OOP
Lecture 11

11/11/2021

Quiz 4 :

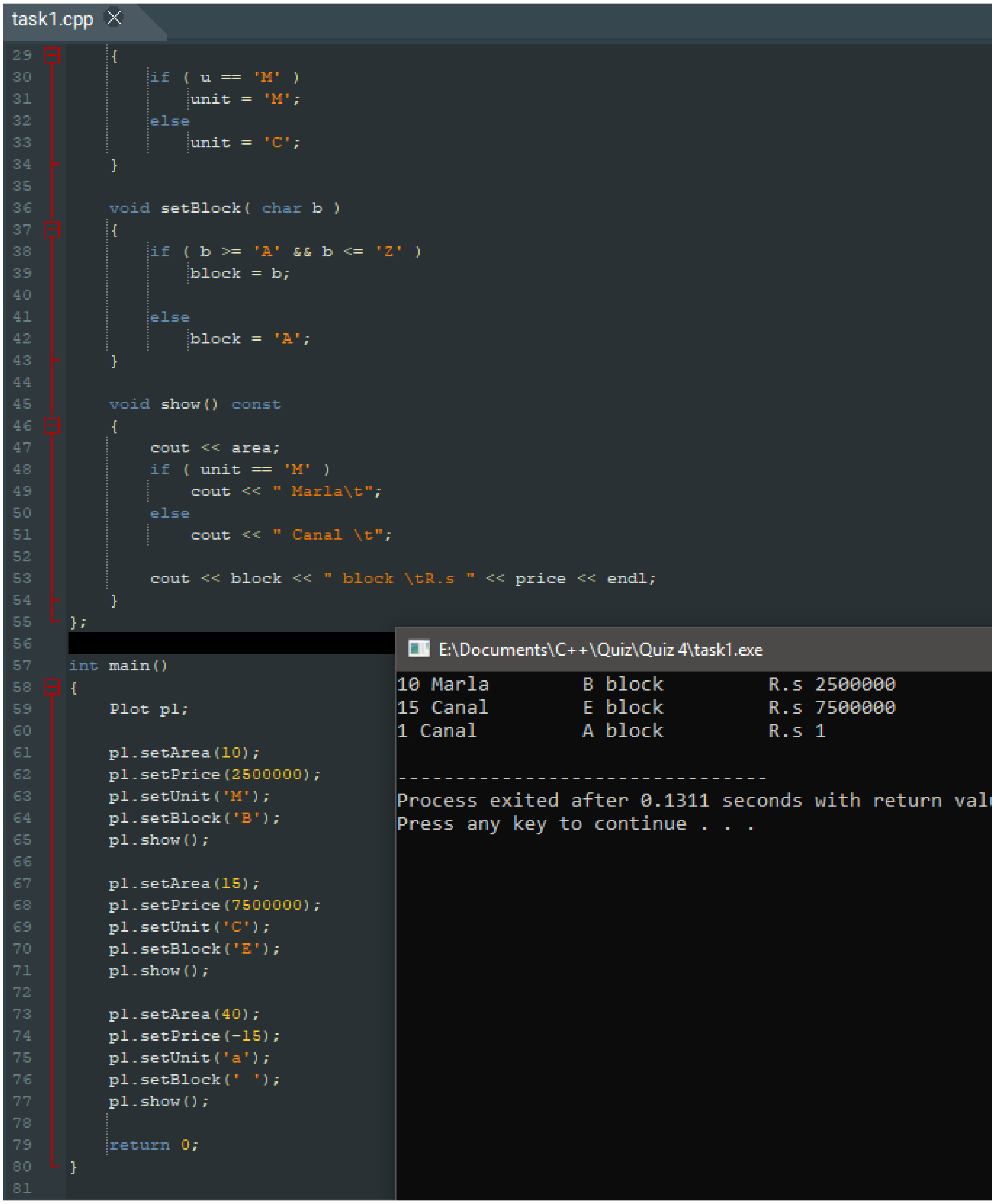
1. Write Class **PLOT** with data members Area[int], Unit[char], Block[char], Price[int]. Area should be greater than zero & less than 20. Unit Should be ‘M’ (for marla) or ‘C’ (for Canal). Block should be ‘A’ to ‘Z’. Price should be greater than 0. Write four setter Functions according to requirements and a show function to print output:

```
10 Marla    B Block     Rs. 2500000
1 Canal     F Block     Rs. 7500000
```

