第 5 章　数组和广义表

数组和广义表可以看成线性表的一种推广，即表中的数据元素本身也可以是线性结构。本章将讨论数组的基本概念和存储结构、特殊矩阵和稀疏矩阵的压缩存储以及广义表的定义和存储结构。

5.1　数　　组

5.1.1　数组的基本概念

数组是由 $n(n\geqslant 1)$ 个相同类型的数据元素构成的有限序列，其逻辑表示如下：

$$a=(a_0,a_1,\cdots,a_{n-1})$$

若 $a_i(i=0,1,\cdots,n-1)$ 是简单元素，则 a 是一维数组；当一维数组的每个元素 a_i 本身又是一个一维数组时，则一维数组扩充为二维数组。同理，二维数组的每个元素本身又是一个一维数组时，则二维数组扩充为三维数组。以此类推，若 a_i 是 $n-1$ 维数组，则 a 是 n 维数组。

在 n 维数组中，每个元素受 $n(n\geqslant 1)$ 个线性关系的约束，它在 n 个线性关系中的序号 $i_1,i_2,\cdots,i_n$，称为该元素的下标。如果数组名为 a，则记下标为 $i_1,i_2,\cdots,i_n$ 的元素为 $a_{i_1i_2\cdots i_n}$。

数组可以看成线性表的推广，数组的特点是数据元素本身可以具有某种结构，但属于同一类型。例如，一维数组可以看作一个线性表，二维数组可以看作数据元素是线性表的线性表。如图 5-1 中所示的二维数组，可以看成一个线性表

$$A=(a_0,a_1,\cdots,a_p)\quad (p=m-1\text{ 或 }n-1)$$

其中每个数据元素是一个列向量形式的线性表，如图 5-1(b)所示，或者是一个行向量形式的线性表，如图 5-1(c)所示。

$$A=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

（a）二维数组

$$A=\left[\begin{bmatrix} a_{00} \\ a_{10} \\ \vdots \\ a_{m-1,0} \end{bmatrix}\begin{bmatrix} a_{01} \\ a_{11} \\ \vdots \\ a_{m-1,1} \end{bmatrix}\cdots\begin{bmatrix} a_{0,n-1} \\ a_{1,n-1} \\ \vdots \\ a_{m-1,n-1} \end{bmatrix}\right]$$

（b）数据元素是列向量

$$A=\left((a_{00}a_{01}\cdots a_{0,n-1}),(a_{10}a_{11}\cdots a_{1,n-1}),\cdots,(a_{m-1,0}a_{m-1,1}\cdots a_{m-1,n-1})\right)$$

（c）数据元素是行向量

图 5-1　二维数组示例

数组一旦被定义，它的维数和维界就不再改变。因此，数组除了初始化和销毁之外，通常只有以下两种操作。

(1) 读操作：给定一组下标，读取相应的数据元素。

(2) 写操作：给定一组下标，存储或修改相应的数据元素。

下面给出数组的抽象数据类型定义。

```
ADT Array
{
    数据对象：
        D={a_{j1 j2 … jn} | j_i=0,1,…b_{i-1}, i=1,2,…n, b_i 为第 i 维的长度}
    数据关系：
        R={r_1, r_2, …, r_n}
        r_i={<a_{j1 … ji … jn}, a_{j1 … ji+1 … jn}> | 0≤j_k≤b_{k-1}, 1≤k≤n 且 k≠i, 0≤j_i≤b_i-2, i=2,3,…,
        n}
    基本操作：
        InitArrary()
        操作结果：初始化数组，即为数组分配空间。
        DistroyArrary()
        初始条件：数组已存在。
        操作结果：销毁数组。
        Value(&e, index1, index2, …, indexn)
        初始条件：n 维数组已存在。
        操作结果：若下标不越界，则将所指定的数组元素的值赋值给 e。
        Assign(e, index1, index2, …, indexn)
        初始条件：n 维数组已存在。
        操作结果：若下标不越界，则将 e 的值赋值给所指定的数组元素。
}
```

5.1.2 数组的存储结构

由于很少对数组进行插入和删除操作，也就是说，一旦建立了数组，其元素个数和元素之间的关系就不再发生变动，因此，采用顺序存储结构表示数组比较合适。

1. 一维数组的存储结构

对于一维数组$(a_0, a_1, \cdots, a_{n-1})$，按元素顺序存储到一块地址连续的内存单元中。假设第一个元素 a_0 的存储地址用 $\mathrm{LOC}(a_0)$ 表示，每个元素占用 L 个存储单元，则任一数组元素 a_i 的存储地址 $\mathrm{LOC}(a_i)$ 都可由式(5-1)求出：

$$\mathrm{LOC}(a_i) = \mathrm{LOC}(a_0) + i \times L \quad (1 \leqslant i \leqslant n-1) \tag{5-1}$$

