

Assignment 6

This assignment introduces the concept of *operator overloading*, the use of the `friend` keyword, and the concept of "const correctness".

In mathematics, a *matrix* (plural *matrices*) is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. The dimensions of the matrix below are 2 x 2 (read "two by two"), because there are two rows and two columns.

$$\begin{bmatrix} 1 & 9 \\ 2 & 5 \end{bmatrix}$$

The individual items in a matrix are called its *elements* or *entries*. Provided that they are the same size (have the same number of rows and columns), two matrices can be added or subtracted element by element. The rule for matrix multiplication, however, is that two matrices can be multiplied only when the number of columns in the first equals the number of rows in the second. Any matrix can be multiplied element-wise by a scalar from its associated field.

If the matrix is square (has the same number of rows as columns), it is possible to deduce some of its properties by computing its *determinant*. For example, a square matrix has an inverse if and only if its determinant is not zero.

Note that in mathematics, the rows and columns of a matrix are numbered starting from 1, not 0 like a C++ array.

1. Initial Setup

1. Log in to Unix.
2. Run the `setup` script for Assignment 6 by typing:

```
setup 6
```

2. Files You Must Write

You will need to write one class for this assignment. A main program to test your class will be provided.

The `matrix` class

The `matrix` class represents a two by two square matrix using a two-dimensional array of integers. This class should be implemented as two separate files.

The class definition should be placed in a header file called `matrix.h`. Include header guards to make it impossible to accidentally `#include` it more than once in the same source code file.

The `matrix` class should contain the following `private` data member:

- a two-dimensional array of integers with two rows and two columns. The member function descriptions below will refer to this as the *matrix array*.

In addition to the data member described above, the class definition should also contain prototypes for the member functions described below.

The implementations of the class member functions should be placed in a separate source code file called `matrix.cpp`. Make sure to `#include "matrix.h"` at the top of this file.

The `matrix` class should have the following member functions:

- "Identity matrix" constructor

Parameters: This default constructor has no parameters.

Logic: Set the elements of the matrix array to the "identity matrix", such that all the elements on the main diagonal are equal to 1 and all other elements are equal to 0, e.g.:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- "Array initialization" constructor

Parameters: This constructor takes one argument, a two-dimensional array of integers with two rows and two columns.

Logic: Set the elements of the matrix array to the corresponding elements in the array passed into the constructor.

- `determinant()`

Parameters: None.

Returns: The integer determinant of the `matrix` object.

Logic: The determinant of a 2-by-2 matrix is given by

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

- `operator+()`

Parameters: This member function takes one parameter, a reference to a constant `matrix` object, representing the right operand of the matrix addition expression. The left operand of the expression is represented by `this`, which points to the `matrix` object that called the member function.

Returns: The result of the matrix addition of the left and right operands (a new `matrix` object).

Logic: The sum $\mathbf{A} + \mathbf{B}$ of two 2-by-2 matrices \mathbf{A} and \mathbf{B} is calculated entrywise: $(\mathbf{A} + \mathbf{B})_{ij} = \mathbf{A}_{ij} + \mathbf{B}_{ij}$, where $1 \leq i \leq 2$ and $1 \leq j \leq 2$:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

For example:

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 3+5 & 1+6 \\ 4+2 & 2+3 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 6 & 5 \end{bmatrix}$$

- `operator*()`

Parameters: This member function takes one parameter, an integer representing the right operand of the scalar multiplication. The left operand of the expression is represented by `this`, which points to the `matrix` object that called the member function.

Returns: The result of multiplying the elements of the matrix left operand by the integer right operand (a new `matrix` object).

Logic: The product \mathbf{Ac} of a matrix \mathbf{A} and a number c (also called a scalar in the parlance of abstract algebra) is computed by multiplying every entry of \mathbf{A} by c : $(\mathbf{Ac})_{ij} = \mathbf{A}_{ij} \cdot c$. For example:

$$\begin{bmatrix} 2 & 4 \\ 3 & 1 \end{bmatrix} \cdot 2 = \begin{bmatrix} 2 \cdot 2 & 4 \cdot 2 \\ 3 \cdot 2 & 1 \cdot 2 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 6 & 2 \end{bmatrix}$$

- `operator*()`

Parameters: This member function takes one parameter, a reference to a constant `matrix` object, representing the right operand of the matrix multiplication expression. The left operand of the expression is represented by `this`, which points to the `matrix` object that called the member function.

Returns: The result of multiplying the elements of the matrix left operand by the matrix right operand (a new `matrix` object).

Logic: The product of two 2-by-2 matrices is given by

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a \cdot e + b \cdot g & a \cdot f + b \cdot h \\ c \cdot e + d \cdot g & c \cdot f + d \cdot h \end{bmatrix}$$

For example:

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 8 & 7 \end{bmatrix} = \begin{bmatrix} 3 \cdot 5 + 1 \cdot 8 & 3 \cdot 6 + 1 \cdot 7 \\ 4 \cdot 5 + 2 \cdot 8 & 4 \cdot 6 + 2 \cdot 7 \end{bmatrix} = \begin{bmatrix} 23 & 25 \\ 36 & 38 \end{bmatrix}$$

Note that unlike ordinary multiplication, matrix multiplication is not commutative.

- `operator==()`

Parameters: This member function takes one parameter, a reference to a constant `matrix` object, representing the right operand of the relational expression. The left operand of the expression is represented by `this`, which points to the `matrix` object that called the member function.

Returns: A boolean value.

Logic: Return `true` if all elements of the left operand are equal to the corresponding elements of the right operand. Otherwise, the member function should return `false`.

- `operator!=()`

Parameters: This member function takes one parameter, a reference to a constant `matrix` object, representing the right operand of the relational expression. The left operand of the expression is represented by `this`, which points to the `matrix` object that called the member function.

