Arrays and Matrices
Chapter - 7

2D Arrays

The elements of a 2-dimensional array a declared as:

```
int [][]a = new int[3][4];
```

may be shown as a table

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

Rows Of A 2D Array

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

Columns Of A 2D Array

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

2D Array Representation In C++

2-dimensional array x

a, b, c, d

e, f, g, h

i, j, k, l

view 2D array as a 1D array of rows

x = [row0, row1, row 2]

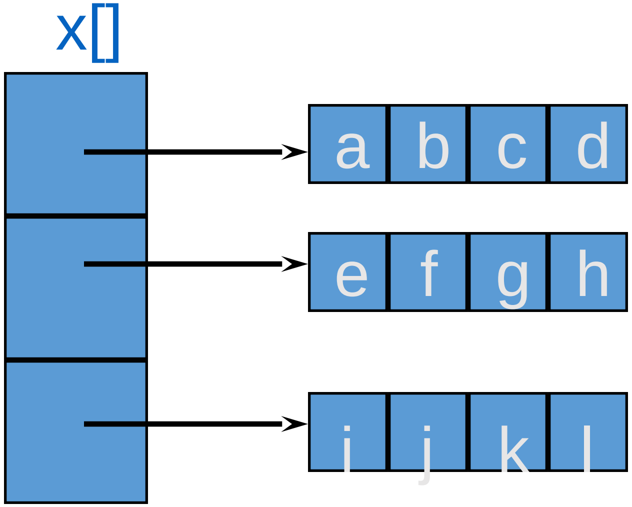
row 0 = [a,b, c, d]

row 1 = [e, f, g, h]

row 2 = [i, j, k, l]

and store as 4 1D arrays

2D Array Representation In C++



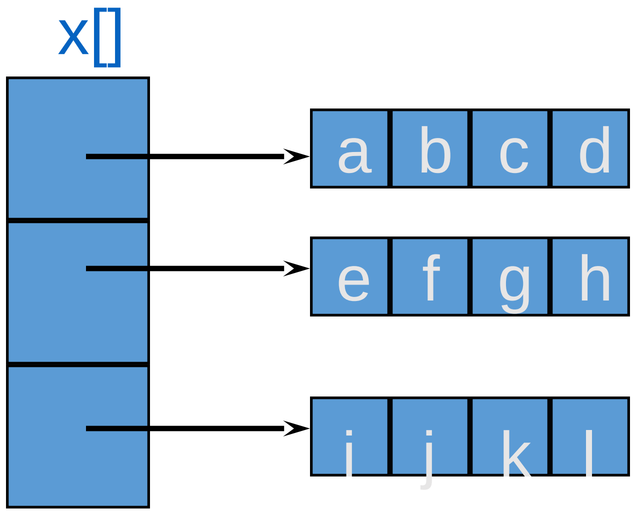
space overhead = overhead for 4 1D arrays

= 4 * 4 bytes

= 16 bytes

= (number of rows + 1) x 4 bytes

Array Representation In C++



- This representation is called the array-of-arrays representation.
- Requires contiguous memory of size 3, 4, 4, and 4 for the 4 1D arrays.
- 1 memory block of size number of rows and number of rows blocks of size number of columns

Row-Major Mapping

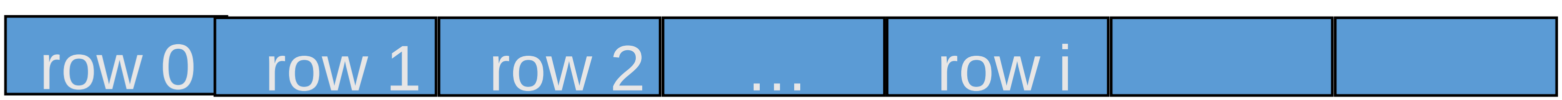
- In general, for 3 dimension,

  $map(i_1,i_2,i_3) = i_1u_2u_3+i_2u_3+i_3$
- Example 3 x 4 array:

  a b c d

  e f g h

  i j k l
- Convert into 1D array y by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.
- We get y {a, b, c, d, e, f, g, h, i, j, k, l}

