Lecture 11.2

Topics:

1. `friend` Function of Class – Functions & Operator Function Overloading
2. Operator Overloading – Revisited

---

1. `friend` Function of Class – Functions & Operator Function Overloading

Objects provide operations that can be used to perform on data. Up to now, these operations are mostly function members. However, there are cases where a regular function that is not a class member is preferred in accessing private members of a class. This is accomplished through a **`friend`** function.

1.1 General Description

friend functions are often used in designing some of the I/O functions, operator overloading, and multiple-class access. The cases of I/O functions will not be presented here. The multiple-class access is given next and followed with operator overloading.

By its own nature, **friend** functions are declared inside class definition(s). A **`friend`** function is NOT a class member but HAS ACCESS TO THE PRIVATE MEMBERS of class. A `friend` function can have access to two or more classes by proper declaration in each of the class definitions.

A `friend` function is declared using the `friend` keyword as follows,

```
class ClassName {
  //members
public:
  //members

  friend ReturnType friendFunctionName( Type1 , Type2 ); //Declaration

};

ReturnType friendFunctionName( Type1 x, Type2 y ) //Definition
{
  //Function Body
}
```

Note that the scope resolution operator (`"::"`) is not needed in the definition of `friend` function. Thus, `friend` function is not linked to any object and it can be made to be friends with more than one class (multiple class case).

An important point about **`friend`** function is that if it is declared in the base class then it is not inherited by any derived class.

1.2 Examples

Three examples with friend functions are given. The first example has a simple friend function declaration and call involving one class. The remaining two examples involve with multiple-class friend functions.

Example 1

```
// Program Name: cis25L1121.cpp
// Discussion:   Class -- friend Function
#include <iostream>
using namespace std;
```

```
class OneFriend {
  int x;
public:
  OneFriend( void ) {
    x = 0;
  }
  OneFriend( int a ) {
    x = a;
  }
  ~OneFriend() {
    cout << "\nOneFriend Destructor!\n";
  }
  void setX( int value ) {
    x = value;
    return;
  }
  int getX( void ) {
    return x;
  }
  friend int square( const OneFriend &old );
  friend int multiplyTwo( const OneFriend &, const OneFriend & );
};

int square( const OneFriend& old ) {
  return ( old.x * old.x );
}

int multiplyTwo( const OneFriend& old1, const OneFriend& old2 ) {
  return ( old1.x * old2.x );
}

int main( void ) {
  OneFriend ofObj( 10 );
  int iY = square( ofObj );
  cout << "Square of x in ofObj is " << iY << endl;

  return 0;
}
```

OUTPUT

```
Square of x in ofObj is 100

OneFriend Destructor!
```

The friend function `square()` is declared inside class `OneFriend` with an argument which is an object. Regarding object class, `square()` can access any private member of this object. For examples, within the definition of `square()` the identifier `old.x` is allowed in any expression. Clearly, the use of friend function is not necessary in this example and can be substituted by other function.

Example 2

```
// Program Name: cis25L1122.cpp
// Discussion:   Class -- Friend Function & Forward Reference
#include <iostream>
using namespace std;

class Apt;        //Forward Reference
class House {
  int area;
public:
  House() {
    area = 0;
  }
```

```
  void setArea( int a ) {
    area = a;
    return;
  }
  int getArea( void ) {
    return area;
  }
  friend int compareHouseApt( const House& h, const Apt& a );
};

class Apt {
  int area;
public:
  Apt() {
    area = 0;
  }
  void setArea( int a ) {
    area = a;
    return;
  }
  int getArea( void ) {
    return area;
  }
  friend int compareHouseApt( const House& h, const Apt& a );
};

// Returning a positive No. if (area of a house > area of an apartment)
// Returning ZERO if (area of a house = area of an apartment)
// Returning a negative No. if (area of a house < area of an apartment)

int compareHouseApt( const House& h, const Apt& a ) {
  return ( h.area - a.area );
}

int main(void) {
  House h;
  Apt a;

  h.setArea( 2000 );
  a.setArea( 1100 );
  int iY = compareHouseApt( h, a );

  if ( iY > 0 )
    cout << "\nThe house is larger than the apartment by "
         << "(in Square Feet) " << iY << endl;
  else if ( iY < 0 )
    cout << "\nThe house is smaller than the apartment by "
         << "(in Square Feet) " << -iY << endl;
  else
    cout << "\nThe house is the same size as the apartment"
         << endl;

  cout << endl;
  return 0;
}
```

