



## 第7章 数组和矩阵

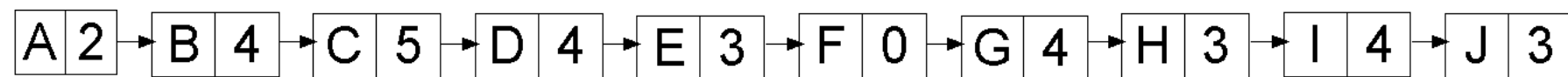
---

——改头换面：从数据结构的视角重新诠释这两个概念



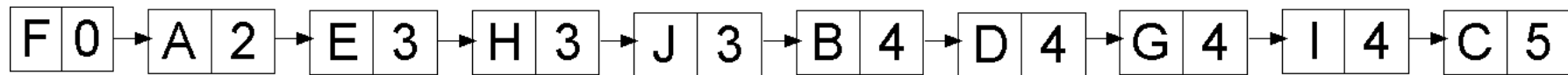
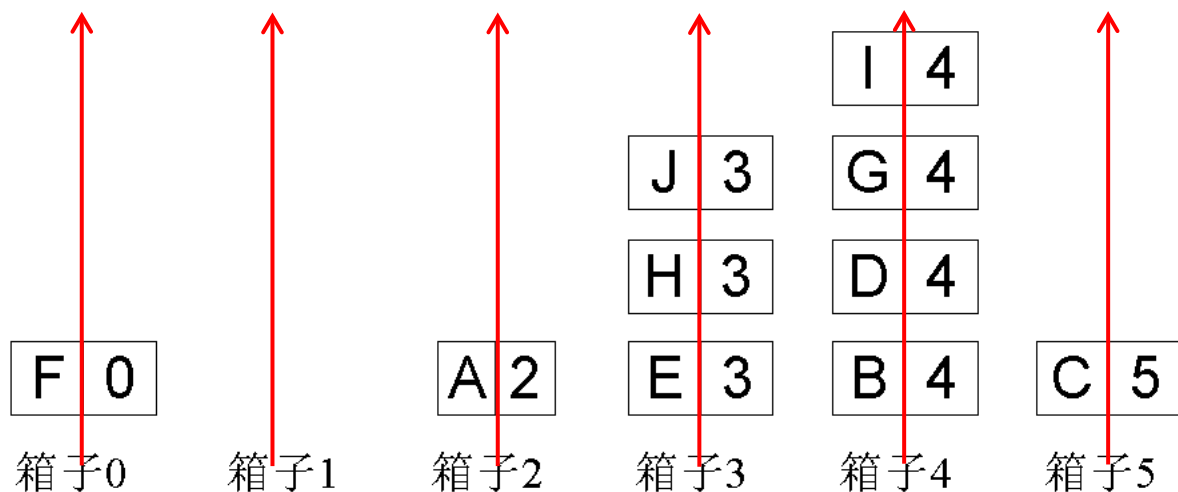
计算机学院

# 箱子排序的思想



## Step1:

将元素  
分配到  
箱子



## Step2:

遍历并连接箱子



# 箱子排序的优化

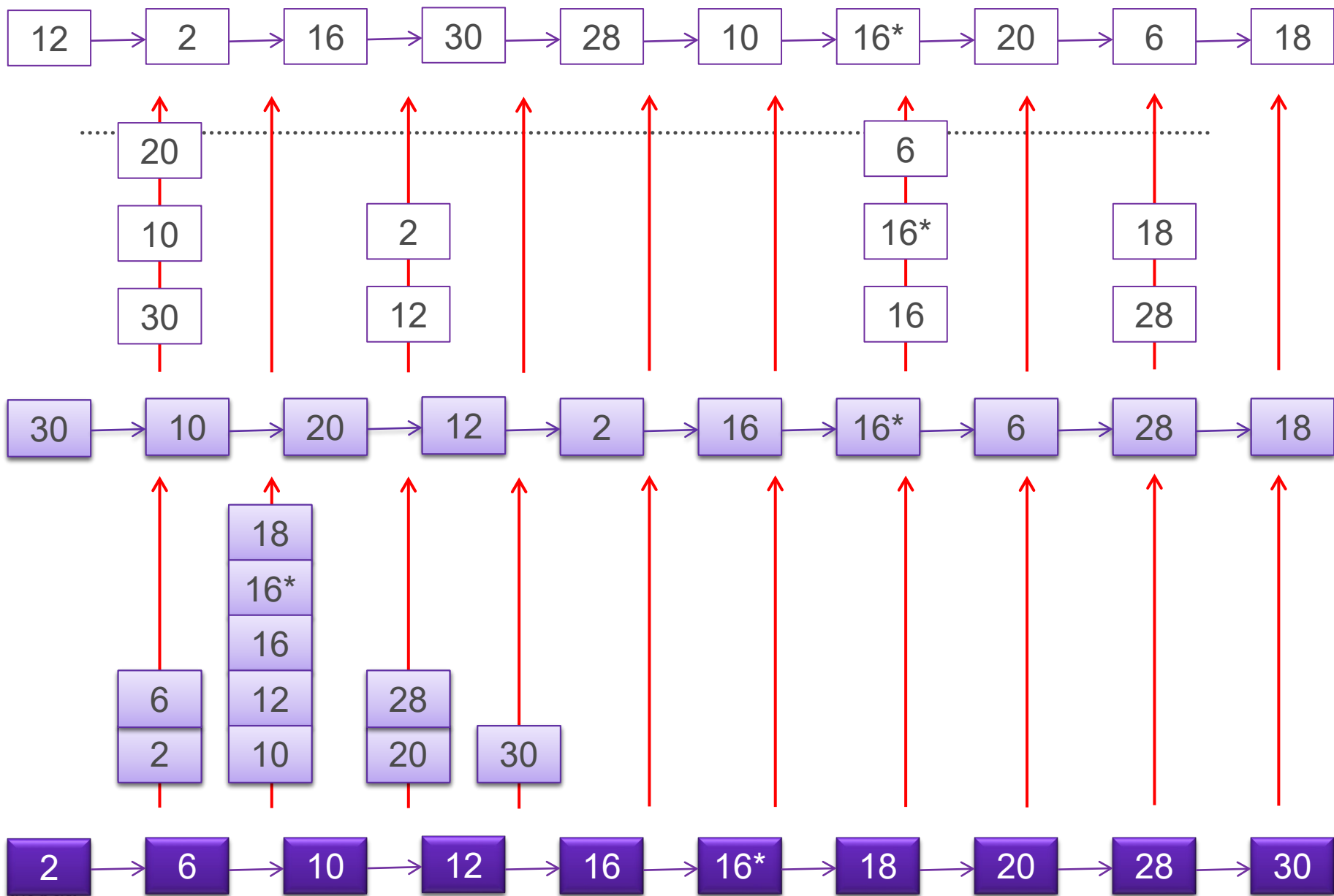
作为独立函数	作为Chain的成员函数
操作Node	操作指针
需要频繁地new和delete	无需频繁地new和delete
$2*n+range$	$n+2*range$
$\Theta(n+range)$	$\Theta(n+range)$



# 箱子排序难以解决的问题

1. 前提:  $\Theta(n + \text{range})$
2. 适用场景:  $n \gg \text{range}$ , 当  $n=1000, \text{range}=10$  时
  - $n + 2 * \text{range} = 1020$ ,  $n * (n-1) / 2 = 499500$
3. 不适用场景:  $n \ll \text{range}$ , 当  $n=10, \text{range}=1000$ 
  - $n + 2 * \text{range} = 2010$ ,  $n * (n-1) / 2 = 45$
4. 改进思路
  - 因为箱子排序的复杂程度与  $\text{range}$  密切相关, 所以考虑是否能以几组小箱子替代?





