Object-oriented Programming Operator Overloading Part 2

YoungWoon Cha

Computer Science and Engineering

KU KONKUK UNIVERSITY

Review

THREE ROLES OF AN OBJECT

Objects of user-defined types can play three different roles in a function:

Host object, Parameter object, Returned object.

We need to carefully study the issues related to objects in each role to understand the process of writing a function or overloading operators.

```
// Declaration
void input(...);
// Definition
void Fun :: input(...)
{
      ... ;
}
// Call
fun1.input(...);
```

```
// Declaration
void one(Type& para);
// Definition
void Fun :: two(Type& para)
{
     ... ;
}
// Call
fun1.one(para);
```

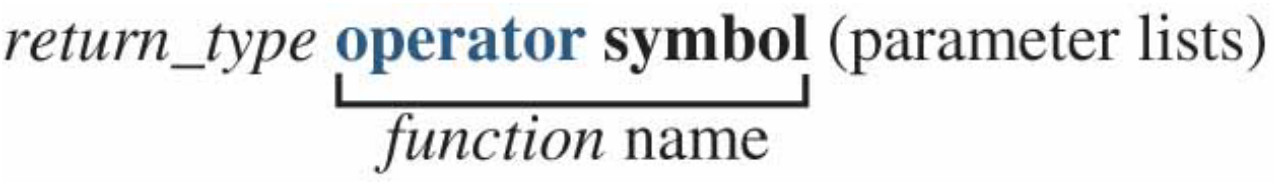
```
// Declaration
Fun& one();
// Definition
Fun& Fun :: one()
{
     ...;
}
// Call
Fun fun2 = fun1.one();
```

OVERLOADING PRINCIPLES

Overloading is a powerful capability of the C++ language that allows the user to redefine operators for user-defined data types, possibly with a new interpretation.
For example, instead of using a function call to add two fractions, we can overload the addition symbol (+) to do the same thing.

```
add(fr1, fr2)                    fr1 + fr2
```

To overload an operator for a user-defined data type, we need to write a function named *operator function*, a function that acts as an operator.



The whole purpose of operator overloading is to use the operator itself to mimic the behavior of built-in types.

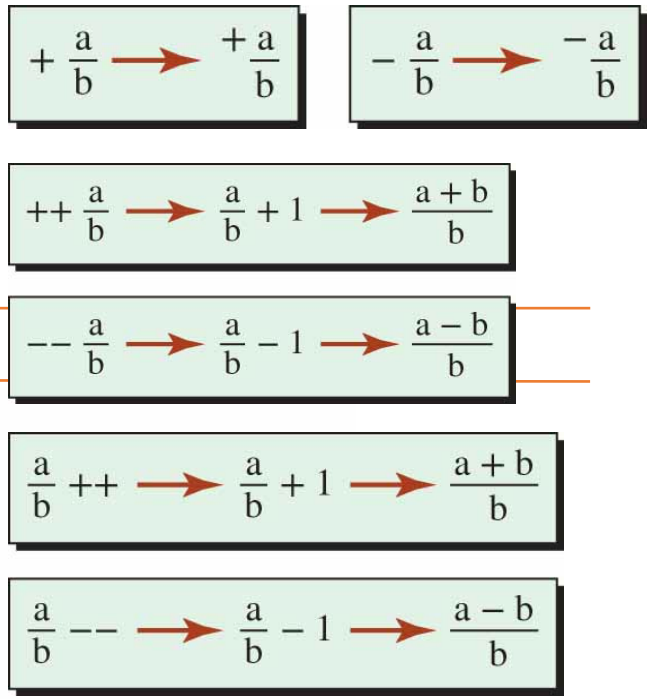
```
-fr  //operator
```

```
fr.operator-()  // funct
```

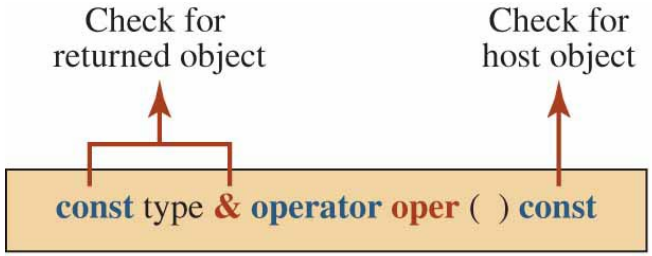
Unary Operators

In a unary operator, the only operand becomes the host object of the *operator function*. We have no parameter object.

This means that we should only think about two objects: the host object and the returned object as shown in the Figure.



