示。

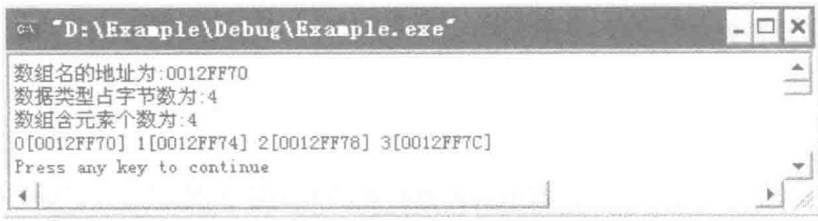

图 1.3 例 1.1 程序运行结果

从程序运行结果可以看出，整型数据占 4 个字节，C/C++用 8 位十六进制数表示地址(address 或 location)，数组名 A 与首元素 A[0]的地址均为 0012FF70，增加 4 字节后的下一个地址为 0012FF74，它是 A[1]的存储地址，再增加 4 字节后的下一个地址为 0012FF78，它是 A[2]的存储地址，再增加 4 字节后的下一个地址为 0012FF7C，它是 A[3]的存储地址。这表明，数组元素 A[0]，A[1]，A[2]，A[3]在内存中的存储位置是连续的。还必须特别指出的是，这段程序在不同计算机上运行时，数组元素所分配的地址会有些差异。

如果记数组的首地址为 a=0012FF70，数组 A 中元素的存储地址具有以下规律：

元素值	A[0]	A[1]	A[2]	A[3]
存储地址	a	a+1×4	a+2×4	a+3×4

一般地，若一维数组被声明为 A(0…n−1)，若 a 为数组 A 在内存中的起始地址，d 为每个数组元素占用内存的字节数，则数组元素 A(i)的存储地址为

$$Loc(A(i))=a+i\times d$$

称此式为数组元素的寻址公式。

若一维数组被声明为 A(1…n)，则数组元素 A(i)的存储地址为

$$Loc(A(i))=a+(i-1)\times d$$

例 1.2 给出以下程序运行的结果，叙述它的意义。

```
#include<iostream>
using namespace std;
void main()
{   int A[4];                                    //定义含4个整型元素的一维数组,但并未赋值
```

```
    cout<<"数组名的地址为:"<<A<<endl;
    cout<<"数组含元素个数为:"<<sizeof(A)/sizeof(int)<<endl;
    for(int i=0;i<4;i++)
        cout<<"A["<<i<<"]["<<&A[i]<<"]"<<" ";  //输出数组元素的存储地址
    cout<<endl;
}
```

解

程序运行的结果如图 1.4 所示。

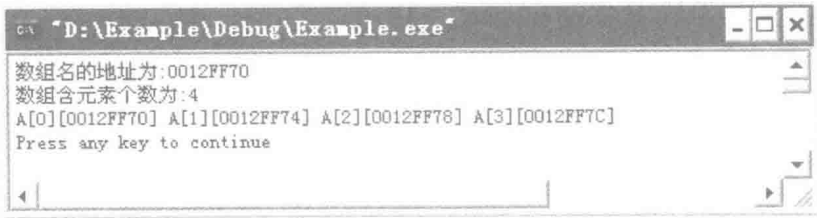

图 1.4 例 1.2 程序运行结果

由例 1.2 可知，程序中定义的数组，必须给出数据类型和元素个数这两个参数的值，可以不给数组元素赋值，编译系统会根据这两个参数值，为数组预先分配内存空间，这种内存分配方式称为静态内存分配。

1.1.2 二维数组

在 C/C++语言中二维数组的定义方式如下：

数据类型 数组名称[行数][列数];

例如

```
int A[2][3]
```

它的逻辑结构如图 1.5 所示。

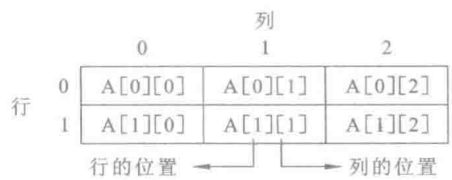

图 1.5 二维数组的逻辑结构

二维数组 A 在内存中却是用一个 2×3 的一维数组来存储的，其存储结构如图 1.6 所示。

元素值	A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]
下标	0	1	2	3	4	5

图 1.6 二维数组的存储结构

由图 1.6 可以看出，在存储二维数组时，分别将它的第 0 行，第 1 行，…，依次存入一个一维数组，这种将二维数组逐行存储到一个一维数组的方法，称为以行优先存储。

