OOP
Lecture 13

18/11/2021

Class Activity :

(Missed!! ? Don't panic… It wasn't graded 😋)

Write the following functions:

```
class Matrix{
        int **elements, rows, columns;

        Matrix(int r, int c) // Initialize 2D array with random values
        Matrix( const Matrix &m)
        ~Matrix()

}
```

Solution :

First, we will check what we wrote in the class and then we will compare it with the official solution.

It should be clear till now that the first function is the parameterized constructor, second is the copy constructor, and third is the destructor.

```
class Matrix{
        int **elements, rows, cols;

  public:
                // Parameterized Constructor
                Matrix(int r, int c)
                {
                        int i, j;
                        if ( i <= 0 || c <= 0 )
                                return;
        rows = r;
        cols = c;

        elements = new int*[rows];
        for ( i = 0; i < rows; i++ )
        {
                elements[i] = new int[cols];
                for ( j = 0; j < cols; j++ )
                        elements[i][j] = rand() % 9 + 1;
        }
}

// Copy Constructor
Matrix(const Matrix &m)
{
        int i, j;
        rows = m.rows;
        cols = m.cols;

        elements = new int*[rows];
        for ( i = 0; i < rows; i++ )
        {
                elements[i] = new int[cols];
                for ( j = 0; j < cols; j++ )
                        elements[i][j] = m.elements[i][j];
        }
}

// Destructor
~Matrix()
{
        int i;
        for ( i = 0; i < rows; i++ )
                delete[] elements[i];
```

```
                    delete[] elements;
            }
    };
```

Here, we can see that we are repeating our code in parameterized and copy constructor, and similarly if we extend this program to overload the assignment (=) operator, we need to write this code again and even in assignment operator we need to delete the previous memory as well.

Just a quick question.
Why do we delete the previous memory in the assignment operator?
We delete the previous memory in the assignment operator because the new size of the array which is to be copied in the previous array may be different. Suppose we are copying an object of matrix 3x3 into an existing object whose size is 2x2. In this case, our new matrix can't fit in the existing object. So, we first delete the previous memory and then we allocate new memory of the new size and then make a copy.

So, we were discussing the duplication of code in the constructors. The solution to avoid duplication in our code is that we declare the functions with required code and call that function again and again. So, in the previous example we can write functions to allocateMemory(); deallocateMemory(); and copyMatrix(); to make our code more efficient and readable.

Official Solution :

```
class Matrix{

    int **elements, rows, cols;
```

```
        void allocateMemory(int r, int c)
        {
                elements = new int*[r];
                        for ( int i = 0; i < r; i++ )
                                elements[i] = new int[c];
        }

        void copyMatrix(const Matrix &m)
        {
                int i, j;
                for ( i = 0; i < rows; i++ )
                        for ( j = 0; j < cols; j++ )
                                elements[i][j] = m.elements[i][j];
        }

        void deallocateMemory(void)
        {
                int i;
                for ( i = 0; i < rows; i++ )
                        delete[] elements[i];

                delete[] elements;
        }

public:
                // Parameterized Constructor
                Matrix(int r, int c)
                {
                        int i, j;
                        if ( i <= 0 || c <= 0 )
                                return;

                        rows = r;
                        cols = c;
```

```
            allocateMemory(rows, cols);

            for ( i = 0; i < rows; i++ )
                  for ( j = 0; j < cols; j++ )
                        elements[i][j] = rand() % 9 + 1;
      }

      // Copy Constructor
      Matrix(const Matrix &m)
      {
            int i, j;

            rows = m.rows;
            cols = m.cols;

            allocateMemory(rows, cols);
            copyMatrix(m);
      }

      void operator = (const Matrix &m)
      {
            deallocateMemory();
            allocateMemory(m.rows, m.cols);
            copyMatrix(m);
      }

      // Destructor
      ~Matrix()
      {
            deallocateMemory();
      }
};
```

In this way, we can avoid duplication of code in our program.

