**CPSC 1071   
Lab 11 - References and Operator Overloading**

**Objectives**

In today's lab, we will be implementing parts of a *Matrix* class in c++ using several new techiques: references and operator overloading. We can make use of the many new features that C++ uses to implement a matrix class. We will also learn how to use operator overloading to more naturally express operations on matrices, and practice using reference parameters.

**Task 1: understand and implement the *Matrix* class**

Review the following Matrix class specification and the implementation details.

class Matrix {

public:

Matrix(int, int);

Matrix(const Matrix&);

~Matrix();

Matrix& operator=(const Matrix&);

// Matrix Matrix operations

Matrix operator+(const Matrix&);

Matrix operator\*(const Matrix&);

Matrix operator-(const Matrix&);

void print();

double\* operator[](int);

const double\* operator[](int) const;

int getRows() const;

int getCols() const;

Matrix transpose() const;

private:

int rows;

int cols;

double\*\* mat;

};

In the above specification of a Matrix class, we have added several special methods that are prefixed with the word *operator* followed by an operation can be performed my a matrix and another type.

**Operator Overloading**

Operator overloading enables custom operations to be defined for a given object. For example, in a C version of a Matrix implemention, we would have to define the matrix add operation by creating a function such as,

void matrix\_add(double m1[][3], double m2[][3], double result[][3])

which takes two 2-dimensional arrays of type double as the input parameters, and returns the result in the third parameter.

In C++, we could have an add method with the following prototype:

Matrix add(Matrix &rhs);

Given:

Matrix a;

Matrix b;

Matrix c;

We would then add two matrices using the following:

c = a.add(b);

which is somewhat awkward.

By using operator overloading in our C++ version, a slight improvement to this would be to use the same notation to add two matrices together as we would to add two integers, floats, or doubles:

Matrix a;

Matrix b;

Matrix c = a + b;

Operator overloading in fact gives us the power to do this as well as providing us with many more operations to perform as you will implement in today's lab.

|  |
| --- |
| **a note on operators** |
| In case you're unaware, symbols such as '+', '-', '\', etc are commonly referred to as binary (infix) operators. Infix means the name sits in the middle of two (left and right hand side) arguments. |

Matrix operator+(const Matrix&);

The syntax for this can admittedly be kind of confusing if you're seeing it for the first time, so we'll walk through it here. As we're used to, the return type, *Matrix*, is the first thing we declare, so think of this as the type that results from applying the *+* operator. Next, we write *operator* followed immediately by the symbol we want to overload, in this case: *+* (note that C++ internally has a fixed list of acceptable operator symbols, so you can't just use anything as an operator, like *j* -- C++ must approve of it).

You'll note also that we've declared only a single parameter, *const Matrix&*, which can be kind of confusing since we already stated that *+* should have two parameters -- one corresponding to the left-hand-side of the operator, and one for the right-hand-side. In this case however, the *Matrix*object itself serves as an *implicit* left hand side argument, this is why we declare only one formal parameter to the function.

**Using the new operator with arrays**

When we want to create a pointer array using the using malloc we would normally type something like:

int\* x = (int\*)malloc(sizeof(int) \* 10);

Now, with C++ we can replace this with the new operator:

int\* x = new int[10];

Note the syntax if we want to create an array of 10 elements for the integer array.

If we wanted to create a 2-D array in C++, we can extend this further by first allocating the number of rows that we want and the number of columns that we want.

int\*\* x = new int\*[rows];

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

x[i] = new int[cols];

}

This will be the type of initialization we want for our *Matrix* constructor.

**Deleting our *Matrix***

Since we used the *new* operator to create our matrix, we will need to use the *delete* operator to delete our matrix. Similar to our constructor we first begin by deleting each of the columns row by row.

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

delete[] x[i];

}

delete[] x;

Note the syntax of *delete*. Since we are deleting an array of items we use *delete[]* rather than just *delete*. This code will be used in the destructor.

**Task 2: Getting the files for today's lab**

Create a **"lab11"** directory and copy [lab11.tar](https://www.cs.clemson.edu/course/cpsc1070/labs/lab11/Public/lab11.tar) to the directory using:

lab1071copy 11

Next untar the file, i.e.:

**tar -xvf lab11.tar**

You should now have the following files in your directory:

main.cpp Makefile matrix.cpp matrix.h

**Task 3: Implement the Matrix class**

Use your favorite text editor to open matrix.cpp, and write the required code.

**Task 4: Testing**

To compile and test your code type this:

make

./lab11

**Handin**

To hand in your project use the **tar cvf** command to turn in all of your **.cpp** and **.h** files and name the file **lab11\_handin.tar**.

Use [Handin](https://handin.cs.clemson.edu/courses/) website to turn in your lab11.