Vidyalankar

S.Y. Diploma : Sem. IV [CO/CD/CM/CW/IF]

Object Oriented Programming

Prelim Question Paper Solution

1. (a) (i) Use of Scope Resolution Operator

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the scope resolution operator. This can be used to uncover a hidden variable. It takes the following form :

:: variable-name

This operator allows access to the global version of a variable. For example, ::count means the global version of the variable count (and not the local variable count declared in that block). Program illustrates this featues.

Scope Resolution Operator

```
# include     <iostream>

using namespace std;
int m = 10;                    // global m

int main ( )
{
          int m = 20;          // m redeclared  , local to main
*
          {
               int k = m;
               int m = 30;          // m declared again
                                    // local to inner block

               cout   <<     "we are in inner block  \n*;
               cout   <<     "k = " << k << "\n";
               cout   <<     "m = " << m << "\n";
               cout   <<     ": : m = "  "<<  ::m << "\n";
          }

          cout   <<     "\nWe are in outer block \n";
          cout   <<     "m = " << m << "\n";
          cout   <<     "::m = " << ::m << "\n" ;

          return 0;
}
```

1. (a) (ii) Two operators used with pointers

C++ provides two pointer operators, which are (a) Address of Operator & and (b) Declaration Operator*.

A pointer is a variable that contains the address of another variable or you can say that a variable that contains the address of another variable is said to "point to" the other variable. A variable can be any data type including an object, structure or again pointer itself.

The .(dot) operator and the → (arrow) operator are used to reference individual members of classes, structures, and unions.

The Address of Operator &
The & is a unary operator that returns the memory address of its operand. For example, if var is an integer variable, then &var is its address. This operator has the same precedence and right-to-left associativity as the other unary operators.

You should read the & operator as "the address of" which means &var will be read as "the address of var".

The Declaration Operator *
The second operator is indirection Operator*, and it is the complement of &. It is a unary operator that returns the value of the variable located at the address specified by its operand.

1. (a) (iii) Program

char name[10] = 10*1 = 10 bytes
int rollno = 2 bytes
float percentage = 4 bytes

Total : 16 bytes

1. (a) (iv) Constructors

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

A constructor is declared and defined as follows :

```
// class with a constructor

class integer
{
        int m, n;
    public;
        integer(void);          // constructor declared
        ….
        ….
};
integer : : integer(void)       // constructor defined
{
        m = 0; n = 0;
}
```

Vidyalankar

1. (a) (v) Polymorphism

Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple form'. We have already seen how the concept of polymorphism is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time an therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding or static binding or static linking. Also known as compile time polymorphism, early binding simply means that an object is bound to its function call at compile time.

1. (a) (vi) Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1) **Public Inheritance**
This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.
```
class Subclass : public Superclass
```

2) **Private Inheritance**
In private mode, the protected and public members of super class become private members of derived class
```
class Subclass : Superclass     //By default its private inheritance
```

3) **Protected Inheritance**
In protected mode, the public and protected members of Super class becomes protected members of Sub class.
```
class subclass : protected Superclass
```

1. (a) (vii) Use of this pointer in C++

this Pointer

C++ uses a unique keyword called this to represent an object that invokes a member function. this is a pointer that points to the object for which this function was called. For example, the function call A.max( ) will set the pointer this to the address of the object A. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions. Consider the following simple example:

```
class ABC
{
    int a;
    ….
    ….
};
```


Vidyalankar

The private variable 'a' can be used directly inside a member function, like
a = 123;

We can also use the following statement to do the name job:
this→a = 123;

Since C++ permits the use of shorthand form a = 123, we have not been using the pointer this explicitly so far. However, we have been implicitly using the pointer this when overloading the operators using member function.

1. (a) (viii) Output

Welcome
Welcome
Welcome

1. (b) (i) Multiple Constructors in a Class

So far we have used two kinds of constructors. They are:

```
integer( );            // No arguments
integer(int, int);     // Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from main( ). C++ permits us to use both these constructors in the same class. For example, we could define a class a follows :

