7) Follows bottom-up approach in program design.

Applications of OOP

1) client-server system
2) object oriented database
3) Real time system design
4) simulation and modeling system
5) Hypertent, Hyper media
6) Neural Networking and parallel computing
7) Desision support system
8) AI and expert system

Difference between structured and OOP

Structured (POP)	OOP
1. Large programs are divided into smaller self contained program segment known as function.	1. Programs are divided into entity called objects
2. Focuses on process/logical structure and then data req^{d} for that process.	2. Object oriented programming is designed which focuses on data.
3. Structured programming follows top-down approach.	3. OOP follows bottom-up approach.

4. Data and function donot tie with each other.	4. ~~OOP~~ Data & function are tied together.
5. structure programming is less secure as there is no way of data hiding.	5. OOP is more secure because of the data hiding.
6. structured programming can solve moderately complex program.	6. OOP can solve any complex program.
7. Provides less reusability and more functional dependency.	7. Provide less functional dependency and more reusability.
8. Data moves freely around the system from function to function.	8. Data is hidden and cannot be accessed by external functions.
9. E.g. C, FORTRAN, COBOL, Pascal, ALGOL.	9. E.g. C++, C#, Java

A/an key concepts on OOP:

Date
Page

1) Everything is an object.
2) computation is performed by object communicating with each other, requesting that other objects perform action. Object communicates by sending & receiving messages.
3) Each object has its own memory.
4) Every object is an instance of class, a class simply represents a group of similar object
5) A class is a repository for behaviour associated with an object.
6) Classes are organized into a singly rooted tree structure, called the inheritance hierarchy.

Coping with complexity

At early stage of computing, most programs were written in assembl language by a single individual. As program becomes more complex, programmer found out difficult to remember all the information needed to know in order to develop or debug their software. (which values were contained in what register).

This complexity introduced higher level programming languages such as C, C++, Java, Javascript, Python, Html, Css, etc.

1. Non-linear Behaviour of complexity

As programming project become larger, an interesting phenomenon was observed that: a task that would take one programmer 2 months to perform couldnot be done by two programmers working for one month.

- Message for an object is a request for execution of a procedure.
- Invoke the function in receiving object that generates the desired result

object.functionName (information).

Q) computation as simulation
Q) Difference between structure & class..

1) Explain computation as simulation.
2) In case of Object Oriented Programming, Explain how do we have the view that computation is simulation?

→ OOP framework is different from the traditional conventional behaviour of computer. Traditional model can be viewed as the computer is a data manager, following some pattern of instructions, wandering through memory, pulling values out of various memory transforming them in some manner, and pushing the results back into other memory.

The behaviour of a computer executing a program is a process-stat or pigeon-hole model. By examining the values in the slots, one can determine the state of the machine or the results produced by a computation. This model may be a more or less accurate picture of what takes place inside a computer. Real world problem solving is difficult in the traditional model.

In object oriented model we speak of objects, messages, and responsibility for some action. We never mention memory addresses, variables, assignments, or any of the conventional

programming terms. This model is process of creating a host of helpers that forms a community and assists the programmer in the solution of a problem.

The view of programming as creating a universe is in many ways similar to a style of computer simulation called "discrete event driven simulation". In a discrete event-driven simulation, the user creates computer models of the various elements of the simulation, describes how they will interact with one another, and sets them moving. This is almost identical to OOP in which user describes what various entities, object in program are, how will interact with one another and finally set them moving. Thus in OOP, we have view that computation is simulation.

DM ID Mosk

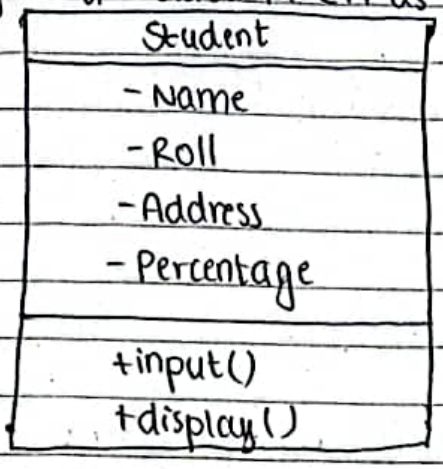
Difference between Structure & Class

Date
Page

	Aspect	Structure	Class
1.	Data Members	Public by Default	Private by default
2.	Member Functions	Can have, but not commonly used	Commonly used for encapsulating behaviour
3.	Inheritance	Doesnot support inheritance	Supports Inheritance
4.	Default Accessibility	Members are public by default.	Data members are private member functions are public by default.
5.	Memory Allocation	Typically allocated memory on stack	can be allocated memory on both stack & heap
6.	Usage	suitable for simple data structures.	suitable for complex systems with encapsulation inheritance & polymorphism
7.	Keyword	It is declared using the struct keyword.	It is declared using the class keyword.

8.	syntax	struct structureName
		{
		type stru-member1;
		type stru-member2;
		};
			class className
			{
			data-member;
			member-function;
			};

Q) WAP to define the class in C++ as shown in class diagram.



Defining the member function inside the class:

```
#include <iostream>
using namespace std;
class student
{
  private: char name[20];
           int roll;
           char address[20];
           float percentage;
public:
 void input()
  {
   cout << "Enter details of student" << endl;
   cin >> name >> roll >> address >> percentage;
  }
void display()
  {
   cout << "Name =" << name << "Roll =" << roll << endl;
cout << "Address =" << address << "Percent =" << percentage << endl
  }
```

Syntax of declaring object:
classname objectname;

Date
Page

```
int main()
{
  Student obj;        //creation of object
  obj.input();
  obj.display();
  return 0;
}
```

Defining the member function outside of class:

```
#include <iostream>
using namespace std;
class Student
{
  private:  char name[20], address[20];
            int roll;
            float percentage;
  public :  void input();
            void display();
};
void student :: input()
{
cout <<"Enter details of student"<<endl;
cin >> name >> roll >> address >>percentage;
}
void student :: display()
{
cout <<" Name ="<<name<<endl <<" Roll =" <<roll <<endl;
cout <<"Address ="<<address<<endl<<" Percent ="<<percentage <<endl;
}
```

```
int main()
{
    student obj;
    obj.input();
    obj.display();
    return 0;
}
```

#Accessing private member function:

Although it is normal practice to place all data items in private section and all the function in public, some situation may require certain function to be hidden (like private data) from outside call. We can place these function in the private section.

A private member function can only be called by another function that is i.e. member of its class.

Even an object cannot call private function using dot operator.

```
#include <iostream>
using namespace std;
class sample
{
  private: int m;
           void read();
  public: void update();
          void display();
};
```

```
void sample ::read()
 {
   cout <<"Enter the value of m"<<endl;
   cin >>m;
 }
void sample :: update()
 {
    read();
    m=m+5;
 }
void sample :: display()
 {
   cout <<" value after update = "<<m<<endl;
 }
int main()
 {
  sample s;
  s.update();
  s.display();
  return 0;
}
```

Object as function argument:

Like any other data type, an object can be passed as function argument. This can be done in two ways:

1. A copy of entire object is passed to the function (pass by value)
2. Only the address of the object is passed to the function (pass by reference)

Q) Program to perform addition of two time in hour and minute format by passing object as argument:

```
#include <iostream>
using namespace std;
class Time
{
private: int hours;
         int min;
public : void gettime(int h, int m)
         {
          hours = h;
          min = m;
         }
         void display()
         {
          cout << "hours=" << hours << "and min=" << min << endl;
         }
        void sum (Time t1, Time t2)
        {
          min = t1.min + t2.min;
          hours = min / 60;
```

```
        min = min % 60
        hours = hours + t1.hours + t2.hours;
    }
};
int main()
{
Time t1, t2, t3;
//cout << "Enter first time" << endl;
 t1.getTime(2,45);
//cout << "Enter second time" << endl;
 t2.getTime(3,20);
 t3.sum(t1, t2)
 t3.display();
 return 0;
}
```

Putting data by programmer.

Taking data from user

Or;

```
public:
 void getTime()
 {
  cout << "Enter time in hour" << endl;
  cin >> hour;
  cout << "enter time in min" << endl;
  cin >> min;
 }
```

Inside main:

```
Time t1, t2, t3;
cout << "Enter first time" << endl;
t1.getTime();
cout << "Enter 2nd time" << endl;
t2.getTime();
t3.sum(t1, t2);
t3.display();
return 0;
}
```

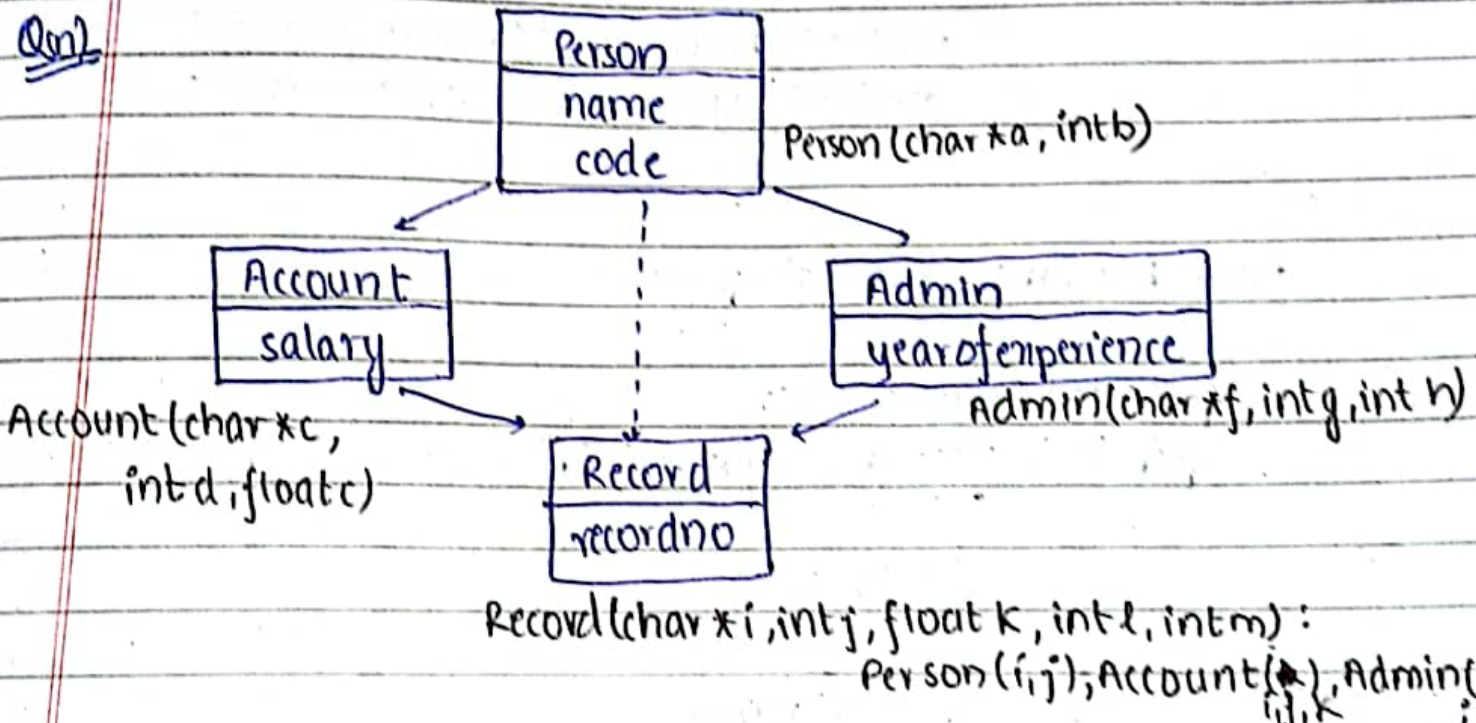
Date
Page

Q) WAP to add two complen numbers by passing objed as function argument.

```
#include <iostream>
 using namespace std;
  class Complen
  {
   private : int real;
             int img;
   public : void getData();
            void display();
            void addComplen(complex c1,complen c2);
   };                                  //void addComplen(complen, complen)
 void complen :: getData()
  {
   cout << "Enter real part" <<endl;
   cin >> real;
   cout << "Enter imaginary part" <<endl;
   cin >> img;
  }
 void complen :: display()
  {
   cout << Real << "+" << img << "i" <<endl;
  }
 void complen :: addComplen (complen c1, complen c2)
  {
   real = c1.real + c2.real;
   img = c1.img + c2.img;
  }
```

```
int main()
{
  Gamma ob(4,3,2);
  ob.showa();
  ob.showb();
  ob.showc();
  return 0;
}
```

