C++ Programming
Homework 7

- Define a template class Matrix that behaves like a two dimmensional array. I'll outline it here for you, but you'll have to do all the C++ specific implementation. Define each template class in a .h file with the same name as the class. Do not define a .cpp file for either of the template classes. Define the methods inside the class declaration (not separately down in the .cpp file as we have been doing for non-template classes). REMEMBER: DO NOT COPY AND PASTE ANY CODE BECAUSE YOU ARE STILL LEARNING THE LANGUAGE.
- (30 points) Convert the following class Array into a template where the element type is a template parameter. Define it in a file Array.h with the method definitions inside the class declaration. **Be sure you know what all the iomanip objects do in the print function (and know how to use them). A final quiz question depends on your knowing and understanding them.**

```
#include <cassert>
class Array
{
private:
  int len;
  int * buf;
public:
  Array( int newLen )
    : len( newLen ), buf( new int[newLen] )
  {
  }
  Array( const Array & l )
    : len( l.len ), buf( new int[l.len] )
  {
    for ( int i = 0; i < l.len; i++ )
      buf[i] = l.buf[i]; // note = is incorrect if buf
elements are pointers
  }
  int length()
  {
    return len;
  }
  int & operator [] ( int i )
  {
    assert( 0 <= i && i < len );
    return buf[i];
  }
  void print( ostream & out )
  {
    for (int i = 0; i < len; ++i)
      out << setw(8) << setprecision(2) << fixed << right <<
```

```
buf[i]; // #include <iomanip>
  }
  friend ostream & operator << ( ostream & out, Array & a )
  {
    a.print( out );
    return out;
  }
  friend ostream & operator << ( ostream & out, Array * ap )
  {
    ap->print( out );
    return out;
  }
  // note the overloading of operator << on a pointer as
well
};
```

- (30 points) Write the following class Matrix as a template where the element type is a template parameter. Define it entirely in Matrix.h similar to how you did for template class Array.

```
class Matrix
{
private:
  int rows, cols;
  // define m as an Array of Array pointers using the
  // template you defined above
public:
  Matrix( int newRows, int newCols )
    : rows( newRows ), cols( newCols ), m( rows )
  {
    for (int i = 0; i < rows; i++ )
      m[i] = new Array < Element >( cols );
  }
  int numRows()
  {
    return rows;
  }
  int numCols()
  {
    return cols;
  }
  Array < Element > & operator [] ( int row )
  {
    return * m[row];
  }
  void print( ostream & out )
```

```
  {
    // write this one too, but use Array::operator<<
  }
  friend ostream & operator << ( ostream & out, Matrix & m )
  {
    m.print( out );
    return out;
  }
};
```

- (20 points) Ensure your Matrix works correctly with the following main function. Define main in main.cpp which includes Matrix.h. Test Matrix with at least two different element types (e.g., int and double). I sketched the program below off the top of my head (without compiling), so there may be some errors, but it will be good practice for you to eliminate them.

```
template
  < typename T >
void fillMatrix( Matrix <T> & m )
{
  int i, j;
  for ( i = 0; i < m.numRows(); i++ )
    m[i][0] = T();
  for ( j = 0; j < m.numCols(); j++ )
    m[0][j] = T();
  for ( i = 1; i < m.numRows(); i++ )
    for ( j = 1; j < m.numCols(); j++ )
    {
      m[i][j] = T(i * j);
    }
}
void test_int_matrix()
{ // here is a start, but make it better
    Matrix < int > m(10,5);
    fillMatrix( m );
    cout << m;
}
void test_double_matrix()
{ // here is a start, but make it better
    Matrix < double > M(8,10);
    fillMatrix( M );
    cout << M;
}
int main()
{
   test_int_matrix();
```

```
    test_double_matrix();
    return 0;
}
```

- (20 points) Define an exception class, IndexOutOfBoundsException, and have your Array and Matrix class throw an instance of it if any index is out of bounds. Add another function, test_double_matrix_exceptions(), which will catch this exception and do something appropriate, like print an error message and continue processing. Modify your main, by adding a call to this new function as shown below. Ensure that generate_exception indexes a Matrix out of bounds to cause this exception to be thrown. Add proper throw specifications to any function that may throw this exception.

```
void generate_exception( Matrix < double > &m )
{
     // insert a for loop that cause the exception
 }
void test_double_matrix_exceptions()
{
     // PUT YOUR TRY/CATCH AROUND THE INSTRUCTIONS BELOW
     cout << "Starting...\n";
     Matrix < double > M(8,10);
     fillMatrix( M );
     cout << M;
     generate_exception( M );
     cout << "Done\n";
}
int main()
{
   for (int i=0; i<3; ++i)
   {
       test_int_matrix();
       test_double_matrix();
       test_double_matrix_exceptions();
   }
   return 0;
}
```