

4/11/2022

## Project\_2 Matrix Calculator Report

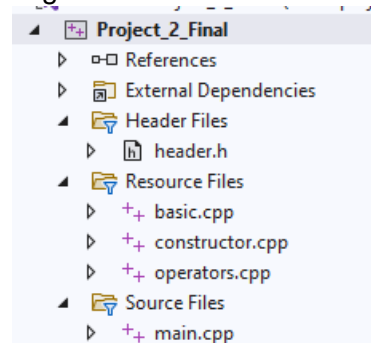
By: Ruize Ding

### 1. Abstract:

- a. The class covered functions, arrays, pointers, classes, and structures. The primary purpose of this project was to demonstrate proficiency in memory allocation and incorporate the concepts into a working C++ program. This project has us building a matrix calculator that performs arithmetic operations, in addition to performing Boolean, transpose, and identity matrix operations. To complete this project, we were first given the starter codes. These codes identified the classes, constructors, operators, and functions we had to write out. This is a case of applying pre-existing information to the problem of the matrix calculator. I used the class notes and lectures in addition to online resources such as the C++ reference website, Stack overflow, and discord to complete this project. In the end, I was able to build a successful matrix calculator that minimizes memory leaks and maximizes operator overloading.

### 2. Introduction:

- a. The following section will introduce the program and how the files were organized. The program contains the following files shown in the screenshot below. It was organized using a single header file and several resource files containing the necessary code for the classes and functions. The main function contains the output to the console and shows that the project is fully functioning.



- i.
  - ii. The "header.h" file is almost identical to the one given to us in the assignment report.
  - iii. The "basic.cpp" contains the setter and getter in addition to the clear, empty, reallocate and return rows/columns functions.
  - iv. The "constructors.cpp" file has the code for the constructors used in the header. There are four constructors in addition to a destructor: the default constructor, parameterized constructor, copy constructor, copy assignment, and the destructor.
- b. Memory allocation is important in C++ as it allows the use of pointers and passing variables by reference. This allows objects to exist beyond the current scope. These methods make the program much more effective and allow for more complex structures. Dynamic allocation for example saves space when memory is not being used and only allocates when it's needed. This improves efficiency.

### 3. Methods/Software:

- The following section will go in-depth through each individual section of code and explain the specific methods and contents.
- First is the “header.h” file, although this was given to us in the assignment outline, I have made a few changes.

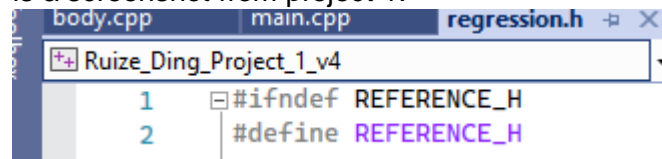
```

1  #pragma once
2
3  #include <string>      // std::string
4  #include <cstdint>    // std::size_t
5  #include <ostream>    // std::ostream
6
7  namespace mv1 {
8      // Matrix implemented with the C-style array, using a double and dynamic memory allocation
9      class Matrix {
10     public:
11         // Constructs an empty matrix (Default Constructor)
12         Matrix();
13
14         // Constructs and initialises a matrix (parameterized constructor)
15         Matrix(std::size_t row, std::size_t col, int fill = 0);
16
17         // Copy constructor
18         Matrix(const Matrix& rhs);
19
20         // Copy assignment
21         Matrix& operator=(const Matrix& rhs);
22
23         // Destructs matrix (destructor)
24         ~Matrix();
25
26         // Element mutator (setter)
27         int& at(std::size_t row, std::size_t col);
28         // Element accessor (getter)
29         int at(std::size_t row, std::size_t col) const;
30
31         void clear(); // Delete contents
32         bool empty() const; // 'true' if matrix contains no elements
33
34         std::size_t rows() const; // Returns the number of rows
35         std::size_t columns() const; // Returns the number of columns
36
37         // Compound assignment
38         Matrix& operator+=(const Matrix& rhs); // Matrix addition
39         Matrix& operator-=(const Matrix& rhs); // Matrix subtraction
40         Matrix& operator*=(const int rhs); // Scalar multiplication
41
42         friend Matrix make_identity(std::size_t n);
43
44         friend Matrix operator+(Matrix lhs, const Matrix& rhs);
45         friend Matrix operator-(Matrix lhs, const Matrix& rhs);
46
47         friend Matrix operator*(Matrix lhs, const int rhs);
48         friend Matrix operator*(const int lhs, Matrix rhs);
49         friend Matrix operator*(const Matrix& lhs, const Matrix& rhs);
50
51         friend std::ostream& operator<<(std::ostream& stream, const Matrix& rhs);
52         friend std::string csv(const Matrix& mat);
53
54         friend Matrix transpose(const Matrix& mat);
55
56         void reallocate(std::size_t rows, std::size_t cols, int fill = 0);
57     private:
58         std::size_t rows_;
59         std::size_t columns_;
60         int** data_;
61     };

```

- “#prgama once” which ensures that the file, the header, is only used once. It replaces the more inefficient “#ifndef” and “#define”. The following

is a screenshot from project 1.



