Chapter 4
Arrays and Matrices

In practices, data is often available in tabular form. Although arrays are the most natural way to represent such a form, it is not always efficient, particularly, when a large chunk of their elements are zeros.

C++ does support an array representation, but it lacks several crucial features, such as subscript validation, and fails to support some important operations, such as addition and assignment.

Also, arrays supported by C++ begins its index from 0, while matrices starts at 1.
Arrays

Each instance of the data object array is a set of pairs in the form of \((\text{index}, \text{value})\). No two pairs in this set have the same value. It also comes with the following set of functions: Create() creates an array; Store(index, value) adds this pair to set, while deleting the existing pair with index as its index. Retrieve(index) returns the value with index as its index.

For example, given the following array: high= \{(sunday, 82), (monday, 79), \ldots, (saturday, 91)\}. We can change the temperature for Monday to 83, by performing Store(monday, 83), and find out the temperature of Saturday by performing Retrieve(Saturday).
Indexing a C++ array

The array structure of C++ comes with certain restriction. Its index must be of the form of $[i_1][i_2] \cdots [i_k]$, where every $i_j$ must be a non-negative integer. Such an array is called a $k$–dimensional, and $i_j$ is called the $j^{th}$ coordinate.

A $k$–dimensional array, score, can be declared as follows:

```cpp
int score[u1][u2]...[uk];
```

With such a declaration, indexes with $i_j$ in the range of $0 \leq i_j < u_j$, $1 \leq j \leq k$ is permitted. Hence, score will have $n = u_1u_2 \cdots u_k$ values, and occupies a space of $n \times \text{sizeof(int)}$. 
Row and major mappings

When implemented, any multi-dimensional array is mapped to a one dimensional array.

Assume \( a \) is a \( k \)-dimensional array of integers, then we have to find out a mapping between its index \([i_1][i_2] \cdots [i_k]\) to a value in the range of \([0, n - 1]\) so that, when \( a \) is stored from start on, that index will be mapped to a location in the range \([\text{start}, \text{start} + n \times \text{sizeof}(\text{int}) - 1]\). More specifically, the value with index \([i_1] \cdots [i_k]\) will be stored in \( \text{sizeof}(\text{int}) \) bytes beginning with \( \text{start} + \text{map}(i_1, i_2, \cdots, i_k) \times \text{sizeof}(\text{int}) \).

When \( k = 1 \), the mapping is simply

\[
\text{map}(i_1) = i_1.
\]

So a value with index \( i_1 \) is mapped to \( \text{start} + i_1 \times \text{sizeof}(\text{int}) \).
When \( k = 2 \), there are at least two ways to specify the mapping: *row major*, or *column major*. Given the following array:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

In *row major*, the order will be [0][0], [0][1], [0][2], [0][3], …, [2][2], [2][3]; while in the *column* major, the order will be [0][0], [1][0], [2][0], …, [1][3], [2][3].

Generally speaking, the index \((i_1, i_2)\) will be mapped to \(i_1 \times u_2 + i_2\), under row major, and to \(i_2 \times u_1 + i_1\), under column major. For example, for the above 3 \(\times\) 4 array, if we use the row major, then index \([1, 2]\) will be mapped to 6. It will be mapped to 7, under the column major.
The Array1D class

The C++’s support of one-dimensional array is rather weak. For example, with the C++ array `int a[9]`, we are allowed to access `a[90]`. This often causes some unpredictable problems. Also, in C++, we can not output the whole array, or assign values to it, and we can’t perform add and subtract on arrays, either.

Thus the need for the following Array1D class, in which the elements are stored in `X.element`, with the $i^{th}$ element is stored in `X.element[i]`, $0 \leq i < \text{size}$. 
template<class T>

class Array1D {

friend ostream& operator<<
    (ostream&, const Array1D<T>&);

public:
    Array1D(int size = 0);
    Array1D(const Array1D<T>& v);
    ~Array1D() {delete [] element;}
    T& operator[](int i) const;
    int Size() {return size;}
    Array1D<T>& operator=(const Array1D<T>& v);
    Array1D<T> operator+(const Array1D<T>& v) const;
    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
};
The codes for the constructor, copy constructor, and validated subscript, are given as follows:

```
Array1D<T>::Array1D(int sz){
    if (sz < 0) throw BadInitializers();
    size = sz;
    element = new T[sz];
}
```

```
Array1D<T>::Array1D(const Array1D<T>& v){
    size = v.size;
    element = new T[size];
    for (int i = 0; i < size; i++)
        element[i] = v.element[i];
}
```

```
T& Array1D<T>::operator[](int i) const{
    if (i < 0 || i >= size) throw OutOfBounds();
    return element[i];
}
```
Finally, we look at the code for the array assignment.

```cpp
Array1D<T>& Array1D<T>::operator=(const Array1D<T>& v){
    if (this != &v) {// not self-assignment
        size = v.size;
        delete [] element;
        element = new T[size];
        for (int i = 0; i < size; i++)
            element[i] = v.element[i];
    }
    return *this;
}
```

