第4章　数组和矩阵

在实际应用过程中，数据通常以表的形式出现。尽管用数组来描述表数据是最自然的方式，但有时为了减少程序的时间和空间需求，通常会采用自定义的描述方式，比如，当表中大部分数据全为0时。

本章首先检查了多维数组的行主描述形式和列主描述形式。通过行主描述和列主描述，可以把多维数组映射成一维数组。

尽管C++支持多维数组，但它无法保证数组下标的合法性。同时， C++也未能提供数组的输入、输出以及简单的算术运算（如数组赋值和数组相加）。为了克服这些不足，我们针对一维数组和二维数组分别设计了类Array1D和类Array2D。

矩阵通常被描述成一个二维数组。不过，矩阵的索引通常从1开始，而不是像数组那样从0开始，并且通常使用 $A(i,j)$而不是$A[i][j]$来引用矩阵中的元素 (i,j)。为此，设计了另一个类Matrix 以便更好地描述矩阵。

本章还考察了一些具有特殊结构的矩阵，如对角矩阵、三对角矩阵、三角矩阵和对称矩阵。与采用二维数组描述矩阵相比，采用公式化的方法来描述这些特殊矩阵所需要的空间将大大减小，同时，公式化的描述方法还可以显著地节省诸如矩阵加和矩阵减操作所需要的运行时间。

本章的最后一节给出了稀疏矩阵（即大部分元素为 0的矩阵）的公式化描述和链表描述，这两种描述方法对于0元素都做了特殊处理。

4.1　数组

4.1.1　抽象数据类型

数据对象*array*的每个实例都是形如（index，value）的数据对集合，其中任意两对数据的*index*值都各不相同。有关数组的操作如下：

• *Create*——创建一个初始为空的数组（即：不含任何数据）

• *Store*——向数组中添加一对（index，value）数据，如果数组中已经存在索引值与 *index*相同的数据对，则删除该数据对。

• *Retrieve*——返回具有给定index值的数据对的value值。

ADT4-1给出了具有上述三种操作的抽象数据类型 Array。

ADT 4-1　数组的抽象数据类型描述

抽象数据类型*Array*{
实例
　形如 (index,value)的数据对集合，其中任意两对数据的 index值都各不相同
操作
　Create()：创建一个空的数组
　Store(*index, value*)：添加数据(index, value)，同时删除具有相同index值的数据对（如果存在）
　Retrieve(*index*)：返回索引值为index的数据对

例4-1 上个星期每天的高温（华氏度数）可用如下的数组来表示：

high={(sunday, 82), (monday, 79), (tuesday, 85), (wednesday, 92),(thursday, 88), (friday, 89), (saturday, 91)}

数组中的每对数据都包含一个索引（星期）和一个值（当天的温度），数组的名称为*high*。通过执行如下操作，可以将monday的温度改变为83。

Store(monday, 83)

通过执行如下操作，还可以确定friday的温度：

Retrieve(friday)

也可以采用如下的数组来描述每天的温度：

high={(0,82), (1,79), (2,85), (3,92), (4,88), (5,89), (6,91)}

在这个数组中，索引值是一个数值，而不是日期名，数值（0，1，2，...）代替了一周中每天的名称（sunday，monday，tuesday，...）。

4.1.2 C++数组

尽管在C++中数组是一个标准的数据结构，但C++数组的索引（也称为下标）必须采用如下形式：

$[i_1][i_2][i_3]...[i_k]$

i_j 为非负整数。如果k为1，则数组为一维数组，如果k 为2，则为二维数组。i_1 是索引的第一个坐标，i_2 是第二个，i_k 是第k个。在C++中，值为整数类型的k 维数组score可用如下语句来创建：

int score$[u_1][u_2][u_3]...[u_k]$

其中u_i 是正的常量或由常量表示的正的表达式。对于这样一个数组描述，索引 i_j 的取值范围为：$0 \le i_j < u_j$，$1 \le j \le k$。因此，该数组最多可以容纳$n=u_1 u_2 u_3 ... u_k$ 个值。由于数组score中的每个值都是整数，所以需要占用 sizeof(int)个字节，因而，整个数组所需要的内存空间为sizeof(score)=n*sizeof(int)个字节。C++编译器将为数组预留这么多空间。假如预留空间的始地址为start，则该空间将延伸至start+size(score)-1。

4.1.3 行主映射和列主映射

为了实现与数组相关的函数Store和Retrieve，必须确定索引值在[*start*,*start*+*n**sizeof(score)-1]中的相应位置。实际上就是把数组索引$[i_1][i_2][i_3]...[i_k]$映射到[0, *n*-1]中的某个数$map(i_1, i_2, i_3, ..., i_k)$，使得该索引所对应的元素值存储在以下位置：

$$start + map(i_1, i_2, i_3, ..., i_k)*\text{sizeof}(\text{int})$$

当数组维数为1时（即k=1），使用以下函数：

$$map(i_1) = i_1$$

当数组维数为2时，各索引可按图4-1所示的表格形式进行排列，第一个坐标相同的索引位于同一行，第二个坐标相同的索引位于同一列。

在图4-1中从第一行开始，依次对每一行中的每个索引从左至右进行连续编号，即可得到

图4-2a 所示的映射结果。这种把二维数组中的位置映射为[0, *n*-1]中某个数的方式被称为行主映射。C++中即采用了这种行主映射模式。图4-2b 中给出了另一种模式，称之为列主映射。在列主映射模式中，对索引的编号从最左列开始，每一列按照从上到下的次序进行排列。

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

图4-1 int score[3][6]的索引排列表

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

a)

0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17

b)

图4-2 二维数组的映射

a) 行主映射 b) 列主映射

行主次序所对应的映射函数为：

$$map\ (i_1, i_2) = i_1 u_2 + i_2$$

其中u_2是数组的列数。可以注意到，在行主映射模式中，在对索引$[i_1][i_2]$进行编号时，第0，...，i_1-1 行中的$i_1 u_2$个元素以及第i_1行中的前i_2个元素都已经被编号。

让我们用图4-2a 中的3×6数组来验证上述的行主映射函数。由于列数为 6，所以映射公式变成：

$$map\ (i_1, i_2) = 6i_1 + i_2$$

因此有*map*(1,3)=6+3=9，*map*(2,5)=6*2+5=17。与图4-2a 中所给出的编号相同。

可以扩充上述的行主映射模式来得到二维以上数组的映射函数。注意在行主次序中，首先列出所有第一个坐标为0的索引，然后是第一个坐标为1的索引，…。第一个坐标相同的所有索引按其第二个坐标的递增次序排列，也即各个索引按照词典序进行排列。对于一个三维数组，首先列出所有第一个坐标为0的索引，然后是第一个坐标为1的索引，…。第一个坐标相同的所有索引按其第二个坐标的递增次序排列，前两个坐标相同的所有索引按其第三个坐标的递增次序排列。例如数组score[3][2][4]中的索引按行主次序排列为：

[0][0][0], [0][0][1], [0][0][2]], [0][0][3], [0][1][0], [0][1][1], [0][1][2], [0][1][3],
[1][0][0], [1][0][1], [1][0][2], [1][0][3], [1][1][0], [1][1][1], [1][1][2], [1][1][3],
[2][0][0], [2][0][1], [2][0][2], [2][0][3], [2][1][0], [2][1][1], [2][1][2], [2][1][3]

三维数组的行主映射函数为：

$$map(i_1, i_2, i_3) = i_1 u_2 u_3 + i_2 u_3 + i_3$$

可以观察到，所有第一个坐标为i_1的元素都位于第一个坐标小于i_1的元素之前，第一个坐标都相同的元素数目为$u_2 u_3$。因此第一个坐标小于i_1的元素数目为$i_1 u_2 u_3$，第一个坐标等于i_1且第二个坐标小于i_2的元素数目为$i_2 u_3$，第一个坐标等于i_1且第二个坐标等于i_2且第三个坐标小于i_3的元素数目为i_3。

4.1.4 类Array1D

尽管C++支持一维数组，但这种支持很不够。例如，可以使用超出正常范围之外的索引值来访问数组。考察如下的数组a：

```
int a[9]
```

可以访问数组元素a[－3]，a[9]和a[90]，尽管－3，9和90是非法的索引。允许使用非法的索引通常会使程序产生无法预料的行为并给调试带来困难。并且 C++数组不能使用如下的语句来输出数组：

```
cout << a << endl;
```

也不能对一维数组进行诸如加法和减法等操作。为了克服这些不足，定义了类 Array1D(见程序4-1)，该类的每个实例X都是一个一维数组。X的元素存储在数组X.element之中，第i 个元素位于X.element [i]，0≤i<size。

程序4-1 一维数组的类定义

```
template<class T>
class Array1D {
  public:
    Array1D(int size = 0);
    Array1D(const Array1D<T>& v); // 复制构造函数
    ~Array1D() {delete [] element;}
    T& operator[](int i) const;
    int Size() {return size;}
    Array1D<T>& operator=(const Array1D<T>& v);
    Array1D<T> operator+() const; // 一元加法操作符
    Array1D<T> operator+(const Array1D<T>& v) const;
    Array1D<T> operator-() const; // 一元减法操作符
    Array1D<T> operator-(const Array1D<T>& v) const;
    Array1D<T> operator*(const Array1D<T>& v) const;
    Array1D<T>& operator+=(const T& x);
  private:
     int size;
     T *element; //一维数组
};
```

这个类的共享成员包括：构造函数，复制构造函数，析构函数，下标操作符 []，返回数组大小的函数Size，算术操作符＋、－、*和＋＝。此外还可以添加其他的操作符。程序 4-2给出了构造函数和复制构造函数的代码。构造函数有点违背 ANSI C++的要求，它允许数组具有0个元素。如果我们不希望出现这种违规行为，可以对这段代码进行适当的修改。

程序4-2 一维数组的构造函数

```
template<class T>
Array1D<T>::Array1D(int sz)
{// 一维数组的构造函数
  if (sz < 0) throw BadInitializers();
  size = sz;
```

```
    element = new T[sz];
}
template<class T>
Array1D<T>::Array1D(const Array1D<T>& v)
{// 一维数组的复制构造函数
    size = v.size;
    element = new T[size];  // 申请空间
    for (int i = 0; i < size; i++) // 复制元素
        element[i] = v.element[i];
}
```

