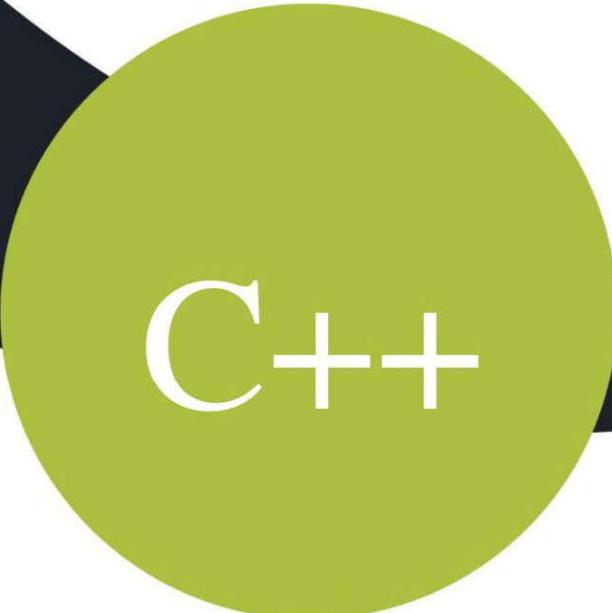
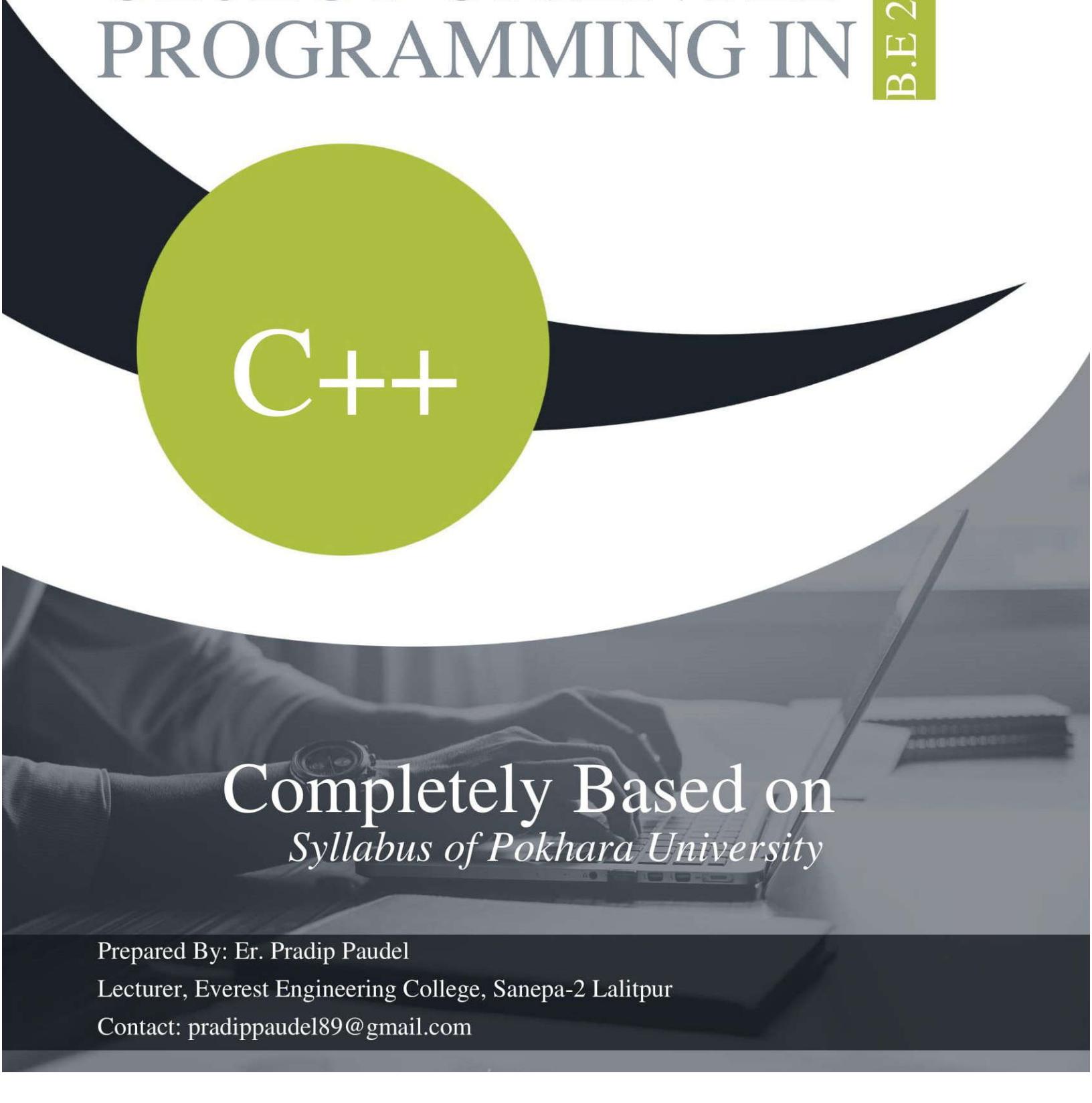


