More on OOP Design Concepts References, Constant Modifiers, this Pointer, Operator Overloading, friend functions, and Static Members

In this Week

- Design of classes
  - Reference type Parameters
  - Constant Modifiers
  - Constant objects
  - Constant member functions
- Class as a data type inside the same class
- The **this** pointer
- Operator overloading
- Friend functions
- Static member variables and functions

Design of Classes

- Let's consider the mathematical rational number system of the form **a/b** with integers **a** and **b** such that **b ≠ 0**
- We would like to design a class to represent rational number objects
- Typically, we would like to have constructors, getters, setters and other member functions to work with the numerators and denominators of rational numbers
- See the following class design

Design of Classes

```
class RationalNumber
{
    /*
    This class is designed to represent Rational Number objects.
    A rational number is a number of the form a/b with integers
    a and b such that b is different from 0.
    */
private:
    int a, b;
public:
    //Constructors
    RationalNumber();
    RationalNumber(int, int);

    //Getters
    int getNumerator();
    int getDenominator();

    //Setters
    void setNumerator(int num);
    void setDenominator(int denom);

    //Additional member functions
    double toDouble();
    void standardize();
    void reduce();
    void print();
};
```

Design of Classes

```
//Constructors
RationalNumber::RationalNumber()
{
    //As a default object, let us construct 0/1 rational number
    a = 0;
    b = 1;
}

RationalNumber::RationalNumber(int num, int denom)
{
    //If the denominator parameter is 0, ignore it and use 1
    a = num;
    b = denom != 0 ? denom : 1;
    //Now that the object is created, standardize and reduce it
    standardize();
    reduce();
}

//Getters
int RationalNumber::getNumerator()
{
    return a;
}
int RationalNumber::getDenominator()
{
    return b;
}
```

Design of Classes

```
//Setters
void RationalNumber::setNumerator(int num)
{
    a = num;
    //Now that numerator of an existing object is modified,
    //standardize it and reduce it
    standardize();
    reduce();
}
void RationalNumber::setDenominator(int denom)
{
    //If the denominator parameter is 0, ignore it and use 1
    b = denom != 0 ? denom : 1;
    //Now that denominator of an existing object is modified,
    //standardize it and reduce it
    standardize();
    reduce();
}

//Additional member functions
double RationalNumber::toDouble()
{
    return static_cast<double>(a)/b;
}
void RationalNumber::standardize()
{
    if (b < 0)
    {
        a *= -1;
        b *= -1;
    }
}
```

Design of Classes

```
void RationalNumber::reduce()
{
    if (a == 0)
    {
        b = 1;
        return;
    }
    else
    {
        //Remeber that the denominator is NEVER zero by design
        //Therefore here both numerator and denominator are non-zero.
        int m = abs(a);
        int n = abs(b);
        int gcd = m < n ? m : n;
        while (gcd > 0)
        {
            if (m % gcd == 0 && n % gcd == 0)
                break;
            gcd--;
        }
        a /= gcd;
        b /= gcd;
    }
}
void RationalNumber::print()
{
    cout << a << "/" << b;
}
```

Design of Classes

```
int main()
{
    //Test constructors
    RationalNumber r1, r2(-5, 6);
    RationalNumber *r3  = new RationalNumber();
    RationalNumber *r4;
    r4 = new RationalNumber(4, -6);

    //Test getters
    cout << "r1 numerator is " << r1.getNumerator() << endl;
    cout << "r3 denominator is " << r3->getDenominator() << endl;

    //Test setters, standardize, reduce and print member functions
    r2.setDenominator(-10);
    cout << "r2 is now "; r2.print(); cout << endl;

    //Print all the objects
    cout << "r1 = "; r1.print(); cout << endl;
    cout << "r2 = "; r2.print(); cout << endl;
    cout << "r3 = "; r3->print(); cout << endl;
    cout << "r4 = "; r4->print(); cout << endl;

    //Test toDouble member function
    cout << "In double format, r4 = " << r4->toDouble() << endl;

    //Delete objects created on the heap
    delete r3;
    delete r4;

    system("Pause");
    return 0;
}
```

Design of Classes

- For efficiency purposes, parameters of functions should pass by **reference**
- The declaration of the **RationalNumber** class with all parameters passing by reference is shown below
- Apply the same modification to the implementation of the member functions in order to have a correct class definition

Design of Classes

```
class RationalNumber
{
    /*
    This class is designed to represent Rational Number objects.
    A rational number is a number of the form a/b with integers
    a and b such that b is different from 0.
    */
private:
    int a, b;
public:
    //Constructors
    RationalNumber();
    RationalNumber(int&, int&);

    //Getters
    int getNumerator();
    int getDenominator();

    //Setters
    void setNumerator(int& num);
    void setDenominator(int& denom);

    //Additional member functions
    double toDouble();
    void standardize();
    void reduce();
    void print();
};
```

Design of Classes

- Now try to run the main program that we have already implemented
- We will see that the following code segments in the main program

  RationalNumber r2(-5, 6);

  r4 = new RationalNumber(4, -6);

  r2.setDenominator(-10);

  will have syntax errors!
- The reason is that a literal values -5, 6, 4 or -6 can not pass by reference because they are not **L-values**!