Guideline for overloading unary operators



Prototype

1. The only operand is the host object. Can it be constant?
2. The returned object is the result. Can it be returned by reference? Can it be constant?

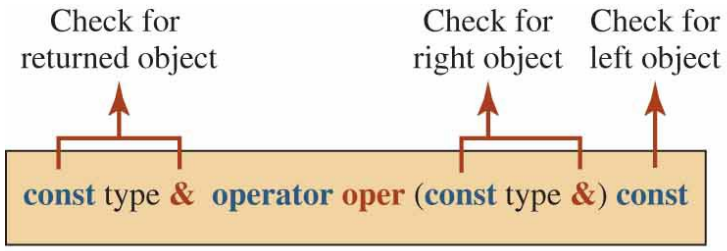
Checklist

Binary Operators

When we overload a binary operator as a member function, we need to consider one of the operands as the host object and the other as the parameter object.

It is common to overload only those binary operators as member functions in which the left operand(*lvalue*) has a different role than the right operand(*rvalue*) (= , +=, -= , *= , /= and %=).

Guideline for binary operators



Prototype

```
fract1 += fract2
fract1 -= fract2
fract1 *= fract2
fract1 /= fract2
fract1 %= fract2
```

1. The right object is the parameter object. Can it be passed by reference? Can it be constant?
2. The left object is the host object. Can it be constant?
3. The returned object is value of the operation. Can it be returned by reference? Can it be constant?

Checklist

KU KONKUK UNIVERSITY

More Operator Overloading (Cont'd)

Subscript Operator

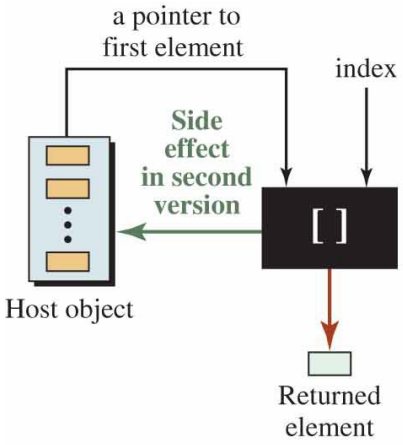
Another operator that can be overloaded as a member function is the subscript operator ([ ]).

The subscript operator is a binary operator in which the left operand is the name of an array and the right operand is a type that defines the index of the element in the array. e.g. arr[k];

This means that we should overload this operator only when our type is an array or something behaving like an array (such as string or a list).

If we want to overload this operator correctly, we need to have two versions.

In the first version, the operator is overloaded as an accessor function (for read-only); in the second version, it is overloaded as a mutator function.




```
type operator[ ] (int index) const
type & operator[ ] (int index)
```

```
// mutator operator [ ] (read + write)
arr[0] = 22.31;
// accessor operator [ ] (read-only)
const arr2(arr); cout << arr2[0];
```

Note:
The first version is used as an accessor (no side effect).
The second version is used as a mutator (return-by-reference).

Interface File for an Array Class

File array.h

```
1  /*****************************************************************
2   * The interface file for an Array class                         *
3   *****************************************************************/
4  #ifndef ARRAY_H
5  #define ARRAY_H
6  #include <iostream>
7  #include <cassert>
8  using namespace std;
9
10 class Array
11 {
12      private:
13          double* ptr;
14          int size;
15      public:
16          Array (int size); // Constructor
17          ~Array( ); // Destructor
18          Array(cosnt Array & other); // Copy Constructor
19          double& operator[ ] (int index) ; // Mutator
20          double operator[ ] (int index) const; // Accessor
21 };
22 #endif
```

Implementation File for Array Class Part 1

File Array.cpp

```
1  /*****************************************************************
2   * The implementation file for Array class                       *
3   *****************************************************************/
4  #include "array.h"
5
6  // Constructor (allocating memory in the heap)
7  Array :: Array (int s)
8  :size (s)
9  {
10       ptr = new double [size];
11 }
12 // Destructor (freeing memory in the heap)
13 Array :: ~Array( )
14 {
15       delete [ ] ptr;
16 }
17 // Copy Constructor
18 Array :: Array (const Array other)
19 {
20       size = other.size;
21       ptr = new double [size];
22       for (int i=0; i < size; i++)
23            ptr[i] = other.ptr[i];
24 }
```

