2019-fall

1a

→ Data abstraction refers to providing only essential information and hiding their back ground details.
The main use of data abstraction mechanism are:

i) To manage complexity that arises due to interconnections of software components.
ii) To change the internal implementation without affecting the user level code.
iii) To avoid the code duplication. i-e programmer doesn't have to undergo the same task everytime to perform similar operation.
iv) To increase the readability of the code as it eliminates the possibility of displaying the complex working of the code.
v) To increase security of an application or program as only important details are provided to the users.

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

class Demo {
    private:
        int a, b;       // hidden

    public:
        void get(int x, int y)
        {
            a = x;
            b = y;
        }
        void sum()
        {
            cout << "Sum =" << a+b;
        }
};

int main()
{
    Demo d;
    d.get(3, 6);
    d.sum()
}
```

1b)

→ Different types of access specifiers:

i) private
ii) protected
iii) public

```
#include<iostream>
using namespace std;
class Customer{
    float p,r,si;
    int t;
  public:
    void set(float pr, int ti, float ra=8);
    void display();
};

void Customer::set(float pr, int ti, float ra)
{
    p=pr;
    t=ti;
    r=ra;
}

void Customer::display()
{
    si=(p*t*r)/100;
    cout<<"Simple Interest ="<<si<<endl;
}
```

```
int main()
{
    float p;
    int t,i;
     Customer C[3];
    for (i=0; i<3; i++)
{
   cout<<" Enter principle and time:" <<endl;
    cin>>p>>t;
     c[i].set(p,t);
     c[i].display();
}

}
```

2a.

→ Data hiding means protecting the data members of a class from unauthorized access from outside the class. It ensures exclusive data access to class members and protects object integrity by preventing unintended or intendend changes.

By declaring the data member private in class, the data can be hidden from outside the class. Those private data members cannot be accessed by object directly.

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

class Square {
    int num;
  public:
  void get()
  {
    cout<<"Enter the num:"<<endl;
     cin>>num;
  }
   void display()
  {
    cout<<"Square of a given num ="<<num*
                                    num<<endl;
  }
};

int main()
{
    Square S;
      S.get();
      S.display();
}
```

2b

→ Advantages of inheritance:

i) It supports reusability. The data members and member functions that are defined at parent class can be reused in base class. So there is no need to define the member again.

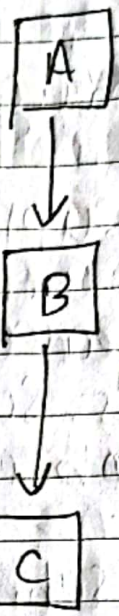
ii) Reusability enhanced reliability. The base class code will be already tested and debugged.

iii) It eliminates duplication of code.

iv) It reduces development and maintenance costs as well as saves time, effort and memory.

v) Software complexity can be easily managed

```
#include<iostream>
using namespace std;
class A {
    int a;
protected:
    int a= 7; int a;
public:
void getaC)
  {
    cout<<"Enter a:";
    cin>>a
  }
};

class B:public A{
    protected:
        int b;
    public:
      void getb();
      {
        cout<<"enter b:";
        cin>>b;
      }
};
```



```
class C : public B {
    protected: int c;
    public:
        void getc()
        {
            cout << "Enter c:";
            cin >> c;
        }

        void sum()
        {
            cout << "Sum = " << a+b+c;
        }
};

main()
{
    C obj;
    obj.geta();
    obj.getb();
    obj.getc();
    obj.sum();
}
```