The screenshot shows a code editor with three tabs: 'body.cpp', 'main.cpp', and 'regression.h'. The 'regression.h' tab is active. Below the tabs, there is a dropdown menu showing 'Ruize\_Ding\_Project\_1\_v4'. The code in the editor is as follows:

```
1  #ifndef REFERENCE_H
2  #define REFERENCE_H
```

- iii. We will be including the standard libraries for “string”, “size\_t”, and “ostream”. “#include <iostream>” will not be necessary for the header because there is nothing being outputted to the console. It will however be used in the main function.
- iv. The next part, “namespace mv1” helps organize the code better under a single namespace. Although not required in this project as all methods are handled under a single namespace, higher complexity projects with multiple namespaces and libraries should be organized using namespaces.
  1. In hindsight, my project 1 should have used namespaces for my linear regression function as I was importing several external plotting libraries.
- v. The class matrix contains all the necessary methods for building and performing operations onto a matrix. It contains all constructors; the setter and getter; clear and check empty functions; return row and column count functions; and the compound operator functions.
  1. The area highlighted in magenta as one is the first change from the provided header file. Here, I am using several friend functions to allow the functions outside this class to access the variables and functions inside the class.
  2. The next change is the location of the “reallocate” function. The given header had this function under private, making it not accessible outside of the class. I needed to use it in the main to change the contents of matrices, so I had to move it to public. I am inferring that the purpose of this being in private is to better organize parts of the constructor code. I have noticed that there are several portions of code in the various constructors that are almost identical. Instead of using the “this->” pointer, it is possible to use the reallocate function to condense the repeat code.
- vi. The constructors are then declared. There are three constructors, a copy assignment and a destructor.
- vii. After that, the setter and getter are declared. The setter is also known as the mutator and allows the program to write to the matrix. The getter is also known as the accessor and allows the program to read.
- viii. The void clear function clears any matrix by deleting all the contents inside. The code for this is identical to that of the destructor.
- ix. The bool empty function will check whether the function is empty. If the function is empty as in there are zero rows and columns, it will return true and false if not empty.
- x. The rows() and columns() functions return the number of rows and columns of the array.

- xi. The compound assignment operators allow the use of the compound equals arithmetic operators of  $+=$ ,  $-=$ ,  $*=$ . These compound arithmetic operators work identical to the traditional arithmetic operators of  $+$ ,  $-$ , and  $*$  they are programmed to replace.
- xii. Private integers and next up and contain variables that can only be accessed inside the class.
  - 1. The `std::size_t rows_` variable holds the number of rows in the matrix. The `std::size_t columns_` variable holds the number of columns. `std::size_t` specifies the number to be within the allowed maximum/minimum size of a matrix. This number is unsigned and ensures that there are no negatives as a matrix cannot be a negative number of rows/columns.
- c. The next section will go into the code in the header file that is not a part of the matrix class.

```

62 // Returns an n-by-n identity matrix
63 Matrix make_identity(std::size_t n);
64
65 Matrix operator+(Matrix lhs, const Matrix& rhs); // Matrix addition
66 Matrix operator-(Matrix lhs, const Matrix& rhs); // Matrix subtraction
67 Matrix operator*(Matrix lhs, const int rhs); // Scalar multiplication
68 Matrix operator*(const int lhs, Matrix rhs); // Scalar multiplication
69 Matrix operator*(const Matrix& lhs, const Matrix& rhs); // Matrix multiplication
70
71 /* Note about matrix multiplication:
72 - The simple implementation used here is cubic in time  $O(n^3)$ ,
73 which means it is infeasible for solving large problems
74 - Returns an empty matrix if either 'lhs' or 'rhs' is empty,
75 or if the inner dimensions are not equal (i.e., the number of
76 columns in 'lhs' does not equal the number of rows in 'rhs') */
77
78 // Comparison operator overloads
79 bool operator==(const Matrix& lhs, const Matrix& rhs);
80 bool operator!=(const Matrix& lhs, const Matrix& rhs);
81
82 // Stream insertion operator overload
83 std::ostream& operator<<(std::ostream& os, const Matrix& rhs);
84
85 // Returns matrix values in comma separated value (CSV) format
86 std::string csv(const Matrix& mat);
87
88 // Returns transpose of the matrix
89 Matrix transpose(const Matrix& mat);
90
91 } // namespace mvl

```

- i.
- ii. Basic it is a matrix calculator only the arithmetic operators make sense to be used like  $+$ ,  $-$ ,  $*$ . Boolean operators also make sense to be used because it allows the user to see if two matrices are equal.
- iii. The “make\_identity” function allows the creation of an identity matrix of size  $n$ . There is only one input required as identity matrices must be square.
- iv. The reason this code is not included inside the Matrix class is that there are too many inputs that it will overload if placed inside. There is however another way of writing the function as shown below.

```

Matrix operator+(const Matrix &rhs) const
{
    Matrix result(row, col);
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            result.matrix[i][j] = this->matrix[i][j] + rhs.matrix[i][j];
        }
    }
    return result;
}

