The purpose of this assignment is to refresh and sharpen your understanding of class templates, dynamic memory allocation, and operator overloading in C++. These features appear in examples throughout our textbook and will figure prominently in code to be written for subsequent assignments.

Consider an abstract data type representing a table of values with functionality for appending rows and columns, modifying the contents of specified cells, extracting a sub-table, and applying a specified operation to all values in the table. Your task is to implement a class template (called `Table`) for this type.

I will test your code using two programs, *table_tester.cpp* and *table_tester2.cpp*, which are provided to you for your own testing purposes. These programs implicitly specify all of the required `Table` constructors and methods. In other words, the constructors and public methods to be provided by your class are precisely the ones needed for the test programs to compile. You will also need a friend function overloading the stream insertion operator.

What to submit: Your header and implementation files (*table.h* and *table.cpp*). Nothing else. Remember, your code must compile to be eligible for credit. The test programs must also compile *without modification* when including your table implementation. If you encounter syntax errors that you do not understand, I will be glad to help you fix them.

Technical details:

1. Client code can construct a `Table` object by specifying the number of rows and columns, or by specifying a single integer if both dimensions are the same. Since your class has pointer data, it needs a copy constructor, destructor, and overloaded assignment operator.

2. The stream insertion (leftshift) operator is overloaded so that client code can output a table in the same way that primitive type values are outputted; for example:

```
Table<int> table(3, 5);
          ⋮
cout << table << endl;
```

Assume that cell (0, 0) is in the top-left corner of the table.

3. The function call operator is overloaded to take two integer arguments indicating the row and column of a cell in the table. The function returns a reference to the value at that position. This allows client code to modify table entries with a natural syntax that is similar to array bracket notation. For example, to assign the value 10 to the cell in row 3, column 4, one would write:

```
table(3, 4) = 10;
```

This is equivalent to:

```
table.operator()(3, 4) = 10;
```

4. The function call operator is also overloaded to take four integer arguments representing the top-left and bottom-right corners of a subtable. This operator does not modify the object but rather returns a new table with values extracted from the original. See the test programs for an illustration of its use.

5. The append_rows and append_cols methods can be used to append a table below or to the right. For example:

```
Table x(3, 5);
Table y(4, 5);
      ⋮
Table z = x.append_rows(y);
```

After this code executes, z would be a 7 x 5 table that can be visualized as a copy of table x on top of a copy of table y. Use assertions to ensure that the dimensions of the tables are compatible for appending.

Note that append_rows and append_cols do not modify the object on which they are called, they return a new table obtained by concatenating the two operands. Basically, any kind of operation on a table is possible through a suitable sequence of append and subtable operations.

6. The addition operator is overloaded as a function template to take an argument of the following type:

```
T (*)(T)
```

Here T is the type parameter for the Table class. This type is read as *pointer-to-function-that-takes-a-T-value-and-returns-a-T-value*. It a pointer, so the parameter would appear after the pointer operator, like this:

```
T (*f)(T)
```

The idea is to provide an intuitive syntax for updating all of the values in a table according to a given rule. In the first test program, it is used like this:

```
cout << t + square;
```

where t is a Table<int> and square is a stand-alone function that takes an integer and returns the square of that integer. The equivalent function call syntax is:

```
cout.operator<<(t.operator+(square));
```

The expression t + square does not modify t, it returns a new Table containing the squares of values in t.

7. Notice that the setw manipulator is used in the test programs. The operator syntax is:

```
cout << setw(5) << t;
```

The equivalent function call syntax is:

```
operator<<(cout.operator<<(setw(5)), t);
```

This has the effect of writing the table values right justified in columns of width 5. Making this work for your Table class is a slightly tricky matter. In your overloaded stream insertion operator, the first thing you should do is obtain the width associated to the ostream operand. You can do this by calling its width method. Then you can call setw with the appropriate argument for each table element that is outputted.