Design of Classes

- In order to solve this syntax error, C++ provides the **const** modifier
- When a reference parameter is made a **const**, then it is syntactically correct to pass literal values arguments to the parameter
- The declaration of the **RationalNumber** class with all parameters passing by reference with **const** modifier is shown below
- Apply the same modification to the implementation of the member functions in order to have a correct class definition
- With these modifications, the same main program will run perfectly fine with no any syntax errors

Design of Classes

```
class RationalNumber
{
    /*
    This class is designed to represent Rational Number objects.
    A rational number is a number of the form a/b with integers
    a and b such that b is different from 0.
    */
private:
    int a, b;
public:
    //Constructors
    RationalNumber();
    RationalNumber(const int&, const int&);

    //Getters
    int getNumerator();
    int getDenominator();

    //Setters
    void setNumerator(const int& num);
    void setDenominator(const int& denom);

    //Additional member functions
    double toDouble();
    void standardize();
    void reduce();
    void print();
};
```

Constant Objects

- We may also construct a constant object so that not to allow any modification to its members
- For example,

  const RationalNumber r(2, 3);
- But then if we try to call any of its member functions including the ones that do not modify any member variable, then we will get an error message **"object has type qualifier that are not compatible with the member function"**. See example below…

Constant Objects

```
int main()
{
    //Construct constant objects
    const RationalNumber r1(2, 3);
    const RationalNumber *r2;
    r2 = new RationalNumber;

    //Try to modify member variables
    r2->setNumerator(-1);

    //Print numerators or denominators
    cout << "r1 numerator is " << r1.getNumerator() << endl;
    cout << "r2 denominator is " << r2->getDenominator() << endl;

    //Print objects and their value in double data type
    cout << "r1 in double format is " << r1.toDouble() << endl;
    r1.print();

    delete r2;

    system("Pause");
    return 0;
}
```

Constant Objects

- Why are we getting such syntax error messages?
- Because when we execute say

```
cout<<r1.getNumerator()<<endl;
```

  then the compiler does not have any guarantee that the **getNumerator** member function will not modify one or more member variables of **r1**
- Yes we as designers and developers of the class know that **r1** member variables will not be modified; but the compiler doesn't know
- Therefore we need to tell the compiler **r1** will not be modified in the **getNumerator()** member function

Constant Member Functions

- **Terminology:- An object that is calling a member function is called the calling object of the function.**
- For example, when we execute the **r1.getNumerator()** then **r1** is the calling object
- Therefore as a rule we should always designate any member function that does not modify any member variable of its calling object and that we might need to call with a constant object as a **constant member function**
- In order to make a member function a constant function, designate it as constant member function both in its declaration and its implementation
- The **RationalNumber** class declaration with these modifications is shown below
- Apply the same modification to the implementation of the member functions in order to have a correct class definition
- We can then invoke correctly a constant member function from either a constant or a non-constant calling objects

Constant Member Functions

```
class RationalNumber
{
    /*
    This class is designed to represent Rational Number objects.
    A rational number is a number of the form a/b with integers
    a and b such that b is different from 0.
    */
private:
    int a, b;
public:
    //Constructors
    RationalNumber();
    RationalNumber(const int&, const int&);

    //Getters
    int getNumerator() const;
    int getDenominator() const;

    //Setters
    void setNumerator(const int& num);
    void setDenominator(const int& denom);

    //Additional member functions
    double toDouble() const;
    void standardize();
    void reduce();
    void print() const;
};
```

Constant Member Functions

- Now, the test main program we have will work perfectly fine except for the statement

  `r2->setNumerator(-1);`

- And this is expected because we should not attempt to modify any member variable of a constant object
- The test main program with the above piece of code commented is shown below

Constant Member Functions

```
int main()
{
    //Construct constant objects
    const RationalNumber r1(2, 3);
    const RationalNumber *r2;
    r2 = new RationalNumber;

    //Try to modify member variables
    //r2->setNumerator(-1); Commented because modification not allowed

    //Print numerators or denominators
    cout << "r1 numerator is " << r1.getNumerator() << endl;
    cout << "r2 denominator is " << r2->getDenominator() << endl;

    //Print objects and their value in double data type
    cout << "r1 in double format is " << r1.toDouble() << endl;
    r1.print();

    delete r2;

    system("Pause");
    return 0;
}
```