Printing Our Objects using cout :

Now we are going to start a very important concept in C++ i.e printing the objects of type class by using cout.

Before starting, we need to clear the difference between printf( ) and cout. We know that anything which is used with parenthesis ( ) is a function. So, it is clear that printf ( ) is a function which is defined in header files iostream.h and also stdio.h. Then, what is cout? Actually, cout is the object of a class. The cout is an object of class ostream. It is defined in the iostream.h header file.

As we know that, normally we print anything using cout like this,
cout << x; or. **cout << “GoodBye World!”;**

The data needed to be displayed on the screen is written at the write side of cout after the insertion operator (<<).

Here, one thing is to be clear that (<<) is a **binary operator.** It means that it will take two operands and work on them. It implies that one operand on the left is cout (which is an object of ostream class) and the second operand can be anything i.e it can be a variable or string or another object of another class(which is to be printed). So, it is clear that we need to overload the insertion operator (<<) in our class.

But, if we go back in our memory, we can remember that the operand on the left side of the operator is the calling object and it will be passed as the current object to the function. But, we know that cout is the object of the ostream class so it will call the function from the ostream class. If we want to call the insertion operator function from our own class then we need to write a friend function of our class.

Friend function:

Friend functions are the global functions which are declared “friend” by the class. These functions can access the private data members of the class. We can declare the friend function by writing the **friend** keyword with the function name.

Overloading (<<) operator to print a matrix :
Code :

```
friend ostream& operator << (ostream &out, const Matrix &m)
{
        int i, j;
        for (i=0;i<m.rows;i++)
        {
                for (j=0;j<m.cols;j++)
                        out << m.elements[i][j] << ' ';
                out << '\n';
        }
        return out;
}
```

Explanation :

- Here, first we wrote the friend keyword to declare this function as a friend of this class.
- Then we wrote the return type of the function which is ostream. Ostream is a class defined in iostream.h, which contains the definitions of input/output functions and objects. Cout is also an object of the class. Here, the same concept is used when we were using our own class name as the return type of the function. Now we are using a built-in class name as a return type. It shows that we are returning an object of ostream class.
- The reference operator (&) is used with the return type of the class to show that we are returning the same object which is received i.e no new object is returned.
- Then, we wrote the **operator** keyword and the operator symbol (<<) to show that we are overloading the insertion operator (<<).
- Then we wrote the parameters of the function. As (<<) is a binary operator it will take 2 operands. Here, confusion may arise that when

