

Practical 11

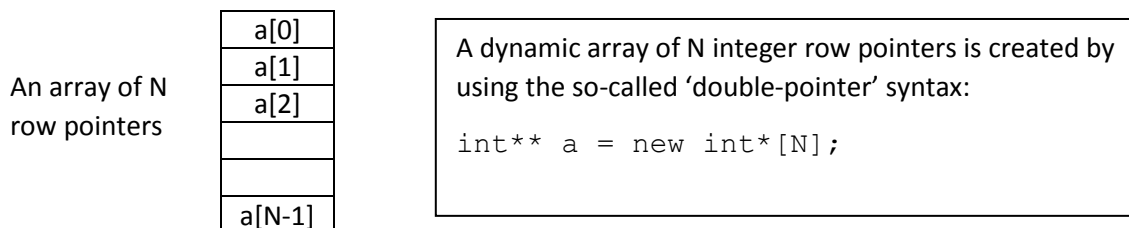
Dynamic Allocation of 2-D Arrays

Wednesday 14th February 2018

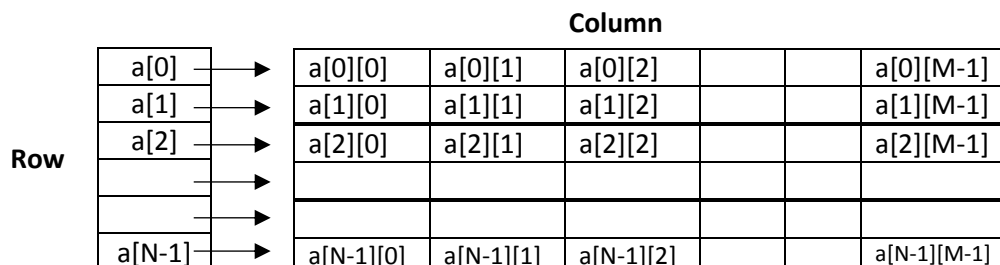
Later in our practical, we will practice the representation of graphs using adjacent matrices. This requires us to dynamically allocate 2-D arrays using C++. The following exercise is focused on this topic. The principles learned here can be extended to creating 3-D, 4-D or even higher dimensional arrays.

Suppose we allocate a 2-D integer array of size $N \times M$ (N rows and M columns). We do this by first allocating a dynamic array of N integer row pointers, and then initialising each row pointer to its own dynamic array of M column integers. The advantage of this approach is that you can address the array elements by using the conventional array indexing syntax you are used to. Graphically, this can be shown as follows:

First, create a dynamic array of N integer **row pointers**, in which each $a[i]$ is a **pointer** of integer type.



Then initialise each row pointer $a[i]$ to its own dynamic array of M column integers.



Initialise each row pointer to its own dynamic array of M column integers by using a loop:

```
for(i = 0; i < N; i++)
    a[i] = new int[M];
```

Now, we can access the array elements by using the conventional indexing syntax. For example,

Accessing the 2-D array elements:

```
a[2][4] = 57;
int b = a[3][8];
```

You should delete the array after use to avoid memory leak. To delete, undo the allocation in reverse order.

Delete the array:

```
for(i = 0; i < N; i++) delete [] a[i];
delete [] a;
```

Program 1

Following the above example, write a simple C++ program to practice the creation, access, and deletion of a dynamic 2-D double array of size $N \times M$, where N and M are arbitrary positive integers.

Program 2

The following exercise will give you an opportunity to revise some of the previously learned C++ techniques, e.g., classes, template classes, and operator overloading. It will also give you an opportunity to practice the handling of 2-D dynamic arrays, and to develop “MATLAB-style” instructions to perform matrix operations, i.e., an operation between whole matrices can be performed by using a single line instruction as if the arrays were scalar variables.

(1) Write a template class `matrix` for creating 2-D matrices and performing matrix operations. The class may take a structure:

```
template <typename T>
class matrix {
public:
    // constructor & destructor
    matrix(int n_rows, int n_cols);
    ~matrix();

    // copy constructor
    matrix(const matrix& o);

    // overload [] to return pointer to the i'th row of data
    T* operator[](int i);

    // overload + and * for matrices addition & multiplication
    matrix operator+(const matrix& a);
    matrix operator*(const matrix& a);

private:
    // matrix dimensions & data
    int num_rows, num_cols;
    T** data;
};
```