```
class   integer
{
        int m, n;
    public:
        integer( ) {m=0; n=0;}          // constructor 1
        integer(int a, int b)
        {m = a; n = b;}                 // constructor 2
        integer(integer & i)
        {m = i.m; n = i.n;}             // constructor 3
};
```

This declares three constructors for an integer object. The first constructor receives no arguments, the second receive two integer arguments and the third receives one integer object as an argument. For example, the declaration
Integer I1;

would automatically invoke the first constructor and set both m and n of I1 to zero. The statement
integer I2(20,40);

would call the second constructor which will initialize the data members m and n of I2 to 20 and 40 respectively. Finally, the statement
Integer I3(I2):

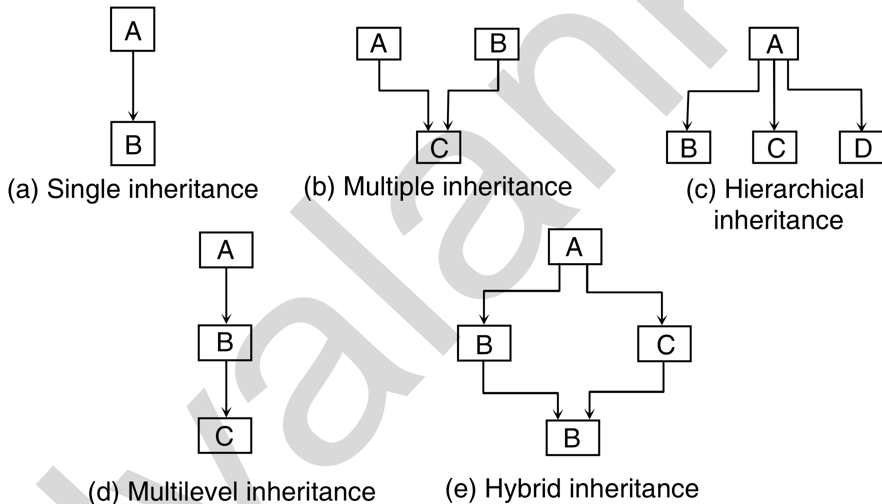
would invoke the third constructor which copies the values of I2 into I3. In other words, it sets the value of every data element of I3 to the value of the corresponding data element of I2. As mentioned earlier, such a constructor is called the copy constructor. The process of sharing the same name by two or

Vidyalankar

more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

1. (b) (ii) Types of Inheritance with example

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance. Figure shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. (Some authors show the arrow in opposite direction meaning "inherited from".)



(a) Single inheritance (b) Multiple inheritance (c) Hierarchical inheritance

(d) Multilevel inheritance (e) Hybrid inheritance

1. (b) (iii) Difference between Constructor and Destructor

Constructor:

1) Constructor is Used to Initialize the Object.
2) Constructor can takes arguments.
3) Constructor overloading can be possible means more than one constructor can be defined in same class.
4) Constructor has same name as class name.
5) Syntax of constructor :

```
class class_name
{
    class_sname( ) { }
    class_name(argulist) { }
};
```

6) Constructors are of following :
   a) Default constructor.
   b) Parameterized constructor
   c) Copy constructor

7) Constructors can be used to dynamically initialize the memory.
8) Constructor indirectly use the new operator to initialize the object.

Destructor

1) Destructor is used to destroy the object that are created in memory previously.
2) Destructor cannot take any arguments.
3) Destructor overloading cannot be possible.
4) Destructor has same name as class name with tiled operator.
5) Syntax of Destructor

```
class class_name
{

    ~class-name(void) { }
};
```

6) Destructor has no any types.
7) Destructor can be used to deallocate the memory.
8) Destructor indirectly use the Delete operator to destroy the object initialize by constructor.

2. (a) Member function of inside and outside class

Member function can be defined in two places :

- Outside the class definition.
- Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined.

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, the ANSI prototype form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which class the function belongs to. The general form of a member function definition is:

```
return-type  class-name  : :  function-name (argument declaration)
    {
            Function body
    }
```

Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

Vidyalankar

```
class item
{
      int number;
      float cost;
public;
      void getdata(int a, float b);          // declaration
            // inline function
      void putdata(void)                     // definition inside the class
      {
            cout   <<   number   <<   "\n";
            cout   <<   cost          <<   "\n";
      }
};
```

2. (b) Arrays within a Class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size = 10;              // provides value for array size

class array
{
            int a[size];          // 'a' is int type array
      public;
            void setval(void);
            void display(void);
};
```

The array variable a[ ] declared as a private member of the class array can be used in the member functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function setval( ) sets the values of elements of the array a[ ], and display( ) function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

2. (c) Arrays of Objects

We know that an array can be of any data type including struct. Similarly, we can also have arrays of variables that are of the type class. Such variables are called arrays of objects. Consider the following class definition:

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

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

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

Vidyalankar

2. (d) Benefits of OOP

OOP offers several benefits to both the program designer and the user. Object orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

2. (e) Friendly Functions

Private members cannot be accessed from outside the class. That is a non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes, manager and scientist, have been defined. We would like to use a function income_tax( ) to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the class as shown below:

```
class ABC
{
        ….
        ….
    public:
        ….
        ….
        friend void xyz(void);          // declaration
};
```