例 1.3 对于整型二维数组 A[2][3]={{0,1,2},{3,4,5}},输出该数组的逻辑结构,并说明它在内存中被存储为一个一维数组。

解

```
#include<iostream>
#include<iomanip>
using namespace std;
void main()
{   int A[2][3]={{0,1,2},{3,4,5}};                    //定义含 6 个元素整型二维数组并赋值
    cout<<"二维数组的逻辑结构如下"<<endl;
    for(int i=0;i<2;i++)
    {   for(int j=0;j<3;j++)
            cout<<setw(3)<<A[i][j];                   //输出 A[i][j]
        cout<<endl;                                   //换行
    }
    cout<<"二维数组的存储结构如下"<<endl;
    int *p;                                           //定义一个整型指针
    p=&A[0][0];                                       //p 指向二维数组的首元素地址
    for(i=1;i<=6;i++)
    {   cout<<setw(3)<<*p;                            //输出 p 所指元素的值
        p++;                                          //让 p 指向下一个元素
    }
    cout<<endl;
}
```

程序运行的结果如图 1.7 所示。

图 1.7 例 1.3 程序运行结果

设二维数组 A 被定义为 A(0…m−1,0…n−1),a 是二维数组 A 在内存中的起始地址,d 为数据元素占用的字节数,且 A 中元素采用行优先存储法。下面,我们来讨论二维数组元素 A(i,j)的存储地址的计算问题。

在 A(i,j)存入内存之前,A(i,j)前面的第 0 行至第 $i-1$ 行(共 $i\times n$ 个元素)已存储到内存,而第 i 行的第 0 列至第 $j-1$ 列(共 j 个元素)也已存储到内存,即有 $i\times n+j$ 个元素已存储到内存,故 A(i,j)在内存中的存储地址为

$$Loc(A(i,j))=a+i\times n\times d+j\times d$$

如果 A 被定义为 A(1…m,1…n),则 A(i,j)在内存中的存储地址应改为

Loc(A(i,j))=a+(i−1)×n×d+(j−1)×d

某些程序设计语言(如 MATLAB),其二维数组是以列优先方式存储在一维数组的。其中 A 被定义为 A(1…m,1…n),则 A(i,j)在内存中的存储地址应为

Loc(A(i,j))=a+(j−1)×m×d+(i−1)×d

例 1.4 若 A(3,3)的位置是 121,A(6,4)的位置是 159,每个元素占字节数 d=1,试求 A(4,5)的位置。

解 不妨设 A(1…m,1…n)且首地址为 a。

如果按以行优先方式存储,可列出方程组

$$a+(3-1)\times n\times 1+(3-1)\times 1=121,\quad a+(6-1)\times n\times 1+(4-1)\times 1=159$$

即

$$a+2n+2=121,\quad a+5n+3=159$$

两式相减得 $3n=37$,方程组无正整数解。

如果按以列优先方式存储,可列出方程组

$$a+(3-1)\times m\times 1+(3-1)\times 1=121,\quad a+(4-1)\times m\times 1+(6-1)\times 1=159$$

即

$$a+2m+2=121,\quad a+3m+5=159$$

两式相减得 $m=35$,$a=49$。

这表明,二维数组 A 是以列优先方式存储的,故

Loc(A(4,5))=49+(5−1)×35+(4−1)=192

1.1.3 三维数组

在 C/C++语言中,三维数组的定义方式如下:

数据类型 数组名称[第一维大小][第二维大小][第三维大小];

例如,int A[2][2][3],即 A(0…1,0…1,0…2),则数组 A 共含有 2×2×3=12 个整型元素,其逻辑结构像一本图书,第 1 维是书的页数,第 2 维是每页的行数,第 3 维是每行的列数,而元素 A(0,1,2)就是第 0 页 1 行 2 列处的一个"字"(如图 1.8 所示)。

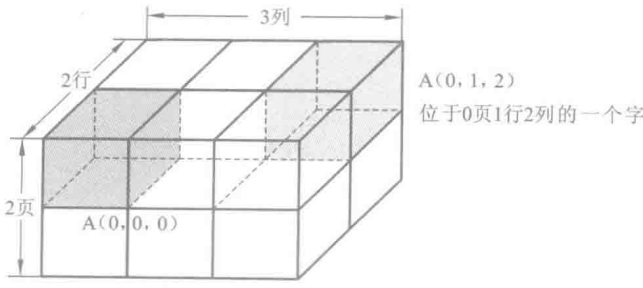

图 1.8 三维数组的逻辑结构

如果声明 A(0…u_1−1,0…u_2−1,0…u_3−1),将 A 中元素以行优先方式存储到一维数

组，是按“页行列”次序进行的。

设数组的起始位置为 a，数组元素占用的字节数为 d。我们来推导数组元素 A(i,j,k)的寻址公式。为通俗起见，我们不妨认为是推导“字”A(i,j,k)在“书”A 中的位置。