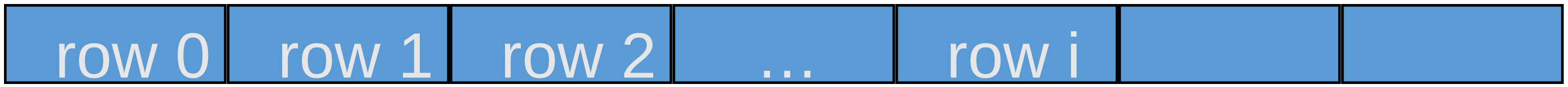


Locating Element x[i][j]

- assume x has r rows and c columns
- each row has c elements
- i rows to the left of row i
- so ic elements to the left of x[i][0]
- so x[i][j] is mapped to position

  ic + j of the 1D array

Space Overhead



4 bytes for start of 1D array +

4 bytes for c (number of columns)

= 8 bytes

Disadvantage: Need contiguous memory of size rc

Column-Major Mapping

a b c d

e f g h

i j k l

- Convert into 1D array y by collecting elements by columns.
- Within a column elements are collected from top to bottom.
- Columns are collected from left to right.
- We get {a, e, i, b, f, j, c, g, k, d, h, l}

Exercise

- Suppose, we want to map the elements of a two dimensional array beginning with bottom row and within a row from left to right order.

a) List the indexed of score[3][5] in this order

b) Develop a mapping function for the score[u_1 u_2]

(a)

The ordering of the indexes is:

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

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

(b)

The elements with first index i_1 are preceded by $(u_1 - i_1 - 1)u_2$ elements that have a larger first index. Since the elements with the same first index are stored from right to left,

$map(i_1, i_2) = (u_1 - i_1 - 1)u_2 + i_2$

Matrix

A mxn matrix is a table with m rows and n columns. m and n are dimensions of the matrix.

a b c d	row 1
e f g h	row 2
i j k l	row 3

- Use notation x(i,j) rather than x[i][j].
- May use a 2D array to represent a matrix.
- Operations performed: Addition, Multiplication, Transpose

Header cla
Matrix with
operator
overloadin

```
template<class T>
class matrix
{
   friend ostream& operator<<(ostream&, const matrix<T>&);
   public:
      matrix(int theRows = 0, int theColumns = 0);
      matrix(const matrix<T>&);
      ~matrix() {delete [] element;}
      int rows() const {return theRows;}
      int columns() const {return theColumns;}
      T& operator()(int i, int j) const;
      matrix<T>& operator=(const matrix<T>&);
      matrix<T> operator+() const; // unary +
      matrix<T> operator+(const matrix<T>&) const;
      matrix<T> operator-() const; // unary minus
      matrix<T> operator-(const matrix<T>&) const;
      matrix<T> operator*(const matrix<T>&) const;
      matrix<T>& operator+=(const T&);
   private:
       int theRows,    // number of rows in matrix
           theColumns; // number of columns in matrix
       T *element;     // element array
};
```

Constructor and copy constructor

```
template<class T>
matrix<T>::matrix(int theRows, int theColumns)
{// matrix constructor.
   // validate theRows and theColumns
   if (theRows < 0 || theColumns < 0)
      throw illegalParameterValue("Rows and columns must be >= 0");
   if ((theRows == 0 || theColumns == 0)
               && (theRows != 0 || theColumns != 0))
      throw illegalParameterValue
      ("Either both or neither rows and columns should be zero");

   // create the matrix
   this->theRows = theRows;
   this->theColumns = theColumns;
   element = new T [theRows * theColumns];
}

template<class T>
matrix<T>::matrix(const matrix<T>& m)
{// Copy constructor for matrices.
   // create matrix
   theRows = m.theRows;
   theColumns = m.theColumns;
   element = new T [theRows * theColumns];

   // copy each element of m
   copy(m.element,
        m.element + theRows * theColumns,
        element);
}
```