# 性能分析

---

- 当 $n=10$ ,  $range=100$ 时,
- 箱子排序:  $10+2*100=210$  (执行步)
- 基数排序:  $2*(10+2*10)=60$  (执行步)
- 简单排序:  $10*9/2=45$  (比较次数)



# 主要内容

---

- 数组
- 矩阵
- 特殊矩阵
  - 对角矩阵、三对角矩阵、三角矩阵、对称矩阵
- 稀疏矩阵
  - 数组描述
  - 链表描述



# 数组ADT

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)，并删除具有相同index值的数据对（若存在）

**Retrieve(index)**：返回索引值为index的数据对

}





# 例

- 接下来一周每天的高温（摄氏度数）：
  - $\text{high} = \{(\text{sunday}, 10), (\text{monday}, 12), (\text{tuesday}, 9), (\text{wednesday}, 10), (\text{thursday}, 13), (\text{friday}, 13), (\text{saturday}, 12)\}$
  - 数据对：（日期名—索引, 当天温度—值）
  - 将monday的温度改为15:  $\text{Store}(\text{monday}, 15)$
  - 获得friday的温度:  $\text{Retrieve}(\text{friday})$
  - 另一种描述：  
 $\text{high} = \{(0, 10), (1, 12), (2, 9), (3, 10), (4, 13), (5, 13), (6, 12)\}$   
日期名→数值→索引



# C++中的数组

---

- 索引（下标）形式：
  - $[i_1][i_2][i_3]\dots[i_k]$
  - $i_j$ ——非负整数
  - $k=1$ ——一维数组， $k=2$ ——二维数组，...



```

#include<iostream>
using namespace std;

int main()
{
    int score[2][3]={1, 2, 3, 4, 5, 6} :
    return 0;
}

```

## Memory 1

Address: 0x0015FA08

0x0015FA08	01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00	....
0x0015FA18	05 00 00 00 06 00 00 00 cc cc cc cc 9b 86 5a 79	....
0x0015FA28	78 fa 15 00 6f 19 aa 00 01 00 00 00 68 34 4d 00	x?..
0x0015FA38	08 1e 4d 00 c5 86 5a 79 00 00 00 00 00 00 00 00	..M.
0x0015FA48	00 e0 fd 7e 00 00 00 00 00 00 00 00 00 00 00 00	.??
0x0015FA58	00 00 16 00 00 00 00 00 3c fa 15 00 0a 00 00 00	....
0x0015FA68	bc fa 15 00 6e 10 aa 00 ad 10 e5 79 00 00 00 00	??..

Autos Locals Memory 1 Threads Modules Watch 1

# 多维数组的保存方式

- 计算机内存——一维连续存储
  - 多维数组如何与真实内存对应（映射）？
  - 多维数组元素  $\rightarrow \text{start} \sim \text{start} + \text{size}(\text{score}) - 1$
  - 实现映射  $[i_1][i_2][i_3] \dots [i_k] \rightarrow [0, n-1]$   
 $\text{map}(i_1, i_2, i_3, \dots, i_k)$
  - 存储位置  
 $\text{start} + \text{map}(i_1, i_2, i_3, \dots, i_k) * \text{sizeof}(\text{int})$
  - 一维数组:  $\text{map}(i_1) = i_1$



# 多维数组的保存方式

## — 二维数组就有些难度

- 第一维下标：行
- 第二维下标：列
- 到底按什么顺序来存储？

[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]



# 行主映射

---

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

- 行主映射：存储顺序为
  - 第一行、第二行、...
  - 每行内：位于第一列的那个元素、第二列、...



# 列主映射

---

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

- 列主映射：编号顺序为
  - 第一列、第二列、...
  - 每列内：位于第一行的那个元素、第二行



# 二维数组的映射函数

- 行主次序

- $\text{map}(i_1, i_2) = i_1 u_2 + i_2$

$u_2$ ——列的数目

- 如上例

$$u_2=6, \text{map}(i_1, i_2) = 6i_1 + i_2$$

$$\text{map}(1, 3) = 6 + 3 = 9$$

$$\text{map}(2, 5) = 6 * 2 + 5 = 17$$

```
int score[u1][u2]
```

[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]





# 扩展至三维数组

- $a[3][2][4]$

- $[0][0][0], [0][0][1], [0][0][2], [0][0][3],$   
 $[0][1][0], [0][1][1], [0][1][2], [0][1][3],$   
 $[1][0][0], [1][0][1], [1][0][2], [1][0][3],$   
 $[1][1][0], [1][1][1], [1][1][2], [1][1][3],$   
 $[2][0][0], [2][0][1], [2][0][2], [2][0][3],$   
 $[2][1][0], [2][1][1], [2][1][2], [2][1][3]$

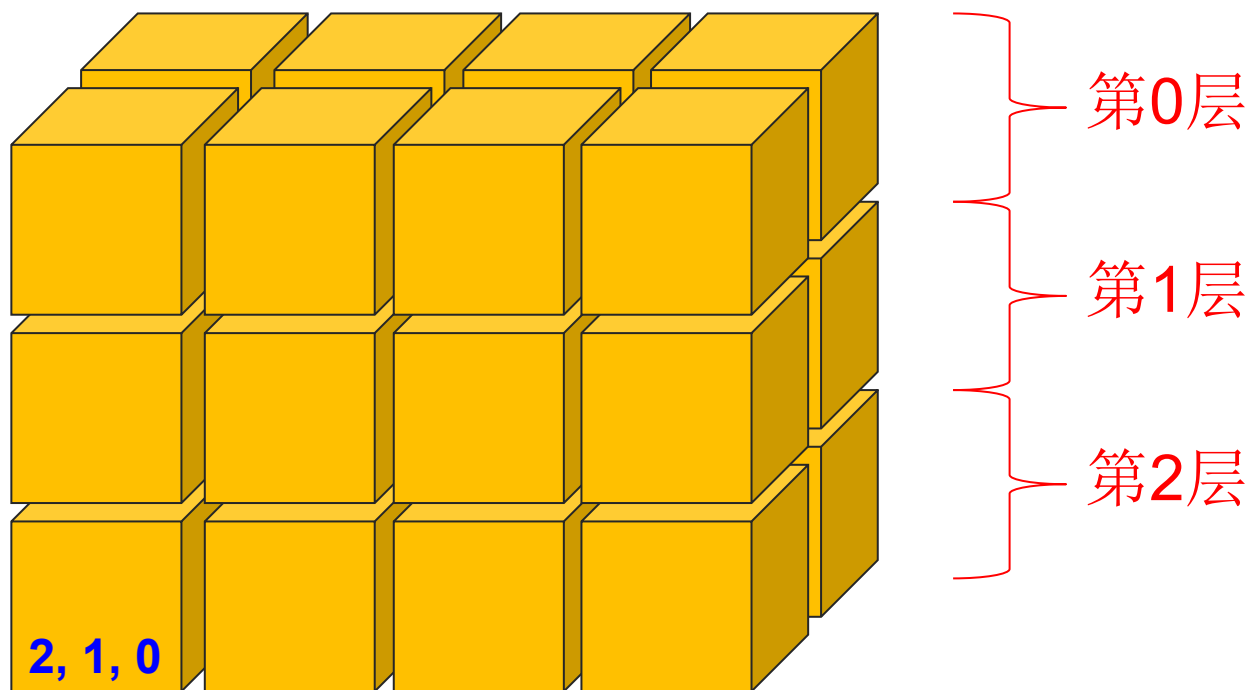
- 映射函数:  $\text{map}(i_1, i_2, i_3) = i_1 u_2 u_3 + i_2 u_3 + i_3$

