BCS - 304

Q1(i)

ans(i) Structures in C++ can contain 2 types of Members.

* Data Member : These are normal C++ variables. We can create a structure with variables of different data type in C++
* Member Function: These are normal C++ functions. Along with variables, we can also include functions inside a structure declarations.

Once the objects are declared of the structure, we can the access the members i.e. Data members and Member functions using the syntax dot (.) between the name of the object and name of the member.

Eg.

```
Class Arpit
{
 Public:
 int age;
 Void printer (){
  cout << " Name is Arpit , age is " << age;
  }
} obj1;
```

Here, object name is obj1, and member are age & printer function

So,

to access these members, we do

obj1.age = 19 (data member)

obj1.printer() ; (Member function)

Q-1 (ii)

Ans 1 (ii) Pointer to structure holds the address of the entire memory block that stores a structure. Defined as the pointer which point to the address of the memory block that stores a structure.

Eg.
```
struct point
{ int value;
};
// Driver code
int main()
{
 struct point s;
 struct point *ptr = &s;
 return 0;
}
```

In the above code s is an instance of struct point and ptr is the struct pointer because it is storing address of struct point.

Eg.
```
# include <iostream.h>
# include <conio.h>
  void main()
 {
  struct date
  {
   short int dd, mm, yy;
  } join_date = {19, 12, 2006};

  date *date_ptr;
  date_ptr = &join_date.
```

```
cout << "Printing the structure elements using the structure variable"
cout << "dd = " << join_date.dd << ", mm = " << join_date.mm << ",
                yy = " << join_date.yy << "\n";
```

```
cout << "\n printing the structure elements using structure pointer
                                                    \n";
cout <<"dd ="<< date_ptr ->dd <<", mm ="<<date_ptr -> mm";
        y = " << date_ptr ->yy <<"\n";
    getch ();
    }
```

Output -

Printing the structure elements using struct variable
dd = 19 , mm = 12, yy = 2006
Printing the struct elements using struct pointer.
dd = 19 , mm = 12, yy = 2006.

Q-2(i)
Ans 2(i)

SOP	OOP
• A Programming paradigm that divides the code into modules or functions	• A programming paradigm based on the concept of objects, which contain data in the form of attributes and code form (procedu
• Divides program to set of function where each function act as subprogram	• Focuses on representing a program using a set of objects which encapsulates data and object.
• Difficult to modify structured programs	• Easier to modify objet oriented program.
• It follows top-down approach	• Follows bottom up approach.
• SOP can solve moderately complex programs	• OOP can solve any complex program
• Structured programming is less secure as there is no way of data hiding	• OOP is more secure with data hiding feature.
• SOP provides less reusability more function dependency	• OOP provides more reusability less function dependency.

Q-2 (ii)

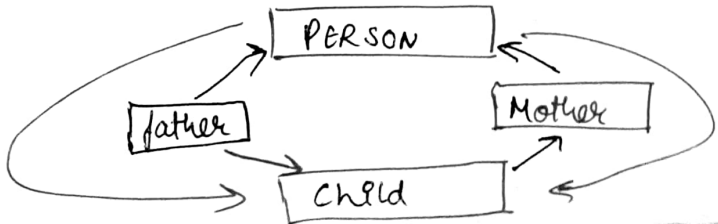
Ans Q(ii) Private mode ⇒ If we derive a sub class from a private base class. Then both public members & protected members of the base class will become private in desired class.

Hybrid Inheritance ⇒ It is implemented by combining more than one type of inheritance. for eg. combining hirar chal inheritance and multiple inheritance.

```
class person { };
class student : [redacted] private person { };
void eat (const Person & p) { }
void study (const student & s) { }
int main()
{
    person p;          // p is a person
    student s;         // s is a student
    eat (p);           // fine, p is a person
    eat (s)            // error !. s isn't a person.
    return 0;
}
```

Diamond problem

The Diamond problem occurs when a child class inherits from two parent classes who both share a common grandparent class. This is illustrated in the diagram below:



Here, we have a class child inheriting from classes father and mother. These two classes, in turn inherit the class Person because both father and mother are person.

As shown, class child inherits the traits of class person twice - once from father and again from mother. This gives rise to ambiguity since the compiler fails to understand which way to go.

Q-3(i)

Ans 3(i) The building block of C++ that leads to object-oriented programming is a class. It is a user-defined data type, which holds its own data members and member function. which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For eg: Consider class of cars, Different names and brands, but share common properties like all have, 4 wheels, speed limit, mileage etc, So, cars is class and other features are properties.

~~Class is a user defined data~~

Object :- An object is an indentifiable entity with some characteristics and behaviour - An object is an instance of a class, when a class is defined, no memory is allocated but when it is instantiated memory is allocated.

```
Class person
{
  char name[20];
  int id;
  public:
    void get details ()
};
int main ()
{
  person p1; // p1 is a object
}
```

Object take up space in memory and have an associated

address like a record in pascal or structure or union in C.

When program is executed the object interact by sending messages to one another.

Each object contains data and code to manipulate the data.

Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the object.

Q-3 ii)
Ans 3(ii)

Friend function is defined function in C++, the protected private data of a class can be accessed using the function.

- They must be prefixed with the 'friend' keyword in declaration.

```
class My class {
    private:
    int a;
    public:
    friend void fun ();
    };
```

can directly access

Keypoints :-

- The function is not in the scope of the class to which it has been declared as friend.
- It can be declared either in the private or the public part
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.

*. Program to define member function outside the class:

→
```
# include <iostream>
using namespace std;
class eg.
{
private
    int val;
public:
    // function to assign value
```

```
    void init_val (int v)
    {
        val = v;
    }

    // function to print value
    void print_val ()
    {
        cout << " val : " << val << endl;
    };
int main()
{
    // create object
    Example Ex;
    Ex . init_val (100);
    Ex . print_val ();
    return 0;
}
```

Output : val : 100

Q-4(i)

Ans 4(i) Simple program using insertion overloading in C++

```
# include <iostream>
using namespace std;
class complex
{
public:
    int real;
    int img;
  friend ostream & operator << (ostream & o, complex & cl);
};
ostream & operator << (ostream & o, complex & cl)
{
o << cl . real << "+i" << cl . img;
return o;
}
int main ()
{
complex cl ;
cl . real = 5
cl . img = 3
 cout << cl;
 operator << (cout, cl);
 return 0;
}
```

Output :

5 + i3

Q-4(ii)

Ans 4(ii) A virtual function is a member function that is declared within a base class and redefined by a derived class. To create virtual function, precede the function's declaration. In the base class with the keyword virtual when a class containing virtual function is inherited. the derived class redefines the virtual function to suit its own needs.

→ Base class pointed can point to derived class object. In this case, using base class pointer if we call some function which is in both classes, then base class function is involved But if we want to involve derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how virtual function support runtime polymorphism.

→ Program Code:

```
Class A
{
  Int a;
  Public:
    A()
    {
      a= 1;
    }
  Virtual void show ()
  {
    cout << a;
  }
};
```

```
class B : public A
{
 int B;
  Public :
   B()
   {
      b = 2;
   }
   virtual void show()
   {
    cout << b;
   }
};
int main()
{
  A * pA;
  B oB;
  pA = & oB;
  pA → show();
  return 0;
 }
```

→ Output is 2 since pA points to object of B and show() is virtual in base class A.

Q-5(i);

Ans 5(i) Private data members of class can be accessed from a non member function by accessing the memory location address of the private data member by the way of pointer variable.

→

```
#include <iostream>
using namespace std;
class A {
int x;
Public:
  A() { x = 2; }
};
class B {
  int y;
  Public:
  B() { y = 3; }
};
int main()
{ A a;
  B b;
  int * p = (int *) & a;
  int * q = (int*) & b;
  int temp;
  * temp = * p;
  * p = * q
  * q = * p
  return 0;
}
```

Q-5 ii

Ans 5(ii)

Program :

```
#include <bits/stdc++.h>
using namespace std;

class Integer {
private:
int i;

public:
   // Parameterised constructor Integer (int i=0)
  {
    this -> i= i;
  }
// overloading the prefix operator Integer operator B++()
{
 Integer temp;
 temp.i = ++i;
 return temp;
 }
// function to display the value of i void display()
{
  cout << "i =" << i << endl;
}
};
```

```
// Driver function
int main ()
{
  integer i1(3);
  cout << "Before increment:" ;
     i1 , display()
  // using the pre-increment operator ;
     integer i2 = ++ i1 ;
    cout << "After pre increment :" ;
     i2 . display();
    }
```

Prefix notation causes a variable to be updated before its value are used in expression,

statement

```
id x 1 = ++ id x 2;
```

has exactly the similar effect as statement

```
id x 1 = id x 2 ++;
```

when operators are overloaded, no distinction in prefix & postfix. The problem is circumvented in advanced implementations of C++.