Year: 2019
Semester: Spring.

Q.N.1

A Ans.

Software development process is not linear, but is an empirical process because the requirements are expected to evolve; the process can't be defined from the start with a set series of steps. Yes, of course there will be plans and prototype design for the development process but it always remain unsure where the software development process halts because of bug and the system problems.

Second part:

Data abstraction is one of the important feature in OOP. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Abstraction using access specifier:

Access specifier are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members.

For example:

- Members declared as public in a class, can be accessed from anywhere in the program.
- Members declared as private in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

We can easily implement abstraction using the above two features provided by access specifier. Say, the members that define the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public member can access the private member as they are inside the class.

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

class ImplementAbstraction{
  private:          //private access specifier
      int a,b;
  public:           //public access specifier.
      void display()
      {
        cout<<"Garbage value of a :"<<a<<endl;
        cout<<"Garbage value of b :"<<b<<endl;
      }
};

int main()
  {
    ImplementAbstraction obj;
    // obj.a or obj.b is not accessible.
    obj.display(); // it is accessible, because it is
                      defined in public
  }.
```

B. Ans.

Differences between structure and class in C++.

| Structure | Class. |
|---|---|
| → A structure is a grouping of variables of various datatypes refrenced by same name. | A class is defined as a collection of related variables and functions contained within a single structure. |
| → All members are set default to 'public'. | All members are set default to private. |
| → Syntax for Declaration:<br>struct structure_name {<br>type struct_member1;<br>type struct_member2;<br>:<br>};<br>Structure instance is called 'structure variable.' | Syntax for Declaration:<br>class class_name {<br>data member;<br>member function;<br>:<br>};<br>A class instance is called 'object'. |
| → It doesnot supports inheritance. | It supports inheritance. |
| → It is used for small amounts of data. | It is used for huge amounts of data. |

Second part:

In C++, there are three access specifiers used in class. they are public, private, protected.

- public: members are accessible from outside the class.
- private: members cannot be accessed (or viewed) from outside the class.
- protected: members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

The following example demonstrates the uses of access specifier:

```
class MyClass {
    public:     // public access specifier.
      int x;    // public attribute.
    private:    // private access specifier
      int y;    // private attribute
};

int main() {
    MyClass myObj;
    myObj.x = 25;  // Allowed (public)
    myObj.y = 50;  // Not allowed (private)

    return 1;
}
```

Q.N.2.

A.Ans.

Friend functions of the class are granted access to private and protected members of the class in C++. They are ~~granted~~ defined outside the class's scope.

There are situations in programming where we would want two classes to share their members. These members may be data members or class functions in such cases, we make the desired function, a friend to those classes. Generally non-member functions cannot access the private member of a particular class. Once declared as a friend function, the function is able to access the private and the protected members of these classes.

Advantage of friend function is that we can access the private and protected members of the class for our usuage.
Disadvantage of friend function is that it doesn't follow data hiding principle of class.

Example program:

```
#include<iostream>
using namespace std;
class Box {
    private:
        int length;
    public:
    friend int printLength(Box); // friend function.
};
```

```
int printLength (Box b)
{
    b.length = 10;
    return b.length;
}

int main() {
    Box b;
    cout << "Length of box: " << printLength(b) << endl;
    return 0;
}
```

It is absolutely necessary to pass class object to the friend function else it will not be possible to access the private and protected data members.

B. Ans.

- Dynamic Memory allocation is the process of allocating memory dynamically as per the requirement of application program.
- the memory allocated dynamically are stored in heap.
- the compiler doesn't free up the dynamically allocated memory. We have to manually freeup memory and should be careful while allocating and deallocating member, as our forgetfullness might bring out load in program.

- We can allocate memory by using new operator and deallocate memory by using delete operator.

```
//Program to dynamically allocate memory for integer value.
#include<iostream.h>
#include<conio.h>

void main(){
    int *ptr1 = NULL;
    ptr1 = new int; // declaring a integer variable dynamically.
    *ptr1 = 100;
    int *ptr2 = new int(1234);

    cout<<"Value at address pointed by pointer variable 1:"
        <<*ptr1<<endl;
    cout<<"Value at address pointed by pointer variable 2:"
        <<*ptr2<<endl;

    delete ptr1; // memory pointed by ptr1 is released.
    delete ptr2; // memory pointed by ptr2 is released.

    getch();