Date
Page

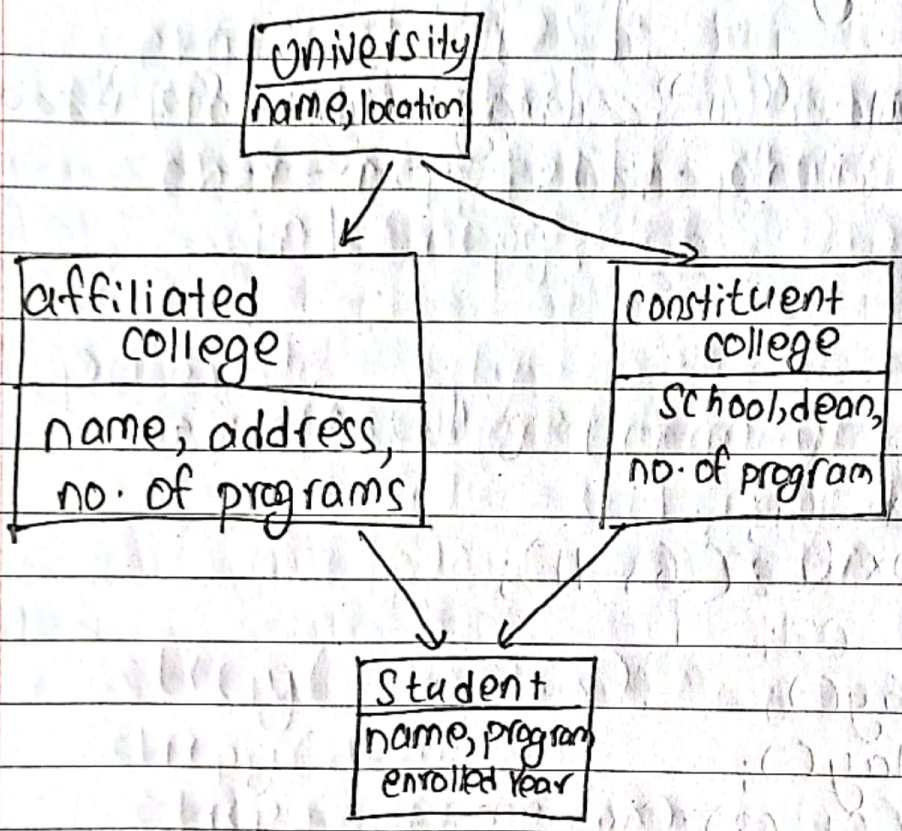
3a.

→ We can have more than one constructor in a same class with same name and different number of argument. In this way, we can overload constructors.

```
#include<iostream>
using namespace std;
class Complex {
    int real, imaginary;
public:
    Complex ()
    {
        real = 3;
        imaginary = 7;
    }

    Complex (int r, int i)
    {
        real = r;
        imaginary = i;
    }

    void add (Complex c1, Complex c2)
    {
        real = c1.real + c2.real;
        imaginary = c1.imaginary + c2.imaginary;
    }
```

```
void display()
{
  cout << real << " + " << imaginary << " i " << endl;
}
};

int main()
{
  Complex c1;
  Complex c2(6, 11);
  Complex c3;
  c1.display();
  c2.display();
  c3.add(c1, c2);
  c3.display();
}
```

4a.




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

class University {
    char uname[25];
    char location[20];

public:
    University( char u[], char l[])
    {
        strcpy(uname, u);
        strcpy(location, l);
    }
```

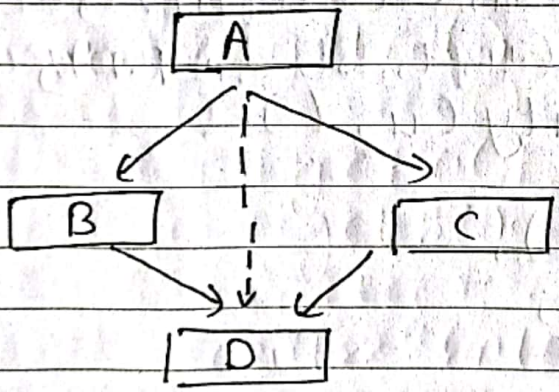
```
void disphy()
{
   cout<<"University name="<<uname<<endl;
   cout<<"University location="<<location<<endl;
  }
};

class affCollege: virtual public University {
   char cname[25];
   char address[25];
   int nop;

 public:
 affCollege(char u[], char l[], char c[],
            char a[], int n): University(u, l)
{
   strcpy(cname, c);
   strcpy(address, a);
    nop = n;
  }

 void display()
{
  cout<<"Affiliated College Name="<<cname<<endl;
  cout<<"Address="<<address<<endl;
   cout<<"Number of programs="<<nop<<endl;
 }
};
```

```
class consCollege : virtual public University {
    char school[25];
    char dean[25];
    ~~char nopp[25];~~ char pname[25];
    ~~int nopg;~~

public:
    ~~cons~~ consCollege(char u[], char l[], char
            s[], char d[], char pn[])
            : University(u, l)
    {
    strcpy(school, s);
    strcpy(dean, d);
    strcpy(pname, pn);
    }

    void display()
    {
    cout<<"School name = "<<school<<endl;
    cout<<"Dean name = "<<dean<<endl;
    cout<<"Name of program = "<<pname<<endl;
    }

};
```

```
class Student: public affcollege,
                public conscollege {

char sname[25];
char program[25];
int eyear;

public:

Student(char u[], char l[], char c[], char a[],
        int n, char s[], char d[], char pn[],
        char sn[], char p[], int y):

        affcollege(u, l, c, a, n),
        conscollege(u, l, s, d, pn),
        University(u, l)
{
strcpy(sname, sn);
strcpy(program, p);
eyear = y;
}

void display()
{
  cout << "Student name = " << sname << endl;
  cout << "Program = " << program << endl;
  cout << "Enrolled year = " << eyear << endl;
}
```

```
int main()
{
Student S("Pokhara", "Nepal", "Everest", "Kalitpur",
          "3", "LRI", "Engineer", "IT",
          "Yor", "II", 2021);

S.University::display();
S.affCollege::display();
S.consCollege::display();
S.display();

}
```

Ub.
→



The duplication of inherited members due to ~~these~~ multiple path can be avoided by making common base class as virtual class.