Solution :

```
class Plot{
        int area;
        int price;
        char unit;
        char block;

  public:
        void setArea( int a )
        {
                if ( a < 1 || a > 19 )
                        area = 1;
                else
                        area = a;
        }

        void setPrice( int p )
        {
                if ( p > 0 )
                        price = p;
                else
                        price = 1;
        }

        void setUnit( char u )
        {
                if ( u == 'M' )
                        unit = 'M';
                else
                        unit = 'C';
        }

        void setBlock( char b )
        {
                if ( b >= 'A' && b <= 'Z' )
                        block = b;

                else
                        block = 'A';
        }

        void show() const
        {
                cout << area;
                if ( unit == 'M' )
                        cout << " Marla\t";
                else
                        cout << " Canal \t";

                cout << block << " block \tR.s " << price << endl;
        }
};
```

Output with main() Function :


task1.cpp

```cpp
    {
        if ( u == 'M' )
            unit = 'M';
        else
            unit = 'C';
    }

    void setBlock( char b )
    {
        if ( b >= 'A' && b <= 'Z' )
            block = b;

        else
            block = 'A';
    }

    void show() const
    {
        cout << area;
        if ( unit == 'M' )
            cout << " Marla\t";
        else
            cout << " Canal \t";

        cout << block << " block \tR.s " << price << endl;
    }
};

int main()
{
    Plot pl;

    pl.setArea(10);
    pl.setPrice(2500000);
    pl.setUnit('M');
    pl.setBlock('B');
    pl.show();

    pl.setArea(15);
    pl.setPrice(7500000);
    pl.setUnit('C');
    pl.setBlock('E');
    pl.show();

    pl.setArea(40);
    pl.setPrice(-15);
    pl.setUnit('a');
    pl.setBlock(' ');
    pl.show();

    return 0;
}
```

E:\Documents\C++\Quiz\Quiz 4\task1.exe

```
10 Marla        B block         R.s 2500000
15 Canal        E block         R.s 7500000
1 Canal         A block         R.s 1

--------------------------------
Process exited after 0.1311 seconds with return val
Press any key to continue . . .
```

Important Points and Common Mistakes:

1. First of all, **Do only what is asked** in the question. Don’t do anything extra. Like in this question, we are asked to write data members, 4 setters and 1 show function. Don’t need to write any extra code like constructors or destructors. We don't even need to write the main function until unless we are specifically asked for it.

2. **Follow the data types** of the data members and use logic to get the appropriate results.

3. **Follow the specified range** of each data member and the program should avoid invalid values. If the default values are then we are bound to that values in case if the user passes any invalid value. But, if we are not given any **default value**, like in this case, we should set the default value in the specified range. Our program should not exceed from the mentioned range. For example, in this program the range of area is 1 to 19. So, if the user passes any other value out of the range, our program can assign any value in this range (even 0 is not in this range). (However, it is just a convention that we assign the smallest or largest value, otherwise it is totally your choice).

4. We should **strictly** follow the output format of the code (In case output is given in the program). Like in this case, there is a tab space between each quantity, and 1 space between the unit and quantity so we should apply appropriate space and \t to match the output.

Task 2: Incomplete Class Rope and main function is written. Write member functions of the rope class used in the main function. For show function see output at the end:

```
Class Rope{
        int feet; //greater than 0
         int inches; // greater than 0 and less than 12, if greater or equal to 12,
     add 1 to feet and place remaining in inches, like for 15 add 1 to feet and
                                                                    store 3 to inches.



……………………
};

int main()
{
        Rope r1, r2(5, 8);
        r1.setFeet(3);
        r1.setInches(14);
        r2.getInches();
        r1.show();

        return 0;
}
```

Solution :