Class as a data type inside same class

- Suppose that we would like to add one more member function named **isEqual** that will work as follows

  bool answer = r1.isEqual(r2);

- This member function will have a **RationalNumber** calling object, it will take one **RationalNumber** type argument, and it will return true if the calling object is equal to the argument and will return false otherwise
- **Wait a minute?**
- **How can we use a class as a data type inside the same class?**
- **Answer:-** C++ allows to use a class as a data type in the same class
- The **RationalNumber** class declaration together with the **isEqual** member function will then look like as follows

Class as a data type inside same class

```
class RationalNumber
{
    /*
    This class is designed to represent Rational Number objects.
    A rational number is a number of the form a/b with integers
    a and b such that b is different from 0.
    */
private:
    int a, b;
public:
    //Constructors
    RationalNumber();
    RationalNumber(const int&, const int&);

    //Getters
    int getNumerator() const;
    int getDenominator() const;

    //Setters
    void setNumerator(const int& num);
    void setDenominator(const int& denom);

    //Additional member functions
    double toDouble() const;
    void standardize();
    void reduce();
    void print() const;
    bool isEqual(const RationalNumber& r) const;
};
```

Class as a data type inside same class

- The implementation of the **isEqual** member function will look like

```
bool RationalNumber::isEqual(const RationalNumber& r) const
{
    int x = r.a;    //We could also do int x = r.getNumerator();
    int y = r.getDenominator(); //We could also do int y = r.b;
    //Now we will compare if a/b is equal to x/y
    if (a*y == b*x)
        return true;
    else
        return false;
}
```

- As can be seen here, the constant object parameter **r** can call **getNumerator()** or **getDenominator()** member functions without any syntax error because these member functions are designated constant
- The following test program shows how to use **isEqual** member function

Class as a data type inside same class

```
int main()
{
    RationalNumber r1, r2(-5, 6);
    RationalNumber *r3  = new RationalNumber();
    RationalNumber *r4;
    r4 = new RationalNumber(4, -6);

    if (r1.isEqual(*r3))
        cout << "r1 and r3 are equal" << endl;
    else
        cout << "r1 and r3 are not equal" << endl;


    if (r4->isEqual(r2))
        cout << "r2 and r4 are equal" << endl;
    else
        cout << "r2 and r4 are not equal" << endl;

    //Delete objects created on the heap
    delete r3;
    delete r4;

    system("Pause");
    return 0;
}
```

Concluding Remarks

- In OOP, the most important step is the design of the classes and as such we should perform the following tasks at the beginning of software development to help us design a software that will be fast to develop, efficient to run, and easy to debug and maintain
  - **What classes** do you need in order to solve a problem
  - **Pass parameters by reference** as much as you can
  - **Make parameters const** if they should not be modified in a member function
  - Make member functions that do not modify any member variable **constant member functions**
  - Remember **a const object can not call a non-const member function!**

The this Pointer

- Consider the setNumerator member function of the **RationalNumber** class. We have chosen the name of its parameter to be **num**. But what if we want to use the same parameter name as the member variable name such as the one shown below?

```
//Setters
void RationalNumber::setNumerator(const int& a)
{
    a = a;
    //Now that numerator of an existing object is modified,
    //standardize it and reduce it
    standardize();
    reduce();
}
```

- We will get a syntax error!

The this Pointer

- Why error?
- Because both the a variables in the statement

  a = a;

  refer to the parameter (variable scope)
- Thus we are trying to modify the value of the constant parameter **a** which is not allowed
- So how can we tell the compiler the left hand side of the assignment operator refers to the member variable while the right hand side is the parameter?

  Answer:- We use the **this** pointer

The **this** Pointer

- C++ provides the **this** pointer in order to access member variables and member functions of a class from within the class with no ambiguity
- The **this** pointer is a pointer to the calling object
- Once again we use the **->** operator with the **this** pointer
- Alternatively we may dereference the this pointer and then use the dot operator as **(*this).**
- Hence **this->a** will access the **a** member variable of a calling object
- Similarly **this->getNumerator()** will access the **getNumerator** member function of a calling object

RationalNumber Class with this Pointer

The **RationalNumber** class setter functions with their parameters modified to have the same names as the member variables is shown below

```
//Setters
void RationalNumber::setNumerator(const int& a)
{
    this->a = a;
    //Now that numerator of an existing object is modified,
    //standardize it and reduce it
    standardize();
    reduce();
}
void RationalNumber::setDenominator(const int& b)
{
    //If the denominator parameter is 0, ignore it and use 1
    this->b = b != 0 ? b : 1;
    //Now that denominator of an existing object is modified,
    //standardize it and reduce it
    standardize();
    reduce();
}
```

Member Variable

Parameter Variable

RationalNumber Class with this Pointer

- In order to appreciate the **this** pointer further in helping us to make some code clear and easy to understand; let us reconsider the **isEqual** member function
- In this member function we need to compare two RationalNumber objects
- Who are the two objects?