When T is a standard type, the complexity of constructor is $\Theta(1)$, as `new` is called only once, otherwise, it will be $O(size)$. It is not $\Theta(size)$ as an exception may be thrown.
The Array2D class

We can similarly declare a class for 2-dimensional array.

class Array2D {
    friend ostream& operator<<(ostream&, const Array2D<T>&);

public:
    Array2D(int r = 0, int c = 0);
    ...
    Array2D<T> operator*(const Array2D<T>& m) const;

private:
    int rows, cols;  // array dimensions
    Array1D<T>* row;  // array of 1D arrays
};
Let’s look at the code for the constructor:

```cpp
Array2D<T>::Array2D(int r, int c){
    if (r < 0 || c < 0) throw BadInitializers();
    if (((!r || !c) && (r || c))
        throw BadInitializers();
    rows = r;
    cols = c;
    row = new Array1D<T> [r];
    for (int i = 0; i < r; i++)
        row[i].ReSize(c);
}
```

After validating the dimensions, we construct an array of rows first, then adjust the size of each row to \( c \), by using the following code:

```cpp
delete [] element;
size=sz;
element=new T[size];
```
Given two arrays, if cols of the first equals to the rows of the second, we can multiply them together.

Array2D<T> Array2D<T>::
    operator*(const Array2D<T>& m) const{
        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;
    }
Homework

4.1. Write a pair of template functions Make3DArray and Delete3DArray to create and delete 3-d arrays, respectively. The user could declare such an array with int ***x, and get access to an element via x[i][j][k].

4.2. List the indices of score[2][3][2][2] in row-major order.

4.3. Obtain the row-major mapping function for a 4-d, 5-d, and k-d arrays.
Matrices

An \( m \times n \) matrix is a table with \( m \) rows and \( n \) columns, where \( m \) and \( n \) are called the dimensions of the matrix.

\[
\begin{array}{cccc}
\text{row 1} & \text{col 1} & \text{col 2} & \text{col 3} \\
7 & 2 & 0 & 9 \\
0 & 1 & 0 & 5 \\
6 & 4 & 2 & 0 \\
\end{array}
\]

Matrixes are often used to organize data. For instance, we can use it to produce a list of assets owned by counties. If there are \( m \) kinds of assets and \( n \) counties, we would have an \( m \times n \) asset matrix.

Among the most commonly applied operations are transpose, addition, and multiplication.
Let $M(m, n)$ be a matrix with $m$ rows and $n$ columns. The transpose of $M$ is defined as an $n \times m$ matrix such that

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

The sum of two $m \times n$ matrixes is also an $m \times n$ matrix $C$, such that

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

The product of an $m \times r$ matrix and a $r \times n$ matrix is an $m \times n$ matrix, $C$, such that

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

If two independent agencies make up two asset matrixes with no duplicates, then the complete asset matrix will be the sum of those two. Suppose there is another $m \times s$ value matrix, which gives, for $(i, j)$, the unit value of asset $i$ under scenario $j$, then the total value of county $i$ under scenario $j$ will be given by $asset^T \times value$. 

15
The *Matrix* class

It is easy to see that an $m \times n$ matrix can be easily implemented by using a 2-D array. But, we have to use array indexes, which is one smaller than the corresponding matrix indexes. Also, we have to use [] instead of (). Thus, for better readability, we declare a matrix class:

```cpp
class Matrix {
  friend ostream& operator<<(ostream&, const Matrix<T>&);
  
  public:
    Matrix(int r = 0, int c = 0);
    ... 
    T& operator[](int i, int j) const;
    ... 
    Matrix<T> operator-(const Matrix<T>& m) const;
    ...
  
  private:
    int rows, cols; // matrix dimensions
    T *element; // element array
};
```
Operations

Below gives definitions of constructor and matrix subscript or. Notice the modified mapping formula.

```cpp
Matrix<T>::Matrix(int r, int c){
    if (r < 0 || c < 0) throw BadInitializers();
    if ((!r || !c) && (r || c))
        throw BadInitializers();
    rows = r; cols = c;
    element = new T [r * c];
}
```

```cpp
T& Matrix<T>::operator()(int i, int j) const{
    if (i < 1 || i > rows || j < 1 || j > cols) throw OutOfBounds();
    return element[(i - 1) * cols + j - 1];
}
```

**Homework 4.4.** Write a function for Matrix that sends back a transposed matrix.
Special matrices

A square matrix has the same number of rows and columns. Some special square matrices are: diagonal, in which $M(i, j) = 0$, for all $i \neq j$, Tridiagonal, in which $M(i, j) = 0$ for $|i - j| > 1$, Lower triangular, in which $M(i, j) = 0$, for all $i < j$, Upper triangular, in which $M(i, j) = 0$, for all $i > j$, and
Symmetric, in which $M(i, j) = M(j, i)$, for all $i$ and $j$.