Remember the Important points of the previous task?? 👦

Referring to the first point, **“Do only what is asked”,** we are asked to write only those functions which are called from the main function. Till now, we successfully understood the functionality of the setter and getter functions and their input and return types. So, if we count, we are asked to write 6 functions in this main function (as a total of six functions are called from the main). These are:

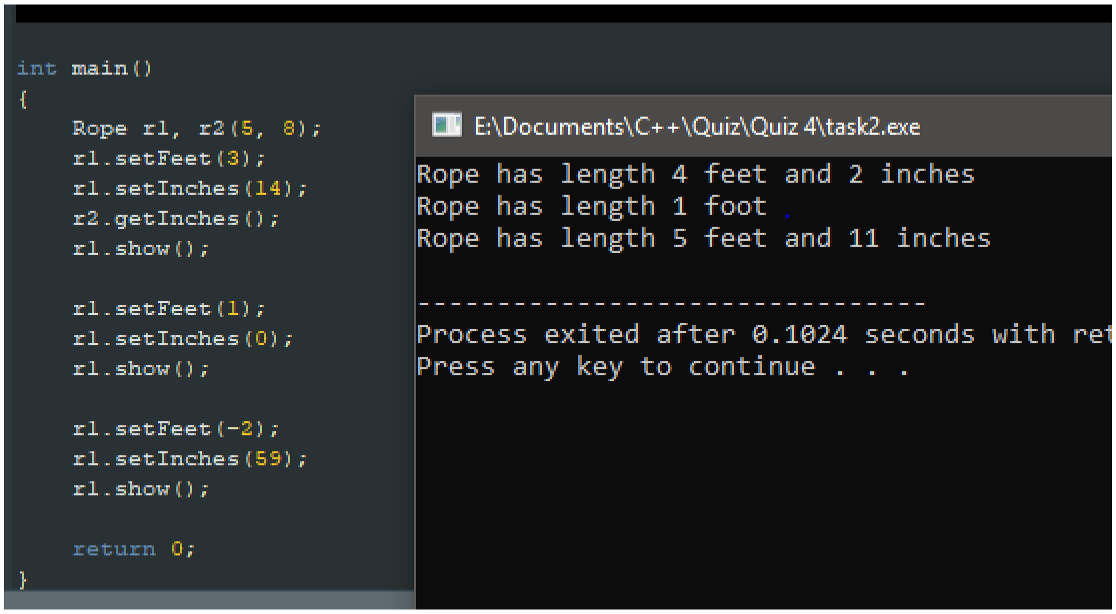
1. Non - parameterized Constructor (called with object r1)
2. Parameterized Constructor (called with object r2)
3. setFeet function (to initialize feet)
4. setInches function (to initialize inches)
5. getInches function (to return inches)
6. Show function (to print the output)

Solution :

```
 public:
  // Non - parameterized Constructor
     Rope()
     {
          // Default values are not mentioned. Should be in range

          feet = 1; // greater than 0
          inches = 1; // greater than 0 and less than 12
     }

     // Parameterized Constructor

     Rope(int f, int i)
     {
          // Avoid duplication of code.
          // Call the setter functions when required
          setFeet(f);
          setInches(i);
     }

     void setFeet( int f )
     {
          if ( f > 0 )
               feet = f;
          else
```

```
                        feet  = 1;
        }

        void setInches( int i )
        {
                int temp;

                if ( inches < 0 )
                        inches = 1;

                else
                {
                        temp = i / 12; // if greater than 12, returns number of
feets, like if i = 15, i / 12 = 1. if i =39, i / 12 = 3 and so on

                        feet = feet + temp; // adding the extra feet excluded from
inches
                        inches = i % 12; // returns the remainder from 12, if i = 5,
returns 5. if i = 18, returns 6 and 1 will be added to feet and so on.
                }
        }

        int getInches( void )
        {
                return inches;
        }

        void show() const
        {
                // Following the output format But adding extra functionalities for
extreme cases like if feet = 0 or inches  = 0;
                cout << "Rope has length ";

                if ( feet == 1 )
                        cout << "1 foot ";
                else if ( feet > 1 )
                        cout << feet << " feet ";

                if ( inches == 1 )
                        cout << "and 1 inch";

                else if ( inches > 1 )
                        cout << "and " <<  inches << " inches";

                cout << '\n';
        }
```

