Assignment 1: `Matrix`

Due December 4, 11:59 PM

Introduction

A *matrix* is a two-dimensional array of numbers. For example, here is a 4 x 5 matrix:

$$A = \begin{pmatrix} 3 & 1 & 4 & 2 & 7 \\ 1 & 5 & 9 & 1 & 8 \\ 2 & 6 & 5 & 2 & 8 \\ 3 & 5 & 8 & 1 & 8 \end{pmatrix}$$

Notice that a 4 x 5 matrix has four rows and five columns.

Given a matrix A, we use the notation A_{ij} to denote the element in row i, column j of that matrix. Conventions vary on whether matrix elements are zero- or one-indexed, but since we're C++ programmers we'll assume that the entries are zero-indexed. This means that if A is the above matrix, $A_{00} = 3$, $A_{12} = 9$, and $A_{21} = 6$.

Matrices are used in all fields of computer science and applied mathematics, including graphics, machine learning, program analysis, and physics simulations. Matrices are useful for a variety of reasons, one of which is that they provide a compact syntax for manipulating large quantities of numbers. For instance, we can add two matrices of the same dimensions by summing their components. Thus if

$$A = \begin{pmatrix} 3 & 1 & 4 & 2 & 7 \\ 1 & 5 & 9 & 1 & 8 \\ 2 & 6 & 5 & 2 & 8 \\ 3 & 5 & 8 & 1 & 8 \end{pmatrix}; B = \begin{pmatrix} 9 & 7 & 9 & 4 & 5 \\ 3 & 2 & 3 & 9 & 0 \\ 8 & 4 & 6 & 4 & 5 \\ 2 & 6 & 4 & 2 & 3 \end{pmatrix}$$

Then

$$A + B = \begin{pmatrix} 12 & 8 & 13 & 6 & 12 \\ 4 & 7 & 12 & 10 & 8 \\ 10 & 10 & 11 & 6 & 13 \\ 5 & 11 & 12 & 3 & 11 \end{pmatrix}$$

Suppose that we want to design a C++ class that encapsulates a matrix, which we'll call `Matrix`. One simple method for implementing `Matrix` would be to back the `Matrix` with a two-dimensional array. If we use the `grid` class from Chapter 25 of the course reader, we could implement `Matrix` as follows: *(You might want to read over the interface for `grid` before proceeding)*

```
class Matrix
{
public:
    /* Constructor creates a matrix of a specific size that's all zeros. */
    Matrix(size_t rows, size_t cols) : elems(rows, cols) {}

    /* Read and write values in the matrix. */
    double getAt(size_t row, size_t col) const
    {
        return elems[row][col];
    }
    void setAt(size_t row, size_t col, double value)
    {
        elems[row][col] = value;
    }

    /* Query the size of the matrix. */
    size_t numRows() const { return elems.numRows(); }
    size_t numCols() const { return elems.numCols(); }

private:
    /* Layer the matrix on top of a grid. */
    grid<double> elems;
};
```

Now that we have a basic implementation of the `Matrix`, let's see how we might implement matrix addition. We'll define a version of `operator+` that takes in two matrices and returns their sum. One implementation is as follows:

```
const Matrix operator+ (const Matrix& one, const Matrix& two)
{
    /* ... check that dimensions agree ... */

    Matrix result(one.numRows(), one.numCols());
    for(size_t row = 0; row < one.numRows(); ++row)
        for(size_t col = 0; col < one.numCols(); ++col)
            result.setAt(row, col, one.getAt(row, col) + two.getAt(row, col));
    return result;
}
```

This function constructs a new matrix of the same dimensions as the original matrices, then sets each element in the resulting matrix to be the sum of the elements from the two input matrices. We could similarly define `operator-` to subtract two matrices and `operator*` to multiply a matrix and a scalar (this multiplies every element in the matrix by the scalar). With these operators in tow, we can write expressions like the following:

```
/* Create four matrices - A, B, C, and D - of the same size. */
Matrix A = /* ... */, B = /* ... */, C = /* ... */, D = /* ... */;

/* Create a new matrix from them. */
Matrix E = 2*A - 4*B + C + D;
```

Through the magic of operator overloading, this will set `E` to be a matrix where $E_{ij} = 2A_{ij} - 4B_{ij} + C_{ij} + D_{ij}$. But exactly how efficient it is to construct this matrix `E`? Recall that C++ will interpret the above statement as

```
Matrix E =
 operator+ (operator+ (operator- (operator* (2, A), operator* (4, B)), C), D);
```

This code contains five calls to overloaded operators. If the matrices are of dimension m x n, then each operator takes O(mn) time, since the implementation has to iterate over every element of each input matrix at least once. Since the number of calls is constant, this code runs in O(mn).

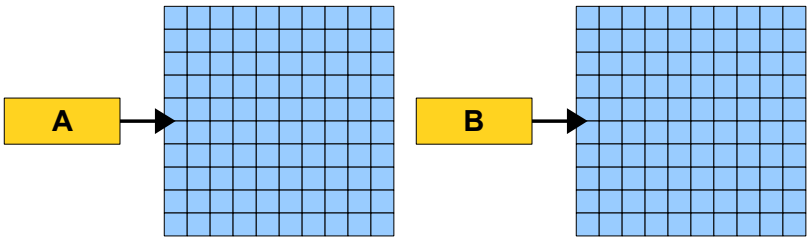
However, suppose that we only need to look at the first column of the matrix `E`. In that case, we don't actually need to compute every entry of `E`, just the entries in the first column. We can compute these values in O(m) time, which is much faster than computing the full sum and then looking at just the first column. This generalizes to a more important result – if only need to look at k elements of the sum, we can do so in O(k) time by just computing the sum of the elements in those positions.

