v. In pointers, dereference operator * is used to:
   a. Address the value of the pointer variable.
   b. Points to the value stored in the variable pointed by the pointer variable
   c. Both a and b
   d. None of the above

Q.3 Write TRUE/FALSE against the following statements.

i. The statement int *ptr; defines an ordinary C++ variable.
ii. The following lines of codes assign the value of the variable Temp to the pointer variable PTemp.

```
float Temp;
float *PTemp = &Temp;
```

iii. Pointer variables can be initialized by addresses of the variables to which they refer.
iv. To assign addresses to pointers, a programmer needs a special type of operator called address-of-operator &.
v. When a variable is declared in C++, the compiler reserves location for this variable in computer memory to store data.

Q.4 Define the term pointer. Describe the advantages of using pointer variables.

Q.5 What is the difference between the dereference operator * and reference operator &? Explain with the help of some lines of code.

Q.6 What is meant by the term pointer initialization? Write a simple program to illustrate this concept.

Q.7 How the declaration of a pointer variable is different from the declaration of a variable.

UNIT 8

OBJECTS AND CLASSES

After the completion of Unit-8, Students will be able to:

- Define class and object
- Know the members of class:
  - Data
  - Functions
- iii. Understand an access specifiers:
  - Private
  - Public
- Know the concept of data hiding
- Define constructor and destructor
  - Default constructor/destructor
  - Users-defined constructor
  - Constructors overloading
- Declare objects to access
  - Data members
  - Member functions
- Understand the concept of the following with daily life examples:
  - Inheritance
  - Polymorphism

INTRODUCTION

In structural or traditional programming, the data and actions are considered as two separate entities which cannot provide a very intuitive representation of real world objects and phenomenon. To eliminate this limitation, Object-Oriented Programming (OOP) concept was introduced which combines data and actions together in one package called class. This allows programmers to write programs in more modular fashion to make them easier to understand. This modularization also provides a higher degree of code-reusability. Object-oriented programming also brings several other useful concepts into the field of programming which are: data hiding, inheritance, encapsulation, abstraction, and polymorphism. This Unit is focused on these features of OOP.

8.1 Classes

Class is one of the main features of Object Oriented Programming language. It combines both data and functions into units or packages and makes the programs modularized.

8.1.1 Class and object

a. Class

A **class** consists of both *attributes* of real world objects and *functions* that perform operations on these attributes. The attributes are called **data members** and the operations are called **member functions**. It is an important feature of object oriented programming (OOP) and is used to hold both data and functions, of real world objects, tightly into a single package.

Classes are generally declared using the keyword **class**, with the following format:

```
class class_name
{
access_specifier_1:
member1;
```



```
access_specifier_2: // Body of the class
member2;
...
};
```

Here, **class_name** is a valid identifier for the class which is followed by the **body of the class** that is enclosed in pair of braces. The body of the class consists of:

- Data members
- Member functions

Data members and member functions are used with the *access specifiers*. An access specifier is one of the three types:

- private
- public
- protected

A C++ class ends with a semicolon (;). Consider the following example to understand the concept of **class**.

```
// Class declaration example
class CRectangle
{
private:
int x, y; // declaration of data members
public:
void set_values (int a, int b); // declaration of member function
int area (); // declaration of member function
};
```

The above statements declare a class named **CRectangle**. This class contains four members:

Two data members x and y of type **int** with **private** access specifier and two member functions with **public** access; **set_values ()** and **area ()**. In this example, only declarations for the member functions has given and not their definitions.

b. Object

A variable of type class is called **object**. In C++, when we declare a variable of type class, we call it *instantiating* the class and the variable itself is called an **instance** or **object** of that class. **Object** is of great importance in OOP, because, a class cannot be used without creating its **object**. The general syntax for creating an object of a class is given below.