Output with main() Function :



```
int main()
{
    Rope r1, r2(5, 8);
    r1.setFeet(3);
    r1.setInches(14);
    r2.getInches();
    r1.show();

    r1.setFeet(1);
    r1.setInches(0);
    r1.show();

    r1.setFeet(-2);
    r1.setInches(59);
    r1.show();

    return 0;
}
```

E:\Documents\C++\Quiz\Quiz 4\task2.exe

```
Rope has length 4 feet and 2 inches
Rope has length 1 foot
Rope has length 5 feet and 11 inches

--------------------------------
Process exited after 0.1024 seconds with ret
Press any key to continue . . .
```

QUIZ Ended

```
}
```

Lecture 11

The Question 4 of the Practice 5 was,
Create class Rational with two parameters p & q, where q must not be zero. In the setter put a check for the value of q, assign 1 if 0 is passed by user. Write parameterized construction, show function and function to add, subtract two rational numbers. Write main to demonstrate the class.

The add,subtract and show functions are as follows:

```
    void add( Rational r1, Rational r2 )
        {
            this->p = (r1.p * r2.q) + (r2.p * r1.q);
            this->q = r1.q * r2.q;
        }

    void subtract( Rational r1, Rational r2 )
        {
            this->p = (r1.p * r2.q) - (r2.p * r1.q);
            this->q = r1.q * r2.q;
        }

    void show()
        {
            if ( p == 0 )
                cout << '0' << endl;
            else if ( q < 0 )
                cout << -p << '/' << -q << endl;
            else
                cout << p << '/' << q << endl;
            cout << "----------------------\n";
        }
```

Now, we discussed the normalize() function to simplify the rational number to its lowest form. Like if we got a rational number like 3/9 so it should convert this number to ⅓ and so on.

Normally in mathematics, when we normalize a rational number we simply start from 2, if both of the numerator and denominator can be whole divided with 2, we divide it and write the rational new number, if it can be divided again with 2, we divide it by 2. And when we reach a point where any of the numbers can not be divided by 2 we increase our factor and check with 3. In programming form, we can say that if both numbers can be divided by the factor, divide them, else increment the number by 1. So, how long will we continue to do this? We continue to divide with the factor until the factor reaches the minimum of denominator and numerator . Like, if the rational number is 100/4, then we increase our factor until 4. OR the rational number 3/99 can have a maximum factor 3.

Code to normalize the Rational numbers :

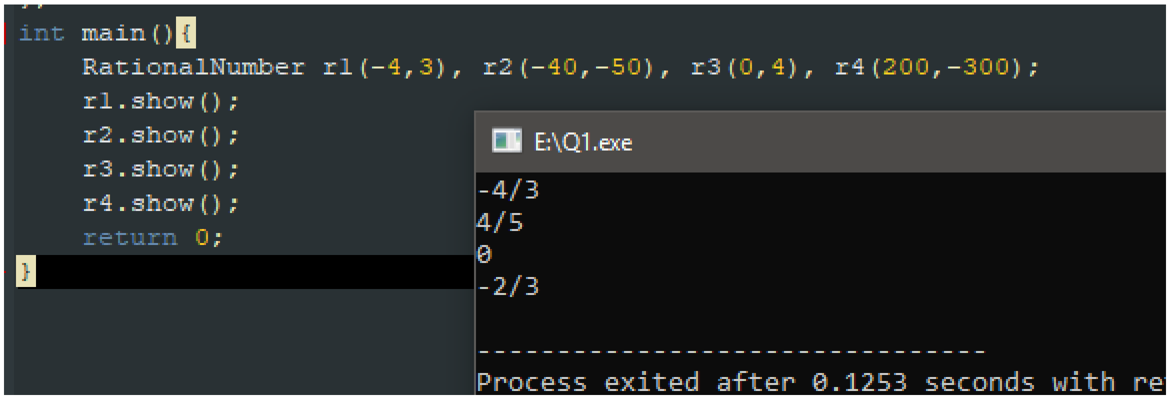
```
// abs means absolute
int abs( int n )
{
        if ( n < 0 )
                return -n;
        else
                return n;
}
```

```
void normalize( int p, int q )
{
        for ( i = 2; i <= abs(p) && i <= (q);       ) //starting our factor from 2 and running the loop until minimum of p or q is reached
        {
                if ( p % i == 0 && q % i == 0 )
                {
                        p = p / i;
                        q = q / i;
                }
                else
                        i++;
        }

                this->p = p;
                this->q = q;
}
```

Why do we use Absolute Function?