The function declaration should be preceded by the keyword friend. The function is defined elsewhere in the program like a normal C++ function. The function

Vidyalankar

definition does not use either the keyword friend or the scope operator ::. The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

2. (f) Pointers to Objects

A pointer can point to an object created by a class. Consider the following statement:

```
item x;
```

where item is a class and x is an object defined to be of type item. Similarly we can define a pointer it_ptr of type item as follows:

```
item *it_ptr;
```

Object pointers are useful in creating objects at run time. We can also use an object pointer to access the public members of an object. Consider a class item defined as follows :

```
class item
{
        int code;
        float price;
    public:

        void getdata(int a, float b)
        {
            code = a;
            price = b;
        }

        void show(void)
        {
            cout  << "Code : " << code << "\n";
                  << "Price : " << price << "\n\n";
        }
    };
```

Let us declare an item variable x and a pointer ptr to x as follows:

```
item x;
item *ptr = &x;
```

The pointer ptr is initialized with the address of x.

We can refer to the member functions of item in two ways, one by using the dot operator and the object, and another by using the arrow operator and the object pointer. The statements

```
x.getdata(100, 75.50);
x.show( );
```

are equivalent to

```
ptr→getdata(100, 75.50);
ptr→show( );
```

Vidyalankar

Since *ptr is an alias of x, we can also use the following method:

```
(*ptr).show( );
```

The parentheses are necessary because the dot operator has higher precedence than the indirection operator*.

3. (a) Application of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of application because it can simplify a complex problem. The promising areas for application of OOP include:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neutral networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

The richness off OOP environment has enabled the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly changing the way the software engineers think, analyze, design and implement systems.

3. (b) Constructors with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor complex( ) can be declared as follows:

```
complex(float real, float imag=0);
```

The default value of the argument imag is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the real variable and 0.0 to imag (by default). However, the statement

```
complex C(2.0, 3.0);
```

assigns 2.0 to real and 3.0 to imag. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor A::A( ) and the default argument constructor A::(int = 0). The default argument constructor can be called with either one argument or no arguments. When called with no

Vidyalankar

arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

A a;

The ambiguity is whether to 'call' A::A( ) or A::A(int = 0).

3. (c) Rules for Operator Overloading

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

i) Only existing operators can be overloaded. New operators cannot be created.
ii) The overloaded operator must have at least one operand that is of user-defined type.
iii) We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.
iv) Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
v) There are some operators that cannot be overloaded. (See Table 1)
vi) We cannot use friend functions to overload certain operators. (See Table 2). However, member functions can be used to overload them.
vii) Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
viii) Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
ix) When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
x) Binary arithmetic operators such as +, –, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Table 1 : Operators that cannot be overloaded

Sizeof	Size of operator
*	Membership operator
::	Pointer-to-member operator
?	Scope resolution operator
:	Conditional operator

Table 2 : Where a friend cannot be used

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
→	Class member access operator

3. (d) Overloading unary minus

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

class space
{
        int x;
        int y;
        int z;
```

Vidyalankar

```
public;
        void getdata(int a, int b, int c);
        void display(void);
        void operator- ( );            // overload unary minus
};
void space : : getdata(int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void space : : display(void)
{
        cout << x << " ";
        cout << y << " ";
        cout << z << "/n ";
}
void space : : operator–( )
{
        x = –x;
        y = –y;
        z = –z;
}

int main( )
{
        space S;
        S.getdata(10, –20, 30) ;

        cout << "S : ";
        s.display ( );

        –S;                            // activates operator-( ) function

        cout << "S : ";
        S.display ( );

        return 0;
```

3. (e) Destructors

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer( ) { }
```

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

Vidyalankar

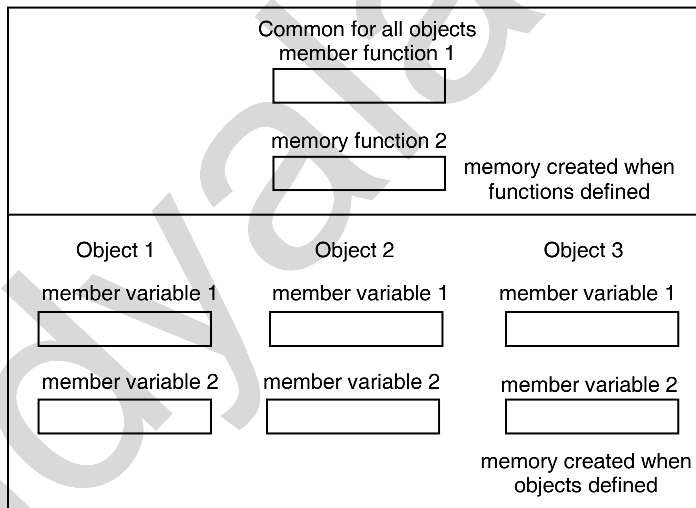
Whenever new is used to allocate memory in the constructors, we should use delete to free that memory. For example, the destructor for the matrix class discussed above may be defined as follows:

```
matrix : : –matrix( )
{
    for(int i=0; i<d1; i++)
    delete p[i];
    delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

3. (f) Memory Allocation for Objects

We have stated that the memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects. This is shown in figure.



4. (a) Program

```
#include <iostream.h>

using namespace std;

// Base class
class shape
{
    public:
        void setWidth (int w)
```

Vidyalankar

```
            {
                width = w;
            }
            void setHeight (int h)
            {
                height = h;
            }
        protected :
            int width;
            int height;
};

// Derived class
class Rectangle: public Shape
{
        public:
            int getArea ( )
            {
                return (width * height);
            }
};

void main ( )
{
        Rectangle Rect;

        Rect.setWidth (5);
        Rect.setHeight (7);

        // Print the area of the object.
        cout << "Total area: " << React.getArea ( ) << end1;
}
```

4. (b) Concept of Virtual Base Classes

A situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in figure. The 'child' has two direct base classes 'parent 1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as indirect base class.

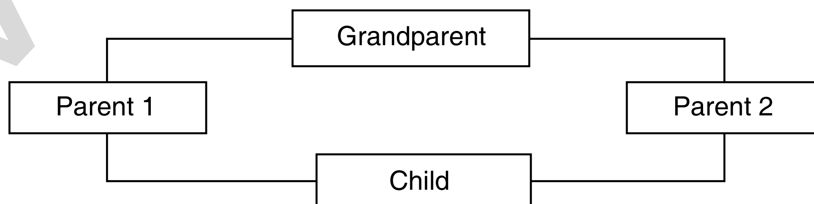


Fig. : Multiple inheritance

Vidyalankar

Inheritance by the 'child' as shown in figure might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have duplicate sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes as shown below:

```
class A                              // grandparent
{
    ….
    ….
};
class B1 : virtual public A          // parent 1
{
    ….
    ….
};
class B2 : public virtual A          // parent 2
{
    ….
    ….
};
class C : public B1, public B2       // child
{
    ….                    // only one copy of A
    ….                    // will be inherited
};
```

4. (c) Function Overloading

Overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded add( ) function handles different types of data as shown below :

```
// Declarations
int add(int a, int b);                    // prototype 1
int add(int a, int b, int c);             // prototype 2
double add(double x, double y);           // prototype 3
double add(int p, double q);              // prototype 4
double add(double p, int q);              // prototype 5
```

Vidyalankar

```
// Function calls
cout << add(5, 10);          // uses prototype 1
cout << add(15, 10.0);       // uses prototype 4
cout << add(12.5, 7.5);      // uses prototype 3
cout << add(5, 10, 15);      // uses prototype 2
cout << add(0.75, 5);        // uses prototype 5
```

4. (d) Static Data Members

A data members of a class can be qualified as static. The properties of a static member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

- It is initialize to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static Member Functions

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

- A static function can have access to only other static members (functions or variables) declared in the same class.
- A static member function can be called using the class name (instead of its objects).

4. (e) We used the copy constructor

```
integer(integer &i);
```

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1. Another form of this statement is

```
integer I2 * I1;
```

The process of initializing through a copy constructor is known as copy initialization. Remember, the statement

```
I2 * I1;
```

will not invoke the copy constructor. However, if I1 and I2 are objects, this statement is legal and simply assigns the values of I1 to I2, member-by-member. This is the task of the overloaded assignment operator(=).

A copy constructor takes a reference to an object of the same class as itself as an argument.

4. (f) Program

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

```
void main( )
{
     int i=0;
     char *ptr[10] = i
               "books",
               "television",
               "computer",
               "sports"
               };
     char str[25];
     clrscr( );
     cout << "\n\n\n\nEnter your favorite leisure pursuit:";
     cin >> str;
     for(i=0; i<4; i++)
     {
               if(!strcmp(str, *ptr[i]))
               {
               cout << "\n\nYour favorite pursuit" << "is available here"
               << endl;
               break;
               }
     }
     if (i==4)
          cout << "\n\nYour favorite "<<" not available here" << endℓ;
     getch( );
}
```

5. (a) Rules for Virtual Functions

i) The virtual functions must be members of some class.
ii) They cannot be static members.
iii) They are accessed by using object pointers.
iv) A virtual function can be a friend of another class.
v) A virtual function in a base class must be defined, even though it may not be used.
vi) The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
vii) We cannot have virtual constructors, but we can have virtual destructors.
viii) While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
ix) When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
x) If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Vidyalankar

5. (b) Pure Virtual Functions

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. For example, we have not defined any object of class media and therefore the function display( ) in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