Qn)



Sol^n

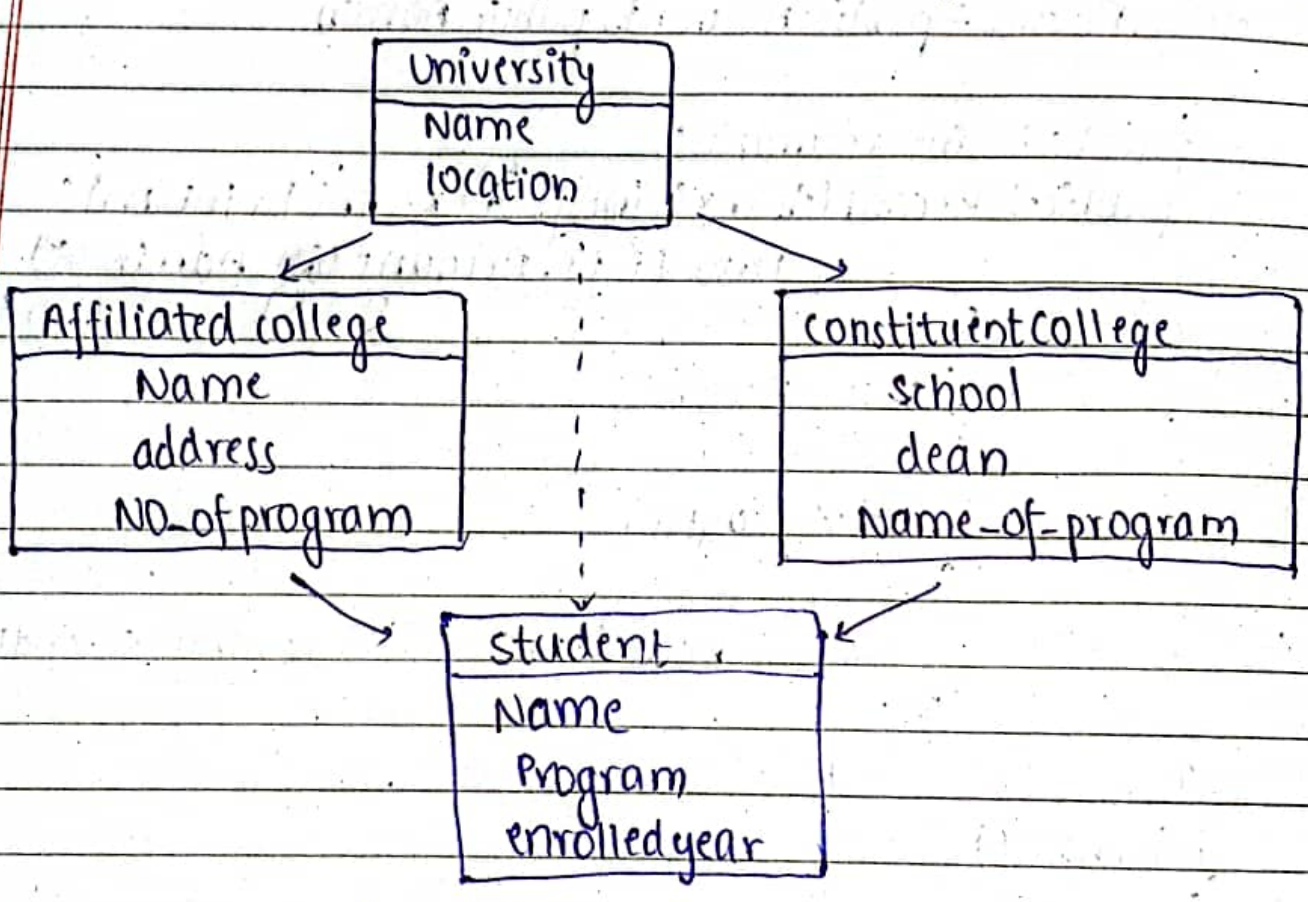
```
#include <iostream>
#usinginclude <string.h>
 using namespace std;

 class Person
  {
```

```
protected: char name[20];
           int code;
public: Person(char *a,int b)
        {
          strcpy(name,a);
          code = b;
        }
        void displayP()
        {
          cout << "Name = " << name << endl;
          cout << " code = " << code << endl;
        }
};
class Account : virtual public Person
{
  protected: float salary;
  public: Account(char *c, int d, float e) : Person(c,d)
          {
            salary = e;
          }
          void displayA()
          {
            cout << "salary = " << salary << endl;
          }
};
class Admin : virtual public Person
{
  protected: int yearofexperience;
```

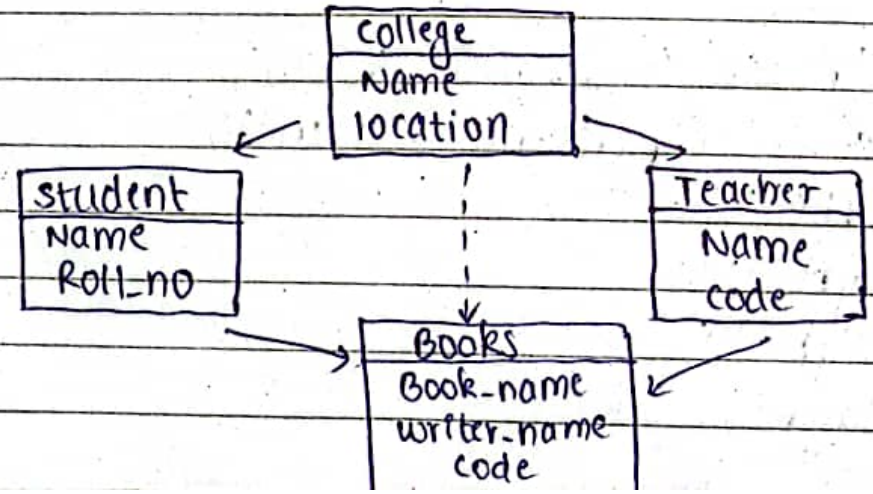
```
public: Admin(char *f, int g, int h): Person(f,g)
        {
          yearofexperience = h;
        }
        void displayAd()
        {
          cout << "Year of experience = " << yearofexperience << endl;
        }
};
class Record : public Account, public Admin
{
  private: int recordno;
  public: Record(char *i, int j, float k, int l, int m):
                Person(i,j), Account(i,j,k), Admin(i,j,l)
        {
          recordno = m;
        }
        void displayR()
        {
          cout << "Record No. = " << recordno << endl;
        }
};
int main()
{
  Record ob("Rudra", 45, 10000, 6, 23);
  cout << "Details of person are :" << endl;
  ob.displayP();
  ob.displayA();
  ob.displayAd();
  ob.displayR();
```

```
    return 0;
}
```

Q) The following figure shows the minimum information required for each class. Write a program by realising the necessary member function to display information of individual object. Every class should contain at least one constructor and should be inherited from other class as well



Q)



Subclass, subtype and principle of substitutability

If we consider the relationship of datatype associated with parent class to the data type associated with the child class, the following observation can be made:

- Instance of the child class must possess all data member associated with the parent class.
- Instance of the child class must implement, through inheritance as least (if not explicitly overridden), all functionality defined for parent class (they can also define new functionality)
- The instance of child class can mimic the behaviour of parent class and should be different from an instance of parent class if substituted in similar situation.

Principle of substitutability [Short note]

It states that if we have two class 'A' and 'B' such that, class B is subclass of A, it should be possible to substitute instance of class B for instance of class A in any situation, with no observable effect.

e.g.
```
#include <iostream>
    using namespace std;
    class A
    {
      public: void display()
              {
                cout<<"class A"<<endl;
              }
};
```

```
class B : public A
{
  public:
    void display()
    {
      cout << "class B" << endl;
    }
};
void test (A a)
{
  a.display();
}
int main()
{
  B b;
  test(b);      // b substituted for object of A
  return 0;
}
```

In above example, the class A object is replaced by object of class B without any error. This is achieved when a child class is inherited from base class publicly.

Date
Page

Imp

Containership (Composition)

When class contains object of another class as its member data, it is termed as containership. The class which contains the object is called container class. containership is also termed as class within class.

syntax:

```
class A
{
  //
};
class B
{
  A obj;
};
```

Here, class B contains the object of class A, so that B is the container class.

e.g.

```
#include <iostream>
using namespace std;
class Employee
{
  private: int eid;
           float salary;
  public: void getData()
          {
           cout <<"Enter id and salary of employee" <<endl;
           cin >> eid >> salary;
          }
```

```
        void display()
        {
            cout<<" Employee id = "<<eid<<endl;
            cout<<" salary : "<< salary<<endl;
        }
};
class Company
{
    private: char cname[20];
             char department[20];
             Employee e;
    public : void getData()
             {
               e.getData();
               cout<<" Enter company name & department"<<endl;
               cin>>cname >> department;
             }
             void display()
             {
               e.display();
               cout<<"company name = "<<name<<endl;
               cout<<"Department = "<< department<<endl;
             }
};
int main()
{
  Company ob;
  ob.getData();
  ob.display();
```

```
    return 0;
}
```

In above example, class Company contains the object of another class Employee. so, Company has a Employee.

COMPOSITION:

Is A rule/Has A rule (IS A/HAS A RELATIONSHIP)

One of the main advantage of object oriented programming is re-use of code. This can be achieved either by using inheritance (is a rule) or by object composition (has a rule).

The difference between is a relationship and has a relationship:

	Is a relationship	Has a relationship
1	If the first concept is specialized instance of second one, then there exist an "Is a" relationship" between them.	1. If the second concept is the component of first and the two are not in any sense the same thing, then there exist a "Has a" relationship between them.
2.	"Is a" relationship refers to inheritance.	2. "Has a" relationship refers to composition.
3.	E.g. Car is a Vehicle	3. E.g. car has a engine
4.	"Is a relationship" asserts the instance of subclass must be more specialized form of the super-class.	4. In Has a relationship, class contains object of another class as its member data to make it as a whole.

5. E.g.
class Vehicle
{
 	
};
class car : public Vehicle
{
 	
};
	5. E.g.
	class Engine
	{
	
	};
	class car
	{
	Engine e;
	};

Difference between Inheritance and composition

Inheritance	composition
1. In inheritance, derived class inherit attribute and method from their parent class.	1. In composition, a class contains object of another class as its member data.
2. It exhibits "is a relationship".	2. It exhibits "has a relationship".
3. E.g. car is a vehicle	3. E.g. car has a engine.

4. class Vehicle
{
//
};
class car : public vehicle
{
//
};
	4. class Engine
	{
	//
	};
	class car
	{
	Engine e;
	}

5. Inheritance leads tight coupling bet^{n} super class and sub class.	5. In composition, relationship bet^{n} class can be decoupled easily. So, it can be easily maintained in the future.
6. It is harder to maintain.	6. It is comparatively easier to maintain.
7. Inheritance supports polymorphism.	7. composition doesn't supports polymorphism.
8. (Example program) (single inheritance)	8. (Example program)

SOFTWARE REUSABILITY:

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

In OOP, the concept of inheritance provides 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 class will have combined feature of both the classes.

Reusability can be provided through the concept of composition

as well. In composition, one class contains object of another class.

