第 5 章　数组和广义表

本章讨论的是数组和广义表这两种数据结构。它们都可看成是线性表的扩展，这是因为表中的数据元素也可以具有线性结构。

5.1　数　　组

5.1.1　数组的基本概念

数组是一种十分常用的结构，现在几乎所有程序设计语言都直接支持数组类型。在对数组进行操作前，要对元素进行定位。本节的主要内容是给出数组的定义及其存储映射的方法。

数组由一组相同类型的数据元素构成。可借助线性表的概念递归进行定义。

数组是一个元素可直接按序号寻址的线性表：

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

若 $a_i(i=0,1,\cdots,m-1)$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，当 a_i 是一个二维数组时，则二维数组扩充为三维数组。以此类推，若 a_i 是 $k-1$ 维数组，则 a 是 k 维数组。

可以看出，在 n 维数组中，每个元素受 n 个线性关系的约束($n\geqslant 1$)，若它在第 1～n 个线性关系中的序号分别为 $i_1,i_2,\cdots,i_n$，则称它的下标为 $i_1,i_2,\cdots,i_n$。如果数组名为 a，则记下标为 $i_1,i_2,\cdots,i_n$ 的元素为 $a_{i_1,i_2,\cdots,i_n}$。

从上面的定义可以看出，如果一个 $n(n>0)$ 维数组的第 i 维长度为 b_i，则此数组中共含有 $\prod_{i=1}^{n} b_i$ 个数据元素，每个元素都受 n 个关系的约束，就其单个关系而言，这 n 个关系都是线性关系。

数组的基础操作如下：

```
ElemType &operator()(int sub0, …)
```

初始条件：数组已存在。

操作结果：重载函数运算符。

5.1.2　数组的顺序表

由于很少在数组中进行插入和删除操作，所以数组的存储通常采用顺序存储方式。设 n 维数组共有 $m\left(=\prod_{i=1}^{n} b_i\right)$ 个元素，数组存储的首地址为 base，数组中的每个元素需要 size 个存储单元，则整个数组共需 m · size 个存储单元。为了存取数组中某个特定下标的元素，必须确定下标为 i_1，i_2，…，i_n 的元素的存储位置。实际上就是把下标 i_1，i_2，…，i_n 映射到 $[0,m-1]$ 中的某个数 Map(i_1，i_2，…，i_n)，使得该下标所对应的元素值存储在以下位置：

$$\text{Loc}(i_1,i_2,\cdots,i_n)=\text{base}+\text{Map}(i_1,i_2,\cdots,i_n)\cdot\text{size}$$

用 $\text{Loc}(i_1,i_2,\cdots,i_n)$表示下标为 $i_1,i_2,\cdots,i_n$ 的数组元素的存储地址。可见，如果已经知道数组的首地址，要确定其他元素的存储位置，只需求出 $\text{Map}(i_1,i_2,\cdots,i_n)$即可。下面讨论 $\text{Map}(i_1,i_2,\cdots,i_n)$的计算。在后面的例子中，都是用 C++ 中表示数组的方式来描述数组及其元素。

对于一维数组，$\text{Map}(i_1)=i_1$。

对于二维数组，各元素可按图 5.1 所示的表格形式进行排列。第 1 维下标相同的元素位于同一行，第 2 维下标相同的元素位于同一列。

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]	a[2][5]
a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]	a[3][5]
a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]	a[4][5]

图 5.1　数组 a[5][6]的元素列表

对于图 5.1 中的元素，从第 1 行开始，依次对每一行中的每个元素从左至右进行连续编号，即可得到图 5.2(a)所示的映射结果。这种按行把二维数组中的元素位置映射为[0，m－1]中的某个数的方式称为行优先映射。C++ 中就采用行优先映射模式。图 5.2(b)中给出了另外一种映射模式，把所有元素按列的次序进行连续编号，称为列优先映射。