  Answer:- The *this calling object and the r parameter object
- We can therefore rewrite the **isEqual** member function as follows which makes it clearer to read

```
bool RationalNumber::isEqual(const RationalNumber& r) const
{
    //Compare the *this object with the r object
    if (this->a * r.b == this->b * r.a)
        return true;
    else
        return false;
}
```

Operator Overloading

- C++ allows to overload the common operators that are defined in C++ language
- There are two different types of operators we can overload
  - Binary Operators
    - **Example:- +, -, *, %, /, +=, -=, *=, %=, /=, ==, !=, >, <, >=, <=, &&, ||, <<, >>, [ ], ()**
  - Unary Operators
    - **Example:- -, ++, --, !**
- Recall that binary operators have two operands while unary operators have only one operand

Overloading Binary Operators

- Given two Rational Numbers **r1** and **r2**, consider the addition

  r1 + r2

- Our aim is to overload the **+** operator and implement it to add two rational numbers
- In C++, the expression **r1+r2**, is interpreted as a function call

  r1.operator+(r2)

- This means we are actually calling a **member function** of the **RationalNumber** class where **r1** is the calling object while **r2** will go as an argument to the operator member function
- That is in C++ any binary operator is implemented as a function and the left hand side operand is the calling object
- **This means whenever we want to implement a binary operator such that the left hand side is an object of our class; then we can simply implement the overloaded binary operator as a member function**

Overloading Binary Operators

- The declaration of an overloaded binary operator will therefore have the following syntax

  returnDatatype operator SYMBOL (parameter list);

- Example

  RationalNumber operator+(const RationalNumber &r) const;

  declares the overloaded binary addition operator implemented as a member function

- The **RationalNumber** class with such overloaded binary operator + is shown below

Overloading Binary Operators

```
    //Additional member functions
    double toDouble() const;
    void standardize();
    void reduce();
    void print() const;
    bool isEqual(const RationalNumber& r) const;

    //Binary operator member functions
    RationalNumber operator+(const RationalNumber& r) const;
};

RationalNumber RationalNumber::operator+(const RationalNumber& r) const
{
    int a1 = this->a;
    int b1 = this->b;
    int a2 = r.a;
    int b2 = r.b;
    //Now we would like to add (a1/b1) + (a2/b2) which is equal to (a1b2 + a2b1)/b1b2
    RationalNumber answer(a1*b2+a2*b1, b1*b2);
    return answer;
}
```

Overloading Binary Operators

- Here is a test code to see the overloaded addition binary operator in action. Analyze the program and determine its output

```
int main()
{
    RationalNumber r1(2, 3), r2(1, 2), r3(-2, 3), r4, *r5;

    //Add r1 and r2 and assign the result to r4
    r4 = r1 + r2;
    //Add r1 and r3 and assign the result to a heap memory pointed to by r5
    r5 = new RationalNumber();
    *r5 = r1 + r3;

    //Print the rational numbers you have got
    cout << "r1 = "; r1.print(); cout << endl;
    cout << "r2 = "; r2.print(); cout << endl;
    cout << "r3 = "; r3.print(); cout << endl;
    cout << "r4 = "; r4.print(); cout << endl;
    cout << "r5 = "; r5->print(); cout << endl;

    //delete any heap memory
    delete r5;

    system("Pause");
    return 0;
}
```

Overloading Binary Operators

- How about expressions like **r1 + 5** how should we implement them?
- The same manner! Because we have a calling object of our class
- The argument to the function is now however an integer
- Thus the declaration of such operator member function will be as follows

  RationalNumber operator+(const int &x) const;

- See below for the declaration and implementation

Overloading Binary Operators

```
    //Additional member functions
    double toDouble() const;
    void standardize();
    void reduce();
    void print() const;
    bool isEqual(const RationalNumber& r) const;

    //Binary operator member functions
    RationalNumber operator+(const RationalNumber& r) const;
    RationalNumber operator+(const int& x) const;
};
```

```
RationalNumber RationalNumber::operator+(const int& x) const
{
    //Instead of writing the actual code to add to rational numbers,
    //We can call the addition binary operator we had just implemented
    RationalNumber temp(x, 1);
    return *this + temp;
}
```