Class_name Object_name;

Thus, for the above class **CRectangle**, the object can be created as follows:

CRectangle R1;

We can also create more than one objects in a single statement like:

CRectangle R1, R2, R3;

Here, **R1, R2, R3** are the objects of the same class **CRectangle** that share the same data member and member functions. The definition of a class does not occupy any memory. It only defines what the class looks like. In order to use a class, a variable of that class type must be declared. When an object is created then memory is set aside for all the data members and member functions of that class.

The following program demonstrates the concept of **class** in C++ and creation of object.

```
#include <iostream.h>
#include <conio.h>
class Test
{
```



```
public:
void print()
{
cout<< "Welcome to the class";
}
}; //End of the class Test
int main()
{
Test t1; // Object creation
t1.print();//call to print() function
getch();
return 0;
}
```

Output of the program

Welcome to the class

Accessing Members of a Class

Data members and member functions of a class can be accessed in the **main** program (i.e. **main** function) by using the object of that class, and dot operator (.) followed by the name of the member i.e. object member.

For example, in the above program, the member function **print()** is accessed by using the following statement:

t1.print ();

Similarly a data member can be accessed as object_name data member with private data member and public functions.

The following program implement the class *CRectangle* discussed above.

```
#include <iostream.h>
#include <conio.h>
class CRectangle
{
private:
int X, Y;
public:
void set_values (int a,int b)
{
X=a;
Y=b;
}
int area()
{
cout<< "area of the rectangle="<<(X*Y);
}
}; //End of class
int main()
{
 CRectangle R1; // Object creation
 R1.set_values(44,22);//call to set_values function
 R1.area(); // call to area function
 getch();
 return 0;
 }
```

Output of the program

Area of the rectangle=968

8.1.2 Members of a class

Class has two members; **data members** and **member functions**.

a. Data member

The attributes of a class are called **data members**. Mostly, data members are declared under the **private** access specifier to make them private to the class in which they are used. Data members can also be used under **public** access specifier.
For ech object of a class, a separate copy of the data members is created in memory.

b. Member function

The functions declared or defined inside the body of the class are called **member functions**. These member functions are usually written under the public access specifier to operate on the data members of the class. Member functions can also be written in the *private* area of the class but the practice is to write them in the *public* area.

```
// Example of a class with its members as public
#include<iostream.h>
#include<conio.h>
class Date
{
public:
int Month;
int Day;
int Year;
void SetDate (int nMonth, int nDay, int nYear)
{
Month = nMonth;
Day = nDay;
Year = nYear;
}
void ShowDate()
```

```
{
cout<<"Month of birth:"<<Month<<endl;
cout<<"Day of birth:"<<Day<<endl;
cout<<"Year of birth:"<<Year;
}
};
int main()
{
Date d1; // Object creation
d1.SetDate(01,01,2012);//call to SetDate function
d1.ShowDate(); // call to ShowDate function
getch();
return 0;
}
```

Output of the program

Month of birth: 01
Day of birth: 01
Year of birth: 2012

8.1.3 Access specifiers

Access specifiers are the determiners that determine which member of a class is accessible in which part of the class/program. The most commonly used access specifiers in C++ are:

- private access specifier
- public access specifier

a. private access specifier

private access specifier tells the compiler that the members defined in a class by preceding this specifier are accessible only within the class and its friend function. ***private*** members are not accessible outside the class.

A program explaining the use of *private* data members within and outside the class is given below.

```
#include<iostream.h>
#include<conio.h>
class Date
{
private:
int Day;
int Month;
int Year;
public:
void SetDate(int nDay, int nMonth, int nYear)
{
Day = nDay;
Month=nMonth;
Year = nYear;
cout<< "Today's date is: "<<Day<< "/"<<Month<< "/"<<Year;
}
};
int main()
{
Date cDate;
//cDate.Day = 10;
//cDate.Month = 01;
//cDate.Year = 2012;
cDate.SetDate(15, 01, 2012); //Valid
getch();
return 0;
}
```

Okay: because private members are accessible within the class

Invalid: because private members cannot be accessible from outside the class

b. public access specifier

***public** members* are accessible from anywhere where the object is visible i.e. within the class, in the derived classes and in the **main** function.