The problem is that at the time that we compute the matrix `E`, we have no idea what operations we're going to perform on it. We might look at every element of `E`, in which case O(mn) is as good as we can do. Then again, we might look at only k elements, in which case we shouldn't do the full computation. Unfortunately, in general it's impossible to figure out what might be done with `E` later in the program, so we're forced to compute the value of every element of `E` when constructing it.

Or are we?

An Alternative Implementation

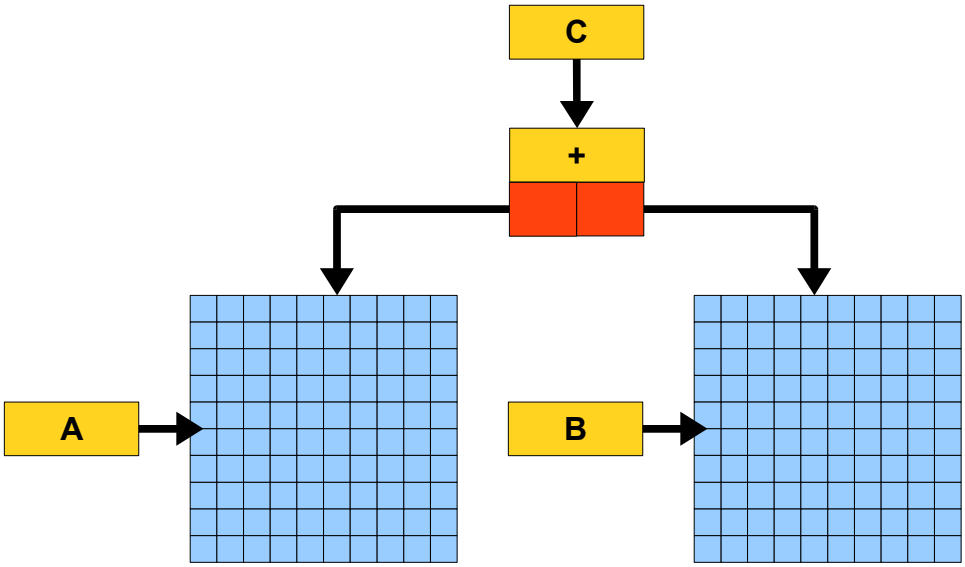
There is a fantastically clever trick that will allow us to get the best of both worlds – creating a matrix that is the sum of two other matrices, but only computing the elements of the resulting matrix that we actually use. The trick is to change the underlying representation of `Matrix`. Instead of implementing `Matrix` as an actual matrix, we can implement `Matrix` as an *expression tree* for a matrix. To understand how this works, suppose that we are given two matrices `A` and `B` of equal dimensions. Then instead of storing the actual matrices `A` and `B` inside of the objects `A` and `B`, we will have `A` and `B` store *pointers* to those matrices, as shown here:



Now suppose that we write the following code:

```
Matrix C = A + B;
```

Instead of creating a new matrix and populating it with the values of `A + B`, we create an expression tree for `A + B` and then set the matrix `C` so that it points at the expression tree. This is shown here:



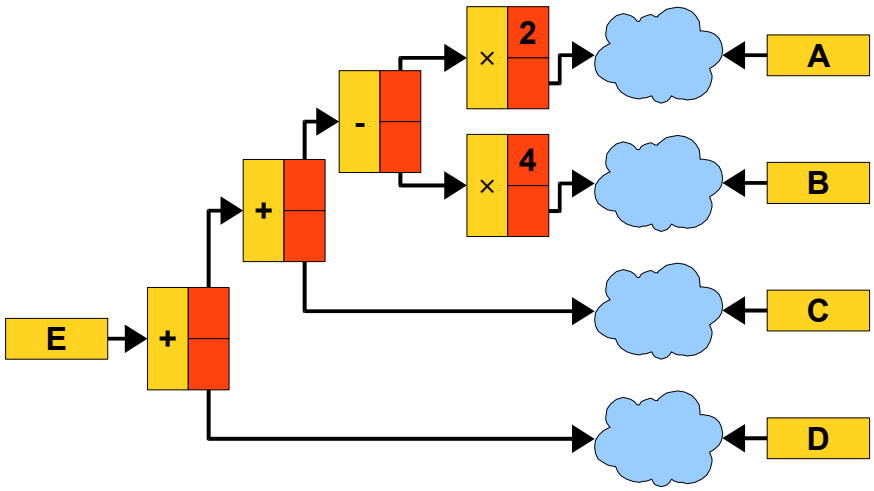
Here, `+` is an expression tree node that represents the sum of two other matrix expressions. The matrix `C` is then implemented as a pointer to this node. Whenever we read an element out of `C`, the result is dynamically computed from the expression tree. For example, suppose we want to read the element at position (2, 3) of `C`. `C` queries the `+` node for the value at position (2, 3). This `+` node in turn queries the concrete matrices pointed at by `A` and `B` for their values at position (2, 3), then returns the sum. In other words, `C` *behaves* as if it is the sum of `A` and `B`, since any time we look up an element of `C` it has the value of the corresponding element of `A + B`. But `C` *isn't* the sum of `A` and `B` – it's just an expression tree that evaluates to the sum – and so we don't have to compute the full matrix when creating `C`. In other words, we're taking the idea of separation of interface from implementation and running wild with it. From a client's perspective, we're adding, subtracting, and multiplying matrices. In reality, we're being lazy and only computing matrix entries when the user wants them.

