B.Sc –COMPUTER SCIENCE

II YEAR III SEMESTER

OBJECT ORIENTED PROGRAMMING WITH C++

Handled by
K.Shanmugavadivu

UNIT -2

FUNCTION PROTOTYPING

A function is a set of statements that take inputs, do some specific computation and produces output.
The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.

return_type function_name([arg1_type arg1_name, ...]) { code }
The general form of a function is:

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

int max(int x, int y)
{
    if (x > y)
    return x;
    else
    return y;
}

int main() {
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);

    cout << "m is " << m;
    return 0;
}
```

Output: m is 20

Parameter Passing to functions

The parameters passed to function are called ***actual parameters***. For example, in the above program 10 and 20 are actual parameters.

The parameters received by function are called ***formal parameters***. For example, in the above program x and y are formal parameters.

There are two most popular ways to pass parameters.

Pass by Value: In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

Pass by Reference or call by reference Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

Parameters are always passed by value in C. For example. in the below code, value of x is not modified using the function fun().

INLINE FUNCTIONS:

Definition:

An inline function is a function that is expanded in line when it is invoked. Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. It is defined by using key word **"inline"**

Necessity of Inline Function:

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times.

Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function.

When a function is small, a substantial percentage of execution time may be spent in such overheads.

C++ proposes a new feature called inline function.

General Form:

inline function-header

{ function body;

}

Eg:

#include<iostream.h> inline float mul(float x, float y)

{

return (x*y);

}

inline double div(double p, double q)

{ return (p/q);

} int main()

{

float a=12.345; float b=9.82; cout<<mul(a,b); cout<<div(a,b); return 0;

}

Properties of inline function:

1.Inline function sends request but not a command to compiler

2.Compiler my serve or ignore the request

3.if function has too many lines of code or if it has complicated logic then it is executed as normal function

Situations where inline does not work:

.

A function that is returning value , if it contains switch ,loop or both then it is treated as

. normal function.

if a function is not returning any value and it contains a return statement then it is treated as normal function

If function contains static variables then it is executed as normal function

□ .

If the inline function is declared as recursive function then it is executed as normal function.

Default Arguments

In C++ programming, we can provide default values for function parameters.
If a function with default arguments is called without passing arguments, then the default parameters are used.
However, if arguments are passed while calling the function, the default arguments are ignored.
only provide the default values from right to left as a parameter in the function. Default arguments plays important role when some arguments always have same value.

const Arguments

By constant argument, it is meant that the function cannot modify these arguments. In order to make an argument constant to a function, we can use the keyword const as shown : int sum (const int a, const int b); The qualifier const in function prototype tells the compiler that the function should not modify the argument. The constant arguments are useful when functions are called by reference.

Function Overloading

Function overloading is a C++ programming feature that allows us to have more than one function having same name but different parameter list, when I say parameter list, it means the data type and sequence of the parameters, for example the parameters list of a function myfuncn(int a, float b) is (int, float) which is different from the function myfuncn(float a, int b) parameter list (float, int). Function overloading is a compile-time polymorphism.

Now that we know what is parameter list lets see the rules of overloading: we can have following functions in the same scope.

```
sum(int num1, int num2)
sum(int num1, int num2, int num3)
sum(int num1, double num2)
```

```
#include <iostream>
using namespace std;
class Addition {
public:
    int sum(int num1,int num2) {
        return num1+num2;
    }
    int sum(int num1,int num2, int num3) {
       return num1+num2+num3;
    }
};
int main(void) {
    Addition obj;
    cout<<obj.sum(20, 15)<<endl;
    cout<<obj.sum(81, 100, 10);
    return 0;
}
```

Output:

```
35
191
```

FRIEND FUNCTIONS: The private members cannot be accessed from outside the class. i.e…. a non member function cannot have an access to the private data of a

class. In C++ a non member function can access private by making the function friendly to a class.

Definition:

A friend function is a function which is declared within a class and is defined outside the class. It does not require any scope resolution operator for defining . It can access private members of a class. It is declared by using keyword **"friend"**

Ex:

```
class sample
{int x,y; public:
sample(int a,int b);
friend int sum(sample s);
};
sample::sample(int a,int b)
{
x=a;y=b; }
int sum(samples s)
{ int sum; sum=s.x+s.y; return 0;
}
void main()
{
Sample obj(2,3); int res=sum(obj);
cout<< "sum="<<res<<endl;
}
```

Friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.
- It can be declared either in the public or private part of a class without affecting its meaning.

- Usually, it has the objects as arguments.

Friend Class:A class can also be declared to be the friend of some other class. When we create a friendclass 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).