}
```

Q.N.3.

A. Ans.

Subtype:

Subtype is subclass relationship in which the principle of substitutability is maintained.

Principle of substitutability:

If we have two classes A and B such that class B is subclass of A, it should be possible to substitute instances of class B for instances of class A in any situation with no observable effect.

this pointer:

'this' pointer holds the address of current calling object when calling a member function.

objectA.functionX();

Here 'this' pointer holds the address of objectA.

- Whenever a non static member function is called, this pointer is passed as hidden argument implicitly.

Example:

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

class Complex
{
```

```
int real,img;
public:
    Complex (int real, int img)
    {
        this -> real = real; // this -> real indicates calling object
        this -> img = img;                    'real' data member.
    }

    void display()
    {
        cout << "Real part : " << this -> real << endl;
        cout << "Imaginary part : " << this -> img << endl;
    }
};

void main()
{
    Complex obj(10,20);
    obj.display();
    getch();
}
```

Virtual function:

- In C++, function overriding is achieved using virtual function.
- A virtual function is a member function which is declared within a base class and overridden by derived class.
- When we use same function name with same signature in base class and derived class, the function in base class is declared as virtual function.

- When the class containing virtual function is inherited. the derived class redefines virtual function to perform task respective to derived class.

// Implementing virtual function.

```
#include<iostream>
using namesp
class Base
{
   public:
     virtual void display()
     {
        cout<<" This is base"<<endl;
     }
};

class Derived : public Base
{
    public:
        void display()
        {
           cout<<" This is derived"<<endl;
        }
};

void main()
{
```

```
Base *baseptr, bobj;
baseptr = &bobj;
cout << "calling from Base class pointer that points to Base
  object " << endl;
baseptr -> display();

Derived dobj;
bpsetr = &dobj;
cout << endl << endl << " calling from Base class's pointer that
points to Derived's object " << endl;
bptr -> display();
getch();
}
```

Output:

Calling from Base class pointer that points to Base object.
This is base.

Calling from Base class's pointer that points to Derived object.
This is derived.

3. Ans.

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

class Base{
  private:
      int xSum, ySum;
```

```
public:
    int x1, x2, y1, y2;
    void input()
    {
        cout << "Enter coordinate of first vector (x1, y1): ";
        cin >> x1 >> y1;
        cout << " Enter coordinate of second vector (x2, y2): ";
        cin >> x2 >> y2;
    }

    friend void display(Base b);
};

class Derived : public Base
{
    public:
        void add_vector(Base b)
        {
            b.xSum = b.x1 + b.x2;
            b.ySum = b.y1 + b.y2;
        }
};

void display(Base b)
{
    cout << "Sum is: (" << b.xSum << ", " << b.ySum << ")";
}
```

```
int main()
{
    Base b;
    b.input();

    Derived d;
    d.add_vector(b);

    display(b);
    return 1;
}
```

Q.N.4.

A. Ans.

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

class Person{
  public:
      char name[30];
      int age;

      void setInfo()
      {
          cout<<"Enter person name and age:";
          cin>>name>>age;
      }
```

```
    void displayInfo(){
        cout << "Name : " << name << endl;
        cout << "Age : " << age;
    }
};

class Student : public Person {

    public:
        int roll;
        Student (){
            roll = 0;
        }

        Student ( int roll) {
            this -> roll = roll;
        }

        void displayStudentRoll() {
            cout << "Roll : " << roll;
        }

};

class Employee : public Person {
    public:
        int id;
        Employee(){
            id = 0;
        }
```

```
Employee (int id) {
    this->id = id;
}

void displayEmployeeID(){
    cout << "ID :" << id;
}

};

int main()
{
    Student student(1);
    student.setInfo();
    student.displayInfo();
    student.displayStudentRoll();

    Employee employee(100);
    employee.setInfo();
    employee.displayInfo();
    employee.displayEmployeeID();

    return 1;
}
```

B. Ans.

Compile-Time polymorphism can be achieved by doing function overloading. or can be done by operator overloading.

Example program:

```
#include <iostream>
#include <cmath>

#define PI 3.14
using namespace std;

class Volume
{
    public:
        //sphere
        float calculateVolume( float radius)
        {
            return (4/3) * PI * pow(radius, 3);
        }

        //cylinder
        float calculateVolume (float radius, float height)
        {
            return PI * pow(radius, 2) * height;
        }
};
```

```
int main()
{
    Volume volume;
    float sphereRadius, cylinderRadius, cylinderHeight;

    cout<<endl<<" enter radius of sphere :";
    cin>> sphereRadius;

    cout<<endl<<" enter radius and height of cylinder :";
    cin>> cylinderRadius >>cylinderHeight;

    cout<<" Volume of sphere :"<<volume.calculateVolume(sphereRadius);
    cout<<endl<<"Volume of cylinder :"<<calculateVolume(cylinder
      Radius, cylinderHeight);

    return 1;
}
```

Runtime polymorphism can be achieved by using virtual function.