A(i,j,k)在第 i 页，它前面有 i 页，而每页有 $u_2 \times u_3$ 个字，共有 $i \times u_2 \times u_3$ 个字；它在第 j 行，它前面有 j 行，而每行有 u_3 个字，共有 $j \times u_3$ 个字；它在第 k 列，它前面有 k 列，而每列有 1 个字，共有 k 个字。总共有 $i \times u_2 \times u_3 + j \times u_3 + k$ 个字，故

$$Loc(A(i,j,k)) = a + i \times u_2 \times u_3 \times d + j \times u_3 \times d + k \times d$$

例 1.5 已知某书的印刷内容为

第 1 页			第 2 页		
111	112	113	211	212	213
121	122	123	221	221	223

在 C/C++中，用三维整型数组 A[2][2][3]存储该书内容，显示其逻辑结构，并说明该三维数组在内存中是以行优先方式存储在一个一维数组中。

解

```
#include<iostream>
#include<iomanip>
using namespace std;
void main()
{   int A[2][2][3];                    //定义 2 页 2 行 3 列的三维数组
    int i,j,k;                         //i 表示页,j 表示行,k 表示列
    //给三维数组赋值
    for(i=0;i<2;i++)
        for(j=0;j<2;j++)
            for(k=0;k<3;k++)
                A[i][j][k]=(i+1)*100+(j+1)*10+(k+1);
    cout<<"三维数组的逻辑结构如下"<<endl;
    for(i=0;i<2;i++)
    {   cout<<"第"<<i<<"页内容"<<endl;
        for(j=0;j<2;j++)
        {   for(k=0;k<3;k++)
                cout<<setw(4)<<A[i][j][k];
            cout<<endl;
        }
    }
    cout<<"三维数组的存储结构如下"<<endl;
    int *p;                            //定义整型指针
    p=&A[0][0][0];                     //让 p 指向 A 的首元素地址
```

```
    for(i=1;i<=12;i++)
    {   cout<<setw(4)<<*p;          //输出 p 所指元素的值
        p++;                        //让 p 指向下一个元素
    }
    cout<<endl;
}
```

程序运行的结果如图 1.9 所示。

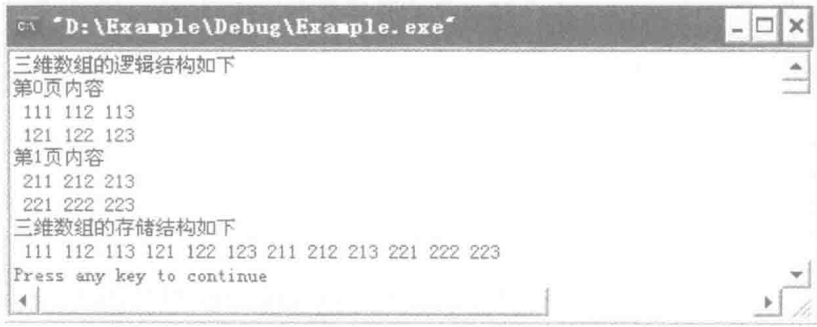

图 1.9 例 1.5 程序运行结果

在 MATLAB 语言中，数组 int A[2][2][3]被表示为 A(1…2,1…2,1…3)，其逻辑结构像一本古籍书，只是第 3 维是页数，第 2 维是每页的列数，第 1 维是每列的行数。而元素 A(1,2,3)就是第 3 页 2 列 1 行处的一个“字”(如图 1.10 所示)。

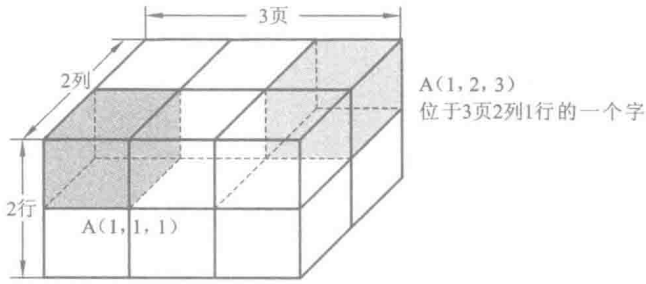

图 1.10 三维数组的逻辑结构

如果声明 A(1…u_1,1…u_2,1…u_3)，将 A 中元素以列优先方式存储到一维数组，是按“行列页”次序进行的。

如果数组的起始位置为 a，数组元素占用的字节数为 d。我们可类似地推导数组元素 A(i,j,k)的寻址公式。

A(i,j,k)在第 k 页，它前面有 $k-1$ 页，而每页有 $u_2 \times u_1$ 个字，共有$(k-1)\times u_2 \times u_1$ 个字；它在第 j 列，它前面有 $j-1$ 列，而每列有 u_1 个字，共有$(j-1)\times u_1$ 个字；它在第 i 行，它前面有 $i-1$ 行，而每行有 1 个字，共有 $i-1$ 个字。总共有$(k-1)\times u_2 \times u_1+(j-1)\times u_1+(i-1)$个字，故