A complete Notes on OBJECT ORIENTED PROGRAMMING IN

B.E 2nd SEM



C++



Completely Based on
Syllabus of Pokhara University

Prepared By: Er. Pradip Paudel

Lecturer, Everest Engineering College, Sanepa-2 Lalitpur

Contact: pradippaudel89@gmail.com

10. CMP 104.3 Object Oriented Programming in C++ (3-1-3)

Evaluation:

	Theory	Practical	Total
Sessional	30	20	50
Final	50	-	50
Total	80	20	100

Course Objectives:

- To familiarize with Object Oriented Concept.
- To introduce the fundamentals of C++
- To enable the students to solve the problems in Object Oriented technique
- To cope with features of Object Oriented Programming

Course Contents:

Chapter	Content	Hrs.
1	Thinking Object Oriented Object oriented programming a new paradigm, a way of viewing world agent, types of classes, computation as simulation, coping with complexity, nonlinear behavior of complexity, abstraction mechanism	4
2	Classes and Methods: Review of structures, classes and inheritance, state, behavior, method, responsibility, encapsulation, data hiding, Functions: friend function, inline function, static function, reference variable, default argument	7
3	Message, Instance and Initialization Message, message passing formalization, message passing syntax in C++, mechanism for creation and initialization (constructor and its types), Issues in creation and initialization: memory map, memory allocation methods and memory recovery	6
4	Object Inheritance and Reusability Introduction to inheritance, Subclass, Subtype, Principle of Substitutability; Forms of polymorphism and their implementation in C++, inheritance merits and demerits, composition and its implementation in c++, The <i>is-a</i> rule and <i>has-a</i> rule, Composition and Inheritance contrasted, Software reusability	9
5	Polymorphism Polymorphism in programming language, Varieties of polymorphism, compile time polymorphism, function overloading, operator overloading, type conversion, polymorphic variable, run time polymorphism, object pointer, this pointer, virtual function, overriding, deferred method, pure polymorphism.	8
6	Template and generic programming Generic and template functions and classes, cases study: container class and the	4

standard template library, Exception handling

7 Object oriented Design

7

Reusability implies non-interference, Programming in small and programming in large, components and behaviors, role of behaviors in OOP, CRC, sequence diagram, Software components, formalizing the interface, interface and implementation, Design and representation of components, coming up with names, implementation components, integration of components

Laboratory Work

There shall be 20 exercises in minimum, as decided by the faculty. The exercises shall encompass a broad spectrum of real-life and scientific problems, development of small program to the development of fairly complex subroutines, programs for engineering applications and problem solving situations. Laboratory assignments will be offered in groups of two to four for evaluation purpose. In general, the Laboratory Work must cover assignments and exercises from the following areas:

1. Data types – control structures, functions and scoping rules.
2. Composite data types, C++ strings, use of " Constant " keyword, pointers and references
3. Classes and data abstraction
4. Inheritance, abstract classes and multiple inheritance
5. Friend functions, friend classes and operator overloading.
6. Static class members
7. Polymorphism, early binding and late binding
8. C++ type conversion
9. Exception handling
10. Function templates, class templates and container classes.