Example program :

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

class Base {
    public :
        virtual void display()
        {
```

```
    cout << "This is base" << endl;
  }
};

class Derived : public Base {
  public:
    void display()
    {
      cout << "This is derived" << endl;
    }
};

void main()
{
    Base *bptr, bobj;
    bptr = &bobj;
    cout << "Calling from Base class's pointer that
    points to Base's object" << endl;
    bptr -> display();

    Derived dobj;
    bptr = &dobj;
    cout << endl << "Calling from Base class's pointer that
    points to Derived's object" << endl;
    bptr -> display();

    getch();
}
```

Here, In this example we had made virtual function in base class so as a result when we call the display() method pointed by base class to its object then the display () method of base class is called, whereas in second case; the base class pointer points to derived class and calls display() method, at this time, the display method of derived class is executed. Hence there is the runtime polymorphism which is possible by making virtual function.

Q.N.5.

A.Ans.

Type casting is the process of converting one form of data-type to another.

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

//polar class.
class Polar {
public:
    float r, th;
    Polar(){}
    Polar(float a, float b) {
        r=a;
        th=b;
    }
};
```

```
class Rectangle {
    float x,y;
public:
    Rectangle(){}
    Rectangle(Polar p){
        x = p.r * cos(p.th);
        y = p.r * sin(p.th);
    }
    void show(){
        cout<<"In rectangular form : x = "<<x<<" and
        y = "<<y;
    }};
```

B. Ans.

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

class Complex
{
    private:
        int real, img;
    public:
        Complex(){ }
        Complex(int real, int img){
            this -> real = real;
            this -> img = img;
        }

        Complex operator + (Complex c){
            Complex temp;
            temp.real = real + c.real;
            temp.img = img + c.img;
            return temp;
        }
};
```

```
int main()
{
    Complex c1(1,2), c2(2,3), c3;
    c3 = c1 + c2.
    cout << "Sum is :" << (c3.real) << "+" << c3.img << "i";
    return 1;
}
```

Q.N.6.

A. Ans.

Through the use of templates we can define a general classes or functions to handlle different datatypes.
It reduces the need of writing the same code again and again thus simplifying the programming technique.

Example :

Let say we are making a swap between two numbers. Our numbers can be of any type, to make sure that it works for different types of number, we make floating, decimal (double) and int data-types having function which excepts these different types of value as a argument. Here, if we see then we can wrote same algorithm for three functions but we are making our code long and non-usefulness. So, in this case functional templates comes in.

To declare template we define at the top of our program and instead writing particular datatype in function parameter and body we make use of template variables, so this simplifies into one ~~prog~~ method only. Hence when paring

the value to the method, we can simply specify the datatype and the method will execute on that accord. That's why template are very very useful.

Second part:

Exception handling technique is the mechanism to detect error and prevent program code from wrong happening during run-time of the program. It is one of the best technique to make sure of all error and prevent the error as per our requirements.

The exception handling can be achieved by embedding code inside try block and embedding the error handling code inside catch block.
The following example will demonstrate the idea of try-catch block (or exception handling).

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

void main()
{
    char error[] = "ERROR; Division by zero";
    float num1, num2, result;
    cout << "enter two numbers: ";
    cin >> num1 >> num2;
    try {
        if (num2 != 0)
        {
```

```
    result = num1/num2;
    cout<<" The result after division is"<<result<<endl;
}
else throw error;

catch(char e[])
{
    cout<<" The denominator must be non zero"<<endl<<
        e<<endl;
}
cout<<endl<<" Thankyou"<<endl;
getch();
}
```

Q.N.7.

A. Message passing in C++:

In C++ classes, we can pass message to objects in various ways. One way is from the method and another way is into constructor.

Let us consider a method, which is member function of class. It accepts a number.

```
class Test{
public:
    Void getNum(int a)
    {
        cout<<" Given Num:"<<a<<endl;
    }
};
```

```
int main()
{
    Test obj;
    obj.getNum(10);  // passing message.
    return 1;
}
```

As, in the example I have defined only one parameter and only one method but we can have multiple methods in which we can pass our data of different types as per the requirement

B. inline function:

- Inline function are special type of function which is used for writing small piece of code.

- These function is faster as compared to normal function and instead of going to function defintion, the function itself gets expanded on that line in which it is called.

- It is defined as similar to normal function but we add inline in front of the return type.

Syntax:

```
inline returnType functionName (parameters)
{
    function body;
}
```

- While writing code for inline function we must make sure that we don't include any loops nor switch statements or other complex & long operation. But if we include so then we will encounter warning or errors from compiler during program execution.

