2019 - Spring

1a

→ As a programming projects become larger, a interesting phenomenon was observed that: ₹a task that would take one programmer 2 months to perform could not be accomplished by two programmers working for one month.

This behaviour of complexity is called non-linear behaviour of complexity.

The reason for this non linear behaviour

i) The interconnections between software components were complicated

ii) A Large amounts of information had to be communicated among various members of the programming team.

Corventional techniques brings about high degree of interconnectedness. This means the dependence of one portion of code on another portion (coupling).

→ Abstraction mechanism is use to control complexity. It is the ability to encapsulate and isolate design and execute information.

Abstraction is a mechanism of displaying only essential information and hiding the details.

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

class Demo{
    int a, b;    // private member
public:
    void get(int a, int b)
    {
        this->a = a;
        this->b = b;
    }
    void display()
    {
        cout<<"sum ="<<a+b;
    }
};

main()
{
    Demo d;
    d.get(7, 11);
    d.display();
}
```

1b.

→

Structure	Class
i. A structure is a collection of variables of different data types under a single unit.	i. A class is a user defined blueprint or prototype from which objects are created.
ii. All members are public by default.	ii. All members are private by default.
iii. It doesn't support inheritance.	iii. It support inheritance.
iv. It's object is created by stack memory.	iv. It's object is created on the heap memory

There are three types of access specifiers in C++.
They are:

i) public
ii) protected
iii) private

2a

→ In some situation non-member function need to access the private data of class or one class wants to access private data of ~~second~~ another class. In this case, we use 'friend function.

Pros of friend function

i) It can access the private data member of class from outside the class.

ii) It allows sharing private members information.

iii) It ~~of~~ can access members without need of inheriting the class.

iv) We can invoke it like any normal function of the class.

Cons of friend function

i) It violates the laws of data hiding by allowing access to private members of class from outside the class.

ii) Runtime polymorphism ~~=~~ in member cannot be done.

iii) Size of memory occupied by objects will be maximum.

```
class B;
class A
{
  int a;
  public:
    void get()
    {
      cout<<"Enter a:";
      cin>>a;
    }

    void display()
    {
      cout<<"Value of a="<<a;
    }
friend void swap(A &ob1, B &ob2);
};

class B
{
  int b;
  public:
    void get()
    {
      cout<<"Enter b:";
      cin>>b;
    }
    void display()
    {
      cout<<"Value of b="<<b;
    }
```

```
    friend void swap(A &ob1, B &ob2);
};

void swap(A &ob1, B &ob2)
{
    int temp;
    temp = ob1.a;
    ob1.a = ob2.b;
    ob2.b = temp;
}

main()
{
    A ob1;
    B ob2;
    ob1.get();
    ob2.get();
    cout<<"Before Swapping:";
    ob1.display();
    ob2.display();
    swap(ob1, ob2);
    cout<<"After Swapping:";
    ob1.display();
    ob2.display();
}
```

b.

→ In C++ dynamic memory allocation can be achieved by using new and delete operator. new is used to allocate memory blocks at runtime dynamically. While, delete is used to de-allocate the memory which has been allocated dynamically.

Eg:

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

 int main()
{
  int i, n;
  int *ptr;
  int sum=0;
  cout<<" How many Enter the size of
                array:";
  cin>>n;
  ptr=new int[n];

 for(i=0; i<n; i++)
{
  cout<<" Enter number:"<<endl;
  cin>>ptr[i];
  sum = sum +ptr[i];
  }
```

```
    cout << "Sum = " << sum;
    delete[] ptr;
    return 0;
}
```

3a

→ Principle of Substitutability:
If we have two classes A and B such that class B is a 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.

The term subtype is used to refer to a subclass relationship in which the principle of substitutability is maintained to distinguish such forms from the general subclass relationship.

The pointer pointing objects are referred to as object pointer.

Class_name *object_pointer_name;
Eg: Student *ptr;

this pointer is a pointer that stores the address of current calling function.

A virtual function is a member function declared in base class with keyword virtual and uses a single pointer to base class pointer to refer to object of different class.

3b.

```
class Derived;
class Bases
    protected:
        int i, j;
    public:
      void get(int a, int b)
      {
          i = a;
          j = b;
      }
    friend Derived display(Derived d);
};

class Derived: public Bases
      public:
        Derived add_vector(Derived D1, Derived D2
      {
          Derived B3;
          B3.i = D1.i + D2.j;
```

```
    D3.j = D1.j + D2.j;
    return D3;
};

Derived display(Derived d)
{
    Derived d1;
    cout << d.i << "i +" << d.j << "j" << endl;
    return d;
}

main()
{
    Derived D1, D2, D3, D4;
    D1.get(3, 6);
    D2.get(7, 11);
    display(D1);
    display(D2);

    D4 = D3.add_vector(D1, D2);
    display(D4);
}
```

46.