```

- 1.
  2. This limits the overload to just one input and is the preferred way of writing the function. After asking around online, I believe it saves memory, but I am not 100% sure why this method is better.
- v. The following functions are the arithmetic operators. It is important to note that there are several multiplication operators. Based on the input that is fed through the overload, the function will determine which operator to use. There are three different types of multiplication involving matrices. The first is the matrix multiplication, if the user sends two matrices into the calculator, it will use this operator. The next is a scalar multiplied by a matrix. Lastly, it is a matrix multiplied by a scalar. All have a different function associated with a different calculation.
  - vi. The next bool functions allow the user to verify whether two matrices are equal. The “equals equals” operator (==) returns true if both functions are identical and returns false if they are not equal. The “not equals” operator (!=) returns true if the functions are not equal and false if the functions are equal.
  - vii. The next function is the stream insertion operator, this operator allows the use of the console output. Basically, the symbol of “<<” is programmed to be the console output so anything after will be printed into the console.
  - viii. The csv string function is very similar to the “<<” and prints out the matrix in the .csv format with “,” in-between all the values.
  - ix. Transpose function is last, it basically flips the rows with the columns.
- d. Constructor.cpp header will be discussed next. Basically, this file contains all the code for the constructors.

```

1  #include "header.h"
2
3  1  mv1::Matrix::Matrix()
4  [ : data_(nullptr), rows_(0), columns_(0) {}
5
6  2  mv1::Matrix::Matrix(std::size_t row, std::size_t col, int fill) {
7  this->rows_ = row;
8  this->columns_ = col;
9  this->data_ = new int* [row];
10
11  for (int i = 0; i < row; i++) {
12  this->data_[i] = new int[col];
13  for (int j = 0; j < col; j++) {
14  this->data_[i][j] = fill;
15  }
16  }
17 }
18
19 3  mv1::Matrix::Matrix(const Matrix& rhs) {
20  this->rows_ = rhs.rows_;
21  this->columns_ = rhs.columns_;
22  this->data_ = new int* [rows_];
23  for (int i = 0; i < rows_; i++) { data_[i] = new int[columns_]; }
24  for (int i = 0; i < rows_; i++) {
25  this->data_[i] = new int[columns_];
26  for (int j = 0; j < columns_; j++) {
27  this->data_[i][j] = rhs.data_[i][j];
28  }
29  }
30 }
31
32 4  mv1::Matrix& mv1::Matrix::operator=(const Matrix& rhs) {
33  if (this != &rhs) {
34  this->rows_ = rhs.rows_;
35  this->columns_ = rhs.columns_;
36  this->data_ = new int* [rows_];
37  for (int i = 0; i < rows_; i++) {
38  this->data_[i] = new int[columns_];
39  for (int j = 0; j < columns_; j++) {
40  this->data_[i][j] = rhs.data_[i][j];
41  }
42  }
43  }
44  return *this;
45 }
46
47 5  mv1::Matrix::~Matrix() {
48  for (int i = 0; i < rows_; i++) { delete[] data_[i]; }
49  delete[] data_;
50 }

```

e.

- i. The first function is known as the default constructor and sets all the default values to zero and the pointer values to "nullptr".

```

3  mv1::Matrix::Matrix()
4  [ : data_(nullptr), rows_(0), columns_(0) {}

```

- 1.
2. The correct syntax for everything is "mv1::Matrix::Matrix()". "Mv1" stands for the namespace everything is in; the first "::Matrix" is for the Matrix class we are calling and the final "::Matrix" is for the constructor that is named the same thing as the class.
3. The three values being set to zero or "nullptr" are the same values in "private:".
4. It is much more efficient to write the code declaring the values as zero using the colon format than declaring them inside the curly brackets. This is known as parameter initialization.
- ii. Next up is the parameterized constructor. The difference is that this time the constructor is being overloaded and has inputs.

```

6  mv1::Matrix::Matrix(std::size_t row, std::size_t col, int fill) {
7      this->rows_ = row;
8      this->columns_ = col;
9      this->data_ = new int* [row];
10
11     for (int i = 0; i < row; i++) {
12         this->data_[i] = new int[col];
13         for (int j = 0; j < col; j++) {
14             this->data_[i][j] = fill;
15         }
16     }
17 }

```

- 1.
2. The constructor has the “row”, “col” and “fill” variables overloaded. “Row” refers to the number of rows in the matrix and is set equal to the initial “rows\_” value using the “this->” pointer. The same is done for the “col” variable as well.
  - a. The “this->” pointer is used to point to the previously declared variables. This allows the code to just use a single instance of the variables.
  - b. Although the “this->” pointer is not explicitly required and is implied, I choose to add it into my code.
3. In line 9, we are declaring “data\_” as a new pointer to the matrix. In the default constructor it was given the “nullptr” status and not linked to a memory address, this time it is given an address to link to.
4. The next part of the code is a nested for loop that loops through the matrix and sets the value equal to the fill passed through the constructor. The first pointer inside the loop declares the columns in the matrix. The second pointer then sets the values equal to fill.
- iii. The next code is the copy constructor. The program uses “.lhs” and “.rhs” to hold two matrices depending on the side of the operation. The initial matrix that is being fed is the “.rhs” or right hand side. Basically, all the values are set equal to the same variable name with “rhs.” in front. This creates a second instance of the variable.

```

19 mv1::Matrix::Matrix(const Matrix& rhs) {
20     this->rows_ = rhs.rows_;
21     this->columns_ = rhs.columns_;
22     this->data_ = new int* [rows_];
23     for (int i = 0; i < rows_; i++) {
24         this->data_[i] = new int[columns_];
25         for (int j = 0; j < columns_; j++) {
26             this->data_[i][j] = rhs.data_[i][j];
27         }
28     }
29 }

```

- 1.
2. The only overload of this function is the “const Matrix& rhs”.
  - a. The “const” means that the value does not change, the “&” after the Matrix means that its referencing the matrix and the “rhs” means that it is the right-hand side matrix.

