2067

Section A $(2 \times 10 = 20)$

1. Discuss the feature of Object-Oriented Programming? Differentiate between Object Oriented Programming and Procedural Based Programming.

Different features of object oriented programming are explained here below:

Object:

Objects are the entities in an object oriented system through which we perceive the world around us. We naturally see our environment as being composed of things which have recognizable identities & behavior. The entities are then represented as objects in the program. They may represent a person, a place, a bank account, or any item that the program must handle.

Class:

Object consists of data and function tied together in a single unit. Functions are used to manipulate on the data. The entire construct of objects can be represented by a user defined data type in programming. The class is the user defined data type used to declare the objects. Actually objects are the variable of the user defined data type implemented as class. Once a class is defined, we can create any number of objects of its type. Each object that is created from the user defined type implemented as class is associated with the data type of that class.

Abstraction:

Abstraction is representing essential features of an object without including the background details or explanation. It focuses the outside view of an object, separating its essential behavior from its implementation. We can manage complexity through abstraction. Let's take an example of vehicle. It is constructed from thousands of parts. The abstraction allows the driver of the vehicle to drive without having detail knowledge of the complexity of the parts. The driver can drive the whole vehicle treating like a single object.

Encapsulation:

The mechanism of wrapping up of data and function into a single unit is called encapsulation. Because of encapsulation data and its manipulating function can be kept together. We can assume encapsulation as a protective wrapper that prevents the data being accessed by other code defined outside the wrapper. By making use of encapsulation we can easily achieve abstraction.

Inheritance:

Inheritance is the process by which objects of one class acquire the characteristics of object of another class. We can use additional features to an existing class without modifying it. This is possible by deriving a new class (derived class) from the existing one (base class).This process of deriving a new class from the existing base class is called inheritance.

Polymorphism:

Polymorphism means 'having many forms'. The polymorphism allows different objects to respond to the same operation in different ways, the response being specific to the type of object. The different ways of using same function or operator depending on what they are operating on is called polymorphism. Example of polymorphism in OOP is operator overloading, function overloading. Still another type of polymorphism exist which is achieved at run time also called dynamic binding.

Procedure oriented programming and Object oriented programming can be differentiated as follows:

Procedure Oriented Programming	**Object Oriented Programming**
Emphasis is given on procedures.	Emphasis is given on data.
Programs are divided into functions.	Programs are divided into objects.
Follow top-down approach of program design.	Follow bottom-up approach of program design.
Generally data cannot be hidden.	Data can be hidden, so that non-member function cannot access them.
It does not model the real world problem perfectly	It models the real world problem very well.
Data move from function to function	Data and function are tied together. Only related function can access them.
Maintaining and enhancing code is still difficult.	Maintaining and enhancing code is easy.
Code reusability is still difficult	Code reusability is easy in compare to procedure oriented approach.
Examples: FORTRAN, COBOL, Pascal, C	Examples: C++, JAVA, Smalltalk

2. What is constructor? Explain their types. Discuss user defined parameterized constructor with suitable example.

A constructor is a special member function that is executed automatically whenever an object is created. It is used for automatic initialization. Automatic initialization is the process of initializing object's data members when it is first created, without making a separate call to a member function. The name of the constructor is same as the class name. Some special characteristics of constructors are:

- Constructors should be defined or declared in the public session.
- They do not have return types.
- They cannot be inherited but a derived class can call the base class constructor.

- Like functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.
- An object with a constructor cannot be used as a member of a union.
- They make ‘implicit calls’ to the new and delete operators when a memory allocation is required.

Types of constructors are: Default Constructor, Parameterized Constructor, and Copy Constructor. A constructor that does not take any parameter is called default constructor. A constructor that takes arguments is called a parameterized constructor. A copy constructor is called when an object is created by copying an existing object.

We can use parameterized constructors in two ways: by calling the constructor explicitly and by calling the constructor implicitly. Calling the constructor explicitly can also be termed as user defined parameterized constructor. The declaration

Rectangle R1 = Rectangle(5, 6, 7);

illustrates the user defined parameterized constructor.

3. Define a *Clock* class (with necessary constructor and member functions) in OOP (abstract necessary attributes and their types). Write a complete code in C++ programming language.