```
#include<iostream>
using namespace std;
class A{
  protected:
    int a;
  public:
  void geta();
    {
      cout<<"Enter a:";
        cin>>a;
    }
};
class B : virtual public A{
  protected:
    int b;
  public:
    void getb()
    {
      cout<<"Enter b:";
        cin>>b;
    }
};
```

```
class C: virtual public A{
    protected:
        int c;
    public:
        void getc()
        {
            cout<<"Enter c:";
            cin>>c;
        }
};

class D: public B, public C{
        int d;
    public:
    void getd()
        {
            geta();
            getb();
            getc();
        cout<<"Enter d:";
            cin>>d;
        }
        void sum()
        { cout<<"sum="<<a+b+c+d;
        }
};

main()
{
    D obj;
    obj.getd();
    obj.sum()
}
```

6b

→ this pointer stores the address of current calling object. this pointer can be used to refer it current class instance variable. this pointer is used as a pointer to the class object instance by member function.

```
class Sum {
    ~~{ pub~~ int a, b;
    public:
        Sum (a, b)
        {
        this -> a = a;
        this -> b = b;
        }
        void display()
        {
        cout << " Sum = " << a+b;
        }
};

main()
{       sum S(3,6);
        S.display();
}
```

Date
Page

The pure polymorphism is a technique used to define the same method with same arguments but different implementations.

```
#include<iostream>
using namespace std;
class Base{
    public:
    virtual void show() = 0;
};
class Derived: public Base{
    public:
    void show()
    {
    cout<<"Base Class"<<endl;
    }
};
class Created: public Base{
    public:
    void show()
    {
    cout<<"Created Class"<<endl;
    }
};
```

```
main ()
{
    Base * bptr[2];

    Derived d;
    Created c;
    bptr[0] = &d;
    bptr[1] = &c;
    bptr[0] -> show();
    bptr[1] -> show();
}
```

6a

→ Software Reusability is the practice of using existing code for a new software. But in order to use the code, that needs to be high quality. And that means, it should be safe, secure, reliable, efficient and maintainable.

In OOP, the concept of inheritance provide the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new call will have combined features of both clases. Reusability can be provided through the concept of composition also. In comporition one class contains the object of another classes to make it as a whole.

b. Software Component

→ A component is an identifiable part of a larger program or construction. Usually, a component provides a particular function or group of related functions. In programming design a system is divided into components that in turn are made up of modules. Component test means testing all related modules that form a component as a group to make sure they make together.

In oop, a component is a reusable program building blak that can be combined with other components to form an application. Examples of components to form an application. Each component is characterized by:

i) Behavior and State
ii) Instances and Clases
iii) Coupling and Cohesion
iv) Interface and Implementation Parno's principles

2020 - fall

1a

→ Features of object oriented Programming are:

i) Emphasis is on data rather than procedure
ii) Programs are divided into objects.
iii) Data structures are designed such that they characterize the objects.
iv) Functions that operate on the data of an object are tied together in the data structure.
v) Data is hidden and cannot be accessed by external function.
vi) Objects can communicate with each other through functions.
vii) New data and function can be easily added whenever necessary
viii) It follows button-up approach in program design

Date
Page

1b.

→ The concept of data hiding indicates that nonmember functions should not be able to access the private data of class. But, it is possible with the help of a friend function. This means, a function has to be declared as friend function to give it the authority to access the private data. Hence, friend function is not a member function of the class.

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

class Demo {
    int a, b;
  public:
    void get(int x, int y)
    {
        a = x;
        b = y;
    }
    friend void sum(Demo d);
};
    void sum(Demo d)
{
    cout<<" Sum = "<<d.a+d.b;
}
```

```
int main()
{
    Demo d;
    d.get(3,6);
    sum(d);
}
```

2b.

→ A constructor that accepts no arguments or parameters is called the default argument.

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

class Demo {
    int a, b;
  public:
    Demo()
    {
        a = 7;
        b = 9;
    }
    void sum()
    {
        cout<<" sum = "<<a+b;
    }
};
```

```
int main()
{
    Demo D1;
    D1.Sum();
}
```

→ The constructor that can take arguments are called parameterized constructor. Arguments are passed when objects are created.

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

public class Demo{
        int a, b;
        public:
            Demo(int x, int y)
            {
                a = x;
                b = y;
            }
        void sum()
        {
            cout<<" Sum = "<<a+b;
        }
};
```

```
int main()
{
    Demo D1(9, 6);
    D1.display(); D1.sum();
}
```

3a.