Consider the following program to demonstrate the visibility of *public* members of a class.

```
// public access specifier example 1
#include<iostream.h>
#include<conio.h>
class PublicTest
{
public:
int X=10; // Public data member
void showPublic() // Public member function
{
cout<< "X= "<<X;
}
};
int main()
{
PublicTest PT1; //PT1 is an object of class PublicTest
PT1.X; //access private data members
PT1.showPublic(); // access public member function
getch();
return 0;
}
```

In the above program, the class *PublicTest* is defined which comprise of one data member X and one member function *showPublic()* in its *public* part. The data member

is accessed within the member function *showPublic()* and outside the class, i.e. in *main()* function, without generating any error message. Similarly, the member function *showPublic()* is also accessed in *main()* function.

If no access specifier is mentioned in a class for a member then it will combined as private by default.

The following program demonstrates private and public access specifier.

```
#include<iostream.h>
#include<conio.h>
class Access
{
int A; // private by default
void GetA() // private by default
{
cout<<"I am private member accessible only through public member function"<<endl;
}
public:
int B; // public
void GetB()
{
GetA();
cout<<"I am B in public"<<endl;
}
};
int main()
{
Access cAccess; //cAccess is an object of the class Access
```

```
//cAccess.A = 2; // WRONG because A is private data member
//cAccess.GetA(); //WRONG because GetA() is private member function
cAccess.B=10; // Okay because B is public data member
cAccess.GetB(); // Okay because GetB() is public member function
getch();
return 0;
}
```

The above program explains the scope (or accessibility) of the *private* and *public* data members and member functions in detail. Also, comments are added with each line of code in *main()* program for the explanation of its accessibility level.

The output of the program is:

```
I am private member accessible only through public member function
I am B in public
```

8.1.4 Data hiding

Data Hiding is an important concept of C++ programming language in which the members of a class are protected against illegal access from outside the class. In C language, **struct** is used but it cannot hide data of its data members. Whereas in C++, we got the facility of hiding data using different access levels: private, protected, public in classes.

Private data members and member functions can only be accessed by the members of the same class defining them and cannot be accessed from outside the class. Similarly, **protected** members (data members and member functions) of a class can be accessed only by the class defining them and its derived classes. By using *friend function*, the *private* and *protected* members (data members and member functions) of a class can also be accessed. Except these, *private* and *protected* members are not accessible

anywhere else and thus provide the facility of data hiding. The **public** access specifier has global access and can be accessed within the same class, derived classes and *main()* program.

The following table shows the level of hiding members of a class.

Access	Public	Protected	Private
Access of members (data, functions) in the same class	Yes	Yes	Yes
Access of members (data, functions) in the derived classes	Yes	Yes	No
Access of members (data, functions) from outside the class i.e. from main()	Yes	No	No

Table 8.1: Access Specifiers and Data Hiding

The concept of data hiding, shown in table 8.1, will be demonstrated after the introduction of inheritance.

8.1.5 Constructor and Destructor

Constructors and destructors are special member functions within the class with the same name as that of the class. The following sections discuss them in detail with the help of examples.

a. Constructor

A **constructor** is a special kind of member function that is executed automatically when an object of the class is instantiated. Constructors are typically used to initialize data members of the class to appropriate default values, or to allow the user to easily initialize those member variables to whatever values are desired.

constructors have specific rules for naming:

- Constructors have the same name as that of the class
- Constructors have no return type (not even void)

Consider the following simple program to explain the concept of constructor in a class.

```
#include <iostream.h>
#include <conio.h>
class ConstTest
{
public:
ConstTest () // Constructor
{
cout<< "I am Constructor";
}
};
int main()
{
ConstTest CT; // CT is an object to the class ConstTest
getch();
return 0;
}
```

Output of the program

I am Constructor

In the above example, a class *ConstTest* is defined which have the constructor *ConstTest()* in its *public* part. In *main()* program, there is no explicit call to this constructor and it is automatically called when the object CT of this class is created. It should be noted that *Constructors* are always called at the moment when objects are created. They cannot be called explicitly as member functions are called.