Textbooks:

1. Budd, T., *An Introduction to Object Oriented Programming*, Second Edition, Addison-Wesley, Pearson Education Asia, ISBN: 81-7808-228-4.
2. R. Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications Ltd. India, 1999

Reference Books:

1. E Balaguruswamy, Object Oriented Programming with C++, Third Edition
2. Tata McGraw-Hill ISBN:0-07-059362-0, Parson David, Object Oriented Programming with C++, BPB Publication\ISBN817029-447-9

CHAPTER 1

Thinking Object Oriented

Procedural Oriented Programming:

Conventional Programming, using high level languages such as COBOL (Common Business Oriented Language), FORTAN (Formula Translation) and C, is commonly known as procedural oriented programming (POP). In procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. POP basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as function. A typical program structure for procedural programming is shown in the figure below. The technique of hierarchical decomposition has been used to specify the task to be completed for solving a problem.

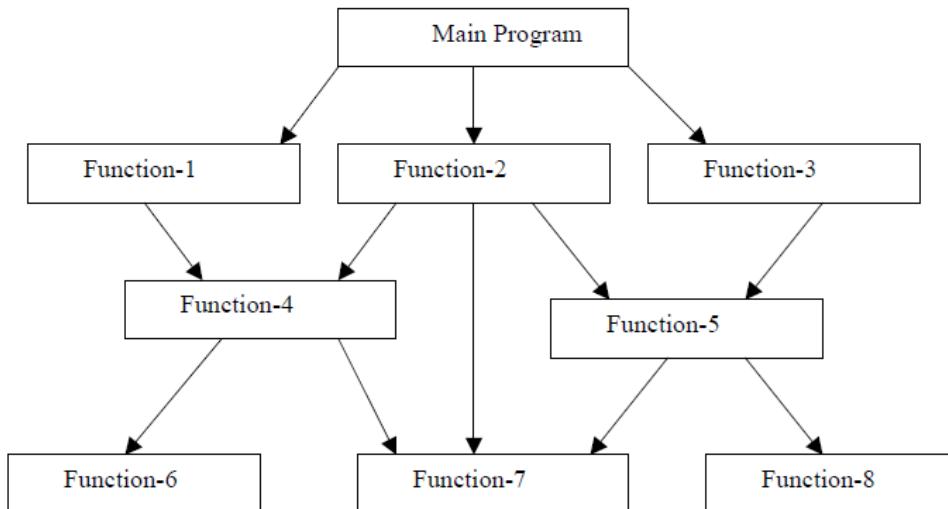
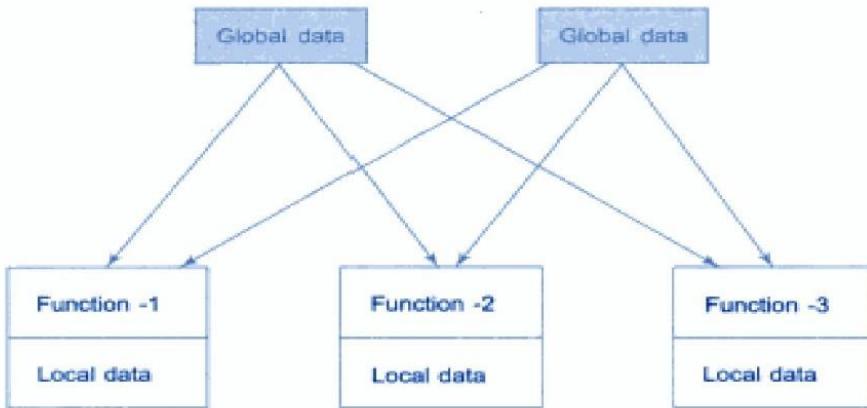


Fig. 1.2 Typical structure of procedural oriented programs

In a multi-function program, many important data items are placed as globally. So that they may be accessed by all the functions. Each function may have its own local data. The figure shown below shows the relationship of data and function in a procedure-oriented program.



Relationship of data and functions in procedural programming

Some features/characteristics of procedure-oriented programming are:

