OBJECT ORIENTED PROGRAMMING IN C++ [UNIT-II]

Learning Objectives

- Creating a class with different access specifier
- Defining member functions
- Using objects
- Inline member functions
- Nested member functions
- Static data member
- Static member functions
- Nested Class

Learning Objectives

- Making functions friendly to a function
- Bridge functions
- Function returning objects

CLASSES

- Most important features of C++ are the classes & Objects.
- Class represents group of similar objects.
- A class is a way to bind the data describing an entity and its associated functions together.
- Class can be used for creating the user defined data types.

CLASSES

- Class instantiation is a way to create objects of this type.

  Syntax: class classname objectname;
  class emp e1,e2, e3;
  OR classname objectname;
  emp e1,e2;
- The members of the classes are the private by default.
- Classes are needed to represent real-world entities.

CLASSES & STRUCTURES

- The main difference between a structure and a class in C++ is that, by default, the members of a class are private while, by default, the members of structure are public.
- Like structure, the data members of the class can not be initialized during their declaration, but they can be initilaized by its members as follows:

```
class float1
 {
……..
 float x, y = 5; // Error data members can not be initialised here

void setfloat()
  {
   x = y = 0.0;
   }
 };
```


CLASSES & STRUCTURE

- Another difference, In C++ the convention of defining objects at the point of class specification is rarely followed, the user would define the objects as and when required, or at the point of their usage.

```
class student
 {
  ……
  ……..
  } s1, s2, s3, s4;
```


SPECIFYING A CLASS

General Format:

```
class classname {
        private:
        [variable declaration;]
        [function declaration;]
        Protected:
        [variable declaration;]
        [function declaration;]
        Public:
        [variable declaration;]
        [function declaration;]
    };
```


SYNTAX

Syntax for Accessing Class Members

ObjectName.DataMember;
e.g; s1. name;

ObjectName.MemberFunction(Actual Para);
e.g; s1.setdata(11, “Anil”);

Defining Class Member Functions

Member functions can be defined in any of the following ways:

- Inside the class specification

```
class classname
{
ReturnType MemberFunction(arguments);
};
```

- **Outside the class specification**

```
ReturnType ClassName :: MemberFunction(arguments)
{
//body of the function
}
```


Example

Member Functions Defined Inside the Body of the Student Class

Example

```
#include <iostream.h>
#include <string.h>
class student
{
private:
int rollno;
char name[20];
public:
```


Example

```
void setdata( int rn, char *namein )
{rollno =rn;
strcpy(name, namein );
}
void outdata()
{cout<<"Roll no ="<<rollno<<endl;
cout<<"Name ="<<name<<endl;}
};
void main(){
student s1, s2, s3;
s1.setdata(1, "xx");
s2.setdata(2, "yy");
s1.outdata();
s2.outdata();
}
```


Member Functions Inside the Class

- The syntax is similar to a normal function definition except that it is enclosed within the body of a class.
- Considered as inline function by default.
- Normally functions with small body are defined inside the class specifications.

 U2.13

Member Functions Outside the Class

- Requires a mechanism of binding the function to the class to which they belongs.
- Done by using the scope resolution operator(::)
- Acts as an identity label to inform the compiler, the class to which the function belongs.

 U2.14

Member Function Characteristics

- A program can have several classes.
- Private members of the class can be accessed by all the members of the class, whereas nonmember functions are not allowed to access .
- However friend functions can access them.
- Member functions of the same class can access all other members of their own class without the use of dot operator.

 U2.15

Member Function Characteristics

•Member functions defined as public act as an interface between the service provider (class) and the service seeker (object).

•A class can have multiple member functions with the same name as long as they differ in terms of argument specifications.

Data Hiding

There are three kind of access control specifiers:

•Private
•Public
•Protected

Private Members

•Strict access control.

•Member functions of the same class can access.

•A mechanism for preventing accidental modifications of the data members.

Program to Private Access

```
#include <iostream.h>
class student
{private:
int rollno;
char name[20];
void setdata(int rn, char* namein );
void outdata();
};
void main(){
student s1;
s1.setdata(1, “xx”); //can’t access
s1.outdata(); //can’t access
}
```


Protected Members

Access control is similar to private members, has more significance in Inheritance.

Protected Members Contd..

```
#include <iostream.h>
class student
{
protected:
int rollno;
char name[20];
void setdata(int rn, char  *namein);
void outdata();
};
void main()
{
student s1;
s1.setdata(1, “xx”); //can’t access protected (same as private)
s1.outdata();  //can’t access
}
```


Public Members

- Visible outside the class
- Can be accessed without any restriction from anywhere in the program.

Public Members Contd..

```
#include <iostream.h>
class student
{public:
int rollno;
char name[20];
void setdata(int rn, char *namein);
void outdata();
};
void main()
{student s1;
s1.rollno; // can access public data
s1.setdata(1, "xx"); //can access public function
s1.outdata(); //can access public function
}
```


Access Boundary Of Objects

Access specifier	Accessible to own class member	Accessible to objects of a class
Private:	Yes	No
Protected:	Yes	No
Public:	Yes	Yes

EMPTY CLASSES / STUB

A class that has neither data nor code.

e.g; class xyz { };

```
class abc
{
};
```

At initial level of project, some of the classes are not fully identified or not fully implemented, hence they are implemented as empty class.

Important for Exception Handling.

Passing Objects as Arguments

An object can be passed as an argument to a function by the following way

- Pass-by-value
- Pass-by-reference
- Pass-by pointer

Returning Objects from functions:

- It is possible to return objects from functions. Syntax is similar to that of returning variables from function.

Program

```
#include<iostream.h>
class complex{
private:
float real;
float imag;
public:
void getdata();
void outdata();
complex add (complex c2);};
complex complex:: add(complex c2)
{complex temp;
temp.real = real + c2.real;
temp.imag = imag + c2. imag;
return temp;
}
```


Program

```
void main()
{
complex c1, c2, c3;
c1.getdata();
c2.getdata();
c3 =c1.add(c2);
c1.outdata();
c2.outdata();
c3.outdata();}
```


Static Data and Member Functions

- Only one copy of static member exists for all the instances of a class.
- Static member functions can access only static members of its class.
- Static data members must be defined and initialize like global variables, other wise the linker generates error.
- Static members define as public can either be accessed through the scope resolution operator with the class name or it can be accessed through the object of a class.

ClassName:: MemberName;
ObjectName.MemberName;

Program for Static Member Access

```
#include <iostream.h>
class MyClass{
static int count ; // static member
int number ;
public :
// initializes object's member and increments function call
void set ( int num ){
number = num ;
++count ;}
void show ( ){
cout << "\nNo. of calls made to set ( ) through any object: "
cout<< count ;
}} ;
```


Program for Static Member Access

```
// static member count is shared by all the objects of class MyClass
int MyClass::count = 0;
void main ( ){
MyClass Obj1 ;
Obj1.show( ) ;
Obj1.set ( 100 ) ;
Obj1.show ( ) ;
MyClass Obj2, Obj3 ;
Obj2.set ( 200 ) ;
Obj2.show ;
// same result even with obj1.show and Obj3.show ( ) ;
Obj2.set ( 250 ) ;
Obj3.set ( 300 ) ;
Obj1.show ( ) ;
// same result even with obj2.show and obj3.show ( ) ;}
```


Class, Objects and Memory Resources

- When an object is created, memory is allocated only to its data members and not to member Function.
- Member functions are created and stored in memory only once when a class specification is declared.
- Member function are same for all objects.
- Storage Space for data members which are declared as static is allocated only once during the class declarations.

Conclusion

- A class is an extension to the structure data type. A class can have both variables and functions as members with different types of access specifiers.
- Its significance is highlighted by the fact that Stroustrup initially gave the name "C with Classes" to his new language.
- A data member of a class can be declared as a static and is used to maintain the values common to the entire class.
- A static member function can have access to the static members declared in the same class and can be called using the class name.

Conclusion

- C++ also allows us to have arrays of objects.
- We may use objects as function arguments.
- A function can also return an object.
- A function declared as function has full access to the private member of the class.
- The keyword const member function does not allow to alter any data in the class.

 U2.34

CONSTRUCTORS & DESTRUCTORS

 U2.35

Learning Objectives

- **Constructors**
- **Default argument constructor**
- **Constructor overloading**
- **Copy constructor**
- **Parameterized constructors**
- **Dynamic initialization**
- **Dynamic Constructors**
- **Destructors**

 U2.36

Constructors and Destructors

- A member function with the same name as its class is called a constructor. It is used to initialize the objects of that class-type with a some initial value.
- If a class has a constructor, each of the objects of that class will be initialized before they are used within the program.
- A constructor for a class is needed, so that the compiler automatically initializes an object as soon as it is created.
- A class constructor is defined without any return-type.
- They are called whenever a program creates an object of that class.

Declaration and Definition

- A constructor is defined like any other member function of a class except that it does not have any return-type associated with it.
- It can be defined either inside the class definition or outside the class definition.

Declaration and Definition

```
class salesperson
{
        private:
                int person_identity ;
                float daily_sales ;
        public:
                salesperson() //default constructor
                {
                        person_identity = 0 ;
                        daily_sales = 0.00 ;
                }
};
```

Note: A constructor can be defined in any of the private, protected or public section.

Default Constructors

- A constructor that does not accept any parameters is called default constructor. In the previous example **salesperson::salesperson()** is the default constructor for class **salesperson,** since it does not take any parameters.
- A default constructor supplied by the compiler does not do any thing special; it just initializes the data members with dummy values.
- **Note:** If a class has no explicit constructor defined, the compiler will supply a default constructor, having no arguments.

Declaration and definition

- A constructor is defined like any other member function of a class except that it does not have any return-type associated with it.
- A constructor can be defined in all the three public, private and protected sections. But generally a constructor should be defined under the public section of the class so that its objects can be created in any function.
- Constructor can be defined either inside the class definition or outside the class definition.

Declaration and definition

Code fragment defines a constructor inside the class definition:

```
class X { int i;
          public:
          int j, int k;
          X() { i=j=k=0;  //constructor
              }
          :
         };
```

The simple constructor X::X() is as an inline member function here.

Declaration and definition

Code fragment defines a constructor inside the class definition:

```
class X { int i;
          public:
          int j, int k;
          X() { i=j=k=0;  //constructor
              }
          :
         };
```

The simple constructor X::X() is an inline member function here.

Declaration and definition

Code fragment can also be defined for a constructor outside the class definition:

```
class X { int i;
          public:
          int j,  k;
          X();  // only declaration as a prototype
          :      // other members
          };
          X ::X() //constructor defined outside
          { i = j = k = 0;

         }
```


Need for Constructors

A structure or an array in c++ can be initialized at the time of their declaration. For example;

```
struct student { int rollno;
                 float marks;
               };
int main()
{student s1= {0, 0.0};
 }

OR, int a[5] = {1,2,3,4,5};
```

Need for Constructors

- But such initialization does not work for a class because class members have their associated access specifier. They might not be available to the outside world (outside their class). The following code snippet is invalid in C++:

```
class student { private:
                int rollno;
                float marks;
                 public:
                   :   // public members
                };
int main()
{ student senior = { 0, 0.0}; // illegal ! Data members are not accessible.
          :
          :
};
```

The private data members are inaccessibe by a non-member function in main().

Need for Constructors

- There can be an alternative solution to it. If we define a member function (say init() inside a class to provide the initial values, as it is shown below:

```
class student { private:
                int rollno;
                 float marks;
                public:
                void init()
                { rollno = 0;
                  marks=0.0;
               }
                :   //other  public members
                };
int main()
{ student senior ; // create an object
  senior. init();     // initialize it
          :
          :
}
```


Need for Constructors

- Here the programmer has to explicitly invoke the function called init(), in order to initialise the object.
- Thus the responsibility of initialization lies solely with the programmer . What if, the programmer fails to invoke init()? The object in such case is full of garbage and might cause havoc to your program.
- Thus the responsibility of initialisation is taken away from the programmer and given to the compiler because the compiler exactly knows when objects are created.
- Therefore every time an object is created , the compiler will automatically initialise it by invoking the initialisation function but if and only if the initialization function bears the same name as that of the class.
- And, Obviously this function is known as a constructor.

c:\tcwin45\bin\noname01.cpp

```cpp
#include <iostream.h>
class Rectangle
{
      private:
         float length ;
         float breadth ;
      public:
         Rectangle( )                                  // Default constructor, without any argument
         {          }
         Rectangle(float l, float b)                   // Constructor with two arguments
         {
            length = l ;
            breadth = b ;
         }
         void Enter_lb(void)
         {
            cout << "\n\t Enter the length of the rectangle: " ;
            cin >> length ;
            cout << "\t Enter the breadth of the rectangle: " ;
            cin >> breadth ;
         }
         void Display_area(void)
         {
             cout << "\n\n\t The area of the rectangle = " << length*breadth ;
         }
};                                                     // End of the class definition
void main(void)
{
      Rectangle r1 ;                 // Invokes the first constructor without any argument
      cout << "\n First rectangle-------->\n" ;
      r1.Enter_lb( );
      r1.Display_area( );
      cout << "\n\n Second rectangle------->\n" ;
      Rectangle r2(5.6, 3.5);             //Invokes the second constructor with two arguments
      r2.Display_area( );
}
```

Output

(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

```
First rectangle-------->

        Enter the length of the rectangle: 4
        Enter the breadth of the rectangle: 5

        The area of the rectangle = 20

Second rectangle------->

        The area of the rectangle = 19.6
```


Copy Constructor

- A copy constructor is a constructor of the form **classname(classname &)**.
- The compiler will use the copy constructor whenever you initialize an instance, using values of another instance of the same type.
- Example:

```cpp
salesperson S1 ; // default constructor used
```

- The second statement will copy the instance of S1 to S2.


```
c:\tcwin45\bin\noname01.cpp
#include <iostream.h>
class Sample
{
      private:
         int i,j;
      public:
         Sample(int a, int b)                          // constructor, with two argument
         {
            i=a;
            j=b;
         cout<<"\nParameterized constructor working\n";
         }
         Sample(Sample &S)                            // copy Constructor
         {
            j=S.j;
            i=S.i;
            cout<<"\nCopy constructor working\n";
         }
         void print()
         {
             cout <<i<<"\n"<<j;
         }
};                                                    // End of the class definition
void main()
{
      Sample S1(4,5) ;                            // Invokes the constructor
      Sample S2(S1);            //Invokes the copy constructor
      S2.print( );
}
```

```
(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

Parameterized constructor working

Copy constructor working
4
5
```

Destructors

- Whenever an object is created within a program, it also needs to be destroyed.
- If a class has constructor to initialize members, it should also have a destructor to free up the used memory.
- A destructor, as the name suggest, destroys the values of the object created by the constructor when the object goes out of scope.
- A destructor is also a member function whose name is the same name as that of a class, but is preceded by tilde ('~').
- For example, the destructor of class **salesperson** will be **~salesperson()**.
- A destructor does not take any arguments nor does it return any value. The compiler automatically calls them when the objects are destroyed.


```
#include <iostream.h>
class Sample
{
      private:
         int i,j;
      public:
         Sample(int a, int b)                          // constructor, with two argument
         {
            i=a;
            j=b;
         cout<<"\nConstructor working\n";
         }
         void print()
         {
             cout<<"\nValue of object of sample class\n";
             cout <<i<<"\n"<<j<<"\n";
         }
         ~Sample()
         {
            cout<<"\nDestructor is working";
         }
};                                                    // End of the class definition
void main()
{
      Sample S1(4,5) ;                            // Invokes the constructor
      S1.print( );
}
```

```
(Inactive C:\TCWIN45\BIN\NONAME01.EXE)

Constructor working

Value of object of sample class
4
5

Destructor is working
```

Need for a Destructor

- During the construction of the object by the constructor, some resources may be allocated for use.
- For example, some memory space may be allocated to the data members. These resources must be de-allocated and the memory space should be freed before the object is destroyed.

Copy Constructor

- A copy constructor is a class constructor that can be used to initialize one object with the values of another object of the same class during the declaration statement.
- In simple words, we can say that, if we have an object called myObject1 and we want to create a new object called myObject2, initialized with the contents of myObject1 then the copy constructor should be used.
- Consider the declaration given below:

```
myClass myObject1; … (1)
myClass myObject2(myObject1); … (2)
```

•The declaration (1) declares an object myObject1 of class myClass.

•The declaration (2) declares another object myObject2 of class myClass as well as initializes it with the contents of existing object myObject1.

Copy Constructor

The copy constructor is, however, defined in the class as a parameterized constructor receiving an object as argument passed by reference.

```
class myClass
{
        -
        public:
                myClass (myClass &myObject)
                {
                }
};
```


Program for Copy Constructor

```
class address
{
        private:
                int houseno;
                char street[10];
                char city[20];
                char state[20];
        public:
                address(int I, char x[10], char y[20], char z[20]) // Parameterized Constructor
                {
                        houseno = i;
                        strcpy(street, x);
                        strcpy(city, y);
                        strcpy(state, z);
                }

                address(address &myobject); // Copy Constructor
                {
                        houseno = myobject.houseno;
                        strcpy(street, myobject.street);
                        strcpy(city, myobject.city);
                        strcpy(state, myobject.state);
                }

                void display();
```

Copy Constructor

- In the above given class, two constructors have been used; a parameterized and a copy constructor sharing a common class name i.e. address. Thus, constructor can be overloaded.
- It may be noted that the code for copy constructor has to be written by the programmer.
- It may be further noted that the argument to a copy constructor has been passed by reference.
- We can not pass the argument by value because this will result in copy calling itself on and on until the compiler runs out of memory.
- Let us now write a program that uses this class to create an object called obj1 with some initial values. It also creates another object called obj2 and copies the contents of obj1 into obj2. It then displays the contents of obj2.

Turbo C++ - [c:\tcwin45\bin\copyconstructor.cpp]

```
#include<iostream.h>
#include<string.h>
class address
{  private:
      int houseno;
      char street[10];
      char city[20];
      char state[20];
   public:
      address(int i, char x[10], char y[20], char z[20])     //Parameterized Constructor
      {
         houseno = i;
         strcpy(street, x);
         strcpy(city, y);
         strcpy(state, z);
      }
      address(address &ob);                                  //Copy Constructor
      void display();
};                                                          //Define Copy Constructor
address::address(address &ob)
{
   houseno = ob.houseno;
   strcpy(street, ob.street);
   strcpy(city, ob.city);
   strcpy(state, ob.state);
}
void address::display()
{
   cout<<"\n House number    =   "<<houseno;
   cout<<"\n Street       =   "<<street;
   cout<<"\n City       =   "<<city;
   cout<<"\n State       =   "<<state;
}
void main()
{
   address obj1(153,"xyz","abc","kjh");                     //Create obj1
   address obj2(obj1);                                      //Create obj2 and initialize with contents of obj1
   obj2.display();                                          //Display the contents of object2
```

Output

```
(Inactive C:\TCWIN45\BIN\COPYCONS.EXE)

House number    =    153
Street          =    xyz
City            =    abc
State           =    kjh
```


Explanation

•An object called **obj1** of class **address** is created with some initial values. Another object **obj2**, a copy of obj1, is created with the help of the copy constructor.

•At this stage, a question which arises is that, this job could have been done by simple assignment of objects i.e. **obj1 = obj2,** then why to use a copy constructor?

•The need of a copy constructor is felt when the class includes pointers which need to be properly initialized. A simple assignment will fail to do this, because both the copies will hold pointers to same memory location.

Deep & Shallow Coping

•**Shallow copy:** when we doesn't define our copy constructor and assignment operator, then compiler defines copy constructor and assignment operator for us and it provides a copying method known as a **shallow copy** ,also known as a **memberwise copy**.

A **shallow copy** copies all of the member variable values. This works efficient if all the member variables are values, but may not work well if member variable point to dynamically allocated memory. The pointer will be copied but the memory it points to will not be copied, the variable in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want. The default copy constructor and assignment operator make shallow copies.

- **Deep copy:**A deep copy copies all member variables, and makes copies of dynamically allocated memory pointed to by the variables. To make a **deep copy**, we have to define our copy constructor and overload the assignment operator.

  Requirements for deep copy in the class

  • A destructor is required to delete the dynamically allocated memory.
  • A copy constructor is required to make a copy of the dynamically allocated memory.
  • An overloaded assignment operator is required to make a copy of the dynamically allocated memory.

Function Overloading

- Concept that allows multiple functions to share the same name with different argument types and numbers.
- Function definition can have multiple forms
- Doesn't permit overloading of functions differing only in their return value


```
#include <iostream.h>

void ConvertFToC(double f, double &c);
void ConvertFToC(float f, float &c);
void ConvertFToC(int f, int &c);
int main()
{
   double df, dc;
   float ff, fc;
   int i_f,i_c;    //if is a reserved word
   df = 75.0;
   ff = 75.0;
   i_f = 75;
```



```
    // The compiler resolves the correct
    // version of ConvertFToC based on
    // the arguments in each call
    cout << "Calling ""double"" version" << endl;
    ConvertFToC(df,dc);
    cout << df << " == " << dc << endl << endl;

    cout << "Calling ""float"" version" << endl;
    ConvertFToC(ff,fc);
    cout << ff << " == " << fc << endl << endl;

    cout << "Calling ""int"" version" << endl;
    ConvertFToC(i_f,i_c);
    cout << i_f << " == " << i_c << endl << endl;
}
```



```
void ConvertFToC(double f, double &c)
{
    cout << "In ""double"" version" << endl;
    c = (f - 32.0) * 5. / 9.;
}

void ConvertFToC(float f, float &c)
{
    cout << "In ""float"" version" << endl;
    c = (f - 32.0) * 5. / 9.;
}
void ConvertFToC(int f, int &c)
{
    cout << "In ""int"" version" << endl;
    c = (f - 32) * 5. / 9.;
}
```


Exercise

```
int pow(int, int);
double pow(double, double);
complex pow(complex , double);
complex pow(complex , int);
complex pow(double, complex);
complex pow(complex, complex);
void k(complex z)
{
int i = pow(2,2);
double d = pow(2.0,2.0);
complex z1 = pow(2, z);
complex z2 = pow(z, 2);
complex z1 = pow(z, z);
double d = pow(2.0,2);
}
```


POLYMORPHISM

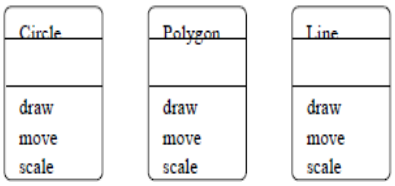
 U2.70

Learning Objectives

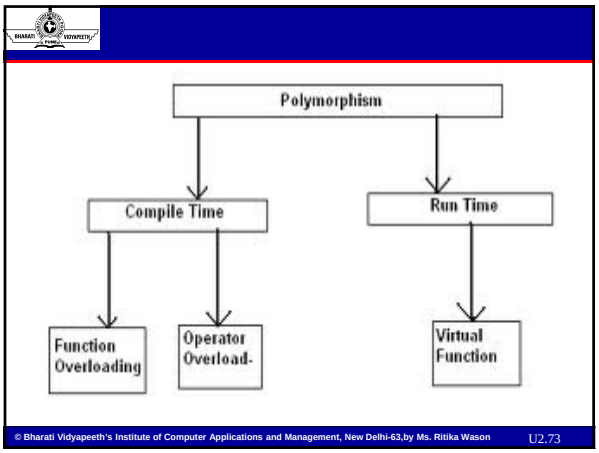
- Polymorphism
- Categorization of Polymorphism techniques
- Function Overloading
- Operator Overloading
- Run type polymorphism/ Virtual Function

 U2.71

Polymorphism is the quality that allows one name to be used for two or more related but technically different purposes. In the following, each graphical object has the same services, although they are implemented differently.



 U2.72



POLYMORPHISM / OVERLOADING

- A Greek term suggest the ability to take more than one form.
- It is a property by which the same message can be sent to the objects of different class.

  Example: Draw a shape (Box, Triangle, Circle etc.), Move (Chess, Traffic, Army).
- Allows to create multiple definition for operators & functions.

  Example: '+' is used for adding numbers / to concatenate two string / Sets of Union and so on.
- There are two types of polymorphism, compile time polymorphism and run time polymorphism. It is also known as early or static binding and run time binding.

 U2.74

POLYMORPHISM (Contd.)

- Function and operator overloading is the example of compile time polymorphism and virtual function is the example of run time polymorphism.
- A Virtual function, equated to zero is called pure virtual function.
- Dynamic Binding/ Late Binding. Run-time dependent. Execution depends on the base of a particular definition.
- Extensively used in implementing inheritance.

 U2.75

Compile time polymorphism

- involves binding of functions based on the
  - number of arguments,
  - type of arguments
  - sequence of arguments.
- This information is known to the compiler at complie time. So compiler is able to select the appropriate function for a particular call at the complie time itself.

Cont….

- The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time.
- This form of association is called early binding. The term early binding stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.

Runtime Polymorphism

- refers to an entity changing its form depending on circumstances.
- A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed

Cont…..

- The term ***late binding*** refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.

Function Overloading

•Concept that allows **multiple functions to share the same name with different argument types and numbers.**
•Function definition can have multiple forms
•Doesn't permit overloading of functions differing only in their return value

Problem Definition :Define the function area () to compute the area of objects of different classes triangle, rectangle, square. Invoke these in the main program.


```
#include<iostream.h>
#include<conio.h>
void area(double length, float breadth)
{
int area;
area=length*breadth;
cout<<"\nthe area of the rectangle:"<< area;
}
void area(float side)
{
        float area;
        area=side * side;
        cout<<"\nthe area of the square:"<<area;
}
void area( int breadth, int height)
{
        float area;
        area= (breadth * height)/2;
        cout<<" \nthe area of the triangle:"<<area;
}
void main()
{
        clrscr();
        area(12);
        area(10.0,23.0);
        area(11,11);
        getch();
}
```


Function Overloading

Function overloading is a concept where several function declarations are specified with a single and a same function name within the same scope. Such functions are said to be overloaded. C++ allows functions to have the same name. Such functions can only be distinguished by their number and type of arguments.

Example:

```
float divide(int a, int b);
float divide(float a, float b);
```

The function **divide()**, which takes two integer inputs, is different from the function **divide()** which takes two float inputs.

What is the need for function overloading?

Every object has characteristics and associated behavior. An object may behave differently with change in its characteristics. Therefore, in order to simulate real world objects in programming environment, it is necessary to have function overloading.

For Example:

```
float AddNumber(float a, float b)
{
        return a + b ;
}
int AddNumber(int a, int b)
{
        return a + b ;
}
void main()
{
        cout<<Add Number( 21, 36 ) ;
        cout<<Add Number( 6.72, 2.22 ) ;
}
```

Function overloading not only implements polymorphism but also reduces number of comparisons in a program and thereby making the program run faster.

Sample for Function Overloading

C++ allows you to overload a function provided the function has the same name but different signatures. The signature can differ in the number of arguments or in the type of arguments, or both. To overload a function, all you need to do is, declare and define all the functions with the same name but different signatures.

Example:

```
void pmsqr(int i);
void pmsqr(char c);
void pmsqr(float f);
void pmsqr(double d);

void pmsqr(int i)
{
        cout<<"\n Integer "<< i <<"'s square is "<<i*i<<"\n";
}
void pmsqr(char c)
{
        cout<<"\n Character "<<c <<" thus no square "<<"\n";
}
void pmsqr(float f)
{
        cout<<"\n Float "<< f <<"'s square is "<<f*f<<"\n";
}
void pmsqr(double d)
{
        cout<<"\n Double "<<d<<"'s square is "<<d*d<<"\n";
}
```


Contd…

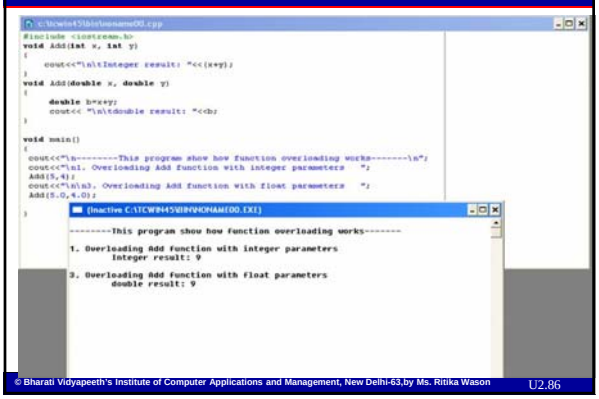
When a function, with same name, is declared more than once in the program, the compiler will interpret the second declaration as follows:

- If the signature of subsequent function matches the previous function, then the second is treated as the re-declaration o fthe first.
- If the signature of both the functions match exactly, but the return type differs, then the second declaration is treated as an erroneous re-declaration of the first and is flagged at compile time as an error.

For example,

```
float square(float f);
double square(float x); //error
```


Contd…



Function overloading

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

 class arith {
 public:
    void calc(int num1)
 {
 cout<<"Square of a given number: "
  <<num1*num1 <<endl;
 }
   void calc(int num1, int num2 )
 {
cout<<"Product of two whole
   numbers: " <<num1*num2
   <<endl;
   }
   };
int main() //begin of main funct
   ion
   {
      arith a;
      a.calc(5);
      a.calc(6,7);
   }
```


OPERATOR OVERLOADING

OPERATOR OVERLOADING

Principle areas

- Extending capability of operators to operate on user defined data
- Extends the semantics of an operator without changing its syntax
- It's not possible to change an operator's precedence.
- It's not possible to create new operators, eg ** which is used in some languages for exponentiation.

Commonly overloaded Operators

- = (assignment operator)
- + - * (binary operator)
- += -= *= (compound assignment operators)
- == != (comparison operators)

Overloadable Operators

- Allows almost all operators except
  - Member access (dot operator)
  - Scope resolution (::)
  - Conditional(?:)
  - Pointer to member(.*)
  - Size of Datatype (sizeof(…))
- General format of overloading

  Returntype operator operatorsymbol([arg1,[arg2]])
- Can be defined operators as member function and friend function

- Overloading without explicit arguments to an operator function is known as unary operator overloading
- overloading with a single explicit argument is known as binary operator overloading
- With friend function, unary operators take one explicit argument and binary operators take two explicit arguments

Binary Operator Over Loading

- General syntax

```
Returntype  operator operatorsymbol (arg )
{
// body of operator function
}
```

- Binary overloaded operator function takes the first object as an **implicit operand** and the second operand must be passed **explicitly**

- **Types**:
  There a three type of operators based on operands used, namely unary, binary, ternary.
  Unary - This works with one operand (e.g: a++, a will be incremented by 1),
  Binary - This works with two operands (e.g: a+b),
  Ternary - This works with three operands (e.g: condition? value 1 : value 2).

Unary Operators

- Operators attached to a single operand (-a, +a, --a, a--, ++a, a++)
- Example illustrates the overloading of unary operators
  - Index operator + ( ) ;
  - Index operator - ( ) ;
  - void operator ++ ( ) ;
  - void operator -- ( ) ;

- Operator Overloading
- ***Unary operator***
- It has only one operand.
- There are two main unary operators, like increment and decrement operator
- ***Increment Operator***
- The increment operator(++), which is to increment the variables or the constants.
- Increment is of two types like preincrement and postincrement operator.

- Unary operators are operators that only deal with one argument (which is generally a single variable).
  Operator Use
- ! Negation
- ++ increment by 1
- -- decrement by 1

 U2.97

Decrement Operator

- The decrement operator(–), which is to decrement the values of variables or constants.
- Its similar like increment types. But here it'll decrease the value instead increasing.

Complement

- The complement operator is used to find one's complement of the numbers.
- The symbol used for complement is ~
- For example int y=0;
- x=~y;
- cout<<x;
- The output will be 1

 U2.98

Logical NOT

- Its a Boolean operator, which has only one operand and the operand is placed in the right of the logical not operator(!).
- For Example:
- if(!a)
- {
- set of stmt to be executed
- }

 U2.99

Sizeof

- The sizeof operator is used to find the size of an array, datatype. It returns the value in **bytes.** It accepts only one operand.

For Example:

```
int a;
cout<<sizeof(a);
```

Output

2

- Binary operators are operators that deal with two arguments, both generally being either variables or constants.

Operator Use

- + addition
- - subtraction
- * multiplication
- / division
- % remainder

= assignment

> boolean greater than

< boolean less than

Ternary Operator

- Ternary operator is also known as conditional operator. It has three operands.
- The operator is ?: .
- Syntax: expr?stmt1:stmt2;
- If the expression is true, then the statement1 will be executed.
- Else statement2 will be executed.

Unary Operator Overloading

```
// Program1.cpp:  Index class with operator overloading for increment operator
 #include  < iostream.h>
class Index
{
  private:
    int  value;                    //  Index  Value
  public:
    Index ( )                      //  No argument constructor
      {
        value  =  0;
      }
    int GetIndex( )            // Index Access
      {  return value;
      }
    void operator + +( )     // prefix or postfix increment operator
       {
         value = value + 1;    // value+ + ;
       }
} ;
```


Unary Operator Overloading

```
void main( )
{  Index idx1, idx2;            // idex1 and idx2 are objects of  Index class
 // display index values
   cout << " \nIndex1 = " << idx1 . GetIndex ( ) ;
   cout << " \nIndex2 = " << idx2 . GetIndex ( ) ;

   // Advance Index objects with + + operators
  + + idx1;              // equivalent to idx1. operator + + ( ) ;
  idx2 + + ;
  idx2 + + ;
  cout  <<  " \ nIndex1 = " << idx1. GetIndex ( ) ;
  cout  <<  " \ nIndex2 = " << idx2. GetIndex ( ) ;
}
```

RUN

```
Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2
```


Example: Unary Operators

```
class UnaryExample
{
    private:
        int m_LocalInt;
    public:
        UnaryExample(int j)
        {
            m_LocalInt = j;
        }
        int operator++ ()
        {
            return (m_LocalInt++);
        }
};
```


Example: Unary Operators (contd.)

```
void main()
{
    UnaryExample object1(10);
   cout << object1++; // overloaded
  operator results in value
     // 11
}
```


Unary Operator Overloading Example

```
class MyClass
{
   //  class data or function stuff
  int operator ++ ( )    // member function definition
  {
        //  body of a function
  }
} ;
```


Unary operator using friend functions

```
Class test{
   int x;
   public:
        test(){ x= 5;}
        test(int a){ x= a;}
   }
   friend test operator++(test& ob);
};
test operator ++(test& ob)
{
        ob.x++;
        return (test(ob.x));
};
```


Binary Operators

- Operators attached to two operands (a-b, a+b, a*b, a/b, a%b, a>b, a>=b, a<b, a<=b, a==b)

Example: Binary Operators

```
class BinaryExample
{
    private:
        int m_LocalInt;
   public:
        BinaryExample(int j)
       {
           m_LocalInt = j;
       }
       int operator+ (BinaryExample& rhsObj)
       {
           return (m_LocalInt + rhsObj.m_LocalInt);
       }
};
```


Example: Binary Operators (contd.)

```
void main()
{
    BinaryExample object1(10), object2(20);
   cout << object1 + object2; // overloaded
  operator called
}
```


An operator function is created using the keyword **operator. Operator functions** can be either members or nonmembers of a class. Nonmember operator functions are almost always friend functions of the class.
A member operator function takes this general form:

ret-type class-name::operator#(arg-list)
{
// operations
}

•operator functions return an object of the class they operate on, but *ret-type* can be any valid type.
•The # is a placeholder. When you create an operator function, substitute the operator for the #. For example, if you are overloading the / operator, use operator/. When you are overloading a unary operator, *arg-list will be empty.*
•When you are overloading binary operators, *arg-list will contain one parameter.*


```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
};
```

```
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```


operator+( ) has only one parameter even though it overloads the binary **+ operator. (You might expect two parameters corresponding to the two operands** of a binary operator.) The reason that **operator+( ) takes only one parameter is that the operand on the left side of the + is passed implicitly to the function through the this pointer. The operand on the right is passed in the parameter op2.**

The fact that the left operand is passed using **this also implies one important point: When binary operators** are overloaded, it is the object on the left that generates the call to the operator function.

ob1 = ob1 + ob2;

In order for the sum of **ob1 and ob2 to be assigned to ob1, the outcome of that operation** must be an object of type **loc.**

It is important to understand that an **operator function can return any type** and that the type returned depends solely upon your specific application. It is just that, often, an operator function will return an object of the class upon which it operates.

EX-

```
class loc {
int longitude, latitude;
public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
```



```
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
```



```
// Overload asignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload prefix ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
```



```
int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
ob1.show();
ob2.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob1.show(); // displays 12 22
ob2.show(); // displays 12 22
ob1 = ob2 = ob3; // multiple assignment
ob1.show(); // displays 90 90
ob2.show(); // displays 90 90
return 0;
}
```


In C++, if the **= is not overloaded, a default assignment operation is created automatically** for any class you define. The default assignment is simply a member- by-member, bitwise copy. By overloading the **=, you can define explicitly what the assignment does** relative to a class. In this example, the overloaded **= does exactly the same thing as** the default, but in other situations, it could perform other operations. Notice that the **operator=( ) function returns *this, which is the object that generated the call. This** arrangement is necessary if you want to be able to use multiple assignment operations such as this:

ob1 = ob2 = ob3; // multiple assignment

look at the definition of **operator++( ). As you can see, it takes no parameters.** Since **++ is a unary operator, its only operand is implicitly passed by using the this pointer.**

Overloading the Shorthand Operators You can overload any of C++'s "shorthand" operators, such as **+=, –=, and the like.**For example, this function overloads **+= relative to loc:**

```
loc loc::operator+=(loc op2)
{
longitude = op2.longitude + longitude;
latitude = op2.latitude + latitude;
return *this;
}
```

When overloading one of these operators, keep in mind that you are simply combining an assignment with another type of operation.

Operator Overloading Using a Friend Function

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a **friend function is not a member of the class, it** does not have a **this pointer. Therefore, an overloaded friend operator function is passed** the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.


```
class loc {
int longitude, latitude;
public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
friend loc operator+(loc op1, loc op2); // now a friend
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
```



```
// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
loc temp;
temp.longitude = op1.longitude + op2.longitude;
temp.latitude = op1.latitude + op2.latitude;
return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
```



```
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
```


There are some restrictions that apply to friend operator functions. First, you may not overload the **=, ( ), [ ], or –> operators by using a friend function. Second, as** explained in the next section, when overloading the increment or decrement operators, you will need to use a reference parameter when using a friend function.

Using a Friend to Overload ++ or – –

If you want to use a friend function to overload the increment or decrement operators, you must pass the **operand as a reference parameter.** This is because **friend functions do not have this pointers.**


```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator=(loc op2);
friend loc operator++(loc &op);
friend loc operator--(loc &op);
};
```

```
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this;        // i.e., return object that generated call
}
```



```
// Now a friend; use a reference parameter.
loc operator++(loc &op)
{
op.longitude++;
op.latitude++;
return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
op.longitude--;
op.latitude--;
return op;
}
```

```
int main()
{
loc ob1(10, 20), ob2;
ob1.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob2.show(); // displays 12 22
--ob2;
ob2.show(); // displays 11 21
return 0;
}
```


Binary Operator Overloading

- Complex numbers consists of two parts: real part and imaginary part. It is represented as (x+iy), where x is the real part and y is the imaginary part. The process of performing the addition operation is illustrated below. Let c1, c2and c3 be three complex numbers represented as follows:

  c1 = x1 + i y1;
  c2 = x2 + i y2;

The operation c3 = c1 + c2 is given by

  c3 = (c1 .x1 + c2 .x2) + i (c1. y1 + c2. y2) ;

```
// Complex1.cpp: Complex numbers operations with binary operator overloading
# include < iostream.h>
class complex
{
  private :
      float real ;          // real part of complex number
      float imag ;          // imaginary part of complex number
 public :
     complex ( )                  // no argument constructor
     {
        real = imag = 0 . 0 ;
     }
```


Binary Operator Overloading

```
 void  getdata ( )                // read complex number
      {      cout <<  "Real  Part  ?  " ;
             cin  >>  real ;
             cout <<  "Imag  Part  ?  " ;
             cin  >>  imag ;
       }
     complex  operator  +  (  complex  c2  ) ;    //  complex  addition
      void outdata (  char  *msg )              //  display  complex  number
     {     cout  <<  endl  <<  msg ;
           cout  <<  " ( "  <<   real  ;
           cout  <<   " ,  "  <<  imag<<" ) "  ;
      }
} ;
 //  add default and c2 complex objects
complex complex : : operator  + (  complex  c2  )
{  complex  temp  ;                          // object temp of complex class
   temp . real  =  real + c2 . real ;        //add real parts
   temp . imag  = imag = c2 . imag ;         //add imaginary  parts
   return ( temp ) ;                         //  return complex object
}
```


Binary Operator Overloading

```
void main ( )
{
     complex c1, c2 , c3 ;          // c1 , c2 , c3 are object of complex class
     cout  <<  " Enter Complex Number c1  . . "  <<  endl ;
     cl . getdata ( ) ;
     cout  <<  " Enter Complex Number c2  . . "  <<  endl ;
     c2 . getdata ( ) ;
     c3 = c1 + c2 ;  // add c1 and c2 and assign the result to c3  i.e;  c3 = c1 . operator+ ( c2 ) ;
     c3 . outdata ( " c3 = c1 + c2 :  "  ) ;  // display result
}
```

RUN

```
Enter Complex Number  c1 . .
Real Part  ?  2.5
Imag Part  ?  2.0
Enter Complex Number  c2 . .
Real Part  ?  3.0
Imag Part  ?  1.5
C3  =  c1 + c2 :  ( 5 . 5 ,  3.5 )
```


Operator Returns Values

The operator function in the previous program Program1.cpp has a subtle defect. An attempt to use an expression such as;

idx1= idx2++;

will lead to a compilation error like *Improper Assignment* because the return type of operator++ is defined as void type. Such an assignment operation can be permitted after modifying the return type of the operator ++() member function of the index class.

```
//Program2.cpp:  Index class with overloaded operator returning an object
#include  < iostream.h>
class Index
{
   private:
       int  value;                    //  Index  Value
 public:
       Index ( )                      //  No argument constructor
      {  value  =  0;
       }
     int GetIndex( )            // Index Access
       { return value;
        }
```


Operator Returns Values

```
Index operator + +( )      // Returns nameless object of class Index
        {    Index temp;                  // temp object
             value = value + 1;          // update index value
              temp.value = value;        //  Initialise temp object
             return temp;                //  Returns temp object
         }
} ;
 void main( )
{
     Index idx1, idx2;              // idex1 and idx2 are objects of  Index class
   // display index values
   cout << " \nIndex1 = " << idx1 . GetIndex ( ) ;
   cout << " \nIndex2 = " << idx2 . GetIndex ( ) ;
   // Returned object of  idx2+ +  is assigned to object idx1
  idx1 = idx2++; // invokes the overloaded function and assigned the return value to the object  idx1 of the
     index
  idx2 + + ;   // Returned object of idx2++ is unused
  cout  <<  " \ nIndex1 = " << idx1. GetIndex ( ) ;
 cout  <<  " \ nIndex2 = " << idx2. GetIndex ( ) ;
}
```


Operator Returns Values

RUN

Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2

Nameless Temporary Objects

In Program2.cpp, an intermediate (a temporary) object temp is created as a return object. A convenient way to return an object is to create a nameless object in the return statement itself. Hence, this program illustrates the overloaded operator function returning a nameless object.

```
//Program3.cpp: Index class with overloaded operator returning nameless object
#include  < iostream.h>
class Index
{
    private:
      int  value;                  //  Index  Value

    public:
      Index ( )                     //  No argument constructor
     {
          value  =  0;
      }

     int GetIndex( )            // Index Access
      {
        return value;
      }
```


Nameless Temporary Objects

```
 Index operator + +( )      // Returns nameless object of class Index
        {
            value = value + 1;         // update index value
            return Index(value);    //  Calls one-argument constructor
        }
} ;
 void main( )
{
     Index idx1, idx2;              // idex1 and idx2 are objects of  Index class
    // display index values
   cout << " \nIndex1 = " << idx1 . GetIndex ( ) ;
   cout << " \nIndex2 = " << idx2 . GetIndex ( ) ;
   // Returned object of  idx2+ +  is assigned to idx1
  idx1 = idx2++; // invokes the overloaded function and assigned the return value to the object idx1 of the
     index
  idx2 + + ;   // Returned object of idx2++ is unused
 cout  <<  " \ nIndex1 = " << idx1. GetIndex ( ) ;
 cout  <<  " \ nIndex2 = " << idx2. GetIndex ( ) ;
}
```


Nameless Temporary Objects

RUN

```
Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2
```


Need For Virtual Functions

When objects of different class in a class hierarchy, react to the same message in their own unique ways, they are said to exhibit polymorphic behaviour. This program has the base class Father and the derived class Son and has a member function show() with the same name and prototype. In C++, a pointer to the base class can be used to point to its derived class objects.

```
//Parent1.cpp: Invoking DC members through BC pointer
#include<iostream.h>
#include<string.h>
 class Father   // Father's name
{ char name[20];
  public:
    Father(char *fname)
     {strcpy(name, fname);  //fname contains Father's name
     }
    void show() //show in base class
     {cout<<"Father's name:" << name << endl;
     }
};
class son: public Father
{ char name[20];  // Son's name
  public:
// 2 Argument constructor; invokes 1 argument constructor of Father
```


Postfix Increment and Decrement Operators: ++ and –

- C++ provides prefix and postfix increment and decrement operators; this section describes only the postfix increment and decrement operators.
- The difference between the two is that in the postfix notation, the operator appears after *postfix-expression*, whereas in the prefix notation, the operator appears before *expression.*

i++;

- The effect of applying the postfix increment operator (**++**) is that the operand's value is increased by one unit of the appropriate type. Similarly, the effect of applying the postfix decrement operator (**––**) is that the operand's value is decreased by one unit of the appropriate type.

- It is important to note that a postfix increment or decrement expression evaluates to the value of the expression **prior to** application of the respective operator. The increment or decrement operation occurs **after** the operand is evaluated.

expre_Postfix_Increment_and_Decrement_Operators.cpp

```
#include <iostream>
using namespace std;
 int main() {
 int i = 10;
cout << i++ << endl;
cout << i << endl;
}
```


In the postfix form, the increment or decrement takes place after the value is used in expression evaluation, so the value of the expression is the same as the value of the operand. For example, the following program prints "++i = 6":

```
#include <iostream>
 using namespace std;
int main() {
int i = 5;
cout << "++i = " << ++i << endl;
 }
```


Conclusion

- Constructors are normally used to initialise variables and to allocate memory.
- Similar to normal functions, constructors can be overloaded.
- When an object is created and initialized at the same time, a copy constructor gets called.
- C++ also provides destructor that destroys the objects when they are no longer required.
- Allocation of memory objects to objects at the times of their construction is known as dynamic construction of objects with the help of new operator and can be deleted through delete operator.

Summary

- A class is a data type defined by the user where class describes the data and its behavior or functionality. It serves as a template to create objects.
- The objects can be passed to as well as returned from functions. Inline, constant, nested, friend and static functions add new dimension to the flexibility of C++ program.
- A constructor is a member function with the same name as that of its class and is used to initialise the objects of that class with a legal initial value.

Summary

- A constructor may be called explicitly as well as implicitly. It has various types depending on the basic demand of the program.
- Objects can be initialized dynamically also with new operator. Make sure the memory allocated through new must be properly deleted through delete. Improper use of new and delete may lead to memory leak.
- A destructor deinitializes an object before it goes out of scope and follows the same access rules of class members.

Review Questions [Objective Types]

1. How can we call member functions from outside the class?
2. Point out the reasons why using new is a better idea than malloc()?
3. What is the difference between the following two statements if a is pointer to an array allocated dynamically?

   delete a;

   delete [ ]a;
4. How can we initialize a const data members?

Review Questions [Objective Types]

5. What is an anonymous class?
6. What is dangling pointer? Give example.
7. Is it necessary to accept a reference in the copy constructor?
8. Can a non-static member function access the static data?
9. Can we use the renew operator in C++ to reallocate memory?
10. Like other operators, can new operator also be overloaded?

Review Questions [Short Answer Types]

1. Design a class from which we can create objects by passing one, two or three arguments. The class should not have more than constructor function.
2. How can we initialize an array of objects?
3. Can we initiate an object s of class sample through a statement like sample s = 10; ?
4. Write a snippet to show memory leak in C++?
5. What is the size of an object of an empty class? And Why?
6. Does the delete operator call the destructor of the class?

Review Questions [Short Answer Types]

7. How can one return an error value from the constructor?
8. The this pointer always contain the address of the object using which the member function is being called. Illustrate the concept with the help of an object.
9. Can we modify the this pointer?
10. Discus the various situations when a copy constructor is automatically invoked.

Review Questions [Long Answer Types]

1. Write a program using a class to store marks list of 15 students and to print the highest marks as well as the average of all marks.
2. Illustrate the scope rules of global class, global object, local class and local object through some code fragment in C++.
3. How nesting of member functions work in a class? Discuss.
4. When declaration of static member and static functions become necessary ? Give example.

Review Questions [Long Answer Types]

5. What should we keep in mind before the objects of an inner class can be declared? Show the validity of your remark in the perspective of a Nested class program.
6. How the working of inline function is different from other functions? Is it also different from #define value replacement directive or macro concept?
7. Define a situation that requires a function to operate on objects of two different classes. Write a program to signify the friend function as bridge.

Review Questions [Long Answer Types]

8. Write a program to maintain a bank account class where you can pass objects as parameters to functions.
9. Write the syntax for creating nameless objects and how they will be destroyed in a program?
10. Create a vector class where array can be dynamically allocated?

Recommended Books

TEXT:

1. A..R.Venugopal, Rajkumar, T. Ravishanker "Mastering C++", TMH, 2009.
2. S. B. Lippman & J. Lajoie, "C++ Primer", 6th Edition, Addison Wesley, 2006.

REFERENCE:

1. R. Lafore, "Object Oriented Programming using C++", Galgotia Publications, 2008.
2. D . Parasons, "Object Oriented Programming with C++", BPB Publication.
3. Steven C. Lawlor, "The Art of Programming Computer Science with C++", Vikas Publication.
4. Schildt Herbert, "C++: The Complete Reference", 7th Ed., Tata McGraw Hill, 2008.

