UNIT-2

Classes and Object, Dynamic Memory Management, Constructor & Destructor

BCA-2nd Sem

Member Functions

- The program defines person as a new data of type class.

- The class person include two basic data type items and two function to operate on that data. it's call Member function.

- There are two type:

1 Outside the class definition

2 Inside the class definition

Outside the class Defination

- The member functions of a class can be defied outside the class definitions. It is only declared inside the class but defined outside the class. The general form of member function definition outside the class definition is:

- Return_typeClass_name::function_name(argumentlist)
  {
  Functionbody
  }
- Where symbol **::** is a supe resolution operator.

Inside the class definition

- The member function of a class can be declared and defined inside the class definition.

EX.

```
Class item
{
        int  number;
        float cost;
```

Inside the class definition

```
Public:
        Void getdata(int a, float b);
        Void putdata(void)
    {
        cout<< number<<“\n”;
        cout<<cost<<“\n”;
    }
};
```

Data Member

- Data member depends on access control of data member.
- If it's public then data member can be easily accessed using the direct member access operator with the object of that class.
- If the data member is defined as private or protected then we can not access the data variables directly.

Data Member

Type of data Member

1.Public

2.Private

3.protected

Data member type

```
#include <iostream>
using namespace std;
class alpha{
private:
 int id;
static int count;
public:
alpha(){count++;
id=count;}}
```

Data member type

```
void print(){
 cout<<"My id is"<<id;
cout<<"countis"<<count;} };
int alpha : :count=0;
void main (){
 alpha a1,a2,a3;
 a1.print();
 a2.print();
 a3.print();}
```

Friend Function

- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.

- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

Friend Function

Example:

```
class Box
 {
 double width;
 public: double length;
 friend void printWidth( Box box );
voidsetWidth(doublewid);
};
```

Friend Function Property

1) Friend of the class can be member of some other class.

2) Friend of one class can be friend of another class or all the classes in one program, such a friend is known as GLOBAL FRIEND.

3) Friend can access the private or protected members of the class in which they are declared to be friend, but they can use the members for a specific object.

Friend Function Property

4) Friends are non-members hence do not get “this” pointer.

5) Friends, can be friend of more than one class, hence they can be used for message passing between the classes.

6) Friend can be declared anywhere (in public, protected or private section) in the class.

Friend Class

- A class can also be declared to be the friend of some other class.

- When we create a friend class then all the member functions of the friend class also become the friend of the other class.

- This requires the condition that the friend becoming class must be first declared or defined (forward declaration).

Friend Class Example

```
#include <iostream>
 using namespace std;
class MyClass{
 friend class SecondClass;
public: MyClass() : Secret(0){}
void printMember()
 { cout << Secret << endl;}
 private: int Secret; };
 }
```

Friend Class Example

```
class SecondClass{
 public: void change( MyClass& yourclass, int x ){
yourclass.Secret = x;} };
 void main()
 {
MyClass my_class;
SecondClass sec_class;
my_class.printMember();
sec_class.change( my_class, 5 );
my_class.printMember();}
```

Array of object

```
class A
 {
 int* myArray;
 A()
 {
myArray = 0;
 }
A(int size)
 {
```

Array of object

```
myArray = new int[size];
 }
~A()
{
delete []
myArray;
}
 }
```

Returning Objects from Functions

```
#include <iostream>
using namespace std;
class Complex{
private: int real;
int imag;
public: Complex():
real(0), imag(0){ }
 void Read() {
```

Returning Objects from Functions

```
cout<<"Enter real and imaginary number
   respectively:"<<endl;
cin>>real>>imag;
 }
Complex Add(Complex comp2)
{
Complex temp;
temp.real=real+comp2.real;
temp.imag=imag+comp2.imag;
return temp;
```

Returning Objects from Functions

```
}
void Display()
{
cout<<"Sum="<<real<<"+"<<imag<<"i";
}
};
int main()
 {
```

Returning Objects from Functions

```
Complex c1,c2,c3;
c1.Read();
c2.Read();
c3=c1.Add(c2);
c3.Display();
return 0; }
```

Nested Classes

- A class can be declared within the scope of another class. Such a class is called a "nested class.