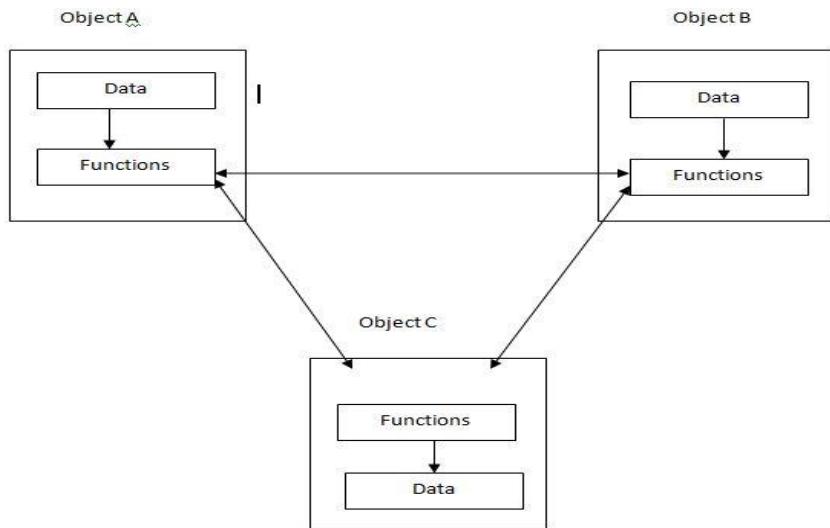
- Emphasis is on doing things (Algorithms).
- Large programs are divided into smaller programs called as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top-down* approach in program design.

Limitations of Procedural Oriented Programming

- Procedural languages are difficult to relate with the real world objects.
- Procedural codes are very difficult to maintain, if the code grows larger.
- Procedural languages does not have automatic memory management .Hence, it makes the programmer to concern more about the memory management of the program.
- The data, which is used in procedural languages are exposed to the whole program. So, there is no security for the data.
- Creation of new data type is difficult. Different data types like complex numbers and two dimensional co-ordinates cannot be easily represented by POP.

Object oriented programming Paradigm

OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The organization of data and functions in the object-oriented programs shown in the figure:



The data of an object can be accessed only by the function associated with that object.

The fundamental idea behind object-oriented programming is to combine or encapsulate both data (or instance variables) and functions (or methods) that operate on that data into a single unit. This unit is called an object. The data is hidden, so it is safe from accidental alteration. An object's functions typically provide the only way to access its data. In order to access the data in an object, we should know exactly what functions interact with it. No other functions can access the data. Hence OOP focuses on data portion rather than the process of solving the problem.

An object-oriented program typically consists of a number of objects, which communicate with each other by calling one another's functions. This is called sending a message to the object. This kind of relation is provided with the help of communication between two objects and this communication is done through information called message. In addition, object-oriented programming supports encapsulation, abstraction, inheritance, and polymorphism to write programs efficiently. Examples of object-oriented languages include Simula, Smalltalk, C++, Python, C#, Visual Basic .NET and Java etc.

Features of Object oriented programming

Some features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are called objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

Advantages of Object-oriented Programming

- It is easy to model a real system as programming objects in OOP represents real objects. The objects are processed by their member data and functions. It is easy to analyze the user requirements.
- Elimination of redundant code due to inheritance, that is, we can use the same code in a base class by deriving a new class from it.
- Modularize the programs. Modular programs are easy to develop and can be distributed independently among different programmers.
- In OOP, data can be made private to a class such that only member functions of the class can access the data. This principle of data hiding helps the programmer to build a secure program that cannot be invaded by code in other part of the program.
- With the help of polymorphism, the same function or same operator can be used for different purposes. This helps to manage software complexity easily.
- Large problems can be reduced to smaller and more manageable problems. It is easy to partition the work in a project based on objects.
- Program complexity is low due to distinction of individual objects and their related data and functions.

Disadvantages of Object Oriented Programming

- Sometimes, the relation among the classes become artificial in nature.
- Designing a program in OOP concept is a little bit tricky.
- The programmer should have a proper planning before designing a program using OOP approach.
- Since everything is treated as objects in OOP, the programmers need proper skill such as design skills, programming skills, thinking in terms of objects etc.
- The size of programs developed with OOP is larger than the procedural approach.
- Since larger in size, that means more instruction to be executed, which results in the slower execution of programs.

Applications of Object-oriented Programming

1. Client-Server System
2. Object Oriented Database
3. Real Time Systems Design
4. Simulation and Modeling System
5. Hypertext, Hypermedia
6. Neural Networking and Parallel Programming
7. Decision Support and Office Automation Systems
8. CIM/CAD/CAM Systems
9. AI and Expert Systems