Advantages of Reusability:

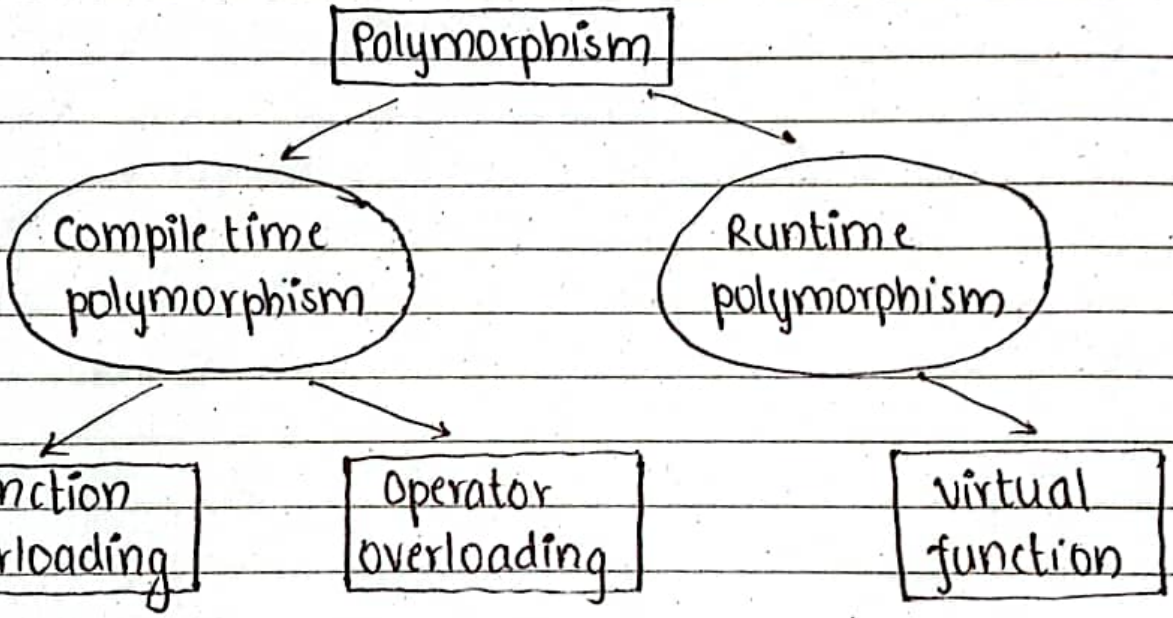
1. Reusability saves the programmer's time and effort.
2. Less development and maintenance cost.
3. Enhance reliability
4. Enhance extensibility
5.

Forms of Inheritance:

POLYMORPHISM

- Polymorphism means one name, multiple form.
- Greek word 'poly' means many and 'morphus' means form.
- c++ polymorphism means that a called-to member function will cause different function to be executed, depending on the type of object invoked by the function.

There are two types of polymorphism: Compile-time polymorphism
Run-time polymorphism



compile time polymorphism

It means that an object is bound to ~~fur~~ its function call at compile time.

- In this type of polymorphism, compiler is able to select the appropriate function for particular function call at compile time.
- This mechanism is also known as Early binding or

static binding or static linking.
- compile time polymorphism is achieved in two ways: Function overloading & operator overloading.

Run time Polymorphism
- The selection of appropriate function is done dynamically at run-time.
- Thus, it is not known which function will be invoked till an object actually makes function call during program execution
- This mechanism is also known as late-binding or dynamic binding.
- Run time polymorphism can be achieved with the help of virtual function.

Function Overloading
The method of using same function name but with different parameter list along with different data-type to perform variety of different task is known as function overloading.

The correct function to be invoked is determined by checking th number and type of argument but not function return type.

```
#include <iostream>
using namespace std;
class CalculateArea
{
    public: int area(int);
            int area(int, int);
            float area(float);
};
```

```
int CalculateArea :: area(int s)
{
  return(s*s);
}
int CalculateArea :: area(int l, int b)
{
  return(l*b);
}
int CalculateArea :: area(float r)
{
  return(3.14*r*r);
}
int main()
{
  CalculateArea ob;
  cout<<"Area of square ="<<ob.area(4)<<endl;
  cout<<"Area of rectangle ="<<ob.area(4,5)<<endl;
  cout<<"Area of circle ="<<ob.area(2.48)<<endl;
  return 0;
}
```

Operator Overloading:

The mechanism of adding special meaning to an operator is called operator overloading.

- It provides flexibility for the creation of new definition for most c++ operators.
- Using operator overloading, we can give additional meaning to normal c++ operations such as (+, -, =, <=, += etc.)

when they are applied to user-defined data type.

Things to be understand :

Usually, operation can be performed only on basic data type.

E.g.
```
int a,b,c;
c=a+b;
```
↳ Possible

```
class complen
{
  //
};
And, we created object as complen c1,c2,c3;
c3 = c1+c2;
```
↳ Not possible

(Because object is user-defined data type)

Operator overloading helps to define uses of operator for user defined data type i.e. objects. After overloading operands used with operators, are object instead of basic data type.

Operators that cannot be overloaded

All operators are not overloaded. The operator that are not overloaded are :

- ↳ class Member access operator (., .*)
- ↳ scope resolution operator (::)
- ↳ sizeof operator (sizeof)
- ↳ conditional operator (?:)

Date
Page

General form of operator function

```
return_type operator op(arglist)
{
    function body
}
```

or,

```
return_type classname :: operator op(arglist)
{
    function body
}
```

Here, return type is the type of value returned by specific operation. 'op' is the operator being overloaded. operator 'op' is the function name where operator is keyword.

Note:

1. Operator function must be either member function (non-static) or friend function.
2. Friend function will have only one argument for unary operator and two for binary operators.
3. Member function has no argument for unary operator and only one argument for binary operator.

Argument can be ~~either~~ passed either as value or by refernce.

* The process of overloading involves the following steps:

Date
Page

1. create a class that defines the data type that is used in the overloading function.
2. Declare the operator function operator op( ) in the public part of class. It may be either a member function or friend function.
3. Define operator function to implement the required operation.

	Member function	Friend function
For unary operator	op x or x op e.g. ++x or x++ x.operator op()	op x or x op e.g. ++x or x++ operator op(x)
For Binary operator	x op y e.g. x + y x.operator op(y)	x op y e.g. x + y operator op(x,y)

{'op' : operator being overloaded
x & y : object }

Overloading unary operator

Unary operator operates on single operand. Some of the unary operators are:

1. Increment operator (++)
2. Decrement operator (--)
3. Unary minus operator (-)

E.g. ++a
Here, a is only one operand.

Overloading unary minus operator

1) Using member function

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

class space
{
  private: int x,y,z;
  public:
        void getData(int a,int b, int c);
        void display();
        void operator-();
};
void space :: getData(int a, int b,int c)
{
    x = a;
    y = b;
    z = c;
}
void space :: display()
{
  cout<<"x = "<<x<<endl;
  cout<<"y = "<<y<<endl;
  cout<<"z = "<<z<<endl;
}
```

```
void space :: operator -()
{
    x = -x;
    y = -y;
    z = -z;
}
int main()
{
    space s;
    s.getData(5,10,15);
    cout << " s :" << endl;
    s.display;
    -s;
    cout << "-s:" << endl;
    s.display();
    return 0;
}
```

Output :

```
s:
x = 5
y = 10
z = 15
-s:
x = -5
y = -10
z = -15
```

2) Using friend function

```
#include <iostream>
using namespace std;
class space
{
   private : int x,y,z;
   public :
        friend void getData(int a, int b, int c);
               void display();
               friend void operator -(space s);
};
void space::getData(int a ,int b, int c)
{
    x=a;
    y=b;
    z=c;
}
void space::display()
{
   cout << "x = "<<x <<"y= "<< y <<endl << "z = "<<z <<endl;
}
void operator -(space s)
{
  s.x = -s.x;
  s.y = -s.y;
  s.z = -s.z;
}
```

```
int main()
{
    space s;
    s.getData(5,10,15);
    cout << " s : " << endl;
    s.display();
    -s;
    cout << " -s : " << endl;
    s.display();
    return 0;
}
```

Q) WAP to ~~ope~~ overload unary minus (-) operator so that statement s2 = -s1 can be returned when s1 and s2 are of type space that represent 3-dimensional co-ordinate system (using friend function)

```
#include <iostream>
using namespace std;
class space
{
  private : int x,y,z;
  public : void getData(int a, int b, int c);
           void display();
           friend space operator - (space &s);
};
void space :: getData(int a, int b, int c)
{
    x = a;
    y = b;  z = c;  }
```

(&s or s जहां pass करेंगे वहां space operator - ();

(Return type object use in main fn.

Using member fn → Blue ink.

Date
Page

```
void space :: display()
 {
    cout<<"x = "<<x<<endl<<"y = "<<y<<endl<<"z = "<<z<<endl;
 }
space operator -(space &s)
{
   space temp;
   temp.x = -s.x;
   temp.y = -s.y;
   temp.z = -s.z;
   return temp;
}
int main()
 {
  space s1,s2;
  s1.getData(5,10,15);
  cout << "s:"<<endl;
  s1.display();
  s2 = -s1;
  cout <<" -s:" <<endl;
  s2.display();
  return 0;
 }
```

```
space Space :: operator -()
temp.x = -x;
temp.y = -y;
temp.z = -z;
```

Q) WAP to overload unary ++ operator or overload prefix increm (++) operator by returning value through object.

Friend fn

Date
Page

```
#include <iostream>
using namespace std;
class counter
{
  private: int count;
  public: void getData(int n)
          {
            count = n;
          }
          void showData()
          {
            cout << " count = " << count << endl;
          }
          Counter operator ++();      friend Counter operator
};                                      ++(counter &c);
Counter counter :: operator ++()      ... :: operator++(counter &c)
{
  Counter temp;
  temp.count = ++count;          temp.count = ++c.count;
  return temp;
}
int main()
{
  counter c1, c2;
  c1.getData(3);
  c1.showData();
  c2 = ++c1;
  c1.showData();
  c2.showData();
  return 0;
}
```

O/p:
count = 3
count = 4
count = 4

Q) WAP to overload post fix increment operator returning value through object.

```
#include <iostream>
using namespace std;
class counter
{
  private: int count;
  public: void getData(int x)
          {
            count = x;
          }
          void showData()
          {
            cout << "count = " << count << endl;
          }
          counter operator ++();
};
counter counter :: operator ++()
{
   counter temp;
   temp.count = count ++;
   return temp;
}
int main() {
counter c1, c2;
c1.getData(3);
c1.showData();
c2 = c1++;
c1.showData();
c2.showData();
return 0; }
```

o/p:

count = 3
count = 4
count = 3

friend fn

Date
Page

without using return type:

```
#include <iostream>
using namespace std;
class counter
{
    private : int count;
    public : void getData (int n)
             {
               count =n;
             }
             void showData()
             {
               cout <<"count = " <<count <<endl;
             }
             void operator ++();      friend void operator++
                                              (counter &s);
};
void counter :: operator ++()          void operator ++ (counter &s)
{                                      {
  count = count ++;                      s.count =s.count ++;
}                                      }
int main()
{
  counter c1;
  c1.getData (3);
  c1.showData();
  c1++;
  c1 .showData();
  return 0;
}
```

Q) Write a program to generate Fibonacci series using operator overloading of (++) operator. Which type of overloading is it?

```
#include<iostream>
using namespace std;
class Fibo
{
  private: int a,b,c;
  public: fibo()
          {
            a=-1;
            b=1;
            c=a+b;
          }
         void display() {
          cout<<c<<endl;
         }
         void operator ++()
         {
          a=b;
          b=c;
          c=a+b;
         }
};
int main()
{
 int i,n;
 Fibo f;
 cout<<"Enter the number of terms"<<endl;
```

```
cin>>n;
for (i=0;i<n;i++)
 {
   f.display();
   ++f;
 }
return 0;
}
```

This is an example of unary operator overloading.

Binary Operator overloading

The operator which operates on two operands is known as binary operators.

For example, c=a+b, where 'a' and 'b' are two operands.

V.I.V.Imp.

Q) Write a program to add two complex number using binary operator overloading. [PU :2013]

```
#include<iostream>
using namespace std;
class complex()
{
  private : int real,img;
  public : complex()
           {

           }
```

} केहे लेख्नेन भने initialize zero (default constructor)

```
complex(int r, int i)
{
    real = r;
    img = i;
}
void display(){
    cout << real << " + i" << img << endl;
}
complex operator + (complex);
};
complex complex :: operator + (complex c2)
{
 complex temp;
 temp.real = real + c2.real;
 temp.img = img + c2.img;
 return temp;
}
int main()
{
 complex c1(2,3);
 complex c2(2,4);
 complex c3;
 c3 = c1 + c2;
 cout << "c1 = ";
 c1.display();
 cout << "c2 = ";
 c2.display();
 cout << "c3 = ";
 c3.display();
 return 0;
}
```

Date
Page

Using friend function:

```
#include<iostream>
using namespace std;
class complen()
{
  private: int real, img;
  public: complen()
          {
          }
          complen(int r, int i){
           real = r;
            img = i;
          }
          void display() {
           cout << real << " +i" << img << endl;
          }
          friend complen operator+(complen, complen);
};
complen operator+(complen c1, complen c2)
{
  complen temp;
   temp.real = c1.real + c2.real;
   temp.img = c1.img + c2.img;
    return temp;
};
int main()
 {
   complen c1(2,3);
   complen c2(2,4);
   complen c3;
```

```
c3 = c1+c2;
cout << "c1, c2 & c3 respectively are ";
c1.display();
c2.display();
c3.display();
return 0;
}
```

Qsn) WAP to overload arithmetic operator (+, -, *, /).