Overloading Binary Operators

- Here is a test main program

```
int main()
{
    RationalNumber r1(2, 3), *r2;

    //Add r1 and 3 and assign the result to the heap memory pointed by r:
    r2 = new RationalNumber;
    *r2 = r1 + 3;

    //Print the rational numbers you have got
    cout << "r1 = "; r1.print(); cout << endl;
    cout << "r2 = "; r2->print(); cout << endl;

    //delete any heap memory
    delete r2;

    system("Pause");
    return 0;
}
```

Overloading Binary Operators

- How about expressions like **5 + r1**
- This is different!!!
- We don't have a **RationalNumber** calling object on the left hand side
- Such operator must be implemented as a **non-member function**; that is a C++ function that does not belong to a class

Operator as Non-Member Function

- A non-member operator function takes its two operands as arguments in the order they are written in the expression involving the operator
- For example, in the expression

5 + r1

- The non-member operator function will have two arguments: **int** and **RationalNumber** in that order
- The function implementation is shown below

Operator as Non-Member Function

```
RationalNumber operator+(const int& x, const RationalNumber& r)
{
    return r + x;
}
int main()
{
    RationalNumber *r1 = new RationalNumber(2, 3);
    RationalNumber r2;
    r2 = 5 + *r1;

    //Print the rational numbers you have got
    cout << "r1 = "; r1->print(); cout << endl;
    cout << "r2 = "; r2.print(); cout << endl;
    delete r1;
    system("Pause");
    return 0;
}
```

Friend Functions

- A non-member operator function might tend to be slow to execute if it makes lots of function calls of the object parameter
- In such cases, we could make such non-member functions friends to the class and then they can access private member variables and member functions of objects of the same class
- In order to declare a non-member function as a friend to a class, put its declaration inside the class declaration and prefix it with **friend**
- Then implement it outside the class as before
- Remember friend function is **NOT** a member function
- The **int + RationalNumber** non-member function implemented as a friend function is shown below

Friend Functions

```
    //Additional member functions
    double toDouble() const;
    void standardize();
    void reduce();
    void print() const;
    bool isEqual(const RationalNumber& r) const;

    //Binary operator member functions
    RationalNumber operator+(const RationalNumber& r) const;
    RationalNumber operator+(const int& x) const;

    //Friend functions
    friend RationalNumber operator+(const int& x, const RationalNumber& r);
};
RationalNumber operator+(const int& x, const RationalNumber& r)
{
    /*
    //The implementation remains the same. No modification is needed
    return r + x;
    */

    //However in order to demonstrate that this non-member friend function
    //can access private member variables of the parameter r, let us write
    //this function in a different way
    RationalNumber temp(r.a, r.b);
    return temp + x;
}
```

Printing and Reading Objects Overloading the << and >> Operators

- Wouldn’t it be nice to be able to do

```
RationalNumber r;
cout << "Please enter a rational number:  ";
cin >> r;
cout << "The rational number is " << r << endl;
```

- In order to do so, we need to first understand the **cin** and **cout** statements of C++
- In fact both cin and cout are simply **OBJECTS**
- Neither cin nor cout is a keyword in C++. They are simply variable names declared in the std namespace

Printing and Reading Objects Overloading the << and >> Operators

- So how do they work then?
- Well they are simply the left hand side operands of the binary operators >> and <<
- The **>>** operator is called the **input streaming operator** and the **<<** operator is called the **output streaming operator**
- That is we have >> and << binary input and output streaming operators that we can overload as well!
- Now look at the expression:

cin >> r

Printing and Reading Objects Overloading the << and >> Operators

- In the **cin >> r** expression, we have a **cin** object on the left hand side and **r** object on the right
- The calling object is therefore the **cin** object
- The **cin** is designed to read int, float, char,…
- To read a **RationalNumber** therefore, we need to overload the >> operator
- How?
- **Answer:- Implement it as a non-member function!**

Printing and Reading Objects Overloading the << and >> Operators

- In order to have access to the private members of the right hand side operand, it is also a good idea to make it a friend function
- The >> operator will then take two arguments: a **cin** and a **RationalNumber** in that order
- What is the data type of cin? It is **istream** to mean input stream
- The **istream** is defined in the iostream library

Printing and Reading Objects Overloading the << and >> Operators

- Also recall that in C++ we can do

```
int x, y, z;
cin >> x >> y >> z;
```

- This chain operation is performed as follows:

```
((cin >> x) >> y) >> z;
```

- First cin >> x is performed. It reads x and returns an **istream** object by **reference**
- The returned istream is then used to read y
- And so on so forth