Differences between structured and Object oriented Programming Language

Structured Programming/procedural oriented Programming	Object oriented Programming
1. Large Programs are divided into smaller self-contained program segment known as functions.	1. Program are divided into entity called objects.
2. Focuses on process/logical structure and then data required for that process.	2. Object oriented Programming is designed which focuses on data.
3. Structured Programming follows top-down approach.	3. Object oriented Programming follows bottom-up approach.
4. Data and functions don't tie with each other.	4. Data and functions are tied together.
5. Structured programming is less secure as there is no way of data hiding.	5. Object oriented programming is more secure as having data hiding feature.
6. Structured programming can solve moderately complex programs.	6. Object oriented programming can solve any complex programs.
7. Provides less reusability and more function dependency.	7. Provide less function dependency and more reliability.
8. Data moves free around the system from one function to another.	8. Data is hidden and cannot accessed by external functions.
9. Eg.C, pascal ,ALGOL	9. Eg.C++,Java and C# (C sharp)

A way of viewing the world

Describe how object oriented programming models the real world problem with reference of agents, method, behavior and responsibilities. [PU: 2017 fall]

To illustrate the major idea of object oriented programming, Let us consider the real world scenario.

Suppose an individual named Chris wishes to send flowers to a friend named Robin, Who lives in another city. Because of the distance, Chris cannot simply pick the flowers and take them to Robin in person. Nevertheless, it is a task that is easily solved.

Chris simply walks to a nearby flower shop, run by a florist named Fred. Chris will tell Fred the kinds of flowers to send to Robin and the address to which they should be delivered .Chris can then can be assured that the flowers can be delivered expediently and automatically.

In above example,

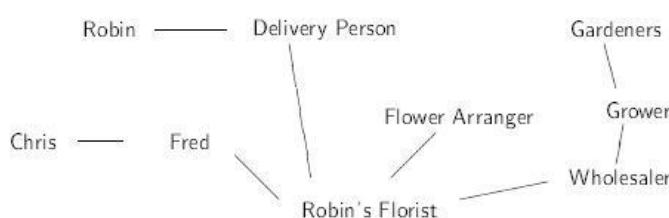
- Fred is a **agent (objects)** to deliver flower.
- **Message** of wishes with specification of flowers is passed to florist (named Fred) who has **responsibility** to satisfy request.
- Fred uses some **method or algorithm or set of operations** to do this.
- There will community of agents to complete a task. Communication is established to all the concern mediators in a channel to deliver the flowers.
- How the florist satisfies the request is not Chris concern (**data hiding/information hiding**).

Agents and Communities

In above example Chris solved his problem with help of agent (object) Fred to deliver the flower. **There will be community of agents to complete a task. In above example Fred will communicate with Robin's florist.**

An object oriented program is structured as a community of interacting agents called objects.

Each object has a role to play. Each object provides a service, or performs an action that is used by other members of community.



The community of agents helping delivery flowers

Messages and methods

Chris request Fred for delivering flower to his friend Robin. This request lead to other requests, which lead to still more requests, until the flowers ultimately reached Chris friend, Robin. We see therefore a members of this community interact with each other making requests.

Action is initiated in object-oriented programming by the transmission of a message to an agent (an object) responsible for the action. The messages encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request. The receiver is the object to whom the message is sent. If the receiver accepts the messages, it accepts the responsibility to carry out the indicated action. In response to a message, the receiver will perform some method to satisfy the request.

Message Versus Procedure calls

In message passing

- There is a designated receiver for that message (the receiver is some object to which the message is sent).
- Interpretation of the message is dependent on the receiver and can vary with different receivers.
- There is a late binding between the message (function or procedure name) and the code fragment (method) used to respond the message.

In a procedure call

- There is no designated receiver.
- There is early (compile-time or link time) binding of name to code fragment in conventional procedure calls.

Responsibilities

A fundamental concept in object oriented programming is to describe behavior in term of responsibilities. Chris's Request for action indicates only the desired outcome (flowers send to Robin) .Fred is free to pursue any technique that achieves the desired objective and in doing so will not be hampered by interference from Chris.

Class and instances

In above scenario Fred is florist we can use florist to represent the category (or class) of all florists. Which means Fred is instance (object) of class Florist.

All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of given class use the same method in response to similar messages.

Summary of Object-oriented Concepts

Alan Kay, considered by some to be the father of object-oriented programming, identified the following characteristics as fundamental to OOP.

