**Lab 18 - Class Templates (Matrices) - Revisited**

### **Goals**

By the end of this lab you should:

* Have a better understanding of forward declaration
* Understand which functions are candidates for being inline functions
* Understand how to overload the function operator
* Understand how to get around the problem of not being able to directly overload the [][] operator.

**Collaboration policy:**

For this assignment, collaboration **IS ALLOWED**.

## **Matrix Class Template Assignment Specifications:**

You will be implementing a templated Matrix class. Note that smaller member functions have been inlined.

The class Matrix maintains a collection of elements, in much the same fashion that the string class manages an array of character values. If ROWS and COLUMNS denote the dimensions of the matrix, then the matrix holds ROWS \* COLUMNS elements. Because the number of rows and columns is variable, we cannot use a two-dimensional array to store the matrix elements. Instead, we use a one-dimensional array. Figure 1 shows the organization. The matrix element from row i and column j is found in the position

i \* COLUMNS + j

The process of subscripting is fundamental to matrices. The C++ language does not directly support a two-dimensional subscript operator. There are two ways this is normally handled, and the starter code illustrates both.

One way is to use the function call operator for subscripting.

class Matrix

{

...

double& operator()(int i, int j); // Return element [i][j]

};

The second solution is slightly more complicated. A subscript will return an intermediate value, an instance of class MatrixRow. This class will retain a pointer to the original matrix, and the row subscript. In a subscript expression, such as

Matrix m;

m[i][j] = 7.0;

The first subscript generates a MatrixRow, and it is this object that receives the second subscript, and ultimately produces the desired element. One thing to note is that indexing a constant matrix will produce a value, while indexing a nonconstant matrix will produce a reference. In order to support this behavior, there is, in addition to MatrixRow, a second class ConstMatrixRow.

Execution time can be improved by expanding small function bodies inline. Subscript operators, as well as other small functions, have all been given inline definitions. This means the function bodies are prefixed with the inline modifier and appear in the interface file, rather than in the implementation file.

In addition to the subscript operators, operations such as += (an elementwise addition of one matrix to another), + (the element-wise addition of two matrices), and \* (scalar or matrix product, depending on the parameters). A stream output operator produces a printed representation of the matrix.

Instead of 3 × 3 matrices, the class Matrix will need to represent matrices with an arbitrary number of rows and columns. Because the size of the internal array cannot be determined at compile time, it is necessary to use dynamic memory allocation. The big three (copy constructor, assignment operator, and destructor) will need to be written. Access methods will return the number of rows and columns in a matrix, which will replace the static constants used in the original version.

In the code, template parameters are only needed for the element type, and not for the row and column bounds. This makes it easier to overload the arithmetic operators.

To create the template version of the class, template prefixes need to be added to the class declarations and member function definitions. References to the class Matrix need to be replaced by Matrix<T>. Member function definitions will need to be moved into the header file.

Because zero may not be an appropriate value for all template parameter types (think of Matrix<string>), the loop that initializes elements in the constructor will need to be eliminated.

To modify these classes to use exceptions we first define a pair of exception classes. The class MatrixMismatchException inherits from the standard class invalid\_argument, while the class MatrixIndexException inherits from out\_of\_range. By using the standard library classes that inherit from class exception, both of these exception classes can be caught by a single catch clause, as shown in the test program. In order to provide a more meaningful error message we include the out of range index value as an argument to the constructor for the class MatrixIndexException. Because the base exception class requires a string, the index value is formatted using a private internal method before the constructor for the base class is called.

**Overloading the Function Call Operator**

The function call operator permits the development of objects that are created using classes, but can be used as if they were functions. The advantage of this is that the class can, like all classes, encapsulate and carry its own data values.

The use of this operator is most easily illustrated with an example. Suppose you need a function that returns a random number between 1 and 100. This is easy to write:

int rand\_100()

{

return 1 + rand() % 100;

}

Now imagine that you need a similar function that returns a random value between a and b, where a and b are quantities that will not be known until run time. This is much harder to do with ordinary functions. The solution is to create a class, and initialize the values of a and b with the constructor:

class RandomInt

{

public:

RandomInt(int ia, int ib);

int operator()();

private:

int a, b;

};

RandomInt::RandomInt(int ia, int ib) : a(ia), b(ib) {}

int RandomInt::operator()()

{

return a + rand() % (b - a + 1);

}

You declare an instance of this class just like you would any other object. The difference is that once created, the object can be invoked just as you would a function:

RandomInt a(7, 15); // Return random values from 7 to 15

cout << "one random value " << a() << "\n";

cout << "and another " << a() << "\n";

Because it can be used as if it were a function, such a value is termed a function object. As with assignment, this operator can only be defined as a member function. The function call operator is unique, in that it is the only operator in the C++ language for which the number of arguments is not fixed. Binary operators, such as operator <<, take only two arguments. Some operators, such as +, have both a one-argument and two-argument form. Like a function, the function call operator takes as many arguments as are specified in the function heading. This can be illustrated by the following generalization of the random number generator, which provides three different overloaded versions of the operator. When no arguments are specified it works as before. When one argument is given it is used as the new upper bound, and when two arguments are specified they are used as the lower and upper bounds:

class RandomInt

{

public:

RandomInt(int ia, int ib);

int operator()();

int operator()(int nb);

int operator()(int na, nb);

private:

int a, b;

};

RandomInt::RandomInt(int ia, int ib) : a(ia), b(ib) {}

int RandomInt::operator()()

{

return a + rand() % (b - a + 1);

}

int RandomInt::operator()(int nb)

{

return a + rand() % (nb - a + 1);

}

int RandomInt::operator()(int na, int nb)

{

return na + rand() % (nb - na + 1);

}

The function selected will be determined by the number of arguments provided: RandomInt r(3, 7);

cout << "random value between 3 and 7 " << r() << "\n";

cout << "random value between 3 and 10 " << r(10) << "\n";

cout << "random value between 23 and 30 " << r(23, 30) << "\n";

**Inline Functions**

When function bodies are very short, such as with the comparison operators in class Rational, the execution time involved in the function call (pushing arguments on to the activation record stack, transfer of control, return from function, popping arguments off the stack) can often be greater than the execution time required by the body of the function itself. Moreover, many processors fetch and decode subsequent instructions as they execute an instruction. Branches and function calls slow down this mechanism.

When an inline function is invoked, the compiler expands the body of the function in place. This eliminates the execution cost of the function call, but results in multiple copies of the function body (one each time the function is called). For this reason, inline functions should only be used when the function body is very short.

To create an inline function the keyword inline is placed before the function body. The function definition is then placed in the interface file, not the implementation file(works with member and nonmember functions):

inline Rational& Rational::operator++()

{

numer += denom;

return \*this;

}

## **What to Submit**

Submit in Canvas the following files (**case sensitive**):

* main.cpp (test harness)
* matrix.h