```
#include <iostream>
using namespace std;
class Arithmetic
{
  private : float num;
  public : void getdata()
           {
            cout << "Enter number" << endl;
            cin >> num;
           }
           void putdata()
           {
             cout << num << endl;
           }
          friend Arithmetic operator +(Arithmetic a, Arithmeti
          friend Arithmetic operator -(Arithmetic a, Arithmetic b)
        friend Arithmetic operator /(Arithmetic a, Arithmetic b
        friend Arithmetic operator *(Arithmetic a, Arithmetic b)
};
```

Date
Page

```
friend Arithmetic operator + (Arithmetic a, Arithmetic b)
{
  Arithmetic temp;
  temp.num = a.num + b.num;
  return temp;
}
friend Arithmetic operator - (Arithmetic a, Arithmetic b)
{
   Arithmetic temp;
   temp.num = a.num + b.num;
   return temp;
}
friend Arithmetic operator * (Arithmetic a, Arithmetic b)
{
   Arithmetic temp;
   temp.num = a.num + b.num;
   return temp;
}
friend Arithmetic operator / (Arithmetic a, Arithmetic b)
 {
    Arithmetic temp;
    temp.num = a.num + b.num;
    return temp;
 }
int main();
 {
   Arithmetic a,b,c;
    a.getdata();
    b.getdata();
     c = a+b;
```

```
cout <<"Addition of two objects = " <<endl;
c.putdata();
cout <<" substraction of two objects = "<<endl;
c = a - b;
cout <<"Division of two objects = "<<endl;
c = a/b;
c.putdata();
return 0;
}
```

Q) WAP to overload two binary operator '+' and '-' so that statement c3 = c1+c2;
c3 = c1 - c2; exists.

(OR)

WAP to find the sum & difference of any two complex number by overloading '+' & '-' operator.

```
#include <iostream>
using namespace std;
class complex
{
private : int real, img;
public : complex()
{
}
complex ( int a, int b)
{
real = a;
img = b;
}
```

```
void display()
{
    cout << "real << "+i"<<img<<endl;
}
complex complex::operator + (complex c2)
{
    complex temp;
    temp.real = real + c2.real;
    temp.img = img+c2.img;
    return temp;
}
complex complex::operator - (complex c2)
{
    complex temp;
    temp.real = real - c2.real;
    temp.img = img - c2.img;
    return temp;
}
int main()
{
    complex c1(2,3);
    complex c2(2,4);
    complex c3;
    c3 = c1 + c2;
    cout << "Addition of two objects =" <<endl;
    c3.display();
    c3 = c1 - c2;
    c3.disp();
    return 0;
}
```

responsibilities → function

(PU 2013)

E.Q) Write a class Time with three integer attributes hour, minute and second. Include the following responsibilities in class:
Default and parameterized constructor
Display method to display time in hour, minute and second format. Appropriate function overload to realize addition of two time objects with '+' operator.

```
#include <iostream>
using namespace std;
class Time
{
  private : int hr, min, sec;
  public : Time()
           {
             hr = 0;
             min = 0;
             sec = 0;
           }
           Time (int h, int m, int s) {
             hr = h;
             min = m;
             sec = s;
           }
           void display() {
           cout << hr << ":" << min << ":" << sec << endl;
           }
          friend Time operator + (Time, Time);
};
```

```
Time operator+ (Time t1, Time t2) {
 Time temp;
 temp.sec = t1.sec + t2.sec;
 temp.min = temp.sec /60;
 temp.sec = temp.sec %. 60;
 temp.min = temp.min + t1.min + t2.min;
 temp.hr = temp.min /60;
 temp.min = temp.min %. 60;
 temp.hr = temp.temp.hr + t1.hr + t2.hr;
 return temp;
}
int main()
 {
 Time t1(4,5,6);
 Time t2 (5,25,4);
 Time t3;
 t3 = t1+t2;
 cout << " First time ="<<endl;
 t1.display();
 cout << "second time ="<<endl;
 t2.display();
 cout <<"sum = "<<endl;
 t3.display();
 return 0;
}
```

Q) WAP to overload "+=" operator to add distance of two objects (Height)

```
#include <iostream>
using namespace std;
class Height
{
private: int feet, inch;
public: Height()
        {
          feet = 0;
          inch = 0;
        }
        Height (int f, int i)
        {
          feet = f;
          inch = i;
        }
        void display()
        {
        cout << feet << "feet" << inch << "inch" << endl;
        }
        void operator += (Height h);
};

void Height :: operator += (Height h)
{
  feet += h.feet;
  inch += h.inch;
  if (inch >= 12)
  {
```

```
        inch -= 12;
        feet++;
    }
}
int main()
{
    Height h1(5,9);
    Height h2(10,5);
    cout << "First height :" <<endl;
    h1.display();
    cout << "Second height :" <<endl;
    h2.display();
    h1+=h2;
    cout << "After addition h1 :" <<endl;
    h1.display();
    return 0;
}
```

V.V.V.Imp.

Q) WAP to concatenate two strings by overloading '+' operator.

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

class StringC
{
  private: char str[50];
  public:
```

```
stringc()
{
}
stringc(char s[])          or, //(char *s)
{
  strcpy(str,s);
}
void display()
{
  cout<<"string is :"<<str<<endl;
}
stringc operator +(stringc s2)
{
  stringc s3;
  strcat (str, s2.str);
  strcpy (s3.str, str);
  return s3;
}
};
int main()
{
  stringc s1("sanjog");
  stringc s2("Gautam");
  stringc s3;
  s1.display();
  s2.display();
  s3 =s1 +s2;
  s3.display();
  return 0;
}
```

Q) write a program to overload == operator to compare two time.

```
#include<iostream>
using namespace std;
class Time
{ private : int hr, min, sec;
  public : Time()
           {
           }
           Time(int h, int m, int s)
           {
             hr = h ; min = m ; sec = s;
           }
           friend int operator == (Time t1, Time t2);
};
int operator == (Time t1, Time t2)
{
  if (t1.hr == t2.hr && t1.min == t2.min && t1.sec = t2.sec)
  {
    return (1);
  }
  else
  { return 0;
  }
}
int main()
{
  Time t1 (3, 15, 45);
  Time t2 (4, 15, 45);
```

```
if (t1 == t2)
{
 cout << "Both the time values are equal";
}
 else
 {
   cout << "Both the time values aren't equal";
 }
return 0;
}
```

Q) WAP to overload '>' operator.

```
#include <iostream>
 using namespace std;
 class Maximum
 {
  private: int n;
  public: Maximum (int a)
          {
           n = a;
          }
          void operator > (Maximum m2)
          {
           if (n > m2.n) {
             cout << "Maximum number is " << n;
           }
           else {
            cout << "Maximum number is " << m2.n;
           }
};
```

```
int main()
{
  Maximum m1(5);
  Maximum m2(10);
  m1 > m2;
  return 0;
}
```

TYPE CONVERSION:

- Type conversion is converting one type of data to another type.
- Compiler automatically converts basic to another basic data type (for e.g. int to float, float to int etc.) by applying type conversion rule provided by the compiler.
- The type of data to the right of the assignment operator (=) is automatically converted to the type of variable on the left.

For example:
The statements:

```
int m;
float n = 3.1415;
m = n;
```

Value stored in m is 3 because, here the fractional part is truncated
compiler does not support automatic type conversion for user-defined data type. Therefore, we must design conversion routines for the type of conversion of user-defined data type.

There are three possible type conversions:

1. Conversion from basic type to class type
2. Conversion from class type to basic type
3. Conversion from one class type to another class type

Conversion from basic to class type (Basic to user-defined type)

The constructor is used for the type conversion takes the single argument which type is to be converted.

Example 1:

```
#include<iostream>
using namespace std;
class Time
{
    int hours, minutes;
    public:
    Time(int t)
    {
        hours = t/60;
        minutes = t%60;
    }
    void display()
    {
    cout<<"Hours = "<<hours<<endl;
    cout<<"Minutes = "<<minutes<<endl;
    }
};
```

Not specified → private

```
int main()
{
    int duration = 65;
    Time t1 = duration;
    t1.display();
    return 0;
}
```

← Time t1 (duration) जस्तो लाग्छ But Assignment operator in type conversion.

After conversion, the hours members of t1 contains the value 1 and minutes member contains the value 5, denoting 1 hour and 5 minutes.

Example 2:

Program to convert meter to feet and inches using (Basic to class type conversion)

```
#include <iostream>
using namespace std;
class Dist
{
  private : int feet;
            float inches;
  public : Dist()
           {
           }
           Dist(float m)
           {
            float f = 3.28083 * m;
            feet = int(f);
            inches = 12 * (f - feet);
           }
```

```
    void display()
    {
     cout << feet << "feet" << inches << "inches" << endl;
    }
};
int main()
{
 float meter = 12.5 ;
  Dist d1;
  d1 = meter;
  d1.display();
  return 0;
}
```

V.V.V.Imp

Make a class called memory with member data to represent bytes, kilobytes, and megabytes. In your program, you should be able to use statements like m1 = 1087665; where m1 is an object of class memory and 1087665 is an integer representing some arbitrary number of bytes. Your program should display memory in a standard format like 1 megabyte 38 kilobytes 177 bytes.

```
#include <iostream>
using namespace std;
class Memory
{
  private: int mb;
           int kb;
           int byte;
```

```
public: Memory()
        {
        }
        Memory(long int m)
        {
          int rem;
          mb = m/(1024*1024);
          rem = m%(1024*1024);
          kb = rem/1024;
          byte = rem%1024;
        }
        void display()
        {
          cout << mb << "Megabytes" << endl;
          cout << kb << "Kilobytes" << endl;
          cout << byte << "Bytes" << endl;
        }
};
int main()
{
  Memory m1;
  long int m = 1087665;
  m1 = m;
  m1.display();
  return 0;
}
```

2. # CLASS TO BASIC TYPE:

Date
Page

C++ allows us to define an overloaded casting operator that could be used to convert a class data type to a basic type. The general form of an overloaded casting operator function, usually referred to as a conversion function is:

```
operator typename()
{
  //function statements
}
```

The function converts a class type to typename. For example, the operator int() converts an class object to type int, operator float() converts the class type object to float and so on.

The casting operator function should satisfy the following conditions:

→ It must be a class member.
→ It must specific return type.
→ It must not have any arguments.