Example program:

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

inline int area(int l, int b){
    return l*b;
}

void main(){
    int len, bre, result;
    cout<<"Enter values of length and breadth";
    cin>>len>>bre;
    result=area(len, bre);
    cout<<"Area of rectangle is"<<result;
    getch();
}
```

(Remaining one)

6. B. Ans.

CRC (Class/Component Responsibility Collaborator (Card) is a technique to identify the class attributes and it's responsibility with other members of the program.

As, mentioned it consists of three sections:

1. Class: A class represents a collection of similar objects.
2. Responsibility: It is something that a class knows or does.
3. Collaborator: Another class that is a class interacts with to fulfil its responsibilities

CRC card:

| Component | |
|---|---|
| | |
| Responsibility | Collaborators |
| | |
| | |

CRC card for student.

| student. | |
|---|---|
| Responsibilities | Collaborators |
| name | |
| rollnumber | Exam Department |
| reg. number | Account Department |
| attend class | Library. |
| take exam | |
| pay fee | |

Sequence diagram:

- sequence diagrams is used to model the interactions between objects in a single use case.
- It illustrates how the different parts of a system interact with each other to carry out a function.

Sequence diagram parts:

- Lifeline: It represents an individual participant in the interaction.

- Activations:

  A thin rectangle on lifeline, represents the period during which an element is performing an operation.

- Call Message:

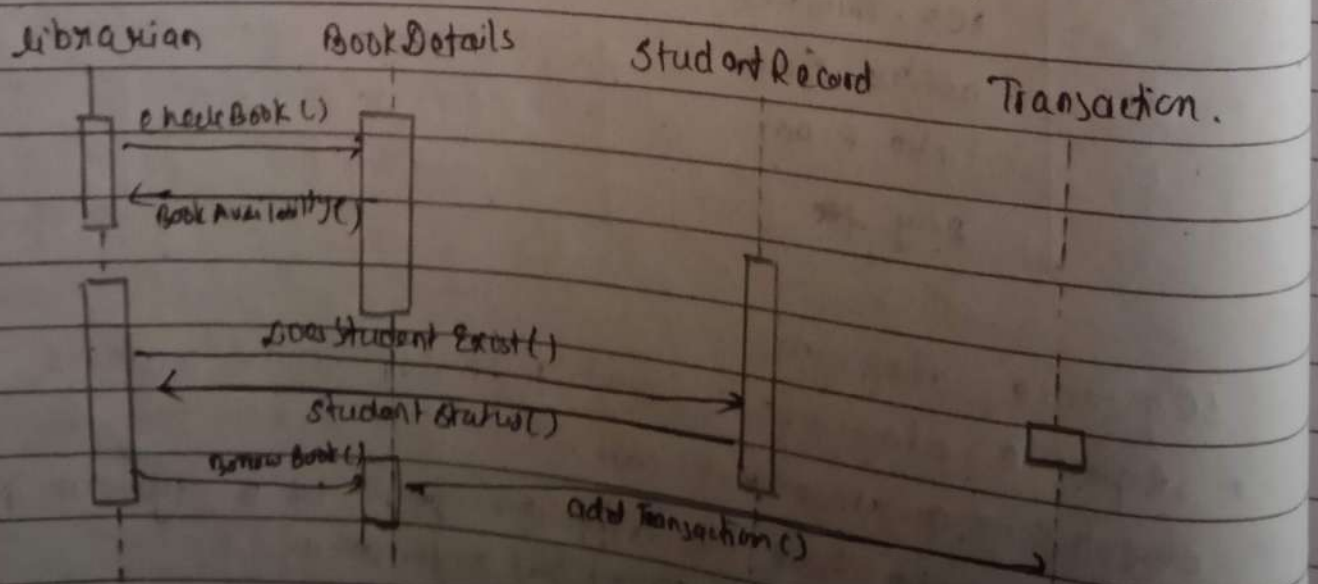
  A message defines a particular communication bet^n lifelines of an interaction.

- Return Message:

  Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.

Example:

(Sequence diag. of borrowing book from library)



Semester : Fall
Year : 2020

Q.N.1.

A. Ans. Object oriented programming language is a type of programming paradigm where classes & its concepts are used in order to simulate circumstances of real world.

There are various key features of object oriented programming language and they are:

1. Inheritance:

From inheritance, we can derive all the properties of the parent class and make use of same funcanalities without need of re-doing it again.

Ex:

```
class Base{
public:
    void method1(){
        ...
    }
    void method2(){
        ...
    }
};

class Derived:public Base{
public:
    ...
};
```

Here the 'Derived' class achieves all the features and can make use of it without redifining it.

2. Polymorphism:

Polymorphism is another great feature in OOP. Through polymorphism we can achieve function overloading (the mechanism of creating function with same name but with different parameter signatures), function overriding (the mechanism of overriding the funcanalities of Base class by child class), operator overlodding (the mechanism in which we program operator to perform operations for our data-types). and many more.

Example of function overloading:

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

void add (int a, int b){
    return a+b;
}

void add (float a, float b){
    return a+b;
}

int main(){
  cout << "Int sum : " <<(add (1,2)<<endl;
  cout << "float sum: " <<add (1.2+6.7)<<endl;
  return 1;
}
```

B. Ans.

Yes, its true that friend function is not the member function of a class because they are not defined inside the class scope. They are just normal function but can only have previlage to access private and protected members of class if they are declared as a friend inside the class.

Example program:

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

class Adder {
    private:
        int a, b, result;
    public:
        void add(){
            cout << " Enter any two numbers : ";
            cin >> a >> b;
            result = a+b;
        }

        friend void displayResult (Adder);
};

void displayResult (Adder a)
{
    cout << "Sum is : " << a.result;
}
```

```
int main()
{
  Adder obj;
  obj.add();
  displayResult(obj);
  return 1;
}
```

since the friend function is defined outside the class, the friend function is called as normal function call without needing any class references. so, in this way function is not the member function of class.

Q.N.2.

A.ANS. Inline function are special types of function which are used for faster execution of smaller reusable code. Their control flow is different than that of normal function. Their code is replaced in that line, where they are called; hence making execution time a bit faster.

For example:

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

inline ~~void~~ int add(int a, int b){
        return a+b;
  }
```

```
int main()
{
    cout <<"Sum of 2 and 3 is" <<add(2,3);
    return 1;
}
```

Here, in this example inline function is created by using inline keyword followed by normal function syntax. In main method its called by passing required arguments. If the operation was too long and we have definition everything on the inline function then the compiler would have thown a warning or the error message. Because inline function is only for small piece of code. For loops, switch statements and recursive function is not suitable for inline function.

B. Ans.

Constructor are member function of class that is used to initialize data when the object associated with that class is created.

Default Constructor: The constructor which doesn't have any parameters is default constructor.

Parameterized constructor: The constructor which can accept parameters are parameterized constructor. They come very handy while making use of initialization for certain specific value for data members.

Example program:

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

class Vechile {
    private:
        int mileage, cost;
    public:
        Vechile() {     // Default constructor.
            mileage = 0;
            cost = 0;
        }

        Vechile(int m, int c) {  // Parameterized
                                 // Constructor
            mileage = m;
            cost = c;
        }

        void display();
};

void Vechile :: display() {
        cout << "Mileage: " << mileage << endl;
        cout << "Cost : " << cost << endl;
    }

int main()
{
        Vechile v;  // normal constructor is used
                    (default)
```

```
v.display(); // displays mileage:0 and cost:0

Vechicle v2(50, 100000); // parametrized constructor is called.
v2.display(); // displays mileage:50 and cost: 100000

return 1;

}
```

Q.N.3.

A. Ans.

Copy constructor is another special type of constructor which is used for copying data of one object to another.

Yes, it is possible to pass object as argument in copy constructor.

Example:

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

class Point {
  private:
      int x, y;
  public:
      Point(int x1, int y1) { x = x1; y = y1; }
      // copy constructor.
      Point(Point &p1) { x = p1.x; y = p1.y }

      int getX() { return x; }
      int getY() { return y; }
};
```

```
int main()
{
  Point P1(10,15); // Normal constructor is called here.
  Point P2=P1; // copy constructor is called here.

  // Accessing value assigned by constructor.
  cout << "P1.x = " << P1.getX() << ", P1.y=" << P1.getY();
  cout << "\n P2.x = " << P2.getX() << ", P2.y =" << P2.getY();
  return 0;
}
```

Output:

P1.x = 10, P1.y = 15

P2.x = 10, P2.y = 15.

B. Ans.

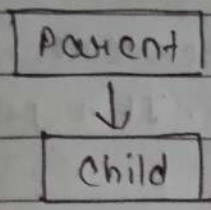
Inheritance supports reusability Because when we inherit some class, we can access ~~acct~~ the data members and function of the parent class, which is beneficial for us, as we don't have to create it again and again, We can make reuse of it with ease.

The different forms of inheritance are:

1. Single level inheritance:

In this type of inheritance, there is one class which only one class is derived from it.