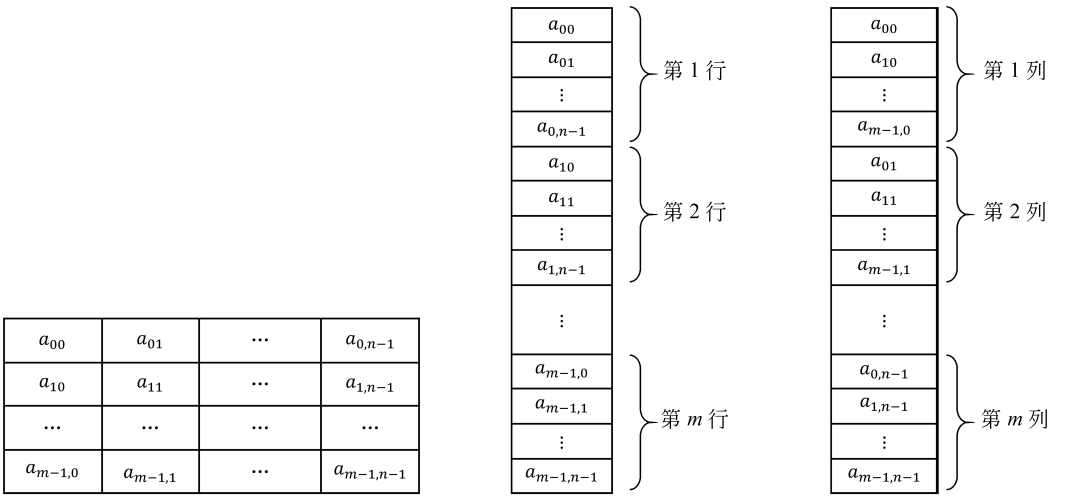
2. 多维数组的存储结构

由于内存单元是一维的结构，而多维数组是多维的结构，因此，采用顺序存储结构存储多维数组时需要将多维结构映射到一维结构。常用的映射方法有两种：以行序为主序(即按行优先)和以列序为主序(即按列优先)。C++ 中的数组就是按行优先存储的。

对于二维数组，按行优先存储的基本思想是：先行后列，先存储行号较小的元素，行号相同者先存储列号较小的元素，如图 5-2(b)所示。假设第一个元素 a_{00} 的存储地址用 $\mathrm{LOC}(a_{00})$ 表示，每个元素占用 L 个存储单元，则二维数组中任一元素 a_{ij} 的存储地址 $\mathrm{LOC}(a_{ij})$ 可由式 5-2 确定。

$$\mathrm{LOC}(a_{ij}) = \mathrm{LOC}(a_{00}) + (n \times i + j) \times L \tag{5-2}$$

二维数组的存储



（a）二维数组　　（b）按行优先存储　　（c）按列优先存储

图 5-2　二维数组的两种存储方式

按列优先存储的基本思想是：先列后行，先存储列号较小的元素，列号相同者先存储行号较小的元素，如图 5-2(c)所示。二维数组中任一元素 a_{ij} 的存储地址 LOC(a_{ij})可由式(5-3)确定。

$$\mathrm{LOC}(a_{ij}) = \mathrm{LOC}(a_{00}) + (m \times j + i) \times L \tag{5-3}$$

对于三维数组和更高维数组，按行优先存储的思想是：最右边的下标先变化，即最右下标从小到大，循环一遍后，右边第二个下标再变化，……，以此类推，最后是最左下标。按列优先存储的思想是：最左边的下标先变化，即最左下标从小到大循环一遍后，左边第二个下标再变化，……，依此类推，最后是最右下标。

由以上讨论可知，数组无论是按行优先存储，还是按列优先存储，都可以在 $O(1)$的时间内计算出指定元素的存储地址，因此，存取数组中任一元素的时间都相等，即数组是一种随机存取结构。