OUTPUT

```
The house is larger than the apartment by (in Square Feet) 900
```

Function `compareHouseApt()` is made to be friends with two classes and, thus, can access private members of these classes. A closer look at the first function declaration of `compareHouseApt()` in class `House` would reveal that two objects of different classes were specified. This is legal because of

the declaration statement "`class Apt;`" just before the definition of class `House` and this is called **forward referencing**.

A function can be made a friend of a class and a member of another. In this case, the scope resolution operator must be used to qualify for one of the declarations.

Example 3

```
// Program Name: cis25L1123.cpp
// Discussion:   Class -- Friend Function & Forward Reference
#include <iostream>
using namespace std;

class Apt;        //Forward Reference
class House {
  int area;
public:
  House() {
    area = 0;
  }
  ~House() {
    cout << "\nHouse Destructor!\n";
  }
  void setArea( int a ) {
    area = a;
    return;
  }
  int getArea( void ) {
    return area;
  }
  int compareHouseApt( const Apt& a );
};

class Apt {
  int area;
public:
  Apt() {
    area = 0;
  }
  ~Apt() {
    cout << "\nApt Destructor!\n";
  }
  void setArea( int a ) {
    area = a;
    return;
  }
  int getArea( void ) {
    return area;
  }
  friend int House::compareHouseApt( const Apt& a );
};

// Returning a positive No. if (area of a house > area of an apartment)
// Returning ZERO if (area of a house = area of an apartment)
// Returning a negative No. if (area of a house < area of an apartment)

int House::compareHouseApt( const Apt& a ) {
  return ( area - a.area );
}

int main( void ) {
  House h;
  Apt a;
  h.setArea( 2000 );
  a.setArea( 1100 );
```

```
  int iY = h.compareHouseApt( a );

  if ( iY > 0 )
    cout << "\nThe house is larger than the apartment by "
         << "(in Square Feet) " << iY << endl;
  else if ( iY < 0 )
    cout << "\nThe house is smaller than the apartment by "
         << "(in Square Feet) " << -iY << endl;
  else
    cout << "\nThe house is the same size as the apartment"
         << endl;

  cout << "\n";
  return 0;
}
```

OUTPUT

```
The house is larger than the apartment by (in Square Feet) 900


Apt Destructor!

House Destructor!
```

As seen in the above example, `compareHouseApt()` is modified to be a member of class `House` while being friend with class `Apt`.

Although friend functions are used as given in this section, their real applications are seen in the operator overloading and I/O functions. In these areas, friend functions would allow many new data type as well as operations to be created for specific tasks.

1.3 `friend` Functions In Operator Overloading

Recall that `friend` function is not a member of the class thus it does not have the use of **`*this`** pointer. That means if a `friend operator function` (FOF) is defined then all operands must be EXPLICITLY declared and properly referred to in the body of FOF.

Thus, considering `10 + v1`, one possible prototype of FOF inside class **`FriendOA`** is given as follows,

```
class FriendOA {
public:
  // other members

  friend FriendOA operator+( int , const FriendOA & );
private:
  int x;
};
```

A closer look at the argument list of this FOF would reveal that the left operand is an object from different class and the right operand is an object from the class related to this FOF. Consider the following examples where FOF is used to handle the **`operator+()`** function.