Overload = operator

```
template<class T>
matrix<T>& matrix<T>::operator=(const matrix<T>& m)
{// Assignment. (*this) = m.
    if (this != &m)
    {// not copying to self
        delete [] element;
        theRows = m.theRows;
        theColumns = m.theColumns;
        element = new T [theRows * theColumns];
        // copy each element
        copy(m.element,
             m.element + theRows * theColumns,
             element);
    }
    return *this;
}
```

Overload ( ) operator

```
template<class T>
T& matrix<T>::operator()(int i, int j) const
{// Return a reference to element (i,j).
   if (i < 1 || i > theRows
       || j < 1 || j > theColumns)
   throw matrixIndexOutOfBounds();
   return element[(i - 1) * theColumns + j - 1];
}
```

Matrix Addition

```
template<class T>
matrix<T> matrix<T>::operator+(const matrix<T>& m) const
{// Return w = (*this) + m.
   if (theRows != m.theRows
       || theColumns != m.theColumns)
      throw matrixSizeMismatch();

   // create result matrix w
   matrix<T> w(theRows, theColumns);
   for (int i = 0; i < theRows * theColumns; i++)
      w.element[i] = element[i] + m.element[i];

   return w;
}
```

Matrix multiplicatio

```
template<class T>
matrix<T> matrix<T>::operator*(const matrix<T>& m) const
{// matrix multiply.  Return w = (*this) * m.
   if (theColumns != m.theRows)
      throw matrixSizeMismatch();

   matrix<T> w(theRows, m.theColumns);  // result matrix

   // define cursors for *this, m, and w
   // and initialize to location of (1,1) element
   int ct = 0, cm = 0, cw = 0;

   // compute w(i,j) for all i and j
   for (int i = 1; i <= theRows; i++)
   {// compute row i of result
      for (int j = 1; j <= m.theColumns; j++)
      { // compute first term of w(i,j)
         T sum =  element[ct] * m.element[cm];

         // add in remaining terms
         for (int k = 2; k <= theColumns; k++)
         {
            ct++;  // next term in row i of *this
            cm += m.theColumns;  // 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 -= theColumns - 1;
         cm = j;
      }

      // reset to start of next row and first column
      ct += theColumns;
      cm = 0;
   }

   return w;
}
```

Special Matrices

- Tridiagonal:

Matrix M is tridiagonal iff M(i,j) = 0 for |i-j| > 1

- Upper Traingular:

Matrix M is upper triangular iff M(i,j) = 0 for I > j

- Symmetric:

Matrix M is symmetric iff M(i,j) = M(j,i) for all i and j

Diagonal Matrix

```
template<class T>
class diagonalMatrix
{
   public:
      diagonalMatrix(int theN = 10);
      ~diagonalMatrix() {delete [] element;}
      T get(int, int) const;
      void set(int, int, const T&);
   private:
      int n;          // matrix dimension
      T *element;     // 1D array for diagonal elements
};

template<class T>
diagonalMatrix<T>::diagonalMatrix(int theN)
{// Constructor.
   // validate theN
   if (theN < 1)
       throw illegalParameterValue("Matrix size must be > 0");

   n = theN;
   element = new T [n];
}
```

Get Method for Diagonal Matrix

```
template <class T>
T diagonalMatrix<T>::get(int i, int j) const
{// Return (i,j)th element of matrix.
   // validate i and j
   if (i < 1 || j < 1 || i > n || j > n)
       throw matrixIndexOutOfBounds();

   if (i == j)
      return element[i-1];   // diagonal element
   else
      return 0;              // nondiagonal element
}
```

Set Method for Diagonal Matrix

```
template<class T>
void diagonalMatrix<T>::set(int i, int j, const T& newValue)
{// Store newValue as (i,j)th element.
   // validate i and j
   if (i < 1 || j < 1 || i > n || j > n)
        throw matrixIndexOutOfBounds();

   if (i == j)
     // save the diagonal value
      element[i-1] = newValue;
   else
      // nondiagonal value, newValue must be zero
      if (newValue != 0)
         throw illegalParameterValue
               ("nondiagonal elements must be zero");
}
```