To see this setup in action, let's return to our original motivating example:

```
/* Create four matrices - A, B, C, and D - of the same size. */
Matrix A = /* ... */, B = /* ... */, C = /* ... */, D = /* ... */;

/* Create a new matrix from them. */
Matrix E = 2*A - 4*B + C + D;
```

If we represent matrices using expression trees, the resulting of this operation is as follows:



Notice that the matrices `A`, `B`, `C`, and `D` are represented as pointers to "clouds" instead of pointers to concrete matrices. This is because `A`, `B`, `C`, and `D` are themselves implemented as matrix expression trees, so it's unclear whether they point directly to concrete matrices or whether they point to more complicated expressions. The clouds in this diagram thus signify "some expression tree."

The expression trees for `A`, `B`, `C`, and `D` are joined together by a combination of `+` and `-` nodes, along with two `×` nodes which represent the product of a matrix by a scalar. If you work out the expression pointed at by `E` from top to bottom, you'll notice that `E = ((2*A - 4*B) + C) + D`, which is exactly the expression we want.

Of course, this solution does not magically make matrix math faster. Because we compute matrix elements using an expression tree, if we read the same matrix elements over and over again, we will reevaluate the expression for those elements multiple times. Similarly, if we do read every element out of the matrix, this approach will be slower than if we had eagerly computed every value because of the overhead from the expression tree. Overall, though, this is a very promising implementation strategy. The next question, of course, is exactly how this translates into actual C++ code.

Implementing the Approach

Implementing `Matrix` using expression trees may at first seem a difficult task, but it turns out to be surprisingly simple. We'll break the problem down into two subproblems:

1. Designing and implementing the expression tree hierarchy.
2. Designing and implementing the `Matrix` class.

We'll begin by discussing what an expression tree of matrices looks like. As you've seen in CS106B/X, an expression tree is a tree composed of polymorphic *expression classes* that each know how to evaluate to a value. In our case, we represent matrix expression trees with a base class called `MatrixExpr`, which is defined as follows:

```
class MatrixExpr
{
public:
    virtual ~MatrixExpr() {} // Polymorphic classes need virtual destructors

    /* Return the number of rows and columns in the matrix. */
    virtual size_t numRows() const = 0;
    virtual size_t numCols() const = 0;

    /* Returns the value of the element at a given position. */
    virtual double getAt(size_t row, size_t col) const = 0;
};
```

The `MatrixExpr` class contains three member functions: `numRows` and `numCols`, which return the number of rows or columns in the matrix, respectively; and `getAt`, which takes in an index and returns the element at that position.

When working with expression trees in CS106B/X, one of the simplest expression classes you encountered was `ConstantExp`, which represents an integer literal. In our case, a "constant expression" is a *matrix* literal. We'll call this class `ConcreteMatrix` to indicate that it actually represents a matrix, and can implement it as follows:

```
class ConcreteMatrix: public MatrixExpr
{
public:
    /* Constructor creates a matrix of a specific size that's all zeros. */
    ConcreteMatrix(size_t rows, size_t cols) : elems(rows, cols) {}

    /* Read and write values in the matrix. */
    double getAt(size_t row, size_t col) const { return elems[row][col]; }

    /* Query the size of the matrix. */
    size_t numRows() const { return elems.numRows(); }
    size_t numCols() const { return elems.numCols(); }

private:
    /* Layer the matrix on top of a grid. */
    grid<double> elems;
};
```

If you'll notice, this implementation is almost the same as our initial implementation of `Matrix`. This is no coincidence – in fact, we would expect this to be the case because at some point our matrix expression tree has to bottom out in an actual matrix!

Now that we have an expression class corresponding to a concrete matrix, we can consider defining an expression class that represents the sum of two matrices. We'll call this class `SumExpr` and can define it as follows:*

* An alternative approach would be to implement a generic `CompoundExpr` class as suggested in the CS106B/X course reader. If you find this approach more intuitive, by all means feel free to implement sums and differences of matrices this way.

```
/* NOTE: When we talk about resource management later in this handout, we
 * will need to revise the definition of this class.  Do not directly use this
 * code in your solution!
 */
class SumExpr: public MatrixExpr
{
public:
    /* Construct the sum of two matrices from two matrix expressions.  We
     * will assume that the dimensions of the matrices agree.
     */
    SumExpr(MatrixExpr* left, MatrixExpr* right) : lhs(left), rhs(right) {}

    /* Destructor cleans up resources. */
    ~SumExpr()
    {
        delete lhs;
        delete rhs;
    }

    /* Return the number of rows / columns in the matrix, which are equal to
     * the number of rows and columns in either of the child matrices.
     */
    size_t numRows() const { return lhs->numRows(); }
    size_t numCols() const { return lhs->numCols(); }

    /* The element at position (row, col) is given as the sum of the elements
     * at position (row, col) in the two child matrices.
     */
    double getAt(size_t row, size_t col) const
    {
        return lhs->getAt(row, col) + rhs->getAt(row, col);
    }

private:
   MatrixExpr* lhs, *rhs;
};
```

The `SumExpr` constructor takes in two `MatrixExpr*`s, one for the left-hand side and one for the right. Looking up the number of rows or columns in the expression simply yields the number of rows or columns in either child expression (this implementation arbitrarily uses the left child). The implementation of `getAt` looks up the elements in the proper position in the left and right child expressions, then returns their sum.

