Object-oriented Programming Operator Overloading Part 1

YoungWoon Cha

Computer Science and Engineering

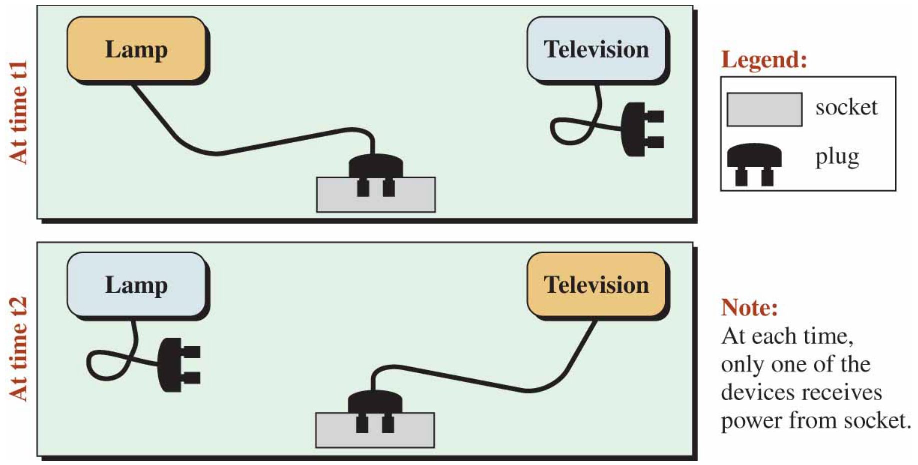
Review

Polymorphic variables

In C++, a pointer or a reference can play the role of a socket that once created can accept plug-compatible objects.

We can define a pointer (or a reference) that can point to the base class; we can then let the pointer point to any object in the hierarchy.

For this reason, the pointer and reference variables are sometimes referred to as *polymorphic variables*.



Note:
At each time, only one of the devices receives power from socket.

```
// Creation of a pointer to the Base class (simulating socket)
Base* ptr;
// Let ptr points to an object of the Base class
ptr = new Base ();
ptr -> print();
delete ptr;
// Let ptr points to an object of the Derived class
ptr = new Derived();
ptr -> print();
delete ptr;
return 0;
```

For polymorphism, we need pointers (or references), we need exchangeable objects, and we need virtual functions.

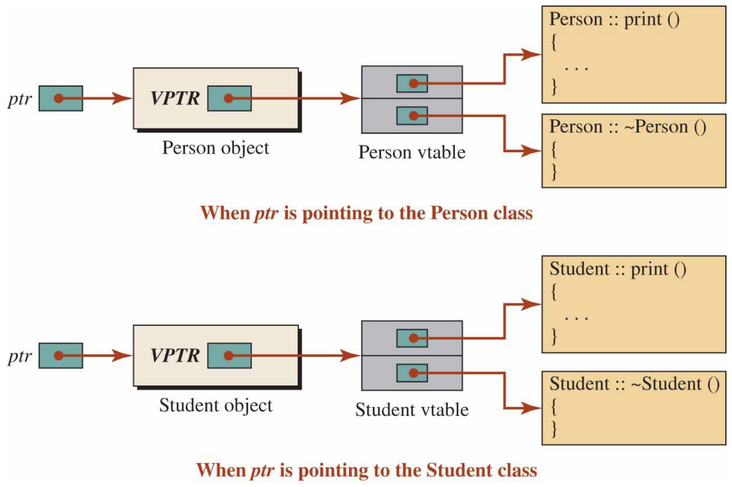
Constructors and Destructors

Constructors and destructors in a class hierarchy are also member functions although special ones. We discuss them separately.

Constructor Cannot Be Virtual

Constructors cannot be virtual because although constructors are member function, the names of the constructors are different for the base and derived classes (different signatures).

Virtual Destructor



Although the name of the destructors are also different in the base and derived class, the destructors are normally not called by their name.

When there is a virtual member function anywhere in the design, we should make the destructors also virtual to avoid memory leaks.

Using Dynamic-Cast Operator

We have seen that in a polymorphic relationship we can *upcast* a pointer, which means to make a pointer to a derived class to be assigned to a pointer to a base class as shown below:

```
Person*  ptr1 = new Student
```

Here the pointer returned from the *new* operator is a pointer to a *Student* object, but we assign it to a pointer that points to a *Person* object (the pointer is upcast).

C++ also allows us to downcast a pointer to make it to point to an object in the lower order of hierarchy.

This can be done using a *dynamic_cast* operator as shown below (*ptr*1 is a pointer to *Person* object).

```
Student*  ptr2 = dynamic_cast <Student*>(ptr1);
```

This casting proves that the *Student* class is a class derived from the *Person* class because *ptr1* can be downcast to *ptr2*.

However it is not recommended because of the overhead involved.

Abstract Classes and Interfaces

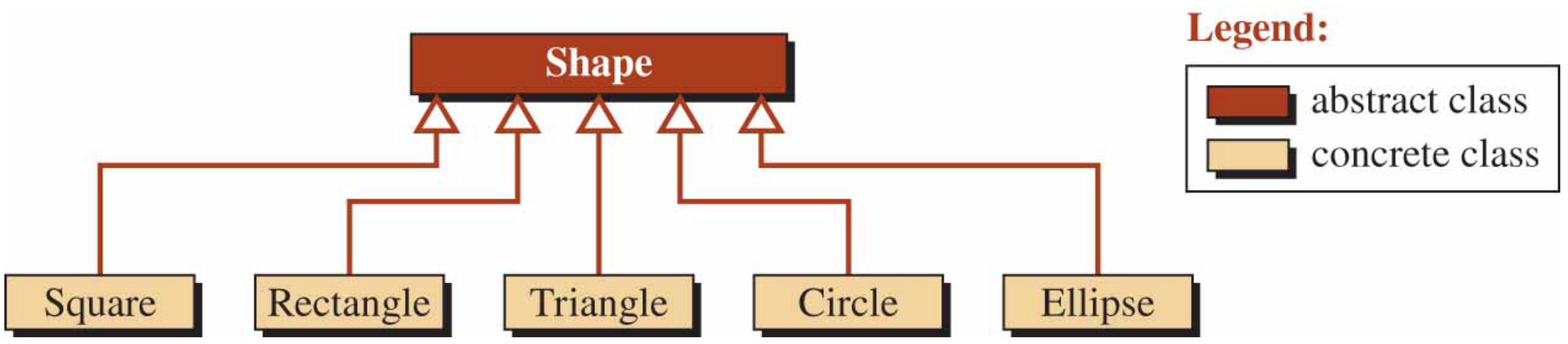
Declaration of Pure Virtual Functions

An *abstract class* is a class with at least one *pure virtual function*.

A *pure virtual function* is a virtual function in which the declaration is set to zero and which has no definition in the abstract class.

```
virtual double getArea(0) = 0;
virtual double getPerimeter(0) = 0;
```

An interface is a special case of an abstract class in which all member functions are pure virtual.