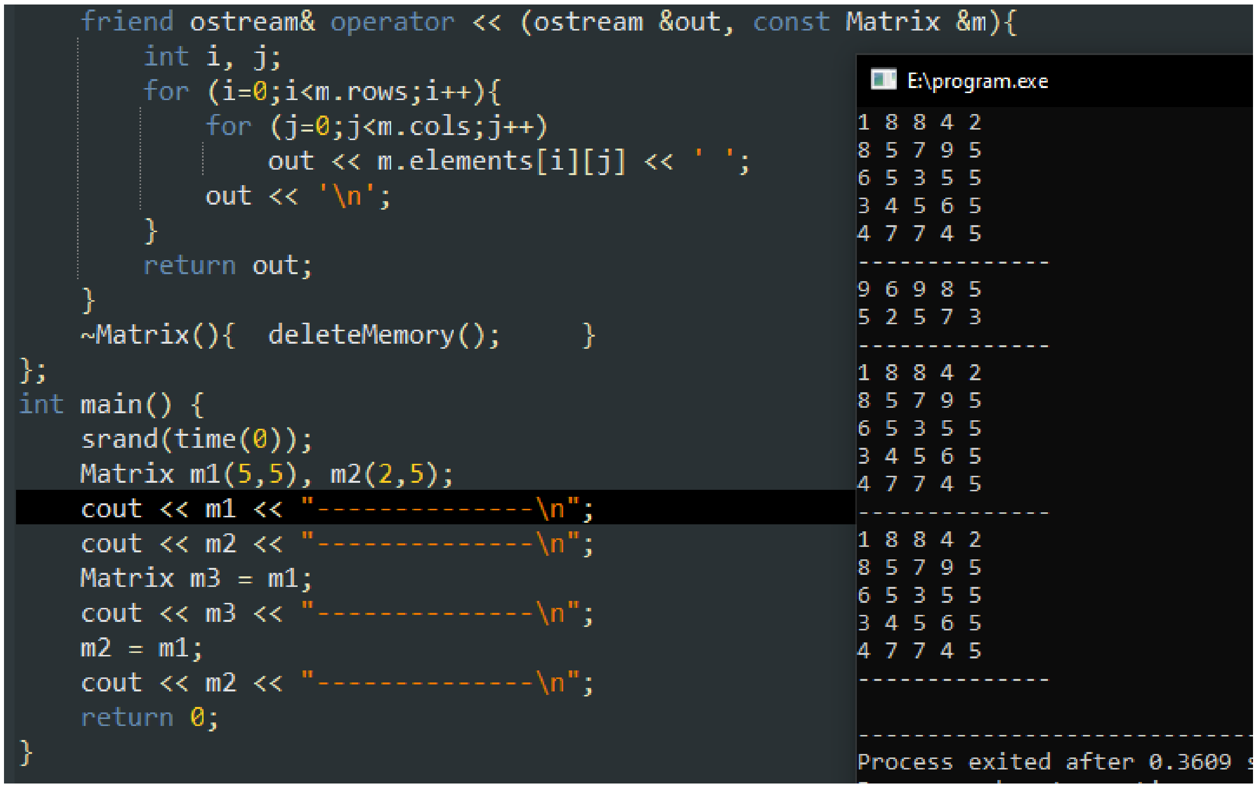
we overloaded the + and - operators we took only 1 parameter and one was the current/caller object. But, now the case is different, when we declared this function as a friend it means that it is a global function which can access the data members of the class but it is not the part of the class. So, no current object will be passed to this function. So, the first parameter is **out,** which should be an object of the ostream class, here **out** is an alias of the **cout,** used in the main function. The second parameter is **m** which should be an object of the class matrix, in this case it will be an alias of the matrix which is to be printed.

- Then we wrote our own logic to decide how our matric should be printed on the screen.
- Notice that now we used “**out <<**“ instead of cout because now we know that out is also an alias of the cout object or any object of class ostream which can be passed from the main function.
- At the end we returned the out object as mentioned in the prototype that we are returning an existing object of class ostream. The benefit of returning the out object is that we can use the multiple insertion operators in one line.
- Now we can simply print our matrix from the main function by writing cout<<matrix; where matrix is the object of the Matric class.

Suppose, cout << m1 << m2 << “END”;

Here, the operator is used 3 times, When first 2 operands (cout<<m1)acts these two will be passed to the friend function, and the m1 matrix will be printed and the function will again return the cout object so this statement will be like cout<<m2<<”END”; similarly, cout and m2 will be passed to the friend function and after printing m2, cout will be returned again and this statement will be like cout<<”END”; now, the next operand is string and we didn’t overload the (<<) operator for a string so, in this case the operator from the ostream will be called the string will be printed on the console.

Sample Program :



```
    friend ostream& operator << (ostream &out, const Matrix &m){
        int i, j;
        for (i=0;i<m.rows;i++){
            for (j=0;j<m.cols;j++)
                out << m.elements[i][j] << ' ';
            out << '\n';
        }
        return out;
    }
    ~Matrix(){  deleteMemory();     }
};
int main() {
    srand(time(0));
    Matrix m1(5,5), m2(2,5);
    cout << m1 << "--------------\n";
    cout << m2 << "--------------\n";
    Matrix m3 = m1;
    cout << m3 << "--------------\n";
    m2 = m1;
    cout << m2 << "--------------\n";
    return 0;
}
```