- b. The following code is identical to the parameterized constructor code except all the variables are being declares as "rhs."
- iv. The next part of code is the copy assignment. This basically sets the "=" key to mean copy in the case there is not a second matrix. If a second matrix does not exist, it will be declared the same as the "rhs".

```

32  mv1::Matrix& mv1::Matrix::operator=(const Matrix& rhs) {
33      if (this != &rhs) {
34          this->rows_ = rhs.rows_;
35          this->columns_ = rhs.columns_;
36          this->data_ = new int* [rows_];
37
38          for (int i = 0; i < rows_; i++) {
39              this->data_[i] = new int[columns_];
40              for (int j = 0; j < columns_; j++) {
41                  this->data_[i][j] = rhs.data_[i][j];
42              }
43          }
44      }
45      return *this;
46  }

```

- 1.
2. Testing is done through the first if statement. If there is no second declaration, then the same for loop sets the matrix to a value.
- v. Lastly is the destructor, it destroys the data\_ matrix variable. Any time a program creates a new address using "new", it must be destroyed using a destructor to save memory.

```

48  mv1::Matrix::~~Matrix() {
49      for (int i = 0; i < rows_; i++) { delete[] data_[i]; }
50      delete[] data_;
51  }

```

- 1.
2. In hindsight, the destructor can be replaced with the clear() function.

```

48  mv1::Matrix::~~Matrix() {
49      clear();
50  }

```

- 3.
- f. The next portion of the code I have named basic. It contains all the functions that are not operators and not constructors. The code is shown below.



```

1  #include "header.h"
2
3  // Element mutator
4  int& mv1::Matrix::at(std::size_t row, std::size_t col) { return data_[row][col]; }
5  // Element accessor
6  int mv1::Matrix::at(std::size_t row, std::size_t col) const { return data_[row][col]; }
7
8  void mv1::Matrix::clear() { // Delete contents
9      for (int i = 0; i < rows_; i++) {
10         delete[] data_[i];
11     }
12     delete[] data_;
13 }
14
15 bool mv1::Matrix::empty() const { // 'true' if matrix contains no elements
16     if (rows_ == 0 && columns_ == 0) return true;
17     else return false;
18 }
19
20 std::size_t mv1::Matrix::rows() const { return rows_; } // Returns the number of rows
21 std::size_t mv1::Matrix::columns() const { return columns_; } // Returns the number of columns
22
23 void mv1::Matrix::reallocate(std::size_t rows, std::size_t cols, int fill) {
24     clear();
25     this->rows_ = rows;
26     this->columns_ = cols;
27     this->data_ = new int*[rows];
28     for (int i = 0; i < rows; i++) {
29         this->data_[i] = new int[cols];
30         for (int j = 0; j < cols; j++) {
31             this->data_[i][j] = fill;
32         }
33     }
34 }

```

- i.
- ii. The first part is the setter and getter or mutator and accessor. The mutator/setter allows the changing of a value while the getter/accessor reads the value. This is the reason why there is an & reference for the mutator/setter because it will actively change the value and return the new one. It is passed by reference and the value changed.

```

3  // Element mutator
4  int& mv1::Matrix::at(std::size_t row, std::size_t col) { return data_[row][col]; }
5  // Element accessor
6  int mv1::Matrix::at(std::size_t row, std::size_t col) const { return data_[row][col]; }
7

```

- 1.
- iii. The second part is the clear() function. This function is identical to the destructor and deletes the matrix. I am unsure if clear() function was intended to be different than the constructor and delete all the values but leave the matrix still existing.

```

8  void mv1::Matrix::clear() { // Delete contents
9      for (int i = 0; i < rows_; i++) {
10         delete[] data_[i];
11     }
12     delete[] data_;
13 }

```

- 1.
2. The current clear() causes a data error when used and is only declared because it is in the header. This is because it causes the value to be deleted twice, once in this function and again in the destructor.
3. The clear function is used in the reallocate function where pre-existing matrices are given a new set of values.
- iv. The bool empty() function checks if the matrix is empty. It does this by checking if the number of rows and columns of the matrix is zero.

```

15 bool mv1::Matrix::empty() const { // 'true' if matrix contains no elements
16     if (rows_ == 0 && columns_ == 0) return true;
17     else return false;
18 }

```

1.