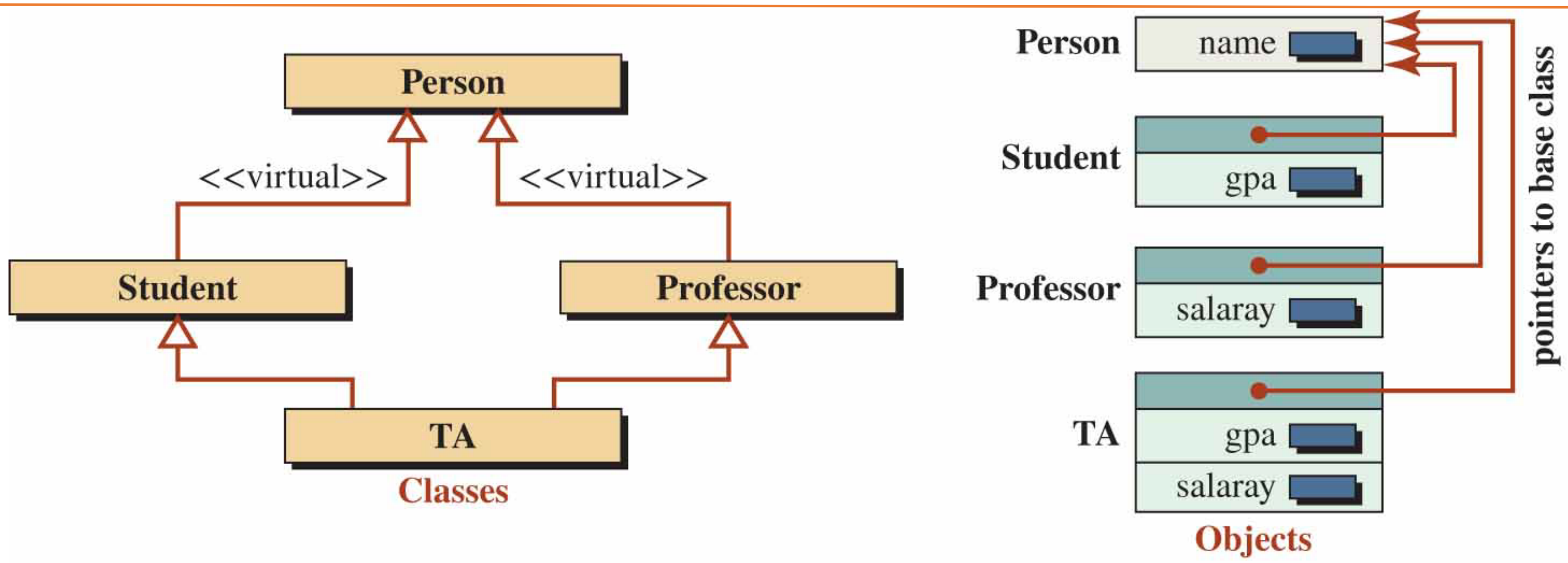


Multiple Inheritance Using Virtual Base

One solution for the problem of duplicated shared data members in multiple inheritance is to use *virtual base inheritance*.

In this type of inheritance, two classes can inherit from a common base using the *virtual* keyword.

Classes and objects in virtual base inheritance



In this case, we have the following four classes.

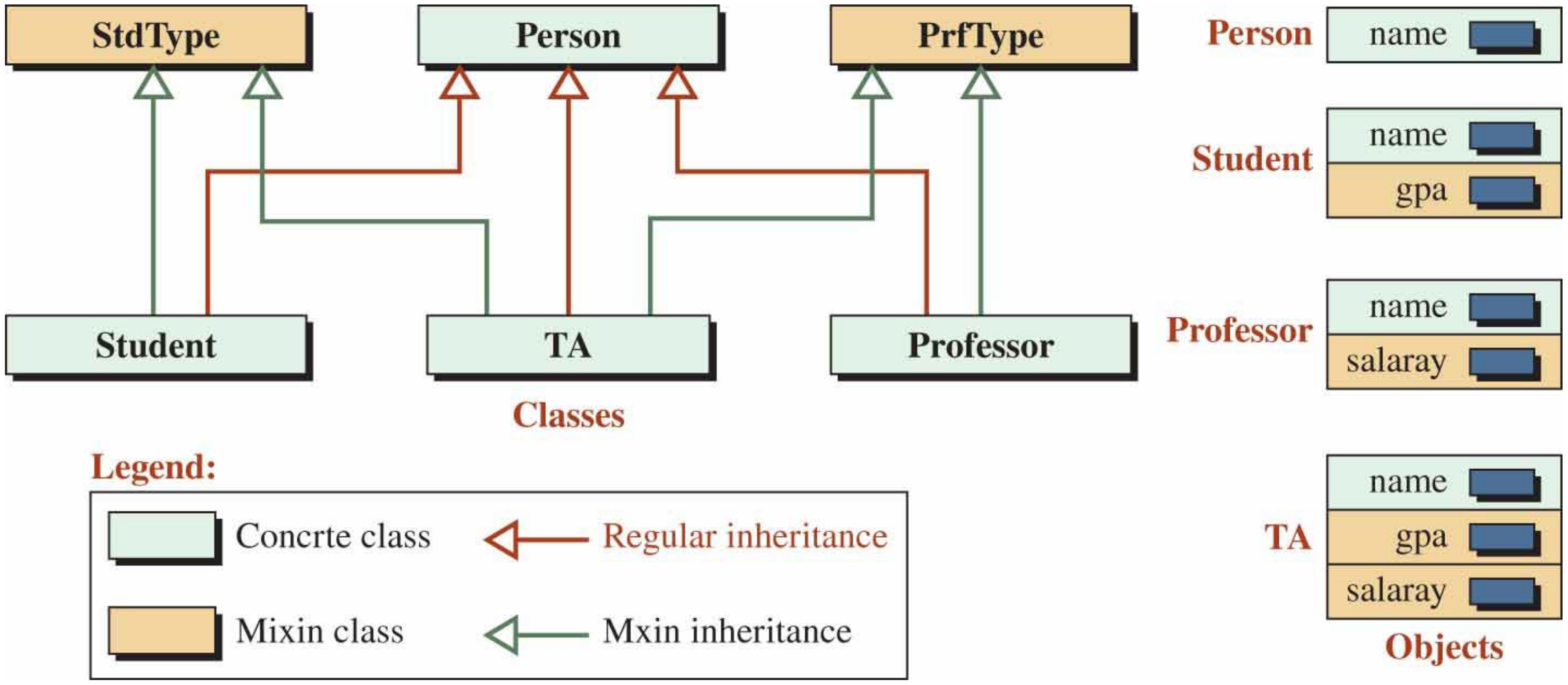
```
class Person {…};
class Student: virtual public Person {…};
class Professor: virtual public Person {…};
class TA: public Student, public Professor {…};
```

Multiple Inheritance Using Mixin Classes

Another solution for the problem of common base classes in multiple inheritance is the use of mixin classes.

A mixin class is never instantiated (it has some pure virtual functions), but it can add data members to other classes.

Multiple inheritance using mixin classes



Three Roles of An Object

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();
```

Host Object Part 1

When we define a non-static member function for a class, the function needs to be called through an instance of the class.