Of course, there are many more expression classes we could create – you'll be implementing some of them in the course of this assignment – but hopefully these two examples give you a better feel for what they look like.

An interesting point about the expression tree we're building is that, unlike the expression trees you've seen in CS106B/X, we don't need to write our own code for parsing expressions and determining order of operations. Because we're constructing the expression tree using overloaded operators, *C++ will automatically parse the expression for us*. Recall that in our example with the matrix `E`, the declaration

```
Matrix E = 2*A - 4*B + C + D;
```

is equivalent to

```
Matrix E = operator+
               (operator+
                    (operator-
                          (operator*(2, A),
                           operator*(4, B)),
                     C),
                D);
```

If each of these overloaded operators generates a `Matrix` whose expression tree applies the appropriate operator to its arguments, then these function calls will construct the correct expression tree.

Now that we've seen how we can represent expression trees, let's see how we might implement `Matrix`. `Matrix` is little more than a wrapper around the expression tree. Here's one possible implementation:

```
class Matrix
{
public:
    /* Constructor creates a matrix of a specific size. */
    Matrix(size_t rows, size_t cols);

    /* Destructor and copy functions are covered in the next section. */

    /* Element access and size queries. */
    double getAt(size_t row, size_t col) const
    {
        return expr->getAt(row, col);
    }

    /* setAt covered in a later section. */

    size_t numRows() const { return expr->numRows(); }
    size_t numCols() const { return expr->numCols(); }

    /* operator+ joins two matrices using a SumExpr. */
    friend const Matrix operator+ (const Matrix& one, const Matrix& two);

private:
    /* This private constructor is used to construct a Matrix from an
     * expression tree.
     */
    explicit Matrix(MatrixExpr* exprTree) : expr(exprTree) {}

    /* The expression tree for this Matrix */
    MatrixExpr* expr;
};

/* The Matrix constructor initializes the Matrix to a ConcreteMatrix of the
 * proper size.
 */
Matrix::Matrix(size_t rows, size_t cols): expr(new ConcreteMatrix(rows, cols))
{
}
```

```
/* The + operator on two matrices computes the sum by joining the two
 * matrices together via a SumExpr.
 */
const Matrix operator+ (const Matrix& one, const Matrix& two)
{
    /* ... some check that the dimensions agree ... */

    /* Because we're a friend of Matrix, we can look inside the Matrix to see
     * its expression tree and the private constructor.
     */
    return Matrix(new SumExpr(one.expr, two.expr));
}
```

Notice that we've made `operator+` a `friend` of `Matrix`, since the implementation of `operator+` now need access to the `expr` data member.

While the implementation of `Matrix` is for the most part straightforward, there are two subtle details that we need to address: resource management and mutating operations.

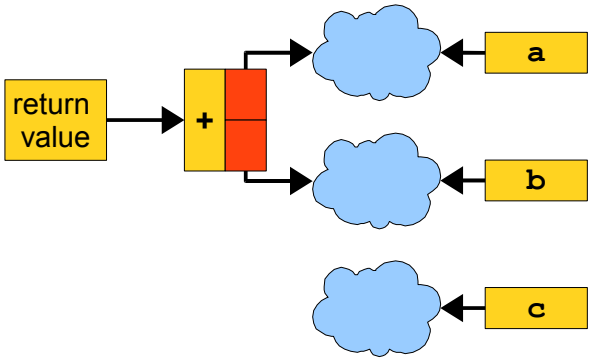
Tricky Aspect 1: Resource Management

Because `Matrix` is now implemented as a pointer to a dynamically-allocated expression tree, we need to ensure that the tree is deallocated at some point. This ends up being a bit trickier than you might expect. For example, consider the following code:

```
Matrix DoSomething()
{
    Matrix a = /* ... create some 10 x 10 matrix ... */
    Matrix b = /* ... create some 10 x 10 matrix ... */
    Matrix c = /* ... create some 10 x 10 matrix ... */

    return a + b;
}
```

This function creates three local `Matrix` objects, then returns the sum of `a` and `b`. In memory, at the point where the matrix `a + b` is returned, memory looks like this:



When `a` and `b` go out of scope as the function returns, they should not deallocate their expression trees because those trees are referenced (albeit indirectly) by the return value. On the other hand, when `c` leaves scope, `c` should deallocate the memory for its expression tree because no other matrices reference that tree. How can we differentiate between these two cases?

Fortunately, resource management is simple thanks to the SharedPtr class we covered in lecture. If you'll recall, SharedPtr is designed to handle resource management exactly along the lines of what's described above. SharedPtrs hold a reference to a resource and track the number of other SharedPtrs that also refer to that resource. Whenever SharedPtr is destroyed, if it is the last SharedPtr to its resource, it deallocates the resource; otherwise nothing happens. Using this fact, we can replace all instances of MatrixExpr* with instances of SharedPtr<MatrixExpr>. This automates reference management and ensures that resources are cleaned up only when they are no longer used. For example, we might rewrite the Matrix class to use SharedPtr as follows:

```
class Matrix
{
public:
    /* Constructor creates a matrix of a specific size. */
    Matrix(size_t rows, size_t cols);

    /* No destructor or copy functions - SharedPtr does this for us. */

    /* Element access and size queries. */
    double getAt(size_t row, size_t col) const;
    size_t numRows() const;
    size_t numCols() const;

    /* setAt covered in a later section. */

    /* operator+ joins two matrices using a SumExpr. */
    friend const Matrix operator+ (const Matrix& one, const Matrix& two);

private:
    /* Construct a Matrix from an expression tree. */
    explicit Matrix(SharedPtr<MatrixExpr> exprTree);

    SharedPtr<MatrixExpr> expr;
};
```