程序4-3给出了重载操作符[ ]的代码。该操作符用来返回指向第i 个元素的引用，这样可以使存储和查询操作按照很自然的方式进行。利用这个操作符，使用语句：

```
X[1] = 2 * Y[3];
```

就可以按照期望的方式进行工作，其中X和Y的类型均为Array1D。代码Y[3]在对象Y上施加操作符[ ]，结果返回指向元素3的一个引用，然后将它乘以2。代码X[1]也调用了操作符[],结果返回一个指向X[1]的引用，之后2*Y[3]的结果将存储在这个引用位置内。

程序4-3 重载下标操作符[]

```
template<class T>
T& Array1D<T>::operator[](int i) const
{// 返回指向第 i个元素的引用
    if (i < 0 || i >= size) throw OutOfBounds();
    return element[i];
}
```

程序4-4给出了赋值操作符的代码。这段代码通过验证等号左右两边是否相同来避免进行自我赋值（即X=X）。为了进行赋值，应该首先释放目标数组 *this 所占用的空间，然后分配能够容纳源数组v 的空间。尽管对new 的调用可能会失败，但并没有捕获所引发的异常，而是把它留给更适合处理这种异常的代码来捕获。如果new 没有引发异常，则把源数组中的元素逐个复制到目标数组中。

程序4-4 重载赋值操作符=

```
template<class T>
Array1D<T>& Array1D<T>::operator=(const Array1D<T>& v)
{// 重载赋值操作符=
    if (this != &v) {// 不是自我赋值
        size = v.size;
        delete [] element; // 释放原空间
        element = new T[size]; // 申请空间
        for (int i = 0; i < size; i++) //复制元素
            element[i] = v.element[i];
        }
    return *this;
}
```

程序4-5给出了二元减法操作符、一元减法操作符和增值操作符的代码。其他操作符的代

码与此类似。

程序4-5 重载二元减法操作符、一元减法操作符和增值操作符

```
template<class T>
Array1D<T> Array1D<T>:: operator-(const Array1D<T>& v) const
{// 返回 w = (*this) - v
  if (size != v.size) throw SizeMismatch();
  // 创建结果数组 w
  Array1D<T> w(size);
  for (int i = 0; i < size; i++)
     w.element[i] = element[i] - v.element[i];
  return w;
}
template<class T>
Array1D<T> Array1D<T>::operator-() const
{// 返回w = -(*this)
  // 创建结果数组 w
  Array1D<T> w(size);
  for (int i = 0; i < size; i++)
     w.element[i] = -element[i];
  return w;
}
template<class T>
Array1D<T>& Array1D<T>::operator+=(const T& x)
{//把 x 加到 (*this)的每个元素上
  for (int i = 0; i < size; i++)
     element[i] += x;
  return *this;
  }
```

复杂性

当T是一个内部C++数据类型（如int, float和char）时，构造函数和析构函数的复杂性为$\Theta(1)$，而当T是一个用户自定义的类时，构造函数和析构函数的复杂性为O(size)。之所以存在这种差别，是因为当T是一个用户自定义类时，在用new（delete）创建（删除）数组element的过程中，对于element的每个元素都要调用一次T的构造函数（析构函数）。下标操作符[ ]的复杂性为$\Theta(1)$，其他操作符的复杂性均为O(size)。（注意复杂性不会是Θ(size)，因为所有操作符的代码都可以引发一个异常并提前终止）。

4.1.5 类Array2D

对于二维数组，可以定义一个类Array2D，见程序4-6。Array2D所采用的描述格式类似于图1-1。二维数组可被视为一维数组的集合。虽然在图1-1中采用一个一维数组指向各个行数组，但在程序4-6中，采用一维数组row来直接存储每个行数组。数组 row之所以被设置成标准的C++一维数组而不是Array1D的一个实例，是因为我们能够完全控制对 row中每个元素的访问（row是一个私有成员）。因此，可以确保使用合法的数组下标。我们不需要 Array1D所提供的

附加功能。row[i]是类型为Array1D的一维数组，它代表二维数组的第i行。请注意，与图1-1不同的是，row[i]不是指向一维数组的指针。另外一种采用行主映射方式描述二维数组的方法见4.2.2节。

程序4-6 二维数组的类定义

```
template<class T>
class Array2D {
  public:
    Array2D(int r = 0, int c = 0);
    Array2D(const Array2D<T>& m); // 复制构造函数
    ~Array2D() {delete [] row;}
    int Rows() const {return rows;}
    int Columns() const {return cols;}
    Array1D<T>& operator[](int i) const;
    Array2D<T>& operator=(const Array2D<T>& m);
    Array2D<T> operator+() const; // 一元加法操作符
    Array2D<T> operator+(const Array2D<T>& m) const;
    Array2D<T> operator-() const; // 一元减法操作符
    Array2D<T> operator-(const Array2D<T>& m) const;
    Array2D<T> operator*(const Array2D<T>& m) const;
    Array2D<T>& operator+=(const T& x);
  private:
     int rows, cols;  // 数组维数
     Array1D<T> *row; // 一维数组的数组
};
```

程序4-7给出了构造函数的代码。缺省情况下创建一个 0行0列的数组，0行、非0列的数组是不允许的。如下语句：

```
row=new Array1D<T>[r];
```

创建一个具有r个位置的一维数组row，每个位置的类型为Array1D<T>。在创建数组row时，对于每个位置都将调用Array1D的构造函数。row[i]是一个具有缺省大小（为0）的一维数组，0≤i<r。由于在创建一个数组时，仅会调用缺省的构造函数，因此在创建数组 row的过程中存在一个for循环，在这个循环中将依次调整row中每个元素的大小。Resize是Array1D的一个新的成员函数，通过执行以下代码，它能把一个一维数组的大小变成sz：

```
delete [ ] element;
size = sz;
element = new T [size];
```

程序4-7 二维数组的构造函数

```
template<class T>
Array2D<T>::Array2D(int r, int c)
{// 二维数组的构造函数
  // 合法的 r 和 c
  if (r < 0 || c < 0) throw BadInitializers();
  if ((!r || !c) && (r || c))
```

```
    throw BadInitializers();
  rows = r;
  cols = c;
  // 分配 r个具有缺省大小的一维数组
  row = new Array1D<T> [r];
  // 调整每个元素的大小
  for (int i = 0; i < r; i++)
    row[i].ReSize(c);
}
```

注意，Array2D的析构函数并未明确释放分配给二维数组每一行元素的空间。不过，在执行delete [ ] row时，对于数组row的每一个元素， delete都将调用Array1D的析构函数，由Array1D的析构函数来释放每行元素所占用的空间。

程序4-8中的复制构造函数首先创建一个具有给定位置数的数组row，然后利用一维数组的赋值操作符复制二维数组中的每一行元素。与程序 4-7中的构造函数不同的是，复制构造函数并未验证m.rows和m.cols的合法性，因为当m被创建时，这两个值都是合法的。赋值操作符的代码类似于复制构造函数的代码，不过，与程序4-4中的赋值操作符代码一样，Array2D的赋值操作也检查了是否为自我赋值。

程序4-8 二维数组的复制构造函数

```
template<class T>
Array2D<T>::Array2D(const Array2D<T>& m)
{// 二维数组的复制构造函数
  rows = m.rows;
  cols = m.cols;
  // 分配指向一维数组的数组
  row = new Array1D<T> [rows];
  // 复制每一行
  for (int i = 0; i < rows; i++)
    row[i] = m.row[i];
}
```

若X的类型为Array2D<T>，则X[i][j]被分解为(X.operator[i]).operator[j]。因此，对于X[i][j]将首先利用实参i来调用Array2D<T>::operator[ ]，如果该操作返回了对象Y，则继续用实参j来调用typeof(Y)::operator[ ]。若typeof(Y)不同于Array2D<T>，则仅对二维数组的第一个索引调用Array2D<T>::operator[ ]。根据这种理解，可以得到程序4-9中的代码，这段代码只是简单地返回一个与第一个索引相对应的一维数组的引用。此时，若执行代码 X[i][j]，则X[i]将调用Array2D<T>::operator[ ]，该操作符返回一个指向X.row[i]（即X的第i行）的引用。由于这个引用的类型为Array1D<T>，所以接下来将调用Array1D<T>::operator[]并返回一个指向相应数组元素的引用。

程序4-9 二维数组的重载操作符[ ]

```
template<class T>
Array1D<T>& Array2D<T>::operator[](int i) const
{//二维数组的第一个索引
  if (i < 0 || i >= rows) throw OutOfBounds();
```

```
    return row[i];
}
```

二元减法操作符的代码见程序 4-10，它只是简单地对二维数组的每一行调用 Array1D <T>::operator-。加法操作符、一元减法操作符、增值操作符和输出操作符的代码与此类似。

程序4-10 二元减法操作符

```
template<class T>
Array2D<T> Array2D<T>:: operator-(const Array2D<T>& m) const
{// 返回 w = (*this) - m.
  if (rows != m.rows || cols != m.cols)
    throw SizeMismatch();
  // 创建存放结果的数组w
  Array2D<T> w(rows,cols);
  for (int i = 0; i < rows; i++)
     w.row[i] = row[i] - m.row[i];
  return w;
}
```

乘法操作符的实现与矩阵乘（见程序2-25）很相似，见程序4-11。

程序4-11 乘法操作符

```
template<class T>
Array2D<T> Array2D<T>:: operator*(const Array2D<T>& m) const
{// 矩阵乘，返回w = (*this) * m.
  if (cols != m.rows) throw SizeMismatch();
  // 创建存放结果的数组 w
  Array2D<T> w(rows, m.cols);
  for (int i = 0; i < rows; i++)
    for (int j = 0; j < m.cols; j++) {
      T sum = (*this)[i][0] * m[0][j];
      for (int k = 1; k < cols; k++)
        sum += (*this)[i][k] * m[k][j];
      w[i][j] = sum;
    }
  return w;
}
```