We need the absolute function to convert negative numbers into positive. If we don't use the absolute function, then our loop cannot run in case of negative numbers. Like, if a number like -3/9 is passed and we start our factor from 2 then the condition 2 < p (2 < -3) is false and the loop will not execute. To overcome this problem we used the absolute function to turn the negative numbers into positive and check our factor. Now, the condition 2 < abs(p) means 2 < abs(-3) or 2 < 3 is true and our loop will continue.

Main Function to demonstrate the Output



```
int main(){
    RationalNumber r1(-4,3), r2(-40,-50), r3(0,4), r4(200,-300);
    r1.show();
    r2.show();
    r3.show();
    r4.show();
    return 0;
}
```

E:\Q1.exe

```
-4/3
4/5
0
-2/3

--------------------------------
Process exited after 0.1253 seconds with re
```

Operators:

In the last Lecture we discussed the assignment operator in detail. If you don't remember what assignment operator is and how it works, please go through the previous lecture again and grasp the idea about it.

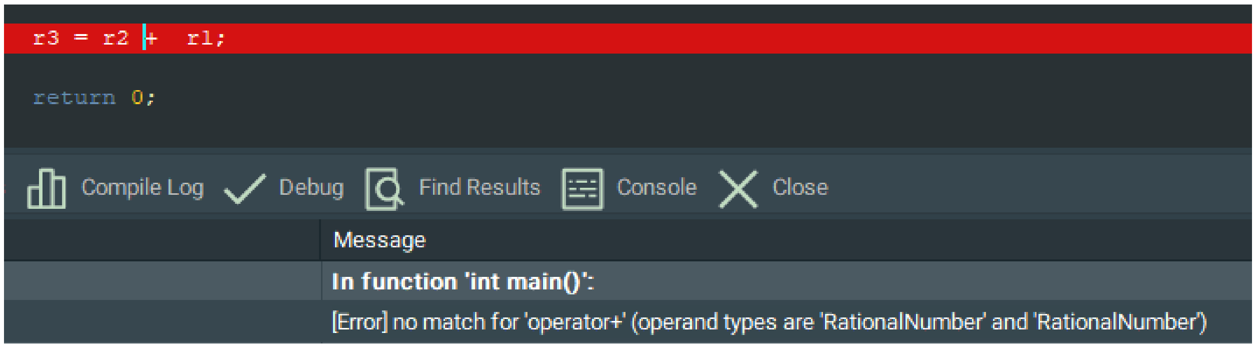
Now, continuing further in this lecture we discussed the add (+) and (+=) operators.

So, the first Question is what are operators?

In mathematics and sometimes in computer programming, an operator is a character that represents an action, as for example + is an arithmetic operator that represents addition.

Operator Overloading :

The Arithmetic operator (+, -, *, /, %) are already defined for the built in data types like int, float, char etc. But, right now, when we are creating/defining our own data types(class represents a data type), So, we need to write our own operators for this purpose. Consider the previous example of rational class, What will happen if we try to add two rational numbers like r1 + r2?



```
r3 = r2 +  r1;

return 0;
```

```
In function 'int main()':
[Error] no match for 'operator+' (operand types are 'RationalNumber' and 'RationalNumber')
```

We can clearly see that the compiler will give an error message. The compiler knows how to add two integers, how to subtract floats, how to multiply doubles as these are already defined (called primitive types) but it does not know how to add two rational numbers, because we at our own defined the rational data type. So, it is our duty to define how the operators will act upon our defined data types. Or in other words, we have to define which action to be performed with which operator. Defining our own operators in a program is called operator loading.