```
virtual void display( ) = 0;
```

Such functions are called pure virtual functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. Such classes are called abstract base classes. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

5. (c) Program

```
#include <iostream.h>
#include <string.h>
void main( )
{
    char str[ ] = "Test";
    int len = strlen(str);
    for(int i=0; i<len; i++)
    {
            cout << str[i] << i[str] <<*(str+i) << *(i+str);
    }
    cout << endl;
    // String reverse
    int lenM = len / 2;
    len – –;
    for (i=0; i<lenM;l i++)
    {
            str[i] = str[i] + str[len – i];
            str[len-i] = str[i] – str[len–i];
            str[len-i] * str[i] – str[len-i];
            str[i] = str[1] – str[len –i];
    }
    cout << "The string reversed i " << str;

    }
```

5. (d) Differentiate between OOP and POP

Object Oriented Programming (OOP)

Some of the striking features of object-oriented programming are :

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.

- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

Procedure Oriented Programming (POP)

Some characteristics exhibited by procedure-oriented programming are :

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

5. (e) Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The functions that operate on these data are sometimes called methods or member functions.

Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).

5. (f) Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

By inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

Vidyalankar

Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

i) Creating classes that define objects and their behavior.
ii) Creating objects from class definitions, and
iii) Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

6. (a) A Function Friendly to Two Classes

```
#include <iostream>

using namespace std:
class ABC:              // Forward declaration
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - //
class XYZ
{
            int x;
    public
            void setvalue(int i) {x = 1;}
            friend void max(XYZ, ABC);
};
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - //
class ABC
{
            int a;
    public;
            void setvalue(int i) {a = i;}
            friend void max(XYZ, ABC);
};
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - //
void max(XYZ m, ABC n)      // Definition of friend
{
            if(m.x >= n.a)
                cout << m.x;
            else
                cout << n.a;
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - //
int main( )
{
            ABC abc;
            abc.setvalue(10);
            XYZ xyz;
            xyz.setvalue(20);
            max(xyz, abc);
            return 0;
}
```

Vidyalankar

6. (b) Overloaded Constructors

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

class complex
{
    float x, y;
public
    complex() { }                          // constructor no arg
    complex(float a) (x = y = a;)          // constructor-one arg
    complex(float real, float imag)        // constructor-two args
    {x = real; y = imag;}
    friend complex sum(complex, complex);
    friend void show(complex);
};
complex sum(complex c1, complex c2)        // friend
{
    complex c3;
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return(c3);
}
void show(complex c)                   // friend
{
    cout << c.x << "+j" << c.y << "\n";
}

int main( )
{
    complex A(2.7, 3.5);                   // define and initialize
    complex B(1.6);                        // define and initialize
    complex C;                             // define

    C = sum(A, B);                         // sum( ) is a friend
    cout << "A = "; show(A);               // show( ) is also friend
    cout << "B ="; show(B);
    cout << "C ="; show(C);

// Another way to give initial values (second method)

    complex P, Q, R;                       // define P, Q and R
    P = complex(2.5, 3.9);                 // initialize P
    Q = complex(1.6, 2.5);                 // initialize Q
    R = sum(P, Q);

    cout << "\n";
    cout << "P = "; show(P);
    cout << "Q = "; show(Q);
    cout << "R = "; show(R);

    return 0;
}
```

Vidyalankar

6. (c) Implementation of Destructors

```
#include <iostream>

using namespace std;

int count = 0;

class alpha
{
    public;
        alpha( )
        {
            cout++;
            cout << "nNo.of object created " < count;
        }

        –alpha ( )
        {
            cout << "\nNo. of object destroyed " << cout;
            cout - -;
        }
};

int main( )
{
    cout << "\n\nENTER MAIN\n";

    alpha A1, A2, A3, A4;
    {
        cout << "\n\nENTER BLOCK1\n";
        alpha A5;
    }

    {
        cout << "\n\nENTER BLOCK2\n";
        alpha A6;
    }
    cout << "\n\nRE-ENTER MAIN\n";

    return 0;
}
```

❑ ❑ ❑ ❑ ❑

Vidyalankar