Lecture 8.1

Topics

1. Member Functions – Accessor & Mutator
2. Class Destructor
3. `const` Qualifier
4. Pointer to Object – Brief

---

1. Member Functions – Accessor and Mutator

Recall that

> Class is an abstraction used to define a new data **type** through the **`class`** keyword. The general syntax of a **`class`** declaration is as follows,
>
> ```
> class ClassName {
> public:
>   //public member functions of class
> protected:
>   //protected member functions
> private:
>   //private member functions and data
> };
> ```

Let us look again at the following `Fraction` class.

```
class Fraction {
public:
  Fraction() { // public member function/method
    iNum = 0;
    iDenom = 1;
  }
private:
  int iNum;    // private member data
  int iDenom;  // provate member data
};
```

The following observations are made:

- Member data are all private.
- No member functions except for the default constructor.
- Objects can be created.
- After created, objects cannot be updated.
- These objects are unusable and wasted – “dead” objects.

As dead objects, they should not be created. To have usable objects, one may need to provide some member functions as follows,

```
class FractionA {
public:
  FractionA() {
    iNum = 0;
    iDenom = 1;
  }

  void setNum( int arg ) {
```

```
    iNum = arg;

    return;
  }

  int getNum( ) {
    return iNum;
  }

  void setDenom( int arg ) {
    if ( arg ) {
      iDenom = arg;
    } else {
      iDenom = 1;
    }
    return;
  }

  int getDenom( ) {
    return iDenom;
  }

private:
  int iNum;
  int iDenom;
};
```

1.1 Accessor – `getNum()`

The accessor **`getNum()`** allows the value of the member data **`iNum`** to be retrieved. The process does not change the object.

For example,

```
FractionA frA;

cout << "Numerator : frA.iNum : " << frA.getNum ()
     << "\nDenominator : frA.iDenom : " << frA.getDenom()
     << endl;
```

1.2 Mutator – `setNum()`

The mutator is a function that will alter or update the object. The mutator `setNum()` will change the member data iNum as follows,

```
frA.setNum( 5 );
frA.setDenom( 9 );

cout << "Numerator : frA.iNum : " << frA.getNum ()
     << "\nDenominator : frA.iDenom : " << frA.getDenom()
     << endl;
```

1.3 Data Encapsulation and Object Creation

An object should have its data encapsulated that means no **direct** access to data to be permitted.

Data should only be accessed (used or modified) through function calls; for examples, accessor and mutator. The class definition should reflect this criterion.

Every object should be created (constructed) with proper (initial) data values. This process is performed through the use of one of the special functions called constructor(s).

Let's briefly revisit constructor again.

1.4 Constructors and Observation

Recall that there can be as many constructors as one would "like" to have. The main goal of using a constructor in creating object is to make sure the creation to be complete and proper.

> Default arguments may also be used for constructors. This may cause ambiguity so that one must be careful when using default arguments with constructors.

For object that has dynamic memory, extra care must be provided to make sure that the no memory sharing is illegal and no memory leaking would be the case.

After an object was created, how would this object be removed or destroyed? When would the object be destroyed?

The answer is "destructor". What is destructor? What can it do? How can it be used? Why do we need it?

Let's consider the destructor next.

2. Destructor

In most cases, a class will have several constructors for initialization and ONE destructor to clean up after the object just before it is disappeared (i.e., out of scope).

What is destructor?

(1) Each class can have ONE AND ONLY ONE destructor.

(2) A destructor is a function that has

- No argument,
- No return value,
- The same name as that of class preceded by a ~ (tilde).

Example

```
/**
 *Program Name: cis25L0811.cpp
 *Discussion:   Class & Object
 */
#include <iostream>
using namespace std;

class FractionA {
public:
  FractionA() {
    iNum = 0;
    iDenom = 1;
  }

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

```
  FractionA( int arg ) {
    iNum = arg;
    if ( arg ) {
      iDenom = arg;
    } else {
      iDenom = 1;
    }
  }

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

  void setNum( int arg ) {
    iNum = arg;

    return;
  }

  int getNum( ) {
    return iNum;
  }

  void setDenom( int arg ) {
    if ( arg ) {
      iDenom = arg;
    } else {
      iDenom = 1;
    }
    return;
  }

  int getDenom( ) {
    return iDenom;
  }

private:
  int iNum;
  int iDenom;
};

int main( void ) {
  FractionA frA;

  cout << "For frA:\n\tNumerator : frA.iNum : "
       << frA.getNum () << "\n\tDenominator : frA.iDenom : "
       << frA.getDenom() << endl;

  frA.setNum( 5 );
  frA.setDenom( 9 );

  cout << "For frA:\n\tNumerator : frA.iNum : "
       << frA.getNum () << "\n\tDenominator : frA.iDenom : "
       << frA.getDenom() << endl;

  FractionA frB( frA );

  cout << "For frB:\n\tNumerator : frB.iNum : "
       << frB.getNum () << "\n\tDenominator : frB.iDenom : "
```

```
        << frB.getDenom() << endl;

  FractionA frC( 5 );

  cout << "For frC:\n\tNumerator : frC.iNum : "
       << frC.getNum () << "\n\tDenominator : frC.iDenom : "
       << frC.getDenom() << endl;

  return 0;
}
```

OUTPUT

```
For frA:
        Numerator : frA.iNum : 5
        Denominator : frA.iDenom : 9
For frB:
        Numerator : frB.iNum : 5
        Denominator : frB.iDenom : 9
For frC:
        Numerator : frC.iNum : 5
        Denominator : frC.iDenom : 5

Destructor Call!