- $\text{map}(2, 1, 0) = 2*2*4 + 1*4 + 0 = 20$  (可以想象魔方)

- 多维类似



# 三维数组示例



# 类Array1D

---

- C++数组的缺陷（**为什么不够用？**）
  - **越界问题**：int a[9]——a[-3]、a[9]、a[90]都允许出现在程序中，可能造成系统崩溃
  - **输出问题**：不支持cout << a << endl;
  - 不支持对数组进行**算术操作**（当作向量对待）
- 类Array1D——功能更强的数组
  - 封装动态一维数组
  - 通过**成员函数**（操作），实现一维数组功能，同时解决上述缺陷



# Array1D类定义

```
template<class T>
class Array1D {
    friend ostream& operator<<
        (ostream&, const Array1D<T>&);
public:
    Array1D(int size = 0);
    Array1D(const Array1D<T>& v); // copy constructor
    ~Array1D() {delete [] element;}
    T& operator[](int i) const;
    int Size() {return size;}
    Array1D<T>& operator=(const Array1D<T>& v);
    Array1D<T> operator+() const; // unary +
    Array1D<T> operator+(const Array1D<T>& v) const;
    Array1D<T> operator-() const; // unary minus
    Array1D<T> operator-(const Array1D<T>& v) const;
    Array1D<T> operator*(const Array1D<T>& v) const;
    Array1D<T>& operator+=(const T& x);
    Array1D<T>& ReSize(int sz);
private:
    int size;
    T *element; // 1D array
};
```

重载下标操作符  
解决越界问题

重载算术  
运算符实  
现数组整  
体运算



# 构造函数

```
template<class T>
```

```
Array1D<T>::Array1D(int sz)
```

```
{// Constructor for one-dimensional arrays.
```

```
    if (sz < 0) throw BadInitializers();
```

```
    size = sz;
```

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

```
}
```

构造一个空的  
维数组



# 拷贝构造函数

构造一个内容与v  
样的一维数组

```
template<class T>
```

```
Array1D<T>::Array1D(const Array1D<T>& v)
```

```
{// Copy constructor for one-dimensional  
  arrays.
```

```
  size = v.size;
```

```
  element = new T[size]; // get space
```

```
  for (int i = 0; i < size; i++) // copy elements
```

```
    element[i] = v.element[i];
```

```
}
```



# 重载 []

```
template<class T>
```

检查越界

```
T& Array1D<T>::operator[](int i) const
```

```
{// Return reference to element i.
```

```
    if (i < 0 || i >= size) throw OutOfBounds();
```

```
    return element[i];
```

```
}
```

- 可像普通数组一样使用Array1D

- $X[1] = 2 * Y[3]$

- $Y[3]$  即  $Y.operator[3]$ ，得到元素3的引用 (T&)

乘以2，赋予X[1]——元素1的引用



返回值

限定符

参数类型

# 重载=：数组内容的复制

```
template<class T>
```

```
Array1D<T>& Array1D<T>::operator=(const Array1D<T>& v)
```

```
{// Overload assignment operator.
```

```
if (this != &v) {// 不允许自身的复制
```

```
size = v.size;
```

```
delete [] element; // 释放原有空间
```

```
element = new T[size]; // 分配与v相同大小的内存空间
```

```
for (int i = 0; i < size; i++) // 复制数据
```

```
element[i] = v.element[i];
```

```
}
```

```
return *this;
```





# 重载二元加法运算符

---

```
template<class T>
```

```
Array1D<T> Array1D<T>::
```

```
    operator+(const Array1D<T>& v) const
```

```
{// Return w = (*this) + v.
```

```
    if (size != v.size) throw SizeMismatch();
```

```
    // create result array w
```

```
    Array1D<T> w(size);
```

```
    for (int i = 0; i < size; i++)
```

```
        w.element[i] = element[i] + v.element[i];
```

```
    return w;
```

```
}
```



# 重载二元减法运算符

---

```
template<class T>
```

```
Array1D<T> Array1D<T>::
```

```
    operator-(const Array1D<T>& v) const
```

```
{// Return w = (*this) - v.
```

```
    if (size != v.size) throw SizeMismatch();
```

```
    // create result array w
```

```
    Array1D<T> w(size);
```

```
    for (int i = 0; i < size; i++)
```

```
        w.element[i] = element[i] - v.element[i];
```

```
    return w;
```

```
}
```



# 重载二元乘法运算符

---

```
template<class T>
```

```
Array1D<T> Array1D<T>::
```

```
    operator*(const Array1D<T>& v) const
```

```
{// Return w = (*this) * v.
```

```
    if (size != v.size) throw SizeMismatch();
```

```
    // create result array w
```

```
    Array1D<T> w(size);
```

```
    for (int i = 0; i < size; i++)
```

```
        w.element[i] = element[i] * v.element[i];
```

```
    return w;
```



# 重载一元减法运算符

---

```
template<class T>
```

```
Array1D<T> Array1D<T>::operator-() const
```

```
{// Return w = -(*this).
```

```
    Array1D<T> w(size);
```

```
    for (int i = 0; i < size; i++)
```

```
        w.element[i] = - element[i];
```

```
    return w;
```

```
}
```



# 重载增量运算符+=

---

```
template<class T>
```

```
Array1D<T>&Array1D<T>
```

```
    ::operator+=(const T& x)
```

```
{ // Add x to each element of (*this).
```

```
    for (int i = 0; i < size; i++)
```

```
        element[i] += x;
```

```
    return *this;
```

```
}
```



# 复杂性简要分析

---

- 构造函数和析构函数
  - T为基本数据类型:  $\Theta(1)$ ——只分配内存
  - T为用户自定义类:  $O(\text{size})$ ——还要调用T的构造函数
- []:  $\Theta(1)$ ——直接返回
- 其他:  $O(\text{size})$ ——逐元素处理



# 类Array2D

---

- 递归：一维数组（行）的一维数组

```
template<class T>
```

```
class Array2D {
```

```
    friend ostream& operator<<
```

```
        (ostream&, const Array2D<T>&);
```

```
public:
```

```
    ...
```

```
    int Rows() const {return rows;}
```

```
    int Columns() const {return cols;}
```

```
    ...
```

```
private:
```

```
    int rows, cols; // array dimensions
```

```
    Array1D<T> *row; // array of 1D arrays
```

```
};
```



# 构造函数

请同学们思考：这个条件是什么意思？要如何测试？你能用其他形式表达同样的意思吗？

```
template<class T>
Array2D<T>::Array2D(int r, int c)
{ // Constructor for two-dimensional arrays.
  // validate r and c
  if (r < 0 || c < 0) throw BadInitializers();
  if ((!r || !c) && (r || c))
    throw BadInitializers();
```

未指定每行大小，  
每行的分配、释放由  
Array1D类负责

```
    rows = r;
    cols = c;
    // allocate r 1D arrays of default size
    row = new Array1D<T> [r];
```

改变数组大小：

```
    // make them right size
    for (int i = 0; i < r; i++)
        row[i].ReSize(c);
```

delete [] element;

size = sz;

element = new T [size];