5.2　矩　　阵

矩阵是很多科学与工程计算问题中研究的数学对象，在高级语言程序设计中，矩阵用二维数组表示。但是，在数值分析中经常出现一些阶数很高的矩阵，同时矩阵中有很多相同的元素或零元素。为了节省存储空间，可以对这类矩阵进行压缩存储，即为多个值相同的元素只分配一个存储空间，对零元素不分配存储空间。

5.2.1　特殊矩阵的压缩存储

特殊矩阵是指值相同的元素或零元素在矩阵中的分布有一定规律的矩阵。特殊矩阵的主要形式有对称矩阵、三角矩阵和对角矩阵，它们都是方阵，即行数和列数相同。

1. 对称矩阵

若一个 n 阶方阵 $\boldsymbol{A}$ 中的元素满足下述性质：

$$a_{ij}=a_{ji}\quad 1\leqslant i,j\leqslant n$$

则称 $\boldsymbol{A}$ 为 n 阶**对称矩阵**。例如，图 5-3 所示为一个 5 阶对称矩阵。

对称矩阵中的元素是按主对角线对称的，即上三角部分和下三角部分中的对应元素相等，因此，在存储时可以只存储主对角线加上三角部分的元素，或主对角线加下三角部分的元素。这样，一个 n 阶对称矩阵只需要 $n(n+1)/2$ 个存储单元，节约了近一半的存储空间。

$$\boldsymbol{A}=\begin{bmatrix}3&6&4&7&8\\6&2&8&4&2\\4&8&1&6&9\\7&4&6&0&5\\8&2&9&5&7\end{bmatrix}$$

图 5-3　对称矩阵

假设采用按行优先方式，将主对角线加下三角部分的元素存储到一个具有 $n(n+1)/2$ 个存储单元的一维数组 sa 中，如图 5-4 所示。矩阵 $\boldsymbol{A}$ 中的元素 a_{ij} 存储在数组元素 sa[k]中，如何根据 i 和 j 确定下标 k 的值？

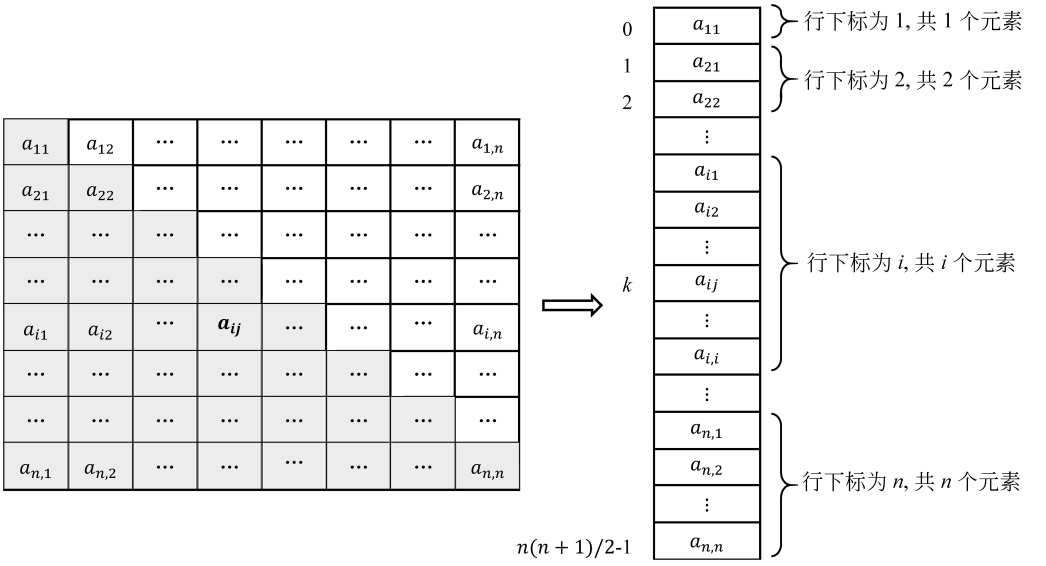


图 5-4　对称矩阵的压缩存储

(1) 若 a_{ij} 是矩阵 $\boldsymbol{A}$ 中主对角线或下三角部分的元素，则有 $i\geqslant j$。元素 a_{ij} 的前面共存储了 $i-1$ 行，第 1 行有 1 个元素，第 2 行有 2 个元素，……，第 $i-1$ 行有 $i-1$ 个元素，这 $i-1$ 行共有 $1+2+\cdots+i-1=i(i-1)/2$ 个元素；第 i 行中，元素 a_{ij} 的前面共存储 $j-1$ 个元素。因此，元素 a_{ij} 的前面共存储了 $i(i-1)/2+j-1$ 个元素，a_{ij} 在数组 sa 中的下标 k 为