- " Nested classes are considered to be within the scope of the enclosing class and are available for use within that scope.

- To refer to a nested class from a scope other than its immediate enclosing scope, you must use a fully qualified name.

Nested Classes Example

```
#include <iostream.h>
class Nest
 {
 public: class Display
 {
 private: int s;
public:
void sum( int a, int b)
{
```

Nested Classes Example

```
s =a+b; }
void show( )
{
cout << "\nSum of a and b is:: " << s;
 } };
};
 void main() {
 Nest::Display x; x.sum(12, 10);
 x.show();
}
```

Namespaces

- Namespace is a new concept introduced by the ANSI C++ standards committee.

- This defines a scope for the identifiers that are used in a program.

- For using the identifier defined in the namespace scope we must include the using directive, like

- Using namespace std;

Namespaces

- Here, std is the namespace where ANSI C++ standard class libraries are defined.
- All ANSI C++ programs must include this directive.
- This will bring all the identifiers defined in std to the current global scope. Using and namespace are the new keyword of C++.

Constructor

- A constructor (having the same name as that of the class) is a member function which is automatically used to initialize the objects of the class type with legal initial values.

Characteristics of Constructor

(i) These are called automatically when the objects are created.

(ii) All objects of the class having a constructor are initialized before some use.

(iii) These should be declared in the public section for availability to all the functions.

Characteristics of Constructor

(iv) Return type (not even void) cannot be specified for constructors.

(v) These cannot be inherited, but a derived class can call the base class constructor.

(vi) These cannot be static.

Characteristics of Constructor

(x) An object of a class with a constructor cannot be used as a member of a union.

(xi) A constructor can call member functions of its class.

(xii) We can use a constructor to create new objects of its class type by using the syntax

Characteristics of Constructor

(vii) Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.

(viii) These can have default arguments as other C++ functions.

(ix) A constructor can call member functions of its class.

Types of Constructor

1 Overloaded Constructors

2 Copy Constructor

3 Dynamic Initialization of Objects

4 Constructors and Primitive Types

5 Constructor with Default Arguments

1.Overloaded Constructors

- Besides performing the role of member data initialization, constructors are no different from other functions.

- This included overloading also. In fact, it is very common to find overloaded constructors.

- For example, consider the following program with overloaded constructors for the figure class :

1.Overloaded Constructors

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
#include<string.h> //for strcpy()
Class figure
{
Private:
Float radius, side1,side2,side3;
```

1.Overloaded Constructors

```
Figure (float s1, floats2, float s3)
{
ide1=s1;
side2=s2;
side3=s3;
radius=0.0;
strcpy(shape,”triangle”);
}
Char shape[10];
    Public:
figure(float r)
```

1.Overloaded Constructors

```
{
radius=r;
strcpy (shape, “circle”);
}
void area()
{
float ar,s;
if(radius==0.0)
{
 a
```

1.Overloaded Constructors

```
if (side3==0.0)
ar=side1*side2;
else
ar=3.14*radius*radius;
cout<<”\n\nArea of the
    “<<shape<<”is :”<<ar<<”sq.units\n”;
} };
Void main()
{
Clrscr();
```

1.Overloaded Constructors

```
Figure circle(10.0); /
Figure rectangle15.0,20.6);
Figure Triangle(3.0, 4.0, 5.0);
Rectangle.area();
Triangle.area();
Getch();//freeze the monitror
}
```

2.Copy Constructor

- It is of the form classname (classname &) and used for the initialization of an object form another object of same type. For example,

2.Copy Constructor

```
Class fun {
Float x,y;
Public:
Fun (floata,float b)
{
x = a;
y = b;
}
Fun (fun &f)
{
```

2.Copy Constructor