Implementation File for Array Class Part 2

File Array.cpp

```
25  // Mutator subscript (read + write)
26  double& Array :: operator[ ] (int index)
27  {
28      if (index < 0 || index >= size)
29      {
30          cout << "Index is out of range. Program terminates.";
31          assert (false);
32      }
33      return ptr [index];
34  }
35  // Accessor subscript (read-only)
36  double Array :: operator[ ] (int index) const
37  {
38      if (index < 0 || index >= size)
39      {
40          cout << "Index is out of range. Program terminates.";
41          assert (false);
42      }
43      return ptr [index];
44  }
```

Application File for Array Class

```
1  #include "array.h"
2
3  int main ( )
4  {
5        // Instantiation of array object with three elements
6        Array arr (3);
7        // Using mutator operator [ ]
8        arr[0] = 22.31;
9        arr[1] = 78.61;
10       arr[2] = 65.22;
11       for (int i = 0; i < 3; i++)
12       {
13           cout << "Value of arr [" << i <<"]: " << arr[i] << endl;
14       }
15       // Using accessor operator [ ]
16       cosnt Array arr2 (arr);
17       for (int i = 0; i < 3; i++)
18       {
19           cout << "Value of arr2 [" << i <<"]: " << arr2[i] << endl;
20       }
21       return 0;
```

```
Run:
Value of arr [0]: 22.31
Value of arr [1]: 78.61
Value of arr [2]: 65.22
Value of arr2 [0]: 22.31
Value of arr2 [1]: 78.61
Value of arr2 [2]: 65.22
```

Function Call Operator

Another unary operator that can be overloaded is the function call operator as shown below:

```
name(list of arguments)
```

The overloading of this operator allows us to create a *function object* (sometimes called a *functor*): an instance of an object that acts like a function.

If a class overloads this operator, the compiler allows us to instantiate an object of this class as though we are calling a function.

The difference between a function object and a function is that the function object can hold its state (a function has no state).

For example, a function object called *Smallest* class can hold the smallest value from the previous call (as a data member).

Interface File for the Smallest Class

File smallest.h

```
1  /*************************************************************
2   * The implementation file for the Smallest class            *
3   *************************************************************/
4  #ifndef SMALLEST_H
5  #define SMALLEST_H
6  #include <iostream>
7  using namespace std;
8
9  class Smallest
10 {
11      private:
12          int current;
13      public:
14          Smallest ( );
15          int operator ( ) (int next); // function call operator
16 };
17 #endif
```

Implementation File for the Smallest Class

File smallest.cpp

```
1  /*************************************************************
2   * The implementation file for the Smallest class            *
3   *************************************************************/
4  #include "smallest.h"
5
6  // Constructor
7  Smallest :: Smallest ( )
8  {
9       current = numeric_limits <int> :: max();
10 }
11 // Overloaded function call operator
12 int Smallest :: operator() (int next)
13 {
14      if (next < current)
15      {
16          current = next;
17      }
18      return current;
19 }
```

Application File to Test the Smallest Class

```
1  /*************************************************************
2   * The application file to test the Smallest class           *
3   *************************************************************/
4  #include "smallest.h"
5  #include <iostream>
6
7  int main ( )
8  {
9       // Instantiation of an smallest object
0       Smallest smallest;
11      // Applying the function call operator to objects
12      cout << "Smallest so far: " << smallest (5) << endl;
13      cout << "Smallest so far: " << smallest (9) << endl;
14      cout << "Smallest so far: " << smallest (4) << endl;
15      return 0;
16 }
```

```
Run:
Smallest so far: 5
Smallest so far: 5
Smallest so far: 4
```

KU KONKUK UNIVERSITY

Operator Overloading as Non-Member

OVERLOADING AS NON-MEMBER PART 1

When we overload a binary operator as a member function, one of the operands needs to be the host object.

This is fine when each operand has a different role in the operation.

However, in some operators such as (a + b) or (a < b), the two operands play the same role and neither of them is related to the result.

In these cases, it is better to use a non-member function.

We have two choices: global functions or *friend* functions.

OVERLOADING AS NON-MEMBER PART 2

We have two choices: global functions or *friend* functions.

There is nothing to prevent us from using a global function for overloading binary operators. However, there is one drawback.

The definition of the operator function is longer and more complicated because it needs to use the accessor and mutator functions of the class to access data members of the class.

So how to access the private date members of the class in a global function? We do not develop these types of functions.

C++ allows functions to be declared as *friend* functions of the class.

A friend function has no host object, but it is granted friendship so that it can access the private data members and member functions of the class without calling the public member functions.

We use friend functions to overload selected operators.