$$k=\frac{i(i-1)}{2}+j-1$$

(2) 若 a_{ij} 是矩阵 $\boldsymbol{A}$ 中上三角部分的元素，则有 $i<j$。a_{ij} 的值等于 a_{ji}，元素 a_{ji} 是矩阵下三角部分的元素，它存放在下标为 $j(j-1)/2+i-1$ 的位置，即

$$k=\frac{j(j-1)}{2}+i-1$$

综合两种情况，得到 k 与 i、j 的关系如下：

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1 & i\geqslant j\\ \dfrac{j(j-1)}{2}+i-1 & i<j\end{cases} \tag{5-4}$$

对称矩阵的类模板的声明及实现如下所示。

```
//类模板的声明
template<typename T>
class SymmetricMatrix
{
protected:
    T * sa;                                          //存储矩阵元素
    int n;                                           //对称矩阵阶数
public:
    SymmetricMatrix(int n);                          //构造一个 n 阶对称矩阵
    ~SymmetricMatrix();                              //析构函数
    T &operator()(int i, int j);                     //重载()运算符
};
//类模板的实现
template <typename T>
SymmetricMatrix<T>::SymmetricMatrix(int n)           //构造函数,n 为矩阵的阶数
{
    if (n<1)
        throw "阶数无效!";
    this->n=n;
    sa=new T[n * (n+1)/2];                           //为一维数组 sa 分配存储空间
}
template <typename T>
SymmetricMatrix<T>::~SymmetricMatrix()               //析构函数
{
    delete []sa;
}
template <typename T>
T &SymmetricMatrix<T>::operator()(int i, int j)      //重载()运算符
{
    if (i<1 || i>n || j<1 || j>n)
        throw "下标越界!";
    if (i>=j)                                        //元素在主对角线或下三角中
        return sa[i * (i-1)/2+j-1];
    else                                             //元素在上三角中
        return sa[j * (j-1)/2+i-1];
}
```

定义对称矩阵类 SymmetricMatrix 后,在程序中就可以使用它定义对称矩阵的对象,通过括号运算符访问矩阵中的数据元素。

```
#include <iostream>
```

```
#include "symmetric_matrix.h"
using namespace std;
void main()
{
    try
    {
        int i, j, n, tmp;
        cout<<"请输入矩阵的阶数: "<<endl;
        cin>>n;
        SymmetricMatrix<int>a(n);                    //定义一个 n 阶对称矩阵
        for (i=1; i<=n; i++)                         //输入数据
        {
            cout<<"请输入矩阵的第"<<i<<"行数据: "<<endl;
            for (j=1; j<=n; j++)
            {
                cin>>tmp;
                a(i, j)=tmp;
            }
        }
        for (i=1; i<=n; i++)                         //显示对称矩阵
        {
            for (j=1; j<=n; j++)
                cout<<a(i, j)<<"\t";
            cout<<endl;
        }
    }
    catch (char * error)
    {
        cout<<error<<endl;
    }
}
```

2. 三角矩阵

三角矩阵有上三角矩阵和下三角矩阵两种。**上三角矩阵**是指矩阵下三角部分中的元素均为常数 c 的 n 阶方阵。**下三角矩阵**是指矩阵上三角部分中的元素均为常数 c 的 n 阶方阵，有时 $c=0$。例如，图 5-5 中的 $\boldsymbol{A}_1$ 为上三角矩阵，$\boldsymbol{A}_2$ 为下三角矩阵。

$$\boldsymbol{A}_1=\begin{bmatrix} 6 & 3 & 7 & 2 & 8 \\ 0 & 4 & 1 & 7 & 5 \\ 0 & 0 & 2 & 6 & 7 \\ 0 & 0 & 0 & 1 & 9 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix} \qquad \boldsymbol{A}_2=\begin{bmatrix} 9 & 1 & 1 & 1 & 1 \\ 8 & 5 & 1 & 1 & 1 \\ 6 & 9 & 3 & 1 & 1 \\ 3 & 7 & 5 & 6 & 1 \\ 5 & 4 & 8 & 2 & 7 \end{bmatrix}$$