1. Everything is an object.
2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving messages. A message is a request for action bundled with whatever arguments may be necessary to complete the task.
3. Each object has its own memory, which consists of other objects.
4. Every object is an instance of a class. A class simply represents a grouping of similar objects, such as integers or lists.
5. The class is the repository for behavior associated with an object. That is, all objects that are instances of the same class can perform the same actions.
6. Classes are organized into a singly rooted tree structure, called the inheritance hierarchy. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

Coping with Complexity

At early stages of computing most programs were written in assembly language by a single individual.

As programs become more complex, programmers found it difficult to remember all the information needed to know in order to develop or debug their software. such as

- Which values were contained in what registers?
- Did a new identifier name conflict with any previously defined name?
- What variables needed to be initialized before control could be transferred to another section of code?

The introduction of high level languages, such as FORTAN, COBOL and ALGOL solved some difficulties while simultaneously raising people's expectations of what a computer could do.

.

1. Non-linear Behavior of complexity

As programming projects became larger, an interesting phenomenon was observed that:

A task that would take one programmer 2 months to perform could not be accomplished by two programmers working for 1 month.

"The bearing of a child takes nine months, no matter how many women are assigned to the task".(refer to Freed Brook's book Mythical Man-Month)

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

The reason for this nonlinear behavior

- The interconnections between software components were complicated
- Large amounts of information had to be communicated among various members of the programming team.

Conventional techniques brings about high degree of interconnectedness.

This means the dependence of one portion of code on another portion (coupling)

So, a complete understanding of what is going on requires a knowledge of both portion of code we are considering and the code that uses it .An individual section of code cannot be understood in isolation.

2. Abstraction Mechanism

Abstraction mechanism is use to control complexity.

It is a ability to encapsulate and isolate design and execute information.

Alternatively, Abstraction is a mechanism to displaying only essential information and hiding the details.

Making use of abstraction

a)Procedure and function:

Procedure and function are main two first abstraction mechanism to be widely used in programming language. They allowed task that were executed repeatedly to be collected in one place and reused rather than being duplicated several times. In addition, procedure gave the first possibility for information hiding.

A set of procedure written by one programmer can be used by many other programmers without knowing the exact details of implementation. They needed only the necessary interface.

b) Block Scoping:

Nesting of function (one function inside another) to solve problem associated with procedure and function, the idea of block and scoping was introduced which permits functions to be nested.

This is also not truly the solution. If a data area is shared by two or more procedure, it must still be declared in more general scope than procedure. For example:

```
int sum()
{
int a,b;
.....
.....
int mul()
{
...
.....
}
}
```

Partially solved the abstraction mechanism

c) Modules:

A module is basically a mechanism that allows the programmers to encapsulate a collection of procedures and data and supply both import and export statement to control visibility feature from outside modules.

Parnas principle for creation and use of modules:

- One must provide the intended user with all the information needed to use the module correctly and nothing more.
- One must provide the implementor with all the information needed to complete the module and nothing more.

d) Abstract data type(ADT):

- To solve instantiation, the problem of what to do, your application needed more than one instance of software abstraction. The key to solve this problem is ADT.
- ADT is simply programmer defined data type that can be manipulated in a manner similar to system provided data types.
- This supported both information hiding as well as creating many instance of new data type.

But message passing is not possible in ADT.

e) Objects- Messages, Inheritance and Polymorphism:

OOP added new ideas to the concept of ADT.

- **Message passing:** Activity is initiated by a request to a specific object, not by invoking of a function.
- **Inheritance** allows different data types to share the same code, leading to a reduction in code size and an increase in functionality.
- **Polymorphism** allows this shared code to be designed to fit the specific circumstances of individual data types.

Computation As Simulation

- Explain Computation as Simulation?
- In Case of Object oriented Programming, Explain how do we have the view that computation is simulation? [PU:2013 Spring]

OOP framework is different from the traditional conventional behavior 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 behavior of a computer executing a program is a process-state 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 word 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 (Like in flower example).

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, 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.**

Power of metaphor

- When programmer think about problems in terms of behavior and responsibilities of objects, they bring with them a wealth of intuition, ideas and understanding from their everyday experience.
- When envisioned as pigeon holes, mailboxes, or slots containing values, there is a little in the programmers background to provide insight into how should be structured.

Avoiding Infinite Regression

Object cannot always respond to a message by politely asking another object to perform some action. The result would be an infinite circle of request ,like two gentle man each politely waiting for the another to go first before entering a doorway. Thus to avoid infinite regression at a time, at least few objects need to perform some work besides passing request to another agent.