Compile time Polymorphism
(function overloading)

```
class Area{
    float p,q,r,s;
    public:
        int calc(int a)
        {
            p=a;
            cout<<"Area of square="<<p*p<<endl;
        }
    int calc(float b, float c)
    {
        q=b;
        r=c;
        cout<<"Area of rectangle="<<q*r<<endl;
    }
    float ~~calc~~ calc(float d)
    {
        s=d;
        cout<<"Area of circle="<<3.14*s*s<<endl;
    }
};
main()
{
    Area A;
    A.calc(3);
    A.calc(6,11);
    ~~A.calc(7);~~ float c=4.5;
    A.calc(c);
}
```

Run time polymorphism
(function overiding)

```
class A {
  public:
    void display()
    {
      cout << "Base class" << endl;
    }
};
class B: public A {
 public:
    void display()
    {
      cout << "Derived Class";
    }
};
 int main()
 {
    B b;
  b.display(); // function overridden.
  }
```

5a

→ Type casting:

```
class Rectangles
    float x, y;
    public:
    Rectangle(float x1, float y1)
    {
        x = x1;
        y = y1;
    }
    void display()
    {
        cout<<"x = "<<x<<"y = "<<y<<endl;
    }
};

class Polar{
    float r, a;
    public:
    Polar(float ra, float an)
    {
        r = ra;
        a = an;
    }
```

```
void display()
{
cout<<"r="<<r<<"a="<<a<<endl;
}

operator Rectangle()
{
    float x2,y2;
    x2 = r * cos(a);
    y2 = r * sin(a);
    return Rectangle(x2,y2);
}
};

main()
{
    Polar P(15,45);
    P.display();
  Rectangle R= P;   // R = operator Rectangle()
    R.display();
}
```

5b.

```
class Complex {
    int i, j;
public:
    complex(int a, int b)
    {
        i = a;
        j = b;
    }

    void display()
    {
     cout << i << "i+" << j << "j" << endl;
    }
Complex
 void operator + (Complex c2)
 {    Complex c;
 c.i = i + c2.i;
 c.j = j + c2.j;
  return c;
}};

main()
{
    Complex c3;
    Complex c1(1, 2);
    Complex c2(4, 5);
```

```
C1.display();
C2.display();
C3=C1+C2;
C3.display();
}
```

6a

→ Template is a concept that enables us to define generic classes and functions.
A template is defined with a parameter that would be replaced by a specified data type at the time of actual use of class or function

```
template <class T>
class Demo
{
    T a, b, S;
public:
    void get(T x, T &y)
    {
        a = x;
        b = y;
    }
```

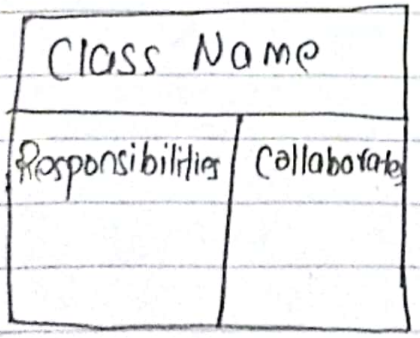
```
Void add()
{
    S = a+b;
    cout << S << endl;
}
};

int main()
{
    Demo <int> S1;
    Demo <float> S2;
    Demo <char> S3;
    S1.get(7, 11);
    S2.get(2.2, 3.5);
    S3.get(p, s);
    S1.add();
    S2.add();
    S3.add();
    return 0;
}
```

→ Exceptions are runtime anomalies or unusual condition that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds or running out of memory.

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

int main()
{
    int a, b, x;
   cout<< "Enter a and b:"<<endl;
   cin>>a>>b;
    x = a-b;

   try {
      if (x != 0)
      {
        cout<<" Result (a/x)="<< a/x <<endl;
      }
      else {
          throw(x);
      }
   }
catch(int i)
```

```
catch(int i)
{
cout << "Math Error! Can't divide by zero" << endl;
}

return 0;
}
```

→ CRC stands for Class Responsibility Collaborators. CRC cards are a brainstroming tool used in the design of object oriented software.
The cards are arranged to show the flow of messages among instances of each class.



Student CRC Card

Student	
Student number Name Address Phone Number Enroll in a seminal Request transcripts	Seminar Transcript

Seminar	
Name Fees Instructor	Student Professor

Professor	
Name Address Task	Seminar

Transcript	
Name Marks semester Grades	Student

7a) Message Passing

Message Passing means the dynamic process of asking an object to perform a specific action.

A message is always given to some object called reciver.

The action performed in response to the message is not fixed but may be differ, depending on the class of the receiver. That is different object may accept the same message ~~the~~ and yet perform different actions.

```
class Student {
int roll;
  public:
    void getdata(int x)
    {
      roll = x;
    }
    void display()
    {
      cout<< "Roll number =" << roll;
    }
};
```

```
int main()
{
    Student S;
    S.getdata(111); // object passing message
    S.display();

}
```

b) Inline function

→ An ~~infi~~ inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code.

A function is made inline by using 'inline' keyword before the function defination.

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

inline int max(int a, int b)
{
    return (a>b)? a:b; // if (a>b)→a
                       // else→b
}
```

```
int main()
{
  int x, y;
  cout << "Enter x and y." << endl;
  cin >> x >> y;
  cout << max() "Max = " << max(x, y);
}
```