

# **Object Oriented Programming with C++(BCS306B)**

**Module - 2**

**Chapter 1 :**

## **ARRAYS, POINTERS, REFERENCES and DYNAMIC ALLOCATION OPERATORS**

### **SYLLABUS:**

Arrays, Pointers, References, and the Dynamic Allocation Operators: Arrays of Objects, Pointers to Objects, The this Pointer, Pointers to derived types, Pointers to class members. Functions Overloading, Copy Constructors: Functions Overloading, Overloading Constructor Functions. Copy Constructors, Default Function Arguments, Function Overloading and Ambiguity.

## 2.1 ARRAYS OF OBJECTS

We know that an array can be of any data type. Hence, we can also have arrays of variables that are of the type class. Such variables are called **array of objects**.

Consider the following class definition:

```
class employee
{
    char name[30];
    float age;
public:
    void getdata();
    void putdata();
};
```

The identifier employee is a user-defined data type and can be used to create objects that relate to different categories of the employee.

**For example, consider the following:**

```
employee managers[3]; //array of manager  
employee foremen[15]; //array of foreman  
employee workers[75]; //array of worker
```

- In above declaration, the array managers contains three objects (managers), namely, managers[0], managers[1] and managers[2], of the type employee class.
- Similarly, the foremen array contains 15 objects (foremen) and the workers array contains 75 objects (workers).
- Since, an array of objects behave like any other array, we can use the usual array-accessing methods to access individual elements and then the dot member operator to access the member functions.

**For example, the statement:**

- `manager[i].putdata();` will display the data of the  $i^{\text{th}}$  element of the array managers. That is, this statement requests the object `manager[i]` to invoke the member function `putdata()`.

```
#include<iostream>
using namespace std;
class Book
{
    int id;
    int price;

public:
    void setId(void)
    {
        price = 153;
        cout << "Enter the id of book" << endl;
        cin >> id;
    }

    void getId(void)
    {
        cout << "The id of this book is " << id << endl;
    }
};
```

```
int main()
{
    Book arr[4];
    for (int i = 0; i < 5; i++)
    {
        arr[i].setId();
        arr[i].getId();
    }

    return 0;
}
```

```
Enter the id of book
5
The id of this book is 5
Enter the id of book
4
The id of this book is 4
Enter the id of book
3
The id of this book is 3
Enter the id of book
2
The id of this book is 2
Enter the id of book
1
The id of this book is 1
```

```
-----
Process exited after 6.365 seconds with return value 0
Press any key to continue . . .
```

## 2.2 Pointers to Objects

- A pointer is a variable that stores the memory address of another variable (or object) as its value.
- Pointers to objects aim to make a pointer that can access the object, not the variables. Pointer to object in C++ refers to accessing an object.
- There are two approaches by which you can access an object. One is directly and the other is by using a pointer to an object in C++.
- **A pointer to an object in C++ is used to store the address of an object.**

For creating a pointer to an object in C++, we use the following syntax:

```
classname *pointertobject;
```

- For storing the address of an object into a pointer in c++, we use the following syntax:

`pointertoobject=&objectname;`

- The above syntax can be used to store the address in the pointer to the object.
- After storing the address in the pointer to the object, the member function can be called using the pointer to the object with the help of an arrow operator.

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

class My_Class
{
    int num;

public:
    void set_number(int value)
        {num = value;}
    void show_number();
};

void My_Class::show_number()
{
    cout << num << "\n";
}
```

```
int main()
{
    My_Class object, *p; // an object is declared and a
                          pointer to it

    object.set_number(1); // object is accessed directly
    object.show_number();

    p = &object;          // the address of the object is
                          assigned to p
    p->show_number(); // object is accessed using the pointer

    return 0;
}
```

## 2.3 The This Pointer

The **this** pointer holds the address of current object, in simple words you can say that this **pointer** points to the current object of the class. Let's take an example to understand this concept.

### Syntax Of 'this' Pointer In C++

When creating or using the 'this' pointer, the 'this' keyword in C++ is used in conjunction with the arrow operator '->' along with the name of the member or method being referred to. The syntax for it is as follows-

```
void functionName()  
{  
    this->memberName = value;  
}
```

Here, functionName refers to the name of the function you are using.

this-> creates the pointer to the memberName, i.e., the respective member of the class or function in question.

```
#include <iostream>
```

```
using namespace std;
```

```
class Demo
```

```
{
```

```
    private:    int num;
```

```
                char ch;
```

```
    public:
```

```
        void setMyValues(int num, char ch)
```

```
        {
```

```
            this->num =num;
```

```
            this->ch=ch;
```

```
        }
```

```
void displayMyValues()  
{  
    cout<<num<<endl;  
    cout<<ch;  
}  
};  
  