Example:

```
#include <iostream>
  using namespace std;
   class Item
   {
     float price;
     int quantity;
     public: item (float p, int q) {
                price = p;
              quantity = q;
```

```
void display()
{
  cout << "Price of items = " << price << endl;
  cout << "Quantity of items = " << quantity << endl;
}
operator float()
{
  return (price * quantity);
}
};
int main()
{
  Item I1 (255.5, 10);
  float total;
  I1.display();
  total = I1;
  cout << "Total amount = " << total << endl;
  return 0;
}
```

Q) WAP to convert feet and inches into meter (class to basic type conversion)

```
#include <iostream>
using namespace std;
class Distance
{
  int feet;
  float inches;
  public: Distance (int f, float i)
          { feet = f;
            inches = i;
```

```
void display()
{
  cout << feet << "feet" << inches << "inches" << endl;
}
operator float()
{
  float f = inches/12;
  f = f + feet;
  return (f/3.28083);
}
};
int main()
{
  Distance d1(5,3.6);
  float m = d1;
  cout << "Distance in feet and inches:" << endl;
  d1.display();
  cout << "Distance in meter =" << m << endl;
  return 0;
}
```

ONE CLASS TYPE TO ANOTHER CLASS

We can convert one class type data to another class type:

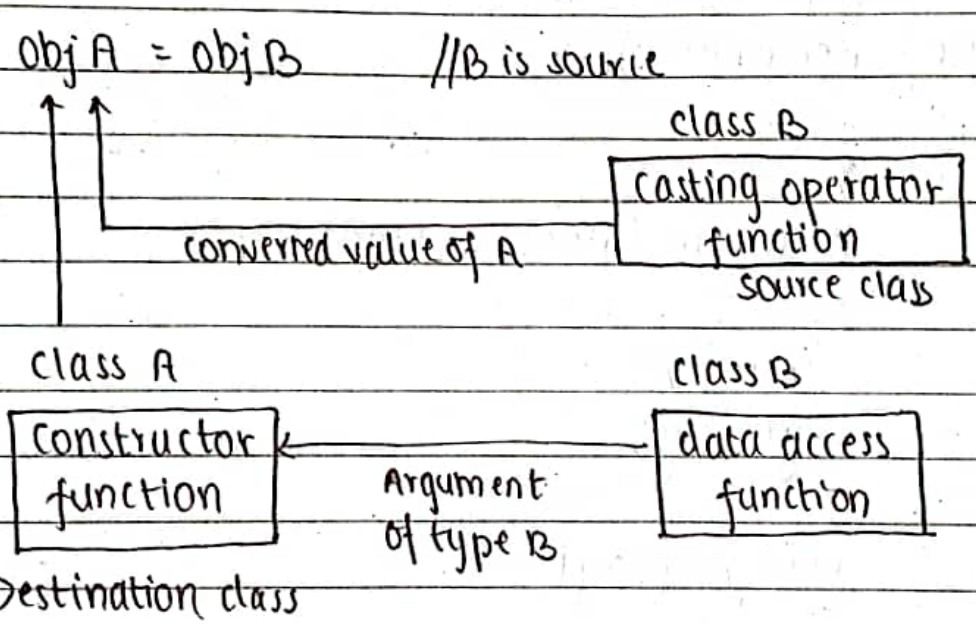
Example:

```
objX = objY;        //objects of different class
```

objX is an object of class X and objY is an object of class Y. The class Y type data is converted to the class X type data and the converted value is assigned to the objX. since the conversion takes place from class Y to class X, Y is known as the source class and X is known as the destination class.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function.

Note : conversion from a class to any other type (or any other class) should make use of casting operator function in source class. On the other hand, to perform the conversion from any other type / class type, constructor should be used in destination class.



rectangle = (polar) source → conversion function (routine)

Date

(2) Conversion routine in source class
(Conversion function in source class) :-

(2) Conversion from polar to rectangle using conversion routine in polar.

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

class Rectangle
{
 private: float xco;
          float yco;
 public: Rectangle()
         {
         }                          // xco = 0.0;
                                    // yco = 0.0;
         Rectangle (float x, float y)
         {
          xco = x;
          yco = y;
         }
         void display()
         {
          cout << "(" << xco << "," << yco << ")" << endl;
         }
};
```

```
class Polar
{
private: float radius;
         float angle;
public:  Polar()
         {
          radius = 0.0;
          angle = 0.0;
         }
         Polar(float r, float a)
         {
          radius = r;
          angle = a;
         }
         void display()
         {
          cout << "(" << radius << "," << angle << ")" << endl;
         }
         operator Rectangle()
         {
          float x = radius * cos(angle);
          float y = radius * sin(angle);
          return Rectangle(x,y);
         }
};
int main() {
 Polar p(10.0, 0.758539);
 Rectangle r;
  r = p;
 (out << "Polar coordinates :";
```

polar ← rectangle
source
conversion fn.

Date
Page

```
    p.display();
    cout << "Rectangular coordinates = ";
    r.display();
    return 0;
}
```

#. Conversion from Rectangle to Polar (using conversion routine in rectangle):

```
#include <iostream>
#include <math.h>
  using namespace std;
  class Polar
  {
    private : float radius;
              float angle;
    public : Polar()
             {
             }
             Polar(float r, float a)
             {
               radius = r;
               angle = a;
             }
             void display()
             {
               cout<<"("<< radius <<","<<angle<<")"<<endl
             }
};
```

```
class Rectangle
{
 private : float xco;
           float yco;
 public : Rectangle ()
          {
          }
          Rectangle (float x, float y)
          {
            xco = x;
            yco = y;
          }
          void display ()
          {
           cout<<"("<<xco<<","<<yco<<")"<<endl;
          }
          operator Polar()
          {
           float a = atan (yco/xco);
           float r = sqrt (xco*xco+yco*yco);
           return Polar (r,a);
          }
};
int main()
{
 Rectangle r (7.071, 7.071);
 Polar p;
 p = r;
 cout <<"Rectangular coordinates =";
 r.display();
```

```
    cout <<"Polar coordinates = ";
    p.display();
    return 0;
}
```

POINTER TO OBJECT POINTER:

The pointer pointing to objects is referred to as an object pointer.

Declaration:

```
class_name *object_pointer_name;
E.g. student *ptr;
```

Here, ptr is an object pointer of the student class type that has been declared, where the student is already defined class.

Initialization:

```
object_pointer_name = &object.
E.g. ptr = &st;
```

```
E.g. student st;
     student *ptr;
     ptr = &st;
```

Here, ptr is an object pointer of student class type and st is an object of the class student.

Note: when accessing members of a class using an object pointer, the arrow operator (->) is used instead to dot operator.

Example:

```
#include<iostream>
using namespace std;
class Student
{
  private : char name[20];
            int roll;
  public : void getData()
           {
            cout<<"Enter name & roll"<<endl;
            cin>>name>>roll;
           }
            void display()
           {
            cout<<"Name ="<<name<<endl<<"Roll No ="<<roll;
           }
};
int main()
 {
    Student ob;
    Student *ptr;
    ptr = &ob;
    ptr -> getData();
    ptr -> display();
    return 0;
}
```

Date
Page

Creating objects at runtime using object pointers

Objects pointer are also used to create the object at runtime by using it with new operator as follows:

E.g. item *ptr = new item;

This statement allocates enough memory space for the object and assigns the address of the memory space to ptr.
We can also create an array of objects using pointers. For e.g.: the statement, item *ptr = new item[10];
(Creates memory for an array of 10 objects of item)

Example:

```
#include<iostream>
using namespace std;
class Item
{
  private: int code;
           float price;
  public : void getData(int c, float p)
           {
             code = c;
             price = p;
           }
           void display()
           {
             cout<<"code = "<<code<<endl;
             cout<<"Price = "<<price<<endl;
           }
};
int main()
{
```

```
int n,i,x;
float y;
cout<<" Enter the number of item"<<endl;
cin>>n;
Item *ptr = new item[n];
Item *d = ptr;
for(i=0; i<n; i++)
{
  cout << "Input code and price of item"<<endl;
  cin>>x>>y;
  ptr->getData(x,y);
  ptr++;
}
for(i=0; i<n; i++)
{
  cout<<"Item :"<<i+1<<endl;
  d->display();
  d++;
}
return 0;
}
```

Pointer to derived class :

Qsn) Can you derive a pointer from a base class ? Explain with a suitable example.

→ Yes, we can derive a pointer from a base class. Pointers to object of base class are type compatible with pointer to objects of the derived class. Therefore, a simple pointer variable can be made to point to objects belonging to different classes.

For example, if B is a base class & D is the derived class from B, then a pointer declared as a pointer to B can also be a pointer to D.

```
B *bptr;        //pointer to class B type variable
B b;            //Base object
D d;            //Derived object
bptr = &b;      //bptr points to object b
                //we can make bptr to point to the object d as follows
bptr = &d;      //bptr points to object d
```

This is perfectly valid with c++ because d is an object derived from class B.

Note:
Although c++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We can access only those members which are inherited from B and not the members that originally belong to D. We may have to use another pointer declared as pointer to derived type.

Example:

```
#include <iostream>
  using namespace std;
  class B
  {
   public: void display() {
             cout<<"Base class"<<endl;
           }
```

```
class D : public B
{
  public : void display(){
            cout << " Derived class" <<endl;
           }
};
int main()
{
   B b1;
   B *bptr;          //Base pointer
   bptr = &b1;       //Base address
   cout << "bptr points to base object "<<endl;
   bptr -> display();
   D d1;
   bptr = &d1;   // Address of the derived object
   cout << "bptr now points to derived object "<<endl;
   bptr -> display();  //Access to base class member
   D *dptr;
   dptr = &d1;
   cout << "dptr is a derived type pointer" <<endl;
   dptr -> display();
   cout << "Using ((*D)bptr)" <<endl;
   ((D*)bptr) -> display();
   return 0;
}
```

short note

Date:
Page:

THIS POINTER:

- This pointer stores the address of current calling function.
- 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 pointer is automatically passed to a member function when it is called.
- This pointer this acts as an implicit argument to all member function.
- This pointers are not accessible for static member functions.
- This pointers are not modifiable.

1. This pointer can be used to refer current class instance variable.

Example:

```
#include <iostream>
using namespace std;
class Employee
{
    private : int eid;
              float salary;
    public : Employee (int eid, float salary)
             {
                 this->eid = eid;
                 this->salary = salary;
             }
```

Date :
Page :

```
    void display()
    {
      cout << "Employee ID =" << eid << endl;
      cout << "salary =" << salary << endl;
    }
};
int main()
{
  Employee e1(101, 25452.55);
  Employee e2(102, 54485.25);
  e1.display();
  e2.display();
  return 0;
}
```

2. One important application of the pointer this is to return the objects it point to.
For example: the statement

```
return *this;
```

inside a member function will return the object that invoked the function.

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

Date :
Page :

```
private : char name[20];
          float age;
public:  Person()
         {
         }
         Person (char *n , float a)
          {
            strcpy(name, n);
            age = a;
          } Person greater (Person n){
          if (n.age >= age) {
          return n;
          }
          else {
          return *this;
          } }
          void display()
          {
           cout << "Name = " << name << endl;
           cout << "Age = " << age << endl;
          }
};
int main() {
 Person p1 ("Ram", 52);
 Person p2 ("Hari", 24);
 Person p3;
 p3 = p1.greater (p2);
 cout << "Enter Elder person is" << endl;
```

Date :
Page :

```
p3.display();
return 0;
}
```

Virtual Function (Run-time polymorphism)

- Virtual means existing in appearance but not in reality.
- Virtual function is declared by using a keyword 'virtual' preceeding the normal declaration of a function.
- When a function is made virtual, c++ determines which function to use at run time based on the type of object pointed by the base pointer rather than the type of pointer.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. so, we create the pointer to the base class that refers to all the derived objects.
- When base class pointer contains the address of the derived class object, always executes the base class function. Here, the compiler simply ignores the contents of the (base) pointer and chooses the member function that matches the type of the pointer.
- This issue can only be resolved using the 'virtual' functi When the function is made virtual, c++ determines which function is to be invoked at the runtime based on the type of object pointed by the base class pointer.

Date :
Page :

In this way, runtime polymorphism can also be achieved.

Example:

```
#include <iostream>
using namespace std;
class B
{
  public: virtual void show()
          {
            cout << "show base" <<endl;
          }
};
class D : public B
{
  public: void show()
          {
            cout <<"show derived" <<endl;
          }
};
int main()
{
   B b1;
   D d1;
   B *bptr;
   cout << "bptr points to base" <<endl;
   bptr = &b1;
   bptr ->show(); //calls base class function
   cout <<"bptr points to derived" <<endl;
```

Date :
Page :

```
    bptr = &d1;
    bptr->show(); //calls derived class function
    return 0;
}
```

Rules of virtual function

- The virtual functions must be members of some class.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions, with the same name but different prototypes, C++ will consider them as overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor. While a base pointer can point to any of the derived object, the reverse is not true. This is to say we cannot use a pointer to derived class to access an object of base type.

Date: .....
Page: .....

Q.1) What is virtual function? When and how we make the function virtual? Explain with suitable example.

→

A virtual function is a member function declared in the base class with the keyword virtual and uses a single pointer to base class pointer to refer to objects of different classes. When we use the same function name in both base and derived classes, the function in the base class is declared as virtual using the keyword virtual preceding its normal declaration. When a function is made virtual, c++ determines which function is use at runtime based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus by making the base pointer to point object of different version of the virtual functions. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Furthermore, when we use virtual functions, different functions can be executed by the same function call. The information regarding which function to invoke is determin at run time.