\[
\begin{array}{ccc}
2 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{array}
\quad
\begin{array}{ccc}
2 & 1 & 0 \\
3 & 1 & 0 \\
5 & 0 & 0 \\
\end{array}
\quad
\begin{array}{ccc}
2 & 0 & 0 \\
5 & 1 & 0 \\
0 & 3 & 0 \\
\end{array}
\]

(a) Diagonal (b) Tridiagonal (c) Lower triangular

\[
\begin{array}{ccc}
2 & 1 & 3 \\
0 & 1 & 3 \\
0 & 0 & 1 \\
\end{array}
\quad
\begin{array}{ccc}
2 & 4 & 6 \\
4 & 1 & 9 \\
6 & 9 & 4 \\
\end{array}
\quad
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{array}
\]

(d) Upper triangular (e) Symmetric
The stack folding problem

Assume that we have a stack of $n$ blocks with block 1 at the bottom and block $n$ at the top, each of which has the height being $h_i$. We are permitted to create sub stacks by finding a point $i$, and creating two stacks, one contains block 1 through block $i$, the other contains block $i + 1$ through $n$. If we repeat this process, there could be many stacks. The question is what will be the maximum height of those stacks?

Since, the height must be the sum of the heights of some successive blocks, we can organize them into a matrix, $H$, such that for $i \leq j$, $H(i, j) = \sum_{k=i}^{j} h_k$, and $H(i, j) = 0$, otherwise. Obviously, $H$ is an upper-triangular matrix.
Represent special matrices

Although an diagonal matrix can be represented by a 2-d array, it is not efficient, since it has only $n$ non-zero elements. A better way is to use $T \mathbf{d}[n]$ to represent it, and use $\mathbf{d}[i-1]$ to represent $D(i, i)$.

In an tridiagonal matrix, all the non-zero elements, $3n - 2$ in total, are located in on the three diagonals: 1) main diagonal, i.e., $i = j$; 2) diagonal below the main one, i.e., $i = j + 1$; and 3) the diagonal above the main one, i.e., $i = j - 1$. One of the mapping methods is to use a 1-D array, such that 1) $(i, i - 1), i \in [2, n]$, is mapped to the location of $i - 2$; 2) for $(i, i), i \in [1, n]$, is mapped to $n - 1 + (i - 1)$; and 3) $(i, i + 1), i \in [1, n - 1]$, is mapped to $2n - 1 + (i - 1)$.
For a lower-triangular matrix, there are $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ elements.

It can be easily mapped into a 1-D array, in which $(i, j), i \geq j$ is mapped to $\frac{i(i-1)}{2} + j - 1$.

Finally, for a symmetric matrix, we can represent only half of it, by using the formula developed for lower-triangular matrix.

**Homework 4.5.** Read through § 4.3, and write a function that maps a tridiagonal $n \times n$ matrix into a 1-d list of size $3n - 2$. 
Sparse matrices

An $m \times n$ matrix is “sparse” if it contains many zeros, otherwise, it is “dense”. Both diagonal and tridiagonal matrices are sparse, since, out of $n^2$ elements, only $\Theta(n)$ of them are non-zeros. It is easy to map them in to a 1-D arrays since the positions for those non-zero elements are well structured. It is not always the case in general. For example, although there are thousands of students, and hundreds of courses, but each student will take only a few courses, and each course can allow 20-30 students, so the registration matrix is a very sparse one, which also does not have a structure.

We will try to find out some efficient ways to represent sparse matrices.
Array representation

One way to represent the non-zero elements in a sparse matrix is to map them into a 1-D array in row-major order, together with a small 2-D array that keeps their locations.

For this purpose, we define the following class:

class Term {
    private:
        int row, col;
        T value;
}
Besides storing the above information, we also have to store the number of rows, columns, as well as the number of non-zero values. Hence, for the above case, the total number of bytes we have to use is \(9\times \text{sizeof}(T)+21\times \text{sizeof}(\text{int})\). Suppose \(T\) is \(\text{int}\), and it takes two bytes to store an integer, we have to spend 60 bytes, while the original \(4 \times 8\) matrix uses just 64 bytes. So, it is not a big saving in this case. But, in other cases, it can save quite a lot.

For example, in a supermarket, if there are 10,000 items, and up to 1,000 customers, then a purchase matrix would require 20 million bytes. If on average a customer buys 20 items, there will be about 20,000 non-zero items. Following our approach, it will take just a 120,006 bytes. This will lead to a huge saving.
Linked representation

A shortcoming for the array representation is that we have to know the number of non-zero elements before we can represent it. This information is available for arrays being input, but quite difficult to get otherwise. An alternative strategy is to use linked representation. It needs extra space to store the pointers, but it is a much more flexible approach.