We have to be careful to use SharedPtr consistently throughout our implementation, since if we use a raw pointer to refer to a MatrixExpr the SharedPtrs won't know about that reference and might accidentally clean up an expression tree that's still in use. Consequently, we'll need to change our implementation of SumExpr to use SharedPtr as well. This is shown here:

```
class SumExpr: public MatrixExpr
{
public:
    SumExpr(SharedPtr<MatrixExpr> left, SharedPtr<MatrixExpr> right)
      : lhs(left), rhs(right) {}

    /* No destructor needed. */

    size_t numRows() const { return lhs->numRows(); }
    size_t numCols() const { return lhs->numCols(); }

    double getAt(size_t row, size_t col) const
    {
        return lhs->getAt(row, col) + rhs->getAt(row, col);
    }
private:
    SharedPtr<MatrixExpr> lhs, rhs;
};
```

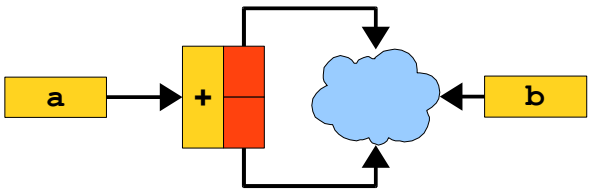
As you implement other expression classes, make sure to use `SharedPtr`s instead of raw pointers.

Tricky Aspect 2: Copy on Write

There is one more tricky aspect to the `Matrix` implementation: if elements are computed via an expression tree, how do we support a function like `setAt` that lets us overwrite a single matrix element? Consider the following example:

```
Matrix a = /* ... some matrix ... */;
Matrix b = a;
a = a + a;
```

In memory, this is represented as follows:



(Make sure you understand why this looks the way it does before proceeding! Try tracing out each step individually to see what's happening.)

Suppose we now write the following:

```
b.setAt(0, 0, 137);
```

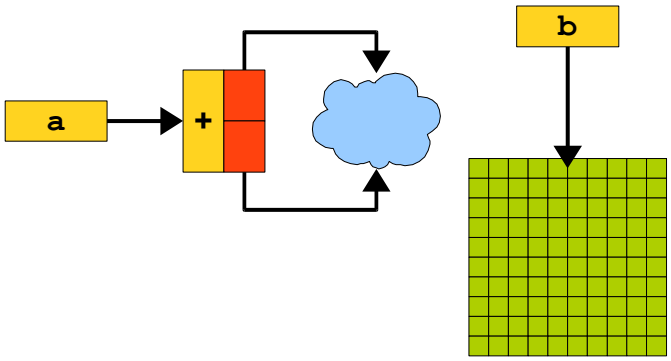
This change means that we need to change the `b` matrix so that the element at position (0, 0) is equal to 137. This poses two challenges:

1. The elements of `b` are stored as an expression tree and computed dynamically, meaning that there might not be a concrete matrix we can modify to change the value at position (0, 0) to 137.
2. The matrix `a` indirectly references the same matrix as `b`, so changes we make to `b`'s expression tree might alter `a`.

Problem (1) above stems from the fact that `b` isn't actually represented as a concrete matrix, and problem (2) stems from the fact that `b` shares its matrix with `a`. We can solve both of these problems at once using the following strategy: whenever we overwrite a matrix element in some matrix `A`, we create a new `ConcreteMatrix`, fill it in with the values of the `A`, then set `A` to use this `ConcreteMatrix` as its expression tree. We then overwrite the specified element with the client's value. This addresses the above problems as follows:

1. Because we change `A`'s expression tree into a `ConcreteMatrix`, we can easily set an element of `A` to a value by overwriting the matching element of the `ConcreteMatrix`.
2. Because the `ConcreteMatrix` we create for `A` is not shared with any other `Matrix` instances, updates to `A`'s expression tree do not change the values of any other matrices.

To get a better sense of how this works in practice, let's trace through what happens in the above situation. To set element (0, 0) of `b` to 137, we first flatten `b`'s tree into a `ConcreteMatrix` and change `b` so that it uses this matrix as its expression tree. This is shown here:



Notice that `b` points to its own concrete matrix, but `a` still uses the old expression tree. Now that `b`'s elements are stored in a concrete matrix, we can safely overwrite the value at position (0, 0) with 137.

This strategy works, but is costly; overwriting a single element of a matrix causes the entire matrix to be evaluated. However, we can apply an optimization to reduce the overhead. Suppose that we have just executed the above statement, causing `b` to refer to its own `ConcreteMatrix`. What happens if we overwrite another value in `b`'s matrix? Since `b` now points to a `ConcreteMatrix` which no other `Matrix` objects reference, we can simply go in and change the underlying `ConcreteMatrix` without worrying that the write will affect another `Matrix`. This is essentially a souped-up version of the *copy-on-write* optimization we talked about briefly at the end of Thursday's lecture.

To summarize, the general pattern for writing elements into a `Matrix` is as follows:

If either:

- The matrix does not point at a `ConcreteMatrix` expression.
- The matrix is not the only matrix pointing at its expression.

Then:

1. Create a new `ConcreteMatrix` expression of the same dimensions as the matrix.
2. Set each element of the `ConcreteMatrix` class to the values of the source matrix by evaluating the expression tree at each point.