复杂性

当T是一个C++内部数据类型时，构造函数和析构函数的复杂性均为 O(rows)，而当T是一个用户自定义的类时，构造函数和析构函数的复杂性为 O(rows*cols)。复制构造函数及operator的复杂性为O(rows*cols)，下标操作符的复杂性为 Θ(1)，乘法操作符的复杂性 O(rows*cols*m.cols)。

练习

1. 扩充Array1D类（见程序4-1），重载操作符 <<（输入一个数组）、+（一元加法）、*=（将每个元素右乘一个类型为T的元素）、/=和-=。试测试代码。

2. 对于一维数组设计一个类Array1D<T1,T2>，其中T1是数组索引的类型，T2是数组元素的类型。T1可以是任何枚举类型。例如，如果T1为bool类型，那么合法的索引值为true和false。该类中应包含Array1D（见程序4-1）的所有共享成员。

3. 对于Array2D类完成练习1。另外再增加一个操作符，用来对两个大小相同的数组进行乘法操作（两个数组中相对应的元素两两相乘）。

4. 编写与二维数组的创建和删除函数（见程序 1-13和程序1-14）相对应的的模板函数Make3DArray和Delete3DArray。可使用如下语句创建一个整数数组*s*：

```
int ***x
```

可以使用语法x[i][j][k]来访问每个元素。试测试代码的正确性。

5. 模仿Array2D<T>设计一个三维数组的类Array3D<T>。

6. 按行主次序列出score[2][3][2][2]的索引。

7. 给出四维数组的行主映射函数。

8. 给出五维数组的行主映射函数。

9. 给出k维数组的行主映射函数。

10. 按列主次序列出score[2][3][4]的索引。注意，此时首先列出第三个坐标为0的所有索引，然后列出第三个坐标为1的所有索引，…。第三个坐标相同的索引按其第二个坐标的次序进行排列，后两个坐标都相同的索引按其第一个坐标的次序排列。

11. 给出三维数组的列主映射函数（参考前面的练习）。

12. 按列主次序列出score[2][3][2][2]的索引。

13. 给出四维数组的列主映射函数。

14. 给出k维数组的列主映射函数。

15. 假定对于一个二维数组采用从最后一行开始，每一行从右至左的方式进行映射

1) 按这种次序列出score[3][5]的索引。

2) 给出score[u_1][u_2]的映射函数。

16. 假定对于一个二维数组采用从最后一列开始，每一列从顶至底的方式进行映射

1) 按这种次序列出score[3][5]的索引。

2) 给出score[u_1][u_2]的映射函数。

4.2 矩阵

4.2.1 定义和操作

一个$m \times n$ 的矩阵（matrix）是一个m行、n 列的表（如图4-3所示），其中m 和n 是矩阵的维数。

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

图4-3 一个5×4的矩阵

矩阵通常被用来组织数据。例如，为了登记世界上的资源，可以首先列出一个感兴趣的资源类型表，在这个表中可以包含矿产（金、银等）、动物（狮子、大象等）、人（物理学家、工程师等）等。对于每一种资源可以确定在每个国家中现存的数量。可以把这些数据列在一个二维的表中，其中每一列对应于一个国家，每一行对应于一种资源。这样就得到了一个 n 列（对应于n 个国家）、m 行（对应于m 种资源）的资源矩阵。可以使用符号$M(i, j)$ 来引用矩阵M 中第i 行、第j 列$1 \le i \le m$，$1 \le j \le n$ 的元素。如果上述资源矩阵中，第 i 行代表猫，第j 列代表美国，那么$assert\ (i, j)$ 就代表美国所拥有的猫的总数。

对于矩阵来说，最常见的操作就是矩阵转置、矩阵加、矩阵乘。一个 $m \times n$ 矩阵的转置矩阵是一个$n \times m$的矩阵M^T，它与M之间存在以下关系：

$$M^T(i, j) = M(j, i),\ 1 \le i \le n,\ \ 1 \le j \le m$$

仅当两个矩阵的维数相同时（即具有相同的行数和列数），才可以对两个矩阵求和。两个$m \times n$ 矩阵A 和B 相加所得到的$m \times n$ 矩阵C 如下：

$$C(i, j) = A(i, j) + B(i, j),\ \ 1 \le i \le n,\ \ 1 \le j \le m$$

仅当一个$m \times n$ 矩阵A 的列数与另一个$q \times p$ 矩阵B 的行数相同时（即$n = q$），才可以执行矩阵乘法$A*B$。$A*B$ 所得到的$m \times p$ 矩阵C 满足以下关系：

$$C(i,j) = \sum_{k=1}^{n} A(i,k) * B(k,j),\ \ 1 \leqslant i \leqslant m,\ \ 1 \leqslant j \leqslant p$$

例4-2　考察上面的资源矩阵。假定有两家公司分别给出了一个资源矩阵，且其中的数据没有重复。若两个$m \times n$ 资源矩阵分别为$assert1$和$assert2$，要得到所需要的资源矩阵，只需把矩阵$assert1$ 和$assert2$ 相加即可。

接下来，假定有另外一个$m \times s$ 矩阵$value$，$value(i, j)$ 代表资源矩阵中第i 行、第j 列资源的单价。令CV(i, j) 代表国家i 所拥有的资源j 的总价值，则CV是一个$n \times s$ 矩阵，满足如下公式：

$$CV = assert^T * value$$

按照二维数组来计算矩阵的转置以及矩阵加和乘的 C++函数见第2章（见程序2-19，2-22，2-24和2-25）。

4.2.2 类Matrix

可以使用 2Darray 来实现矩阵.

可用如下的二维整数数组来描述元素为整数的m×n 矩阵M：

```
int x[m][n]；或Array2D<int> x[m][n];
```

其中$M(i, j)$ 对应于x[i-1][j-1]。这种形式要求使用数组的索引[ ][ ]来指定每个矩阵元素。这种变化降低了应用代码的可读性，同时也增加了出错的概率。这可以通过定义一个类Matrix来克服，在Matrix类中，使用（）来指定每个元素，并且各行和各列的索引值都是从 1 开始的。

在程序4-12的类Matrix中采用一个一维数组element来存储rows×cols矩阵中的rows*cols个元素。

程序4-12　**类**Matrix

```
template<class T>
class Matrix {
```

```
  public:
    Matrix(int r = 0, int c = 0);
    Matrix(const Matrix<T>& m); //复制构造函数
    ~Matrix() {delete [] element;}
    int Rows() const {return rows;}
    int Columns() const {return cols;}
    T& operator()(int i, int j) const;
    Matrix<T>& operator=(const Matrix<T>& m);
    Matrix<T> operator+() const; // 一元加法
    Matrix<T> operator+(const Matrix<T>& m) const;
    Matrix<T> operator-() const; // 一元减法
    Matrix<T> operator-(const Matrix<T>& m) const;
    Matrix<T> operator*(const Matrix<T>& m) const;
    Matrix<T>& operator+=(const T& x);
  private:
     int rows, cols; // 矩阵维数
     T *element;     // 元素数组
};
```

程序4-13给出了构造函数的代码。复制构造函数与赋值操作符的代码相类似，可参考Array1D中的相应代码。

程序4-13 类Matrix的构造函数

```
template<class T>
Matrix<T>::Matrix(int r, int c)
{// 类Matrix的构造函数
  // 验证 r和 c的合法性
  if (r < 0 || c < 0) throw BadInitializers();
  if ((!r || !c) && (r || c))
      throw BadInitializers();
  // 创建矩阵
  rows = r; cols = c;
  element = new T [r * c];
}
```

但，使用1D array 去定义一个Matrix，是个更好的选择.
重定义（，）运算符.

为了重载矩阵下标操作符（），使用了C++的函数操作符（），与数组的下标操作符[]不同的是，该操作符可以带任意数量的参数。对于一个矩阵来说，需要两个整数参数。程序 4-14中的代码返回一个指向矩阵元素(i, j) 的引用。

程序4-14 类Matrix的下标操作符

```
template<class T>
T& Matrix<T>::operator()(int i, int j) const
{// 返回一个指向元素 (i,j)的引用
  if (i < 1 || i > rows || j < 1 || j > cols) throw OutOfBounds();
  return element[(i - 1) * cols + j - 1];
}
```

程序4-15给出了减法操作符的代码。与一维数组的减法操作（见程序 4-5）和二维数组的

减法操作（见程序4-10）相比，程序4-15中矩阵减法操作更接近程序4-5。矩阵加法操作符、一元减法操作符、增值操作符和输出操作符的代码都比较类似。

程序4-15 类Matrix的减法操作符

```
template<class T>
Matrix<T> Matrix<T>:: operator-(const Matrix<T>& m) const
{// 返回 (*this) - m.
  if (rows != m.rows || cols != m.cols)
    throw SizeMismatch();
  // 创建结果矩阵 w
  Matrix<T> w(rows, cols);
  for (int i = 0; i < rows * cols; i++)
    w.element[i] = element[i] - m.element[i];
  return w;
}
```

尽管可以参照二维数组的乘法操作符代码（见程序 4-11）来实现矩阵乘法，但它的效率可能比较低。程序4-16中矩阵乘法代码内的循环结构类似于程序 2-25和程序4-11，它们都有三个嵌套的for 循环。最内层的循环把*this 的第i 行与m 的第j 列相乘，得到元素(i,j)。进入最内层循环时，element[ct] 是第i 行的第一个元素，m.element[cm]是第j列的第一个元素。为了得到第i 行的下一个元素，可将ct 增加1，因为在行主次序中同一行的元素是连续存放的。为了得到第j 列的下一个元素，可将cm增加m.cols，因为在行主次序中同一列的两个相邻元素在位置上相差m.cols。当最内层循环完成时，ct 指向第i 行的最后一个元素，cm 指向第j 列的最后一个元素。对于for j 循环的下一次循环，起始时必须将ct 指向第i 行的第一个元素，而将cm 指向m 的下一列的第一个元素。对ct 的调整是在最内层循环完成后进行的。当for j 循环完成时，需要将ct 指向下一行的第一个元素，而将cm 指向第一列的第一个元素。