Printing and Reading Objects Overloading the << and >> Operators

- Therefore the function we will implement will have a return data type of istream by reference
- Similarly the **cout << r** is implemented as a non-member friend function
- It takes **cout** object (data type **ostream** to mean output stream defined in iostream library) and a **RationalNumber** in that order
- It returns a **reference** to **ostream** object
- See below for implementations

Printing and Reading Objects Overloading the << and >> Operators

```
    //Additional member functions
    double toDouble() const;
    void standardize();
    void reduce();
    void print() const;
    bool isEqual(const RationalNumber& r) const;

    //Binary operator member functions
    RationalNumber operator+(const RationalNumber& r) const;
    RationalNumber operator+(const int& x) const;

    //Unary operator member functions
    RationalNumber operator-() const;   //This is the same as -r
    RationalNumber& operator++();       //This is the same as ++r
    RationalNumber operator++(int DUMMY);//This is the same as r++

    //Friend functions
    friend RationalNumber operator+(const int& x, const RationalNumber& r);
    friend istream& operator>>(istream& in, RationalNumber& r);
    friend ostream& operator<<(ostream& out, const RationalNumber& r);
};
```

Printing and Reading Objects Overloading the << and >> Operators

```
istream& operator>>(istream& in, RationalNumber& r)
{
    cout << endl;
    cout << "\t Enter a numerator ";
    in >> r.a;
    cout << "\t Enter a non-zero denominator ";
    in >> r.b;
    //In case the input value for the denominator is zero, read it again
    while (r.b == 0)
    {
        cout << "\t Denominator can't be zero. Please enter a non-zero denominator ";
        in >> r.b;
    }
    r.standardize();
    r.reduce();
    return in;
}
ostream& operator<<(ostream& out, const RationalNumber& r)
{
    out << r.a << "/" << r.b;
    return out;
}
```

Printing and Reading Objects Overloading the << and >> Operators

- Here is a test main program to test the overloaded input/output stream operators

```
int main()
{
    RationalNumber r1;
    cout << "Please enter a rational number ";
    cin >> r1;
    cout << "You entered the rational number " << r1 << endl;

    RationalNumber r2, r3, *r4;
    cout << "Please enter three rational numbers ";
    cin >> r2 >> r3;
    r4 = new RationalNumber();  //Do not dereference r4 before pointing it to an object
    cin >> *r4;
    cout << "You entered r2 = " << r2 << ", r3 = " << r3 << ", and r4 = " << *r4 << endl;
    delete r4;
    system("Pause");
    return 0;
}
```

Overloading Unary Operators

- Unary operators have one operand
- In mathematics, unary operators are always placed to the left of their operands. Ex **-6**
- In C++ we have unary operators on the left of their operands such as **-r**, **--r**, **++r**
- Such unary operators are implemented as member functions and their calling object is the operand
- However, C++ also has **r++** and **r--** operators
- Such operators are assumed binary operators in C++ with a DUMMY integer type VARIABLE on the right hand side which we don't have to necessarily use
- Therefore **r++** is actually interpreted in C++ as **r ++ DUMMY integer**
- Then the calling object is the left hand side operand object and therefore can be implemented as member functions
- Similarly for the **r--** unary operator

Overloading Unary Operators

- The addition of such unary operators to the **RationalNumber** class is shown below

```
    //Additional member functions
    double toDouble() const;
    void standardize();
    void reduce();
    void print() const;
    bool isEqual(const RationalNumber& r) const;

    //Binary operator member functions
    RationalNumber operator+(const RationalNumber& r) const;
    RationalNumber operator+(const int& x) const;

    //Unary operator member functions
    RationalNumber operator-() const;   //This is the same as -r
    RationalNumber& operator++();       //This is the same as ++r
    RationalNumber operator++(int DUMMY);//This is the same as r++

    //Friend functions
    friend RationalNumber operator+(const int& x, const RationalNumber& r);
    friend istream& operator>>(istream& in, RationalNumber& r);
    friend ostream& operator<<(ostream& out, const RationalNumber& r);
};
```

Overloading Unary Operators

```
RationalNumber RationalNumber::operator-() const    //This is the same as -r
{
    RationalNumber answer(-this->a, this->b);

    return answer;

}

RationalNumber& RationalNumber::operator++()        //This is the same as ++r
{
    //Increment the *this object by 1
    //We have a/b. We need to make it a/b + 1 = (a+b)/b
    this->a += this->b;
    return *this;
}

RationalNumber RationalNumber::operator++(int DUMMY)//This is the same as r++
{
    //First copy the value of the *this object to a temp object
    RationalNumber temp = *this;

    //Now, increment the *this object by 1
    ++(*this);

    //Finally return the value of temp
    return temp;
}
```

