C++ Classes

We want to implement a program to support the processing of fraction,

$$f = \frac{numerator}{denominator}$$

Requirements of the representation (representation invariants)

- *numerator* and *denominator* are integers
- *denominator* > 0
- *numerator* and *denominator* are relatively prime, e.g. $\frac{6}{8}$ is reduced to $\frac{3}{4}$
- only 1 representation of the value zero, $\frac{0}{1}$

Modeling of fraction using C++ class

```
#include <ostream>

// helper function
int gcd(int m, int n)  //compute gcd of two +ve integers
{
   int r;
   while ((r = m % n) > 0)
   {
      m = n;
      n = r;
   }
   return n;
}
```

```
class fraction
{
   //overload the operator<< such that you can output
   //a fraction object to an output stream,
   //e.g. cout << f
   //The purpose of this function is similar to the method
   //toString() in Java.
   // A non-member function can access the private and
   // protected members of a class if it is declared as
   // a friend of that class.
   friend ostream& operator<<(ostream& os, fraction& f);

private:
   int numerator;
   int denominator;

public:
   fraction()  //default constructor
   {
      numerator = 0;
      denominator = 1;
   }

   fraction(int n, int d)   // ensure conformant with the
   {                        // representation invariants
      if (d == 0)
      {
         cerr << "ERROR: denominator is zero." << endl;
         exit(0);
      }

      if (n == 0)
      {
         numerator = 0;
         denominator = 1;
         return;
      }

      if (d < 0)
      {
         n *= -1;
         d *= -1;
      }

      int g;
      if (n < 0)
```

```
            g = gcd(-n, d);
        else
            g = gcd(n, d);

        numerator = n / g;
        denominator = d / g;
    }

    //overload (redefine) the operators
    //keyword 'const' makes the reference 'other' immutable
    fraction operator+ (const fraction& other)
    {
        int n = numerator * other.denominator +
                other.numerator * denominator;

        int d = denominator * other.denominator;

        fraction r(n, d);  //note that r is a local variable !!
        return r;
    }


    bool operator== (const fraction& other)
    {
        return (numerator == other.numerator) &&
               (denominator == other.denominator);
    }

    void print()
    {
        cout << numerator << "/" << denominator;
    }

    // other operators and functions in the class
};
//end of class definition

//This function is NOT a member function of the class fraction.
ostream& operator<<(ostream& os, fraction& f)
{
    os << f.numerator << "/" << f.denominator;
    return os;
}
```

```
//----------------------------------------------------
//codes in other functions that use class fraction

fraction f1; //f1 is initialized to 0/1 by the
             //default constructor fraction()

fraction f2(6, -8); //f2 is reduced to -3/4 by the
                    //constructor fraction(int, int)

fraction f3 = f1 + f2; //f3.operator=(f1.operator+(f2))

f3.print();  //call the member function

cout << f3; //note that the operator<< has been overloaded

//----------------------------------------------------
fraction *p1 = new fraction;
//create a fraction object with default constructor fraction()

fraction *p2 = new fraction(6, 8);
//use constructor fraction(int, int)

p2->print();  //call member function via a pointer

//----------------------------------------------------
fraction f4 = *p1 + *p2;

fraction f5 = f1;  //copy f1 to f5,
                   //f1 and f5 are two distinct objects

fraction *p3 = p1; //pointers p1 and p3 point to the same
                   //object instance

fraction *p4 = *p1 + *p2; //Error !!

fraction *p5 = new fraction;
*p5 = *p1 + *p2; //OK
```

Remark:

Object variables in C++ and Java (for students who have knowledge in Java)

C++	Java

fraction f1;
//f1 is an object instance
//default constructor is
//invoked automatically
 	
fraction f2(6,8);
	fraction f1, f2;
	//f1 and f2 are object
	//reference
	
	f1 = new fraction();
	//instantiate an object
	//instance
	
	f2 = new fraction(6, 8);

fraction *p1, *p2;
//pointer to an fraction
//object
 	
p1 = new fraction;
//instantiate an object
//instance
//no bracket for calling
//default constructor
 	
p2 = new fraction(6, 8);
 	
p1 = p2;
//both p1 and p2 point to
//the same object instance
	No explicit pointer variable
	in Java.

f1 = f2;
//copy contents of f2 to f1
	f1 = f2;
	//both f1 and f2 refer (point)
	//to the same object instance
	
	f1 = f2.clone();
	//make a copy of f2, and set
	//f1 to point to the newly
	//created copy of f2

Static vs dynamic binding of function calls

- In OO-programming, a derived class inherits the features of the base class.
- In a function call, a class object can be passed by value, by reference, or by pointer.
- C++ allows the user to pass an object of a derived class to a formal parameter of the base class type.

- Static binding (compile-time binding) is used
  - if the object parameter is passed by value, or
  - the member function in the base class is not virtual
- Dynamic binding (run-time binding) is used

  (i) if the object parameter is passed by reference or by pointer, and

  (ii) the member function is defined virtual in the base class.

- A virtual function in the base class will have an implementation.
- A virtual function without an implementation is called a pure virtual function.
- Pure virtual function in C++ corresponds to abstract method in Java.
- Java uses dynamic binding (run-time binding) in method invocation.

Example:

```
class baseClass
{
private:
   int x;

public:
   baseClass(int u = 0)
   {
      x = u;
   }

   virtual void print()
   {
      cout << "In baseClass x = " << x << endl;
   }
};
```

```
//derivedClass extends (or inherits) baseClass
class derivedClass: public baseClass

{
private:
   int a;

public:
   derivedClass(int u = 0, int v = 0) : baseClass(u)
   {  //invoke the baseClass constructor to initialize x

      a = v;
   }

   void print()
   {
      cout << "derivedClass::print()" << endl;
      baseClass::print();
      cout << "In derivedClass a = " << a << endl;
   }

};
```

```
//Case 1:

void callPrint(baseClass& b)   //b is passed by reference
{
   b.print();  //dynamic binding is used
}

int main()
{
   baseClass one(5);
   derivedClass two(3, 8);

   callPrint(one);
   callPrint(two);
}
```

```
Output:
In baseClass x = 5
derivedClass::print()
In baseClass x = 3
In derivedClass a = 8
```

```
//Case 2:

void callPrint(baseClass *p)   //p is passed by pointer
{
   p->print();  //dynamic binding is used
}

int main()
{
   baseClass one(5);
   derivedClass two(3, 8);

   callPrint(&one);
   callPrint(&two);
}
```

```
Output:
In baseClass x = 5
derivedClass::print()
In baseClass x = 3
In derivedClass a = 8
```

//Case 3:

```
void callPrint(baseClass b)   //b is passed by value
{
   b.print();   //static binding is used
}


int main()
{
   baseClass one(5);
   derivedClass two(3, 8);

   callPrint(one);
   callPrint(two); //print function in baseClass is executed
 }
```

```
Output:
In baseClass x = 5
In baseClass x = 3
```