We can do anything while defining our own operators, But it is not recommended to change the basic definitions of the operators like the + operator should do addition.

Syntax for operator loading :

```
ReturnType  operator  symbol ( parameters )
{
        ... .Definition. ...
}
```

Here,

- **ReturnType** is the return type of the function. (it is mostly the same data type as of the class i.e class name)
- **operator** is a keyword.
- **symbol** is the operator we want to overload. Like: **+, -, <, >, ++,** etc.
- **parameters** is the arguments passed to the function.

Defining the (+) operator :

The simple addition (+) is like :

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

We will follow the original definition of the + operator. Here, the + operator adds B and C and returns a new object of the same type which is stored in A. The operands (on which action is performed) i.e B and C remained unchanged after the execution of the statement, and results are stored in A as a new object.

Code to add two Rational Objects :

```
Rational operator + ( Rational &r1 )
        {
                Rational newNumber;

                newNumber.p = (this->p * r1.q) + (r1.p * this->q);
                newNumber.q = this->q * r1.q;

                return newNumber;
        }
```

NOTE : are you confused with the & sign used in the parameters of function? 👻

```
cmp decision, "Yes"
je Lec_10.page_5
```

While calling this function in main() we can simply use + operator to add two rational numbers.

Like, r3 = r1 + r2, // Consider the previous example where we used the add() function to add the rational numbers. This statement is exactly equals to,

r3 = r1 + r2 is same as **r3 = r1.add(r2)**

It explains that when we use + operator, between two objects, then the function of + operator is called where the operand (r1) on the left acts as caller object and becomes the current object in the function while the operand (r2) which is on the right is passed to the function as parameter.

Defining the (+=) operator :

The simple addition (+=) is like :

```
A += B;
```

We will follow the original definition of the += operator. Here, the += operator adds A and B and returns a current object which is again stored in A or we can say that the value of A will be updated. The operands on the left i.e B remained unchanged after the execution of the statement, and results are stored in A not as a new object but only the current (previous, called) object is updated.

Code to add two (+=) Rational Objects :

```
void operator += ( Rational &r1 )
        {
                this->p = (this->p * r1.q) + (r1.p * this->q);
                this->q = this->q * r1.q;
        }
```

While calling this function in main() we can simply use += operator to add two rational numbers.

Like, r1 += r2, // Consider the previous example where we used the add() function to add the rational numbers. This statement is exactly equals to,

r1 += r2 is same as **r1 = r1.add(r2)**

And also **r1 = r1 + r2**

It explains that when we use += operator, between two objects, then the function of += operator is called where the operand (r1) on the left acts as caller object and becomes the current object in the function while the operand (r2) which is on the right is passed to the function as parameter.

Relating + and += :

Now, we are trying to relate + and += operators to make our program more simple(or more complex for some reason 😢).
If we focus on the above two codes of the + and += operators, we can see that actually we are doing exactly the same thing in both of the programs. The only difference is the object which is to be updated or the object in which the results of addition is stored. In the first case i.e +, we created a new object and updated that object. While in the other case, we changed the current object, and stored the results into it.

Now consider an example of simple numbers instead of the objects of class.

R = A + B; // R is the object where result is to be stored and A and B should remain unchanged

One method to do this is by simply adding A and B and storing it into R. But, if we have an option of += then we follow another method to do the same thing.

First simply let:

R = A; // We Stored A into R OR in programming context made A copy of a into R 😎

Now, all we need to do is :

R = R + B; //Adding R (which is a copy of A) into B and storing result into R

Or isn't it simply :

R += B;

Hopefully, we gathered the idea that how + and += can be used simultaneously. Now, coming back to the coding portion. If we implement the same logic with the objects of the class and adding them, The code will be like this :

Code :

```
        Rational operator + ( Rational &r1 )
                {
                        Rational newNumber = *this; // Here, *this represents the
        first operand in main function, like A in R = A + B

                        newNumber += r1;
                        return newNumber;
                }

        void operator += ( Rational &r1 )
                {
                         // Here, this represents the newNumber passed from the
        upper function
                        this->p = (this->p * r1.q) + (r1.p * this->q);
                        this->q = this->q * r1.q;
                }
```

We have already covered the theory about this pointer in Lec_9. As this is a pointer which points to the current object so, *this means the value of this pointer which is the current object itself.