Diagonal Matrix

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{matrix}$$

- An n x n matrix in which all nonzero terms are on the diagonal.
- x(i,j) is on diagonal iff i = j
- number of diagonal elements in an n x n matrix is n
- store diagonal only vs n^2 whole

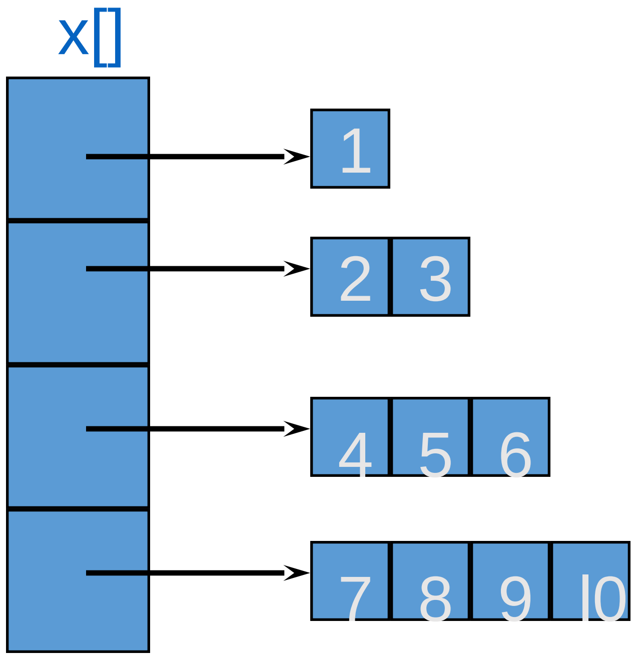
Lower Triangular Matrix

An n x n matrix in which all nonzero terms are either on or below the diagonal.

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{matrix}$$

- x(i,j) is part of lower triangle iff i >= j.
- number of elements in lower triangle is 1 + 2 + … + n = n(n+1)/2.
- store only the lower triangle

Array Of Arrays Representation



Use an irregular 2-D array … length of rows is not required to be the same.

Creating And Using An Irregular Array

```
// declare a two-dimensional array variable
// and allocate the desired number of rows
int ** irregularArray =        int* [numberOfRows];

// now allocate space for the elements in each row
for (int i = 0; i < numberOfRows; i++)
    irregularArray[i] = new int [length[i]];

// use the array like any regular array
irregularArray[2][3] = 5;
irregularArray[4][6] = irregularArray[2][3] + 2;
```

Map Lower Triangular Array Into A 1D Array

Use row-major order, but omit terms that are not part of the lower triangle.

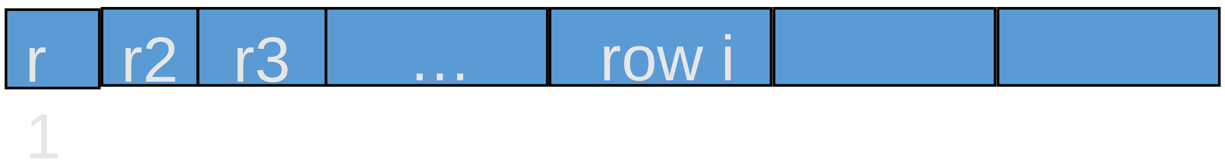
For the matrix

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{matrix}$$

we get

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Index Of Element [i][j]



- Order is: row 1, row 2, row 3, …
- Row i is preceded by rows 1, 2, …, i-1
- Size of row i is i.
- Number of elements that precede row i is

  1 + 2 + 3 + … + i-1 = i(i-1)/2
- So element (i,j) is at position i(i-1)/2

Sparse Matrices

Matrix table of values

0 0 3 0 4

Row 2

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

Column 4

4 x 5 matrix

4 rows

5 columns

20 elements

6 nonzero elements

Sparse Matrices

Sparse matrix #nonzero elements/#elements is small.