Three Types of Friend Functions

1. Declaration of a global function as a friend of a class.

```
class Rect {
  ...
  friend bool equals(Rect r, Rect s);
};
```

2. Declaration of a member function of another class as a friend of a class.

```
class Rect {
  ............
  friend bool RectManager::equals(Rect r, Rect s);
};
```

3. Declaration of all member functions of another class as friends of a class.

```
class Rect {
  ............
  friend RectManager;
};
```

1. A global function as a friend of a class

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

class Rect;
bool equals(Rect r, Rect s);

class Rect {
    int width, height;
public:
    Rect(int width, int height)  {    this->width = width; this->height = height;}
    friend bool equals(Rect r, Rect s);
};

bool equals(Rect r, Rect s) {
    if(r.width == s.width && r.height == s.height) return true;
    else return false;
}

int main() {
    Rect a(3,4), b(4,5);
    if(equals(a, b)) cout << "equal" << endl;
    else cout << "not equal" << endl;
}
```

Forward declaration of ***Rect*** class to prevent the compile error.

Declaration of the ***equals***() as a friend function.

equals() can access the private data members (width, height) of Rect.

Output:

```
not equal
```

2. A member function of another class as a friend of a class

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

class Rect;

class RectManager {
public:
    bool equals(Rect r, Rect s);
};

class Rect {
    int width, height;
public:
    Rect(int width, int height) { this->width = width; this->height = height; }
    friend bool RectManager::equals(Rect r, Rect s);
};

bool RectManager::equals(Rect r, Rect s) {
    if(r.width == s.width && r.height == s.height) return true;
    else return false;
}

int main() {
    Rect a(3,4), b(3,4);
    RectManager man;

    if(man.equals(a, b)) cout << "equal" << endl;
    else cout << "not equal" << endl;
}
```

Forward declaration of ***Rect*** class to prevent the compile error.

Declaration of the ***equals***() of ***RectManager*** class as a friend function

The member function ***equals***() can access the private data members (width, height) of Rect.

Output:

```
equal
```

3. All member functions of another class as friends of a class

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

class Rect;

class RectManager {
public:
    bool equals(Rect r, Rect s);
    void copy(Rect& dest, Rect& src);
};

class Rect {
    int width, height;
public:
    Rect(int width, int height)  { this->width = width; this->height = height; }
    friend RectManager;
};

bool RectManager::equals(Rect r, Rect s) {
    if(r.width == s.width && r.height == s.height) return true;
    else return false;
}

void RectManager::copy(Rect& dest, Rect& src) {
    dest.width = src.width;  dest.height = src.height;
}
```

Forward declaration of ***Rect*** class to prevent the compile error.

Declaration of all the member functions of ***RectManager*** class as a friend function

The all member functions can access the private data members (width, height) of Rect.

```
int main() {
    Rect a(3,4), b(5,6);
    RectManager man;

    man.copy(b, a);
    if(man.equals(a, b)) cout << "equal" << endl;
    else cout << "not equal" << endl;
}
```

Copy a's width and height to b

Output:

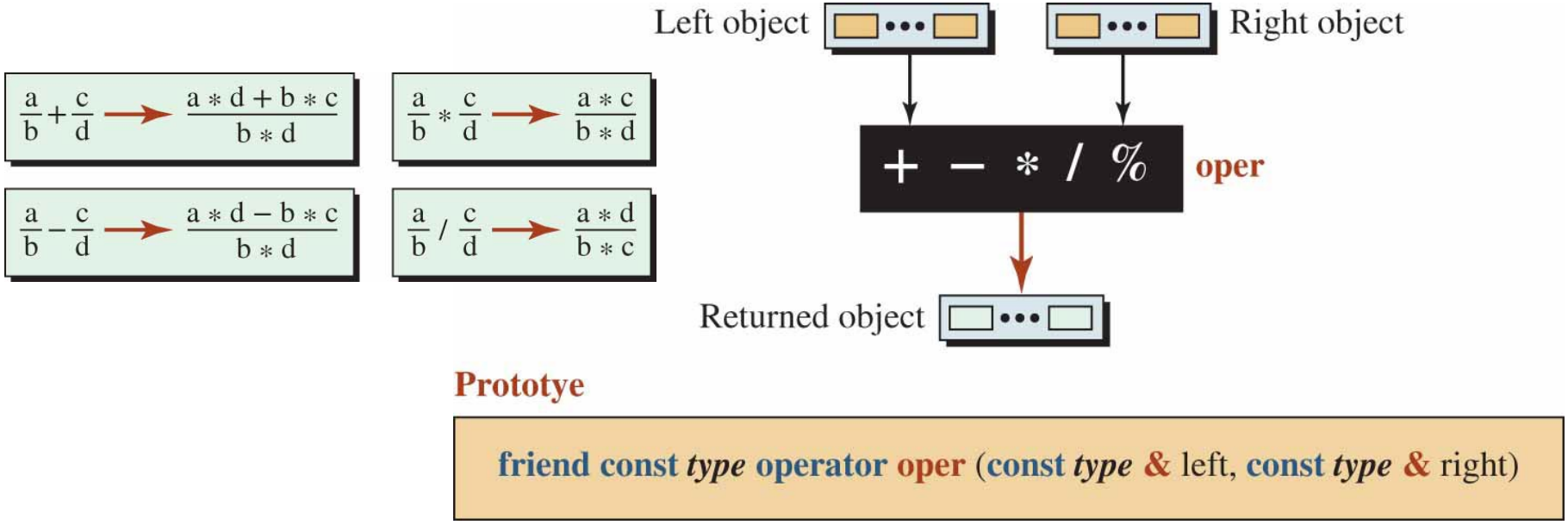
```
equal
```

KU KONKUK UNIVERSITY

Operator Overloading using Friend Functions

Binary Arithmetic Operators Part 1

Overloading the binary arithmetic operators



Prototye