```
#include<iostream>
 using namespace std;
  class B
  {
    public: virtual void show()
```

(e.g. same as ahead)

Date :
Page :

Q) What happens when a base and derived classes have the same functions with the same name and these are accessed using pointers with and without using virtual function.

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

class B
{
public: void display()
        {
          cout << "Display base" <<endl;
        }
        virtual void show()
        {
          cout <<"show base"<<endl;
        }
};
class D : public B
{
public: void display()
        {
          cout << "Display base "<<endl;
        }
        void show()
        {
          cout << "show base"<<endl;
        }
};
```

Date: .....
Page: .....

```
int main()
{
    B b1;
    D d1;
    B *bptr;
    cout << "bptr points to base" << endl;
    bptr = &b1;
    bptr->display(); //calls base class function
    bptr->show(); //calls base class function
    cout << "bptr points to derived" << endl;
    bptr = &d1;
    bptr->display(); //calls base class function
    bptr->show(); //calls derived class function
    return 0;
}
```

Output:

```
bptr points to base.
Display base.
show base
bptr points to derived
Display base
show derived
```

~~Base pointer can~~ point to ~~derived class~~

When bptr is made to point the object d (i.e. bptr = &d1), the statement

```
bptr->display();
```

Date :
Page :

calls only the function associated with the B.
(i.e. B::display())
This is because the compiler actually ignores the content of the pointer bptr and chooses a member function that matches the type of the pointer, whereas the statement

bptr->show();

calls the derived version of show(). This is because function show() has been made virtual in base class.
This is because ~~the~~ first the base pointer has the address of the base class object, then its content is changed to contain the address of the derived object.

Q) How can ~~to~~ you achieve runtime polymorphism in c++? Discuss with a suitable example.

We should use virtual functions and pointers to objects to achieve run-time polymorphism. For this, we use functions having same name, same number of parameters, and similar types of parameters in both base and derived classes. The function in the base class is declared as virtual using the keyword virtual. A ~~fn~~ virtual function uses a single pointer to base class pointer to refer to all the derived objects.

When a function in the base class is made virtual, c++ determines which function to use at run time

Date:
Page:

based on the type of object pointed by the base class pointer, rather than the type of the pointer.

↑ (example of virtual function)

Q) A bookshop sells both books and videotapes. Create an abstract class known as media that stores the title and price of a publication. Create two child classes one for storing the number of pages in a book and another for storing the playing time of a tape. A function display is used in all the classes to display the class contents. Create necessary constructors in the child classes to store the information. In the main, display the information regarding the book and tape using the base pointer (an object pointer of the class media.)

```
#include<iostream>
#include<string.h>
  using namespace std;
  class media
   {
     protected:
       char title[20];
       float price;
     public:
       media(char *t, float p)
        {
          strcpy(title,t);
          price=p;
        }
```

Date :
Page :

```
    virtual void display() = 0;
};
class book : public media
{
    int pages;
public:
    book (char *t, float p, int pag) : media(t,p)
    {
        pages = pag;
    }
    void display()
    {
        cout << "Title : " << title << endl;
        cout << "Price : " << price << endl;
        cout << "Pages : " << pages << endl;
    }
};
class tape : public media
{
    int time;
public:  (or char *t)
    tape (char t[], float p, int tm) : media (t,p)
    {
        time = tm;
    }
    void display()
    {
        cout << "Title : " << title << endl;
```

```
        cout << "Price :" << price << endl;
        cout << "Play time(mins) :" << time << endl;
    }
};
int main()
{
    media m[2];
    book b("OOP", 550.25, 350);
    tape t("computing concepts", 255.6, 55);
    m[0] = &b;
    cout << "Information of Book :" << endl;
    m[0]->display;
    cout << "Information of tape : " << endl;
    m[1] = &t;
    m[1]->display();
    return 0;
}
```

Pics

PURE VIRTUAL FUNCTION :

Date : .....
Page : .....

(Deferred methods / Abstract methods)

- A virtual function will become a pure virtual function when we append "= 0" at the end of the declaration of the virtual function
- Example : virtual void display() = 0;
- Pure virtual functions are also known as "do-nothing" functions.
- A pure virtual function is a function declared in a base class that has no definition (implementation / body)
- It serves only as a placeholder.
- The child classes are allowed to inherit them.
- In this situation, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function. Otherwise, the compilation error will occur.

Program :

```
#include <iostream>
 using namespace std;
 class Book
 {
  public:
   virtual void display() = 0;
};
 class Math : public Book
 {
    public:
     void display()
     {
```

Date: .....
Page: .....

```
    cout << "We are studying Math" << endl;
  }
};
class OOP: public Book
{
  public: void display()
          {
            cout << "We are studying OOP" << endl;
          }
};
int main()
{
  Book *bptr;
  Math m;
  OOP o;
  bptr = &m;
  bptr -> display();
  bptr = &o;
  bptr -> display();
  return 0;
}
```

V.V.Imp.

ABSTRACT CLASS:

Date: .....
Page: .....

- A class having at least one pure virtual function is called an abstract class.
- The object of abstract classes cannot be created.
- Pointers to abstract classes can be created for selecting the proper virtual function.
- An abstract class is designed to act only as a base class. It is a design concept in program development and provides a base upon which the program is built.
- sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. For example, let shape be a base class, we cannot provide an implementation of function draw() is shape, but we know every derived class must have the implementation of draw(). In this case, concept of abstraction is used.

```
#include <iostream>
using namespace std;
class shape
{
  public:
   virtual void draw() = 0;
};
class square : public shape
{
   public:
    void draw()
    {
```

Date :
Page :

```
        cout <<"Implementing method to draw square"<<endl;
    }
};
class circle : public shape
{
  public:
    void draw()
    {
      cout <<"Implementing a method to draw circle"<<endl
    }
};
 int main()
  {
    shape *bptr;
    square s;
    circle c;
    bptr = &s;
    bptr -> draw();
    bptr = &c;
    bptr ->draw();
    return 0;
  }
```

Unit 5

TEMPLATES & GENERIC PROGRAMMING

Date :
Page :

Templates :

Templates is a new concept that enables us to define generic classes and functions and thus provide support for generic programming.

Generic programming is an approach where generic types are used as parameters in Algorithm so that they work for variety of suitable datatypes & data structures.

A templates can be used to create family of classes and function. For example : A class template for an array class enables us to create array of various data type such as int array or float array. We can define templa for function, let us say mul() that would help us create various versions of mul() for multiplying integer, float & double type values.

A template can be considered as a kind of macro. When an object of specific type is defined for actual used, the template definition for that class is substituted with the required data type. since a template is defined with parameter that would be replaced by specified data type at the time of actual used of class or function. so a templates are also called a parameterized class or function.

Date: .....
Page: .....

There are two types of templates:
(i) Function template
(ii) class template

1. Class Template

We can create class templates for generic class operations. sometimes we need a class implementation i.e. same for all classes, only the datatype used are different. Normally, we would need to create different class for each datatype or create different ~~class for each datatype~~ member variables or function within a single class. This promotes the coding redundancy & will be hard to maintain as a change in one class as function should be performed on all class/Function. However, class templates make it easy to reuse the same code for all datatypes!

Syntax:

The general form of class template is:

```
template <class T>
 class class_name
 {
    //class member specification
    //with anonymous Type T
    //whenever appropriate
 };
```

Date :
Page :

The prefin template <class T> tells compiler that we are going to declare a template and use T as a typename in the declaration. 'T' maybe substituted by any datatype including the user-defined data types.

While creating object of a class, it is necessary to mention which type of data is used in that object.

Syntax for creating object :

```
class_name <datatype> objectname;
```

E.g.:

For int : example <int> e1;
//e1 is object of class example

For float : example <float> e2;
//e2 is object of class example

For char : example <char> e3;
//e3 is object of class example

Q) Write a program to add two integer and float data using class template.

```
#include <iostream>
using namespace std;
template <class T>
```

Date :
Page :

```
class sample
{
  private : T a,b,s;
  public : void setData (T x, T y)
           {
             a = x;
             b = y;
           }
           void add()
           {
             s = a+b;
             cout << s << endl;
           }
};
int main()
{
   Sample <int> s1;
   Sample <float> s2;
   s1.setData (5, 8);
   s2.setData (3.5, 8.9);
    cout << "sum of integer value :" << endl;
    s1.add();
    cout << "sum of float value :" << endl;
    s2.add();
    return 0;
}
```

Date :
Page :

Q) WAP to sum and product of two integer and two float using class template.

```
#include <iostream>
using namespace std;
template <class T>
class sample
{
  private: T a,b,s,p;
  public: void setData (T x, T y)
          {
            a = x;
            b = y;
          }
          void calculate()
          {
            s = a+b;
            p = a*b;
            cout << " sum = " << s << " Product = " << p << endl;
          }
};
int main()
{
 Sample <int> s1;
 sample <float> s2;
 s1.setData (5,8);
 s2.setData (3.5, 8.9);
 cout << "calculation for integer :" << endl;
```

Date: .....
Page: .....

```
    s1.calculate();
    cout << "Calculation Of float value: " << endl;
    s2.calculate();
    return 0;
}
```

Q) WAP to find maxm number betn two numbers using class template.

```
#include <iostream>
using namespace std;
template <class T>
class sample
{
  private: T a,b;
  public: void setData(T x, T y)
          {
            a=x;
            b=y;
          }
          T max()
          {
            return (a>b)? a : b;
          }
};
int main()
{
  sample <int> s1;
```

Date :
Page :

```
sample <float> s2;
s1.setData (5,8);
s2.setData (3.5, 8.9);
cout <<" Max of integer value =" << s1.max() <<endl;
cout <<" Max of float value =" <<s2.max() <<endl;
return 0;
}
```

Q) Create a class template to find scalar product of vector of integers and vector of floating point number

```
#include <iostream>
using namespace std;
template <class T>
class Vector
{
  private: T a,b,c;
  public: Vector(T x, T y, T z)
          {
            a=x;
            b=y;
            c=z;
          }
          T operator *(Vector p)
          {
            T sum;
            a = a*p.a;
            b = b*p.b;
```

```
        c = c * p.c;
        sum = a+b+c;
        return sum;
    }
    void display()
    {
      cout<<a<<"i+"<<b<<"j+"<<c<<"k"<<endl;
    }
};
int main()
{
   Vector<int> v1(5,6,7), v2(9,10,11);
   cout<<"v1 =";
   v1.display();
   cout<<"v2 =";
   v2.display();
   cout<<"scalar product of integer value ="<<v1*v2;
   Vector<float> m(1.1,2.2,3.3), n(5.5, 6.6, 7.7);
   cout<<"m =";
    m.display();
   cout<<"n =";
   n.display();
   cout<<scalar product of float value ="<<m*n;
   return 0;
}
```

Date :
Page :

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template. They are declared comma separated list within the template specification as shown below:

```
template <class T1, class T2>
 class class_name
 {
    //class body
 };
```

Example:

```
#include <iostream>
 using namespace std;
 template <class T1, class T2>
  class Test
  {
   private: T1 a;
            T2 b;
   public: Test (T1 x, T2 y)
           {
            a=x;
            b=y;
           }
           void show()
           {
            cout <<a <<endl;
            cout << b <<endl;
           }
};
```

Date :
Page :

```
int main()
{
  Test <float, int > t1 (1.23, 123);
  Test < int, char > t2 (100, 'w');
  t1.show();
  t2.show();
  return 0;
}
```

