C++ Lab 2: Matrix Madness

Constructors, destructors, and dynamic memory management!

This week you'll design and implement a C++ class which can be used to represent and manipulate integer matrices.

What you need to do:

Define a class called `Matrix` which handles all of the operations illustrated in the example "user's code" below:

```
#include <iostream>
#include "Matrix.hh"

using namespace std;

int main() {

  // Create a new matrix with two rows and three columns
  Matrix a(2, 3);

  // Print out some information about the size of this matrix:
  cout << "Matrix a is " << a.getrows() << " rows tall." << endl;
  cout << "Matrix a is " << a.getcols() << " columns wide." << endl;

  // Print out the contents of this matrix (should be all zeroes!):
  for(int r = 0; r < a.getrows(); r++) {
    for(int c = 0; c < a.getcols(); c++) {
      cout << "Element (" << r << ", " << c << ") = " << a.getelem(r,c) << endl;
    }
  }

  // Fill in a few values
  a.setelem(1, 2, -5280);  // bottom right element is now -5280
  a.setelem(0, 1, 123);    // top center element is now 123

  // Create an identical copy of this matrix
  Matrix b = a;

  // Change the original matrix some more
  a.setelem(1, 2, 555);  // bottom right element is now 555

  // Examine some elements of each matrix
  cout << "(1,2) of a = " << a.getelem(1,2) << " [should be 555]" << endl;
  cout << "(1,2) of b = " << b.getelem(1,2) << " [should be -5280]" << endl;

  // So if a and b are different, let's copy a into a new matrix and add b to it:
  Matrix c = a;
  c.add(b);

  // Now let's copy c into another new matrix and subtract a from it:
  Matrix d = c;
  d.subtract(a);

  // Hmm...  that means d should be b, no?
  if (d.equals(b)) {
    cout << "Yay!  d = b!" << endl;
  } else {
    cout << "Uh-oh!  Something went wrong; d isn't b!" << endl;
  }

  // Let's create a tiny 0 by 0 matrix using the default constructor:
  Matrix e;
  cout << "0x0 matrix e is " << e.getrows() << " by " << e.getcols() << endl;

  // Of course, e and d are different, since they have different sizes!
  if (!e.equals(d)) {
    cout << "e and d are indeed different!" << endl;
  } else {
    cout << "Oh well, back to the drawing board...." << endl;
  }

  // Okay, enough of this; destroy all those matrices and end the program!
  return 0;
}
```

(You can get the above file here: simplematrixtest.cc)

Got all that? Here's a quick summary:

- A class declaration provided in `Matrix.hh`
- The class must be called `Matrix` (implementation should be in `Matrix.cc`)

- The matrix has integer elements
- The default constructor creates a 0 by 0 matrix
- The two-argument constructor creates an all-zero matrix of the specified size
- The copy constructor creates a new, independent copy of a matrix
  (Changing the original shouldn't alter the copy, or vice versa!)
- The destructor frees any memory used by the matrix
- Arithmetic methods `add` and `subtract` that mutate the instance upon which they're called
- A comparison method called `equals`
- Accessor methods `getrows`, `getcols` and `getelem`
- A mutator method `setelem`

Dealing With Abuse

Whether it's arithmetic with matrices of incompatible sizes, or a simple attempt to create a matrix with a negative number of columns, sooner or later, someone will try to do something stupid with your matrix class. The question is, what should you do about it?

There are actually several ways to handle such situations, but for this lab you should employ **assertions** to ensure that your class is used properly. Also note that when you are checking inputs, it's not sufficient to only guard a few member functions. That's like locking the front door of a house, but leaving the back door wide open. So, **make sure you write checks for *every* member function that could receive invalid inputs.** Don't leave any holes in your code, because eventually they will come back to bite you.

A quick example of using C's `assert` in a C++ program:

```
// Use the C++ name for the Standard C header assert.h
#include <cassert>

float mySqrt(float x) {
  assert(x >= 0);

  // compute the square root
  ...

  return answer;
}
```

Testing Your Class

For now, use the program above (or one of your own design) to test out all of the methods of your class. We will be expanding on our `Matrix` class next week and we'll give you a more comprehensive test suite then.

To compile the lab, use this command:

```
g++ -Wall simplematrixtest.cc Matrix.cc -o simplematrixtest
```

Any warning or error messages you may see will most likely be due to problems in your class declaration or implementation. If you can't figure out the problem reasonably quickly, ask for help! You should attempt to fix all warnings and errors before proceeding.

Please bear in mind that `simplematrixtest` cannot and will not test everything. Just because it doesn't turn up any problems doesn't mean that none exist. There are also many areas (error handling, coding style, class design) which an automated program can only test in a very limited fashion, if at all. Remember that the quality of your entire program, not just the output of one test, is what matters.

Anything Else?

Between this page, the provided test program, and the material from lecture, you should have a pretty good idea of what your class needs to do and what its public interface needs to look like. How you choose to implement these features is up to you. If you're not sure where to start, ask for help.

In general, be very careful about adding extra features such as different operators, alternate constructors, or new methods to your class. Though you are not prohibited from doing so, you should think hard before expanding your class's public interface. It's often the case that adding too many "features" to a class will actually make it more difficult to use or maintain, especially if the new methods expose too many of the details of your class's internal implementation. The KISS principle is often invoked when making software-design decisions: "Keep It Simple, Stupid!" It's the best way to avoid unnecessary headaches.