# 拷贝构造函数

---

```
template<class T>
```

```
Array2D<T>::Array2D(const Array2D<T>& m)
```

```
{// Copy constructor for two-dimensional arrays.
```

```
    rows = m.rows;
```

```
    cols = m.cols;
```

```
    // allocate array of 1D arrays
```

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

```
    // copy each row
```

```
    for (int i = 0; i < rows; i++)
```

```
        row[i] = m.row[i];
```

调用Array1D重载的赋值运算符，完成一行（一个一维数组）的复制



# 重载 []

---

- `X[i][j]`——`(X.operator[i]).operator[j]`
  - 第一个 `[]`: `Array2D<T>::operator[]`  
返回一个 `Array1D<T>` 的引用  
得到第 `i` 行——一个一维数组
  - 第二个 `[]`: `Array1D<T>::operator[]`  
返回 `T&`——得到所需的元素的引用



# 重载[]的代码

---

```
template<class T>
```

```
Array1D<T>& Array2D<T>::operator[](int i)  
    const
```

```
{// First index of 2D array.
```

```
    if (i < 0 || i >= rows) throw OutOfBounds();
```

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

```
}
```



# 重载二元减法操作符

---

```
template<class T>
```

```
Array2D<T> Array2D<T>::
```

```
    operator-(const Array2D<T>& m) const
```

```
{// Return w = (*this) - m.
```

```
    if (rows != m.rows || cols != m.cols)
```

```
        throw SizeMismatch();
```

```
    Array2D<T> w(rows,cols);
```

```
    for (int i = 0; i < rows; i++)
```

```
        w.row[i] = row[i] - m.row[i];
```

```
    return w;
```

调用Array1D的－  
运算符



# 乘法操作——矩阵乘法

```
template<class T>
Array2D<T> Array2D<T>::
    operator*(const Array2D<T>& m) const
{ // A matrix product. Return w = (*this) * m.
    if (cols != m.rows) throw SizeMismatch();
    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;
}
```

三层循环嵌套



# 复杂性分析

---

- 构造函数和析构函数
  - T——基本类型:  $O(\text{rows})$
  - T——用户自定义类:  $O(\text{rows} * \text{cols})$
- 复制构造函数:  $O(\text{rows} * \text{cols})$
- `[]`:  $\Theta(1)$
- 乘法操作符:  $O(\text{rows} * \text{cols} * m.\text{cols})$



# 小结

---

- 为什么要自定义数据结构“数组”？
- “数组”的ADT应该是什么样子？
- “1维数组”如何定义？
- “2维数组”如何定义？



# 主要内容

---

- 数组
- 矩阵（与2DArray的异同）
- 特殊矩阵
  - 对角矩阵、三对角矩阵、三角矩阵、对称矩阵
- 稀疏矩阵
  - 数组描述
  - 链表描述





# 矩阵的定义

- $m \times n$  矩阵：  $m \times n$  表
- $m$ 、 $n$ ——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



# 矩阵示例

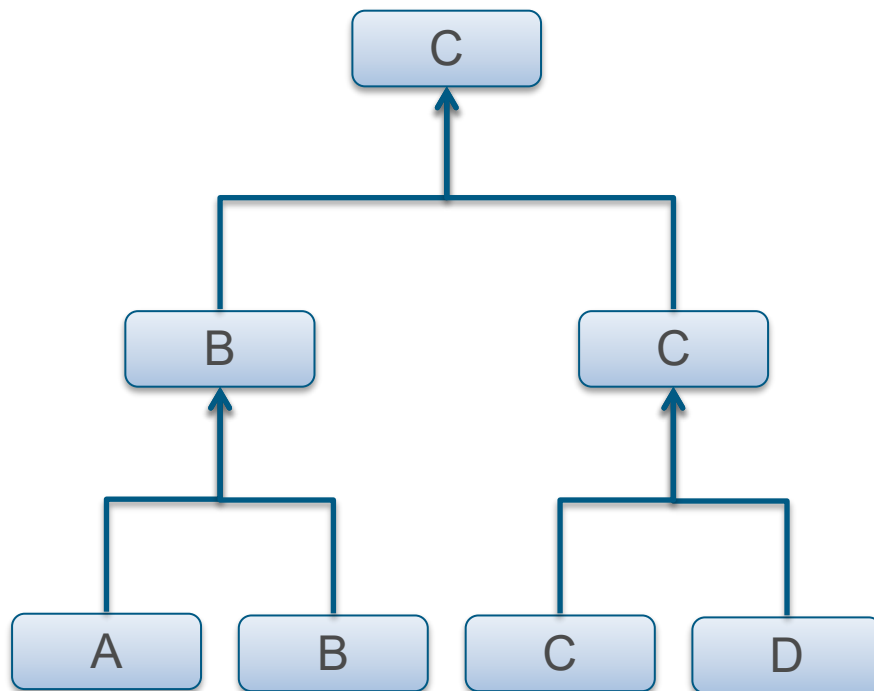
---

- 资源统计
- 资源类型：矿产（金、银等）、动物（狮子、大象等）、人（物理学家、工程师等）...
- 每种资源在每个国家的数量——二维表
  - 列——国家，行——资源  $\rightarrow n$  列、 $m$  行的资源矩阵
  - $M(i, j)$ ——矩阵 $M$ 第 $i$ 行、第 $j$ 列的元素
  - 第 $i$ 行——猫，第 $j$ 列——美国， $M(i, j)$ ——美国所拥有的猫的总数



# 矩阵示例

	A	B	C	D
A	----	1:2	0:0	4:1
B	2:1	----	1:1	0:3
C	0:0	1:1	----	2:2
D	1:4	3:0	2:2	----



# 矩阵的运算

---

- 矩阵转置

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

- 矩阵和—— $C = A + B$

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

- 矩阵乘法—— $C = A * B$

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



# 类Matrix

---

```
template<class T>
```

```
class Matrix {
```

```
    friend ostream& operator<<
```

```
        (ostream&, const Matrix<T>&);
```

```
public:
```

```
    Matrix(int r = 0, int c = 0);
```

```
    Matrix(const Matrix<T>& m); // copy constructor
```

```
    ~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; // unary +
```

区别一：

使用()操作符  
访问矩阵元素



# 类Matrix (续)

---

**Matrix<T> operator+(const Matrix<T>& m) const;**

**Matrix<T> operator-() const; // unary minus**

**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; // matrix dimensions**

**T \*element; // element array**

**};**

区别二:

利用一维数组  
模拟二维矩阵



# 构造函数

---

```
template<class T>
Matrix<T>::Matrix(int r, int c)
{ // Matrix constructor.
    // validate r and c
    if (r < 0 || c < 0) throw BadInitializers();
    if ((!r || !c) && (r || c))
        throw BadInitializers();

    // create the matrix
    rows = r; cols = c;
    element = new T [r * c];
}
```



# 下标操作符 ()

---

```
template<class T>
```

```
T& Matrix<T>::operator()(int i, int j) const
```

```
{// Return a reference to element (i,j).
```

```
    if (i < 1 || i > rows || j < 1 || j > cols)
```

```
        throw OutOfBounds();
```

```
    return element[(i - 1) * cols + j - 1];
```

```
}
```

显式使用映射函数  
二维下标 → 一维数组位置  
行列均从1开始编号





# 减法操作符

```
template<class T>
Matrix<T> Matrix<T>::
    operator-(const Matrix<T>& m) const
{ // Return (*this) - m.
    if (rows != m.rows || cols != m.cols)
        throw SizeMismatch();

    // create result matrix w
    Matrix<T> w(rows, cols);
    for (int i = 0; i < rows * cols; i++)
        w.element[i] = element[i] - m.element[i];
    return w;
}
```