Finally, overwrite the matrix element with the client's value.

The Assignment

This lengthy introduction hopefully has given you a good sense for how to implement a `Matrix` class using expression trees. Your assignment is to build off of the techniques covered in this handout to implement `Matrix`, along with some additional extensions. Don't worry – you'll be provided a fair amount of starter code to work with, and we've just covered the two most challenging implementation details.

To help you work through the code `Matrix` class, I've broken down the assignment into four smaller pieces. You'll be provided testing code to help check that each of the steps works correctly, so hopefully each step will be self-contained.

Task 0: Familiarize yourself with the starter code.

There are several starter files for this assignment containing a lot of prewritten code. Specifically, the starter code contains:

- An interface for `MatrixExpr`.
- A partial implementation of `Matrix`.
- A partial implementation of `ConcreteMatrix`.
- A complete implementation of `SumExpr`.
- An implementation of `SharedPtr` and `grid`.
- Testing code.

Before you start hacking away at the code base, take some time to read over the starter files and the comments to get a feeling for how they work and what functions are at your disposal. If you have any questions about how the code works, let me know and I'll be glad to clarify.

Task 1: Implement `setAt`

Your first task is to implement the `Matrix`'s `setAt` function, which has the following signature:

```
void setAt(size_t row, size_t col, double value);
```

`setAt` should set the element at position (`row`, `col`) to `value` using the algorithm mentioned above. To do this, you'll need to be able to determine whether the expression pointed at by a `Matrix` is a `ConcreteMatrix`. There are many techniques you can use to implement this check, of which two stand out as strong candidates:

- **`dynamic_cast`**. As mentioned in lecture, the `dynamic_cast` operator performs a typecast from one type to another, handing back a null pointer if the cast fails. This can be used to check if a pointer of type `MatrixExpr*` is pointing at a `ConcreteMatrix`; if the result of `dynamic_cast`-ing the pointer to a `ConcreteMatrix*` is non-null, the pointer points to a `ConcreteMatrix`.

  Unfortunately, there's a catch – `dynamic_cast` can only be applied to raw pointers, not `SharedPtr`s. Mixing raw pointers and `SharedPtr`s can be tricky, and to make life simpler I've provided a function called `dynamic_pointer_cast` that behaves like `dynamic_cast` except that it works on `SharedPtr`s. For example:

```
SharedPtr<ConcreteMatrix> ptr = dynamic_pointer_cast<ConcreteMatrix>(m);
if(ptr == NULL)
{
    /* ... m doesn't point at a ConcreteMatrix ... */
}
```

  There is also a `static_pointer_cast` operator that mimics `static_cast` and can be used to perform unchecked typecasts on `SharedPtr`s.
- **`typeid`**. `typeid` is an operator that takes in a type or expression and returns an object of type `type_info` containing information about the argument's type. You can use `typeid` as follows to check whether the expression tree referenced by a `Matrix` is a `ConcreteMatrix`:

```
if(typeid(*m) != typeid(ConcreteMatrix))
{
    /* ... m doesn't point at a ConcreteMatrix ... */
}
```

  Once you know that the `SharedPtr` points to a `ConcreteMatrix`, you can downcast the pointer using `static_pointer_cast`, as mentioned above. To use `typeid`, you will need to `#include <typeinfo>` at the top of your code.

  If you do use `typeid` to determine what type of object is referred to by a pointer, make sure that you pass the *pointee* rather than the *pointer* into `typeid`. Otherwise you'll end up getting the static type of the pointer rather than the dynamic type of the pointee. For example, the following code is legal but doesn't work correctly:

```
if(typeid(m) != typeid(ConcreteMatrix)) // Always true
{
    /* ... m doesn't point at a ConcreteMatrix ... */
}
```

When you think that you have a working implementation, be sure to run the tests in `test-harness.cpp` before proceeding. This will help double-check that your implementation works correctly.

Task 2: Implement the remaining mathematical operators

The version of `Matrix` provided in the starter code only supports `operator+`, but there are several other matrix operations that you may also want to perform. For example, you can subtract two matrices by performing a componentwise subtraction, and can multiply a matrix by a real number by scaling all of the elements of the matrix by that real number.

Your next task is to implement the following functions on the `Matrix` class:

```
class Matrix
{
public:
    /* ... as before ... */

    /**
     * Given two matrices of the same dimension, returns a new matrix of
     * the same dimension whose elements are equal to the componentwise
     * difference of the elements in the first matrix and the second.
     */
    friend const Matrix operator- (const Matrix& one, const Matrix& two);

    /**
     * Given a matrix and a scalar, returns a new matrix whose elements are
     * equal to the elements of the source matrix multiplied by the scalar.
     * Note that there are two versions of this function so that both
     * myMatrix * myScalar and myScalar * myMatrix are legal.
     */
    friend const Matrix operator* (const Matrix& m, double scalar);
    friend const Matrix operator* (double scalar, const Matrix& m);

    /**
     * Given a matrix and a scalar, returns a new matrix whose elements are
     * equal to the elements of the source matrix divided by the scalar.
     * Unlike multiplication, this operation is only defined for the
     * syntax myMatrix / myScalar, since the reverse (myScalar / myMatrix)
     * is not mathematically well-defined.
     */
    friend const Matrix operator/ (const Matrix& m, double scalar);

    /**
     * Returns a matrix equal to the input matrix except that the sign of
     * every element has been inverted.  For example, given the matrix
     *
     *     | 1 2 |
     * m = | 3 4 |
     *
     *                      |-1 -2|
     * The value of -m is   |-3 -4|
     */
    friend const Matrix operator- (const Matrix& m);
};
```

As with `operator+`, these functions should *not* deep-copy the matrix, but should instead return matrices that use expression trees to compute their values. You will need to implement at least one new subclass of `MatrixExpr` to make this code work correctly. As a major time-saving tip, you can use the following identities to implement some of these functions in terms of one another:

```
A - B = A + (-B)
A * k = k * A
A / k = A * (1/k)
   -A = -1 * A
```

You can test your functions using `test-harness.cpp`; see the file comments for more information.