Consider another program to implement the class **CRectangle** including a constructor:

```
#include <iostream.h>
#include <conio.h>
class CRectangle
{
int width, height;
public:
CRectangle (int a, int b) // constructor
{
width = a;
height = b;
}
int area()
{
return (width*height);
}
};
int main()
{
CRectangle r1 (13,14);
CRectangle r2 (15,16);
cout << "Area of Rectangle 1: " << r1.area() << endl;
cout << " Area of Rectangle 2: " << r2.area() << endl;
getch();
return 0;
}
```

Output of the program

Area of Rectangle 1: 182

Area of Rectangle 2: 240

In the above example, the constructor *CRectangle* (*int* a, *int* b) is defined in the class *CRectangle* that initializes the values of width and height with the parameters that are passed to it when objects, r1 and r2, are created with the statement *CRectangle r1 (13,14)* and *CRectangle r1 (13,14)*.

i. Default constructor

If we do not declare any constructor in a class definition, the compiler assumes that a class has a default constructor with no arguments.

```
// default constructor
class DConstructor
{
public:
int a,b,c;
void multiply (int n, int m)
{
a=n;
b=m;
c=a*b;
}
};
```

The compiler assumes that the class *DConstructor* has a default constructor, so the object of this class can be declared by specifying no arguments as given below:

DConstructor DC;

Here, DC is an object to the class *DConstructor* with no arguments. Default constructors are also called *implicit* constructors.

ii. User-defined constructor/ Explicit constructor

When users define constructors in class for their own purpose, especially for initialization of variables, then such types of constructors are called **user defined constructors.** When constructor is declared for a class, the compiler no longer provides an *implicit default* constructor. So, the objects should be declared according to the constructor prototypes that have been defined for the class.

```
// user-defined ( parameterized) constructor example
#include <iostream.h>
#include <conio.h>
class CExample
{
public:
int a,b,c;
CExample (int n, int m)
{
a=n;
b=m;
}
int multiply ()
{
c=a*b;
return c;
}
};
int main()
{
CExample CObj (50,10);
int result=CObj.multiply();
```

```
cout<<"Multiplication Results is: "<<result;
getch();
return 0;
}
```

Output of the program

Multiplication Result is: 500

Here, a constructor *CExample* with two parameters of type *int* has been declared that initializes *private* variables a and b. When the object declaration is done as follows:

CExample CObj (5,10);

The following declaration is not correct, because, we have declared the class to have an *explicit constructor (user-defined constructor)* to which the values should be passed.

CExample CObj;

A user-defined constructor can be either parameterized or non-parameterized. A *parameterized constructor* is the one which accepts arguments while the object is declared and *nonparameterized constructor* is the one whose object declaration is done without specifying arguments. The program given above defines *parameterized constructor*. The following program explains *non-parameterized* constructor.

```
#include <iostream.h>
#include <conio.h>
class CNonParameter
{
public:
CNonParameter ()
{
cout<< "I am non-parameterized constructor";
}
```



```
};
int main()
{
CNonParameter NP;
getch();
return 0;
}
```

Output of the program

I am non-parameterized constructor

In this example, the constructor *CNonParameter*() has no parameters and thus the object creation statement, *CNonParameter NP;*, passes no arguments.

iii. Constructor overloading

Using more than one constructors in the same class is called **constructor overloading.** Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. For an overloaded function, the compiler calls the function whose parameters match the arguments used in the function call. In the case of constructor overloading, that constructor is executed whose parameters match the arguments passed to the object declaration. Consider the following program.

```
// constructors overloading
#include <iostream.h>
#include <conio.h>
class CRectangle
{
int width, length;
```

```
public:
CRectangle()
{
width = 5;
length = 15;
}
CRectangle (int a, int b)
{
width = a;
length = b;
}
int area()
{
return (width*length);
}
};
int main()
{
CRectangle r1 (5,14);
CRectangle r2;
cout << "Area of first rectangle: " << r1.area() << endl;
cout << "Area of second rectangle: " << r2.area() << endl;
getch();
return 0;
}
```

Output of the program

```
Area of first rectangle: 70
Area of second rectangle: 75
```


In the above example, the first object r1 passes the arguments (5,14) to the second parameterized constructor of the class and initializes the variables width with 5 and length with 15 and thus the result calculated is 75. constructor and thus the output generated is 70. The second object r2 calls the non-

b. Destructors

Destructors are special member functions that are executed when an object is destroyed. Destructor fulfills the opposite functionality of constructor and thus de-allocates the memory allocated to an object during its creation. They are called automatically when objects are destroyed.

Like constructors, destructors have the following specific features:

- Destructor has the same name as that of the class, preceded by a tilde symbol ~.
- Destructor cannot take arguments.
- Destructor has no return type.

The second feature given above, implies that only one destructor may exist per class, as there is no way to overload destructors since they cannot be differentiated from each other based on arguments.

Consider the following program to explain the concept of both constructor and destructor.

```
#include <iostream.h>
#include <conio.h>
class A
{
public:
A () //constructor
{
cout<<"I am constructor"<<endl;
```

```
}
~A () // destructor
{
cout<<"I am destructor";
}
};
int main()
{
A a1;
getch();
return 0;
}
```

Output of the program

```
I am constructor
I am destructor
```

In this example, ~A () is a destructor. When the program runs and the object a1 is created, constructor A() is called displaying the message "I am constructor". But, when the control goes out of the program and the object is destroyed, the destructor ~A () is called to de-allocate the memory reserved by the object of the class and a message, "I am destructor", is displayed to verify that the destructor is executed.

8.1. 6 Inheritance and Polymorphism

Object Oriented Programming (OOP) languages have the features of code reusability and polymorphism. Code reusability is the key feature among all other features of OOP which can be achieved by the use of inheritance. Similarly, C++ has the ability to use function name and operator for multiple tasks.

a. Inheritance

A key feature of C++ classes in which new classes are created from existing classes is called inheritance. Inheritance uses the concept of parent and child class. A **Parent Class** is the class from which others classes are derived. It is also called **base class**. A **Sub Class** is a class which inherits features from the base class. A sub-class is also called **child class** or **derived class**. A few examples of inheritance from daily life are given below:

- Consider an apple and a banana. Although an apple and banana are different fruits, both have common feature that they are fruits. Because, apples and bananas are fruits, anything that is true for fruits is also true for apples and bananas. For example, all fruits have a name, a flavor, and are tangible objects. Thus, apples and bananas also have a name, a flavor, and are tangible objects. Apples and bananas inherit these properties from the concept of fruit because they are fruit. Apples and bananas then define some of the properties in different ways as bananas shape is different from apple, also apples have seeds but bananas have no seeds, which make them different from each other.

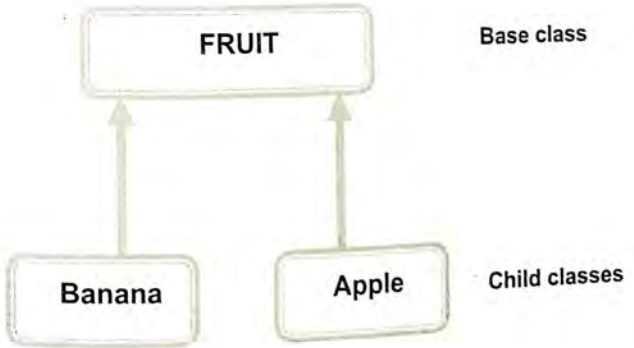

Figure 8.2: Inheritance of Fruit Class

- Here is another example of shape class which has a series of child classes that describe shapes Rectangle and Triangle. They have certain common properties, such as both can be described by means of their sides but they have their own characteristics i.e. triangle has three sides and rectangle has four sides. Square is more special case with four sides of equal length.

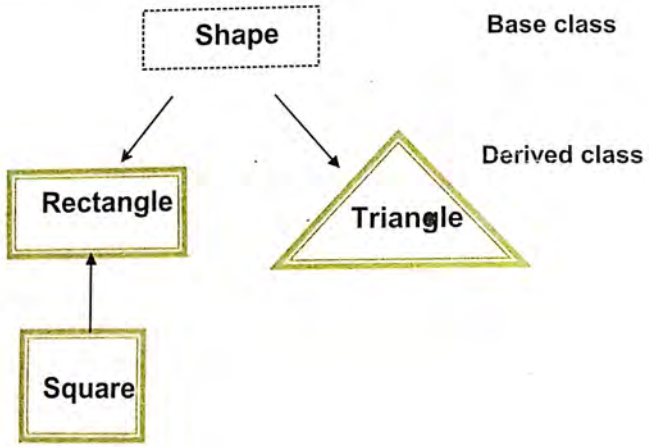

Figure 8.2: Inheritance of Shape Class

- We can also represent patients into indoor and outdoor patients in which both have some common features like name, father name, age, address and disease but indoor patient have 'Ward No' and 'Bed No' as its unique features and the outdoor patient have 'Next date of visit' as its unique feature. It can be represented as follows:

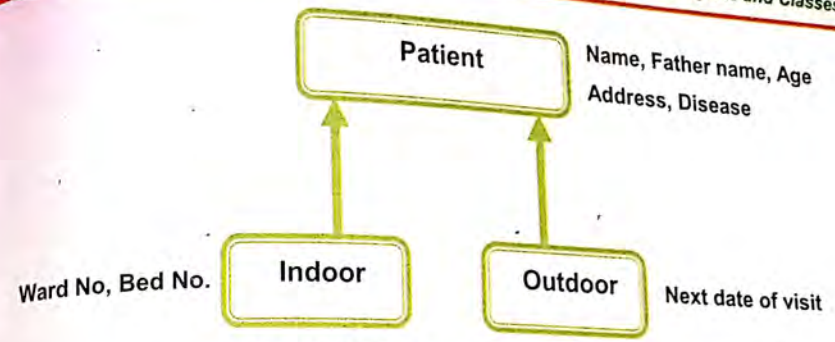

Figure 8.3: Inheritance of Patient Class

In order to derive the child class *Indoor* from the base class *patient*, the following statement is used.

```
class Indoor: public Patient
{
};
```

The **public** access specifier is used to derive a class from its parent class. The keyword **public** may be replaced by any one of the other access specifiers **protected** and **private**.

The following lines of code give the general syntax of using inheritance.

```
class A // base class
{
};
class B: public A //class B is derived from class A
{
};
```

A complete C++ program for the above sketch is given below.

```
#include <iostream.h>
#include <conio.h>
```

```
class A
{
public:
void showa()
{
cout<<"I am in base class A"<<endl;
}
};
class B: public A
{
public:
void showb()
{
cout<<"I am in derived class B"<<endl;
}
};
int main()
{
B b1; // object of the derived class
b1.showa(); // object of B calling function of base class A
b1.showb(); // object of B calling own functiom
getch();
return 0;
}
```

Output of the program

I am in base class A
I am in derived class B

In this example, we have two classes **A** and **B**. Class B is *publically* derived from class A and has therefore right to access the member function of class A. In **main** (), we have created only object **b1** for class B and have called the member functions **showa**() of base class A and **showb**() of derived class B that have produced the output as shown above.

In inheritance, sometimes a class is derived from more than one parent classes and the phenomenon is called **multiple inheritance**. In this case the derived class has properties of base classes as well as its own.

```
class A
{ };
class B
{ };
class c: public A, public B
{ };
```

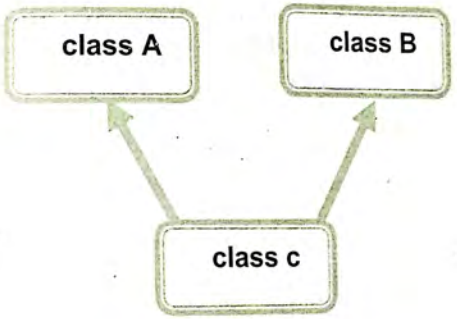

Figure 8.4: Multiple Inheritance

b. Polymorphism

Polymorphism is the ability to use an operator or function in multiple ways. Polymorphism gives different meanings or functionality to the operators or functions. Poly, refers to many, signifies that there are many uses of these operators and functions.

In C++, polymorphism can be achieved by one of the following concepts.

- Function overloading
- Operator overloading
- Virtual functions

Function overloading is the concept of using same function name for related but slightly different purposes in a single program. We have discussed it in detail in unit 6 with the help of programming examples.

Consider the following simple examples to understand the concept of polymorphism with respect to operator overloading.

6 + 10;

The above statement refers to integer addition with the help of + operator. The same + operator can also be used for addition of two floating point numbers or two strings (concatenation) as shown below.

7.15 + 3.78
"Computer" + "Training"

Polymorphism is a powerful feature of every Object Oriented Programming (OOP) language, especially of C++. A single operator '+' behaves differently in different contexts such as *integer* and *float* addition and *strings* concatenation. This concept is known as **operator overloading.**

A **virtual function** or virtual method is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature. This concept is a very important part of the polymorphism portion of object-oriented programming (OOP).

Summary

- A class consists of attributes called data members and function called member function.
- A variable of type class is called object. In C++, when variables of type class are declared they create objects.
- The attributes of a real world objects are called data members.
- The functions declared or defined inside the body of the class are called member functions.
- Access specifiers determine that which member of a class is accessible by which member of the class/program.
- private access specifier tells the compiler that the members defined in a class by preceding this specifier are accessible only within the class and its friend function.
- public members are accessible from anywhere where the object is visible.
- Data Hiding is the concept in which the members of a class are protected against illegal access from outside the class.
- A constructor is a special kind of class member function that is executed when an object of the class is instantiated.
- Destructors are special member functions having the same name as that of the class with a tiled symbol ~ and executed when an object is destroyed.
- To access members of a class, dot operator (.) is used with the member name.
- The phenomenon of creating new classes from existing classes is called inheritance.
- Polymorphism is the ability to use an operator or function in multiple ways.

Exercise

Q.1 Fill in the Blanks.

i. _____ is a member of function that is called automatically when an object is created..

ii. The ability to reuse code already defined for the new purpose, is referred to as_____.

iii. In OOP the main advantage of inheritance include_____.

iv. An object of a class can be defined as " a variable of type _____"

v. Function overloading is used to achieve _____.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. A constructor is called whenever_____.
   a. An object is destroyed
   b. An object is created
   c. A class is declared
   d. A class is used

ii. Destructor is used for _____.
   a. Initializing the values of data members in an object
   b. Initializing arrays
   c. Freeing memory allocated to the object of the class when it was created
   d. Creating an object

iii. The name of destructor is always preceded by the symbol _____
   a. +
   b. %
   c. -
   d. ~

iv. Constructors are usually used for _____.
   a. Constructing programs
   b. Running classes
   c. Initializing objects
   d. All of the above

v. Inheritance is used to_____.
   a. Increase the size of a program
   b. Make the program simpler
   c. Provide the facility of code reusability
   d. Provide the facility of data hiding

Q.3 Write TRUE/FALSE against the following statements.

i. C++ is a pure object-oriented programming language.

ii. One of the main features of object oriented programming language is data hiding.

iii. If you do not declare a constructor in a class, the compiler will furnish a default constructor for you automatically.

iv. A constructor is a function that is called when an instance of a class is deleted.

v. Function overloading is an example of polymorphism.

Q.4 Write a C++ program implementing a class with the name **Circle** having two functions with the names: **GetRadius** and **CalArea.** The functions should get the value for the radius from the user and then calculate the area of the circle and display the results.

Q.5 Write a C++ program implementing inheritance between **Employee** (base class) and **Manager** (derived class).