2. It is important to know that empty is different than zero. If something is equal to zero, say a 5x5 matrix filled with zeros, zero is still a value. If something is empty, it means that it has nothing in it and there are no contents, not even zero.
3. Therefore, the code does not check if all the values are zero and immediately checks if the rows and column numbers are zero.
- v. The next functions of rows() and columns() returns the number of rows/columns of the matrix.

```

20     std::size_t mv1::Matrix::rows() const { return rows_; } // Returns the number of rows
21     std::size_t mv1::Matrix::columns() const { return columns_; } // Returns the number of columns

```

- 1.
- vi. The final bit of code is the reallocate function. Here, it deletes a pre-existing matrix and fills it with a new value. It is a combination of the clear function with the code of the parameterized constructor.

```

23     void mv1::Matrix::reallocate(std::size_t rows, std::size_t cols, int fill) {
24         clear();
25         this->rows_ = rows;
26         this->columns_ = cols;
27         this->data_ = new int* [rows];
28         for (int i = 0; i < rows; i++) {
29             this->data_[i] = new int[cols];
30             for (int j = 0; j < cols; j++) {
31                 this->data_[i][j] = fill;
32             }
33         }
34     }

```

- g. The next part of the code that will be covered will be the operators. This represents the compound arithmetic operators, arithmetic operators, and the Boolean operators. The make\_identity and transpose functions will also be included.

- i. The compound operators are shown below.

```

3     mv1::Matrix& mv1::Matrix::operator +=(const Matrix& rhs) { // Matrix addition
4         if (this->rows() != rhs.rows() || this->columns() != rhs.columns()) { return *this; }
5         for (int i = 0; i < rows_; i++) {
6             for (int j = 0; j < columns_; j++) {
7                 this->data_[i][j] += rhs.data_[i][j];
8             }
9         }
10        return *this;
11    }
12
13    mv1::Matrix& mv1::Matrix::operator -=(const Matrix& rhs) { // Matrix subtraction
14        if (this->rows() != rhs.rows() || this->columns() != rhs.columns()) { return *this; }
15        for (int i = 0; i < rows_; i++) {
16            for (int j = 0; j < columns_; j++) {
17                this->data_[i][j] -= rhs.data_[i][j];
18            }
19        }
20        return *this;
21    }
22
23    mv1::Matrix& mv1::Matrix::operator *=(const int rhs) { // Scalar multiplication
24        for (int i = 0; i < rows_; i++) {
25            for (int j = 0; j < columns_; j++) {
26                this->data_[i][j] *= rhs;
27            }
28        }
29        return *this;
30    }

```

- 1.
2. The compound arithmetic operators used in this case are +=, -= and \*=. When the equals sign is added, it means the first value equals itself and the operation is performed on B.
  - a. For example, A+= B means A = A + B.

3. There is an initial if statement that verifies that the two matrices with the operation being performed on it are the same size. If it is not the same size, the initial matrix value is returned.
    - a. I had gotten this section of code from a friend, but it never occurred to me to include this error checking method inside the code. This prevents errors and allows the code to continue running even if there is an issue with the operation.
  4. The code loops through each of the cells of the matrix and performs the operation onto the value.
- ii. This next section returns the identity matrix for any size n by n matrix.

```

32 // Returns an n-by-n identity matrix
33 mv1::Matrix mv1::make_identity(std::size_t n) {
34     Matrix result(n, n);
35     for (int i = 0; i < n; i++) {
36         for (int j = 0; j < n; j++) {
37             if (i == j) { result.data_[i][j] = 1; }
38             else { result.data_[i][j] = 0; }
39         }
40     }
41     return result;
42 }

```

- 1.
2. The following screenshot shows what an identity matrix is.

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$I_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

- a.
  - b. Basically, when the row number is equal to the column number, the value becomes "1". Else, the value is "0".
- iii. The next two functions are the arithmetic addition and subtraction operators.

```

44  mv1::Matrix mv1::operator+(Matrix lhs, const Matrix& rhs) {           // Matrix addition
45      if (lhs.rows() != rhs.rows() || lhs.columns() != rhs.columns()) { return lhs; }
46      Matrix result(rhs.rows_, rhs.columns_);
47      for (int i = 0; i < rhs.rows_; i++) {
48          for (int j = 0; j < rhs.columns_; j++) {
49              result.data_[i][j] = lhs.data_[i][j] + rhs.data_[i][j];
50          }
51      }
52      return result;
53  }
54
55  mv1::Matrix mv1::operator-(Matrix lhs, const Matrix& rhs) {           // Matrix subtraction
56      if (lhs.rows() != rhs.rows() || lhs.columns() != rhs.columns()) { return lhs; }
57      Matrix result(rhs.rows_, rhs.columns_);
58      for (int i = 0; i < rhs.rows_; i++) {
59          for (int j = 0; j < rhs.columns_; j++) {
60              result.data_[i][j] = lhs.data_[i][j] - rhs.data_[i][j];
61          }
62      }
63      return result;
64  }