Hopefully, we got a clear idea of how we related + and += operators. Now we can do same with the - and -= operators and all the other operators like * and *=, / and /= and all that.

If you still have any confusion related to any of these concepts, plz go through the previous example again to understand the logic before moving further.

Using & sign with Return Types :

Firstly we need to have a clear understanding of what an alias is and how it really works. If you have any confusion regarding the alias, (once again 😁) you can refer to the Lec_10 notes. We have covered it in detail there.

So, firstly we need to understand what exactly happens when the return type of the function is the same as the class name.

Checkout this code again,

```
Rational operator + ( Rational &r1 )
{
        Rational newNumber = *this;
        newNumber += r1;
        return newNumber;
}
```

First, we created a new object of the class Rational here and then manipulated the object, but we all know that all the local variables of the functions are destroyed as soon as the function ends. So, the newNumber object will also be destroyed when the function ends until we return the object.

What exactly happens when a variable or object is returned?

When a function ends, all the variables created in the function are destroyed, until unless we return a specific variable. When we return an object, the original object will be destroyed but a copy of the original object will be returned to the point from where the function is called. Like in this case, we created an object named newNumber inside the function, this object is fully accessible inside the function but once the return statement is executed, this object is destroyed and a new copy of the object is returned to the calling point. Here, the concept of new copy is very important if we relate it with the example discussed in previous lecture of aliases, where we write variable names inside parenthesis, when we write only data type name and variable name, it creates a new variable and stores the value passed from the main function. Same case happens with the return type, when we return an object the original object destroyed (can no longer be accessed with its original name) and a copy of that object comes at the place where the function is called.

Hopefully, we got the idea about the return type of functions. Now, coming towards our final point, i.e using & sign with the return data type. Consider the code :

```
Rational& operator += ( Rational &r1 )
{
        this->p = (this->p * r1.q) + (r1.p * this->q);
        this->q = this->q * r1.q;

        return *this;
}
```

Here, we added the & sign with the return type. So, how is it different from the previous example?

The difference is that when we use & sign with return type, it does not create a copy of the returning object rather it returns the original object to the point from where the function is called.

Verification : Copy of the object is always created with the copy constructor. So, when a function ends in which we created a new object, the destructor of that object will be automatically called and if we are returning an object then the copy constructor will also be called automatically. We can verify this by writing our own copy constructor and writing some cout statements into that constructor to verify whenever it is called and which object is destroyed.

What's the benefit of using & sign with the return type?

The first key benefit of using & sign is that the copy constructor will not be called and it will improve the execution time of the program. Secondly, when we use return *this; it means that we are returning the current object to the original place. The term to remember is when we manipulate the current object in a function, we should use & sign and return the current object but when we create a new object inside the function and have to return the new object we shouldn't use the & sign.

Another benefit of using & sign and returning the current object is that we can use multiple functions at a time.

Like, r1.normalize().show();

Here, if the return type of the normalize function is void then the compiler will give an error because when the normalize function is executed it will return nothing and the statement after the execution of normalize will be only .show() and it means nothing for the compiler...
So, if we used the & sign and return *this; in the normalize function then after its execution the statement will be like this.show(); (in main function "this" means name of object which is r1 in this case). So, we can use multiple functions at a time and we can do a lot more stuff like if we try to do A = B += C; it will give an error if the return type of += is void.

So, it is highly recommended to return *this object and using & sign with it. Even with the setter functions as well.

So, our final code to add two rational objects looks like this,

```
Rational operator + ( Rational &r1 )
        {
                Rational newNumber = *this;
                return newNumber += r1;
        }
Rational& operator += ( Rational &r1 )
        {
                this->p = (this->p * r1.q) + (r1.p * this->q);
                this->q = this->q * r1.q;
                return *this;
        }
```

You can continue it further by writing several functions like - and -= etc.

----THE END----