C++ Foundation with Data Structures

Lecture: Object Oriented Programming Concepts (OOPS - 2) Notes

a. *Initialization List*

After studying the concept of classes, we will now learn the usage of an initializer list. We can specify initial values for data members of a class using the initialization list in a constructor. We write the initializer list in the definition of a constructor after the argument list and before the constructor body.

Syntax:

```
constructor_name (arglist) : initialization-list
{
        //assignment-section
}
```

Now, let us see an example to use initialization list to provide initial values to the base constructors and also to initialize its own class members.

Example Code:

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

```
class first {
        int a;
    public:
        first(int i) {
            a = i;
            cout<<"\n constructor of first class";
        }
        void display_first() {
            cout<<" a = "<<a<<endl;
        }
};
```

*/*class first*/*

```
class second {
        float var1, var2;
    public:
        second(float i, float j) : var1(i), var2(j+var1)  {
            cout<<"\n constructor of second class";
        }
        void display_second() {
```

*/*initializer list specified with constructor of class second to initialize values of var1 and var2*/*

cout<<" var1 = "<<var1<<endl;
cout<<" var2 = "<<var2<<endl;
}
};	

class third : public second, public first{
int p;
public:
third(int i, float j) : first(i), second(j, j), p(i) {
cout<<"\n constructor of third class";
}
void display_third() {
cout<<" p = "<<p<<endl;
}
};
	/*initializer list specified with constructor of class third to initialize value of p and to provide initial values to the base constructors of classes first and second*/

int main() {
third obj(5, 7.6);
cout<<"\n Member values: "<<endl;
obj.display_first();
obj.display_second();
obj.display_third();
return 0;
}
	/*Here, in main(), we created object of class third and passed values to it.*/

Output:

constructor of second class
constructor of first class
constructor of third class
Member values:
a = 5
var1 = 7.6
var2 = 15.2
p = 5

Description:

Here, we have used three classes: first, second and third. In class second, we used the initialization list to initialize its own class members var1 and

var2 with their respective values in the parenthesis in initializer list. While, we inherited class third from classes first and second. So, we used initializer list in class third to provide initial values to the base constructors (of classes first and second) as well as to initialize its own class member. In main(), we created object of class third and passed values to it. The initializer list in constructor of third class will then initialize its data member and provide initial values to constructors of first and second classes.

We can use an initialization list in any of the following scenarios:

i. When a class has constant but non-static data members.
ii. When a class has reference data members.
iii. When we wish to make an explicit call to a constructor that takes arguments for a data member which is an object of another class.
iv. When we want to make an explicit call from a derived class constructor to a base class constructor that takes arguments.

1. *Constant data members*

We can declare data members of a class as constant using the keyword ***const***. Such a data member must be initialized by the constructor using an initialization list. Once it is initialized, a ***const*** data member may never be modified, not even in the constructor or destructor.

2. *Reference data members*

We can use a reference variable to provide an alias (alternative name) for a previously defined variable. Data members of a class can be declared as references using the symbol **&** before the variable names. We can use references as class members only if we have initialized them by constructor.

b. *Static data members*

Till now, we have studied that all objects of a class share one copy of each member function and have separate copies of the data members. Thus, change made in a data member by one object was not reflected in another object. However, we can accomplish this reflection of change in all objects of the class by declaring our data members as static.

Some important points to note about static data members are given below:

i. We can define a class member as static using ***static*** keyword. A static member is shared by all objects of the class, that is, only one

copy of a static member is created which is shared by all objects of the class.

ii. Static data is initialized to zero by default when the first object is created, if no other initialization is present.

iii. A static data member can't be initialized inside the class definition but it can be assigned value outside the class by redeclaring it using the scope resolution operator (**::**) along with the class to which it belongs. Here, as we can observe, we declared static data member in public access section so that we can access it directly outside the class using its class name.

Now, let us see an example code to declare and use a static data member of a class. The following code maintains and displays the count of the number of objects created of the given class.

Example Code:

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

class Sample {
  public:

      //static data member declaration
      static int object_cnt;

      Sample() {
          cout <<"Constructor called" << endl;
          object_cnt++;
      }
};

//static data member assignment
int Sample::object_cnt = 0;

int main() {
      Sample obj1;
      Sample obj2;
      cout<<"Total number of objects created: "<<Sample::object_cnt
      <<endl;
```

```
        return 0;
}
```

Output:
Constructor called
Constructor called
Total number of objects created: 2

Description:
Here, we declared and assigned initial value 0 to a static data member object_cnt of class Sample. As we learned now, only one copy of a static data member is shared by all objects of the class, so, here, each time we declare an object of class Sample, its constructor increments the value of static data member object_cnt which is shared by all objects of class Sample. Thus, we can display the number of created objects by accessing the static data member object_cnt using scope resolution operator (::) along with its class name.

c. Constant functions

We can declare a class function to be constant. We must do this in the function's prototype as well as in its definition by using the keyword **const** after the function's argument list.

Now, let us implement an example to use constant function in our code to return the value of a data member of the class.

Example Code:

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

class Sample {
        int var = 5;
public:
        int get() const;        // const function prototype inside class
};

int Sample::get() const     // const function definition
{
  return var;
```

```
}

int main()
{
   Sample s;
   cout<<s.get();
   return 0;
}
```

Output:
5

Description:
Here, we used a constant function to return value of a data member of the class. Any attempt to change a data member of the object that called a const function will result in a syntax error. const objects can only call const functions while objects that are not const can call either const or non-const functions.