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

(a) 行优先映射

0	5	10	15	20	25
1	6	11	16	21	26
2	7	12	17	22	27
3	8	13	18	23	28
4	9	14	19	24	29

(b) 列优先映射

图 5.2　数组 a[5][6]的映射

可以看出，一个 b_1 行 b_2 列的二维数组 $a[b_1][b_2]$的行优先映射所对应的映射函数为

$$\text{Map}(i_1,i_2)=i_1b_2+i_2$$

读者可用图 5.1 中的 5 行 6 列的数组来验证上述的行优先映射函数。因为列数为 6，所以映射公式为 $\text{Map}(i_1,i_2)=6i_1+i_2$，从而有 $\text{Map}(2,3)=6\times2+3=15$，$\text{Map}(3,5)=6\times3+5=23$，与图 5.2(a)中所给出的编号完全相同。

可以扩充上述行优先映射模式，得到二维以上数组的行优先映射函数。在二维数组的行优先次序中，首先列出所有第一个下标为 0 的元素，然后是第一个下标为 1 的元素……第一个下标相同的所有元素按其第二个下标的递增次序排列。以此类推，对于三维数组，首先列出所有第一个下标为 0 的元素，然后是第一个下标为 1 的元素……第一个下标相同的所有元素按其第二个下标的递增次序排列，前两个下标相同的所有元素按其第三个下标的递增次序排列。例如数组 a[4][2][4]中的元素，按行优先次序排列为

a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][0][3]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[0][1][3]
a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][0][3]	a[1][1][0]	a[1][1][1]	a[1][1][2]	a[1][1][3]
a[2][0][0]	a[2][0][1]	a[2][0][2]	a[2][0][3]	a[2][1][0]	a[2][1][1]	a[2][1][2]	a[2][1][3]

$a[3][0][0]$ $a[3][0][1]$ $a[3][0][2]$ $a[3][0][3]$ $a[3][1][0]$ $a[3][1][1]$ $a[3][1][2]$ $a[3][1][3]$

三维数组 $a[b_1][b_2][b_3]$的行优先映射函数为

$$\mathrm{Map}(i_1,\ i_2,\ i_3)=i_1b_2b_3+i_2b_3+i_3$$

类推可得，n 维数组 $a[b_1][b_2]\cdots[b_n]$的行优先映射函数为

$$\mathrm{Map}(i_1,i_2,\cdots,i_n)=i_1b_2b_3\cdots b_n+i_2b_3\cdots b_n+\cdots+i_{n-1}b_n+i_n=\sum_{j=1}^{n-1}i_j\prod_{k=j+1}^{n}b_k+i_n=\sum_{j=1}^{n}c_ji_j$$

进一步可得如下公式：

$$\mathrm{Loc}(i_1,i_2,\cdots,i_n)=\mathrm{base}+(i_1b_2b_3\cdots b_n+i_2b_3\cdots b_n+\cdots+i_{n-1}b_n+i_n)\mathrm{size}$$

$$=\mathrm{base}+\sum_{j=1}^{n}c_ji_j\,\mathrm{size}$$

其中 $c_n=1,c_{j-1}=b_jc_j,1<j\leqslant n$。

在有些语言中，数组各维的下标范围不一定为$[0,b_j-1](j=1,2,\cdots,n)$，而是某个闭区间$[l_j,h_j]$内的整数，则其行优先和列优先映射函数要随之改变。设 n 维数组的第 k 维的下标范围是$[l_k,h_k]$，记

$$d_k=h_k-l_k+1,\quad k=1,2,\cdots,n$$

显然，d_k 为第 k 维的长度(元素个数)，则此时有

$$\mathrm{Map}(i_1,i_2,\cdots,i_n)=(i_1-l_1)d_2d_3\cdots d_n+(i_2-l_2)d_3\cdots d_n+\cdots+(i_{n-1}-l_{n-1})d_n+(i_n-l_n)$$

$$=\sum_{j=1}^{n-1}(i_j-l_j)\prod_{k=j+1}^{n}d_k+(i_n-l_n)$$

列优先映射的映射函数可以类推得到，下面为列优先队列的结论。

n 维数组 $a[b_1][b_2]\cdots[b_n]$的列优先映射函数为

$$\mathrm{Map}(i_1,i_2,\cdots,i_n)=i_1+b_1i_2+\cdots+b_1b_2\cdots b_{n-2}i_{n-1}+b_1b_2\cdots b_{n-1}i_n$$