```
friend const type operator oper (const type & left, const type & right)
```

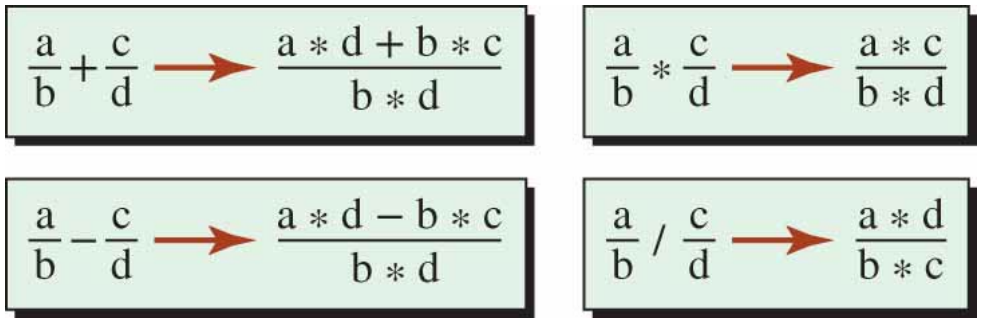
The two operands must already exist.

We create a new object inside the function and return it as a constant object.

We can pass the two operands as reference, but the return object cannot be a reference type because it is created inside the function definition.

Binary Arithmetic Operators Part 2

Binary arithmetic operators for the Fraction class

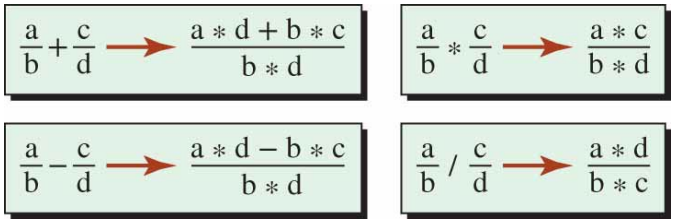



```
// Declarations of addition operator
friend const Fraction operator+(const Fraction& left,
                                const Fraction& right);
// Definition of addition operator
const Fraction operator+(const Fraction& left,
                         const Fraction& right)
{
    int newNumer = left.numer * right.denom +
                   right.numer* left.denom;
    int newDenom = left.denom * right.denom;
    Fraction result(newNumer, newDenom);
    return result;
}
```

In-Class Exercise #1

Binary Arithmetic Operators

Let's modify the Fraction class with the following application code. Please use *Friend* functions instead of member functions.




```
int main()
{
   // Creation of two new objects and testing friend arithmetic operations
   Fraction fract11(1, 2);
   Fraction fract12(3, 4);
   cout << "fract11: " << fract11.print() << endl;
   cout << "fract12: " << fract12.print() << endl;
   Fraction fract111 = fract11 + fract12;
   Fraction fract112 = fract11 - fract12;
   Fraction fract113 = fract11 * fract12;
   Fraction fract114 = fract11 / fract12;
   cout << "fract11 + fract12 : " << fract111.print() << endl;
   cout << "fract11 - fract12 : " << fract112.print() << endl;
   cout << "fract11 * fract12 : " << fract113.print() << endl;
   cout << "fract11 / fract12 : " << fract114.print() << endl << endl;
   return 0;
}
```