E:\program.exe

```
1 8 8 4 2
8 5 7 9 5
6 5 3 5 5
3 4 5 6 5
4 7 7 4 5
--------------
9 6 9 8 5
5 2 5 7 3
--------------
1 8 8 4 2
8 5 7 9 5
6 5 3 5 5
3 4 5 6 5
4 7 7 4 5
--------------
1 8 8 4 2
8 5 7 9 5
6 5 3 5 5
3 4 5 6 5
4 7 7 4 5
--------------

--------------------------
Process exited after 0.3609 s
```

Overloading Globally :

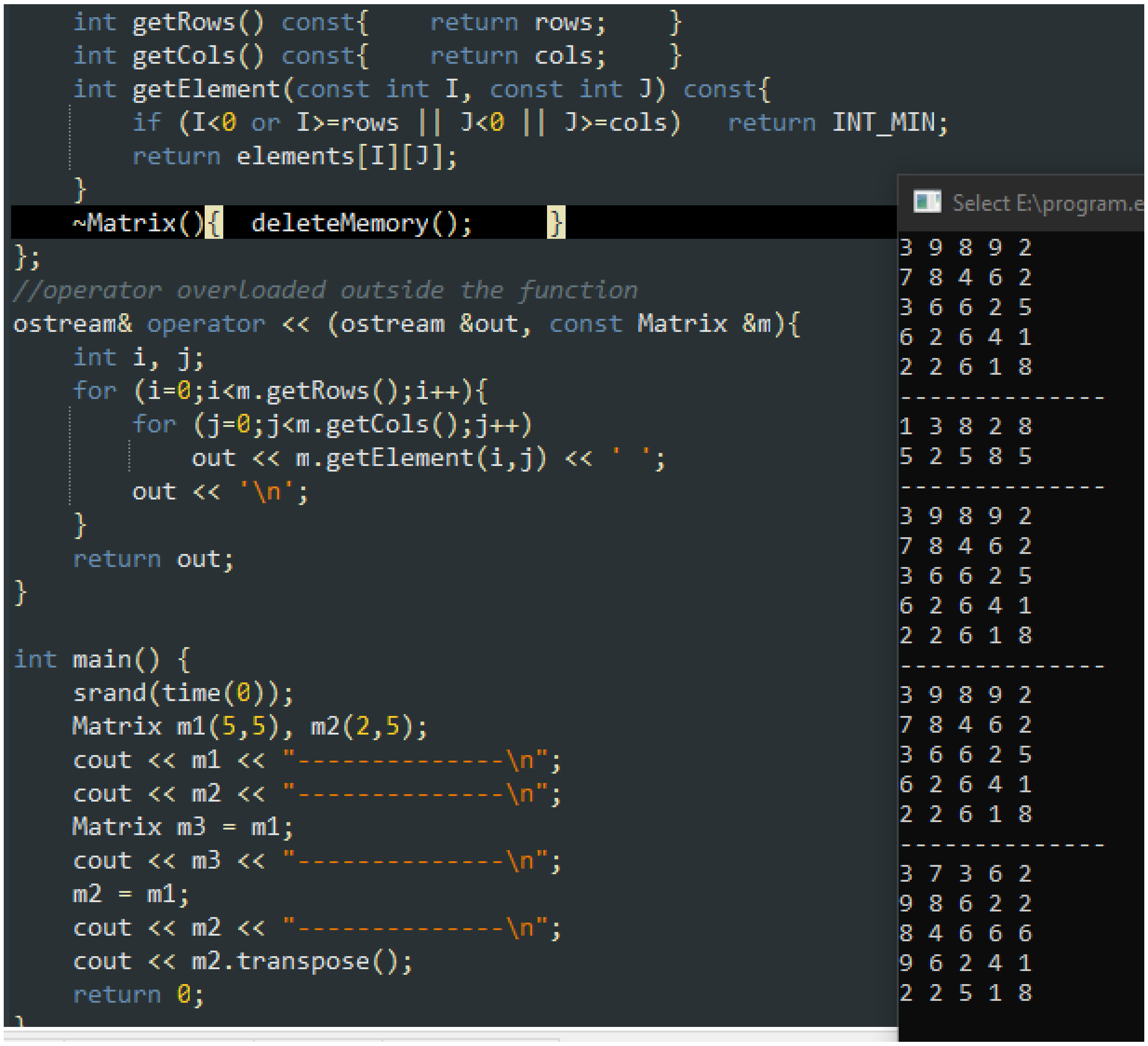
Defining a function globally means that we are defining the function outside the class. The operator overloading can also be done globally. The main points to define a function globally are very important. The most important of which is how we are going to access the data members of the class. There are three ways to do this:

1. By defining all the data members of the class as public.
2. By using the getter functions to return the data members of the class.
3. By declaring the function as a friend of the class.

1. The point 1 is simple i.e if we define the data members of the class as public i.e if we used the public keyword before defining the data members of the class, then the data members will be defined as the public and can be accessed from anywhere in the program.

2. By using the getter the functions we can return the data members of the class, in order to implement our logic.

Code :



```
    int getRows() const{    return rows;    }
    int getCols() const{    return cols;    }
    int getElement(const int I, const int J) const{
        if (I<0 or I>=rows || J<0 || J>=cols)   return INT_MIN;
        return elements[I][J];
    }
    ~Matrix(){  deleteMemory();     }
};
//operator overloaded outside the function
ostream& operator << (ostream &out, const Matrix &m){
    int i, j;
    for (i=0;i<m.getRows();i++){
        for (j=0;j<m.getCols();j++)
            out << m.getElement(i,j) << ' ';
        out << '\n';
    }
    return out;
}

int main() {
    srand(time(0));
    Matrix m1(5,5), m2(2,5);
    cout << m1 << "--------------\n";
    cout << m2 << "--------------\n";
    Matrix m3 = m1;
    cout << m3 << "--------------\n";
    m2 = m1;
    cout << m2 << "--------------\n";
    cout << m2.transpose();
    return 0;
```

Select E:\program.e

```
3 9 8 9 2
7 8 4 6 2
3 6 6 2 5
6 2 6 4 1
2 2 6 1 8
--------------
1 3 8 2 8
5 2 5 8 5
--------------
3 9 8 9 2
7 8 4 6 2
3 6 6 2 5
6 2 6 4 1
2 2 6 1 8
--------------
3 9 8 9 2
7 8 4 6 2
3 6 6 2 5
6 2 6 4 1
2 2 6 1 8
--------------
3 7 3 6 2
9 8 6 2 2
8 4 6 6 6
9 6 2 4 1
2 2 5 1 8
```

Overloading the + operator globally, just to make us clear about out concept clearer about the overloading.

```
#include <iostream>
using namespace std;

class A
{
        int x;
public:
        A(const int X)
{
                this->x = X;
        }
        int getX() const { return x;      }
};
class B
{
        float f;
public:
        B(const float F)
        {
                this->f = F;
        }
        float getF() const { return f;    }
};
//operator overloaded outside the function
void operator + (const A &objA, const B &objB)
{
        cout << objA.getX() + objB.getF() << '\n';
}

int main()
{
        A objectA(3);
        B objectB(4.5);
        objectA + objectB;
        return 0;
}
```

Writing in Files :

An important concept to overload the insertion operator (<<) is that with this we can also write in the files. Suppose an object is created in the main function of the ostream class and a file is opened into it, it means that when we call the inserting operator function by simply writing the object_name << class_object; like, if we want to print our matrix object into a file than we will open the file into an object like this,

```
ostream outputFile("file.txt");
```

And then, whenever we call the insertion operator function like this,

```
outputFile << m2;
```

Then, it will call the friend function of the class which we have overloaded, because the outputFile is also an object of the ostream class just like cout. The only difference is that now when we try to print our matrix then our matrix will be printed into the file instead of the console/screen.