（a）上三角矩阵　　（b）下三角矩阵

图 5-5　三角矩阵

对上三角矩阵进行压缩存储时，除了存储主对角线和上三角中的元素外，再加一个存储常数 c 的空间。因此，上三角矩阵可以用一个具有 $n(n+1)/2+1$ 个存储单元的一维数组存储，常数 c 存储在数组最后的位置。

若 a_{ij} 是矩阵 $\boldsymbol{A}$ 中主对角线或上三角部分的元素，则有 $i \leqslant j$。在按行优先的存储方式下，元素 a_{ij} 的前面共存储了 $i-1$ 行，第 1 行有 n 个元素，第 2 行有 $n-1$ 个元素，……，第 $i-1$ 行有 $n-i+2$ 个元素，这 $i-1$ 行共有：

$$n+(n-1)+\cdots+(n-i+2)=\frac{(i-1)(2n-i+2)}{2}$$

个元素；第 i 行中，元素 a_{ij} 的前面共存储了 $j-i$ 个元素。因此，元素 a_{ij} 的前面共存储了 $(i-1)(2n-i+2)/2+j-i$ 个元素，所以，元素 a_{ij} 在数组中的下标 k 为

$$k=\frac{(i-1)(2n-i+2)}{2}+j-i$$

若 a_{ij} 是矩阵 $\boldsymbol{A}$ 中下三角部分的元素，则有 $i>j$，其值为常数 c，存储在数组的最后一个位置，下标为 $n(n+1)/2$。

综上，得到 k 与 i、j 的关系如下：

$$k=\begin{cases}\dfrac{(i-1)(2n-i+2)}{2}+j-i & i \leqslant j \\ \dfrac{n(n+1)}{2} & i>j\end{cases} \tag{5-5}$$

对下三角矩阵进行压缩存储时，除了存储主对角线和下三角中的元素外，再加一个存储常数 c 的空间。同上三角矩阵一样，下三角矩阵也可以用一个具有 $n(n+1)/2+1$ 个存储单元的一维数组存储，常数 c 存储在数组的最后位置。

采用类似对称矩阵的推导过程，得到 k 与 i、j 的关系如下：

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1 & i \geqslant j \\ \dfrac{n(n+1)}{2} & i<j\end{cases} \tag{5-6}$$

3. 对角矩阵

在对角矩阵中，所有非零元素都集中在以主对角线为中心的带状区域，除了主对角线和若干条次对角线的元素外，其他元素都为零。例如，图 5-6 所示为一个对角矩阵。对这种矩阵，也可按某个原则（以行为主，或以对角线的顺序）将其压缩存储到一维数组中。

$$\boldsymbol{A}=\begin{bmatrix}5 & 8 & 0 & 0 & 0\\ 6 & 4 & 1 & 0 & 0\\ 0 & 7 & 2 & 6 & 0\\ 0 & 0 & 3 & 1 & 9\\ 0 & 0 & 0 & 2 & 4\end{bmatrix}$$

图 5-6　对角矩阵

以上讨论的对称矩阵、三角矩阵和对角矩阵的压缩存储方法是把有一定分布规律的、值相同的数据元素压缩存储到一个连续的存储空间中，这样的压缩存储只需在算法中按公式做映射即可实现特殊矩阵元素的随机存取。

5.2.2　稀疏矩阵的压缩存储

如果一个矩阵中的非零元素个数远远小于矩阵元素的总数，且非零元素的分布无一定规律，则称之为**稀疏矩阵**。在稀疏矩阵和稠密矩阵之间没有一个精确的界限，可以用稀疏因

子描述矩阵的稀疏程度。设一个 m 行 n 列的矩阵中有 t 个非零元素，则稀疏因子 $\delta = t/(m \times n)$。通常，在 $\delta < 0.5$ 时，就可以认为矩阵是稀疏的。

当稀疏矩阵的阶很高时，其中零元素会很多，如果用一个二维数组直接存储稀疏矩阵，将存储大量零元素，造成存储空间的浪费。但如果不存储零元素，而只存储非零元素，元素的存储顺序又不能反映它们的逻辑关系，因此需要显示地指出每个元素在原矩阵中的逻辑位置。一种直观、常用的方法是对每个非零元素用三元组（行号，列号，元素值）表示。三元组的结构定义如下：

```
template<typename T>
struct Element
{
    int row, col;                                    //非零元素的行下标与列下标
    T value;                                         //非零元素的值
};
```