```
fract11: 1/2
fract12: 3/4
fract11 + fract12 : 5/4
fract11 - fract12 : -1/4
fract11 * fract12 : 3/8
fract11 / fract12 : 2/3
```

Equality and Relational Operators Part 1

We can also overload the two equality and four relational operators using *friend* functions.

The structure of the operator function is the same as the arithmetic operators except that the return value is of type Boolean.

The two operands must already exist.

Overloading the equality and relational operators

$\frac{a}{b} == \frac{c}{d} \rightarrow a * d == b * c$

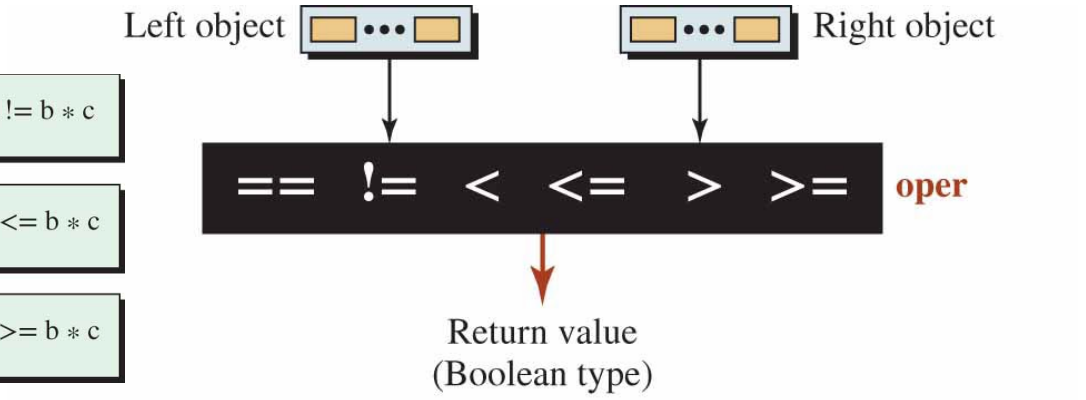
$\frac{a}{b} != \frac{c}{d} \rightarrow a * d != b * c$

$\frac{a}{b} < \frac{c}{d} \rightarrow a * d < b * c$

$\frac{a}{b} <= \frac{c}{d} \rightarrow a * d <= b * c$

$\frac{a}{b} > \frac{c}{d} \rightarrow a * d > b * c$

$\frac{a}{b} >= \frac{c}{d} \rightarrow a * d >= b * c$

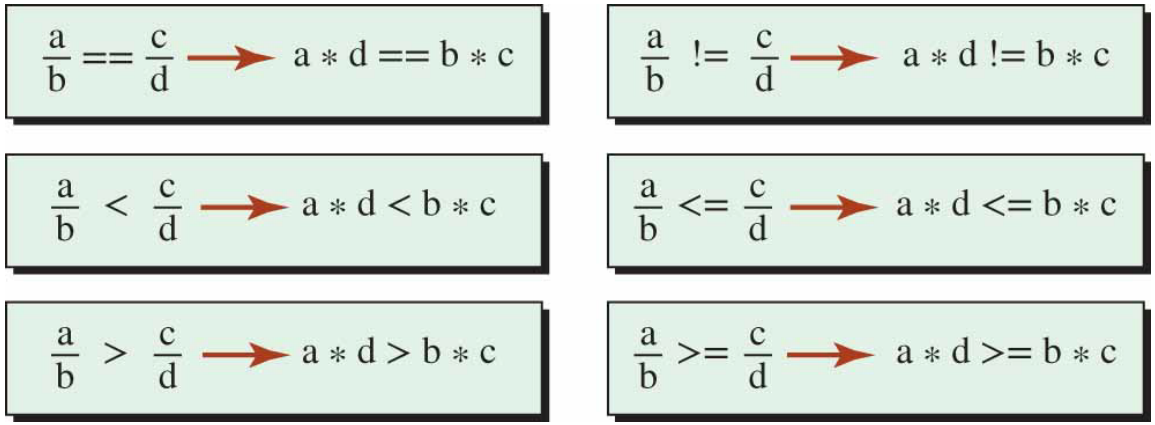


Prototye

```
friend bool operator oper (const type & left, const type & right)
```

Equality and Relational Operators Part 2

Equality and relational operators for Fraction

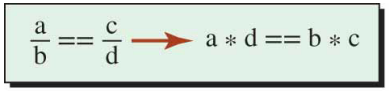


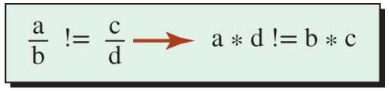