(2) Use the following program to test your code:

```
int main()
{
    // dimensions of a matrix
    int nRows = 3, nCols = 3;

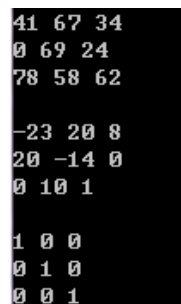
    // create an interger matrix of nRows x nCols
    matrix<int> a(nRows, nCols);
    // set the matrix with random values
    for (int i = 0; i < nCols; i++)
        for (int j = 0; j < nCols; j++)
            a[i][j] = rand() % 100;
    // print the matrix
    for (int i = 0; i < nCols; i++) {
        for (int j = 0; j < nCols; j++) cout << a[i][j] << " ";
        cout << endl;
    }
    cout << endl;

    // create two nRows x nCols double matrices
    matrix<double> x(nRows, nCols);
    x[0][0] = 1; x[0][1] = 2; x[0][2] = 3;
    x[1][0] = 0; x[1][1] = 1; x[1][2] = 4;
    x[2][0] = 5; x[2][1] = 6; x[2][2] = 0;
    matrix<double> y(nRows, nCols);
    y[0][0] = -24; y[0][1] = 18; y[0][2] = 5;
    y[1][0] = 20; y[1][1] = -15; y[1][2] = -4;
    y[2][0] = -5; y[2][1] = 4; y[2][2] = 1;

    // matrix addition: u = x + y
    matrix<double> u = x + y;
    for (int i = 0; i < nRows; i++) {
        for (int j = 0; j < nCols; j++) cout << u[i][j] << " ";
        cout << endl;
    }
    cout << endl;

    // matrix multiplication: v = x * y
    matrix<double> v = x * y;
    for (int i = 0; i < nRows; i++) {
        for (int j = 0; j < nCols; j++) cout << v[i][j] << " ";
        cout << endl;
    }
    cout << endl;
    return 0;
}
```

(3) Your code should generate the results as below (note the first section are random numbers)



```
41 67 34
0 69 24
78 58 62

-23 20 8
20 -14 0
0 10 1

1 0 0
0 1 0
0 0 1
```

Some notes for the development:

1. The class has a parameterised constructor which takes the numbers of rows and columns to the private data member `num_rows`, `num_cols`, and creates a dynamic empty matrix of the specified dimensions using the 2-D pointer `data`.
2. The class has a destructor which is invoked automatically to delete the matrix `data` when an object is destroyed or out of scope.
3. The class must have a copy constructor to facilitate the matrix operations through overloaded operators (points 5, 6 below). Do test what would happen without a copy constructor!
4. By overloading the operator `[]`, one should be able to access the private elements of the matrix, i.e. `data[i][j]`, through the object as if it were the matrix, e.g.,

```
matrix<int> m(5, 4); // declare a 5x4 int matrix object m
m[0][2] = 4;        // access data[0][2] by overloading []
```

However, C++ does not support overloading double-square `[]`. This problem can be overcome by just overloading operator `[]` and making it return a pointer to the specific row of `data`. Since pointers support subscripting by `[]` (the pointer-array equivalence of C++, see Lecture 2), we take the returned row pointer and access the specific column using subscripting by `[]`. Thus, access by the notation `[]` is achieved! See the examples in the testing `main` function, enclosed above.

5. This class overloads operator `+` to perform the addition of two matrices (which must have the same size) and return their sum, which is also a matrix. As such, one can add two matrices to produce a sum matrix by using a single line instruction: `u = x + y`; as if `u`, `x` and `y` are scalar variables; see the example in the `main` function. But actually, matrices addition involves element by element addition for all the elements. This should be implemented in the `operator+` function, in which each specific element in the sum matrix equals the sum of the corresponding elements of the two matrices being added, i.e.,

$$u[i][j] = data[i][j] + a.data[i][j]$$

where `data` belongs to the object on the left of `+` which generates the call to the operator function, and `a.data` belongs to the object on the right of `+` (or the argument of the function).

6. This class also overloads operator `*` to perform the multiplication of two matrices and return their product, which is also a matrix. As such, one can multiply two matrices to produce a new matrix by using a single line instruction: `v = x * y`; as if `v`, `x` and `y` are scalar variables; see the example in the `main` function. According to mathematics, two matrices can be multiplied if their sizes satisfy certain conditions (two equal-sized square matrices can always be multiplied). Each element in the product matrix is calculated using the formula:

$$v[i][j] = \sum_{k=0}^{num_cols-1} data[i][k] * a.data[k][j]$$

In our example, matrix `x` is said to be the *inverse* of matrix `y` (and vice versa), because `x * y` results in a unit (or identity) matrix:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

i.e., with ones on the main diagonal and zeros elsewhere. This is in analogy to saying that $\frac{1}{4}$ is the inverse of 4 (and vice versa) because $4 * \frac{1}{4} = 1$.