

Electronic copies of handouts for L02 can be found at

<http://www.enel.ucalgary.ca/People/Norman/engg233-L02-fall2004/>

A quick review of using a vector of vectors as a two-dimensional array. You are expected to understand how the following program works.

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

typedef vector<double> Row;
typedef vector<Row> Matrix;

int main()
{
    // Create a Matrix with 3 rows and 5 columns.
    Matrix m(3);
    for (int row = 0; row < 3; row++)
        m.at(row).resize(5);

    // Put new values in Matrix elements.
    for (int row = 0; row < m.size(); row++)
        for (int col = 0; col < m.at(0).size(); col++)
            m.at(row).at(col) = row + 0.01 * col;

    // Display matrix one row at a time.
    cout << showpoint << fixed << setprecision(2);
    for (int row = 0; row < m.size(); row++) {
        for (int col = 0; col < m.at(0).size(); col++)
            cout << setw(10) << m.at(row).at(col);
        cout << endl;
    }

    return 0;
}
```

The output is:

```
0.00      0.01      0.02      0.03      0.04
1.00      1.01      1.02      1.03      1.04
2.00      2.01      2.02      2.03      2.04
```

The “vector-of-vectors” approach to handling two-dimensional arrays is reasonably convenient, and *you are expected to use it in the final exam*. However, certain aspects of using a vector of vectors are annoying. It would be nicer if we could re-write main as something like this:

```
int main()
{
    // Create a Matrix with 3 rows and 5 columns.
    Matrix m(3, 5);

    // Put new values in Matrix elements.
    for (int row = 0; row < m.nrow(); row++)
        for (int col = 0; col < m.ncol(); col++)
            m.at(row, col) = row + 0.01 * col;

    // Display matrix one row at a time.
    cout << showpoint << fixed << setprecision(2);
    for (int row = 0; row < m.nrow(); row++) {
        for (int col = 0; col < m.ncol(); col++)
            cout << setw(10) << m.at(row, col);
        cout << endl;
    }

    return 0;
}
```

The main function at the bottom of page 1 can be made to work, but to do so `Matrix` can no longer simply be a typedef for vector of vector of doubles. Instead `Matrix` has to be a *class type*.

To set up `Matrix` as a class type, we need to write a *class definition* and a collection of *member function definitions*. Here is a class definition:

```
class Matrix {
public:
    // Constructors.
    Matrix(); // Make an empty matrix.

    Matrix(int requested_nrow,
           int requested_ncol); // Make matrix with requested dimensions.

    // Mutator functions.
    void resize(int requested_nrow,
                int requested_ncol); // Change dimensions.

    double& at(int row_index, int col_index); // Element access.

    // Accessor functions.
    int nrow() const; // How many rows?
    int ncol() const; // How many columns?
    const double& at(int row_index, int col_index) const; // Element access.

private:
    std::vector<std::vector<double>> storage_M;
    int nrow_M;
    int ncol_M;
};
```

And here are the member function definitions:

```
Matrix::Matrix()
: storage_M(0), nrow_M(0), ncol_M(0)
{ }

Matrix::Matrix(int requested_nrow, int requested_ncol)
: storage_M(0), nrow_M(0), ncol_M(0)
{
    // We've started with an empty matrix. Give it the requested
    // dimensions, but only if the requested dimensions make sense.
    if (requested_nrow > 0 && requested_ncol > 0) {
        nrow_M = requested_nrow;
        ncol_M = requested_ncol;
        storage_M.resize(nrow_M);
        for (int r = 0; r < nrow_M; r++)
            storage_M.at(r).resize(ncol_M);
    }
}

void Matrix::resize(int requested_nrow, int requested_ncol)
{
    if (requested_nrow > 0 && requested_ncol > 0) {
        nrow_M = requested_nrow;
        ncol_M = requested_ncol;
        storage_M.resize(nrow_M);
        for (int r = 0; r < nrow_M; r++)
            storage_M.at(r).resize(ncol_M);
    }
    else {
        nrow_M = 0;
        ncol_M = 0;
        storage_M.resize(0);
    }
}

double& Matrix::at(int row_index, int col_index)
{
    return storage_M.at(row_index).at(col_index);
}

int Matrix::nrow() const { return nrow_M; }

int Matrix::ncol() const { return ncol_M; }

const double& Matrix::at(int row_index, int col_index) const
{
    return storage_M.at(row_index).at(col_index);
}
```