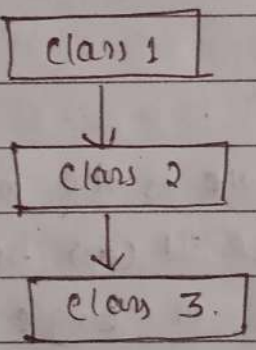
Structure:




2. Multi-Level inheritance:

In this type of inheritance, multiple childs are derived from the one-another parent. i.e. it is similar to single inheritance but another single inheritance is added onto it to make it multi-level inheritance.

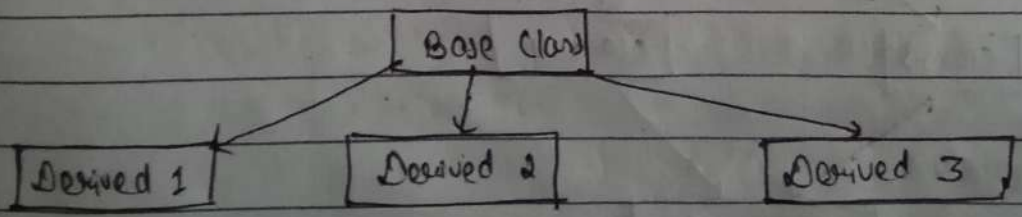
Structure:




3. Hierarchical inheritance:

In this type of inheritance, multiple classes are inherited from single base class.

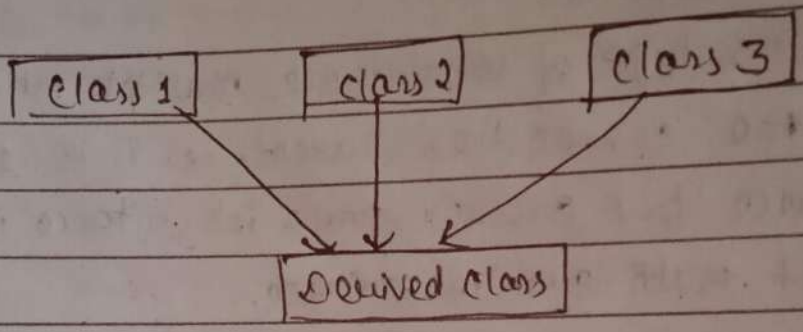
Structure:



4. Multiple Inheritance:

In this type of inheritance, a single class is derived from multiple classes. It is just the opposite of hierarchical inheritance.

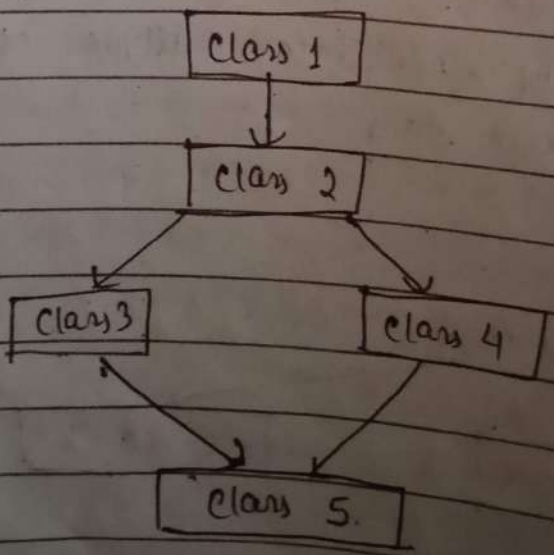
Structure:




5. Hybrid inheritance:

Hybrid inheritance is the mixed form of two or more inheritance types. It can be whether multiple and multilevel inheritance. or single and multiple inheritance and soon.

Structure:



Q.N.4.

A. Ans.

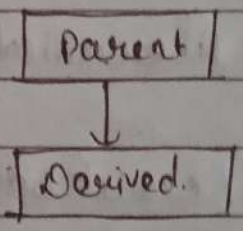
Since private data members cannot be accessed by derived classes hence a new access specifier 'protected' is introducted which makes its data member accessible to the derived classes. This solves the problem of sharing private like data betn the classes. So, often protected access specifier is like private access specifier.