→ A copy constructor is used to declare and initialize an object with another object of same type.

We cannot pass object as an argument in copy constructor.

```
#include <iostream>
using namespace std;
class Demo {
    int a, b;
public:
    Demo() { }
    Demo(int x, int y)
    {
        a = x;
        b = y;
    }
    Demo(Demo &d)
    {
        a = d.a;
        b = d.b;
    }
```

```
void sum()
{
    cout<<"Sum = "<<a+b;
  }
};
int main()
{
  Demo D1(11,22);
  Demo D2(D1); // copy constructor called;
  D1.display();
  D2.disp D1.sum();
          D2.sum();
}
```

3b

→ Using the concept of inheritance the data member and member function of class can be reused. When once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of existing ones. This mechanism of deriving a new class from an old one is called inheritance.

Forms of inheritance

i) Sub classing for specialization
The child class is the special case of the parent class, i.e the child class is a subtype of parent class.

ii) Sub classing for specification
The parent class defines behaviour that is implemented in child class but not in parent class.

iii) Sub classing for construction
The child class make use of behaviour provided by the parent class but is not a subtype of parent class.

iv. Sub classing for generalization
The child class modifies or overrides some of the methods of the parent class.

v. Sub classing for extension
The child class adds new functionality to the parent class but does not change any inherited behaviour.

vi. Sub classing for limitation
The child class restricts the use of some of the behaviour inherited from parent class.

vii. Sub classing for varience
The child class and parent class are varients of each other and the class - subclass relationship is arbitrary

viii. Subclassing for combination
The child class inherits features from more than one parent class.

4b

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

class Demo{
    int a, b, num;
  public:
    void get()
    {
      cout<<"Enter the number: ";
       cin>>num;
    }
    Demo operator *(Demo);
};

Demo Demo::operator *(Demo d)
{
    Demo temp;
    temp.num= num * d.num;
    return temp;
}

int main()
{
      Demo d1,d2,d3;
      d1.get();
      d2.get();
      d3=d1*d2;
}
```

5b.

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

template<class T>
void swap(T &x, T &y)
{
    T temp;
    temp=x;
    x=y;
    y=temp;
}

int main()
{
    int i=7, j=11;
    float a=9, b=3;
    cout<<"Before swapping:";
    cout<<"i="<<i<<"j="<<j<<endl;
    cout<<"a="<<a<<"b="<<b<<endl;

    swap(i, j);
    swap(a, b);
    cout<<"After swapping:";
    cout<<"i="<<i<<"j="<<j<<endl;
    cout<<"a="<<a<<"b="<<b<<endl;
}
```

6a.

i) Interface and Implementation

It is possible for one programmer to know how to use a component developed by another programmer, without needing to know how the component is implemented. The purposeful omission of implementation details behind a simple interface is known as information hiding. The component encapsulates the behaviour, showing only how the component can be used, not the detailed actions it performs.

This naturally leads to two different views of software system. The interface view is seen by other programmers. It describes what a software component can perform. The implementation view is the face seen by the programmer working on a particular component. It describes how a component goes about completing a task.

ii) Coupling and Cohesion

Cohesion is the degree to which the responsibilities of single component form a meaningful unit. High cohesion is achieved by associating in a single component form a meaningful unit. High cohesion is achieved by associating in a single component tasks that are related in some manner. Probably the most frequent way in which tasks are related is through the necessity to access a common data value. This is the overriding theme that joins, for example, the various responsibilities of the Recipe component.

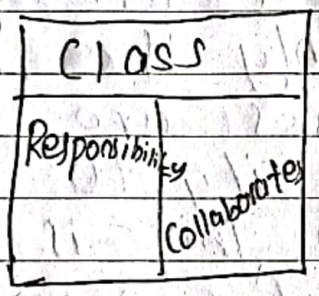
Coupling, on the otherhand, describes the relationship between software components. In general, it is desirable to reduce the amount of coupling as much as possible, since connections between software components inhibit ease of development, modification or reuse.

6b.

→ CRC cards are brainstroming tool used ~~to do~~ in design of object oriented software.

CRC cards help to identify and specify objects or components of an application in an informal way.

~~Class~~



Library	
knows all available lendables search for lendable	Lendable

Librarian	
check in Check out search	Lendable Borrower

Borrower	
Name Date Dues/Fines	Library

Date	
current date lend date return date	Borrower

Lendable	
title registration code date	