```
// Declaration of equality operator
friend  bool   operator = =(const Fraction& left,
                            const Fraction& right);
// Definition of equality operator
bool operator = =(const Fraction& left,
                  const Fraction& right)
{
    return (left.numer * right.denom = =
                    right.numer * left.denom);
}
```

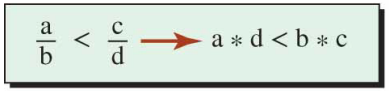
In-Class Exercise #2

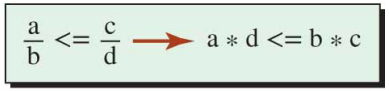
Equality and Relational Operators

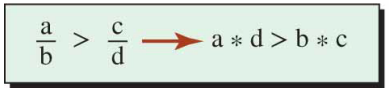
Let's modify the Fraction class with the following application code. Please use *Friend* functions instead of member functions.

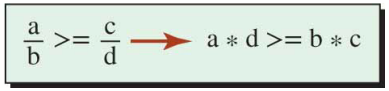













```
int main()
{
   // Creation of two new objects and testing relational operators
   Fraction fract13(2, 3);
   Fraction fract14(1, 3);
   cout << "fract13: " << fract13.print() << endl;
   cout << "fract14: " << fract14.print() << endl;
   cout << "fract13 == fract14: " << boolalpha;
   cout << (fract13 == fract14) << endl;
   cout << "fract13 != fract14: " << boolalpha;
   cout << (fract13 != fract14) << endl;
   cout << "fract13 > fract14: " << boolalpha;
   cout << (fract13 > fract14) << endl;
   cout << "fract13 < fract14: " << boolalpha;
   cout << (fract13 < fract14) << endl << endl;
   return 0;
}
```

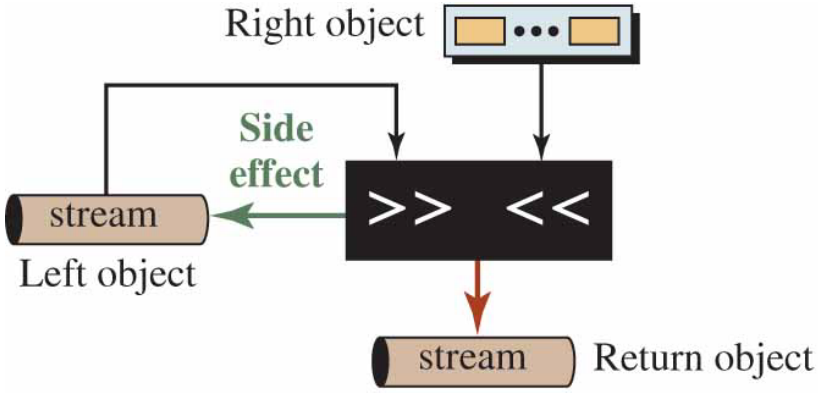
```
fract13: 2/3
fract14: 1/3
fract13 == fract14: false
fract13 != fract14: true
fract13 > fract14: true
fract13 < fract14: false
```

Extraction and Insertion Operators Part 1

The value of a fundamental type can be extracted from an input stream object using the extraction (>>) or inserted into an output stream using the insertion operator (<<).

We can overload these two operators for our class types as shown in the Figure.

Overloading insertion or extraction operators



Note:
The >> operator changes the right object, which means it cannot be passed as constant reference.

Prototyes

```
friend istream & operator >> (istream & left, type & right)
friend ostream & operator << (ostream & left, const type & right)
```

Each of these operators is a binary operator, but the left operand is an object of the *istream* class in the case of the extraction operator and the object of the *ostream* class in the case of the insertion operator.

Extraction and Insertion Operators Part 2

Overloaded extraction and insertion operator

```
// Declaration of >> operator
friend  istream&  operator >>(istream& left, Fraction& right) ;
// Definition of >> operator
istream& operator>>(istream& left, Fraction& right)
{
     cout << "Enter the value of numerator: " ;
     left >> right.numer;
     istream << "Enter the value of denominator: " ;
     left >> right.denom;
     right.normalized();
     return left ;
}
// Definition of << operator
friend  ostream&   operator <<(ostream& left, const Fraction& right);
// Definition of << operator
ostream&  operator << (ostream& left, const Fraction& right)
{
     left << right.numer << "/" << right.denom << endl;
     return left;
}
```

In-Class Exercise #3

Extraction and Insertion Operators

Let's extend the Fraction class with the following application code. Please use *Friend* functions instead of member functions. We will no longer use *print*() member function. Use Insertion operator(<<) instead.