We may overload a const function with a non-const function. The selection of which one to call is made by the compiler by referring to the context in which the function is called.

Constructors and destructors can never be declared as const. They are always allowed to modify an object even if the object is const.

d. Static functions

Similar to static data members, we can even define class member functions as static using the ***static*** keyword.

We must remember the following points about static member functions:

i. A static member function can have access to only other static members (functions or variables) declared in the same class. While, a static data member can be accessed by static as well as non-static functions.

ii. Only one copy of a static member function is shared by all objects of the class so it can be called directly using the class name with the scope resolution operator as follows:

class_name :: static_function_name;

So, let us see an example to use static member function in our class to count the number of objects created for the given class.

Example Code:

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

class sample
{
   static int cnt;                    //static data member
public:
   sample()
   {
      cnt++;
   }
   static void object_count()      //static member function
   {
      cout<<"No. of objects created: "<<cnt<<endl;
   }
};

int sample :: cnt;

int main()
{
   sample obj1, obj2;
   sample::object_count();
   sample obj3;
   sample::object_count();
   return 0;
}
```

Output:

No. of objects created: 2
No. of objects created: 3

Description:
Here, we used static member function object_count() to display the static data member cnt which stores the count of objects created for the given class. Since it is static, so we can directly access it using class-name.

e. Const Objects

We may also declare an object of a class as constant using the **const** keyword. We may observe the const property of an object once the constructor has finished execution and before the class destructor executes.

So, constructor and destructor can modify a const object, but other methods of the class can't modify it. Only const methods can be called for a const object.

Syntax:

const *class-name object-name(args);*

Suppose, now we want to declare a const object of class date, then we will declare it as follows:

Example:

const date obj(5, 2, 1996);

Here, we have declared a const object named 'obj' of the date class, which stores the date passed as argument to it.

Operator Overloading

- It is a type of polymorphism in which we can overload an operator to give special user defined meanings to it.

- We can overload all the C++ operators except the following:
    i. Class member access operators (**.**), (**.***)
    ii. Scope resolution operator (**::**)
    iii. Size operator (**sizeof**)
    iv. Conditional operator (**?:**)

- Operator functions must be either member functions (non-static) or friend functions. A friend function will have only one argument for unary operators and two for binary operators; while a member function has no arguments for unary operators and only one for binary operators.

- General Syntax of a member operator function:

```
return-type class-name :: operator # (arg-list)
{
        //Function body. Operations are defined here
}
```

  Here, we will substitute # by the operator being overloaded.

- General forms for the prefix and postfix increment (++) and decrement (--) operator functions for overloading unary operators are as follows:

`type operator ++() {`
`  //body of prefix operator`
`}`
	`//Prefix increment`

`type operator ++(int x) {`
`  //body of postfix operator`
`}`
	`//Postfix increment`

`type operator --() {`
`  //body of prefix operator`
`}`
	`//Prefix decrement`

`type operator --(int x) {`
`  //body of postfix operator`
`}`
	`// Postfix decrement`

Now, let us implement an example code to overload the binary operator '+' for concatenating two given strings into one string.

Example Code:

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

class concat   {
        char str[80];
        public:
                void getdata();

                // function to overload '+' operator
                concat operator+(concat c);

                void display();
};

void concat::getdata()   {
        cin.getline(str, 80);
}

concat concat::operator+(concat c)   {
        concat temp;
        strcpy(temp.str, str);
        strcat(temp.str, c.str);
        return temp;
}

void concat::display()   {
        cout<<str;
}

int main()   {
        concat c1, c2, c3;
        cout<<"Enter first string: ";
        c1.getdata();
        cout<<"Enter second string: ";
        c2.getdata();

        //invoking operator overloaded function
        c3=c1+c2;
```

```
        cout<<"Concatenated string: ";
        c3.display();
        return 0;
}
```

Output:

Enter first string: basket
Enter second string: ball
Concatenated string: basketball

Description:

Here, we overloaded the binary operator '+' for concatenating two given strings into one string. We declared the operator function as member function of the class. We declared c1, c2 and c3 as objects of class concat, where each object has a string as its data member. We then invoked the overloaded operator function as: c3 = c1 + c2, which signifies that c1 invoked the member operator function and passed the object c2 as parameter. The overloaded operator function then returned the object containing the concatenated string. Thus, the object c3 contains the concatenated string.