Example 4

```
// Program Name: cis25L1124.cpp
// Discussion:   Overloaded OFs for '+' with friend Functions
#include <iostream>
using namespace std;
```

```
class FriendOA {
  int x;
public:
  FriendOA() {
    x = 0;
  }
  FriendOA( int a ) {
    x = a;
  }
  ~FriendOA() {
    cout << "\nOne Destruction!" << endl;
  }
  void setX( int value ) {
    x = value;
    return;
   }
  int getX( void )
  {
    return x;
  }
  FriendOA operator+( const FriendOA &);
  FriendOA operator+( int );
  friend FriendOA operator+( int , const FriendOA & );
};

FriendOA FriendOA::operator+( const FriendOA& lOp ) {
  FriendOA temp;
  temp.x = x + lOp.x;
  return temp;
}
FriendOA FriendOA::operator+( int a ) {
  FriendOA temp;
  temp.x = x + a;
  return temp;
}
FriendOA operator+( int rOp, const FriendOA& lOp ) {
  FriendOA temp;
  temp.x = rOp + lOp.x;
  return temp;
}

int main( void ) {
  FriendOA v1( 10 ), v2, v3, v4;

  v2 = v1 + 5;
  cout << "\nValue of x in object v2 is " << v2.getX() << endl;

  v3 = 10 + v2;
  cout << "\nValue of x in object v3 is " << v3.getX() << endl;

  v4 = v3 + v2;
  cout << "\nValue of x in object v4 is " << v4.getX() << endl;

  return 0;
}
```

OUTPUT

```
One Destruction!

One Destruction!

Value of x in object v2 is 15

One Destruction!
```

```
One Destruction!

Value of x in object v3 is 25

One Destruction!

One Destruction!

Value of x in object v4 is 40

One Destruction!

One Destruction!

One Destruction!

One Destruction!
```

Clearly, by letting OF be friend and not function member of class, the declaration of FOF provides the means to specify all required arguments. Thus, the above example accepts an integer to be added by an object of type `FriendOA`.

2. Operator Overloading – Revisited

Recall that operator overloading allows additional flexibility in creation of classes (new data types) with operations relative to classes.

Let us revisit 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.

1. 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.

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

   The function that overloads any of the operators **`( )`**, **`[ ]`**, **`->`**, or **`=`** for a class must be declared as a member of the class.

   If the leftmost operand of the operator **#** is an object of a different type, the function that overloads the operator **#** for a class must be a nonmember of that class. That means the function is a friend of the class.

   If the operator function that overloads the operator **#** for a class is the member of that class, the leftmost operand of **#** must be an object of the given class (that has the operator function).

3. 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., '`.`', '`.*`', '`::`', '`?:`'); in addition, preprocessor operators may not be overloaded.

Note that 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 Overloading Stream Insertion Operator `<<`

The general syntax of the stream insertion operator is given as follows,

Prototype:

```
friend ostream& operator<<( ostream & , const ClassName & );
```

Definition:

```
friend ostream& operator<<( ostream &out, const ClassName &cnObj ) {
  //Local declarations
  //Output operations
  //out << operand

  return out; //Returning stream object
}
```

2.4 Overloading Stream Extraction Operator `>>`

The general syntax of the stream extraction operator is given as follows,

Prototype:

```
friend istream& operator>>( istream & , ClassName & );
```

Definition:

```
friend istream& operator>>( istream &in, ClassName &cnObj ) {
  //Local declarations
  //Output operations
  //in >> operand

  return in; //Returning stream object
}
```

2.5 Overloading Assignment Operator =

One of the **built-in operations** on a given class is the **assignment operation**. The assignment operator performs **bit-wise** copy of data members of the given class. This copying process is termed as **shallow copy**.

As shown previously, if there are dynamic data then the built-in assignment (or shadow copying) may not work correctly. The assignment operator must be overloading to handle these specific cases.

Note that to overload the assignment operator with respect to a class, the operator function `operator=()` must be a member of that class.

The general syntax of the assignment operator is given as follows,

Prototype:

```
const ClassName& ClassName::operator=( const ClassName & );
```

Definition:

```
const ClassName& ClassName::operator=( const ClassName& cnObj ) {
  //Local declarations
  //Comparison operations
  //Allocating memory
  //Copying data

  return *this; //Returning the object
}
```

2.6 Examples

The definitions for the operator functions are given below.

```
ostream& operator<<( ostream &out, const Fraction& frac ) {
  out << "\nFraction Components -- "
      << "\n\tNumerator   : " << frac.iNum
      << "\n\tDenominator : " << frac.iDenom << endl;

  return out;
}

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

  return *this;
}
```