```

- 1.
  2. The if statement from previous is again used.
  3. Here addition represents the lhs + rhs while subtraction is lhs – rhs.
- iv. The following code will be of the multiplication operator. There are three multiplication operators, two for scalar multiplication and one for matrix multiplication.

```

66  mv1::Matrix mv1::operator*(Matrix lhs, const int rhs) {             // Scalar multiplication
67      Matrix result(lhs.rows_, lhs.columns_);
68      for (int i = 0; i < lhs.rows_; i++) {
69          for (int j = 0; j < lhs.columns_; j++) {
70              result.data_[i][j] = lhs.data_[i][j] * rhs;
71          }
72      }
73      return result;
74  }
75
76  mv1::Matrix mv1::operator*(const int lhs, Matrix rhs) {             // Scalar multiplication
77      Matrix result(rhs.rows_, rhs.columns_);
78      for (int i = 0; i < rhs.rows_; i++) {
79          for (int j = 0; j < rhs.columns_; j++) {
80              result.data_[i][j] = lhs * rhs.data_[i][j];
81          }
82      }
83      return result;
84  }
85
86  mv1::Matrix mv1::operator*(const Matrix& lhs, const Matrix& rhs) { // Matrix multiplication
87      if (lhs.columns() != rhs.rows()) { return Matrix(lhs.rows(), lhs.columns(), 0); }
88      Matrix result(rhs.rows_, rhs.columns_);
89      for (int i = 0; i < rhs.rows_; i++) {
90          for (int j = 0; j < rhs.columns_; j++) {
91              for (int k = 0; k < lhs.columns(); k++) {
92                  result.at(i, j) = result.at(i, j) + (lhs.at(i, k) * rhs.at(k, j));
93              }
94          }
95      }
96      return result;
97  }

```

- 1.
2. Scalar multiplication can take two forms, first is if the matrix is multiplied by the constant. The second is if the constant is multiplied by the matrix in that order.
  - a. In both cases, a new matrix, the result matrix is created and is set equal to the matrix multiplied by the scalar.
3. Matrix multiplication is a totally new way of doing multiplication. The formula is shown in the below screenshot.

## 3 × 3 Matrix Multiplication

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} (aj + bm + cp) & (ak + bn + cq) & (al + bo + cr) \\ (dj + em + fp) & (dk + en + fq) & (dl + eo + fr) \\ (gj + hm + ip) & (gk + hn + iq) & (gl + ho + ir) \end{bmatrix}$$

- a.
  - b. You multiply the first row with the first column and add all the values together. This is shown in the complexity of the next formula.
  - c. For matrix multiplication it is necessary that the number of rows in Matrix A is equal to the number of columns in Matrix B and the number of columns in Matrix A is equal to the number of rows in Matrix B. The if statement first checks this condition before proceeding. If it is not correct, then the operation will not proceed and both matrices are returned with the value of zero. This again prevents errors if the user inputs the wrong value.
- v. This code now shows the two Boolean operators. These are the “==” and “!=” operators. They are used to determine whether two matrices are equal or not.

```

99 bool mv1::operator==(const Matrix& lhs, const Matrix& rhs) {
100     if (lhs.rows() != rhs.rows() || lhs.columns() != rhs.columns()) { return false; }
101     for (int i = 0; i < lhs.rows(); i++) {
102         for (int j = 0; j < lhs.columns(); j++){
103             if (lhs.at(i, j) == rhs.at(i, j)) { return true; }
104         }
105     }
106     return false;
107 }
108 bool mv1::operator!=(const Matrix& lhs, const Matrix& rhs) {
109     if (lhs.rows() != rhs.rows() || lhs.columns() != rhs.columns()) { return true; }
110     for (int i = 0; i < lhs.rows(); i++) {
111         for (int j = 0; j < lhs.columns(); j++){
112             if (lhs.at(i, j) != rhs.at(i, j)) { return true; }
113         }
114     }
115     return false;
116 }

```

- 1.
  2. The first part of these two functions is the if statement. It checks to see if the two matrices are of equal size. If the matrices are not the same size, the second part of the function no longer needs to be run because they are not equal. This saves on memory and makes the script much more efficient.
  3. If the matrices are the same size, it will run through each cell to check if the values are the same. If they are identical, it will return true for “==” and false for “!=”.
- vi. This next function is the “<<” operator which represents system output. Uses the ostream library and prints out the matrix. The csv function then allows the user to print out the matrix in a .csv format with commas separating each value.

```

118 std::ostream& mv1::operator<<(std::ostream& os, const Matrix& rhs) {
119     for (int i = 0; i < rhs.rows_; i++) {
120         for (int j = 0; j < rhs.columns_; j++) {
121             os << rhs.data_[i][j] << " ";
122         }
123         os << std::endl;
124     }
125     return os;
126 }
127
128 // Returns matrix values in comma separated value (CSV) format
129 std::string mv1::csv(const Matrix& mat) {
130     std::string output = "";
131     for (int i = 0; i < mat.rows(); i++) {
132         for (int j = 0; j < mat.columns(); j++) {
133             output += std::to_string(mat.at(i, j)) + ",";
134         }
135         output += "\n";
136     }
137     return output;
138 }

```

- 1.
  2. Both functions share a similar format with the nested for loops, the primary difference is that the csv function prints out commas after each output.
- vii. Transpose function is then shown. The transpose basically flips the rows with the columns. This is done by running through the matrix and calling the value using the accessor/getter and setting that equal to the new value.

```

140 mv1::Matrix mv1::transpose(const Matrix& mat) {
141     Matrix result(mat.columns(), mat.rows());
142     for (int i = 0; i < result.rows(); i++) {
143         for (int j = 0; j < result.columns(); j++) {
144             result.at(i, j) = mat.at(j, i);
145         }
146     }
147     return result;