- 对应元素相减——类似一维数组实现



# 乘法操作符

---

```
template<class T>
Matrix<T> Matrix<T>::
    operator*(const Matrix<T>& m) const
{ // Matrix multiply. Return w = (*this) * m.
    if (cols != m.rows) throw SizeMismatch();
    Matrix<T> w(rows, m.cols); // result matrix
    int ct = 0, cm = 0, cw = 0;
    // compute w(i,j) for all i and j
    for (int i = 1; i <= rows; i++) {
        // compute row i of result
        for (int j = 1; j <= m.cols; j++) {
            // compute first term of w(i,j)
            T sum = element[ct] * m.element[cm];
```



# 乘法操作符（续）

---

```
// add in remaining terms
for (int k = 2; k <= cols; k++) {
    ct++; // next term in row i of *this
    cm += m.cols; // next in column j of m
    sum += element[ct] * m.element[cm];
}
w.element[cw++] = sum; // save w(i,j)
// reset to start of row and next column
ct -= cols - 1;
cm = j;
}
// reset to start of next row and first column
ct += cols;
cm = 0;
}
return w;
```



# 复杂性分析

---

- 构造函数
  - T是基本数据类型:  $O(1)$
  - T是用户自定义类:  $O(\text{rows} * \text{cols})$
- 复制构造函数:  $O(\text{rows} * \text{cols})$
- 下标操作符:  $\Theta(1)$
- 乘法操作符:  $O(\text{rows} * \text{cols} * m. \text{cols})$



# 主要内容

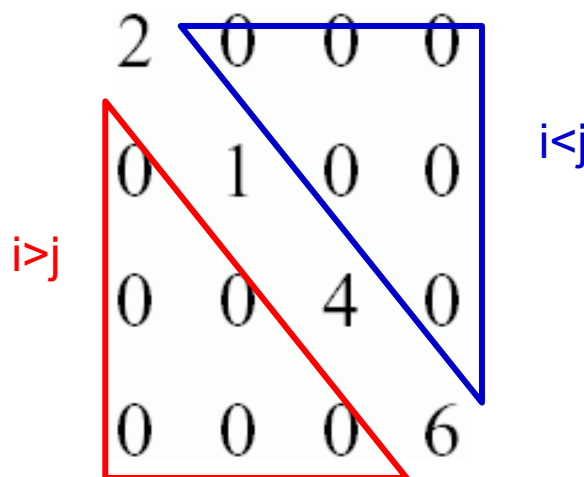
---

- 数组
- 矩阵
- 特殊矩阵
  - 对角矩阵、三对角矩阵、三角矩阵、对称矩阵
- 稀疏矩阵
  - 数组描述
  - 链表描述



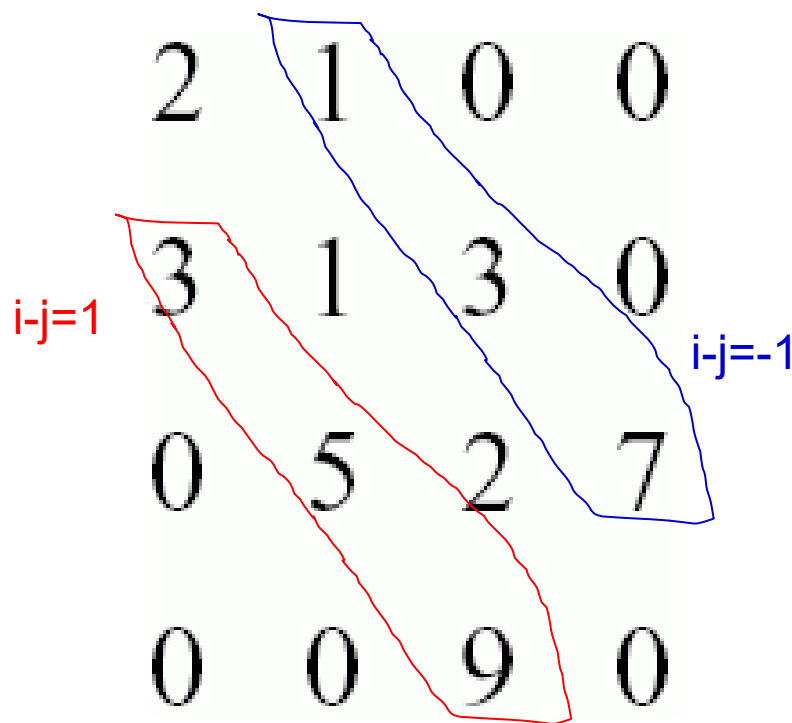
# 特殊矩阵

- 方阵：行数和列数相等的矩阵
- 对角矩阵M阵（diagonal）
  - 对所有  $i \neq j$  时,  $M(i, j) = 0$



# 三对角矩阵 (tridiagonal)

- 对所有  $|i - j| > 1$  时,  $M(i, j) = 0$



# 下三角矩阵 (lower triangular)

- 对所有  $i < j$  时,  $M(i, j) = 0$

2	0	0	0
5	1	0	0
0	3	1	0
4	2	7	0





# 上三角矩阵 (upper triangular)

- 对所有  $i > j$  时,  $M(i, j) = 0$

2	1	3	0
0	1	3	8
0	0	1	6
0	0	0	0



# 对称矩阵 (symmetric)

---

- 对所有的  $i$  和  $j$  ,  $M(i, j) = M(j, i)$

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



# 城市距离

- $\text{distance}(i, j)$ ——城市  $i$  和城市  $j$  的距离
- 显然,  $\text{distance}(i, j) = \text{distance}(j, i)$

	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



# 对角矩阵的高效存储

---

- 普通二维数组
  - $T \text{ } d[n][n]$ ; ——  $n \times n$  对角矩阵  $D$
  - $d[i-1][j-1]$  ——  $D(i, j)$
  - 空间需求:  $n^2 * \text{sizeof}(T)$
- $n$  个元素 —— 一维结构  $\rightarrow$  一维数组
  - $T \text{ } d[n]$ ;
  - $d[i-1] \leftrightarrow D(i, i)$
  - 其他 0 元素无需保存



# DiagonalMatrix类

---

```
template<class T>
class DiagonalMatrix {
public:
    DiagonalMatrix(int size = 10)
        {n = size; d = new T [n];}
    ~DiagonalMatrix() {delete [] d;} // destructor
    DiagonalMatrix<T>& Store(const T& x, int i, int j);
    T Retrieve(int i, int j) const;
private:
    int n; // matrix dimension
    T *d; // 1D array for diagonal elements
};
```



# 保存数据

---

```
template<class T>
```

```
DiagonalMatrix<T>& DiagonalMatrix<T>::
```

```
    Store(const T& x, int i, int j)
```

```
{// Store x as 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;
```

```
}
```

⊙ (1)



# 获取数据

---

```
template <class T>
```

```
T DiagonalMatrix<T>::Retrieve(int i, int j) const
```

```
{// Retrieve D(i,j).
```

```
    if (i < 1 || j < 1 || i > n || j > n)
```

```
        throw OutOfBounds();
```

```
    if (i == j) return d[i-1];
```

```
    else return 0;
```

```
}
```