Q) WAP to perform sum of two integers, two float, one integer and one float using class template.

```
#include <iostream>
using namespace std;
template <class T1, class T2>
class Test
{
  private: T1 a;
           T2 b,s;
  public: Test (T1 x, T2 y)
          {
            a = x;
            b = y;
          }
          void add()
          {
            s = a+b;
            cout<< "sum =" <<s <<endl;
          }
};
```

Date: .....
Page: .....

```
int main ()
{
  Test <int, int > t1 (123, 123);
  Test <float, float > t2 (1.2 , 2.3);
  Test < int, float > t3 (10, 2.3);
  cout << "sum of two integer = " <<endl;
  t1.add ();
  cout << "sum of two float = " <<endl;
  t2.add ();
  cout <<"sum of one integer and one float = "<<endl
  t3.add ();
  return 0;
}
```

Q) WAP to perform sum and product of one integer and one floa using class template.

```
#include <iostream>
 using namespace std ;
 template < class T1, class T2 >
  class Test
  {
    private : T1 a;
              T2 b, p, s;
    public : Test (T1 x, T2 y)
             {
               a = x;
               b = y;
             }
```

Date: .....
Page: .....

```
    void add()
    {
      s = a + b;
      cout << "sum = " << s << endl;
    }
    void product()
    {
      p = a * b;
      cout << "Product = " << p << endl;
    }
};
int main()
{
  Test < int, float > t1 (3, 3.5);
  cout << "sum of integer and float = " << endl;
  t1.add();
  cout << "Product of integer and float = " << endl;
  t1.product();
  return 0;
}
```

Q) Define a class called stack, and implement generic methods to push and pop elements from stack

```
#include <iostream>
  using namespace std;
#define max 10
```

Date :
Page :

```
template <class T >
class stack
{
  private : T stk [max];
            int top;
  public : stack ()
           {
             top = -1;
           }
          void push (T data)
          {
            if (top = = (max-1))
            {
              cout << "stack is full" << endl;
            }
            else
            {
              top++;
              stk [top] = data;
            }
          }
         void pop()
         {
          if (top = = -1)
          {
           cout << "stack is empty " << endl;
          }
```

Date: .....
Page: .....

```
        else
        {
          top--;
        }}
    void show()
      {
        for(int i = top; i >= 0; i--)
        {
          cout << "stack [" << i << "]" << stk[i] << endl;
        }
      }
};
int main()
{
  stack <char> s;
  s.push('a');
  s.push('b');
  s.push('d');
  s.push('e');
  s.push('f');
  s.show();
  cout << "Top is poped" << endl;
  s.pop();
  s.show();
  return 0;
}
```

Date :
Page :

Function Template

- Function template can be used to create family of functio with different argument type.
- A single function template can work with different data types.
- Any type of function argument is accepted by the functio
- The general format of function template is as :

```
template <class T>
return-type function-name (arguments of type T)
{
    //body of function with type T
    - - - - -
}
```

Q) WAP to display largest among two numbers using function template.

```
#include <iostream>
using namespace std;
template <class T>
T large (T x, T y)
{
    return (x>y)? x : y;
}
int main()
{
    int i1 =4, i2 = 5;
    float f1 = 50.6, f2 = 10.5;
```

Date:
Page:

```
char c1 = 'a', c2 = 'A';
cout << large (i1, i2) << " is larger" << endl;
cout << large (f1, f2) << " is larger" << endl;
cout << large (c1, c2) << " has larger ASCII " << endl;
return 0;
}
```

Q) WAP to create function template to swap two values.

```
#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 i1 = 4, i2 = 5;
    float f1 = 50.6, f2 = 10.5;
    cout << "Before swapping the values are :" << endl;
    cout << " i1 = " << i1 << " i2 = " << i2 << endl;
    cout << "f1 = " << f1 << "f2 = :" << f2 << endl;
    swap (i1, i2);
    swap (f1, f2);
```

```
cout<<" After swapping values are"<<endl;
cout <<"i1 =" << i1 << " i2 = " << i2 << endl;
cout <<"f1 ="<< f1 <<"f2 = " <<f2 <<endl;
return 0;
}
```

Q) Write a function template to calculate average & multiplication of number.

```
#include <iostream>
using namespace std;
template <class T>
void calculate (T a, T b)
{
    T avg, pro;
    avg = (a+b)/2;
    pro = a*b;
    cout << "Average =" <<avg << "Product =" <<pro <<end
}
int main()
{
    int i1 = 4, i2 = 5;
    float f1 = 50.6, f2 = 10.5;
    cout << "Average and product of int are" <<endl;
    calculate (i1, i2);
    cout <<" Average and product of float are "<<endl;
    calculate (f1, f2);
    return 0;
}
```

Date :
Page :

Imp

Q) Write a function template to calculate sum and average of two integer and two floats ~~using funct~~

```
#include<iostream>
using namespace std;
template <class T>
void calculate (T a, T b)
{
    T avg, sum;
    sum = a+b;
    avg = (a+b)/2;
    cout << "sum = " << sum << " Average = " << avg << endl;
}
int main()
{
    int i1 = 4, i2 = 5;
    float f1 = 50.6, f2 = 10.5;
    cout << "sum and average of integer value :" << endl;
    calculate (i1, i2);
    cout << "sum and average of floating value = " << endl;
    calculate (f1, f2);
    return 0;
}
```

Date :
Page :

Function ~~wit~~ Template with multiple parameter

We can use more than one generic datatype in template statement using comma separated list as shown below.

```
Template <class T1, class T2>
return_type function_name (arguments with type T1,T2)
{
  //body of function
}
```

Q.) WAP to add two integers, two floats and one integer and one float number respectively. Display the final result in float.

```
#include <iostream>
using namespace std;
template <class T1,class T2>
float calculate (T1 a, T2 b)
{
   return (a+b);
}
int main()
{
   int i1,i2,i3;
   float f1,f2,f3;
   float s1,s2,s3;
   cout << "Enter two integer values" <<endl;
```

Date:
Page:

```
cin >> i1 >> i2;
s1 = calculate(i1, i2);
cout << "Enter two float values" << endl;
cin >> f1 >> f2;
s2 = calculate(f1, f2);
cout << "Enter one integer and one float value" << endl;
cin >> i3 >> f3;
cout << "sum of two integer = " << s1 << endl;
s3 = calculate(i3, f3);
cout << Sum of two floating = " << s2 << endl;
cout << Sum of one int and one float = " << s3 << endl;
return 0;
}
```

v. Imp 3

TEMPLATE FUNCTION OVERLOADING:

- A template function may be overloaded either by template function or ordinary function of its name. In such cases, the overloading resolution is accomplished as follows:

1. Calling an ordinary function that must exactly match.
2. Call the template function that could be created with an exact match.
3. Try normal overloading resolution to ordinary function and calls the ~~func~~ one that matches.

Date:
Page:

- An error is generated if no match is found.

Note:
No automatic conversions are applied to argument on template function.

For example:

* Overloading function template by another template function

```
#include <iostream>
 using namespace std;
 template <typename T>
  T maximum (T a, T b, T c)
   {
     if (a>b && a>c)
      {
        return a;
      }
    else if (b>c)
     {
       return b;
     }
    else
     {
       return c;
     }
}
```

Date: .....
Page: .....

```
template <typename T>
T maximum (T a, T b)
{
  if (a>b)
  {
    return a;
  }else
  {
    return b;
  }
}
template <typename T>
T display (T a)
{
  cout <<"maximum =" <<a <<endl;
}

int main()
{ float max;
  max = maximum (4.2, 6.5);
  display (max);
  max = maximum (4, 6, 3);
  display (max);
  char ch;
  ch = maximum ('a', 'x');
  display (ch);
  ch = maximum ('a', 'b', 'c');
  display (ch);
  return 0;
}
```

Date :
Page :

* Overloading function template by non-template function

In case of overloading function by non template function the compiler gives priority to ordinary function with exact match. If the exact match is not with the ordinary function, then compiler prefers the template function.

```
#include<iostream>
using namespace std;
template<typename T>
void maximum (T a, T b, T c) :"
{
   if (a>b && a>c)
   {
      cout<<a<<" is maximum"<<endl;
   }
   else if (b>c)
   {
      cout<<"b<<" is maximum"<<endl;
   }
   else
   {
      coult<<c<<" is maximum "<<endl;
   }
}
```

Date: .....
Page: .....

```
void maximum (int a ,int b , int c)
{  if (a>b && a>c)
   {
     cout << a << " is maximum" <<endl;
   }
   else if (b>c)
   { cout << b<< " is maximum" <<endl;
   }
   else
   { cout << c << " is maximum" <<endl;
   }
}
int main()
{
  maximum (1,2,3);
  maximum (1.2,2.2,3.2);
  return 0;
}
```

Q1

Imp: exception handling / Multi...

Date: .....
Page: ..... main (7,8)

Unit - 6

EXCEPTION HANDLING & STREAM I/O

V.Imp

Exception Handling

- Exceptions are run-time 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 it bounds
  - → Running out of memory space or disk space

- The exception handling is a mechanism to detect and report an exceptional circumstances at run time, so that appropriate action can be taken. It provides a typesafe, integrated approach for coping with the unusual predictable problem that arise while executing a program.

Types of exception:

1. synchronous
2. Asynchronous

1. Synchronous

Errors such as array index out of range and overflow belongs to synchronous type exception.

Asynchronous

Errors that are caused by the events beyond the control of program (such as keyboard interrupts) are called asynchronous exceptions.

Date:
Page:

Note: The exception handling mechanism in C++ can handle only synchronous exception.

- The exception handling mechanism such as the separate error handling code that performs the following task:
  1. Find the problem (Hit the exception)
  2. Inform the error has occured (Throw the exception)
  3. Receive the error information (catch the exception)
  4. Take the corrective action (Handle the exception)

- The error handling code mainly consist of two segments, one to detect error and throw exception and other to catch the exception and to take appropriate actions.

- C++ exception handling mechanism is basically built upon three keywords namely try, throw, catch.

1. The keyword try is used to preface a block of statements (surrounded by curly braces) which may generate exception. This block of statement is known as try block.

2. When an exception is detected, it is thrown using a throw statement, in the try block.

3. A catch block is defined by the keyword catch, catches the exception thrown by throw statement in try block and handles it appropriately.

Date :
Page :

```
try
{
  // block of statements which detect and throw exception
  throw exception
catch (type arg)
}
catch (type arg) // catch the exception
{
  // block of statement that handles the exception
}
```

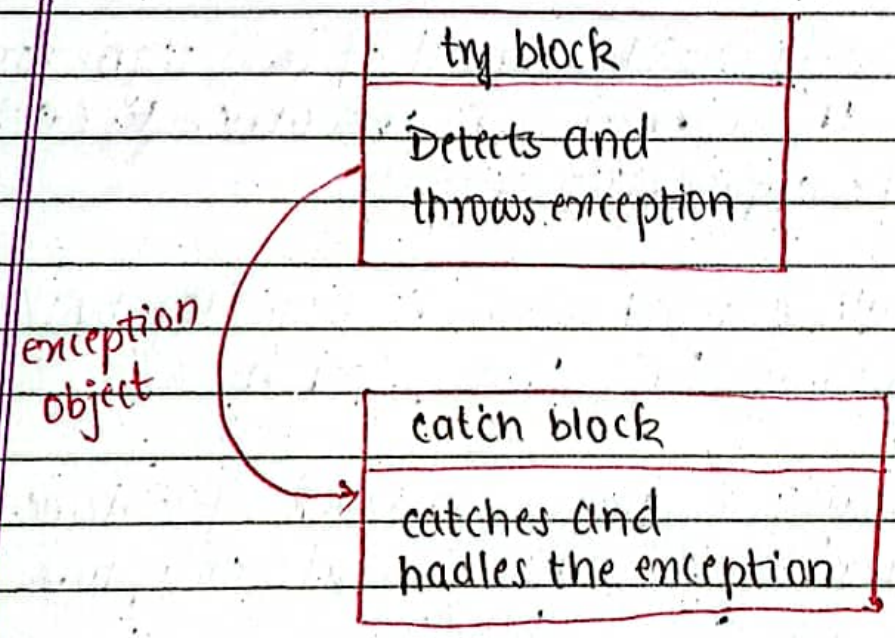


Fig. the block throwing exception

When the try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block. If the type of object thrown matches the argument type in the catch statement, then the catch block is executed for handling the exception.