```

- 1.
- h. The main function has several parts, it first creates a matrix and fills it with unique values, then it checks if its empty and prints it in .csv format. It also transposes the matrix. After that it runs through the matrix operations and verifies that the compound arithmetic operators are equal to the arithmetic operators.
- i. This is the first part of the main function. It creates a new function and replaces all the values inside with unique values. It also prints out the matrix in .csv format using commas and returns the number of rows/columns. It uses the setter/mutator to change individual values and set them to the new values after the initial matrix is initialized.

```

4 int main() {
5     std::cout << "Print a 3x3 matrix filled with 1: " << std::endl;
6     mv1::Matrix m0(3, 3, 1);
7     std::cout << m0 << std::endl;
8     m0.at(0, 0) = 1;
9     m0.at(0, 1) = 9;
10    m0.at(0, 2) = 7;
11
12    m0.at(1, 0) = 5;
13    m0.at(1, 1) = 2;
14    m0.at(1, 2) = 6;
15
16    m0.at(2, 0) = 1;
17    m0.at(2, 1) = 4;
18    m0.at(2, 2) = 8;
19
20    std::cout << "Modifying the matrix to have unique values: " << std::endl;
21    std::cout << m0 << std::endl;
22
23    std::cout << "Printing out matrix in .csv format: " << std::endl;
24    std::cout << csv(m0) << std::endl;
25
26    std::cout << "Checking if the matrix is empty: " << m0.empty() << std::endl;
27    std::cout << "Matrix row count: " << m0.rows() << std::endl;
28    std::cout << "Matrix column count: " << m0.columns() << std::endl;
29    std::cout << std::endl;

```

- 1.
- ii. The second part of the main function creates an identity matrix of size 4. It calls the "make\_identity" function.

```

31     std::cout << "Identity Matrix Size 4: " << std::endl;
32     std::cout << mv1::make_identity(4) << std::endl;

```

- 1.
- iii. After that, the third part has it creating a new matrix called transpose\_matrix that will get transposed. The original matrix and the transpose matrix are both printed out to show the difference.

```

34     std::cout << "The following matrix will get transposed: " << std::endl;
35     mv1::Matrix transpose_Matrix(4, 2, 0);
36     std::cout << transpose_Matrix << std::endl;
37
38     std::cout << "Transposed Matrix: " << std::endl;
39     std::cout << mv1::transpose(transpose_Matrix);
40     std::cout << std::endl;

```

- 1.
- iv. This next part is used to perform the arithmetic operations. I use both the arithmetic operator and compound operator to verify that the values are identical.

```

42     mv1::Matrix m1(3, 3, 9);
43     mv1::Matrix m2(3, 3, 7);
44
45     std::cout << "Matrix 1: " << std::endl;
46     std::cout << m1 << std::endl;
47     std::cout << "Matrix 2: " << std::endl;
48     std::cout << m2 << std::endl;
49     std::cout << "Matrix Addition: " << std::endl;
50     std::cout << "Matrix 1 + Matrix 2: " << std::endl << m1 + m2 << std::endl;
51     m1 += m2;
52     std::cout << "Matrix 1 += Matrix 2: " << std::endl << m1 << std::endl;
53     std::cout << "The + and += operators are identical." << std::endl;
54     std::cout << std::endl;
55
56     m1.reallocate(3, 3, 9);
57     m2.reallocate(3, 3, 7);
58
59     std::cout << "Matrix Subtraction: " << std::endl;
60     std::cout << "Matrix 1 - Matrix 2: " << std::endl << m1 - m2 << std::endl;
61     m1 -= m2;
62     std::cout << "Matrix 1 -= Matrix 2: " << std::endl << m1 << std::endl;
63     std::cout << "The - and -= operators are identical." << std::endl;
64     std::cout << std::endl;

```

1.



2. The `realloc` function is being used to change the value back to the originally intended declaration of the matrix.
- v. After addition and subtraction, we next perform multiplication. It is important to note that both matrix multiplication and scalar multiplication are performed. The Boolean operations are performed after and verify that the two matrices are indeed not equal to each other.

```

66         m1.reallocate(3, 3, 9);
67         m2.reallocate(3, 3, 7);
68
69         std::cout << "Matrix Multiplication: " << std::endl;
70         std::cout << "Matrix 1 * Matrix 2: " << std::endl << m2 * m1 << std::endl;
71         std::cout << std::endl;
72
73         std::cout << "Scalar Multiplication: " << std::endl;
74         std::cout << "Matrix 1 * 6: " << std::endl << m1 * 6 << std::endl;
75         m1 *= 6;
76         std::cout << "Matrix 1 *= 6: " << std::endl << m1 << std::endl;
77         std::cout << "The * and *= operators are identical." << std::endl;
78         std::cout << std::endl;
79         std::cout << "9 * Matrix 2: " << std::endl << 9 * m2 << std::endl;
80         std::cout << std::endl;
81
82         std::cout << "Matrix 1 == Matrix 2: " << (m1 == m2) << std::endl;
83         std::cout << "Matrix 1 != Matrix 2: " << (m1 != m2) << std::endl;
84         std::cout << std::endl;