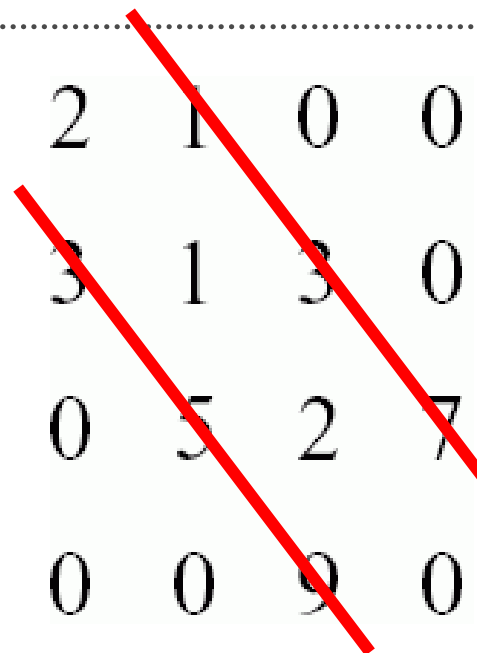
- $\Theta(1)$



# 三对角矩阵

- 非0元素位置

- 主对角线:  $i=j$
- 下对角线:  $i=j+1$
- 上对角线:  $i=j-1$



2	1	0	0
3	1	3	0
0	5	2	7
0	0	9	0





# 三对角矩阵的存储

- $3n-2$ 个元素的一维数组

- 逐行映射:  $t[0:9]=[2, 1, 3, 1, 3, 5, 2, 7, 9, 0]$
- 逐列映射:  $t=[2, 3, 1, 1, 5, 3, 2, 9, 7, 0]$
- 对角线映射（下对角线、主对角线、上对角线顺序）：  
 $t=[3, 5, 9, 2, 1, 2, 0, 1, 3, 7]$

2	1	0	0
3	1	3	0
0	5	2	7
0	0	9	0



# 映射函数： 对角线映射

请注意映射函数的推导过程，这是后续C++实现的前提和关键！

2	1	0	0
3	1	3	0
0	5	2	7
0	0	9	0



下对角线  $i - j = 1$       元素数:  $n - 1$      $M(i, j) = a[i-2]$

主对角线  $i = j$       元素数:  $n$        $M(i, j) = a[n-1+i-1]$

上对角线  $i - j = -1$       元素数:  $n - 1$      $M(i, j) = a[2n-1+i-1]$



# TridiagonalMatrix类

---

```
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; // matrix dimension
```

```
    T *t; // 1D array for tridiagonal
```



# 保存数据

```
template<class T>
TridiagonalMatrix<T>& TridiagonalMatrix<T>::
    Store(const T& x, int i, int j)
{ // Store x as T(i,j)
if ( i < 1 || j < 1 || i > n || j > n)
    throw OutOfBounds();

switch (i - j) { // 对角线映射方式
    case 1: // lower diagonal
        t[i - 2] = x; break;
    case 0: // main diagonal
        t[n + i - 2] = x; break;
    case -1: // upper diagonal
        t[2 * n + i - 2] = x; break;
    default: if(x != 0) throw MustBeZero();
}
return *this;
}
```



# 获取数据

---

```
template <class T>
T TridiagonalMatrix<T>::Retrieve(int i, int j) const
{ // Retrieve T(i,j)
if ( i < 1 || j < 1 || i > n || j > n)
    throw OutOfBounds();

switch (i - j) {
    case 1: // lower diagonal
        return t[i - 2];
    case 0: // main diagonal
        return t[n + i - 2];
    case -1: // upper diagonal
        return t[2 * n + i - 2];
    default: return 0;
}
```



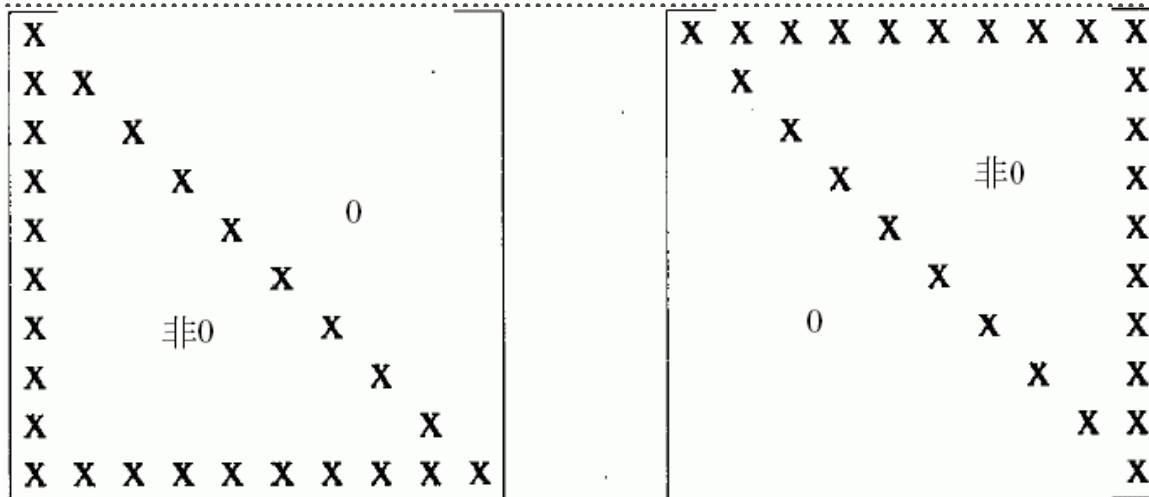
# 小思考

---

- 讲义中的三对角矩阵使用了对角线映射
  - 使用按行映射实现三对角矩阵的保存数据操作
  - 使用按列映射实现三对角矩阵的获取数据操作



# 三角矩阵



- 下三角：第1行1个元素，第2行2个...
- 上三角：第1行n个元素，第2行n-1个...
- 元素数目  $\sum_{i=1}^n i = n(n+1)/2$



# 描述方式

- 二维结构
- 用一维数组模拟保存
  - 行映射:  $I[0:9] = (2, 5, 1, 0, 3, 1, 4, 2, 7, 0)$
  - 列映射:  $I = (2, 5, 0, 4, 1, 3, 2, 1, 7, 0)$

2 0 0 0

5 1 0 0

0 3 1 0

4 2 7 0





# 映射公式

2	0	0	0
5	1	0	0
0	3	1	0
4	2	7	0

- 行映射

- $L(i, j)$  之前的元素数目

- 第  $i$  行之前的元素数目:  $1 + 2 + \dots + (i - 1)$
    - 第  $i$  行  $L(i, j)$  之前的元素数目:  $j - 1$