$$=i_1+\sum_{j=2}^{n}\Big(\prod_{k=1}^{j-1}b_k\Big)i_j=\sum_{j=1}^{n}c_ji_j$$

$$\mathrm{Loc}(i_1,i_2,\cdots,i_n)=\mathrm{base}+(i_1+b_1i_2+\cdots+b_1b_2\cdots b_{n-2}i_{n-1}+b_1b_2\cdots b_{n-1}i_n)\mathrm{size}$$

$$=\mathrm{base}+\sum_{j=1}^{n}c_ji_j\,\mathrm{size}$$

其中 $c_1=1,c_{j+1}=b_jc_j,1\leqslant i<n$。

设 n 维数组的第 k 维的下标范围是$[l_k,h_k]$，记

$$d_k=h_k-l_k+1,\quad k=1,2,\cdots,n$$

d_k 为第 k 维的长度(元素个数)，此时有

$$\mathrm{Map}(i_1,i_2,\cdots,i_n)=(i_1-l_1)+d_1(i_2-l_2)+\cdots+d_1d_2\cdots d_{n-1}(i_n-l_n)$$

$$=(i_1-l_1)+\sum_{j=2}^{n}\Big(\prod_{k=1}^{j-1}d_k\Big)(i_j-l_j)$$

**5.1.3 数组的类模板定义

在 C++ 的数组中，n 维数组的下标采用形式$[i_1][i_2]\cdots[i_n]$，i_j 为非负整数。但 C++ 本身无法保证数组下标 i_j 的合法性，为克服 C++ 标准数组的不足，下面给出一个数组类模板声明：

```
//数组类模板
template<class ElemType>
class Array
{
protected:
//数据成员
    ElemType *base;                                  //数组元素基址
    int dim;                                         //数组维数
    int *bounds;                                     //数组各维长度
    int *constants;                                  //数组映像函数常量

//辅助函数模板
    int Locate(int sub0, va_list &va) const;         //求元素在顺序存储中的位置

public:
//抽象数据类型方法声明及重载编译系统默认方法声明
    Array(int dm, ...);                              //由维数 dm 与随后的各维长度构造数组
    virtual ~Array();                                //析构函数模板
    ElemType &operator()(int sub0, ...);             //重载函数运算符
    Array(const Array<ElemType>&source);             //复制构造函数模板
    Array<ElemType>&operator=(const Array<ElemType>&source); //重载赋值运算符
};
```