```
#include <iostream.h> class sample_1
{ friend class sample_2;//declaring friend class int a,b;
        public:
        void getdata_1()
{ cout<<"Enter A & B values in class sample_1"; cin>>a>>b;
        }

        void display_1()
        {
        cout<<"A="<<a<<endl; cout<<"B="<<b<<endl;
                }
        }; class sample_2
{ int c,d,sum; sample_1 obj1;
public: void getdata_2()
{ obj1.getdata_1();
        cout<<"Enter C & D values in class sample_2";
        cin>>c>>d;
        } void sum_2()
{ sum=obj1.a+obj1.b+c+d;
        }

                void display_2()
{ cout<<"A="<<obj1.a<<endl; cout<<"B="<<obj1.b<<endl; cout<<"C="<<c<<endl;
        cout<<"D="<<d<<endl;
        cout<<"SUM="<<sum<<endl;
```

```
        } }; int main() { sample_1 s1; s1.getdata_1(); s1.display_1();
sample_2 s2; s2.getdata_2(); s2.sum_2(); s2.display_2();
    }
```

```
    Enter A & B values in class sample_1:1 2
    A=1
    B=2
    Enter A & B values in class sample_1:1 2 3 4
    Enter C & D values in class sample_2:A=1
    B=2
    C=3
    D=4
    SUM=10
```

Virtual Functions

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.

5. A class may have virtual destructor but it cannot have a virtual constructor.

concept of Virtual Functions

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

class base {
public:
   virtual void print()
   {
      cout << "print base class" << endl;
   }

   void show()
   {
      cout << "show base class" << endl;
   }
};

class derived : public base {
public:
   void print()
   {
      cout << "print derived class" << endl;
   }

   void show()
   {
      cout << "show derived class" << endl;
   }
};
```

```
int main()
{
    base* bptr;
    derived d;
    bptr = &d;

    // virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output:

```
print derived class
show base class
```

Classes and Objects

Class

A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.
Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

Example

Create a class called "MyClass":

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};
```

- The class keyword is used to create a class called MyClass.
- The public keyword is an **access specifier**, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.
- Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called **attributes**.
- At last, end the class definition with a semicolon ;.

Objects

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

```
ClassName ObjectName;
```

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.
This access control is given by Access modifiers in C++. There are three access modifiers : **public, private and protected**.

filter_none

edit

play_arrow

brightness_4

```
// C++ program to demonstrate
// accessing of data members

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    // Access specifier
    public:

    // Data Members
    string geekname;

    // Member Functions()
    void printname()
    {
       cout << "Geekname is: " << geekname;
    }
};

int main() {
```

```
    // Declare an object of class geeks
    Geeks obj1;

    // accessing data member
    obj1.geekname = "Abhi";

    // accessing member function
    obj1.printname();
    return 0;
}
```

Output:

```
Geekname is: Abhi
```

Member Functions

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name alng with function name.

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

```
class Cube
{
    public:
    int side;
    int getVolume()
    {
```

```
        return side*side*side;      //returns volume of cube
    }
};
```

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

```
class Cube
{
    public:
    int side;
    int getVolume();
}

// member function defined outside class definition
int Cube :: getVolume()
{
    return side*side*side;
}
```

The main function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot . operator.

Arrays with in a Class

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows –

type arrayName [arraySize];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows –

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array –

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5^{th} in the array a value of 50.0. Array with 4^{th} index will be 5^{th}, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above –

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

double salary = balance[9];

The above statement will take 10^{th} element from the array and assign the value to salary variable

Multi-dimensional arrays

C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.

Memory Allocation for Objects

Memory for objects is allocated when they are declared but not whenclass is defined. All objects in a given class uses same member functions. The member functions are created and placed in memory only once when they are defined in class definition

STATIC CLASS MEMBERS

Static Data Members

Static Member Functions

Static Data Members:

A data member of a class can be qualified as static. A static member variable has certain special characteristics:

. It is initialized to zero when the first object of its class is created. No other initialization is permitted.Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

- It is visible only within the class, but its lifetime is the entire program.