```
cout<<”\ncopy constructor at work\n”;
X = f.x;
Y = f.y;
}
Void display (void)
{
{
Cout<<””<<y<<end1;
}
};
```

3.Dynamic Initialization of Objects

- The class objects can be initialized at run time (dynamically).
- We have the flexibility of providing initial values at execution time.

3.Dynamic Initialization of Objects

```
#include <iostream.h>
#include <conio.h>
Class employee {
Int empl_no;
Float salary;
Public:
Employee()  {}
Employee(int empno,float s) {
Empl_no=empno;
Salary=s; }
```

3.Dynamic Initialization of Objects

```
Employee (employee &emp)
{
Cout<<”\ncopy constructor working\n”;
Empl_no=emp.empl_no;
Salary=emp.salary;
}
Void display (void)
{
Cout<<”\nEmp.No:”<<empl_no<<”salary:”<<salary<<
   end1; }
```

3.Dynamic Initialization of Objects

```
};
 Void main()
{ int eno;
float sal;
clrscr();
cout<<”Enter the employee number and salary\n”;
cin>>eno>>sal;
employee obj1(eno,sal);
```

3.Dynamic Initialization of Objects

```
cout<<"\nEnter the employee number and salary\n";
cin>eno>>sal;
employee obj2(eno,sal);
obj1.display();
employee obj3=obj2;
obj3.display();
getch();
}
```

4.Constructors and Primitive Types

- In C++, like derived type, i.e. class, primitive types (fundamental types) also have their constructors.

- Default constructor is used when no values are given but when we given initial values, the initialization take place for newly created instance.
- Example:

- float x,y;
- int a(10), b(20);
- float i(2.5), j(7.8);

4.Constructors and Primitive Types

```
Class add
{
Private:
Int num1, num2,num3;
Public:
Add(int=0,int=0);
Void enter (int,int);
Void sum();
Void display();
};
```

4.Constructor with Default Arguments

```
add::add(int n1, int n2)
 {
num1=n1;
num2=n2;
num3=n0; }
Void add ::sum()
{
Num3=num1+num2;
}
Void add::display () {
Cout<<"\nThe sum of two numbers is "<<num3<<end1;
   }
```

Destructor

- The syntax for declaring a destructor is :

-name_of_the_class()

{

}

- It does not take any parameter nor does it return any value. Overloading a destructor is not possible and can be explicitly invoked. I
- n other words, a class can have only one destructor. A destructor can be defined outside the class. The following program illustrates this concept :

Destructor

```
#include<iostream.h>
#include<conio.h>
class add {
private :
int num1,num2,num3;
public :
add(int=0, int=0);
void sum();
void display();
~ add(void);  };
```

Destructor

```
Add:: ~add(void)
{
Num1=num2=num3=0;
Cout<<”\nAfter the final execution, me, the object has
    entered in the”
<<”\ndestructor to destroy myself\n”;
 }
Add::add(int n1,int n2)
{
num1=n1;
num2=n2;
num3=0;
}
```

Destructor

```
Void add::sum() {
num3=num1+num2;
}
Void add::display () {
Cout<<"\nThe sum of two numbers is "<<num3<<end1;
}
void main()
{
```

Destructor

```
Addobj1,obj2(5),obj3(10,20):
Obj1.sum();
Obj2.sum();
Obj3.sum();
cout<<”\nUsing obj1 \n”;
obj1.display();
cout<<”\nUsing obj2 \n”;
obj2.display();
cout<<”\nUsing obj3 \n”;
obj3.display();
}
```

Characteristics of Destructors

(i) These are called automatically when the objects are destroyed.

(ii) Destructor functions follow the usual access rules as other member functions.

(iii) These de-initialize each object before the object goes out of scope.

(iv) No argument and return type (even void) permitted with destructors.

Characteristics of Destructors

(v) These cannot be inherited.

(vi) Static destructors are not allowed.

(vii) Address of a destructor cannot be taken.

(viii) A destructor can call member functions of its class.

(ix) An object of a class having a destructor cannot be a member of a union.

REFRENCES

- Learn Programming in C++ By Anshuman Sharma, Anurag Gupta, Dr.Hardeep Singh, Vikram Sharma