```

- 1.
2. All operations are shown, and the values are deemed to be correct.
- vi. The final part of the function allows the user to input what type of matrix they want. It starts off by asking the user to input the dimensions and fill of a matrix. It then prints out this selected matrix. After that, it allows the user to generate an identity matrix of pre-specified size.

```

86         int rows_, columns_, fill;
87         std::cout << "The following allows the user to generate a matrix with a specified fill." << std::endl;
88         std::cout << "Enter number of rows: ";
89         std::cin >> rows_;
90         std::cout << "Enter the number of columns: ";
91         std::cin >> columns_;
92         std::cout << "Enter the fill of the matrix: ";
93         std::cin >> fill;
94         mv1::Matrix user_Matrix(rows_, columns_, fill);
95         std::cout << user_Matrix << std::endl;
96
97         std::cout << "The following allows the user to generate an identity matrix of pre-specified size." << std::endl;
98         std::cin >> rows_;
99         std::cout << mv1::make_identity(10);
100    }

```

- 1.
- 2.
3. This works by using "cin" and asking the user for a data input.
4. There is a small change made to the final line, instead of inputting 10, it will now input the `rows_` variable which is the user input.
- i. This is all the code explained, in the results section, the output to the console will be discussed.
4. Results:
  - a. This next section will go through the console outputs and user input sections.
  - b. This is the first part of the code. All outputs are as expected.



```

Print a 3x3 matrix filled with 1:
1 1 1
1 1 1
1 1 1

Modifying the matrix to have unique values:
1 9 7
5 2 6
1 4 8

Printing out matrix in .csv format:
1,9,7,
5,2,6,
1,4,8,

Checking if the matrix is empty: 0
Matrix row count: 3
Matrix column count: 3

```

i.

c. This next part produces an identity matrix and performs the transpose function.

```

Identity Matrix Size 4:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

The following matrix will get transposed:
0 0
0 0
0 0
0 0

Transposed Matrix:
0 0 0 0
0 0 0 0

```

i.

d. Here, we perform the arithmetic and compound arithmetic operators.

```

Matrix 1:
9 9 9
9 9 9
9 9 9

Matrix 2:
7 7 7
7 7 7
7 7 7

Matrix Addition:
Matrix 1 + Matrix 2:
16 16 16
16 16 16
16 16 16

Matrix 1 += Matrix 2:
16 16 16
16 16 16
16 16 16

The + and += operators are identical.

Matrix Subtraction:
Matrix 1 - Matrix 2:
2 2 2
2 2 2
2 2 2

Matrix 1 -= Matrix 2:
2 2 2
2 2 2
2 2 2

The - and -= operators are identical.

```

i.

- ii. All values are identical as intended.
- e. Matrix multiplication is shown. The same matrix as before will be used for this calculation.

```
Matrix Multiplication:
Matrix 1 * Matrix 2:
189 189 189
189 189 189
189 189 189

Scalar Multiplication:
Matrix 1 * 6:
54 54 54
54 54 54
54 54 54

Matrix 1 *= 6:
54 54 54
54 54 54
54 54 54

The * and *= operators are identical.

9 * Matrix 2:
63 63 63
63 63 63
63 63 63

Matrix 1 == Matrix 2: 0
Matrix 1 != Matrix 2: 1
```

- i.
- ii. All values are identical and calculated correctly.
- iii. The matrices are not equal, which is why the “==” operation has a 0 marking it as false while “!=” is marked as 1 making it true that they are not equal.
- f. This final section asks the user to input the parameters to generate a matrix.

```
The following allows the user to generate a matrix with a specified fill.
Enter number of rows: 5
Enter the number of columns: 5
Enter the fill of the matrix: 8
8 8 8 8 8
8 8 8 8 8
8 8 8 8 8
8 8 8 8 8
8 8 8 8 8
```

- i.
- ii. In this case, I am generating a 5x5 matrix filled with 8. The program does what is intended to.
- g. Lastly, it asks the user to input a number to generate an identity matrix off of.

```
The following allows the user to generate an identity matrix of pre-specified size.
7
1 0 0 0 0 0 0
0 1 0 0 0 0 0
0 0 1 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 1
```

- i.
- ii. Here a size 7 identity matrix is generated.

- h. The console output is as intended and there appear to be no errors.
5. Discussion/Conclusions:
- a. This project effectively taught me how to use classes, pointers, and operators. I learned how to use multiple constructors and overloading operators/constructors.
  - b. The most difficult part of this project was getting the constructors working the understanding the correct way to use pointers and references. I was confused initially about how to properly use pointers to allocate memory and declare variables.
    - i. The “new” keyword was initially very confusing to me, but I was able to correctly figure out the implementation.
  - c. I also learned about the nuances of class/function implementation. Some overloaded functions/operators cannot be included inside the class because there are too many parameters.
  - d. Although the majority of the header was commented with instructions, there were some parts such as “reallocate” which was not given much instruction. I was confused on the proper implementation of many of the functions and only realized after the fact when writing this report.
  - e. Another aspect that confused me was the “correct” implementation of the constructor. I know some people used two for loops. I am still confused on which loop format is preferred.
  - f. In the end, I was able to successfully complete the project all the while learning the correct implementations. I believe I have met all objectives for this project.