Lecture 9.3

Topics

1. Objects in Functions – Return Value

---

1. Objects in Functions – Return Value

Recall that functions in C++ can operate on objects as arguments. Objects can be passed to functions by values or by references.

Functions will operate on different object scopes based on the declaration of objects and the definitions of the functions being considered. We have discussed and will continue to discuss the use of objects as function arguments.

However, let's look at another use of object in function. Certainly, information of an object can also be returned to the calling environment. More specifically, a copy of an object or a reference of an object may be returned.

What would involve in returning object from a function?

Previously, the process of returning the value(s) from a function to its calling environment was explained previously.

Let's reconsider the process (briefly again) in the following discussion.

1.1 Returning Value (Copy of Object) from Function

To get a return value from a function, two tasks must be accomplished accordingly. They are identified as

(1) Closing out the called function, and

(2) Generating and assigning the return value in the calling environment.

The details of two tasks are summarized below.

1.1.1 Task #1: Closing out Called Function

To close out the called functions, two steps are followed:

i. The return value or object specified in the return statement (belonged to the called function) will be copied and saved by the system. This means a NEW VALUE or OBJECT would be created; it is only a temporary and anonymous object.
ii. The called function is popped out of the function stack (which is a data structure that used to orderly handle all functions that are being called) with a successful indication. This would loosely mean that the function is done.

1.1.2 Task #2: Generating & Assigning Value/Object

After the first task is done, the second task will take place. Two general steps are given as follows:

i. The new value/object created in step (ii) above will be assigned to a variable or object in the calling environment based on the invocation of the called function.

ii. After a successful assignment, the new value/object above is then out of scope and destroyed.

Let's consider an example of returning a value from a function.

1.2 Returning Object Information by Value – Example

In this kind of returning, the value of the return from a function is not a pointer or a reference. It represents a data point which may be a simple value or an object. For a simple value case, it is fairly easy to understand the result without going through the process.

Let's consider the case where the object is being returned as the case for a **`Fraction`** object.

Example 1

```
//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() const;
  void setNum( int iOld );

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

  void print() const;

private:
  int iNum;
  int iDenom;
};

void printFractionArray( Fraction [], int );

Fraction doubleFraction( 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( 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() const{
  return iNum;
}

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

int Fraction::getDenom() const {
  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 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;
}

/**
 *Program Name:  cis25L0931Driver.cpp
 *Discussion:    Returning Object
 */

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

using namespace std;

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

  Fraction frB;

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

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

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

  frB = doubleFraction( frA );

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

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

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

  cout << endl;
```

```
  return 0;
}
```

OUTPUT

```
Before calling doubleFraction():

frA:
        Numerator:   4
        Denominator: 5

frB:
        Numerator:   0
        Denominator: 1

Destructor Call!

Destructor Call!

After calling doubleFraction():

frA:
        Numerator:   4
        Denominator: 5

frB:
        Numerator:   4
        Denominator: 10


Destructor Call!

Destructor Call!
```

In the above example, the output shows several calls to the destructor to remove the objects. A closer look would reveal that these are copies of objects that were created and destroyed as the function being executed.

Explanation will be given in class, where a key to remember is that each object will have to be removed through a destructor.