上面用到了变长参数,通过重载函数运算符(),实现了数组元素的存取,例如定义一个二维数组 Array $a(2,3,3)$,$a(1,2)$为第 1 行,第 2 列的元素,同时还对下标的合法性进行了检查,下面是数组类模板的实现部分。

```
//数组类模板的实现部分
template <class ElemType>
Array<ElemType>::Array(int dm, ...)
//操作结果:由维数 dm 与随后的各维长度构造数组
{
    if (dm<1)
    {   //出现异常
        cout <<"维数不能小于 1!" <<endl;             //提示信息
        exit(1);                                     //退出程序
    }
    dim=dm;                                          //数组维数为 dm
    bounds=new int[dim];                             //分配数组各维长度存储空间
    va_list va;                                      //变长参数变量
    int elemTotalNum=1;                              //元素总数
    int tempPos;                                     //临时变量

    va_start(va, dim);                               //初始化变长参数变量 va,用于存储变长
        //参数信息,dm 为省略号左侧最右边的参数标识符
    for (tempPos=0; tempPos<dim; tempPos++)
```

```
    {   //为各维长度赋值并计算数组总元素个数
        bounds[tempPos] =va_arg(va, int);                //取出变长参数作为各维长度
        elemTotalNum =elemTotalNum * bounds[tempPos];    //统计数组总元素个数
    }
    va_end(va);                                          //结束变长参数的引用

    base =new ElemType[elemTotalNum];                    //分配数组元素空间
    constants =new int[dim];                             //分配数组映像函数常量
    constants[dim -1] =1;
    for (tempPos =dim -2; tempPos >=0; --tempPos)
        constants[tempPos] =bounds[tempPos +1] * constants[tempPos +1];
                                                         //计算数组映像函数常量
}

template <class ElemType>
Array<ElemType>::~Array()
//操作结果:释放数组所占用空间
{
    if (base !=NULL) delete []base;                      //释放数组元素空间
    if (bounds !=NULL) delete []bounds;                  //释放各维长度空间
    if (constants !=NULL) delete []constants;            //释放映像函数常量空间
}

template <class ElemType>
int Array<ElemType>::Locate(int sub0, va_list &va) const
//操作结果:求元素在顺序存储中的位置,sub0 为第 1 维下标,va 为第 2 至第 dim 维下标的可变
//参数
{
    if (!(sub0 >=0 && sub0 <bounds[0]))                  //第 1 维下标范围错
    {   //出现异常
        cout <<"下标出界!" <<endl;                       //提示信息
        exit(2);                                         //退出程序
    }
    int position =constants[0] * sub0;              //初始化元素在顺序存储中的位置
    for (int tempPos =1; tempPos <dim; tempPos++)
    {
        int sub =va_arg(va, int);                   //取出第 tempPos+1 维数组元素下标
        if (!(sub >=0 && sub <bounds[tempPos]))     //第 tempPos+1 维下标范围错
        {   //出现异常
            cout <<"下标出界!" <<endl;              //提示信息
            exit(3);                                //退出程序
        }
        position +=constants[tempPos] * sub;        //累加乘积求元素在顺序存储中的位置
    }
    return position;                                //返回元素在顺序存储中的位置
```

```
}

template <class ElemType>
ElemType &Array<ElemType>::operator()(int sub0, ...)
//操作结果:重载函数运算符,sub0 为第 1 维的下标
{
    va_list va;                                    //变长参数变量
    va_start(va, sub0);                            //初始化变长参数变量 va,用于存储变长
        //参数信息,sub0 为第 1 维的下标,sub0 为省略号左侧最右边的参数标识符
    int position =Locate(sub0, va);                //元素在顺序存储中的位置
    va_end(va);                                    //结束变长参数变量的引用
    return * (base +position);                     //返回数组元素
}

template <class ElemType>
Array<ElemType>::Array(const Array<ElemType>&source)
//操作结果:由数组 source 构造新数组——复制构造函数模板
{
    dim =source.dim;                               //数组维数

    int elemTotalNum =1;                           //元素总数
    int tempPos;                                   //临时变量
    for (tempPos =0; tempPos <dim; tempPos++)
    {   //统计数组总元素个数
        elemTotalNum =elemTotalNum * source.bounds[tempPos];
                                                   //计算数组总元素个数
    }
    base =new ElemType[elemTotalNum];              //为数组元素分配存储空间
    for (tempPos =0; tempPos <elemTotalNum; tempPos++)
    {   //复制数组元素
        base[tempPos] =source.base[tempPos];
    }

    bounds =new int[dim];                          //为数组各维长度分配存储空间
    constants =  new int[dim];                     //为数组映像函数常量分配存储空间
    for (tempPos =0; tempPos <dim; tempPos++)
    {   //复制数组各维长度与映像函数常量
        bounds[tempPos] =source.bounds[tempPos];        //各维长度
        constants[tempPos] =source.constants[tempPos]; //映像函数常量
    }
}

template <class ElemType>
Array<ElemType>&Array<ElemType>::operator =(const Array<ElemType>&source)
//操作结果:将数组 source 赋值给当前数组——重载赋值运算符
```

```
{
    if (&source !=this)
    {
        if (base !=NULL) delete []base;                    //释放数组元素空间
        if (bounds !=NULL) delete []bounds;                //释放各维长度空间
        if (constants !=NULL) delete []constants;          //释放映像函数常量空间

        dim =source.dim;                                   //数组维数

        int elemTotalNum =1;                               //元素总数
        int tempPos;                                       //临时变量
        for (tempPos =0; tempPos <dim; tempPos++)
        {   //统计数组总元素个数
            elemTotalNum =elemTotalNum * source.bounds[tempPos];
                                                           //计算数组总元素个数
        }
        base =new ElemType[elemTotalNum];                  //为数组元素分配存储空间
        for (tempPos =0; tempPos <elemTotalNum; tempPos++)
        {   //复制数组元素
            base[tempPos] =source.base[tempPos];
        }

        bounds =new int[dim];                          //为数组各维长度分配存储空间
        constants =  new int[dim];                     //为数组映像函数常量分配存储空间
        for (tempPos =0; tempPos <dim; tempPos++)
        {   //复制数组各维长度与映像函数常量
            bounds[tempPos] =source.bounds[tempPos];        //各维长度
            constants[tempPos] =source.constants[tempPos]; //映像函数常量
        }
    }
    return *this;
}
```