Returns: A Boolean value.

Logic: Return `false` if the left operand equals the right operand. Otherwise, the member function should return `true`.

Note: Member functions that do not alter the data members of the object that called them should be declared to be `const`. That will allow them to be called using `const` objects.

In addition to the member functions described above, you will need to write two standalone functions. These functions are not (and can not be) member functions. You should

1. Include a `friend` declaration for each of these functions in the `matrix` class declaration.
2. Put the definitions for these functions in `matrix.cpp`.

- `operator<<()`

This function will be called when the stream insertion operator `<<` is used to print a `matrix` object. For example:

```
int array1[2][2] = {{1, 9}, {2, 5}}    // 2d array

matrix m1(array1);                     // A matrix created from that 2d
array

cout << m1;                            // Compiler generated function call:
operator<<(cout, m1);
```

In the example code above, the `ostream` object `cout` (the left operand in the stream insertion expression) will be passed to the function as the first parameter, while the `matrix` object `m1` (the right operand) will be passed to the function as the second parameter.

Parameters: This function takes two parameters. The first is a reference to an `ostream` object, representing the left operand of the stream insertion expression.

The second is a reference to a constant `matrix` object, representing the right operand of the expression.

Returns: A reference to an `ostream` object (i.e., the first parameter).

Logic: Print the elements of the matrix separated by commas. Use square brackets around each row of the matrix and around the matrix as a whole.

For example, printing the object `m1` from the example above, which represents the 2x2 matrix

$$\begin{bmatrix} 1 & 9 \\ 2 & 5 \end{bmatrix}$$

should produce the output

```
[[1, 9], [2, 5]]
```

- `operator*()`

Parameters: This member function takes two parameters, an integer representing the left operand of the scalar multiplication, and a reference to a constant `matrix` object, representing the right operand of the scalar multiplication.

Returns: The result of multiplying the elements of the matrix right operand by the integer left operand (a new `matrix` object).

Logic: The product $c\mathbf{A}$ of a number c (also called a scalar in the parlance of abstract algebra) and a matrix \mathbf{A} is computed by multiplying every entry of \mathbf{A} by c : $(c\mathbf{A})_{ij} = c \cdot \mathbf{A}_{ij}$. For example:

$$2 \cdot \begin{bmatrix} 2 & 4 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 2 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 6 & 2 \end{bmatrix}$$

3. Files We Give You

The setup script will create the directory `Assign6` under your `csci241` directory. It will copy a makefile named `makefile` to the assignment directory.

You will also receive a driver program named `assign6.cpp` that contains a `main()` function that will test your `matrix` class.

4. Output

The correct output for this assignment is shown below:

1. Testing identity matrix constructor

```
m1 = [[1, 0], [0, 1]]
```

2. Testing array initialization constructor

```
m2 = [[5, 7], [3, 2]]
m3 = [[2, 3], [1, 4]]
```

3. Testing determinant

```
det[[5, 7], [3, 2]] = -11
det[[2, 3], [1, 4]] = 5
```

4. Testing matrix addition

```
[[5, 7], [3, 2]] + [[2, 3], [1, 4]] = [[7, 10], [4, 6]]
[[2, 3], [1, 4]] + [[5, 7], [3, 2]] = [[7, 10], [4, 6]]
```

5. Testing scalar multiplication

```
[[5, 7], [3, 2]] * 2 = [[10, 14], [6, 4]]
4 * [[5, 7], [3, 2]] = [[20, 28], [12, 8]]
[[2, 3], [1, 4]] * 2 = [[4, 6], [2, 8]]
4 * [[2, 3], [1, 4]] = [[8, 12], [4, 16]]
```

6. Testing matrix multiplication

```
[[5, 7], [3, 2]] * [[2, 3], [1, 4]] = [[17, 43], [8, 17]]
[[2, 3], [1, 4]] * [[5, 7], [3, 2]] = [[19, 20], [17, 15]]
[[2, 3], [1, 4]] * [[1, 0], [0, 1]] = [[2, 3], [1, 4]]
[[1, 0], [0, 1]] * [[2, 3], [1, 4]] = [[2, 3], [1, 4]]
```

`det(m2 * m3)` and `det(m2) * det(m3)` are equal

7. Testing relational operators

```
[[5, 7], [3, 2]] and [[5, 7], [3, 2]] are equal
[[5, 7], [3, 2]] and [[2, 3], [1, 4]] are not equal
[[2, 3], [1, 4]] and [[5, 7], [3, 2]] are not equal
[[1, 0], [0, 1]] and [[2, 3], [1, 4]] are not equal
```

If you would like a copy of this output to compare against your own program's output using the `diff` command, it is available on Unix at the pathname

```
/home/turing/t90kjm1/CS241/Output/Fall2022/Assign6/output6.txt.
```

5. Hints

The driver program is designed to make this assignment easy to develop incrementally. Simply comment out all of the lines of the driver program that call functions that you haven't written yet. You should be able to write, test, and debug function at a time. I would suggest writing the functions in the following order:

1. "Identity matrix" constructor
2. `operator<<()` function
3. "Array initialization" constructor
4. `determinant()` method
5. `operator+()` method (matrix addition)
6. `operator*()` method (scalar multiplication with integer as right operand)
7. `operator*()` function (scalar multiplication with integer as left operand)
8. `operator*()` method (matrix multiplication)
9. `operator==()` method
10. `operator!=()` method

Remember that member functions that do not change any data members of the object that called them should be made `const` so that they may be called by `const` objects. If not, you will get the syntax error message "error: passing 'const matrix' as 'this' argument discards qualifiers" when the member function is called by a `const` object.