- **Derive *Wall_Clock* class from *Clock* class adding necessary attributes.**
- **Create two objects of *Wall_Clock* class with all initial state to 0 or NULL.**

```
#include<iostream.h>

#include<conio.h>

class clock

{       protected:

        char model_no[10];

        float price;

        char manufacturer[50];

        public:

        void getclockdata()

        {       cout<<"Enter clock manufacturer:"<<endl;

                cin>>manufacturer;
                cout<<"Enter model number:"<<endl;
                cin>>model_no;
                cout<<"Enter price:"<<endl;
                cin>>price;
        }
        void clockdisplay();
};
class wall_clock: public clock
{       int hr, min, sec;
        public:
        void wall_clock()
        {       model_no=NULL;
                manufacturer=NULL;
                price=0.0;
                hr=0;
                min=0;
                sec=0;
        }
        void getwallclockdata()
        {       cout<<"Enter hour, minute and seconds:"<<endl;
                cin>>hr>>min>>sec;
        }
        void wallclockdisplay()
        {       cout<<"Time="<<hr<<":"<<min<<":"<<sec<<endl;
        }
};
void clock: clockdisplay()
{       cout<<"Model number="<<model_no<<endl;
```

```
        cout<<"Manufacturer="<<manufacturer<<endl;
        cout<<"Price="<<price<<endl;
}
int main()
{       wall_clock W1, W2;
        cout<<"Enter data for W1:"<<endl;
        W1.getclockdata();
        W1.getwallclockdata();
        cout<<"Value of W1:"<<endl;
        W1.clockdisplay();
        W1.wallclockdisplay();
        cout<<"Value of W2:"<<endl;
        W2.clockdisplay();
        W2.wallclockdisplay();
        return 1;
}
```

Section B $(8 \times 5 = 40)$

4. How can you classify objects? Why dynamic object is needed?

The class declaration does not define any objects but only specifies what they will contain. Once a class has been declared, we can create variables (objects) of that type by using the class name (like any other built-in type variables). For example:

```
rectangle r;
```

creates a variable (object) r of type rectangle. We can create any number of objects from the same class. For example:

```
rectangle r1, r2, r3;
```

Objects can also be created when a class is defined by placing their names immediately after the closing brace. For example:

```
class rectangle {
………
………
………
}r1, r2, r3;
```

Dynamic object is needed due to following reasons:

- The Dynamic Object class enables us to define which operations can be performed on dynamic objects and how to perform those operations.
- This class can be useful if you want to create a more convenient protocol for a library.

5. What is operator overloading? Explain their type with suitable examples.

Operator overloading is one of the most exciting features of C++. The term operator overloading refers to giving the normal C++ operators additional meaning so that we can use them with user-defined data types. For example, C++ allows us to add two variables of user-defined types with the same syntax that is applied to the basic type. Operator overloading is done with the help of a special function, called operator function. The general form of an operator function is:

```
return-type operator op(arguments)
{
Function body
}
```

There are two types of operator overloading: Unary operator overloading and Binary operator overloading. Unary operators are those operators that act on a single operand. ++, -- are unary operators. Binary operators are those that work on two operands. Examples are +,-,*, /, % for arithmetic operations, +=,-=,*= and /= for assignment operations and >, <, <=,>=, == and! = for comparison operations.
Overloading a binary operator is similar to overloading unary operator except that a binary operator requires an additional parameter.

6. Why type conversion is necessary in OOP? Explain with example, the type conversion routine.

When two operands of different types are encountered in the same expression, then the either one of the type variable needs to be converted into the other variable or the other needs to be converted to the first one in order to make the same type variable of two operands for the calculation. There are two types of type conversion: Implicit type conversion and Explicit type conversion.

When two operands of different types are encountered in the same expression, the lower type variable is converted to the type of the higher type variable by the compiler automatically. This is called Implicit type conversion. Sometimes, a programmer needs to convert a value from one type to another in a situation where the compiler will not do it automatically. This is called Explicit type conversion. For this C++ permits explicit type conversion of variables or expressions as follows:

```
(type-name) expression        //C notation
type-name (expression)        //C++ notation
```

For example,

```
int a = 10000;
int b = long(a) * 5 / 2;      //correct
int b = a * 5/2;              //incorrect
```

7. What is Inheritance? Explain their types with suitable examples.

Inheritance (or derivation) is the process of creating new classes, called derived classes, from existing classes, called base classes. The derived class inherits all the properties from the base class and can add its own properties as well. The inherited properties may be hidden (if private in the base class) or visible (if public or protected in the base class) in the derived class.

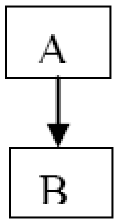
On the basis of this concept, there are five types of inheritance.

Single Inheritance:

In single inheritance, a class is derived from only one base class.

Example

```
class A
{       members of A
};
class B : public A
{       members of B
};
```

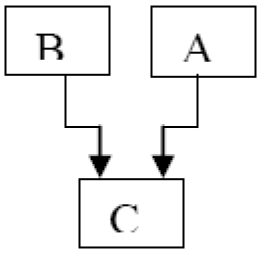


Multiple Inheritance:

In this inheritance, a class is derived from more than one base class.

Example:

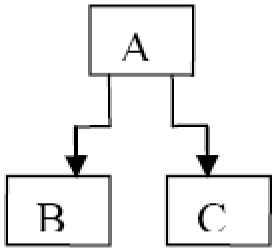
```
class A
{       members of A
};
```



```
class B
{         members of B
};
class C : public A, public B
{         members of C
};
```

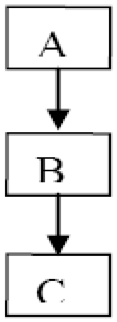
Hierarchical Inheritance:
In this type, two or more classes inherit the properties of one base class.
Example:

```
class A
{         members of A
};
class B
{         members of B
};
class C : public A, public B
{         members of C
};
```



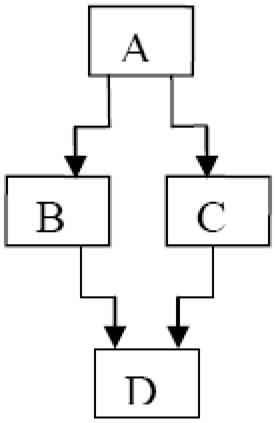
Multilevel Inheritance:
The mechanism of deriving a class from another derived class is known as multilevel inheritance. The process can be extended to an arbitrary number of levels.
Example:

```
class A
{         members of A
};
class B : public A
{         members of B
};
class C : public B
{         members of C
};
```



Hybrid Inheritance:
This type of inheritance includes more than one type of inheritance mentioned previously.
Example

```
class A
{         members of A
};
class B : public A
{         members of B
};
class C : public A
{         members of C
};
```



```
class D : public B, public C
{        members of D
};
```

8. What is friend function? Why it is used in OOP? Explain with an example.

The concepts of data hiding and encapsulation dictate that private members of a class cannot be accessed from outside the class, that is, non-member functions of a class cannot access the non-public members (data members and member functions) of a class. However, we can achieve this by using friend functions. To make an outside function friendly to a class, we simply declare the function as a friend of the class. Furthermore, a fried function also acts as a bridging between two classes. For example, if we want a function to take objects of two classes as arguments and operate on their private members, we can inherit the two classes from the same base class and put the function in the base class. But if the classes are unrelated, there is nothing like a friend function.

Example:

```
class sample

{        int a;

         int b;

         public:

         void setvalue()

         {        a = 25;

                  b = 40;

         }

         friend float mean(sample s);

};

float mean(sample s)

{        return float(s.a + s.b)/2;

}

void main()

{        clrscr();

         sample x;

         x.setvalue();
```

```
        cout<<"Mean value = "<<mean(x);

        getch();

}
```

9. What is container class? Differentiate container class from inheritance.

Inheritance has two types of relationships. One is "kind of" and the other is "has a". The "has a" type of relationship is called containership. We say that a bulldog has a large head, meaning that each bulldog includes an instance of a large head. In object oriented programming, has a relationship occurs when one object is contained in another.

The difference in container class from inheritance are as follows:

- Inheritance is the ability for a class to inherit of properties and the behavior from a parent class by extending it while containership is the ability of a class to contain another objects as member data.
- If a class is extended, it inherits all the public and protected properties and the behaviors and those behavior may be overridden by the sub class but if a class is contained in another, container class does not have the ability to add and modify of contained.

10. Explain the role of virtual function in OOP.

Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Furthermore, when we use virtual functions, different functions can be executed by the same function call. The information regarding which function to invoke is determined at run time. We should use virtual functions and pointers to objects to achieve run time polymorphism. For this, we use functions having same name, same number of parameters, and similar type of parameters in both base and derived classes. The function in the base class is declared as virtual using the keyword virtual. When a function in the base class is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base class pointer, rather than the type of the pointer.

For example:

```
class A
{       public:
        virtual void show()
        {       cout<<"This is class A\n";
        }
};
class B : public A
{       public:
        void show()
        {       cout<<"This is class B\n";
        }
};
class C : public A
```

```
{       public:
        void show()
        {       cout<<"This is class C\n";
        }
};
void main()
{       clrscr();
        A *p, a;
        B b;
        C c;
        p = &b;
        a->show();
        p = &c;
        a->show();
        p = &a;
        a->show();
        getch();
}
```

11. Explain about “this” pointer with suitable example.

The keyword “this” identifies a special type of pointer. Suppose that you create an object named x of class A, and class A has a non-static member function f(). If you call the function x.f(), the keyword “this” in the body of f() stores the address of x. You cannot declare the “this” pointer or make assignments to it. A static member function does not have a “this” pointer.

Example of “this” pointer:

```
#include <iostream>

using namespace std;

struct X

{       private:

        int a;

        public:

        void Set_a(int a)

        {       // The 'this' pointer is used to retrieve 'xobj.a'

                // hidden by the automatic variable 'a'
                this->a = a;
        }
        void Print_a()
        {       cout << "a = " << a << endl;
        }
};
int main()
{       X xobj;
        int a = 5;
        xobj.Set_a(a);
        xobj.Print_a();
}
```

12. WAP to find the square of given integer using inline function.

```
#include<iostream.h>
#include<conio.h>
inline void square(int a)
{       int sq;
        sq=a*a;
        cout<<"Square="<<sq<<endl;
}
void main()
{       clrscr();
        int x;
        cout<<"Enter a number:"<<endl;
        cin>>x;
        square(x);
        getch();
}
```

13. WAP to convert feet into meter.

```
#include<iostream.h>
#include<conio.h>
void main()
{       float feet, meter;
        cout<<"Enter the feet:"<<endl;
        cin>>feet;
        meter=0.3048 *feet;
        cout<<"Equivalent meter is:"<<meter<<endl;
        getch();
}
```