Examples:

- Diagonal
  - Only elements along diagonal may be nonzero
  - n x n matrix ratio is $n/n^2 = 1/n$
- Tridiagonal
  - Only elements on 3 central diagonals may be nonzero
  - Ratio is $(3n-2)/n^2 = 3/n – 2/n^2$

Sparse Matrices

- Lower triangular
- Only elements on or below diagonal may be nonzero
  - Ratio is $n(n+1)(2n^2) \sim 0.5$

These are structured sparse matrices. Nonzero elements are in a well-defined portion of the matrix.

Sparse Matrices

An n x n matrix may be stored as an n x n array.

This takes $O(n^2)$ space.

The example structured sparse matrices may be mapped into a 1D array so that a mapping function can be used to locate an element quickly; the space required by the 1D array is less than that required by an n x n array (next lecture).

Representation Of Unstructured Sparse Matrices

Single linear list in row-major order.

- scan the nonzero elements of the sparse matrix in row-major order (i.e., scan the rows left to right beginning with row 1 and picking up the nonzero elements)
- each nonzero element is represented by a triple

  (row, column, value)

- the list of triples is stored in a 1D array

Single Linear List Example

$$\begin{matrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{matrix}$$

list =

$$\begin{array}{l} \text{row} \\ \text{column} \\ \text{value} \end{array} \begin{bmatrix} 1 & 1 & 2 & 2 & 4 & 4 \\ 3 & 5 & 3 & 4 & 2 & 3 \\ 3 & 4 & 5 & 7 & 2 & 6 \end{bmatrix}$$

Single Linear List

- Class SparseMatrix
  - Array of triples of type MatrixTerm

    row, col, value

    rows, // number of rows

    cols, // number of columns

    terms, // number of nonzero elements

    capacity; // size of
- Size of generally not predictable at time of initialization.
  - Start with some default capacity/size (say 10)
  - Increase capacity as needed

Approximate Memory Requirements

500 x 500 matrix with 1994 nonzero elements, 4 bytes per element

2D array	500 x 500 x 4 = 1million bytes
Class SparseMatrix	3 x 1994 x 4 + 4 x 4
	= 23,944 bytes

Array Resizing

```
   (newSize < terms)          "Error";
MatrixTerm *temp =        MatrixTerm[newSize];
copy(smArray, smArray+terms, temp);
          [] smArray;
smArray = temp;
capacity = newSize;
```

Array Resizing

- To avoid spending too much overall time resizing arrays, we generally set newSize = c * oldSize, where c >0 is some constant.
- Quite often, we use c = 2 (array doubling) or c = 1.5.
- Now, we can show that the total time spent in resizing is O(s), where s is the maximum number of elements added to smArray.

Matrix Transpose

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

0 0 0 0

0 0 0 2

3 5 0 6

0 7 0 0

4 0 0 0

Matrix Transpose

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0

0	0	0	0
0	0	0	2
3	5	0	6
0	7	0	0
4	0	0	0

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

2	3	3	3	4	5
4	1	2	4	2	1
2	3	5	6	7	4

Matrix Transpose

$$\begin{matrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{matrix} \qquad \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 3 & 5 & 0 & 6 \\ 0 & 7 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{matrix}$$

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

Step 1: #nonzero in each row of transpose.

= #nonzero in each column of original matrix

= [0, 1, 3, 1, 1]

Step2: Start of each row of transpose

= sum of size of preceding rows of transpose

= [0, 0, 1, 4, 5]

Step 3: Move elements, left to right, from original list to transpose list.

Matrix Transpose

Step 1: #nonzero in each row of transpose.

= #nonzero in each column of original matrix

= [0, 1, 3, 1, 1]

Step2: Start of each row of transpose

= sum of size of preceding rows of transpose

= [0, 0, 1, 4, 5]

Step 3: Move elements, left to right, from

Complexity

m x n original matrix

t nonzero elements

Step 1: O(n+t)

Step 2: O(n)

Step 3: O(t)

Overall O(n+t)