```
void Fun :: functionOne(...)
{
     ...
}

int main()
{
     fun1.functionOne(...); // fun1 is object affected by functionOne
     fun2.functionOne(...); // fun2 is object affected by functionOne
     fun3.functionOne(...); // fun3 is object affected by functionOne
     ...
}
```

The member function *functionOne* operates on *fun*1, in the first call, on *fun*2 in the second call, and on *fun*3 on the third call.

The object the function operates on in each case is called the *host object*.

Host Object Part 2

```
void Fun :: functionOne(...)
{
      ...
}

int main()
{
      fun1.functionOne(...); // fun1 is object affected by functionOne
      fun2.functionOne(...); // fun2 is object affected by functionOne
      fun3.functionOne(...); // fun3 is object affected by functionOne

      ...
}
```

People who have written instance member functions may be wondering that there were no reference to the *host object* inside the body of the function.

The definition of the function cannot include the name of the host object because it is changing all of the time.

The member function accesses the host object indirectly through the *this* pointer.

The host object is the one that is pointed to by the *this* pointer.

Host Object Part 3

Assume that the Fun class has only one integer member function named ***num***.

In the Table, the left section shows how we normally define the function; the right section shows how the compiler changes it to access the host object pointed to by the *this* pointer.

```cpp
// Written by the user
void Fun :: multiplyByTwo()
{
    cout << num * 2;
}
```

```cpp
// Changed by the compiler
void Fun :: multiplyByTwo()
{
    cout << (this -> num) * 2;
}
```

We can even directly use the code at the right-hand side to define this function, but it is easier to write it as shown in the left-hand side.

In other words, the host object is always the object pointed to by the *this* pointer.

Host Object Part 4

Protection

The nature of some member functions requires change in the host object; some other functions should not change the host object.

To prevent a member function from changing the host object, we need to show that we want the host object to be constant.

Since the host object is invisible in the member function, we need to use the *const* modifier at the end of the function header to warn the compiler that we do not want the host object to be changed.

This needs to be done both in the function declaration and function definition, but the function call is the same in both cases.

```
// The host object can be changed

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

```
// The host object cannot be changed

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

Parameter Objects

Three Ways to Pass

A parameter object is different from a host object. The host object is the hidden part of the member function; a parameter object needs to be passed to the member function.

Pass-by-value is normally not used when the parameter is an object of a user-defined type because it involves calling the copy constructor, making a copy of the object, and then passing it to the function. It is very inefficient.

Pass-by-reference is the most common method we encounter in practice. We do not copy the object, we just define an alias name in the function header to be able to access the object. The function can use this name to access the original object and operate on it. There is no cost of copying.

Pass-by-pointer, is not very common unless we already have a pointer pointed to the object (such as the case when the object created in the heap) and we pass the pointer to the function.

The most common way to pass a user-defined object to a function is pass-by-reference.

Parameter Objects

Protection

Since we normally use pass-by-reference to pass an object of user-defined type to a function, the function can change the original object.

If we want to prevent this change (most of the time), we should insert the *const* modifier in front of the parameter, which means that the function cannot change the original object.

```
The parameter cannot be changed
// Declaration
void one(Type& para);
// Definition
void Fun :: two(Type& para)
{
     ... ;
}
// Call
fun1.one(para);
```

```
// The parameter cannot be changed
// Declaration
void two(const Type& para);
// Definition
void Fun :: two(const Type& para)
{
     ... ;
}
// Call
fun1.two(para);
```

Returned Objects

Three Ways to Return. **A constructor and the destructor do not return an object, they create or destroy the host object. Other functions may return (an instance of) an object.**

Return-by-value. When we return an object by value, the function needs to call the copy constructor, create a copy of object, and return the copy. (this is expensive)

We have to return by value if the returned object is created in the body of the function.

Return-by-reference eliminates the cost of copying, but it is not possible to use it when the object is created inside the body of the function. When the function terminates, the object is destroyed and we cannot make a reference to a destroyed object.

It is possible when the object to be returned is a parameter object passed by reference (or pointer). The origin of an object exists even the object is terminated. We can make a reference to it.

Return-by-pointer has the same limitation and advantage of pass-by-reference but it is seldom used unless the original object is created in the heap.

We have to return by reference or by pointer if the returned object is passed as the parameter to the function.

Returned Objects

Protection: **If the returned object is to be used only as an *rvalue*, it should be protected from change by using the *const* specifier.**

On the other hand, if the returned object is to be used as an *lvalue*, there should not be a *const* specifier.

Two types of return-by-value

```
// The object can be changed
// Declaration
Fun one(int value);
// Definition
Fun Fun :: one(int value)
{
       ... ;
}
// Call
fun1.one(value) = ...;
```

```
// The object cannot be changed
// Declaration
const Fun functOne(int value);
// Definition
const Fun Fun :: two(int value)
{
       ... ;
}
// Call
fun1.two(value);
```

Two examples of return-by-reference

```
// The object can be changed
// Declaration
Fun& one();
// Definition
Fun& Fun :: one()
{
       ...;
}
// Call
fun1.one() = ...;
```

```
//The object cannot be changed
// Declaration
const Fun& two();
// Definition
const Fun& Fun :: two()
{
       ... ;
}
// Call
fun1.two();
```

Operator Overloading

OVERLOADING PRINCIPLES

C++ uses a set of symbols called operators to manipulate fundamental data types.

Most of these symbols are overloaded to handle several data types.

For example, the symbol for the addition operator can be used to add two values of type *int*, *long*, *long long*, *double*, and *long double*.

This means that the following two expressions in C++ use the same symbol with two distinct interpretations.

```
14 + 20                    14.21 + 20.45
```

OVERLOADING PRINCIPLES (Continued)

C++ language goes one step further when it uses the symbol << to mean two different things when applied to an integer data type and the *ostream* class.

The following two expressions give two different interpretations for this symbol.

```
x << 5;                    cout << 5;
```

In the left expression, the symbol << means to shift the bits in the integer object five positions to the left; in the right expression it means apply the insertion operator on the *cout* object to print the value of the fundamental data type 5.

OVERLOADING PRINCIPLES (Continued)

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
```

Three Categories of Operators

We can divide the operators in C++ into three categories: *non-overloadable*, *not-recommended for overloading*, and *overloadable* as shown in Table 10.1.

Table 10.1 *Operators in C++ and their overloadability*

Operator	Arity	Name	Overloadability
`::`	primary	scope	Non-overloadable
`[ ]`	postfix	array subscript	Overloadable
`( )`	postfix	function call	Overloadable
`.`	postfix	member selector	Non-overloadable
`->`	postfix	member selector	Overloadable
`++`	postfix	postfix incremen	Overloadable
`--`	postfix	postfix decremen	Overloadable
`++`	prefix	prefix increment	Overloadable
`--`	prefix	prefix decrement	Overloadable
`~`	unary	bitwise not	Overloadable
`!`	unary	logical not	Overloadable
`+`	unary	plus	Overloadable
`-`	unary	minus	Overloadable
`*`	unary	dereference	Overloadable

Three Categories of Operators Part 1

Table 10.1 *Continued*

Operator	Arity	Name	Overloadability
&	unary	address-of	Not recommended
new	unary	allocate object	Overloadable
new [ ]	unary	allocate array	Overloadable
delete	unary	delete object	Overloadable
delete[ ]	unary	delete array	Overloadable
type	unary	Type conversion	Overloadable
.*	unary	ptr to member	Non-overloadable
-> *	unary	ptr to member	Overloadable
*	binary	multiply	Overloadable
/	binary	divide	Overloadable
%	binary	modulo (remainder)	Overloadable
+	binary	add	Overloadable
-	binary	subtract	Overloadable
<<	binary	bitwise shift left	Overloadable
>>	binary	bitwise shift right	Overloadable
<	binary	less than	Overloadable

Three Categories of Operators Part 2

Table 10.1 *Continued*

Operator	Arity	Name	Overloadability
<=	binary	less or equal	Overloadable
>	binary	greater	Overloadable
>=	binary	greater or equal	Overloadable
==	binary	equal	Overloadable
!=	binary	not equal	Overloadable
&	binary	bitwise and	Not recommended
^	binary	bitwise ex or	Overloadable
|	binary	bitwise or	Not recommended
&&	binary	logical and	Not recommended
||	binary	logical or	Not recommended
? :	binary	conditional	Not overloadable
=	binary	simple assignment	Overloadable
oper=	binary	comp. assignment	Overloadable

Non-Overloadable Operators

The C++ language does not allow us to overload the operators marked as not overloadable in Table 10.1.

Not Recommended for Overloading

There are five operators that can be overloaded, but C++ strongly recommend that we do not overload them.

The reason for not recommendation is that C++ defines some special meaning and some procedures for overloading that the user may not be able to fulfill.

Overloadable Operators

The rest of operators as defined in Table 13.7 can be overloaded and it is guaranteed that the overloading follows the natural behavior of the operators as defined in C++.

We do not discuss all of them here; we discuss only the most common ones.

Rules of Overloading

Before we try to overload some operators for our user-defined data types, we need to be aware of the rules and restrictions about overloading:

- ❑ ***Precedence.*** **Overloading cannot change the precedence (order) of the operator. e.g. c + a*b**
- ❑ ***Associativity.*** **Overloading cannot change the associativity of the operator. e.g. (a-b)*c ≠ a-(b*c)**
- ❑ ***Commutativity.*** **Overloading cannot change the commutativity of the operator. e.g. a-b ≠ b-a**
- ❑ ***Arity.*** **Overloading cannot change the arity(the number of terms) of the operator. e.g. a++ ≠ a++b**
- ❑ ***No New operators.*** **We cannot invent operators; we can only overload the existing overloadable operators. e.g. $, #, @**
- ❑ ***No Combination.*** **We cannot combine two operator symbols to create a new one. e.g. +^***

Operator Function

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 name of this function starts with the reserved word *operator* and is followed by the symbol of the operator that we need to overload.

The Figure shows the general form of an operator function prototype.

$$\textit{return\_type}\ \underbrace{\textbf{operator symbol}}_{\textit{function}\ \text{name}}\ \text{(parameter lists)}$$

Member versus Nonmember Functions

Most of the overloadable operators can be defined either as a member function or a non-member function.

A few can be overloaded only as member functions; a few needs to be overloaded only as non-member functions.

Since the syntax for member and non-member operator functions is different, we discuss them separately.

Using Operator or Operator Function

After overloading, we can either use the operator itself or the function operator.

For example, imagine we have overloaded the unary minus operator for our Fraction class as a member function, we can then apply the operator on a fraction object or invoke the fraction operator as shown below.

The left version is more concise and intuitive.

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
```

Code Example #1

Fraction Class

A *fraction* is a ratio of two integers such as 3/4, 1/2, 7/5, and so on.
In C++, there is no built-in type that can represent a fraction.
For the rest of the *operator overloading* exercises, we will use the following Fraction class codes.

Fraction Class Part 1

```
/*****************************************************************
* Object-Oriented Programming, Spring, 2023
* Fraction Class
*
* Written by YoungWoon Cha (youngcha@konkuk.ac.kr)
* April, 2023
*****************************************************************/
#include <iostream>
#include <string>
using namespace std;

class Fraction
{
private:
   int numerator;
   int denominator;

public:
   Fraction();
   Fraction (int numer, int denom);
   Fraction(const Fraction& fract);
   ~Fraction();

   int getNumerator() const {return numerator; }
   int getDenominator() const { return denominator; }
   void setNumerator(int numer);
   void setDenominator(int denom);
   string print();

private:
   bool normalize();
   int gcd(int n, int m);
};
```

Fraction Class Part 2

```
Fraction::Fraction()
: numerator(0), denominator(1)
{
}

Fraction::Fraction(int numor, int denom = 1)
: numerator(numor), denominator(denom)
{
   normalize();
}

Fraction::Fraction(const Fraction& fract)
: numerator(fract.numerator), denominator(fract.denominator)
{
}

Fraction :: ~Fraction()
{
}

string Fraction::print() const
{
   return to_string(numerator) + "/" + to_string(denominator);
}

void Fraction::setNumerator(int numer)
{
   numerator = numer;
   normalize();
}

void Fraction::setDenominator(int denom)
{
   denominator = denom;
   normalize();
}
```

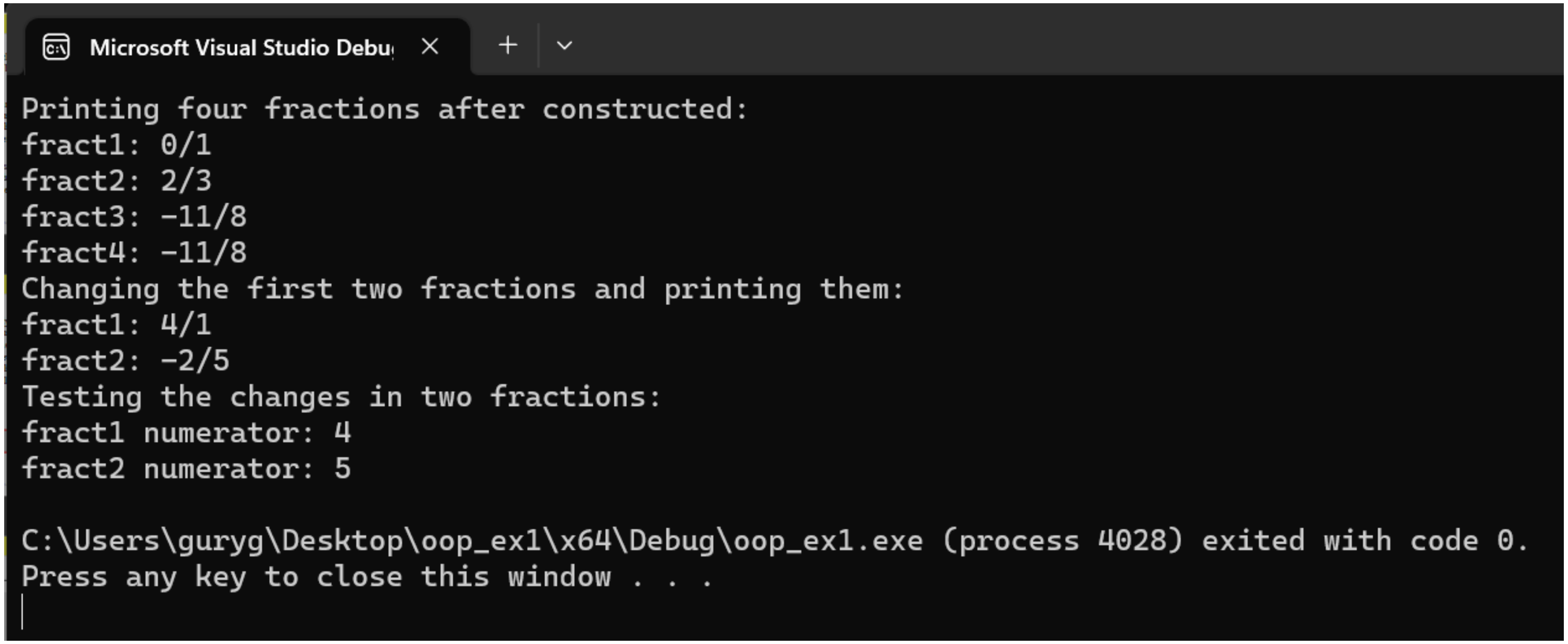
Fraction Class Part 3

```cpp
bool Fraction::normalize()
{
   // Handling a denominator of zero
   if (denominator == 0)
   {
      cout << "Invalid denomination. Need to quit." << endl;
      return false;
   }
   // Changing the sign of denominator
   if (denominator < 0)
   {
      denominator = -denominator;
      numerator = -numerator;
   }
   // Dividing numerator and denominator by gcd
   int divisor = gcd(abs(numerator), abs(denominator));
   numerator = numerator / divisor;
   denominator = denominator / divisor;
   return true;
}

int Fraction::gcd(int n, int m)
{
   int gcd = 1;
   for (int k = 1; k <= n && k <= m; k++)
   {
      if (n % k == 0 && m % k == 0)
      {
         gcd = k;
      }
   }
   return gcd;
}
```

Fraction Class Part 4

```cpp
int main()
{
   // Instantiation of some objects
   Fraction fract1;
   Fraction fract2(14, 21);
   Fraction fract3(11, -8);
   Fraction fract4(fract3);

   // Printing the object
   cout << "Printing four fractions after constructed: " << endl;
   cout << "fract1: " << fract1.print() << endl;
   cout << "fract2: " << fract2.print() << endl;
   cout << "fract3: " << fract3.print() << endl;
   cout << "fract4: " << fract4.print() << endl;

   // Using mutators
   cout << "Changing the first two fractions and printing them:" << endl;
   fract1.setNumerator(4);
   cout << "fract1: " << fract1.print() << endl;
   fract2.setDenominator(-5);
   cout << "fract2: " << fract2.print() << endl;

   // Using accessors
   cout << "Testing the changes in two fractions:" << endl;
   cout << "fract1 numerator: " << fract1.getNumerator() << endl;
   cout << "fract2 numerator: " << fract2.getDenominator() << endl;

   return 0;
}
```

Fraction Class Part 5



```
Printing four fractions after constructed:
fract1: 0/1
fract2: 2/3
fract3: -11/8
fract4: -11/8
Changing the first two fractions and printing them:
fract1: 4/1
fract2: -2/5
Testing the changes in two fractions:
fract1 numerator: 4
fract2 numerator: 5

C:\Users\guryg\Desktop\oop_ex1\x64\Debug\oop_ex1.exe (process 4028) exited with code 0.
Press any key to close this window . . .
```

Unary Operator Overloading

Lvalue, Rvalue

A *lvalue*(at the left-hand side) can be thought as the destination of a value.
= Stored value. e.g. a variable.
It is accessible after the expression.
The memory location can be identified.

A *rvalue*(at the right-hand side) can be thought as the source of a value.
= Temporary value.
It is not accessible after the expression.
The memory location is unknown.

$$\underset{\text{The variable z is a lvalue}}{\underline{\textbf{int z}}} = \underset{\text{13 is a rvalue}}{\underline{\mathbf{10 + 3}}};$$

Side Effect

A side effect is any modification of the state of memory.

```
int z = 10 + 3; // 1 side effect
int y = z++; // 2 side effects
3 + 4; // 0 side effect
```

The left operand is an *lvalue* object that receives the side effect of the operation; the right operand is an *rvalue* object that should not be changed in the process.

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.

$$+\frac{a}{b} \rightarrow +\frac{a}{b}$$

$$-\frac{a}{b} \rightarrow -\frac{a}{b}$$

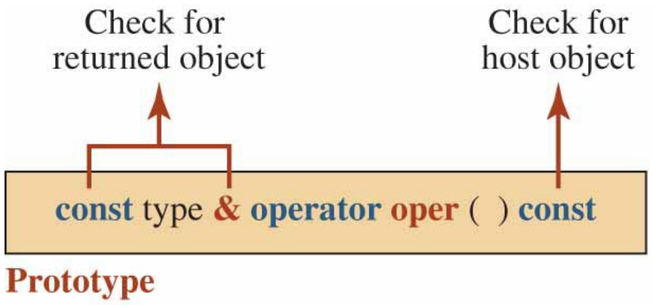
$$++\frac{a}{b} \rightarrow \frac{a}{b}+1 \rightarrow \frac{a+b}{b}$$

$$--\frac{a}{b} \rightarrow \frac{a}{b}-1 \rightarrow \frac{a-b}{b}$$

$$\frac{a}{b}++ \rightarrow \frac{a}{b}+1 \rightarrow \frac{a+b}{b}$$

$$\frac{a}{b}-- \rightarrow \frac{a}{b}-1 \rightarrow \frac{a-b}{b}$$

Guideline for overloading unary operators



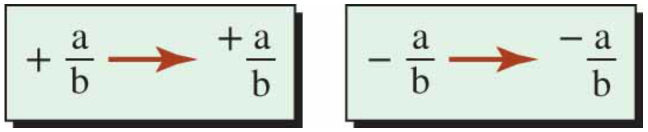
```
const type & operator oper ( ) const
```

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?

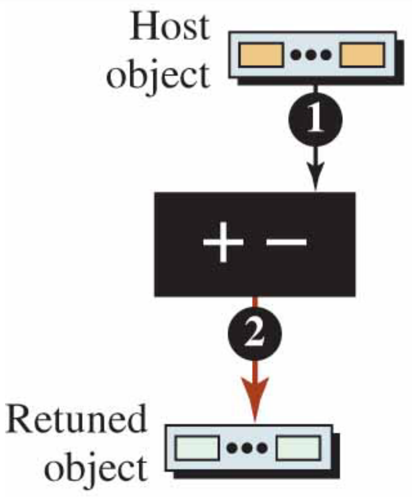
Checklist

Unary Operator Plus or Minus



$+\frac{a}{b} \rightarrow +\frac{a}{b}$
$-\frac{a}{b} \rightarrow -\frac{a}{b}$

We overload the plus and minus operators as shown in the Figure.



Prototye

const type **operator** ± ( ) **const**

Checklist

1. Host object does not change (constant).
2. Returned object
   a. is created as a new object (no reference).
   b. is used as *rvalue* (constant).

```
// Declaration of Unary plus
const Fraction  operator+() const;
// Definition of Unary plus operator
const Fraction   Fraction :: operator+ () const
{
     Fraction temp (+numer, denom); // a new object
     return temp;
}
// Declaration of Unary minus
const Fraction  operator-() const;
// Definition of Unary minus operator
const Fraction   Fraction :: operator- () const
{
     Fraction temp (-numer, denom); // a new object
     return temp;
}
```

```
Fraction fract1(2, 3);
Fraction fract2(1, 2);
Fraction fract3 = +fract1;
Fraction fract4 = -fract2;
```

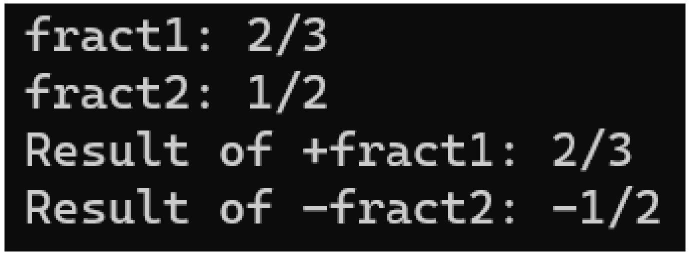
In-Class Exercise #1

Unary Plus and Minus

$$+\frac{a}{b} \rightarrow \frac{+a}{b} \qquad -\frac{a}{b} \rightarrow \frac{-a}{b}$$

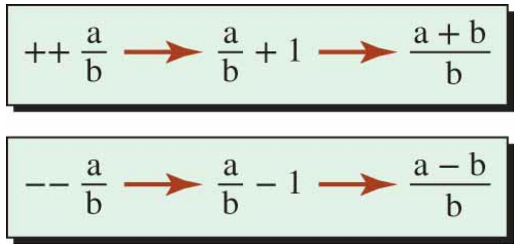
Let’s extend the Fraction class with the following application code.

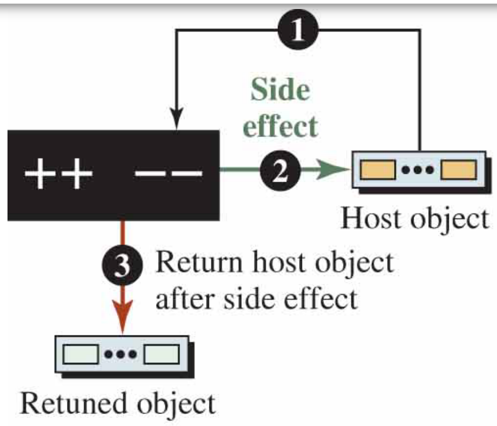
```cpp
int main()
{
    // Creation of two objects and testing the plus and minus operator
    Fraction fract1(2, 3);
    Fraction fract2(1, 2);
    cout << "fract1: " << fract1.print() << endl;
    cout << "fract2: " << fract2.print() << endl;
    Fraction fract3 = +fract1;
    Fraction fract4 = -fract2;
    cout << "Result of +fract1: " << fract3.print() << endl;
    cout << "Result of -fract2: " << fract4.print() << endl << endl;
    return 0;
}
```

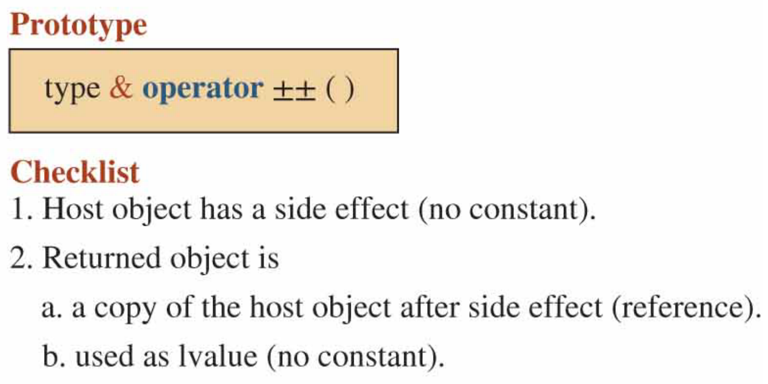


```
fract1: 2/3
fract2: 1/2
Result of +fract1: 2/3
Result of -fract2: -1/2
```

Pre-Increment and Pre-Decrement Operators


$$++\frac{a}{b} \rightarrow \frac{a}{b}+1 \rightarrow \frac{a+b}{b}$$
$$--\frac{a}{b} \rightarrow \frac{a}{b}-1 \rightarrow \frac{a-b}{b}$$




Prototype

type & operator ±± ( )

Checklist

1. Host object has a side effect (no constant).
2. Returned object is
   a. a copy of the host object after side effect (reference).
   b. used as lvalue (no constant).

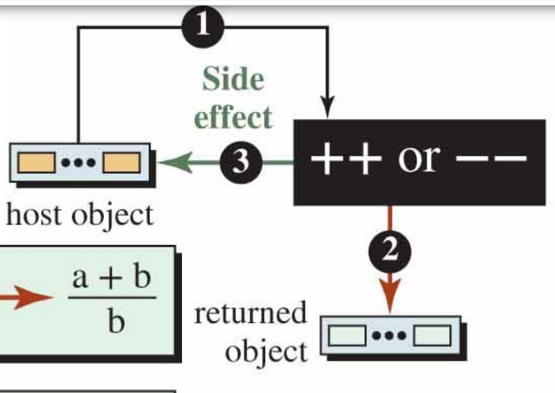
We now overload the prefix increment and the prefix decrement operator.

Each operator has a side effect changing its operand; the returned object is a copy of the changed object. This means that the returned value must be an lvalue.

```
Fraction fract3(3, 4);
Fraction fract4(4, 5);
++fract3;
--fract4;
Fraction fract55 = fract5++;
Fraction fract66 = fract6--;
```

```
// Declaration of pre-increment operator
Fraction& operator++();
// Definition pre-increment operator
Fraction&  Fraction :: operator++()
{
      numer = numer + denom;
      this -> normalize();
      return *this;
}
// Declaration of pre-increment
Fraction& operator--();
// Definition pre-decrement operator
Fraction&  Fraction :: operator--()
{
      numer = numer - denom;
      this -> normalize();
      return *this;
}
```

Post-Increment and Post-Decrement Operators Part 2



$$\frac{a}{b}++ \rightarrow \frac{a}{b}+1 \rightarrow \frac{a+b}{b}$$

$$\frac{a}{b}-- \rightarrow \frac{a}{b}-1 \rightarrow \frac{a-b}{b}$$

Prototye

```
const type operator ±± (int)
```

It doesn't have any specific meaning, it's just used to differentiate it from pre-increment/decrement.

Checklist

1. Host object has a side effect (no constant).
2. Returned object
   a. needs to be created as a temporary object (no reference).
   b. is used as rvalue (constant).

We now overload the postfix increment and the postfix decrement operators.

Since we cannot return the host object before changing it, we need to create a temporary object out of the host object, apply the side effect to the host object, and then return the temporary object.

The dummy integer parameter creates a unique signature to distinguish between the prototype of the pre-operator and post-operator; it is ignored.

```
Fraction fract3(3, 4);
Fraction fract4(4, 5);
++fract3;
--fract4;
Fraction fract55 = fract5++;
Fraction fract66 = fract6--;
```

```
// Declaration of post-increment operator
const Fraction operator++(int);
// Definition of post-increment operator
const Fraction  Fraction :: operator++(int dummy)
{
     Fraction temp(numer, denom);
     ++(*this);
     return temp;
}
// Declaration of post-decrement operator
const Fraction operator--(int);
// Definition of post-decrement operator
const Fraction  Fraction :: operator--(int dummy)
{
     Fraction temp(numer, denom);
     --(*this);
     return temp;
}
```

In-Class Exercise #2

Prefix/Postfix increment/decrement

Let’s extend the Fraction class with the following application code.

```
int main()
{
   // Creation of four objects and testing the ++ and -- operators
   Fraction fract3(3, 4);
   Fraction fract4(4, 5);
   Fraction fract5(5, 6);
   Fraction fract6(6, 7);
   cout << "fract3: " << fract3.print() << endl;
   cout << "fract4: " << fract4.print() << endl;
   cout << "fract5: " << fract5.print() << endl;
   cout << "fract6: " << fract6.print() << endl << endl;
   ++fract3;
   --fract4;
   Fraction fract55 = fract5++;
   Fraction fract66 = fract6--;
   cout << "Result of ++fract3: " << fract3.print() << endl;
   cout << "Result of --fract4: " << fract4.print() << endl;
   cout << "Result of fract5++: " << fract5.print() << endl;
   cout << "Result of fract6--: " << fract6.print() << endl << endl;
   return 0;
}
```

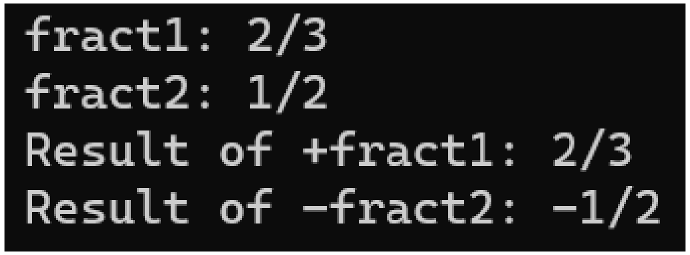
In-Class Exercise #3

Unary Plus and Minus

$$\boxed{+\frac{a}{b} \rightarrow +\frac{a}{b}} \quad \boxed{-\frac{a}{b} \rightarrow -\frac{a}{b}}$$

Let’s modify the Fraction class with the following application code.

```cpp
int main()
{
   // Creation of two objects and testing the plus and minus operator
   Fraction fract1(2, 3);
   Fraction fract2(1, 2);
   cout << "fract1: " << fract1.print() << endl;
   cout << "fract2: " << fract2.print() << endl;
   +fract1;
   -fract2;
   cout << "Result of +fract1: " << fract1.print() << endl;
   cout << "Result of -fract2: " << fract2.print() << endl << endl;
   return 0;
}
```



```
fract1: 2/3
fract2: 1/2
Result of +fract1: 2/3
Result of -fract2: -1/2
```

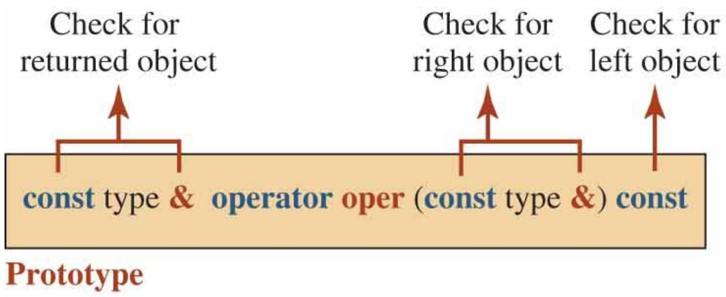
Binary Operator Overloading

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



Check for returned object
Check for right object
Check for left object
const type & operator oper (const type &) const
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

Assignment Operator

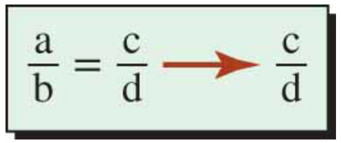
Assignment is an asymmetric operation in which the left operand receives the side effect of the operation; the right operand should not be changed in the process.

Both objects must exist; we only change the left object to be exactly a copy of the right object.

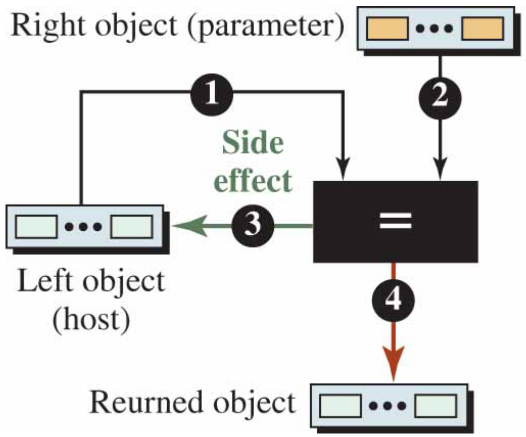
In an assignment operator, both left object (host) and right object (parameter) must already exist.

Overloading the assignment operator

`fract1 = fract2`



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



Prototype

type & operator = (const type &)

Checklist

1. Host object has a side effect (no constant).
2. Returned object
   a. is created from the right object (reference).
   b. can be chained (no constant).

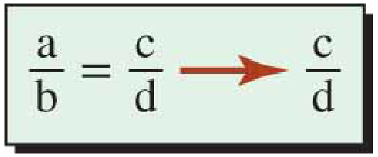
First the host object (left operand) cannot be constant because of the side effect.

Second the returned object cannot be constant because we can chain this operator (x = y = z).

Assignment Operator

- The first is that we should verify that the host and the parameter objects are not the same object (do not have the same address).
- Assignment is associative from right to left. This is the reason the returned object needs to be returned by reference.

```
// Declaration of inequality operator
const bool operator!=(const Fraction& right);
// Definition of inequality operator
const bool Fraction :: operator!=(const Fraction& right)
{
     return this->numer * right.denom != right.numer * this->denom;
}

// Declaration of assignment operator
Fraction& operator=(const Fraction& right);
// Definition of assignment operator
Fraction& Fraction :: operator=(const Fraction& right)
{
     if (*this != right) //or check in another way
     {
          numer = right.numer;
          denom = right.denom;
     }
     return *this;
}
```



```
fract1 = fract2
```

```
fract1 != fract2
```

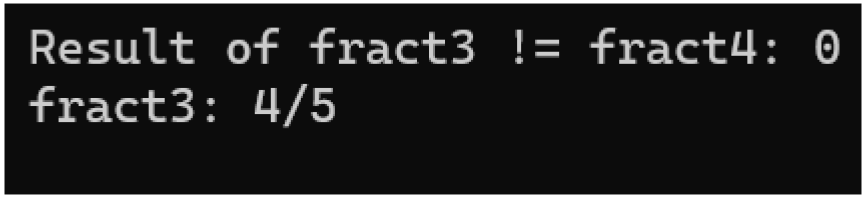
In-Class Exercise #4

Assignment, Inequality Operators

$$\frac{a}{b} = \frac{c}{d} \longrightarrow \frac{c}{d}$$

Let’s extend the Fraction class with the following application code.
# Also, try to implement equality (‘==’) operator and test it.

```
int main()
{
   // Testing assignment & inequality operators
   Fraction fract3(3, 4);
   Fraction fract4(4, 5);
   if (fract3 != fract4)
   {
      fract3 = fract4;
   }
   cout << "Result of fract3 != fract4: "
        << to_string(fract3 != fract4) << endl;
   cout << "fract3: " << fract3.print() << endl << endl;
   return 0;
}
```



```
Result of fract3 != fract4: 0
fract3: 4/5
```

Compound Assignment Operators Part 1

Another set of operators that we can overload is the compound assignment operators (+=, -=, *=, /=, and %=).

We know the meaning of the compound assignment.

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

In this operation, *fract*1 is the host object and *fract*2 is the parameter object.

The process is the same as the assignment operator except that we need to carefully define what is the numerator and denominator of the host object.

The returned object is the host object after change.

Compound Assignment Operators Part 2

```
// Declarations of += operator
Fraction& operator+=(const Fraction& right);
// Definition of += operator
Fraction& Fraction :: operator+=(const Fraction& right)
{
     numer  = numer * right.denom + denom * right.numer;
     denom = denom * right.denom;
     normalize();
     return *this;
}
```

```
// Declaration of -= operator
Fraction& operator-=(const Fraction& right);
// Definition of -= operator
Fraction& Fraction :: operator-=(const Fraction& right)
{
     numer = numer * right.denom - denom * right.numer;
     denom = denom * right.denom;
     normalize();
     return *this;
}
```

$$\frac{a}{b} \mathrel{+}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} + \frac{c}{d} \longrightarrow \frac{a*d+b*c}{b*d}$$

$$\frac{a}{b} \mathrel{-}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} - \frac{c}{d} \longrightarrow \frac{a*d-b*c}{b*d}$$

$$\frac{a}{b} \mathrel{*}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} * \frac{c}{d} \longrightarrow \frac{a*c}{b*d}$$

$$\frac{a}{b} \mathrel{/}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} \,/\, \frac{c}{d} \longrightarrow \frac{a*d}{b*c}$$

Compound Assignment Operators Part 3

```
// Declaration of *= operator
Fraction& operator*=(const Fraction& right);
// Definition of *= operator
Fraction& Fraction :: operator*=(const Fraction& right)
{
     numer = numer * right.numer;
     denom = denom * right.deonm;
     normalize();
     return *this;
}
```

```
// Declaration of /= operator
Fraction& operator/=(const Fraction& right);
// Definition of /= operator
Fraction& Fraction :: operator-=(const Fraction& right)
{
     numer = numer * right.denom - denom * right.numer;
     denom = denom * right.denom;
     normalize();
     return *this;
}
```

$$\frac{a}{b} \mathrel{+}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} + \frac{c}{d} \longrightarrow \frac{a*d+b*c}{b*d}$$

$$\frac{a}{b} \mathrel{-}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} - \frac{c}{d} \longrightarrow \frac{a*d-b*c}{b*d}$$

$$\frac{a}{b} \mathrel{*}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} * \frac{c}{d} \longrightarrow \frac{a*c}{b*d}$$

$$\frac{a}{b} \mathrel{/}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} \;/\; \frac{c}{d} \longrightarrow \frac{a*d}{b*c}$$

In-Class Exercise #5

$$\frac{a}{b} \mathrel{+}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} + \frac{c}{d} \longrightarrow \frac{a*d+b*c}{b*d}$$

$$\frac{a}{b} \mathrel{-}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} - \frac{c}{d} \longrightarrow \frac{a*d-b*c}{b*d}$$

$$\frac{a}{b} \mathrel{*}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} * \frac{c}{d} \longrightarrow \frac{a*c}{b*d}$$

$$\frac{a}{b} \mathrel{/}= \frac{c}{d} \longrightarrow \frac{a}{b} = \frac{a}{b} \,/\, \frac{c}{d} \longrightarrow \frac{a*d}{b*c}$$

Compound Assignment Operators

Let's extend the Fraction class with the following application code.

```cpp
int main()
{
   // Testing compound assignment operators
   Fraction fract7(3, 5);
   Fraction fract8(4, 7);
   Fraction fract9(5, 8);
   Fraction fract10(7, 9);

   fract7 += 2; // == Fraction(2, 1)
   fract8 -= 3; // == Fraction(3, 1)
   fract9 *= 4; // == Fraction(4, 1)
   fract10 /= 5; // == Fraction(5, 1)

   cout << "Result of fract7 += 2: " << fract7.print() << endl;
   cout << "Result of fract8 -= 3: " << fract8.print() << endl;
   cout << "Result of fract9 *= 4: " << fract9.print() << endl;
   cout << "Result of fract10 /= 5: " << fract10.print() << endl << endl;
   return 0;
}
```

```
Result of fract7 += 2: 13/5
Result of fract8 -= 3: -17/7
Result of fract9 *= 4: 5/2
Result of fract10 /= 5: 7/45
```

In-Class Exercise #6

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

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

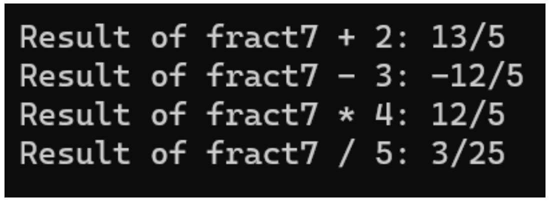
$$\frac{a}{b} \mathrel{*}= \frac{c}{d} \rightarrow \frac{a}{b} = \frac{a}{b} * \frac{c}{d} \rightarrow \frac{a*c}{b*d}$$

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

Binary Arithmetic Operators

Let's extend the Fraction class with the following application code.

```cpp
int main()
{
   // Testing binary arithmetic operators
   Fraction fract7(3, 5);

   Fraction fract8 = fract7 + 2; // == Fraction(2, 1)
   Fraction fract9 = fract7 - 3; // == Fraction(3, 1)
   Fraction fract10 = fract7 * 4; // == Fraction(4, 1)
   Fraction fract11 = fract7 / 5; // == Fraction(5, 1)

   cout << "Result of fract7 + 2: " << fract8.print() << endl;
   cout << "Result of fract7 - 3: " << fract9.print() << endl;
   cout << "Result of fract7 * 4: " << fract10.print() << endl;
   cout << "Result of fract7 / 5: " << fract11.print() << endl << endl;
   return 0;
}
```



```
Result of fract7 + 2: 13/5
Result of fract7 - 3: -12/5
Result of fract7 * 4: 12/5
Result of fract7 / 5: 3/25
```

Summary

Three Roles of An Object

Unary Operator Overloading

Binary Operator Overloading

What’s Next?

Reading Assignment

- ❑ **Read Chap. 13. Operator Overloading**
- ❑ **Read Chap. 14. Exception Handling**
- ❑ **Read Chap. 15. Generic Programming: Templates**

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr