# C++ Lab 3: More Matrix Meddling

*Operator overloading, and `const`-correctness!*

---

This week, we'll work `Matrix` into a production-worthy class, and then we'll compile it against a more rigorous test program to put it through its paces.

**NOTE:**
Guidelines for operator overloading in C++ are available here. Be sure to review them carefully! These guidelines will help ensure that your implementations are correct, and that you don't do unnecessary work.

Here's what you need to do:

1. **Overload the assignment operator for your matrix class.**

   In other words, given matrices `a` and `b`, we should be able to write:

   ```
   a = b;
   ```

   Be sure to handle self-assignment properly! Also support operator chaining, *e.g.*

   ```
   a = b = c;
   ```

   Remember that chains of assignment operators are evaluated from right to left. Keep in mind that code from a well-written assignment operator can often be used in a copy constructor, and vice versa.

2. **Overload the *compound assignment operators* `+=` and `-=`.**

   Each of these should work just like your intuition would tell you they would. (Otherwise, overloading operators would probably be a bad idea.) You should remove your `add` and `subtract` member functions, since these operators do the same thing, but the code you wrote last week will probably be quite useful.

   Just like last week, if the RHS matrix is the wrong dimensions, trip an assertion to indicate the problem.

3. **Overload the `+`, `-` operators, too.**

   Remember that these methods will return a *new* `Matrix` object. (*Hint:* Use the code you wrote in part 2 to make this simple.)

4. **Overload the `*` and `*=` operators.**

   Again, you can just define one, and then define the other in terms of the first one. Refer to any linear algebra textbook if you've forgotten how to multiply matrices.

   *Remember that the `Matrix` class supports non-square matrices,* and multiplication of non-square matrices is well defined, as long as their dimensions satisfy particular constraints. Of course, this makes the implementation of `*=` more interesting, since the dimensions of the LHS can change. (Refer to the lecture 3 slides if you need any hints of how to do this more easily!)

5. **Replace your `equals` member function with an operator overload of `==` too.**

   Just like with your `add` and `subtract` member functions, `==` should replace the `equals` method, since it's now unnecessary. Your class' public interface should be as clean and simple as possible.

   Also add an `operator!=` member function for symmetry. Remember to implement this in terms of `==` to guarantee correctness, and to make your job easier.

   Note that unlike the arithmetic operations, it is perfectly acceptable to compare two matrices with different dimensions. The result of the comparison will simply be false. So, don't assert that the matrices are the same size in your `==` and `!=` implementations; just return false if they are different sizes.

6. **Evaluate all of your methods for *const correctness*.**

   Use the `const` keyword as necessary, to ensure that your `Matrix` class provides the maximum possible protection from side effects. For our purposes, there are basically three places that you should consider putting a `const` keyword:

   1. before an argument type, e.g.,

      ```
      MyClass & someFunc(const SomeClass &other)
      ```

   2. after a member-function name, e.g.,

      ```
      MyClass & someFunc(...) const
      ```

   3. before a return type, e.g.,

      ```
      const MyClass & someFunc(...)
      ```

or

```
const MyClass anotherFunc(...)
```

(You will usually encounter combinations of these three scenarios.)

Make sure you understand the implications of each of these, and then apply them meaningfully to your code. *Remember that, in particular, there are well defined best-practices for operator overloads.* Follow these best-practices!

7. **Compile your Matrix class against a new, more rigorous test suite `checkmatrix.cc` like so:**

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

Run `checkmatrix` and use it to help you debug any errors in your matrix implementation. You should not consider this week's lab complete until your class passes the test suite with flying colors. Of course, if you get totally stuck, ask for help!

---