程序4-16 类Matrix的乘法操作符

```
template<class T>
Matrix<T> Matrix<T>:: operator*(const Matrix<T>& m) const
{// 矩阵乘法，返回w = (*this) * m.
  if (cols != m.rows) throw SizeMismatch();
  Matrix<T> w(rows, m.cols);  // 结果矩阵
  // 为 *this, m和 w定义游标
  // 并设定初始位置为(1,1)
  int ct = 0, cm = 0, cw = 0;
  // 对所有的i和j计算w(i,j)
  for (int i = 1; i <= rows; i++) {
    // 计算出结果的第 i 行
    for (int j = 1; j <= m.cols; j++) {
      // 计算w(i,j)的第一项
      T sum =  element[ct] * m.element[cm];
      // 累加其余项
      for (int k = 2; k <= cols; k++) {
        ct++;  // 指向*this第i行的下一个元素
```

```
            cm += m.cols;  // 指向m 的第j 列的下一个项
            sum += element[ct] * m.element[cm];
            }
          w.element[cw++] = sum;  // 保存w(i,j)
          // 重新调整至行首和下一列
          ct -= cols - 1;
          cm = j;
          }
        // 重新调整至下一行的行首和第一列
        ct += cols;
        cm = 0;
        }
      return w;
    }
```

复杂性

当T是一个内部数据类型时，矩阵构造函数复杂性为 O(1)，而当T是一个用户自定义的类时，构造函数的复杂性为O(rows*cols)。复制构造函数的复杂性为O(rows*cols)，下标操作符的复杂性为Θ(1)，乘法操作符的复杂性O(rows*cols*m.cols)。所有其他矩阵操作符的渐进复杂性分别与Array2D类中对应操作符的渐进复杂性相同。

练习

17. 扩充Matrix类（见程序4-12），重载操作符<<（输入一个矩阵）、+（一元加法）、*=（将每个元素右乘一个类型为T的元素）、/=和-=。测试所编写的代码。

18. 扩充Matrix类，增加一个共享成员Tranpose，用来返回转置后的矩阵。

19. 通过测量减法和乘法操作所需要的时间来比较Array2D类和Matrix 类的性能。阐述采用行主映射的优点。

4.3 特殊矩阵

4.3.1 定义和应用

方阵（square matrix）是指具有相同行数和列数的矩阵。一些常用的特殊方阵如下：

• 对角矩阵（diagonal） M是一个对角矩阵当且仅当$i \neq j$ 时有$M(i, j)=0$。如图4-4a 所示。

• 三对角矩阵（tridiagonal） M是一个三对角矩阵当且仅当 $|i - j| > 1$时有$M(i, j)=0$。如图4-4b 所示。

2 0 0 0	2 1 0 0	2 0 0 0	2 1 3 0	2 4 6 0
0 1 0 0	3 1 3 0	5 1 0 0	0 1 3 8	4 1 9 5
0 0 4 0	0 5 2 7	0 3 1 0	0 0 1 6	6 9 4 7
0 0 0 6	0 0 9 0	4 2 7 0	0 0 0 0	0 5 7 0
a)	b)	c)	d)	e)

图4-4 4×4矩阵

a) 对角矩阵 b) 三对角矩阵 c) 下三角矩阵 d) 上三角矩阵 e) 对称矩阵

• 下三角矩阵（lower triangular） M是一个下三角矩阵当且仅当$i < j$ 时有$M(i, j)=0$。如图4-4c 所示。

• 上三角矩阵（upper triangular） M是一个上三角矩阵当且仅当$i > j$ 时有$M(i, j)=0$。如图4-4d 所示。

• 对称矩阵（symmetric） M是一个对称矩阵当且仅当对于所有的i 和j 有$M(i, j)=M(j, i)$。如图4-4e 所示。

例4-3 考察佛罗里达州的六个城市Gainesville, Jacksonville, Miami, Orlando,Tallahassee和 Tampa。将这六个城市从1至6进行编号。任意两个城市之间的距离可以用一个6×6的矩阵来表示。矩阵的第i行和第i列代表第i 个城市，$distance(i, j)$代表城市i 和城市j 之间的距离。图4-5给出了相应的矩阵。由于对于所有的i 和j 有$distance(i, j)= distance(j, i)$，所以该矩阵是一个对称矩阵。

	GN	JX	MI	OD	TL	TM
GN	0	73	333	114	148	129
JX	73	0	348	140	163	194
MI	333	348	0	229	468	250
OD	114	140	229	0	251	84
TL	148	163	468	251	0	273
TM	129	194	250	84	273	0

GN = Gainesville	OD = Orlando
JX = Jacksonville	TL = Tallahassee
MI = Miami	TM = Tampa

距离单位：公里

图4-5 城市距离矩阵

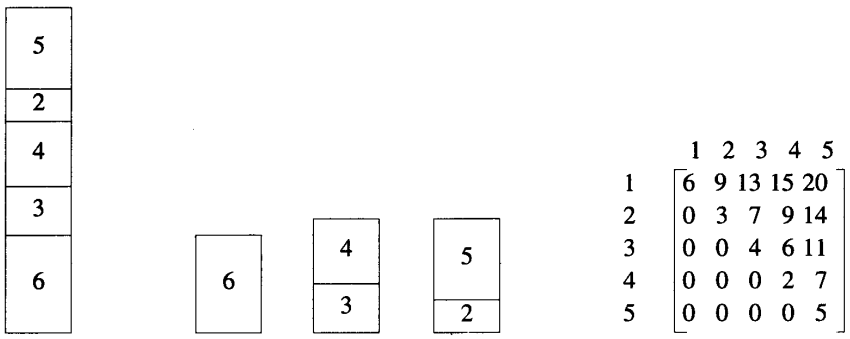


图4-6 堆栈折叠

a) 堆栈 b) 3个折叠堆栈 c) 矩阵

例4-4 假定有一个堆栈，其中有n 个纸盒，纸盒1位于栈顶，纸盒n 位于栈底。每个纸盒的宽度为w，深度为d。第i 个纸盒的高度为h_i。堆栈的体积为$w*d*\sum_{i=1}^{n} h_i$。在堆栈折叠（stack folding）问题中，选择一个折叠点i 把堆栈分解成两个子堆栈，其中一个子堆栈包含纸盒 1至i，另一个子堆栈包含纸盒i+1至n。重复这种折叠过程，可以得到若干个堆栈。如果创建了 s 个堆栈，则

这些堆栈所需要的空间宽度为$s*w$，深度为d，高度h为最高堆栈的高度。s个堆栈所需要的空间容量为$s*w*d*h$。由于h是第i至第j纸盒所构成的堆栈的高度(其中$i \le j$)，因此h的可能取值可由$n \times n$矩阵H给出，其中对于$i>j$有$H(i, j)=0$。即有$h=\sum_{k=1}^{j} h_k$，$i \le j$。由于每个纸盒的高度可以认为是大于0，所以$H(i, j)=0$代表一个不可能的高度。图4-6a给出了一个五个纸盒的堆栈。每个矩形中的数字代表纸盒的高度。图4-6b给出了五个纸盒堆栈折叠成三个堆栈后的情形，其中最大堆栈的高度为7。矩阵H是一个上三角矩阵，如图4-6c所示。

4.3.2 对角矩阵

可以采用如下所示的二维数组来描述一个元素类型为T的n×n对角矩阵D：

```
T d[n][n]
```

使用数组元素d[i-1][j-1]来表示矩阵元素$D(i, j)$，这种描述形式所需要的存储空间为n^2*sizeof(T)。由于一个对角矩阵最多包含n个非0元素，所以可以采用如下所示的一维数组来描述对角矩阵：

```
T d[n]
```

使用数组元素d[i-1]来表示矩阵元素$D[i, i]$。根据对角矩阵的定义，所有未在一维数组中出现的矩阵元素均为0。可以采用程序4-17所示的C++类DiagonalMatrix来实现这种描述。

程序4-17 DiagonalMatrix类

```
template<class T>
class DiagonalMatrix {
  public:
    DiagonalMatrix(int size = 10)
      {n = size; d = new T [n];}
    ~DiagonalMatrix() {delete [] d;} // 析构函数
    DiagonalMatrix<T>&
      Store(const T& x, int i, int j);
    T Retrieve(int i, int j) const;
  private:
    int n; //矩阵维数
    T *d;  // 存储对角元素的一维数组
};
template<class T>
DiagonalMatrix<T>& DiagonalMatrix<T>:: Store(const T& x, int i, int j)
{// 把x存为 D(i,j).
  if (i < 1 || j < 1 || i > n || j > n)
     throw OutOfBounds();
  if (i != j && x != 0) throw MustBeZero();
  if (i == j) d[i-1] = x;
  return *this;
}

template <class T>
T DiagonalMatrix<T>::Retrieve(int i, int j) const
{// 返回D(i,j).
  if (i < 1 || j < 1 || i > n || j > n)
```

```
    throw OutOfBounds();
  if (i == j) return d[i-1];
  else return 0;
}
```

对于存储和搜索操作，提供了两个不同的函数（而不是靠重载操作符（）来完成）。在存储一个值时，必须保证不会在非对角线位置放置一个非 0值，而在搜索一个值，没有必要检查对角线以外的值，因此有必要对这两种情形区别对待。程序 4-17中Store和Retrieve的复杂性均为Θ(1)。

4.3.3 三对角矩阵

在一个$n \times n$ 三对角矩阵T 中，非0元素排列在如下的三条对角线上：

1) 主对角线——$i = j$。

2) 主对角线之下的对角线（称低对角线）——$i = j+1$。

3) 主对角线之上的对角线（称高对角线）——$i = j-1$。

这三条对角线上的元素总数为$3n-2$，故可以使用一个拥有$3n-2$个位置的一维数组来描述T，因为仅需要存储三条对角线上的元素。考察图4-4b 所示的4×4三对角矩阵，三条对角线上共有10个元素。如果把这些元素逐行映射到t中，则有t[0:9]=[2,1,3,1,3,5,2,7,9,0]；如果逐列映射到t中，则有t=[2, 3, 1, 1, 5, 3, 2, 9, 7, 0]；如果按照对角线的次序（从最下面的对角线开始）进行映射，则有t=[3, 5, 9, 2, 1, 2, 0, 1, 3, 7]。正如我们所看到的，可以有三种不同的选择来进行T到t的映射。每一种映射方式都要求有相对应的 Store和Retrieve函数。程序 4-18定义了C++类TridiagonalMatrix，其中采用了对角线映射方式。

程序4-18 TridiagonalMatrix类

```
template<class T>
class TridiagonalMatrix {
  public:
    TridiagonalMatrix(int size = 10)
      {n = size; t = new T [3*n-2];}
    ~TridiagonalMatrix() {delete [] t;}
    TridiagonalMatrix<T>& Store (const T& x, int i, int j);
    T Retrieve(int i, int j) const;
  private:
    int n; // 矩阵维数
    T *t;  // 存储三对角矩阵的一维数组
};
template<class T>
TridiagonalMatrix<T>& TridiagonalMatrix<T>:: Store(const T& x, int i, int j)
{// 把x存为 T(i,j)
  if ( i < 1 || j < 1 || i > n || j > n)
    throw OutOfBounds();
  switch (i - j) {
    case 1: // 低对角线
      t[i - 2] = x; break;
    case 0: // 主对角线
```

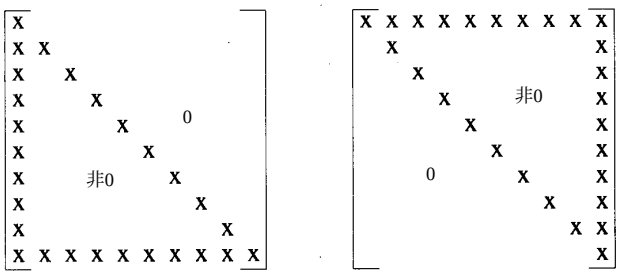
```
        t[n + i - 2] = x; break;
      case -1: // 高对角线
        t[2 * n + i - 2] = x; break;
      default: if(x != 0) throw MustBeZero();
      }
    return *this;
}
template <class T>
T TridiagonalMatrix<T>::Retrieve(int i, int j) const
{// 返回T(i,j)
  if ( i < 1 || j < 1 || i > n || j > n)
    throw OutOfBounds();

  switch (i - j) {
    case 1: //低对角线
        return t[i - 2];
    case 0: // 主对角线
        return t[n + i - 2];
    case -1: // 高对角线
        return t[2 * n + i - 2];
    default: return 0;
    }
}
```

4.3.4 三角矩阵

在一个三角矩阵中，非0元素都位于图4-7所示的“非0”区域。在一个下三角矩阵中，非0区域的第一行有1个元素，第二行有2个元素，…，第n行有n个元素；而在一个上三角矩阵中，非0区域的第一行有n个元素，第二行有n-1个元素，…，第n行有1个元素。对于这两种情况，非0区域的元素总数均为：

$$\sum_{i=1}^{n} i = n(n+1)/2$$



a) b)

图4-7 下三角矩阵和上三角矩阵

a) 下三角矩阵 b) 上三角矩阵

两种三角矩阵都可以用一个大小为$n(n+1)/2$的一维数组来进行描述。考察把一个下三角矩阵L映射到一个一维数组l，可采用按行和按列两种不同的方式进行映射。如果按行的方式进行映射，则对于图4-4c的4×4下三角矩阵可得到$l[0:9]$=(2, 5, 1, 0, 3, 1, 4, 2, 7, 0)；若按列的方式进行映射，则得到l=(2, 5, 0, 4, 1, 3, 2, 1, 7, 0)。

考察一个下三角矩阵中的元素$L(i, j)$。如果$i<j$，则$L(i, j)=0$；如果$i \geq j$，则$L(i, j)$位于非0区域。在按行映射方式中，在元素$L(i, j)$之前共有$\sum_{k=1}^{i-1} k+j-1=i(i-1)/2+j-1$个元素位于非0区域，这个表达式同时给出了$L(i, j)$在$l$中的位置。采用这个公式，可得到程序4-19所示的Store和Retrieve函数，二者的时间复杂性均为$\Theta(1)$。

程序4-19 LowerMatrix类

```
template<class T>
class LowerMatrix {
  public:
    LowerMatrix(int size = 10)
      {n = size; t = new T [n*(n+1)/2];}
    ~LowerMatrix() {delete [] t;}
    LowerMatrix<T>& Store(const T& x, int i, int j);
    T Retrieve(int i, int j) const;
  private:
    int n; //  矩阵维数
    T *t;  // 存储下三角矩阵的一维数组
};
template<class T>
LowerMatrix<T>& LowerMatrix<T>:: Store(const T& x, int i, int j)
{// 把x 存为 L(i,j).
  if ( i < 1 || j < 1 || i > n || j > n)
     throw OutOfBounds();
  // 当且仅当 i ≥ j 时(i,j) 位于下三角
  if (i >= j) t[i*(i-1)/2+j-1] = x;
  else if (x != 0) throw MustBeZero();
  return *this;
}

template <class T>
T LowerMatrix<T>::Retrieve(int i, int j) const
{//返回 L(i,j).
  if ( i < 1 || j < 1 || i > n || j > n)
     throw OutOfBounds();
  // 当且仅当 i ≥ j 时(i,j) 位于下三角
  if (i >= j) return t[i*(i-1)/2+j-1];
  else return 0;
}
```

4.3.5 对称矩阵

一个$n \times n$的对称矩阵可以用一个大小为$n(n+1)/2$的一维数组来描述，可参考三角矩阵的存储模式来存储矩阵的上三角或下三角。可以根据已存储的元素来推算出未存储的元素。

练习

20. 1) 扩充DiagonalMatrix类（见程序4-17），增加以下共享成员：输入、输出、加、减、乘和矩阵转置函数。注意每种函数所得到的结果是一个用一维数组表示的对角矩阵。重载操作符<<，>>，+，－和*。

2) 试测试代码。

3) 每个函数的时间复杂性是多少？

21. 1) 扩充TridiagonalMatrix类（见程序4-18），增加以下共享成员：输入、输出、加、减、乘和矩阵转置函数。重载适当的C++操作符。

2) 试测试代码。

3) 每个函数的时间复杂性是多少？

22. 1) 设计一个C++类TriByCols，它把一个$n \times n$ 的三对角矩阵按列的方式映射到一个大小为$3n$-2的一维数组。设置以下共享成员：输入、输出、存储、搜索、加、减和矩阵转置函数。

2) 试测试代码。

3) 每个函数的时间复杂性是多少？

23. 对于类TriByRows完成练习22，它把一个$n \times n$ 的三对角矩阵按行的方式映射到一个大小为$3n$-2的一维数组。

24. 两个三对角矩阵的乘积仍然是一个三对角矩阵吗？

25. 对于一个上三角矩阵，模仿程序4-19设计一个C++类UpperMatrix。

26. 扩充类LowerMatrix，增加一个共享成员函数，该函数可用来执行两个下三角矩阵的加法运算。函数的时间复杂性是多少？

27. 编写一个函数，该函数能把一个下三角矩阵（类型为 LowerMatrix）转置成一个上三角矩阵（类型为UpperMatrix）。函数的时间复杂性是多少？

28. 如图4-8所示，在一个$n \times n$ 的C- 矩阵中，除第一行、第n 行和第一列外，其他的元素均为0。一个C- 矩阵最多有$3n$-2个非0项。可把一个C- 矩阵压缩存储到一个一维数组，方法是首先存储第一行，然后是第n 行，最后是第一列中的剩余元素。

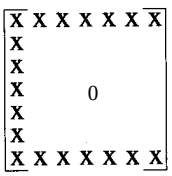
1) 给出一个4×4的C-矩阵样例及其压缩存储格式。

2) 按照上述思想设计一个C++类CMatrix，它用一个一维数组t描述一个$n \times n$的C-矩阵，要求提供两个共享成员函数Store和Retrieve。

3) 使用适当的测试数据来测试代码。

29. 编写一个对两个下三角矩阵（类型为LowerMatrix）进行乘法运算的函数，所得到的结果用一个二维数组来描述。函数的时间复杂性是多少？

30. 编写函数，对一个下三角矩阵和一个上三角矩阵进行乘法运算（两个矩阵都是按行的方式存储在一个一维数组中），所得到的结果用一个二维数组来描述。函数的时间复杂性是多少？



x表示可能为非0
所有其他项均为0

图4-8 C- 矩阵

31. 假定按行的方式把对称矩阵的下三角区域存储在一个一维数组中。试设计一个 C++类LowSymmetric，要求提供两个共享成员函数 Store 和Retrieve，这两个函数的时间复杂性应为$\Theta(1)$。

32. 令A和B是两个$n\times n$下三角矩阵。两个矩阵的下三角区域中的元素总数为 n(n+1)。设计一种存储模式，用以在一个数组$d[n+1][n]$中同时存储这两个下三角区域。（提示：如果把A的下三角区域与B^T的上三角区域进行合并，就可以得到一个$(n+1)\times n$的矩阵）。对于A和B，分别给出其相应的存储函数和搜索函数。每个函数的复杂性应为$\Theta(1)$。

*33. 带状方阵（square hand matrix）$D_{n,a}$是一个$n\times n$的矩阵，其中所有非0元素都落在包围主对角线的带状区域中。如图4-9所示，带状区域包含主对角线以及主对角线两边的a-1条对角线。

1) 在$D_{n,a}$的带状区域中有多少个元素？

2) 对于$D_{n,a}$带状区域中的元素$d_{i,j}$来说，i和j之间应满足什么关系？

3) 假定按对角线的方式把$D_{n,a}$的带状区域映射到一个一维数组b中，从最下面的一条对角线开始映射。例如，图4-9中的带状方阵$D_{4,3}$可按如下形式来存储：

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]	b[10]	b[11]	b[12]	b[13]
9	7	8	3	6	6	0	2	8	7	4	9	8	4
d_{20}	d_{31}	d_{10}	d_{21}	d_{32}	d_{00}	d_{11}	d_{22}	d_{33}	d_{01}	d_{12}	d_{23}	d_{02}	d_{13}

给出一个公式，用来计算带状区下方元素$d_{i,j}$的存储位置（在上例中d_{10}的位置为2）。

4) 使用3) 中的映射方法设计一个C++类SquareBand，要求提供两个共享成员函数 Store和Retrieve，这两个函数的时间复杂性分别是多少？

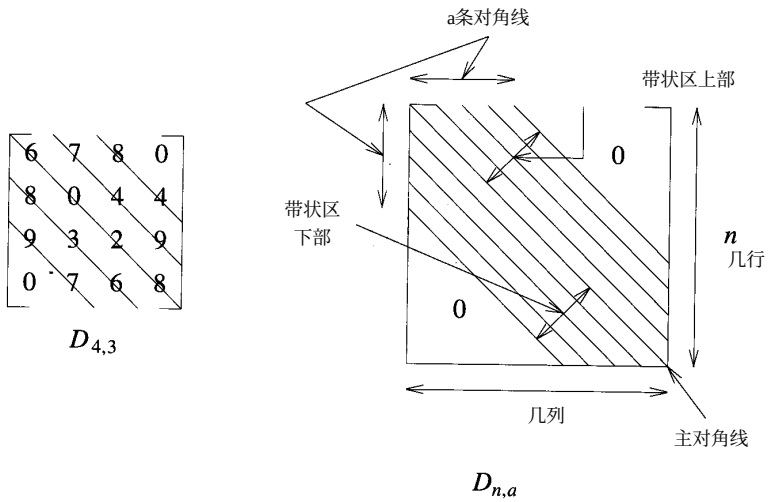


图4-9 带状方阵

34. 一个$n\times n$的矩阵T是一个等对角矩阵（Toeplitz matrix）当且仅当对于所有的i和j有$T(i,j)=T(i\text{-}1,j\text{-}1)$，其中$i>1$，$j>1$。

1) 证明一个等对角矩阵最多有$2n$-1个不同的元素 。

2) 给出一种映射模式，把一个等对角矩阵映射到一个大小为$2n$-1的一维数组中。

3) 采用2) 中的映射模式设计一个C++ 类Toeplitz，用一个大小为$2n$-1 的一维数组来存储等对角矩阵。要求给出两个共享成员函数 Store 和Retrieve，这两个函数的时间复杂性应为$\Theta(1)$。

4) 编写一个共享成员函数，用来对两个按2) 中所给方式存储的等对角矩阵进行乘法运算，

所得到的结果存储在一个二维数组中。该函数的时间复杂性是多少？

35. 一个$n\times n$的矩阵M是一个反对角矩阵（antidiagonal）当且仅当对于所有满足$i+j\neq n+1$的i和j有$M(i,j)=0$。

1) 给出一个4×4反对角矩阵的样例。

2) 证明反对角矩阵M最多有n个非0元素。

3) 设计一种映射模式，用来把一个反对角矩阵映射到一个大小为n的一维数组之中。

4) 采用3) 中的映射模式设计一个C++类AntiDiagonal，要求给出两个共享成员函数Store和Retrieve。

5) Store 和Retrieve 的时间复杂性分别是多少？

4.4 稀疏矩阵

4.4.1 基本概念

如果一个$m\times n$矩阵中有“许多”元素为0，则称该矩阵为稀疏矩阵（sparse）。不是稀疏的矩阵被称为稠密矩阵（dense）。在稀疏矩阵和稠密矩阵之间并没有一个精确的界限。$n\times n$的对角矩阵和三对角矩阵都是稀疏矩阵，二者都有$O(n)$个非0元素和$O(n^2)$个0元素。一个$n\times n$的三角矩阵是稀疏矩阵吗？它至少有$n(n-1)/2$个0元素，最多有$n(n+1)/2$个非0元素。在本节中我们规定若一个矩阵是稀疏矩阵，则其非0元素的数目应小于$n^2/3$，在有些情况下应小于$n^2/5$，因此可将把三角矩阵视为稠密矩阵。

诸如对角矩阵和三对角矩阵这样的稀疏矩阵，其非0区域的结构很有规律，因此可以设计一个很简单的存储结构，该存储结构的大小就等于矩阵非0区域的大小。本节中主要考察具有不规则非0区域的稀疏矩阵。

例4-5 某超级市场正在开展一项关于顾客购物品种的研究。为了完成这项研究，收集了1000个顾客的购物数据，这些数据被组织成一个矩阵*purchases*，其中*purchases*(i,j)表示顾客j所购买的商品i的数量。假定该超级市场有10 000种不同的商品，那么*purchases*将是一个10 000×1000的矩阵。如果每个顾客平均购买了20种不同商品，那么在10 000 000个矩阵元素将大约只有20 000个元素为非0，并且非0元素的分布没有很明确的规律。

超级市场有一个10 000×1的价格矩阵*price*，*price*(i)代表商品i的单价。矩阵*spent*=*purchases*^{T}**price*是一个1000×1的矩阵，它给出每个顾客所花费的购物资金。如果用一个二维数组来描述矩阵*purchases*，那么将浪费大量的存储空间，并且计算*spent*时也将耗费更多的时间。

4.4.2 数组描述

可以按行主次序把无规则稀疏矩阵映射到一维数组中。例如图4-10a中的4×8矩阵按行主次序进行存储可得到：2，1，6，7，3，9，8，4，5。

为了重建矩阵结构，必须记录每个非0元素所在的行号和列号，所以在把稀疏矩阵的非0元素映射到数组中时必须提供三个域：row（矩阵元素所在行号）、col（矩阵元素所在列号）和value（矩阵元素的值）。为此，定义如下所示的模板类Term：

```
template <class T>
class Term {
  private:
```

```
    int row, col;
    T value;
};
```

如果a是一个类型为Term的数组，那么图4-10a 中的稀疏矩阵按行主次序存储到a 中所得到的结果如图4-10b 所示。除了存储数组a 以外，还必须存储矩阵行数、矩阵列数和非0项的数目。所以存储图4-10a 中的九个非0元素所需要的存储器字节数为21*sizeof(int)+9*sizeof(T)。如果用一个 4×8的数组来描述这个矩阵，则需要的字节数为 32*sizeof(T)。假定 T是int 类型且sizeof(T)=2，那么图4-10b 中的描述需60个字节，而采用4×8的数组则需要64个字节。对于这个例子用一维数组来进行存储并没有节省出多少空间。不过，对于例 4-5中的矩阵purchase，其一维数组描述需60 000*sizeof(T)个字节，而二维数组描述则需10 000 000*sizeof(T)个字节。如果一个整数占2个字节，则节省的空间为19 880 000字节。

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

a)

a[]	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7	3	9	8	4	5

b)

图4-10 稀疏矩阵及其数组描述

a) 4×8矩阵 b) 数组描述

稀疏矩阵的数组描述并不能使Store和Retrieve函数的执行效率也得到相应提高。Store函数所需要的时间为O(非0元素数目)，因为可能需要移动这么多元素以便为新元素留出空间。如果采用折半搜索，则Retrieve函数所需要的时间为O(log[非0元素数目])。如果采用标准的二维数组来描述矩阵，这两种函数所需要的时间均为 $\Theta(1)$。不过，诸如转置、加和乘这样的矩阵操作可以得到高效地执行。

1. 类SparseMatrix

可以定义一个类SparseMatrix（见程序4-20），用来把稀疏矩阵按行主次序映射到一维数组中。这个类是Term的友元。私有成员rows，cols和terms代表稀疏矩阵中的行数、列数和非 0元素的数目。MaxTerms是数组a（用来存储非0元素）的容量。在定义共享成员时，没有定义加法操作符+，因为它会创建一个临时结果，这个临时结果必须复制到所返回的环境才可以使用。由于SparseMatrix的复制构造函数将会复制每一个元素，因此操作符 +中的复制代价太大，如使用Add函数则可以避免，它将会告诉你把结果送到哪里去。

程序4-20 类SparseMatrix

```
template<class T>
class SparseMatrix
{
  friend ostream& operator<< (ostream&, const SparseMatrix<T>&);
  friend istream& operator>> (istream&, SparseMatrix<T>&);
  public:
    SparseMatrix(int maxTerms = 10);
    ~SparseMatrix() {delete [] a;}
    void Transpose(SparseMatrix<T> &b) const;
```

```
    void Add(const SparseMatrix<T> &b, SparseMatrix<T> &c) const;
  private:
    void Append(const Term<T>& t);
    int rows, cols;  //矩阵维数
    int terms;  // 非0元素数目
    Term<T> *a;   // 存储非0元素的数组
    int MaxTerms; // 数组a的大小;
};
```

程序4-21给出了SparseMatrix的构造函数，程序4-22 给出了输入操作符<< 和输出操作符>>的代码，其中operator<<和operator>>必须是类Term的友元。operator>>按行主次序输出稀疏矩阵的非0元素，而operator<<则按行主次序输入稀疏矩阵并建立内部数组描述。 operator<<和operator>>的时间复杂性均为Θ(terms)。练习37和38要求对程序4-22中的代码进行细化。

程序4-21 类SparseMatrix的构造函数

```
template<class T>
SparseMatrix<T>::SparseMatrix(int maxTerms)
{// 稀疏矩阵的构造函数
  if (maxTerms < 1) throw BadInitializers();
  MaxTerms = maxTerms;
  a = new Term<T> [MaxTerms];
  terms = rows = cols = 0;
}
```

程序4-22 类SparseMatrix的输入和输出函数

```
// 重载<<
template <class T>
ostream& operator<<(ostream& out, const SparseMatrix<T>& x)
{// 把*this 送至输出流
  // 输出矩阵的特征
  out << "rows = " << x.rows << " columns = " << x.cols  << endl;
  out << "nonzero terms = " << x.terms << endl;
  // 输出非0元素，每行1个
  for (int i = 0; i < x.terms; i++)
    out << "a(" << x.a[i].row << ',' << x.a[i].col << ") = " << x.a[i].value << endl;
  return out;
}
// 重载>>
template<class T>
istream& operator>>(istream& in, SparseMatrix<T>& x)
{// 输入一个稀疏矩阵
  // 输入矩阵的特征
  cout << "Enter number of rows, columns, and terms" << endl;
  in >> x.rows >> x.cols >> x.terms;
  if (x.terms > x.MaxTerms) throw NoMem();
  // 输入矩阵元素
```

```
  for (int i = 0; i < x.terms; i++) {
    cout << "Enter row, column, and value of term " << (i + 1) << endl;
    in >> x.a[i].row >> x.a[i].col >> x.a[i].value;
    }
  return in;
}
```

如果输入的元素数目超出了数组*this.a 的大小，则operator>> 将引发一个异常。一种处理异常的方法是删除数组a，然后使用new 重新分配一个更大的数组。

2. 矩阵转置

程序4-23给出了函数Transpose的代码。转置后的矩阵被返回到b 中。首先验证b 中是否有足够的空间来存储被转置矩阵的非 0元素。如果空间不足，要么重新分配一个更大的数组 b.a，要么引发一个异常。在此程序中引发了一个异常。如果 b中有足够的空间来容纳转置矩阵，则创建两个数组ColSize和RowNext。ColSize[i]是指矩阵第i列中的非0元素数，RowNext[i]代表转置矩阵第i行的下一个非0元素在b中的位置。对ColSize的计算是在前两个for循环中通过简单地检查每个矩阵元素来完成的。对RowNext的计算是在第三个for循环中完成的。RowNext[i]的初值为转置矩阵中第0行至第i-1列中的元素数目（即原矩阵中第0列至第i-1列中的元素数目）。在最后一个for循环中，非0元素被复制到b中相应位置。

程序4-23 转置一个稀疏矩阵

```
template<class T>
void SparseMatrix<T>:: Transpose(SparseMatrix<T> &b) const
{// 把 *this 的转置结果送入 b
  // 确信 b 有足够的空间
  if (terms > b.MaxTerms) throw NoMem();
  // 设置转置特征
  b.cols = rows;
  b.rows = cols;
  b.terms = terms;
  // 初始化
  int *ColSize, *RowNext;
  ColSize = new int[cols + 1];
  RowNext = new int[rows + 1];
  // 计算 *this每一列的非0元素数
  for (int i = 1; i <= cols; i++) // 初始化
    ColSize[i] = 0;
  for (int = 0; i < terms; i++)
    ColSize[a[i].col]++;
  // 给出 b中每一行的起始点
  RowNext[1] = 0;
  for (int i = 2; i <= cols; i++)
    RowNext[i] = RowNext[i - 1] + ColSize[i - 1];

  // 执行转置操作
  for (int i = 0; i < terms; i++) {
    int j = RowNext[a[i].col]++; // 在b中的位置
```

```
        b.a[j].row = a[i].col;
        b.a[j].col = a[i].row;
        b.a[j].value = a[i].value;
        }
    }
```

尽管程序4-23比按二维数组存储矩阵（见程序 2-22）更复杂，但对于有很多个 0元素的矩阵，程序4-23要快得多。不难发现，对于例4-5中的矩阵purchase，采用一维数组描述来完成转置操作要比按二维数组描述完成转置操作更快。 Transpose的时间复杂性为O(cols+terms)。

3. 两个矩阵相加

在两个矩阵相加的代码中使用了程序 4-24中的函数Append，它把一个非0项添加到一个稀疏矩阵的非0项数组的尾部。该函数的时间复杂性为 (1)。

程序4-24 添加一个非0元素

```
template<class T>
void SparseMatrix<T>::Append(const Term<T>& t)
{// 把一个非0元素 t添加到 *this之中
  if (terms >= MaxTerms) throw NoMem();
  a[terms] = t;
  terms++;
}
```

程序4-25中的代码对两个矩阵 *this和b执行加法操作，所得到的结果放入 c中。矩阵c的获得是通过从左至右依次扫描两个矩阵中的元素来实现的。在扫描过程中使用了两个游标： ct（矩阵*this的游标）和cb（矩阵b的游标）。在每一次while循环过程中，需要确定(*this).a[ct]的位置或是在b.a[cb]之前，或是相同，或是在b.a[cb]之后。判断的方法是分别计算出这两项的索引（行主次序）。在Add函数中， *this的元素索引由indt给出，b的元素索引由indb给出。

程序4-25 两个稀疏矩阵相加

```
template<class T>
void SparseMatrix<T>::Add(const SparseMatrix<T> &b, SparseMatrix<T> &c) const
{// 计算 c = (*this) + b.
  // 验证可行性
  if (rows != b.rows || cols != b.cols)
    throw SizeMismatch(); // 不能相加
  // 设置结果矩阵c的特征
  c.rows = rows;
  c.cols = cols;
  c.terms = 0; // 初值
  // 定义*this 和 b的游标
  int ct = 0, cb = 0;
  // 在 *this 和 b 中遍历
  while (ct < terms && cb < b.terms) {
  // 每一个元素的行主索引
    int indt = a[ct].row * cols + a[ct].col;
    int indb = b.a[cb].row * cols + b.a[cb].col;
```

```
        if (indt < indb) {// b 的元素在后
        c.Append(a[ct]);
            ct++;} // *this的下一个元素
        else {if (indt == indb) {// 位置相同
                //仅当和不为0时才添加到 c中
                if (a[ct].value + b.a[cb].value) {
                   Term<T> t;
                   t.row = a[ct].row;
                   t.col = a[ct].col;
                   t.value = a[ct].value + b.a[cb].value;
                   c.Append(t);}
                ct++; cb++;}  // *this 和 b的下一个元素
             else {c.Append(b.a[cb]); cb++;} // b的下一个元素
            }
        }
      // 复制剩余元素
      for (; ct < terms; ct++)
        c.Append(a[ct]);
      for (; cb < b.terms; cb++)
        c.Append(b.a[cb]);
    }
```

函数Add中的while循环最多会执行terms+b.terms次，因为在循环过程中ct,cb会不断增值。第一个for循环最多会执行terms次，而第二个for循环最多会执行O(b.terms)次。另外，每个循环中的每一次循环只需要常量时间。因此函数Add的时间复杂性为O(terms+b.terms)。如果用二维数组来描述每个矩阵，则两个矩阵相加需耗时O(rows*cols)。当terms+b.terms远小于rows*cols时，稀疏矩阵的加法执行效率将大大提高。

4.4.3 链表描述

用一维数组来描述稀疏矩阵所存在的缺点是：当我们创建这个一维数组时，必须知道稀疏矩阵中的非0元素总数。虽然在输入矩阵时这个数是已知的，但随着矩阵加法、减法和乘法操作的执行，非0元素的数目会发生变化，因此如果不实际计算，很难精确地知道非0元素的数目。正因为如此，只好先估计出每个矩阵中的非0元素数目，然后用它作为数组的初始大小。在设计SparseMatrix类时就采用了这样的策略。在该类的代码中，当结果矩阵非0元素的数目超出所估计的数目时将引发一个异常。不过也可以重写这些代码，在非0元素的数目超出所估计的数目时分配一个新的、更大的数组，然后从老数组中把元素复制出来并删除老数组。这种额外工作将使算法的效率降低，并留下了新数组到底应该取多大的问题。如果新数组不够大，还得重复上述分配和复制过程；如果新数组太大，又会浪费很多空间。一种解决办法就是采用基于指针的描述。这种方法需要额外的指针存储空间，但可以节省对数组描述中其他一些信息的存储。最重要的是，它可以避免存储的再分配以及部分结果的复制。

1. 描述

链表描述的一种可行方案是把每行的非0元素串接在一起，构成一个链表，如图4-11所示。图中每个非阴影节点代表稀疏矩阵中的一个非0元素，它有三个域：col（非0元素所在列号）、value（非0元素的值）和link（指向下一个非阴影节点的指针）。仅当矩阵某行中至

少包含一个非0元素才会为该行创建一个链表。在行链表中，每个节点按其 col值的升序进行排列。

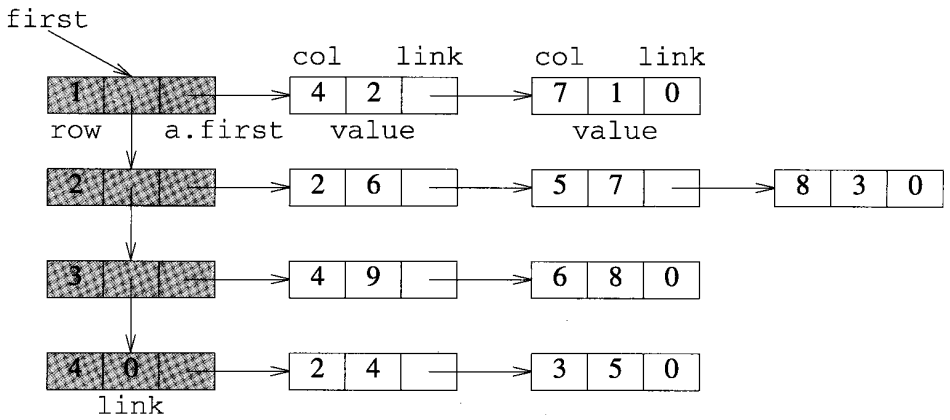


图4-11 图4-10a 中矩阵的链表描述

用另外一个链表把所有的行链表（即非阴影链表）收集在一起，如图 4-11中的阴影节点所示。每个阴影节点也有三个域：row（相应行链表的行号）、link（指向下一个阴影节点的指针）和a（指向行链表， a.first 指向行链表中的第一个节点）。各阴影节点按其row值的升序排列。每个阴影节点可以被视为一个行链表的头节点，因此阴影链表可以被视为头节点链表。空的头节点链表代表没有非0元素的矩阵。

2. 链表节点类型

图4-11中非阴影节点所对应的链表可以被定义为Chain<CNode<T>>类型，其中CNode的定义见程序 4-26。阴影节点所对应的链表可以被定义为 Chain<HeadNode<T>>类型，其中HeadNode<T>的定义见程序4-26。

程序4-26 稀疏矩阵描述中所使用的链表节点

```
template<class T>
class CNode {
  public:
    int operator !=(const CNode<T>& y)
      {return (value != y.value);}
    void Output(ostream& out) const
      {out << "column " << col << " value " << value;}
  private:
    int col;
    T value;
};
template<class T>
ostream& operator<<(ostream& out, const CNode<T>& x)
  {x.Output(out); out << endl; return out;}
template<class T>
class HeadNode {
```

```
  public:
    int operator !=(const HeadNode<T>& y)
      {return (row != y.row);}
    void Output(ostream& out) const
      {out << "row " << row;}
  private:
    int row;
    Chain<CNode<T> > a; //行链表
};
template<class T>
ostream& operator<<(ostream& out, const HeadNode<T>& x)
  {x.Output(out); out << endl; return out;}
```

3. 类LinkedMatrix

现在可以来定义类LinkedMatrix，见程序4-27。其中共享函数的实现以及操作符 >>和<<的重载都要求LinkedMatrix，operator>>和operator<<是类CNode和HeadNode的友元。

程序4-27 用链表描述稀疏矩阵的类定义

```
template<class T>
class LinkedMatrix
{
  friend ostream& operator<<
       (ostream&, const LinkedMatrix<T>&);
  friend istream& operator>>
       (istream&, LinkedMatrix<T>&);
  public:
    LinkedMatrix(){}
    ~LinkedMatrix(){}
    void Transpose(LinkedMatrix<T> &b) const;
    void Add(Const LinkedMatrix<T> &b, LinkedMatrix<T> &c) const;
  private:
    int rows, cols;       // 矩阵维数
    Chain<HeadNode<T> > a; // 头节点链表
};
```

4. 重载>>

程序4-28按行主次序输入所有的非0元素并创建图4-11所示的链表。它首先要求输入矩阵的维数以及非0元素的个数，然后输入各个非0元素并把它们收集到各行链表中。用变量H代表当前行链表的头节点。如果下一个非0元素不属于当前行链表，则将当前行链表添加到矩阵x的头节点链表x.a之中（虚设的第0行链表除外）。接下来，H被设置为指向一个新的行链表，同时将刚才那个非0元素添加到这个新的行链表之中。如果新的非0元素属于当前行链表，则只需简单地把它添加到链表H.a中即可。注意函数Append和Zero是3.4.3节所定义的扩充链表类的成员。Append在链表的尾部添加一个节点，而 Zero则把first置为0但并不删除链表中的节点。operator>>的复杂性为O(terms)。

程序4-28 输入一个稀疏矩阵

```
template<class T>
istream& operator>>(istream& in, LinkedMatrix<T>& x)
{// 从输入流中输入矩阵x
  x.a.Erase(); // 删除x中的所有节点
  // 获取矩阵特征
  int terms;  // 输入的元素数
  cout << "Enter number of rows, columns, and terms" << endl;
  in >> x.rows >> x.cols >> terms;
  // 虚设第0行
  HeadNode<T> H;  // 当前行的头节点
  H.row = 0;     // 当前行号
  // 输入 x的非0元素
  for (int i = 1; i <= terms; i++) {
    // 输入下一个元素
    cout << "Enter row, column, and value of term " << i << endl;
    int row, col;
    T value;
    in >> row >> col >> value;
    // 检查新元素是否属于当前行
    if (row > H.row) {// 开始一个新行
      // 如果不是第0行，则把当前行的头节点 H添加到头节点链表x.a之中
      if (H.row) x.a.Append(H);
      //为新的一行准备H
      H.row = row;
      H.a.Zero();}  // 置链表头指针 first=0
    // 添加新元素
    CNode<T> *c = new CNode<T>;
    c->col = col;
    c->value = value;
    H.a.Append(*c);
    }
  // 注意矩阵的最后一行
  if (H.row) x.a.Append(H);
  H.a.Zero();  // 置链表头指针为0
  return in;
}
```

5. 重载<<

为了输出用链表表示的稀疏矩阵，使用一个链表遍历器（见 3.4.4节）依次检查头节点链表中的每个节点。对于头节点链表中的每个节点，输出其相应的行链表。程序 4-29给出了操作符<<的实现代码。代码的时间复杂性与非0元素的数目呈正比。

程序4-29 输出一个稀疏矩阵

```
template<class T>
ostream& operator<<(ostream& out, const LinkedMatrix<T>& x)
```

```
{// 把矩阵 x 送至输出流
  ChainIterator<HeadNode<T> > p;// 头节点遍历器
  // 输出矩阵的维数
  out << "rows = " << x.rows << " columns = " << x.cols << endl;
  // 将 h 指向第一个头节点
  HeadNode<T> *h = p.Initialize(x.a);
  if (!h) {out << "No non-zero terms" << endl;
          return out;}
  // 每次输出一行
  while (h) {
    out << "row " << h->row << endl;
    out << h->a << endl;  //输出行链表
    h = p.Next();         // 下一个头节点
    }
  return out;
}
```

6. 函数Transpose

对于转置操作，可以采用箱子来从矩阵 *this 中收集位于结果矩阵同一行的非0元素。bin[i]是结果矩阵b 中第i 行非0元素所对应的链表。在程序 4-30的while 循环中，沿着输入矩阵 *this的头节点链表按行主次序依次检查每个非 0元素（在每个行链表中按从左至右的次序检查）。用链表遍历器p 来遍历头节点链表，用链表遍历器 q 遍历行链表。在按照上述次序遍历 *this的过程中，把遇到的每个非0元素添加到相应的箱子链表中。最后用一个 for循环来收集各箱子链表，并创建结果矩阵所需要的头节点链表。

程序4-30 转置一个稀疏矩阵

```
template<class T>
void LinkedMatrix<T>::
    Transpose(LinkedMatrix<T> &b) const
{// 转置 *this，并把结果放入b
  b.a.Erase(); // 删除b中所有节点
  // 创建用来收集b中各行元素的箱子
  Chain<CNode<T> > *bin;
  bin = new Chain<CNode<T> > [cols + 1];
  // 头节点遍历器
  ChainIterator<HeadNode<T> > p;
  // h 指向*this的第一个头节点
  HeadNode<T> *h = p.Initialize(a);
  // 把 *this的元素复制到箱子中
  while (h) { // 检查所有的行
    int r = h->row; // 行链表中的行数
    // 行链表遍历器
    ChainIterator<CNode<T> > q;
    // z指向行链表的第一个节点
    CNode<T> *z = q.Initialize(h->a);
    CNode<T> x;  // 临时节点
    // *this第r行中的元素变成b中第r列的元素
```

```
        x.col = r;
        //检查 *this中第r行的所有非0元素
        while (z) {// 遍历第 r行
          x.value = z->value;
          bin[z->col].Append(x);
          z = q.Next();  // 该行的下一个元素
          }
        h = p.Next(); // 继续下一行
        }
      // 设置b的维数
      b.rows = cols;
      b.cols = rows;
      // 装配 b的头节点链表
      HeadNode<T> H;
      // 搜索箱子
      for (int i = 1; i <= cols; i++)
        if (!bin[i].IsEmpty()) {// 转置矩阵的第 i 行
          H.row = i;
          H.a = bin[i];
          b.a.Append(H);
          bin[i].Zero();} // 置链表头指针为0
      H.a.Zero(); // 置链表头指针为0
      delete [] bin;
    }
```

while循环所需要的时间与非0元素的数目呈线性关系，而for循环所需要的时间与输入矩阵的列数呈线性关系。因此总的时间与这两个量的和呈线性关系。

练习45要求你实现Add函数以及其他基本函数。

练习

36. 对于一个按行主次序存储在一维数组中的稀疏矩阵，编写其 Store和Retrive函数。函数的时间复杂性分别是多少？

37. 细化程序4-22中的operator>>，要求验证：是否按行主次序输入链表，每个非 0元素的行号和列号是否有效，所输入的元素是否为非0。

38. 修改程序4-22中的operator>>，要求如果x.MaxSize<x.terms，则分配一个更大的数组来存储矩阵元素。

39. 编写类SparseMatrix的复制构造函数。

40. 修改程序4-24中的Append函数，要求如果当前数组 a 没有足够的空间，则分配一个更大的数组a。

41. 扩充类SparseMatrix（见程序4-20），增加两个共享成员函数——矩阵加和矩阵乘。

42. 假定按列主次序把一个稀疏矩阵映射到一个一维数组中

1) 给出图4-10a 中稀疏矩阵的描述。

2) 对按照这种方式存储的稀疏矩阵，编写相应的 Store和Retrieve函数。

3) Store和Retrieve函数的时间复杂性分别是多少？

*43. 编写一个函数，对两个存储在一维数组中的稀疏矩阵进行乘法运算。假定两个矩阵都

是按行主次序存储的。所得到的结果也要求按行主次序存储。

*44. 编写一个函数，对两个存储在一维数组中的稀疏矩阵进行乘法运算。假定两个矩阵都是按列主次序存储的。所得到的结果也要求按列主次序存储。

*45. 通过为下列操作增加共享成员扩充类LinkedMatrix：

1) 存储一个元素，已知行号、列号和元素的值。

2) 在矩阵中查找具有给定行号和列号的元素。

3) 两个稀疏矩阵相加。

4) 两个稀疏矩阵相减。

5) 两个稀疏矩阵相乘。

*46. 可以采用如下所述的链表来描述稀疏矩阵。链表节点中包括如下域： down，right，row，col和value。稀疏矩阵的每个非0元素都用一个节点来表示。0元素不必存储。所有的节点链接在一起形成两个循环链表。第一个链表——行链表——使用 right域按行的次序（每行按列号的次序）连接所有节点。第二个链表——列链表——使用down域按列的次序（每列按行号的次序）连接所有节点。这两个链表共享同一个头节点。此外，有一个附加的节点用来存储矩阵的维数。

1) 给出一个5×8的矩阵，其中正好有9个非0元素，并且在每一行和每一列中至少有一个非0元素。对于这个稀疏矩阵，给出其相应的链表描述。

2) 假定一个m×n 矩阵中有t 个非0元素，若按上述形式来描述该矩阵，则t应该有多小才能保证所需要的存储空间比使用一个m×n 数组要来得少？

3) 设计一个合适的外部描述（即可用来输入和输出），该描述不应要求输入0元素。

4) 采用2) 中的描述完成练习37。

5) 对于类的每个共享成员，给出其时间复杂性。这些复杂性与采用二维数组来描述矩阵所得到的复杂性相比有哪些不同？