Overloading Unary Operators

- Observe that we are returning a reference in the pre-increment ++ operator function
- This is important in order for us to be able to have the functionality of

```
RationalNumber r1(2, 5), r2, r3;
r2 = ++++r1;
r3 = (++r1)++;
```

- For post-increment, we are returning a local variable from the function (R-value) therefore it can not return by reference

Overloading Unary Operators

- Here is an example test main program

```
int main()
{
    RationalNumber r1(1, 2), r2, r3, r4, r5;
    r2 = -r1;              //r2 = -1/2 and r1 = 1/2
    r3 = ++++++r1;         //r3 = 7/2 and r1 = 7/2
    r4 = r2++;             //r4 = -1/2 and r2 = 1/2
    r5 = (++++r4)++;       //r5 = 3/2 and r4 = 5/2.
    //Here bracket is a must. Otherwise result is different.

    //Print the rational numbers you have got
    cout << "r1 = " << r1 << endl;  //output 7/2
    cout << "r2 = " << r2 << endl;  //output 1/2
    cout << "r3 = " << r3 << endl;  //output 7/2
    cout << "r4 = " << r4 << endl;  //output 5/2
    cout << "r5 = " << r5 << endl;  //output 3/2

    system("Pause");
    return 0;
}
```

Operator Overloading Remarks

- We can only overload existing operators of C++ language: We can not invent new operators
- The number of operands of an operator is determined by C++ language and can’t be changed
- Precedence of operators is determined by C++ language and can’t be changed
- At least one operand of an overloaded operator must be a **class** or a **struct** type
- The dot and scope resolution operators (**.** and **::**) can not be overloaded

Operator Overloading with structs

- When working with structs, we overload operators as non-member functions as demonstrated in the following example…

```
struct RationalNumber
{
    int a, b;
};
RationalNumber operator-(const RationalNumber &r)
{
    RationalNumber answer;
    answer.a = -r.a;
    answer.b = r.b;
    return answer;
}
bool operator == (const RationalNumber &r1, const RationalNumber &r2)
{
    if (r1.a * r2.b == r1.b * r2.a)
        return true;
    else
        return false;
}
int main()
{
    RationalNumber r1;
    r1.a = 1;
    r1.b = 5;
    cout << "r1 is " << r1.a << "/" << r1.b << endl;

    RationalNumber r2 = -r1;
    cout << "r2 is " << r2.a << "/" << r2.b << endl;

    if (r1 == r2)
        cout << "r1 and r2 are equal." << endl;
    else
        cout << "r1 and r2 are not equal." << endl;

    system("Pause");
    return 0;
}
```

Static Member Variables and Static Member Functions

- Consider the Rational Number class and suppose you have created several objects inside a main program similar to:

  RationalNumber r, p, q(1,3), a, b(0,2);
- Now, we would like to track how many RationalNumber objects we have created in our program?
- So far we don't have a way to know except by manually counting the objects
- However, C++ allows us to have a counter in the class that will be initialized to zero and will be incremented by one every time we create (construct) an object

Static Member Variables and Static Member Functions

- The first question to ask ourselves is therefore who does the counter belong to?
- Should every object created have a counter of its own just like every object has its own member variables?
- The answer is obviously NO!
- The counter is common to all objects created
- In fact, the counter belongs to the class!!!

Static Member Variables

- When a specific information is needed to be common to all objects; that is, when by design we would like a certain information to belong to the class then such information is designated as **static** in C++
- A **static member variable** is a variable that belongs to the class; thus common to all objects
- In order to designate a **member variable** as a **static**, prefix its declaration with the keyword **static**

Static Member Variables

- The declaration of a static member variable named **count** that we will use to count the number of **RationalNumber** objects created is shown below

```
class RationalNumber
{
    /*
    This class is designed to represent Rational Number objects.
    A rational number is a number of the form a/b with integers
    a and b such that b is different from 0.
    */
private:
    int a, b;
    //Static member variables
    static int count;   // = 0 Initialization NOT allowed here
public:
    //Constructors
    RationalNumber();
    RationalNumber(const int&, const int&);
```

Declaration of Static Member Variable

Static Member Variables

- Now the question is where should we initialize this static member variable?
- During declaration? Unfortunately not allowed!
- In the main program? Unfortunately no because it is private and can't be accessed in the main program!
- To have a setter? No because the user will then have the ability to modify the value of the count using the setter even without creating an object!
- For this reason, C++ allows us to initialize it right after the declaration of the class as follows