将稀疏矩阵的非零元素对应的三元组所构成的集合，按行优先排成一个线性表，称为三元组表，则稀疏矩阵的压缩存储转化为三元组表的存储。例如，图 5-7 所示稀疏矩阵 $\boldsymbol{A}$ 的三元组表为((1,2,6),(2,4,3),(3,5,4),(5,1,1),(5,3,5),(6,4,7))。

稀疏矩阵采用三元组表存储，一定程度上节省了存储空间，但也丧失了随机存取特性。

1. 三元组顺序表

以顺序表存储三元组表，可得到稀疏矩阵的顺序存储结构——三元组顺序表。例如，图 5-7 所示稀疏矩阵的三元组顺序表如图 5-8 所示。

$$\boldsymbol{A}=\begin{bmatrix} 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 \end{bmatrix}$$

图 5-7　稀疏矩阵

下标	row	col	vlaue
0	1	2	6
1	2	4	3
2	3	5	4
3	5	1	1
4	5	3	5
5	6	4	7
maxSize−1	空闲	空闲	空闲

图 5-8　稀疏矩阵 $\boldsymbol{A}$ 的三元组顺序表

三元组顺序表的类模板定义及实现如下，成员变量实现三元组顺序表的存储结构，其中数组 elems 用于存储非零元素的三元组，maxSize 为数组的长度，即稀疏矩阵中非零元素的最大个数，rows 和 cols 分别存储稀疏矩阵的行数和列数，num 存储稀疏矩阵中非零元素的实际个数；成员函数实现稀疏矩阵的基本操作，包括返回稀疏矩阵的行数或列数，返回或设置元素的值等。

```
//三元组顺序表类模板的定义
template<typename T>
```

```
class SparseMatrix
{
protected:
    Element<T> * elems;                                    //存储稀疏矩阵的三元组表
    int rows, cols, num;                                   //行数、列数及非零元素的个数
    int maxSize;                                           //非零元素的最大个数
public:
    SparseMatrix(int rs, int cs, int size);                //构造函数
    ~SparseMatrix();                                       //析构函数
    int GetRows();                                         //返回稀疏矩阵的行数
    int GetColumns();                                      //返回稀疏矩阵的列数
    int GetNum() ;                                         //返回稀疏矩阵非零元素的个数
    bool SetElem(int r, int c, T v);                       //设置指定位置的元素值
    bool GetElem(int r, int c, T &v);                      //返回指定位置的元素值
};
//三元组顺序表类模板的实现
template <typename T>
SparseMatrix<T>::SparseMatrix(int rs, int cs, int size)
//构造函数,rs、cs、size 分别表示行数、列数和非零元素的最大个数
{
    if (rs<1 || cs<1)
        throw "行数或列数无效!";
    maxSize=size;
    rows=rs;
    cols=cs;
    num=0;
    elems=new Element<T>[maxSize];
}
template <typename T>
SparseMatrix<T>::~SparseMatrix()                           //析构函数
{
    delete []elems;
}
template <typename T>
int SparseMatrix<T>::GetRows()                             //返回稀疏矩阵的行数
{
    return rows;
}
template <typename T>                                      //返回稀疏矩阵的列数
int SparseMatrix<T>::GetColums()
{
    return cols;
```

```
}
template <typename T>                                    //返回非零元素的个数
int SparseMatrix<T>::GetNum()
{
    return num;
}
template <typename T>
bool SparseMatrix<T>::SetElem(int r, int c, T v)
{
    int i=0, j;
    if (r<1 || r>rows || c<1 || c>cols)                  //下标越界
        return false;
    while (i<num && r<elems[i].row)                      //查找第 r 行的第一个非零元素
        i++;
    while (i<num && r==elems[i].row && c<elems[i].col)//查找第 r 行第 c 列的元素
        i++;
    if (elems[i].row==r && elems[i].col==c)              //找到
    {
        if (v!=0)                                        //若 v≠0,则修改元素值
            elems[i].value=v;
        else                                             //若 v=0,则删除三元组
        {
            for (j=i+1; j<num; j++)                      //前移元素
                elems[j-1]=elems[j];
            num--;                                       //非零元素个数减 1
        }
        return true;
    }
    else if (v!=0)                                      //若未找到,则将元素插入三元组表
    {
        if (num>=maxSize)
            return false;
        else
        {
            for (j=num-1; j>=i; j--)                     //后移元素
                elems[j+1]=elems[j];
            elems[i].row=r;                              //插入元素
            elems[i].col=c;
            elems[i].value=v;
            num++;                                       //非零元素个数增 1
        }
    }
```