int main()  
{  
    Demo obj;  
    obj.setMyValues(100, 'A');  
    obj.displayMyValues();  
    return 0;  
}
```

Output:

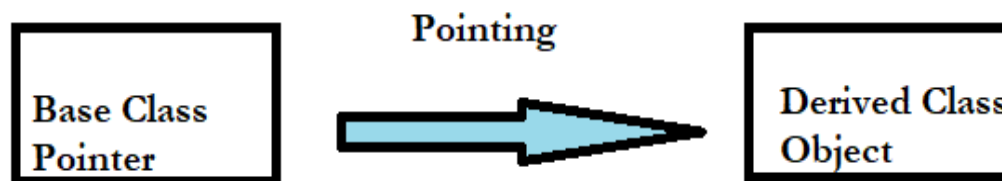
100

A

- Here you can see that we have two data members num and ch. In member function setMyValues() we have two local variables having same name as data members name.
- In such case if you want to assign the local variable value to the data members then you won't be able to do until unless you use this pointer, because the compiler won't know that you are referring to object's data members unless you use this pointer.
- This is one of the example where you must use **this** pointer.

## 2.4 Pointers to derived types :

- In general, a pointer of one type cannot point to an object of a different type. However, there is an important exception to this rule that relates only to derived classes.
- To begin, assume two classes called **B** and **D**. Further, assume that **D** is derived from the base class **B**. In this situation, a pointer of type **B \*** may also point to an object of type **D**. More generally, a base class pointer can also be used as a pointer to an object of any class derived from that base



We Point Derived class object using Base Class Pointer

```
class base
{
    //Data Members
    //Member Functions
};
class derived: public base
{
    //Data Members
    //Member functions
};
void main ( )
{
    base *ptr;
    //pointer to class base
    derived obj ;
    ptr = &obj ;
    //indirect reference obj to the pointer
    //Other Program statements
}
```

## What is Upcasting?

Upcasting is converting a derived-class reference or pointer to a base class. In other words, upcasting allows us to treat a derived type as though it were its base type.

```
#include<iostream.h>
class base
{
    public:  int n1;
    void show(){
        cout<<"\nn1 = "<<n1;
        }
};

class derive: public base
{
    public:
        int n2;
        void show(){
            cout<<"\nn1 = "<<n1;
            cout<<"\nn2 = "<<n2;
        }
};
```

```
int main()
{
    base b;

    base *bptr; //base pointer cout<<"Pointer of base
                class points to it";

    bptr=&b; //address of base class
    bptr->n1=23; //access base class via base pointer
    bptr->show();

    derive d;

    cout<<"\\n";

    bptr=&d; //address of derive class
    bptr->n1=63;

    //access derive class via base pointer
```

```
    bptr->show() ;  
    return 0;  
}
```

### Output:

The pointer of the base class points to it n1 = 23

The pointer of base class points to derive a class n1=63

The derived class inherits all members and member functions of a base class. Another name for Derived class is sub-class. It can inherit properties and methods of Base Class. All the code can be reused.

## 2.5 Pointers to class Members

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;
```

**Both declaration and assignment can be done in a single statement too.**

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

## Using Pointers with Objects

For accessing normal data members we use the dot . operator with object and -> with pointer to object.

But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

**Object.\*pointerToMember**

and with pointer to object, it can be accessed by writing,

**ObjectPointer->\*pointerToMember**

Lets take an example, to understand the complete concept

```
class Data{
public:    int a;
        void print()
        { cout << "a is "<< a; }
};

int main()
{
    Data d, *dp;
    dp = &d;          // pointer to object

    int Data::*ptr=&Data::a;  // pointer to data member 'a'

    d.*ptr=10;
    d.print();

    dp->*ptr=20;
    dp->print();
}
```

**Output :**

a is 10

a is 20

**Module - 2**

**Chapter 2 :**

# **Functions Overloading, Copy Constructors:**

## 2.6 Functions Overloading

- Two or more functions can have the same name but different parameters; such functions are called function overloading in C++.
- The function overloading in the C++ feature is used to improve the readability of the code. It is used so that the programmer does not have to remember various function names.
- If any class has multiple functions with different parameters having the same name, they are said to be overloaded.
- The easiest way to remember this rule is that the parameters should qualify any one or more than one of the following conditions:
  - They should have a different type
  - They should have a different number
  - They should have a different sequence of parameters.

An overloaded function must have:

- Different types of parameter
- Different number of parameter

**Example:**

```
void myfun();
```

```
void myfun(int a);
```

```
void myfun(float a);
```

```
void myfun(int a, int b);
```

```
void myfun(int a, double b);
```

### **Advantages**

- Memory space is saved by using function overloading.
- Function overloading allows us to get different behavior with the same function name.
- Execution of the program becomes fast.
- Function overloading is used for code reusability.
- Maintaining and debugging code becomes easy.

```
#include <iostream>
using namespace std;
class addition
{
Public:
    void add(int a, int b)
    {
        cout << "sum = " << (a + b);
    }

    void add(double a, double b)
    {
        cout << endl << "sum = " << (a + b);
    }
}

// Driver code
int main()
{
    add(10, 2);
    add(5.3, 6.2);
    return 0;
}
```

### Output

sum = 12

sum = 11.5

## 2.7 Overloading Constructor Functions

- We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading.
- Overloaded constructors essentially have the same name (exact name of the class) and different by number and type of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