$$Loc(A(i,j,k))=a+(k-1)\times u_2 \times u_1 \times d+(j-1)\times u_1 \times d+(i-1)\times d$$

1.2 矩阵及其运算

矩阵(matrix)是一个数学概念，矩阵 $\boldsymbol{A}_{m\times n}=(a_{ij})_{m\times n}$ 的展开形式为

$$\boldsymbol{A}_{m\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

矩阵 $\boldsymbol{A}_{m\times n}$ 实际上是由 $m\times n$ 个数排列而成的 m 行、n 列的数阵。显然，它也可以看成二维数组 A(1…m,1…n)。

而关于矩阵的加法、乘法和转置运算，均可以通过二维数组来实现。

1.2.1 矩阵相加

在 C/C++语言中，矩阵 $\boldsymbol{A}_{m\times n}$ 可以用二维数组 A[m][n]来存储，元素 a_{ij} 可用 A[i][j]来表示；还可以定义一个指针变量 * p=&.A[0][0]，用 p[i * n+j]来表示。

两个同型矩阵相加，就是对应位置的元素相加。由于矩阵元素具有两种下标表示法，因此，可以写出两种不同风格的两矩阵相加程序。

例 1.6 求以下矩阵 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的相加结果 $\boldsymbol{C}$。

$$\boldsymbol{A}=\begin{bmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \end{bmatrix},\quad \boldsymbol{B}=\begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \end{bmatrix}$$

解

(1) 矩阵元素采用二维下标表示法。

```
#include<iostream>
using namespace std;
//矩阵输出
void Output(int E[2][3],int m,int n)
{   for(int i=0;i<m;i++)
    {   for (int j=0;j<n;j++)
            cout<<E[i][j]<<"\t";                    //输出 E 的第 i 行
        cout<<endl;                                 //换行
    }
}
//矩阵加法
void MatrixAdd(int A[2][3],int B[2][3],int C[2][3],int m,int n)
{   for(int i=0;i<m;i++)
        for(int j=0;j<n;j++)
            C[i][j]=A[i][j]+B[i][j];                //对应位置上的元素相加
}
//主函数
```

```
void main()
{ int A[2][3]={{1,3,5},{7,9,11}};                    //定义矩阵 A
  int B[2][3]={{9,8,7},{6,5,4}};                     //定义矩阵 B
  int C[2][3];                                       //定义矩阵 C
  cout<<"矩阵 A"<<endl;
  Output(A,2,3);                                     //输出矩阵 A
  cout<<"矩阵 B"<<endl;
  Output(B,2,3);                                     //输出矩阵 B
  MatrixAdd(A,B,C,2,3);                              //C=A+B
  cout<<"矩阵 C=A+B"<<endl;
  Output(C,2,3);                                     //输出矩阵 C
}
```

(2) 矩阵元素采用一维下标表示法。

```
#include<iostream>
using namespace std;
//矩阵输出
void Output(int *E,int m,int n)
{  //输出 m 行 n 列矩阵 E
    for(int i=0;i<m;i++)
    {   for(int j=0;j<n;j++)
            cout<<E[i*n+j]<<"\t";                    //输出 E 的第 i 行
        cout<<endl;                                  //换行
    }
}
//矩阵加法
void MatrixAdd(int *A,int *B,int *C,int m,int n)
{  //m 行 n 列矩阵 A,B 相加并赋值给矩阵 C
     for(int i=0;i<m;i++)
         for(int j=0;j<n;j++)
             C[i*n+j]=A[i*n+j]+B[i*n+j];             //对应位置上的元素相加
   }
//主函数
void main()
{ int A[2][3]={{1,3,5},{7,9,11}};                    //定义矩阵 A
  int B[2][3]={{9,8,7},{6,5,4}};                     //定义矩阵 B
  int C[2][3];                                       //定义矩阵 C
  cout<<"矩阵 A"<<endl;
  Output(&A[0][0],2,3);                              //输出矩阵 A
  cout<<"矩阵 B"<<endl;
  Output(&B[0][0],2,3);                              //输出矩阵 B
```