Initializing Lists :

Initializer List is used in initializing the data members of a class. The list of members to be initialized is indicated with the constructor as a comma-separated list followed by a colon. The benefit of using the initializer list is that with this we can also initialize the constant variables (like const int variables) of the class.

Syntax :

```
class class_name{
        ------------
        const int size;
  Public:
```

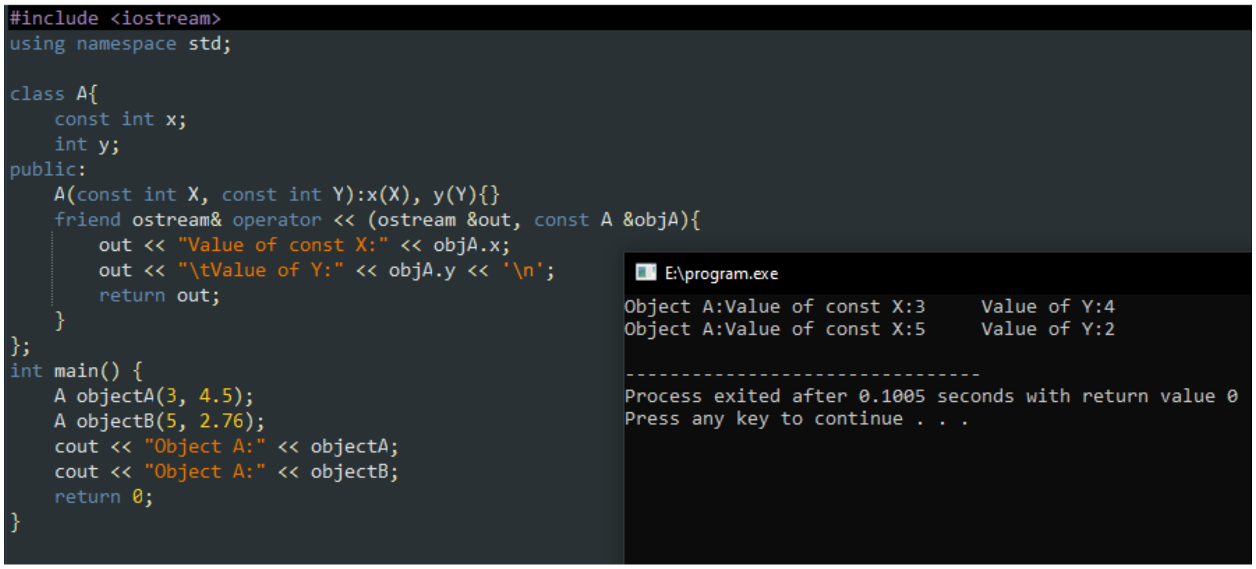
```
                //Constructor
                class_name(   ) : variable_name(value)
                {
                }

        };
```

Example :

```
        class A
{
        const int x;
        int y;
public:
        A(const int X, const int Y): x(X),  y(Y)
        {
        }
};
```

We can initialize both const and normal variables here, but we cannot initialize const variables anywhere. We can only use initializer lists and this list should be i n front of the constructors only. Otherwise it will give an error.

Sample Program :



```cpp
#include <iostream>
using namespace std;

class A{
    const int x;
    int y;
public:
    A(const int X, const int Y):x(X), y(Y){}
    friend ostream& operator << (ostream &out, const A &objA){
        out << "Value of const X:" << objA.x;
        out << "\tValue of Y:" << objA.y << '\n';
        return out;
    }
};
int main() {
    A objectA(3, 4.5);
    A objectB(5, 2.76);
    cout << "Object A:" << objectA;
    cout << "Object A:" << objectB;
    return 0;
}
```

E:\program.exe

```
Object A:Value of const X:3     Value of Y:4
Object A:Value of const X:5     Value of Y:2

--------------------------------
Process exited after 0.1005 seconds with return value 0
Press any key to continue . . .
```

--*THE END*--