上面直接给出了多维数组的定义，没有重载下标运算符[]，要重载此运算符，可分别定义一维数组，二维数组……当然，这样定义数组通用性要差些。

5.2 矩　　阵

5.2.1 矩阵的定义和操作

矩阵可描述为二维数组。矩阵的下标通常从 1 开始，而不像 C 和 C++ 或其他语言中的数组那样从 0 开始，并且常用 $a(i, j)$来引用矩阵中第 i 行第 j 列的元素。

一个 $m \times n$ 矩阵是一个 m 行、n 列的表，如图 5.3 所示。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

图 5.3　$m \times n$ 矩阵示意图

矩阵与二维数组有很多相似之处，一般用如下的二维数组来描述一个 $m \times n$ 矩阵 $\boldsymbol{a}_{m \times n}$：

```
ElemType a[m][n];
```

矩阵中的元素 $a(i,j)$对应于二维数组的元素 $a[i-1][j-1]$。这种形式要求使用数组的下标[][]来指定每个矩阵元素。这种变化降低了应用代码的可读性，也增加了出错的概率。可以通过定义一个类 Matrix 来克服这个问题。在 Matrix 类中，将矩阵元素按照行优先次序存储到一个一维数组 elems 中，另外通过重载函数运算符()实现使用(i,j)来指定每个元素，并且根据矩阵的约定，其行列下标值都从 1 开始。

矩阵的基础操作如下。

1) int GetRows() const

初始条件：矩阵已存在。

操作结果：返回矩阵行数。

2) int GetCols() const

初始条件：矩阵已存在。

操作结果：返回矩阵列数。

3) ElemType &operator()(int i, int j)

初始条件：矩阵已存在。

操作结果：重载函数运算符。

下面是矩阵的具体类模板声明及实现。

```
//矩阵类模板
template<class ElemType>
class Matrix
{
protected:
//数据成员
    ElemType * elems;                                    //存储矩阵元素
    int rows, cols;                                      //矩阵行数和列数

public:
//抽象数据类型方法声明及重载编译系统默认方法声明
    Matrix(int rs, int cs);                              //构造一个 rs 行 cs 列的矩阵
    virtual ~Matrix();                                   //析构函数模板
    int GetRows() const;                                 //返回矩阵行数
    int GetCols() const;                                 //返回矩阵列数
    ElemType &operator()(int row, int col);              //重载函数运算符
    Matrix(const Matrix<ElemType> &source);              //复制构造函数模板
    Matrix<ElemType> &operator = (const Matrix<ElemType> &source);
                                                         //重载赋值运算符
};
```

构造函数模板首先检查给定的行、列数值是否合法，然后为存储矩阵元素的指针 elems 分配存储空间。

```
template <class ElemType>
Matrix<ElemType>::Matrix(int rs, int cs)
//操作结果:构造一个 rs 行 cs 列的矩阵
{
    if (rs <1 && cs <1)
    {   //出现异常
        cout <<"行数或列数无效!" <<endl;                    //提示信息
        exit(1);                                            //退出程序
    }
    rows =rs;                                               //rs 为行数
    cols =cs;                                               //cs 为列数
    elems =new ElemType[rows * cols];                       //分配存储空间
}
```