Date: _____
Page: _____

If they donot match, the program is aborted with the help of abort function, which is executed implicitly by the compiler. When no exception is detected and thrown, the control goes to the statement immediately after the catch block i.e. catch block is skipped.

e.g.

```
#include<iostream>
using namespace std;
int main()
{ int a,b,x;
  cout<<"Enter value of 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)
  {
    cout<<"Exception caught : DIVIDE BY ERROR"<<endl;
  }
  cout<<" END";
  return 0;
}
```

Date :
Page :

Output : enter values of a and b
20 15
Result (a/n) = 4
END

enter value of a and
10 10
Exception caught: DIVI
BY ZER
END

MULTIPLE CATCH STATEMENT

- It is possible that a program statement has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a 'try' (much like condition in switch statement).

- When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that matches is executed. After executing the handler, the control goes to the first statement, after the last catch block for that 'try' When no match is found, the program is terminated.

Note : It is possible that arguments of several catch statements matches the type of an exception. In such case, the first handler that matches the exception type is executed.

Date: .....
Page: .....

e.g. #include <iostream>

```
using namespace std;
void test (int n)
{
  try
  { if (n == 1)
      throw n;
    else if (n == 0)
      throw 'n';
    else if (n == -1)
      throw 1.0;
    cout << "End of try-block" << endl;
  }
  catch (char c)
  { cout << "caught a character" << endl;
  }
  catch (int m)
  { cout << "caught an integer" << endl;
  }
  catch (double d)
  { cout << "caught a double" << endl;
  }
  cout << "End of try-catch system" << endl;
}
int main()
{ cout << "Testing multiple catches" << endl;
  cout << "n == 1" << endl;
  test (1);
  cout << "n == -1" << endl;
  test (-1);
```

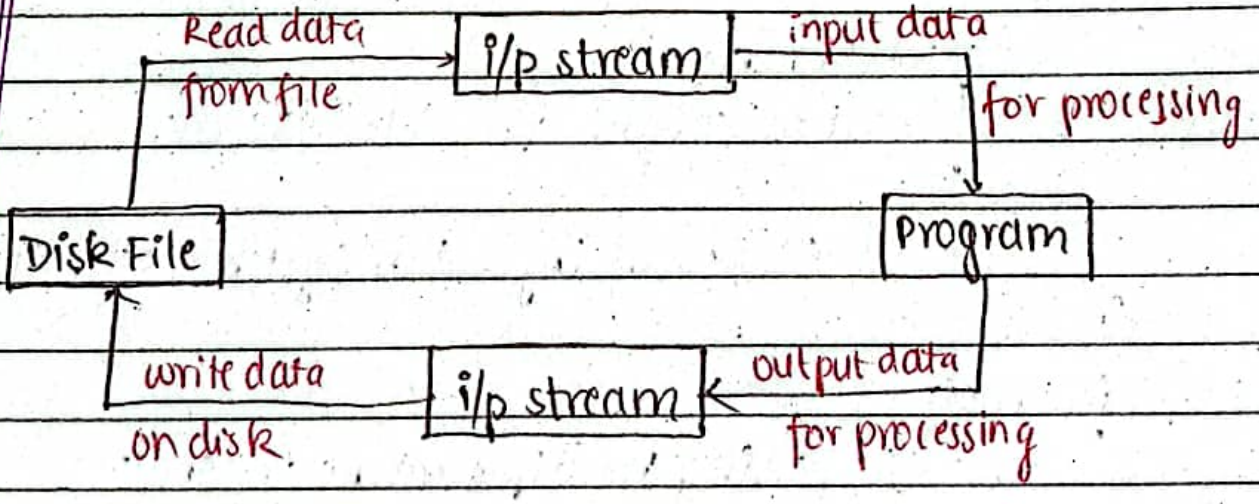
Date :
Page :

```
    cout<< "x = = 2" <<endl;
    test (2);
    return 0;
}
```

FILE HANDLING :

A file is the fundamental unit that store important user data. There are many different sorts of file, which are represented by their extensions.
E.g .txt, .cpp, .html, .php etc.

File Handling stands for the manipulation of files storing relevant data using a programming language (which is C++ in our case)



In C++, there are mainly three stream classes
fstream, ifstream, ofstream

Date:
Page:

1 ofstream
This stream class signifies the o/p file stream and is applied to create files for writing information to files.

2 ifstream
This stream class signifies the i/p file stream and is applied for reading information from files.

3 fstream
This stream class can be used for both read and write from/to files.

Opening a file:

To start working with file, first we need to open it in our program.

For opening file,
void open (const char *filename, ios::open_mode)
file name
opening mode

File opening modes:

Modes	Description
in	Open files for reading purpose
out	Open file for writing purpose
app	To append data to end of the file
trunc	If file already exist, all content will be removed immediatel
binary	Operation are performed in binary mode rather than text

Date :
Page :

Closing a file

It is simply done by close function.

Writing to file

The steps req^d to writing to a file are:

1. Create a file stream object, capable of writing a file such as an object of ofstream or fstream class.
2. Open a file by calling the open method with the stream object.
3. Check whether the file has to be successfully open, if yes, then start writing.

```
#include<iostream>
#include<fstream>
  using namespace std;
  int main()
  {
     fstream FILE;
     FILE.open("new_file.txt", ios::out);
     if(!FILE)
     {
       cout<<"File creation failed"<<endl;
       exit(0);
     }
     cout<<"New File created"<<endl;
     FILE<<"Hello World!! \n";
```

Date:
Page:

```
FILE<<"My first file program";
FILE.close();
return 0;
}
```

Reading from a file

The following steps must be followed before reading a file:

- create a file stream object capable of reading a file, such as an object of ifstream or fstream.
- Open a file through open mode with a stream object.
- check whether a file has been successfully opened, if yes then start reading.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream FILE;
    FILE.open("new_file.txt", ios::in);
    if(!FILE)
    {
      cout << "File opening failed";
      exit(0);
    }
    cout <<"Reading from file..."<<endl;
    while(!FILE.eof())
    {
```

```
        FILE >>ch;
        cout <<ch;
    }
    FILE.close();
    return 0;
}
```

Q) Write a cpp program to input n integer from user write it to a file integer.txt. Now, read number from file & calculate average. Also append it at last of a file.

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

int main()
{
  int i,n,num,sum = 0;
  float avg = 0;
  cout << "Enter total number of integer to read:" <<endl;
  cin >> n;
  fstream FILE;
  FILE.open("integer.txt", ios::out);
  if (!FILE)
  {
    cout << " File opening failed " <<endl;
    exit(0);
  }
```

Date: .....
Page: .....

```
for (int i=0 ; i<n; i++)
{
  cout<<"Enter "<<i+1 <<"Number" <<endl;
  cin >> num;
  FILE <<num<<" ";
}
FILE. close();
FILE. open("integer.txt", ios:: in);
if (!FILE)
{
  (out <<"File opening failed" <<endl;
  exit (0);
}
while (FILE >> num)
{
   sum = sum +num;
}
 FILE. close();
FILE.Open ("integer.txt", ios :: app);
if (! FILE)
{
  cout <<"File opening failed";
  exit (0);
}
avg = (float) sum/n;
cout <<"Average is :" <<avg;
FILE <<" \n Average = "<<avg <<endl;
FILE.close();
return 0;
}
```

Date :
Page :

3) Writing & Reading class object

* Writing object to a file
  FILE.write ((char *)& object_name, sizeof (object_name));

* Reading object from file
  FILE.read ((char *) & object_name, sizeof(object_name));

Qsn) Create a class called student with name, address & roll_no. Write the data to the file student details.txt Read the data from the file & display it

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

  class student
  {
     private: char name[50], address[50];
              int roll;
     public: void input()
             {
               cout<<"Enter student details "<<endl;
               cin>> name >> address >> roll;
             }
              void display()
              {
```

Date :
Page :

```
        cout << "Name = " << name << endl;
        cout << "Address = " << address << endl;
        cout << "Roll = " << roll << endl;
    }
};
int main()
{
    student s;
    fstream FILE;
    FILE.open("student_details.txt", ios::out);
    if (!FILE)
    {
        cout << "File opening failed " << endl;
        exit(0);
    }
    s.input();
    FILE.write((char*)&s, sizeof(s));
    cout << "File opened and data written successfully" << endl;
    FILE.close();
    FILE.open("student_details.txt", ios::in);
    if (!FILE)
    {
        cout << "File opening failed" << endl;
        exit(0);
    }
    FILE.read((char*)&s, sizeof(s));
    s.display();
    FILE.close();
    return 0;
}
```

Date :
Page :

Q) Write a.cpp program to i/p record of n students (name, address, rollno.) and save it to file student.txt. Read the file and display the record of student whose address is Kathmandu.

```
#include<iostream>
#include<fstream>
#include<string.h>
using namespace std;
class student
{
  private: char name[50], address[50];
           int roll;
  public:
    void input()
    {
     cout<<"Enter student details"<<endl;
     cin>>name>>address>>roll;
    }
    void display()
    {
     cout<<"Name ="<<name<<endl;
     cout<<"Address ="<<address<<endl;
     cout<<"Roll ="<<roll<<endl;
    }
    char *getaddress()
    {
     return address;
    }
};
```

Date: .....
Page: .....

```
int main()
{
  int i,n;
  student s[100];
  fstream FILE;
  cout << "Enter total no. of students " <<endl;
  cin>>n;
  FILE.open ("student.txt", ios ::out) ;
  if (!FILE)
  {
    cout << "File opening failed ";
    exit (0);
  }
  for (i=0; i<n; i++)
  {
    cout<< "Enter details of "<<i+1 <<."student" <<endl;
    s[i]. input ();
    FILE.write ((char*)& s[i], sizeof ((s[i])));
  }
  cout<<"File opened and closed successfully" <<endl;
  FILE close ();
  FILE.open ("student.txt", ios: in);
  if (!FILE)
  {
    cout <<"File opening failed" <<endl;
    exit (0);
  }
  for (i=0, i<n; i++)
  {
```

```
FILE.read((char*)&s[i], sizeof(s[i]));
if(strcmp(s[i].getAddress(), "Kathmandu")==0)
{
  cout<<"The details of student"<<endl;
  s[i].display;
}
}
FILE.close();
return 0;
}
```

CRC (Class Responsibility Collaborator)

A CRC card is a tool used in object oriented design to help define and organize the responsibilities and interactions of classes in a system. Each card represents a class and include three main sections.

Class Name	
Responsibilities	Collaborators

- the name of the class
- the functions or tasks that the class is responsible for
- other classes that this class interacts with to fulfill its responsibilities

Date: .....
Page: .....

Example: let us consider a simple example involving a library management system. The CRC card for 'Book' class is as:

Book	
Responsibilities	Collaborators
1. Store book details (e.g. title, author, ISBN)	library: interact to check in and check out books, update availability.
2. Provide information about the book (e.g. getTitle(), getAuthor())	user: Request book information and performs checkouts or returns.
3. Update book information (e.g. setTitle(), setAuthor(), setISBN())	catalog: used to search and list books.
4. Manage the availability of book status (e.g. isAvailable(), checkout(), returnBook())	

* CRC card for student class in student enrollment system

P.T.O.

Date :
Page

Student

Responsibilities	Collaborators
1. Store student details (e.g. name, id, DOB etc.)	1. Enrollment system · To check and manage student course enrollment.
2. Update personal information	2. course : To get details about the courses the student is enrolled in.
3. Enroll in course	
4. View enrollment history	

* CRC card for course

Course

Responsibilities	Collaborators
1. Store course details (e.g. course id, title, description, credits etc.)	1. student To manage which student are enrolled in the course.
2. Manage course schedule.	
3. Track enrolled students.	2. Enrollment system For handling enrollments and withdrawals
4. Provide course information to students.	3. Instructor To provide course related content and updates.