Destructor Call!

Destructor Call!
```

3. `const` Qualifier

Let's consider the following code fragment,

```
for ( int iCount = 0; iCount < 256; iCount ++ )
{
  .. ..
}
```

There are two inherent problems with the above code:

i. Readability, and

ii. Maintainability.

What are they?

3.1 Readability

The value of **`256`** is being used to test the loop. What makes this value of 256 matter? Why not other numbers?

> Note that 256 in the above paragraph is referred to as a "**magic number**"; it is a value that seems to be used without any significance within the context of its own use.

In reading the code, one can certainly change this magic number without any justification with regarding to the execution flow. Thus, a value without logical meaning is not a good parameter to be used.

3.2 Maintainability

What if there are 1000 occurrences of this 256 value in the program, and one needs to convert many of these into some other values? To do this conversion, one must be able to identify which occurrences need to be converted. One wrong assertion may produce erroneous results, and it is a tedious search-and-replace process.

So what should be done to resolve the above two issues for the code fragment? Let's look at the option.

3.3 An Alternative

One can modify the code as follows,

```
int iMax;

for ( int iCount = 0; iCount < iMax; iCount ++ )
{
  .. ..
}
```

The above fix has one additional available `iMax`. It presents a trade-off where the value such as 256 is now localized and can be controlled; but it does have a problem of its own.

In the code, `iMax` is an **lvalue**. That means `iMax` can be changed within the program (during the execution); for example, an erroneous operation (funny but it happens) such as

```
if ( iMax = 1 )
{
  //…
}
```

What is wrong here?

3.4 The Alternative

The solution here is to use the **`const`** qualifier. A `const` qualifier will enforce the object to be a constant. For example,

```
const int iMax = 256;
const int const_Max = 256;
```

It should be read as follows,

> "`iMax` is a variable of type `int` defined as a constant value, which is initialized with value 256", and
>
> "`const_Max` is a variable of type `int` defined as a constant value, which is initialized with value 256".

Now, `const_Max` is the read-only variable, and as a constant, any attempt to change the value of `const_Max` from within the program will result in a compile-time error.

The two restrictions on a **`const`** variable are:

> (1) A const variable must be initialized after it was declared, and
> (2) Once a `const` variable was defined, its value cannot be changed.

3.5 Examples

Example

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

```
int main(){
  //const int iMax; // Error: No Initialization
  const int iMax = 256;

  //iMax = 512; // Error: l-value specifies const object

  int* iPtr;

  //iPtr = &iMax; // Error: '=' : cannot convert from
                  //              'const int *' to 'int *'

  char cTest;
  cout << "\nEnter a character to quit!  ";
  cin >> cTest;

  return 0;
}
```

In the above code, the errors are shown to reflect the attempt to change the values to constant objects.

4. Pointer to Object – Brief

A pointer to object can be declared just as those for variables.

Example

```
int main( void ) {
  FractionA frA;

  cout << "For frA:\n\tNumerator : frA.iNum : "
       << frA.getNum () << "\n\tDenominator : frA.iDenom : "
       << frA.getDenom() << endl;

  frA.setNum( 5 );
  frA.setDenom( 9 );

  cout << "For frA:\n\tNumerator : frA.iNum : "
       << frA.getNum () << "\n\tDenominator : frA.iDenom : "
       << frA.getDenom() << endl;

  FractionA frB( frA );

  cout << "For frB:\n\tNumerator : frB.iNum : "
       << frB.getNum () << "\n\tDenominator : frB.iDenom : "
       << frB.getDenom() << endl;

  FractionA frC( 5 );

  cout << "For frC:\n\tNumerator : frC.iNum : "
       << frC.getNum () << "\n\tDenominator : frC.iDenom : "
       << frC.getDenom() << endl;

  FractionA* frPtr;

  frPtr = &frA;
  cout << "For frA:\n\tNumerator : frA.iNum : "
       << ( *frAPtr ).getNum ()
       << "\n\tDenominator : frA.iDenom : "
```

```
            << ( *frAPtr ).getDenom()
            << endl;

      cout << "For frA:\n\tNumerator : frA.iNum : "
            << frAPtr->getNum ()
            << "\n\tDenominator : frA.iDenom : "
            << frAPtr->getDenom()
            << endl;

      return 0;
}
```

Mostly, a pointer to object will be pointing to a dynamic object such as follows,

```
FractionA* frPtr;
frPtr = new FractionA();  // dynamic object through a
                          // default constructor
```

Let's discuss the above example in class.