Task 3: Implement extensions

At this point, you have developed a working `Matrix` class that uses expression trees to reduce the amount of computation needed for common matrix operations. However, this class is far from complete and there are several interesting extensions that you can build on top of `Matrix`. Rather than dictating what other features you might want to add on top of `Matrix`, I'm leaving this up to you. *Implement at least one interesting extension to the* `Matrix` *class*. My definition of "interesting" is fairly flexible – it might be a feature might be one that makes you go "Wow! That's really cool!" or it might be a programming task that teaches you something new about what's possible using C++. You can implement whatever extensions you'd like, but if you'd like some suggestions, you might want to try out some of these ideas:

- **Matrix multiplication.** One major operation on matrices that we did not implement in this assignment is *matrix multiplication*. Given two matrices A and B of dimension m x n and n x p, respectively, the product AB is defined as the m x p matrix such that

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

  Matrix multiplication is a particularly expensive operation – without very clever optimizations, multiplying an m x n matrix and an n x p matrix takes O(mnp) time. Implement `operator*` on two `Matrix` objects to lazily compute the product of the two matrices (that is, the result is an expression tree for the product).
- **Matrix transpose.** Given an m x n matrix **A**, the *transpose* of **A**, denoted $\mathbf{A}^T$, is the n x m matrix such that $\mathbf{A}^T_{ij} = \mathbf{A}_{ji}$. That is, the transpose of **A** is the matrix **A** with the rows and columns swapped. Using expression trees, it's extremely easy to compute a matrix transpose – just create a `MatrixExpr` class whose `getAt` function flips the order of the parameters and invokes `getAt` on a stored `MatrixExpr`. Write a function `Transpose` that computes the transpose of a matrix.
- **Matrix tiling.** Given an m x n matrix $\boldsymbol{A}$, a *matrix tiling* of **A** is a matrix formed by tiling **A** horizontally and vertically some number of times. For example, if **A** is the matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, then tiling **A** thrice horizontally and twice vertically yields

$$\begin{pmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{pmatrix}$$

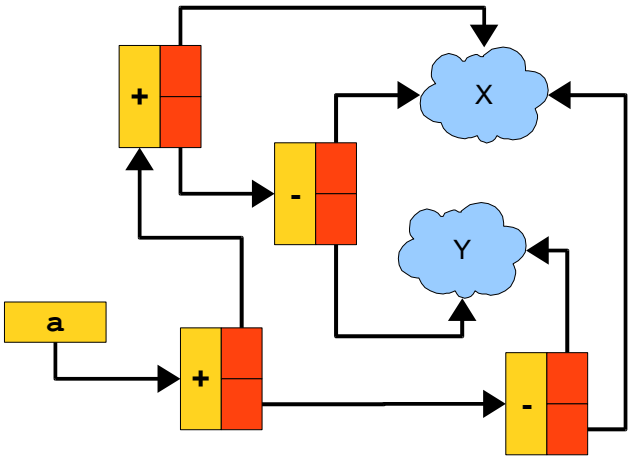
  Using expression trees, we can construct a matrix tiling by simply storing the matrix we want to tile and then using modular arithmetic to map each point in the tiling back into the original matrix. This makes it possible to talk about arbitrarily large matrix tilings without actually having to represent the tiling in memory – we just store the matrix being tiled and compute the elements of the tiling dynamically. Write a function `TileMatrix` that takes in a matrix and a number of repeats in each direction, then returns a matrix equal to the input matrix tiled that many times.

- **Diagonal matrices**. A *diagonal matrix* is a (usually square) matrix that is zero everywhere except on the main diagonal. For example, the following matrix is diagonal:

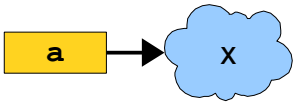
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