Static data member is defined by keyword **„static"**

Static Member Functions

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties:

A static function can have access to only other static members (functions or variables) declared in the same class.

A static member function is to be called using the class name (instead of its objects) as follows: **class-name :: function-name;**

Arrays of Objects

Arrays of variables of type "class" is known as "Array of objects". An array of objectsis stored inside the memory in the same way as in an ordinary array.

Syntax:

```
class class_name
{

        private:        data_type members;
public:
data_type members;
member functions;
```

};

Array of objects:

Class_name object_name[size]; Where size is the size of array

Ex:

Myclass obj[10];

Objects as Function Arguments

Objects as Function Arguments: Objects can be used as arguments tofunctions This can be done in three ways

a. Pass-by-value or call by value
b. Pass-by-address or call by address
c. Pass-by-reference or call by reference

a.Pass-by-value –A copy of object (actual object) is sent to function and assigned to the object of calledfunction (formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal object are not reflected to actual object. write a program to
swap values of two objects

b.Pass-by-address: Address of the object is sent as argument to function.
Here ampersand(&) is used as address operator and arrow (->) is used as de referencing operator. If any change made to formal arguments then there is a change to actual arguments

c.Pass-by-reference:A reference of object is sent as argument to function.
Reference to a variable provides alternate name for previously defined variable. If any change made to reference variable then there is a change to original variable.
A reference variable can be declared as follows

```
Datatype & reference variable =variable;
```

Friendly Functions

The private members cannot be accessed from outside the class. i.e.… a non member function cannot have an access to the private data of a class. In C++ a non member function can access private by making the function friendly to a class.

Definition:

A friend function is a function which is declared within a class and is defined outside the class. It does not require any scope resolution operator for defining . It can access private members of a class. It is declared by using keyword **"friend"**

Ex:

```
class sample
{ int x,y; public:
sample(int a,int b);
friend int sum(sample s);
};
sample::sample(int a,int b)
{
x=a;y=b; }
int sum(samples s)
{ int sum; sum=s.x+s.y; return 0;
}
void main()
{
Sample obj(2,3); int res=sum(obj);
cout<< "sum="<<res<<endl;
}
```

Friend Class:A class can also be declared to be the friend of some other class. When we create a friendclass 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).

Returning Objects

Returning Object as argument

Syntax:

```
object = return object_name;
```

An object is used to access the class members. Like normal variable, object can be pass as function argument.

const Member Functions

Like member functions and member function arguments, the objects of a class can also be declared as **const**. an object declared as const cannot be modified and hence, can invoke only const member functions as these functions ensure not to modify the object.
A const object can be created by prefixing the const keyword to the object declaration.
Any attempt to change the data member of const objects results in a compile-time error.

Syntax:

```
const Class_Name Object_name;
```

- When a function is declared as const, it can be called on any type of object, const object as well as non-const objects.
- Whenever an object is declared as const, it needs to be initialized at the time of declaration. however, the object initialization while declaring is possible only with the help of constructors.

A function becomes const when the const keyword is used in the function's declaration.
The idea of const functions is not to allow them to modify the object on which they are called. It is recommended the practice to make as many functions const as possible so that accidental changes to objects are avoided.
Following is a simple example of a const function.

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

class Test {
    int value;
public:
    Test(int v = 0) {value = v;}
```

```
    // We get compiler error if we add a line like "value = 100;"
    // in this function.
    int getValue() const {return value;}
};

int main() {
    Test t(20);
    cout<<t.getValue();
    return 0;
}
```

Output:

```
20
```

Pointers to Members,

Pointer to Data Members of Class

We can use pointer to point to class's data members (Member variables).

Syntax for Declaration :

```
datatype class_name :: *pointer_name;
```

Syntax for Assignment:

```
pointer_name = &class_name :: datamember_name;
```

Constructors and Destructors.

Introduction to Constructors: C++ provides a special member function called the constructor which enables an object to initialize itself when it is created.

.

Definition:- A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same name as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
class integer
{
        int m,n;
        public:
        integer( );
        ……..
        ……..
};
integer :: integer( )
{
        m=0;
        n=0;
}

int main()
{  integer obj1;
                        ……..
                        ……..
}
```

integer obj1; => not only creates object obj1 but also initializes its data members m and n to zero.