Static Member Variables

```
    //Additional member functions
    double toDouble() const;
    void standardize();
    void reduce();
    void print() const;
    bool isEqual(const RationalNumber& r) const;

    //Binary operator member functions
    RationalNumber operator+(const RationalNumber& r) const;
    RationalNumber operator+(const int& x) const;

    //Unary operator member functions
    RationalNumber operator-() const;   //This is the same as -r
    RationalNumber& operator++();       //This is the same as ++r
    RationalNumber operator++(int DUMMY);//This is the same as r++

    //Friend functions
    friend RationalNumber operator+(const int& x, const RationalNumber& r);
    friend istream& operator>>(istream& in, RationalNumber& r);
    friend ostream& operator<<(ostream& out, const RationalNumber& r);
};
/*
Any static member variable must be initialized outside the class declaration.
For clarity purposes, it is a good idea to initialize any static member variable
right below the class declaration.
*/
int RationalNumber::count = 0;//Initialization of static member variable requires re-declaration
```

Initialization of Static Member Variable

Static Member Variables

- Now the **count** static member variable should be incremented by 1 every time we construct an object
- That is every time a constructor member function is invoked
- Therefore it must be incremented by 1 inside each of the constructor member functions as shown below

Static Member Variables

```
/*
Any static member variable must be initialized outside the class declaration.
For clarity purposes, it is a good idea to initialize any static member variable
right below the class declaration.
*/
int RationalNumber::count = 0;//Initialization of static member variable requires re-declaration

//Constructors
RationalNumber::RationalNumber()
{
    //As a default object, let us construct 0/1 rational number
    a = 0;
    b = 1;
    //Increment the static member variable count
    count++;
}

RationalNumber::RationalNumber(const int& num, const int& den)
{
    //If the denominator parameter is 0, ignore it and use 1
    a = num;
    b = den != 0 ? den : 1;
    //Now that the object is created, standardize and reduce it
    standardize();
    reduce();
    //Increment the static member variable count
    count++;
}
```

Static Member Functions

- With the design we now have, the count will be initialized before the main program starts running and will be incremented by one every time we construct an object
- In order to get the count value, we will obviously need a public getter member function
- It is a good idea not to allow such a function any access to any non-static member variable or non-static member function because non-static member variables and functions belong to objects but this function does not belong to any object
- A member function designed to work with static member variables and that is not allowed to access any non-static member variable or function is designated as a **static member function**
- In order to designate a member function as static, prefix it with the keyword static in its declaration (but not its implementation)

Static Member Functions

```
class RationalNumber
{
    /*
    This class is designed to represent Rational Number objects.
    A rational number is a number of the form a/b with integers
    a and b such that b is different from 0.
    */
private:
    int a, b;
    //Static member variables
    static int count;    // = 0 Initialization NOT allowed here
public:
    //Constructors
    RationalNumber();
    RationalNumber(const int&, const int&);

    //Getters
    int getNumerator() const;
    int getDenominator() const;
    static int getCount();  //A static member function can not be constant function
    .
    .
    .
}
```

Declaration of a static member function

```
int RationalNumber::getCount()
{
    return count;
}
```

Implementation of a static member function

Accessing Static Member Functions

- In order to access a static member function outside the class, we use the class name together with the scope resolution operator as

```
cout << RationalNumber::getCount() << endl;
```

- We can also access a static member function using an object as we always do with non-static member functions

```
cout << r.getCount() << endl;
```

  where r is a **RationalNumber** object.

- As a final remark note that const modifier is not allowed for static member functions, you can not use the **this** pointer inside a static member function, a static member function can not access non-static member variables or non-static member functions, and that you can not access non-static member functions with a class name
- See the following test main program…

Accessing Static Member Functions

- The following program demonstrates the count of objects using static member variable. Determine its output.

```
int main()
{
    cout << "At the start " << RationalNumber::getCount() << " objects are constructed." << endl << endl;

    RationalNumber r1, r2(-2, 3), *r3;
    cout << "Now " << RationalNumber::getCount() << " objects are constructed." << endl;
    cout << "Now " << r1.getCount() << " objects are constructed." << endl;
    cout << "Now " << r2.getCount() << " objects are constructed." << endl;
    cout << "Now " << r3->getCount() << " objects are constructed." << endl << endl;

    r3 = &r1;
    cout << "Now " << RationalNumber::getCount() << " objects are constructed." << endl;
    cout << "Now " << r1.getCount() << " objects are constructed." << endl;
    cout << "Now " << r2.getCount() << " objects are constructed." << endl;
    cout << "Now " << r3->getCount() << " objects are constructed." << endl << endl;

    r3 = new RationalNumber;
    cout << "Now " << RationalNumber::getCount() << " objects are constructed." << endl;
    cout << "Now " << r1.getCount() << " objects are constructed." << endl;
    cout << "Now " << r2.getCount() << " objects are constructed." << endl;
    cout << "Now " << r3->getCount() << " objects are constructed." << endl;
    delete r3;
    system("Pause");
    return 0;
}
```