$$\left(\sum_{k=1}^{i-1} k\right) + (j - 1) = \frac{i(i-1)}{2} + j - 1$$



# 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; // matrix dimension
```

```
    T *t; // 1D array for lower triangle
```



# 保存数据

---

```
template<class T>
```

```
LowerMatrix<T>& LowerMatrix<T>::
```

```
    Store(const T& x, int i, int j)
```

```
{// Store x as L(i,j).
```

```
    if ( i < 1 || j < 1 || i > n || j > n)
```

```
        throw OutOfBounds();
```

```
// (i,j) in lower triangle if i >= j
```

```
    if (i >= j) t[i*(i-1)/2+j-1] = x;
```

```
    else if (x != 0) throw MustBeZero();
```

```
    return *this;
```



# 获取数据

---

```
template <class T>
```

```
T LowerMatrix<T>::Retrieve(int i, int j) const
```

```
{// Retrieve L(i,j).
```

```
    if ( i < 1 || j < 1 || i > n || j > n)
```

```
        throw OutOfBounds();
```

```
    // (i,j) in lower triangle if i >= j
```

```
    if (i >= j) return t[i*(i-1)/2+j-1];
```

```
    else return 0;
```

```
}
```



# 对称矩阵

---

- 上三角区域和下三角区域的内容相同
- 只保存其中一个区域即可
- 未保存的元素值——对称元素的值必然保存
- 使用三角矩阵相同方式处理



# 特殊矩阵小结

---

- 本质
  - 如何利用特殊矩阵的“特殊性”减少存储空间？
- 关键
  - 找到矩阵中元素与一维数组的映射关系！



# 主要内容

---

- 数组
- 矩阵
- 特殊矩阵
  - 对角矩阵、三对角矩阵、三角矩阵、对称矩阵
- 稀疏矩阵
  - 数组描述
  - 链表描述



# 稀疏（sparse）矩阵

---

- “许多”元素为0的矩阵

且 非0区域结构无规律

- 对应稠密矩阵
- 区分两者的界限？





# 例：顾客购物数据

---

- $\text{purchases}(i, j)$ ——顾客 $j$ 购买商品 $i$ 的数量
- 1000个顾客，10000种商品
- 顾客平均购买20种商品，0.2%非0元素！
- 价格矩阵 $\text{price}$ ：  $10000 \times 1$
- 顾客花费矩阵 $\text{spent} = \text{purchases}^T * \text{price}$
- 二维数组描述
  - $\text{purchases}$ 浪费空间
  - $\text{spent}$ 的计算性能差



# 数组描述：三元组

- 压缩存储，只保存非0元素

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[]	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



# 数组描述

---

- 非0元素用三元组表示——Term对象
- 一维数组保存所有非0元素——行主顺序

```
template <class T>
```

```
class Term {
```

```
private:
```

```
    int row, col;
```

```
    T value;
```

```
};
```



# 与简单二维数组的性能对比

---

- 空间复杂性

- 每个元素:  $2 * \text{sizeof}(\text{int}) + \text{sizeof}(T)$
- 240000: 40000000, 节省99.4%!

- 时间复杂性

- Store :  $O(\text{非0元素数目})$
- Retrieve :  $O(\log(\text{非0元素数目}))$
- 但是, 转置、加、乘...显著提高性能!



# SparseMatrix类

---

```
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;
```



# SparseMatrix类

---

**private:**

```
void Append(const Term<T>& t);  
int rows, cols; // matrix dimensions  
int terms; // current number of nonzero terms  
Term<T> *a; // term array  
int MaxTerms; // size of array a;  
};
```



# 构造函数

---

```
template<class T>
```

```
SparseMatrix<T>::SparseMatrix(int maxTerms)
```

```
{// Sparse matrix constructor.
```

```
    if (maxTerms < 1) throw BadInitializers();
```

```
    MaxTerms = maxTerms;
```

```
    a = new Term<T> [MaxTerms];
```

```
    terms = rows = cols = 0;
```

```
}
```



# 输出

---

```
template <class T>
ostream& operator<<(ostream& out,
                    const SparseMatrix<T>& x)
{ // Put *this in output stream.
  out << "rows = " << x.rows << " columns = "
    << x.cols << endl;
  out << "nonzero terms = " << x.terms << endl;

  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;
}
```

$\Theta(\text{terms})$





# 输入

---

```
template<class T>
istream& operator>>(istream& in, SparseMatrix<T>&
    x)
{ // Input a sparse matrix.
    cout << "Enter number of rows, columns, and terms"
        << endl;
    in >> x.rows >> x.cols >> x.terms;
    if (x.terms > x.MaxTerms) throw NoMem();

    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;
}
```

$\Theta(\text{terms})$



# 矩阵转置算法

- 转置矩阵：  $M'(i, j) = M(j, i)$ 
  - $M'$ 的第 $i$ 行—— $M$ 的第 $i$ 列
  - $M'$ 行主次序存储—— $M$ 的列主次序存储
  - 关键：  $M(i, j)$ 在列主次序中排在第几个位置？——目前 $M$ 是行主次序保存，无法直接获得此排位
  - 先求 $M$ 的第 $i$ 列（ $M'$ 的第 $i$ 行）从第几个位置开始？ ←  
遍历 $M$ 的元素，统计每列元素数目ColSize



# 矩阵转置算法（续）

- 于是， $M'$ 每行（第一个元素）的起始位置

$RowNext$

- $RowNext[0]=0$   
 $RowNext[1]=ColSize[0],$   
 $RowNext[2]=RowNext[1]+ColSize[1], \dots,$   
 $RowNext[i]=RowNext[i-1]+ColSize[i-1], \dots$

- 转置算法

- 扫描数组——行主顺序扫描矩阵元素 $M(i, j)$ 
  - 保存到 $M'$ 的 $RowNext[j]$ 处
  - $RowNext[j]++$



# 矩阵转置

---

```
template<class T>
void SparseMatrix<T>::
    Transpose(SparseMatrix<T> &b) const
{ // b保存转置结果.

    // make sure b has enough space
    if (terms > b.MaxTerms) throw NoMem();

    // set transpose characteristics
    b.cols = rows;
    b.rows = cols;
    b.terms = terms;
```



# 矩阵转置

原矩阵每列非0元素数目

**// initialize to compute transpose**

转置矩阵每行中，下一个非0元素在b中位置

**int \*ColSize, \*RowNext;**

**ColSize = new int[cols + 1];**

**RowNext = new int[cols + 1];**

**// find number of entries in each column of \*this**

**for (int i = 1; i <= cols; i++) // initialize**

**ColSize[i] = 0;**

**for (i = 0; i < terms; i++) // 计算每列元素数目**

**ColSize[a[i].col]++;**

原矩阵中的非零元素列表



# 矩阵转置

---

```
// 计算转置矩阵每行（原矩阵每列）第一个元素在b中位置
// 第i行起始位置：行1元素数+...+行i-1元素数
RowNext[1] = 0;
for (i = 2; i <= cols; i++)
    RowNext[i] = RowNext[i - 1] + ColSize[i - 1];
// perform the transpose copying from *this to b
for (i = 0; i < terms; i++) {
    int j = RowNext[a[i].col]++; // a[i]在b中位置
    b.a[j].row = a[i].col;
    b.a[j].col = a[i].row;
    b.a[j].value = a[i].value;
}
```



# 分析

```

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[]	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

i	1	2	3	4	5	6	7	8
colsize	0	2	1	2	1	1	1	1
rownext	0	2	3	5	6	7	8	9

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



# 上述转置算法实质上是...

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

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





# 复杂性为什么降低？

---

- 二维数组:  $O(\text{rows} * \text{cols})$
- SparseMatrix:  $O(\text{cols} + \text{terms})$
- terms远小于 $\text{rows} * \text{cols}$



# 在尾部添加新元素

---

```
template<class T>
void SparseMatrix<T>::Append(const Term<T>& t)
{ // Append a nonzero term t to *this.
    if (terms >= MaxTerms) throw NoMem();
    a[terms] = t;
    terms++;
}
```

