Lecture 10.2

Topics
1. Passing-by-Reference – Revisited
2. Operator Overloading – Basic Arithmetic Operators

---

1. Passing by Reference (PBR) – Revisited

In PBR, no copy is made and no destructor would be called when a function returns. However, any changes made to the object inside the function will affect the objects used as arguments.

It is important to note that a reference is not a pointer. They are two different derived data types. When an object is passed by reference, the member access operator is the dot (‘ **.** ’) and not the arrow (‘ **->** ‘). Consider the following example.

Example 1

```
Fraction& returnFraction( Fraction& frOld ) {

  return frOld;
}

int main( void ) {
  Fraction frA( 4, 5 );

  Fraction frB;

  cout << "\nBefore calling returnFraction(): " << endl;

  cout << "\nfrA: ";
  frA.print();

  cout << "\nfrB: ";
  frB.print();

  frB = returnFraction( frA );

  cout << "\nAfter calling returnFraction(): " << endl;

  cout << "\nfrA: ";
  frA.print();

  cout << "\nfrB: ";
  frB.print();

  cout << endl;

  return 0;
}
```

OUTPUT

```
Before calling returnFraction():

frA:
        Numerator:   4
        Denominator: 5
```

```
frB:
        Numerator:   0
        Denominator: 1

After calling returnFraction():

frA:
        Numerator:   4
        Denominator: 5

frB:
        Numerator:   4
        Denominator: 5


Destructor Call!

Destructor Call!
```

Note that when the execution is completed, there are only two destructor calls as opposed to having additional destructor calls due to copies of objects as in the case of PBV.

During the execution of `returnFraction()`, the reference of **`frA`** is passed to from `main()` to `returnFraction()`. There is no (additional) object created while executing `returnFraction()`. Thus, there is no destructor call during the execution of `returnFraction()`.

2. Operator Overloading -- Basic Arithmetic Operators

Operator overloading is just another form of function overloading (one form of polymorphism). That means the same interface can be used to define many different operations. However, operator overloading has additional rules and it also requires some restrictions related to the meaning and interpretation of the operators being overloaded.

Why operator overloading? The answer is to allow additional flexibility in creation of classes (new data types) with operations relative to classes.

Let us consider the rules and restrictions regarding the formation of operator overloading.

2.1 Rules

There are rules that one must observe in creating an overloading operator.

(i) First, operator overloading is achieved through an **operator function** (**OF**) of following form,

```
ReturnType ClassName::operator#( argList ) {
  //Operation Definition
}
```

where **#** is substituted by the actual operator being overloaded and the **`operator`** keyword is kept as shown.

(i) Secondly, OF may either be a **function member** or a **friend** of a class for which it is defined.

(ii) Next but not last, an operator is always overloaded relative to a class.

Technically, OF can perform any task and return any type. However, it should be kept within bound (and meaning) of the usual definition of the operator. Most often, OF returns an object from the

same class relative to the class for which it is created. Its argument list `argList` may contain different data type(s) as well as number of argument(s) depending on the type of operator being overloaded.

2.2 Restrictions

There are restrictions on forming the arguments based on specific operator.

(1) The precedence of the operator cannot be changed, and
(2) The number of operands that an operator takes cannot be altered.

In the discussion below, following operators will be considered or mentioned:

```
Arithmetic (Binary) Operators:   +    -    =    *    /
Increment/Decrement Operators:   ++   --
Logical Operators:               &&   ||   etc.
Relational Operators:            ==   >    etc.
```

There are some operators that cannot be overloaded (e.g., '`.`', '`::`', `etc.`); in addition, preprocessor operators may not be overloaded.

Note!

C++ defines operators very broadly. To limit the discussion, only a few basic binary and unary operators will be considered.

Regarding inheritance, except for the **=** operator, OFs are inherited by derived class; and a derived class is free to overload any operator it chooses relative to itself.

2.3 Binary Operator Overloading – Assignment (=) Operator

To overload a **binary** operator, an OF needs **only one** parameter. This parameter takes on object on the right side of the operator; while the object on the left side is the one that generates the operation (call). The left object is implicitly passed to function using **`this`** pointer. An OF can have many variations but may not have default arguments.

```
frB = frB;
```

Example

```
//FILE #1 - Class Specification File

/**
 *Program Name:  myfraction.h
 *Discussion:     Class with and without
 *                      Data Encapsulation
 */

#ifndef MYFRACTION_H
#define MYFRACTION_H

#include <iostream>
using namespace std;

class Fraction {
public:
  Fraction();
  Fraction( const Fraction& frOld );
  Fraction( int iOld );
  Fraction( int, int );
```

```
  ~Fraction();

  int getNum();
  void setNum( int iOld );

  int getDenom();
  void setDenom( int iOld );

  void print() const;

  void operator=( const Fraction& );

private:
  int iNum;
  int iDenom;
};

void printFractionArray( Fraction [], int );

Fraction doubleFraction( Fraction& );

Fraction& returnFraction( Fraction& );

#endif

//FILE #2 - Implementation File
/**
 *Program Name:  myfraction.cpp
 *Discussion:    Implementation File:
 *                 Objects in Functions -
 *                   Fraction Class
 */

#include <iostream>
#include "myfraction.h"

Fraction::Fraction() {
  iNum = 0;
  iDenom = 1;
}

/*
Fraction::Fraction( const Fraction& frOld ) {
  iNum = frOld.iNum;
  iDenom = frOld.iDenom;
}
*/


Fraction::Fraction( const Fraction& frOld ) {
  iNum = -99;
  iDenom = 1001;
}


Fraction::Fraction( int iA, int iB ) {
  iNum = iA;
  if ( iB ) {
```

```
    iDenom = iB;
  } else {
    iDenom = 1;
  }
}

Fraction::Fraction( int iOld ) {
  iNum = iOld;
  if ( iOld ) {
    iDenom = iOld;
  } else {
    iDenom = 1;
  }
}

Fraction::~Fraction() {
  cout << "\nDestructor Call!" << endl;
}

int Fraction::getNum() {
  return iNum;
}

void Fraction::setNum( int iOld ) {
  iNum = iOld;
  return;
}

int Fraction::getDenom() {
  return iDenom;
}

void Fraction::setDenom( int iOld ) {
  if ( iOld ) {
    iDenom = iOld;
  } else {
    iDenom = 1;
  }

  return;
}

void Fraction::print() const {
  cout << "\n\tNumerator:   " << iNum
       << "\n\tDenominator: " << iDenom << endl;

  return;
}

void Fraction::operator=( const Fraction& frOld ) {
  iNum = frOld.iNum;
  iDenom = frOld.iDenom;

  return;
}
```

```
//Non-Member Functions

void printFractionArray( Fraction frAry[], int iSize ) {
  for ( int i = 0; i < iSize; i++ ) {
    cout << "\nFraction frAry[ " << i <<" ] : "
         << "\n\t" << frAry[ i ].getNum()
         << "\n\t" << frAry[ i ].getDenom();
  }

  cout << endl;
  return;
}

Fraction doubleFraction( Fraction& frOld ) {
  Fraction frTemp;

  frTemp.setNum( frOld.getNum() );

  frTemp.setDenom( 2 * frOld.getDenom() );

  return frTemp;
}


Fraction& returnFraction( Fraction& frOld ) {
  return frOld;
}

//File #3 – Driver
/**
 *Program Name:  cis25L1021Driver.cpp
 *Discussion:    Returning Object
 */

#include <iostream>
#include "myfraction.h"

using namespace std;

int main( void ) {
  Fraction frA( 4, 5 );

  Fraction frB;

  cout << "\nBefore calling operator=(): " << endl;

  cout << "\nfrA: ";
  frA.print();

  cout << "\nfrB: ";
  frB.print();

  frB = frA;

  cout << "\nAfter calling operator=(): " << endl;

  cout << "\nfrA: ";
  frA.print();
```

```
  cout << "\nfrB: ";
  frB.print();

  cout << endl;

  return 0;
}
```

OUPUT

```
Before calling operator=():

frA:
        Numerator:   4
        Denominator: 5

frB:
        Numerator:   0
        Denominator: 1

After calling operator=():

frA:
        Numerator:   4
        Denominator: 5

frB:
        Numerator:   4
        Denominator: 5


Destructor Call!

Destructor Call!
```

2.4 Binary Operator Overloading – Addition (+) Operator

Instead of calling `addFraction()` to add two `Fraction` objects, let's consider a possibility of using the following expression

```
frC = frA + frB;
```

In C++, it is possible to have the above expression. In many ways, the above expression would reflect a direct operation of adding two objects. The performance is done using operator function.

The **+** operator is overloaded to perform an addition operation using two objects. The **OF** requires one **right operand** of specific type, e.g., `Fraction, OA, int, etc.`, while the **left operand** of type of the defined class (such as Rectangle) is given implicitly to the (operator) function call.

The result of the operator function call is then returned to an object of any desired type.

> *Regarding the RETURN statement, it is always true that a copy or a reference of the object is being sent back to the calling environment (CE) for every object specified in the return statement. This applies to all functions and operator functions (OFs) under any circumstances.*
>
> For statements related to copies of arguments to be passed to function, it is true that a copy of object with current state will be generated for every object that is EXPLICITLY specified in the argument list. Copy of IMPLICIT argument will NOT be generated.

Therefore, binary OFs with implicit argument or operand on the left hand side of the operators will not require copies of these objects to be generated. Instead, the operations rely on **`this`** pointer to obtain the value requested. Remember that function member generates a **`this`** pointer that points to object every time a function is being called.

Examples will be given and discussed in class.