Example program of different types of inheritance:

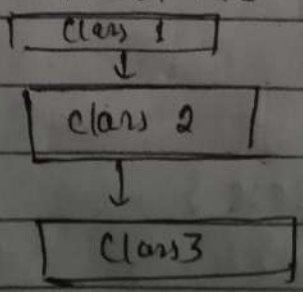
1. Single inheritance or single level inheritance.

```
class Parent {
    ....
    ....
};

class Derived : public Parent {
    ....
    ....
};
```




2. Multi level inheritance

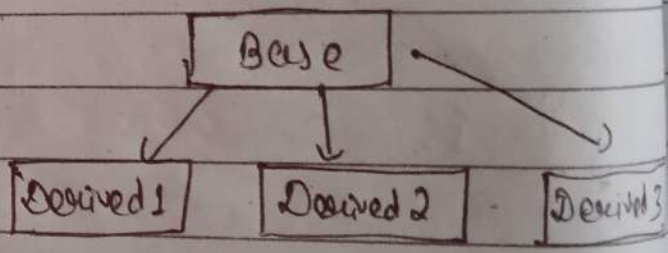


```
class Class1 {
    ....
    ....
};

class Class2 : public Class1 {
    ....
    ....
};

class Class3 : public Class2 {
    ....
    ....
};
```

3. Hierarchical Inheritance:



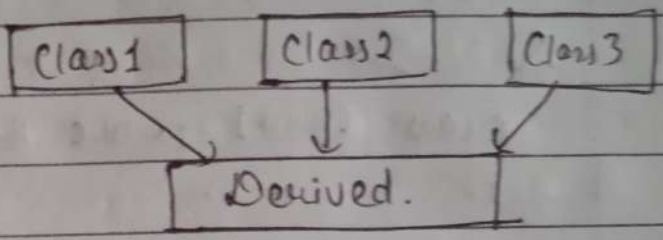

```
class Base {
    ....
    ....
};

class Derived1 : public Base {
    ....
    ....
};

class Derived2 : public Base {
    ....
    ....
};
```

```
class Derived3 : public Base {
    . . . .
    . . . .
};
```

4. Multiple inheritance :



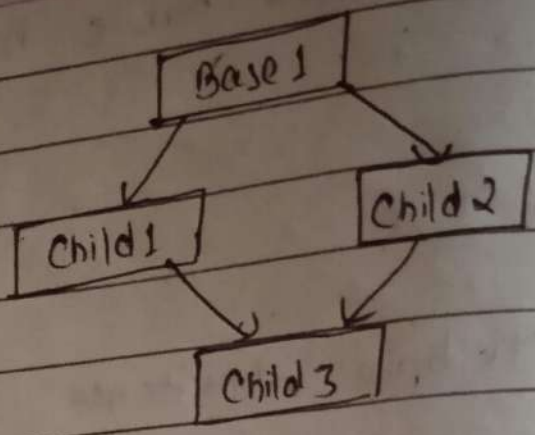

```
class Class1 {
    . . . .
    . . . .
};

class Class2 {
    . . .
    . . . .
};

class Class3 {
    . . . .
    . . . .
};

class Derived : public Class1, public Class2, public Class3 {
    . . . .
    . . . .
};
```

5. Hybrid inheritance:




```
class Base1 {
    . . . .
    . . . .
};

class Child1 : public Base1 {
    . . . .
    . . . .
};

class Child2 : public Base2 {
    . . . .
    . . . .
};

class Child3 : public Base1, public Base2 {
    . . . .
    . . . .
};
```

6. Ans.

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

class Complex
{
    private:
        int real, img;
```

```
public:
    void input()
    {
      cout << "enter real and imaginary part:";
      cin >> real >> img;
    }

    Complex operator * (Complex c) {
        Complex temp;
        temp.real = real * c.real;
        temp.img = img * c.img;
        return temp;
    }

    void display() {
        cout << "Real :" << real << " img :" << img;
    }
};

int main(){
    Complex c1, c2;
    c1.input();
    c2.input();
    Complex result = c1 * c2; // multiplication of two objects.
    result.display();
    return 1;
}
```

Output:
Enter real and imaginary part : 1 2
Enter real and imaginary part : 2 3.
Real : 2 img : 6

Q.N. 5.

A. Ans.

In inheritance, when the base class and derived class contains the function with same name and same parameter signature, then at calling time of the function, it is found to be pointing at the base class function whereas derived class function is never called by base class pointer. Hence in order to overcome this problem and do function overriding virtual function are used.

Virtual function is similar to normal function in syntax but is followed by virtual keycoard at beginning of the function.

By making use of virtual function we can point to current version of function and make our program run.

Let us take an example program:

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

class Base
{
```

```
public:
    void print() {
        cout << "Printing from Base class" << endl;
    }
};

class Derived : public Base {
public:
    void print() {
        cout << "Printing from Derived class" << endl;
    }
};

int main() {
    Base *bptr, baseObj;
    bptr = &baseObj;
    bptr -> print();
    // output : Printing from Base class.

    Derived dObj;
    bptr = &dObj;
    bptr -> print();
    // output : Printing from Base class.
}
```

Here, In the above program, though base class pointer is pointing to derived class object but still the function call is made of the base class. Because the compiler doesn't get the current version of function. Hence to solve this conflict we add virtual keyword in front of method of base class.