```
int main()
{
   // Testing extraction operator
   Fraction fract15;
   cin >> fract15;
   cout << "fract15: " << fract15 << endl << endl;

   // Creation of two new objects and
   // testing friend arithmetic operations
   Fraction fract16(1, 2);
   Fraction fract17(3, 4);
   cout << "fract16: " << fract16 << endl;
   cout << "fract17: " << fract17 << endl;
   cout << "fract16 + fract17 : " << fract16 + fract17 << endl;
   cout << "fract16 - fract17 : " << fract16 - fract17 << endl;
   cout << "fract16 * fract17 : " << fract16 * fract17 << endl;
   cout << "fract16 / fract17 : " << fract16 / fract17 << endl << endl;
   return 0;
}
```

```
Enter the value of numerator: 2
Enter the value of denominator: 4
fract15: 1/2

fract16: 1/2
fract17: 3/4
fract16 + fract17 : 5/4
fract16 - fract17 : -1/4
fract16 * fract17 : 3/8
fract16 / fract17 : 2/3
```

KU KONKUK UNIVERSITY

Type Conversion

TYPE CONVERSION

When we are working with fundamental data types, we sometimes use mixed types in an expression and expect the system to make them the same type according to the rules of conversion.

For example, in the following expression statement that finds the perimeter of a circle with radius 5, the system does two conversions from integer 5 to double 5.0 and integer 2 to 2.0 to find the perimeter:

```
double perimeter = 2 * 5 * 3.1415;
```

We can do the same, with some limitation, to change a fundamental type to a user-defined type and vice versa.

In this section, we will implement following type conversions:

- **Integer → Fraction**
- **Double → Fraction**
- **Fraction → Double**

Fundamental Type to Class Type Part 1

To change a fundamental type to a class type (when it make sense), we create a new parameter constructor (also referred to as conversion constructor).

For example, we can convert an integer to a fraction and also a real to a fraction.

In both cases, we create a constructor with one parameter.

Converting an Integer to a Fraction

This can be done easily using a constructor that takes one integer parameter that is set as numerator; the denominator is set to 1.

```
// Declaration
Fraction(int n);
// Definition
Fraction :: Fraction(int n)
: numer(n), denom(1)
{
}
```

Fundamental Type to Class Type Part 2

Converting a Real to a Fraction

We can use a constructor to convert a real number to a fraction. We can think about a fraction as a real number. For example, the fraction 7/4 is the real 1.75.

```
// Declaration
Fraction(double value);
// Definition
Fraction :: Fraction(double value)
{
     denom = 1;
     while ((value - static_cast <int>(value)) > 0.0)
     {
          value *= 10.0;
          denom *= 10;
     }
     numer = static_cast <int>(value);
     normalize();
}
```

Class Type to Fundamental Type

Sometimes we need to convert a class type into a fundamental type.

This can be done using a conversion operator, an operator function in which the term operator and the operator symbol are replaced with a fundamental data type as shown below.

```
// Declaration
operator double();
// Definition
Fraction :: operator double()
{
      double num = static_cast <double>(numer);
      return (num / denom);
}
```

The syntax of the operator looks strange (no return value), but we need to think of it as a constructor for the double type.

It takes a fraction and constructs a double value out of it.

We know that a constructor has no return value; it constructs a type.

In the application, we use its function operator instead of using the operator as:

```
fract1.operator double () // func operator
```

```
double(fract1) //operator X work
```

In-Class Exercise #4

Fundamental Type to Class Type

Let's extend the Fraction class with the following application code. Please use conversion constructors.

```
int main()
{
   // Using convertor constructor to create two new objects
   Fraction fract18(5); // Changing an integer to a fraction
   Fraction fract19(23.45); // Changing a double value to a fraction
   cout << "fract18: " << fract18 << endl;
   cout << "fract19: " << fract19 << endl << endl;
   // Changing a fraction to a double
   Fraction fract20(9, 13);
   cout << "double value of fract20 (9, 13): ";
   cout << setprecision(3) << fract20.operator double() << endl << endl;
   return 0;
}
```

```
fract18: 5/1
fract19: 469/20

double value of fract20 (9, 13): 0.692
```

Summary

Binary Operator Overloading

Friend Functions

Operator Overloading using Friend Functions

Type Conversion

KU KONKUK UNIVERSITY

What's Next?

Reading Assignment

- ❑ **Read Chap. 14. Exception Handling**
- ❑ **Read Chap. 15. Generic Programming: Templates**

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr