Lecture 9.4

Topics

1. Dynamic Objects – **new** and **delete** Operators
2. Dynamic Arrays of Objects

---

1. Dynamic Objects – new and delete operators

A dynamic object is an object created during run-time (also called "**late binding**" object). In the applications where dynamic objects are needed, these dynamic objects can be created through a combination of using a pointer and the **new** operator.

1.1 new Operator

Recall that the **new** operator is used to request memory from dynamic memory area or space (also call "free store"). The dynamic memory space is shared by several applications and managed by the computer operating system.

A similar process of creating and working with dynamic data and memory exists for objects. As the requested memory is allocated, the **new** operator will then create the object(s) and return the memory location where these objects are residing.

The general syntax for creating one dynamic object is as follows,

```
DataType *dtPtr;
dtPtr = new DataType;
```

where

`DataType` is any valid data type
`dtPtr` is the pointer to `DataType`

In the above syntax, if the `new` operator is successful in allocating the memory then an object will be created and the address of the object will be stored in `dtPtr`. If the operation is not possible then the pointer `dtPtr` will receive a value of `0`.

Examples

```
int* iPtr; // declaring a pointer to int
iPtr = new int; // one memory block for int is allocated,
                // the value is unknown
*iPtr = 5; // 5 is stored in the memory block (location) at
           // the address given in iPtr
*iPtr = *iPtr + 15;  // value pointed to by iPtr is added to
                     // 15; the result is stored to the memory
                     // location iPtr
Fraction* frPtr1;  // declaring a pointer to Fraction
frPtr1 = new Fraction; // one memory block for Fraction
                       // is allocated; the default
                       // constructor is used to create
                       // the object. The memory of this
                       // memory block is stored in frPtr1
(*frPtr1).print(); // print out the information for the
                   // object pointed to by frPtr1
```

```
Fraction* frPtr2;  // declaring a pointer to Fraction
frPtr2 = new Fraction( 5, 7 );
                          // one memory block for Fraction
                          // is allocated; the constructor with
                          // two double arguments is used to
                          // create the object. The memory of
                          // this memory block or object is
                          // stored in frPtr2
(*frPtr2).print();  // print out the information for the
                    // object pointed to by frPtr2
```

1.2 Scope of Existence of Dynamic Object – `delete` Operator

Recall that an automatic or local object will exist until the program execution is finished. The memories for all local objects will be reclaimed by the computer/system accordingly. What about the dynamic objects? How long would a dynamic object exist? How would the system reclaim its memory?

A dynamic object will exist until an instruction is given to remove (destroy) this object. The removal instruction is specified through a call to the **`delete`** operator.

The syntax of the delete call is as follows,

```
delete dtPtr;
```

where

- **`delete`** is the operator
- **`dtPtr`** is a pointer that is currently pointing to a dynamic memory location

After the `delete` call, the memory location is no longer belonging to the (current) application. The memory address stored in `dtPtr` becomes invalid and it should not be used.

For examples,

```
delete iPtr;  // releasing an int block back to the system
delete frPtr1;  // destroying a Fraction object and
                // releasing the memory back to the system
delete frPtr2;  // destroying a Fraction object and
                // releasing the memory back to the system
```

Most likely, there will be several dynamic objects are needed instead of just one. That means an array of dynamic objects (or, dynamic array of objects) would be needed. The questions will be:

- How would this be accomplished?
- How do we create a dynamic array?

2. Dynamic Arrays of Objects

A dynamic array of objects can also be created through a pointer. The general syntax and steps are given as follows,

```
DataType* dtPtr;
dtPtr = new DataType[ iArraySize ];
```

where

`DataType` is any valid data type

`dtPtr` is the pointer to `DataType`
`iArraySize` is any integral expression

Examples

```
iPtr = new int[ 5 ]; // allocating a block of 5 int's and
                     // storing the address of this block
                     // in iPtr

frPtr1 = new Fraction[ 2 ]; // allocating a block of memory
                            // for 2 Fraction objects; the
                            // objects are created by the
                            // default constructor
```

2.1 Access Array Member with Array Notation

The array notation to access the individual member of the array is as follows,

`iPtr[ 0 ]` is the first member of the array
`iPtr[ 1 ]` is the second member of the array
`iPtr[ 4 ]` is the fifth member of the array

`frPtr1[ 0 ]` is the first Fraction object of the array
`frPtr1[ 1 ]` is the second Fraction object of the array

Examples

```
iPtr[ 0 ] = 8;
iPtr[ 1 ] = iPtr[ 0 ] + 6;

frPtr1[ 0 ].print();
frPtr1[ 1 ].print();
```

Note that the dot (**.**) operator is used to access the `Fraction` members as before. The expression on the left of the dot operator is a `Fraction` object, which is accessing the `print()` function member.

2.2 Access Array Member with Pointer Notation

The pointer notation to access the individual member of the array is as follows,

`iPtr` is the address of the first member of the array

`*( iPtr )` is the first member of the array

`( iPtr + 1 )` is the address of the second member of the array

`*( iPtr + 1 )` is the second member of the array

Examples

```
*( iPtr ) = 9;

*( iPtr + 1 ) = *( iPtr ) + 6;


*( frPtr1 ).print(); // the first object of the array accesses
                     // print() function member using the
                     // dot operator

*( frPtr1 + 1 ).print(); // the second object of the array
                         // accesses print() function member
                         // using the dot operator

( frPtr1 )->print(); // the first object of the array accesses
```

```
                        // print() function member using the
                        // arrow operator
( frPtr1 + 1 )->print(); // the second object of the array
                           // accesses print() function member
                           // using the arrow operator
```

A complete example is given below.

Example 1

```
// FILE #1 - Class Specification File
/**
 *Program Name:  myFraction01.h
 *Discussion:   Class Fraction
 */
#ifndef MYFRACTION01_H
#define MYFRACTION01_H

class Fraction01 {
public:
  Fraction01();
  Fraction01( const Fraction01& frOld );
  Fraction01( int iOld );
  Fraction01( int, int );

  ~Fraction01();

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

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

  void print() const;

private:
  int iNum;
  int iDenom;
};

void printFractionArray( Fraction01 [], int );

#endif


// FILE #2 - Implementation File
/**
 *Program Name:  myFraction01.cpp
 *Discussion:    Implementation File:
 *                 Objects in Functions -
 *                   Fraction01 Class
 */
#include <iostream>
#include "myFraction01.h"
using namespace std;

Fraction01::Fraction01() {
  iNum = 0;
  iDenom = 1;
}
```

```
Fraction01::Fraction01( const Fraction01& frOld ) {
  iNum = frOld.iNum;
  iDenom = frOld.iDenom;
}

Fraction01::Fraction01( int arg1, int arg2 ) {
  iNum = arg1;
  if ( arg2 )
    iDenom = arg2;
  else
    iDenom = 1;
}

Fraction01::Fraction01( int arg ) {
  iNum = arg;
  if ( arg )
    iDenom = arg;
  else
    iDenom = 1;
}

Fraction01::~Fraction01() {
  cout << "\nDestructor Call!" << endl;
}

int Fraction01::getNum() {
  return iNum;
}

void Fraction01::setNum( int arg ) {
  iNum = arg;
  return;
}

int Fraction01::getDenom() {
  return iDenom;
}

void Fraction01::setDenom( int arg ) {
  if ( arg )
    iDenom = arg;
  else
    iDenom = 1;

  return;
}

void Fraction01::print() const {
  cout << "\n\tNumerator:   " << iNum
       << "\n\tDenominator: " << iDenom << endl;

  return;
}

void printFractionArray( Fraction01 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;
}

// FILE #3 – Application Driver
/**
 *Program Name:  cis25L0941Driver.cpp
 *Discussion:    Pointers to Dynamic Objects - new
 */
#include <iostream>
#include "myFraction01.h"
using namespace std;

int main( void ){
  int i;
  Fraction01* frPtr;

  //creating dynamic array
  frPtr = new Fraction01[ 2 ];

  //print/display a Fraction01 object using function member
  cout << "\nRight after creating the array, "
       << "\nprinting with for-loop:" << endl;

  for ( i = 0; i < 2; i++ ) {
    cout << "\ni = " << i << " :" << flush;
    frPtr[ i ].print();
  }

  cout <<"\nprinting with printFractionArray():" << endl;
  printFractionArray( frPtr, 2 );

  //updating the array
  for ( i = 0; i < 2; i++ ) {
    ( *( frPtr + i ) ).setNum( i + 4 );
    ( *( frPtr + i ) ).setDenom( i + 5 );
  }

  //print/display a Fraction01 object using function member
  cout << "\nAfter updating the objects, "
       << "\nprinting with for-loop:" << endl;

  for ( i = 0; i < 2; i++ ) {
    cout << "\ni = " << i << " :" << flush;
    ( frPtr + i )->print();
  }

  cout <<"\nprinting with printFractionArray():" << endl;
  printFractionArray( frPtr, 2 );

  cout << endl;

  return 0;
}
```

OUTPUT

```
Right after creating the array,
printing with for-loop:
```

```
i = 0 :
        Numerator:   0
        Denominator: 1

i = 1 :
        Numerator:   0
        Denominator: 1

printing with printFractionArray():

Fraction frAry[ 0 ] :
        0
        1
Fraction frAry[ 1 ] :
        0
        1

After updating the objects,
printing with for-loop:

i = 0 :
        Numerator:   4
        Denominator: 5

i = 1 :
        Numerator:   5
        Denominator: 6

printing with printFractionArray():

Fraction frAry[ 0 ] :
        4
        5
Fraction frAry[ 1 ] :
        5
        6
```

There are two (2) dynamic object pointed to by `frPtr`. What happens to the destructor calls?

2.3 Removing Objects – `delete` Operator

To remove/destroy a dynamic object, a call to the `delete` operator must be made as above. In Example 1, the two dynamic objects are not destroyed by the destructor calls. That means they might still exist after the application is completed.

How can this be? What should we do to clean these dynamic objects?

2.3.1 How can this be?

The situation here is referred to as "memory leak". In many compilers and systems, these dynamic objects from a particular application will be destroyed by the system based on how the "**garbage collection**" process was implemented. That means it may be done immediately after the application finishes its run or it may wait for some times before the system sweeps its dynamic memory space and then collects the garbage.

During the waiting time for the objects to be garbage-collected, these memory blocks are unusable and this may prevent other applications to use these memory blocks. This phenomenon is called "memory leak".

In some cases, these garbage-objects may persist and not be collected. This is the worst case because the system's dynamic memory will be crowded and become more fragmented as this kind dynamic objects getting created. The system will slow down and eventually hang up.

2.3.2 What should we do to clean these dynamic objects?

The answer is that the program/application must be responsible for the creation and destruction of objects through proper and explicit calls to **`new`** and **`delete`** operators.

A fix to the above example is through the following **`delete`** calls,

```
delete [] iPtr;
delete [] frPtr;
```

The `delete` operator will release the block of objects at the given memory location (address). The `delete` operator will call the appropriate destructor(s) to accomplish the task.

Example 2

```
/**
 *Program Name:  cis25L0942Driver.cpp
 *Discussion:    Pointers to Dynamic Objects -- delete
 */
#include <iostream>
#include "myFraction01.h"

using namespace std;

int main( void ) {
  int i;

  Fraction01* frPtr;

  //creating dynamic array
  frPtr = new Fraction01[ 2 ];

  //updating the array
  for ( i = 0; i < 2; i++ ) {
    ( *( frPtr + i ) ).setNum( i + 4 );
    ( *( frPtr + i ) ).setDenom( i + 5 );
  }

  //print/display a Fraction01 object using function member
  cout << "\nAfter updating the objects, "
       << "\nprinting with for-loop:" << endl;

  for ( i = 0; i < 2; i++ ) {
    cout << "\ni = " << i << " :" << flush;
    ( frPtr + i )->print();
  }

  delete [] frPtr;

  cout << endl;
```

```
  return 0;
}
```

OUTPUT

```
After updating the objects,
printing with for-loop:

i = 0 :
        Numerator:   4
        Denominator: 5

i = 1 :
        Numerator:   5
        Denominator: 6

Destructor Call!

Destructor Call!
```