Q5,

```
virtual void print()
{
    cout << "Printing from Base class";
}
```

Now, what happens is, ~~base~~ derived class will also have virtual version of its function. Hence at this time we get called from respective method of respective class.

B. Ans.

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

template<typename T>
void swapData(T a, T b)
{
    cout<<endl<< "Before swapping: "<<a<< ", "<<b;
    T temp;
    temp=a;
    a=b;
    b=temp;

    cout<<endl<<" After swapping: "<<a<<", "<<b<<endl;
}

int main()
{
    float a=12.0, b=12.32;
    swapData<float>(a,b);
```

```
swapData<int>(12,14);
swapData<char>('a','d');
return 1;
}
```

Output:

Before swapping : 12, 12.32
After swapping : 12.32, 12

Before swapping : 12, 14
After swapping : 14, 12.

Before swapping : a, d
After swapping : d, a

Q.N.6.

A. Ans.

Interface : An interface is an empty shell, there are only the signatures of the methods, which implies that the methods do not have a body. The interface can't do anything. It's just a pattern.

Example:

```
class MotorVechile{
    public:
        virtual void run() const =0;
        virtual int getFuel() const=0;
}
```

Implementation : It is the actual substance behind the idea, the actual definition of how the interface will do what we except to it.

The implementation of previous interface is,

```
class Car : public MotorVechile {
    int fuel;
    public:
        void run() const override
        {
            printf ("Wroooom !n");
        }

        int getFuel() const override
        {
            return this -> fuel;
        }
};
```

Coupling and Cohesion:

- If any services provided by component is done within the component then it is cohesive.
- If the component cannot do all the work itself and interact with other components to do the work then it is coupling.
- It is desireable to reduce the amount of coupling as much as possible.

- Coupling is not good because of dependency in the components, if problem in one, the problem occurs in another too.

B. Ans.

CRC card is a class (component) responsibility collaborator card where the required class, responsibility and possible collaborators is mentioned so as, they will be very useful when the development of classes need to be divided between software engineers, since the cards can be physically handed over to them.

| Class: Library | |
|---|---|
| Responsibilities | Collaborator |
| knows all available lendables | |
| search lendable | Lendable. |

| Class: Librarian | |
|---|---|
| Responsibilities | Collaborator. |
| checks in lendable<br>search for lendable | Lendable<br>Library |

| Class: Borrower | |
|---|---|
| Responsibilities | Collaborator |
| Name<br>Roll<br>Address<br>Borrowed Items() | |

| class: Date | |
|---|---|
| Responsibilities | Collaborator |
| currentDate,<br>compareDate() | |

| Class: Lendable | |
|---|---|
| Responsibilities | Collaborator |
| checkOverDue()<br>checkout() | Date |

Q.N.7.

A. This pointer:

This pointer is a special type of pointer which is used to point to the data members and methods of current calling object. The 'this' pointer is an implict parameter to all member functions.

Friend functions do not have a 'this' pointer, because friends are not members of a class. Only member function can make use of 'this' pointer.

Example program:

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

class Person{
  private:
    int age;
  public:
    void setAge(int age){
      this->age = age;
    // this pointer refers to current calling object
    // data member age.
    }

    void getAge(){
      return age;
        // also we can directly use variable name
    }
};
```

```
int main()
{
    Person p;
    p.setAge(10);
    cout << " Given age : " << p.getAge();
    return 1;
}
```

C. Template Class :

- A class template is a kind of class which has members of template type.
- Class template lets us define the behavior of a class without actually knowing what data type will be handled by the operations of class.
- Template class provide the mechanism for creating applications with generic type which are common applications such as linked list, stacks, queues, etc.

General form of class template.

Step 1 : defining template

```
template <typename T>
```

step 2 : Defining class with members of template type.

```
class className
{
    class members with datatype T wherever appropriate
}
```

Step 3: Defining Object

class_name <specific data type for T> objectName;

Example program:

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

template<typename T>
class Rectangle{
    T length, breadth;
    public:
        Rectangle();
        Rectangle(T x, T y)
        {
            length=x;
            breadth=y;
        }
        T area();
};

template<typename T>
Rectangle<T>::Rectangle(){
        length=0;
        breadth=0;
}

template<typename T>
T Rectangle<T>::area(){
        return (length*breadth);
}
```

```
void main(){
    Rectangle<int> obj1(5,8 9);
    Rectangle<double> obj2(12.34, 56.78);
    cout << obj1.area()<<endl;
    cout <<obj2.area()<<endl;
    cout <<sizeof(obj1)<<endl<<sizeof(obj2);
   //getch();
}
```