- 调用者应保证满足行主顺序
- $\Theta(1)$



# 矩阵相加

a[]	0	1	2	3
row	1	2	2	4
col	4	2	4	1
value	2	3	-4	7

b[]	0	1	2
row	2	2	3
col	2	4	4
value	6	4	5

c[]	0	1	2	3
row	1	2	3	4
col	4	2	4	1
value	2	9	5	7

扫描两个矩阵元素  
比较行主次序位置

1、a的元素靠前：放入  
结果矩阵，继续扫描

2、同一位置：相加的和  
放入结果矩阵

3、相加为0不要放

4、b靠前：类似1

5、a或b全部处理完：将  
另一个剩余元素放入结  
果矩阵



# 矩阵相加

---

```
template<class T>
void SparseMatrix<T>::Add(const SparseMatrix<T> &b,
                          SparseMatrix<T> &c) const
{ // Compute c = (*this) + b.

    // verify compatibility
    if (rows != b.rows || cols != b.cols)
        throw SizeMismatch(); // incompatible matrices

    // set characteristics of result c
    c.rows = rows;
    c.cols = cols;
    c.terms = 0; // initial value

    // 两个指针，用于遍历*this和b，实现对应元素相加
    int ct = 0, cb = 0;
```



# 矩阵相加

---

```
// move through *this and b adding like terms
while (ct < terms && cb < b.terms) {
    // 计算两个元素的行主次序编号
    int indt = a[ct].row * cols + a[ct].col;
    int indb = b.a[cb].row * cols + b.a[cb].col;

    if (indt < indb) { // b的元素次序靠后, 显然, *this的
        当前
        c.Append(a[ct]); //元素即为结果——b的对应位置为0
        ct++; } // next term of *this
```



# 矩阵相加

---

**else {if (indt == indb) {** // 两个元素序号相同，应将它  
们相加

**// 注意：相加结果为0不应放入结果矩阵c!**

**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);}**



# 矩阵相加

---

```
        ct++; cb++;} // next terms of *this and b
    else {c.Append(b.a[cb]); // *this元素次序靠后的情况
        cb++;} // next term of b
    }
```

// 某个矩阵处理完毕，另一个未完，将剩余元素添加入c即可

```
for (; ct < terms; ct++)
    c.Append(a[ct]);
for (; cb < b.terms; cb++)
    c.Append(b.a[cb]);
}
```



# 分析

---

- while循环和两个for循环的循环总次数最多为 $\text{terms} + \text{b. terms}$   
→时间复杂度 $O(\text{terms} + \text{b. terms})$





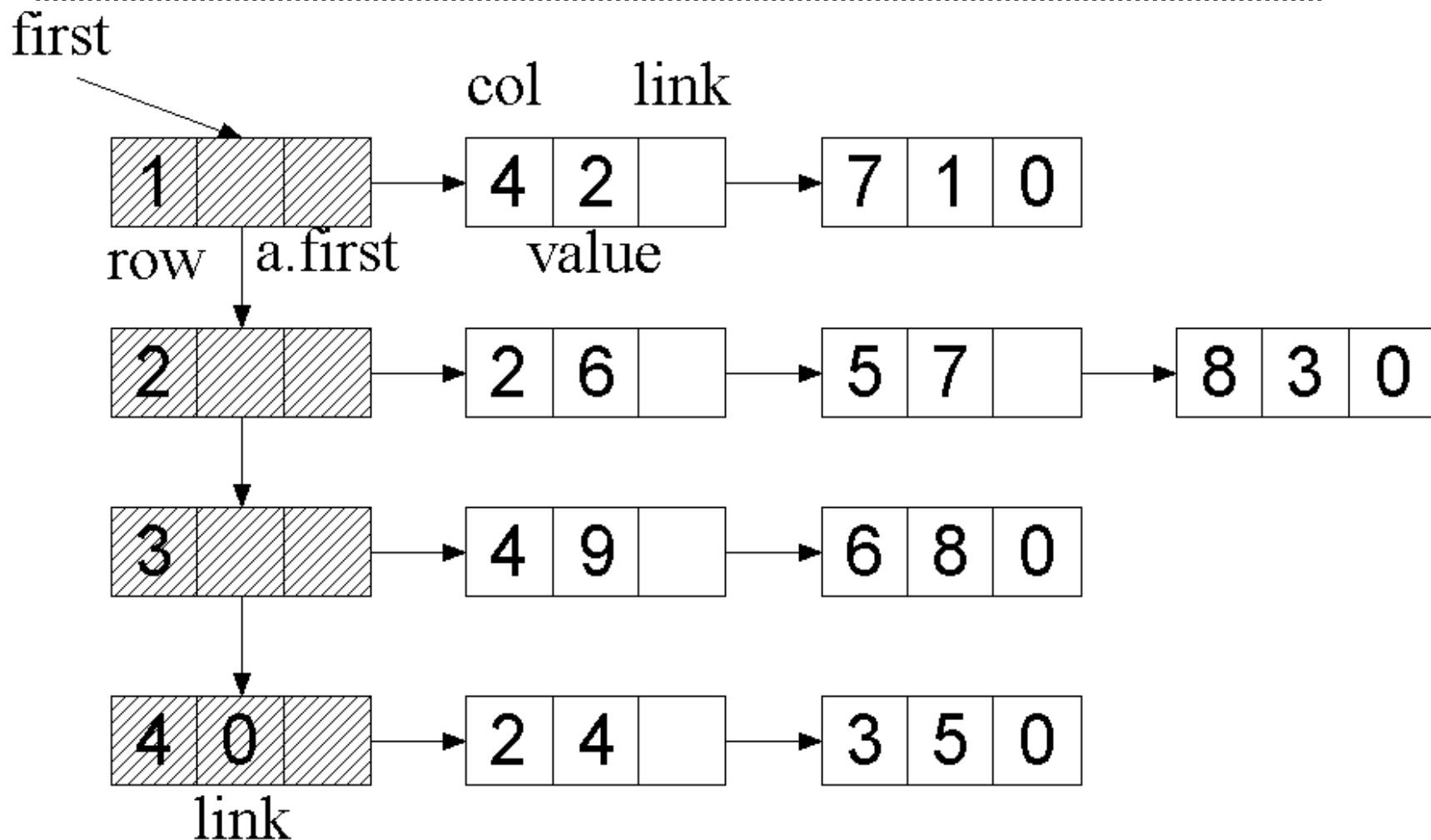
# 链表描述

---

- 数组描述的缺点
  - 创建矩阵时，必须知道非0元素总数
  - 加、减、乘操作→非0元素数目发生变化
  - 估计一个数目，可能浪费，可能不足
  - 不足→分配更大空间→数据复制，效率低！
- 链表描述（使用Chain、ChainIterator）
  - 指针占用额外空间，但很少
  - 空间占用与实际元素数匹配，无需数据复制



# 描述方法



# 节点类

```
template<class T>
class CNode {
    friend LinkedMatrix<T>;
    friend ostream& operator<<
        (ostream&, const LinkedMatrix<T>&);
    friend istream& operator>>
        (istream&, LinkedMatrix<T>&);
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; }
```



# 行头节点类

---

```
template<class T>
class HeadNode {
    friend LinkedMatrix<T>;
    friend ostream& operator<<
        (ostream&, const LinkedMatrix<T>&);
    friend istream& operator>>
        (istream&, LinkedMatrix<T>&);
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; // row chain
};
template<class T>
ostream& operator<<(ostream& out, const HeadNode<T>& x)
    {x.Output(out); out << endl; return out;}
```



# LinkedMatrix类

---

```
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;
private:
    int rows, cols;      // matrix dimensions
    Chain<HeadNode<T> > a; // head node chain
};
```



# 本节课我们学习了：

---

- 数组
  - 1维数组、2维数组
- 矩阵
  - 一般矩阵
  - 对角矩阵、三对角矩阵
  - 上三角矩阵、下三角矩阵
  - 对称矩阵
  - 稀疏矩阵



---

# 本章结束