There is no need to write any statement to invoke the construct or function.

CHARACTERISTICS OF CONSTRUCTOR

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return type, not even void.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other c++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.

They make „implicit calls" to the operators new and delete when memory allocation is required.

Constructors are of 3 types:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

1.Default Constructor: A constructor that accepts no parameters is called the **default constructor**.

```
#include<iostream.h>
#include<conio.h> class item
        { int m,n; public: item()
{
m=10; n=20;
}
void put();
};
void item::put()
{
         cout<<m<<n;
}
void main()
       { item t;
          t.put(); getch(); }
```

2.Parameterized Constructors:-The constructors that take parameters are called parameterized constructors.

```
#include<iostream.h>
class item
{
int m,n;          public:
           item(int x, int y)
           {
           m=x;
```

```
        n=y;
        }
  };
```

When a constructor has been parameterized, the object declaration statement such as item t; may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in 2 ways: item t=item(10,20); //explicit call

item t(10,20); //implicit call

Eg:

```
#include<iostream.h>          #include<conio.h>
class item
{
int m,n;          public:
        item(int x,int y)
{       m=x;   n=y;
        }
        void put();
};
void item::put()
{
        cout<<m<<n;
}
void main()
{
item t1(10,20);
item t2=item(20,30);
t1.put();       t2.put();       getch();
}
```

3.Copy Constructor: A copy constructor is used to declare and initialize an object fromanother object. Eg:
item t2(t1); or item t2=t1;

1. The process of initializing through a copy constructor is known as copy initialization.
2. t2=t1 will not invoke copy constructor. t1 and t2 are objects, assigns the values of t1 to t2.
3. A copy constructor takes a reference to an object of the same class as itself as an argument. #include<iostream.h>

```
class sample
        {
        int n;          public:
sample() { n=0; }
sample(int a)
{
n=a;
}
sample(sample &x)
{ n=x.n;
}
void display()
 { cout<<n;
}
}; void main()
{ sample A(100); sample B(A); sample C=A;
sample D;
D=A;
A.display();
B.display();
```

```
C.display();
D.display();
}
```

Output: 100 100 100 100

Multiple Constructors in a Class: Multiple constructors can be declared in a class. There can be anynumber of constructors in a class.

```
class complex
{
float real,img; public:
        complex()//default constructor
        {
        real=img=0;
        }
        complex(float r)//single parameter parameterized constructor
        {
        real=img=r;
        }
        complex(float r,float i) //two parameter parameterized constructor
        {
        real=r;img=i;
        }
        complex(complex&c)//copy constructor
        {
real=c.real; img=c.img;

        }
        complex sum(complex c )
{
```

```
complex t;
        t.real=real+c.real;
t.img=img+c.img;     return t;
        }
        void show()
        {
        If(img>0)
                cout<<real<<"+i"<<img<<endl;
        else
                {
                img=-img;
                cout<<real<<"-i"<<img<<endl;
                }
        }
};
void main()
{
complex c1(1,2);      complex c2(2,2);     compex c3;   c3=c1.sum(c3);       c3.show();

}
```

DESTRUCTORS: A destructor, is used to destroy the objects that have been created by a constructor.

Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded **by a tilde. Eg: ~item() { }**

1. A destructor never takes any argument nor does it return any value.
2. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.
3. It is a good practice to declare destructors in a program since it releases memory space for future use.

```
#include<iostream>  using namespace std;       class Marks
{
public:
int maths;
int science;

//constructor Marks() {        cout << "Inside Constructor"<<endl;
      cout << "C++ Object created"<<endl;
}

//Destructor
  ~Marks() {  cout << "Inside Destructor"<<endl;         cout << "C++ Object
destructed"<<endl;
}
};

int main( )
{
  Marks m1; Marks m2; return 0; }
```

Output:

```
        Inside Constructor
        C++ Object created
        Inside Constructor
        C++ Object created
        Inside Destructor
        C++ Object destructed
        Inside Destructor
        C++ Object destructed
```