Here is the code syntax for the declaration of constructor overloading in C++:

```
class ClassName {
    public:
        ClassName() {
            body; // Constructor with no parameter.
        }
        ClassName(int x, int y) {
            body; // Constructor with two parameters.
        }
        ClassName(int x, int y, int z) {
            body; // Constructor with three parameters.
        }
        ClassName(ClassName & object) {
            body; // Constructor with the same class
object as a parameter.
        }
        // Other member functions.
};
```

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

```
class construct
{
public:
    float area;

    // Constructor with no parameters
    construct()
    {
        area = 0;
    }

    // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }
}
```

```
void disp()
{
    cout<< area<< endl;
}

};

int main()
{
    // Constructor Overloading
    // with two different constructors of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

**Output:**

**0  
200**

## 2.8 Copy Constructors

- Copy constructors are the member functions of a class that initialize the data members of the class using another object of the same class.
- It copies the values of the data variables of one object of a class to the data members of another object of the same class. A copy constructor can be defined as follows:

```
class class_name {  
    Class_name(Class_name &old_object){ //copy constructor  
        Var_name = old_object.Var_name;  
        ...  
    ... } }
```

A copy constructor in C++ is further categorized into two types:

1. Default Copy Constructor
2. User-defined Copy Constructor

**Default Copy Constructors:** When a copy constructor is not defined, the C++ compiler automatically supplies with its self-generated constructor that copies the values of the object to the new object.

```
#include <iostream>
using namespace std;
class A
{
    int x, y;
public:
    A(int i, int j)
    {
        x = i; y = j;
    }
    int getX()
    { return x; }
    int getY()
    { return y; }
};
```

```
int main()
{
    A ob1(10, 46);
    A ob2 = ob1; // 1
    cout << "x = " << ob2.getX() << " y = " << ob2.getY();
    return 0;
}
```

*Output:*

**X=10    Y=46**

Here, in line 1, even without the copy constructor, the values of ob1's variable members copy fine to the member variables of ob2.

**User-defined Copy Constructors:** In case of a user-defined copy constructor, the values of the parameterized object of a class are copied to the member variables of the newly created class object. The initialization or copying of the values to the member variables is done as per the definition of the copy constructor.

```
#include <iostream>

using namespace std;

class Example
{
    public: int a;

        Example(int x) // parameterized constructor
            { a=x; }

        Example(Example &ob) // copy constructor
            { a = ob.a; }

};
```

```
int main()
{
    Example e1(36);
    // Calling the parameterized constructor
    Example e2(e1);
    // Calling the copy constructor
    cout<<e2.a;
    return 0;
}
```

Output :

36

## 2.9 Default Function Arguments

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

// defining the default arguments
void display(char = '*',int = 3);

int main()
{
    int count = 5;
    cout << "No argument passed: ";

    // *, 3 will be parameters
    display();
    cout << "First argument passed: ";

    // #, 3 will be parameters
    display('#');
    cout << "Both arguments passed: ";
```

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument.

In case any value is passed, the default value is overridden

```
// $, 5 will be parameters
    display('$', count);
    return 0;
}

void display(char c, int count)
{
    for(int i = 1; i <= count; ++i)
    {
        cout << c;
        cout << endl;
    }
}
```

Output :

No argument passed: \*\*\*

First argument passed: ###

Both arguments passed: \$\$\$\$

1. `display()` is called without passing any arguments.

In this case, `display()` uses both the default parameters `c = '*'` and `n = 1`.

2. `display('#')` is called with only one argument. In this case, the first becomes `'#'`.

3. The second default parameter `n = 1` is retained.

3. `display('#', count)` is called with both arguments.

4. In this case, default arguments are not used.

## 2.10 Function Overloading and Ambiguity

- If the compiler can not choose a function amongst two or more overloaded functions, the situation is Ambiguity in Function Overloading.

```
void Fun (int score, int total , float sgpa)  
void Fun ( Float average, char grade)
```

- In the above example, it can be obvious for us to see how can a call to an overloaded function be ambiguous.
- As the first function has three parameters namely, score, total and grade, having three data types orderly as, integer, integer and float.
- The second function has two parameters, average with integer data type and grade with the character data type.

```
#include<iostream>
using namespace std;
void test(float s,float t)
{
    cout << "Function with float called ";
}
void test(int s, int t)
{
    cout << "Function with int called ";
}
int main()
{
    test(3.5, 5.6);
    return 0;
}
```

It may appear that the call to the function test in main() will result in output “Function with float called” but the code gives following error:

In function 'int main()':

13:13: error: call of overloaded 'test(double, double)' is ambiguous test(3.5,5.6);

**Rectifying the error:** We can simply tell the compiler that the literal is a float and NOT double by providing **suffix f**. Look at the following code :

```
#include<iostream>
using namespace std;
void test(float s, float t)
{
    cout << "Function with float called ";
}
void test(int s, int t)
{
    cout << "Function with int called ";
}
int main()
{
    test(3.5f, 5.6f); // Added suffix "f" to both values to
                      // tell compiler, it's a float value
    return 0;
}
```

**Output:**

Function with float called