  Given a diagonal matrix of dimension n x n, we can store the diagonal matrix naïvely in $O(n^2)$ space by encoding the entire matrix. A much more efficient way of storing a diagonal matrix using only $O(n)$ space is to store just the elements on the diagonal. If we then look up an element in the matrix, the value is 0 if it is off of the diagonal and otherwise equal to the element on the diagonal in the given position. A special case of a diagonal matrix is the *identity matrix*, which is a diagonal matrix whose diagonal entries are all ones. Provide the user some means for creating diagonal and identity matrices.
- **Function application.** Given a function that takes in a real number and returns a real number (such as square root or cosine), we can define the application of the function to a matrix as the matrix whose elements are the result of applying the function to each element in the original matrix. Write a function `Apply` that takes in a `Matrix` and a function, then returns a `Matrix` whose elements are equal to the elements of the input `Matrix` with the specified function applied. You may find the `Function` class from the Chapter 30 of the course reader helpful.
- **Forcing evaluation.** Provide a means for clients of `Matrix` to force evaluation of the `Matrix` into a `ConcreteMatrix` using techniques similar to what you've already developed for the `setAt` case.
- **Compound assignment operators**. The compound assignment operators are operators like `+=` and `*=` that apply an operator to two values and store the result back into one of the values. The current implementation of `Matrix` supports `operator+` etc. without the matching compound assignment operators. Add these extra operators to the `Matrix` class.
- **Relational operators.** Currently, we cannot store `Matrix` objects in an STL `set` or as keys in an STL `map` because we have not implemented the < operator or any of the other relational operators. Implement the six relational operators on `Matrix`, minimizing the number of element lookups (i.e. computations) that you need to perform.
- **Parameterized matrix types.** We chose to represent the `Matrix` as a matrix of `double`s, but it is meaningful to create matrices of other types as well (`float`s or `complex<double>`s, for example). Parameterize the `Matrix` class over the type of elements that will be stored in it.
- **Value caching.** Every time we query a `Matrix` for the value of an element at a position, the element is recomputed using the expression tree. It would be useful to have a means for efficiently caching and looking up values that have already been computed. See if you can find a way to have the `Matrix` cache its values up to some point, and then fully evaluate the matrix when it looks like the user will be looking up a "large number" of values (for some definition of "large number").

- **Unified `getAt`/`setAt` syntax.** The `grid` class from the course reader uses operator overloading to allow clients to select elements out of the `grid` using the syntax `myGrid[row][col]`. See if you can implement this functionality for the `Matrix`. If you want to really flex your C++ muscles, you can try having the bracket syntax return a proxy object that detects whether the `Matrix` element is being read or written, which can be used to avoid deep-copying matrix elements unnecessarily. Send me an email or come talk to me after lecture and I can provide a description of this technique.
- **Compile-time dimensional analysis**. When working with the `Matrix` class, it is possible to add or subtract matrices whose dimensions do not agree; this produces undefined behavior at runtime. In general, we cannot detect all possible dimension mismatches without actually running the program, but in the special case where the matrix dimensions are known at compile-time it is possible to encode information about the matrix dimensions into the type system. For example, a 3 x 5 matrix might be a `Matrix<3, 5>`, while a 10 x 10 matrix would be a `Matrix<10, 10>`. Using techniques like those described in Chapter 24 of the course reader, modify the `Matrix` class so that it is illegal to add or subtract matrices whose dimensions don't agree.
- **Expression tree optimization.** Consider the following expression tree:



This monstrosity is completely equivalent to the much simpler tree



The expression trees generated by the `Matrix` class are not subject to any sort of optimization and it is possible to end up with very complex expressions that simplify down to much more compact (and therefore efficient) operations. For a challenge that will make you feel like you are ready for anything, see if you can implement an algorithm that simplifies the expression trees generated by the `Matrix` class. If you end up with a working implementation, I would love to show it off to future CS106L classes.

Hints and Advice

This assignment is not as difficult as it may appear. Although this handout is fairly lengthy, the key concepts can be expressed succinctly and elegantly in C++. My implementation of the assignment runs under 300 lines, including comments and several of the extensions described above. That said, there are some points to be aware of when implementing `Matrix`. Here are some tips and tricks on how to avoid common pitfalls:

- *Make sure that you understand what code has already been written for you*. There is a lot of starter code that should make it easier to implement the trickier parts of the assignment. Before writing any code, take some time to read over what functions are available to you. In particular, make sure that you check out `SharedPtr`, particularly `dynamic_pointer_cast` and `static_pointer_cast`.
- *Make sure that your code consistently uses `SharedPtr` when working with `MatrixExpr`s*. If you use raw pointers in some places and `SharedPtr`s in others, you are likely to have incorrect reference counts on your `MatrixExpr` instances and will probably experience particularly nasty runtime errors. You should be able to implement this assignment without ever using a raw `MatrixExpr*`.
- *Be as lazy as possible*. With the exception of `setAt`, none of the functions that you write for this assignment should require you to eagerly evaluate any matrices. As much as possible, try to implement the matrix operations using expression trees of `MatrixExpr` rather than `for`-looping over `MatrixExpr`s and calling the `getAt` function. The provided test code contains several tests that will run for inordinately long periods of time unless you're lazily evaluating your matrices, which should help diagnose any cases where you're being too eager. And no, this tip doesn't mean that you should wait to the last minute to start the assignment. ☺
- *Test your code thoroughly!* You are provided a test suite that can help smoke out latent errors in your code. Once you've completed a task, make sure to test it extensively before moving on to the next section. Otherwise, you risk errors in one part of the code masquerading as errors in another.
- *Don't hesitate to ask questions!* **The point of this assignment is to give you a chance to play around with C++ and build cool software, not to punish you for not understanding a particular aspect of the language**. If you're having trouble understanding the starter code, run into inexplicable runtime errors, or can't seem to get some part of the assignment working, please send me an email or talk to me after lecture. I genuinely love this material and want to help you learn it, so don't hesitate to ask questions if you need to.

Deliverables

Once you've completed the above tasks, please submit the following:

- Any source files you modified to implement the `Matrix` class or your extensions. You don't need to submit starter files that you didn't modify.
- A test program that showcases at least one of the extensions you implemented. For example, if you implemented lazy matrix multiplication, this program might create two matrices, take their product, and print a single element from the result.

Once you've completed the assignment, email the above files to htiek@cs.stanford.edu. Please include your name and SUNetID on top of all of your files. Then pat yourself on the back – you've just completed the last assignment for CS106L and are now a veteran C++ programmer!