重载函数运算符()用来指定矩阵中第 row 行、第 col 列的元素,根据行优先存储时的地址映射关系,实际上就是取数组 elems 中的第(row－1) * cols＋(col－1)个元素。

```
template <class ElemType>
ElemType &Matrix<ElemType>::operator()(int i, int j)
//操作结果:重载函数运算符
{
    if (row <1 || row >rows || col <1 || col >cols)
    {   //出现异常
        cout <<"下标出界!" <<endl;                          //提示信息
        exit(2);                                            //退出程序
    }

    return elems[(row -1) * cols +col -1];                  //返回元素
}
```

5.2.2 特殊矩阵

如果值相同的元素或零元素在矩阵中按一定的规律分布,这样的矩阵称为特殊矩阵,可以用特殊方法进行存储和处理,以便提高空间和时间效率。下面首先介绍相关的几个概念。

方阵(square matrix):是行数和列数相同的矩阵。下面介绍的特殊矩阵都是方阵。

对称(symmetric)矩阵:a 是一个对称矩阵当且仅当对于所有的 i 和 j 有 $a(i, j)=a(j, i)$,如图 5.4(a)所示。

三对角(tridiagonal)矩阵:a 是一个三对角矩阵当且仅当 $|i-j|>1$ 时有 $a(i, j)=0$(或常数 c),如图 5.4(b)所示。

下三角(lower triangular)矩阵:a 是一个下三角矩阵当且仅当 $i<j$ 时有 $a(i, j)=0$(或常数 c),如图 5.4(c)所示。

上三角(upper triangular)矩阵:a 是一个上三角矩阵当且仅当 $i>j$ 时有 $a(i, j)=0$(或常数 c),如图 5.4(d)所示。

6	1	3	5		1	2	0	0		9	0	0	0		6	3	7	2
1	2	8	5		5	3	3	0		5	3	0	0		0	4	1	7
3	8	4	8		0	9	8	5		6	1	2	0		0	0	2	0
5	5	8	9		0	0	7	6		8	4	3	4		0	0	0	1

(a)对称矩阵　(b)三对角矩阵　(c)下三角矩阵　(d)上三角矩阵

图 5.4　n 阶特殊矩阵示例

特殊矩阵的基础操作如下。

1) int GetOrder() const

初始条件：特殊矩阵已存在。

操作结果：返回特殊矩阵阶数。

2) ElemType &operator()(int row, int col)

初始条件：特殊矩阵已存在。

操作结果：重载函数运算符。

1. 三对角矩阵

在一个 $n\times n$ 的三对角矩阵中，三条对角线之外的元素都是 0 或为一个常数 c。

主对角线：主对角线上元素的行列下标值 i 和 j 之间满足 $i=j$；

主对角线之下的对角线(称为低对角线)：低对角线上元素的行列下标值 i 和 j 之间满足 $i=j+1$；

主对角线之上的对角线(称为高对角线)：高对角线上元素的行列下标值 i 和 j 之间满足 $i=j-1$。

由于主对角线上有 n 个元素，两条次对角线上分别有 $n-1$ 个元素，所以这三条对角线上的元素总数为 $3n-2$。因为其他元素都为常数 c，所以只需要存储三条对角线上的元素及常数 c，可以使用一个有 $3n-1$ 个存储单元的一维数组 elems 来存储三对角矩阵中三条对角线上的元素及常数 c。对于图 5.4(b)所示 4×4 的三对角矩阵，三条对角线上共有 11 个元素。如果三对象线之外的常量 c 映射到 elems[0]中，三对角线矩阵的元素逐行映射到数组 elems 中，则有

elems：0,1, 2, 5, 3, 3, 9, 8, 5, 7, 6

如果逐列映射到数组 T 中，则有

elems：0, 1, 5, 2, 3, 9, 3, 8, 7, 5, 6

如果按照对角线的次序(从最上面的对角线开始)进行映射，则有

elems：0, 2, 3, 5, 1, 3, 8, 6, 5, 9, 7

在实际使用时，可根据具体的操作需要，在这三种不同的映射方式中选择一种。下面的类模板声明采用的是对角线映射方式。

```
//三对角矩阵类模板
template<class ElemType>
class TriDiagonalMatrix
{
protected:
//数据成员
```