Varieties of Classes

Based on different forms of responsibility classes can be used for many different purposes and be categorized as follows:

i) Data Manager (Data or State classes)

- Main responsibility is to maintain data or state information
- For example, in an abstraction of playing card, a major task for the class Card is simply maintain the data values that describe rank and suit.
- Often recognizable as nouns in problem description and are usually the fundamental building blocks of design.

ii) Data Sink or Data Sources

- These type of classes generate data, such as random number generator, or that accept data and then process them further, such as a class performing output to disk or file.
- These types of classes don't hold data for any period of time, but generates it on demand (for a data source) or processes it when called upon (for a data sink).

iii) View or Observer classes

- Most of the application is to display of information on an output device such as terminal screen.
- Because the code performing this activity is often complex, frequently modified and largely independent of the actual data being displayed, it is good programming practice to isolate display behavior in classes other than those that maintain the data being displayed.

iv) Facilitator or Helper Class

- These are classes that maintain little or no state information themselves but assist in the execution of complex tasks.
- For example, in displaying a playing card image we use the services of facilitator class that handles the drawings of lines and text on the display device. Another facilitator class might help maintain linked list of cards.

Previous Board Exam Questions from this chapter

- 1) What is object orientation? Explain the difference between structured and Object oriented Programming approach.**[PU:2006 spring]**
- 2) Why OOP is known as a new paradigm? Illustrate with certain examples. **[PU:2005 fall]**
- 3) What is class? Explain the different types of classes.**[PU:2005 fall]**
- 4) Describe Object Oriented Programming as a new paradigm in Computer programming field.**[PU:2015 Spring]**
- 5) What makes OOP a new paradigm? Explain your answer with suitable points. **[PU:2010 fall]**
- 6) What influence is an object oriented approach said to have on software system design? What is your own opinion ?Justify through example.**[PU:2009 fall]**
- 7) Explain the advantages of object oriented paradigm.
- 8) What are the Critical issues that are to be considered while designing the large Programming? Why? **[PU:2009 spring]**
- 9) Why Object oriented Programming is Superior than Procedural-Oriented Programming. Explain.**[PU:2016 fall]**
- 10) What are the main features of Object Oriented Programming . **[PU:2013 fall]**
- 11) What are the mechanism of data abstraction? Explain the difference between structured and Object Oriented Programming Approach?**[PU:2013 fall]**
- 12) What is the significance of forming abstractions while designing an object oriented system? In case of object oriented Programming, Explain how do we have view that computation is simulation? **[PU:2013 spring]**
- 13) What makes OOP better than POP. Explain with features of OOP. **[PU:2014 fall]**
- 14) With the help of object oriented Programming, explain how can object oriented Programming cope in solving complex problem. Explain computation as simulation.
[PU: 2014 spring][PU: 2018 fall]
- 15) How does making use of abstraction help in designing of an object oriented System. Explain with an example.**[PU:2015 fall]**
- 16) What is the use of abstraction mechanism in C++?Explain with example.**[PU: 2019 fall]**
- 17) Describe how object oriented Programming models the real word object problem with reference of agents , method, behavior and responsibilities.**[PU:2017 fall]**
- 18) What are the shortcoming of procedural Programming? Explain the notation of “Everything is an object” in an object oriented programming. **[PU:2017 spring]**
- 19) Explain the encapsulation and data abstraction.
- 20) Write a short notes on:
 - Abstraction **[PU:2006 spring]**
 - Non-linear behavior of Complexity **[PU :2014 fall] [PU:2015 spring] [PU:2009 spring]**

CHAPTER 2

Classes and Methods

Review of structures

- Structure is the collection of variables of different types under a single name for better visualization of problem.
- Structure can hold data of one or more types.

Declaring structure

The **struct** keyword defines a structure type followed by an identifier (name of the structure). Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

```
struct Person
{
    char name[50];
    int age;
    float salary;
};
```

Here a structure person is defined which has three members: name, age and salary.

Defining structure variable

Once you declare a structure person as above. You can define a structure variable as:

```
Person p1;
```

Here, a structure variable p1 is defined which is of type structure Person.

When structure variable is defined, only then the required memory is allocated by the compiler.

Considering having 16 bit system, the memory of float is 4 bytes, memory of int is 2 bytes and memory of char is 1 byte. Hence, 56 bytes of memory is allocated for structure variable p1.

How to access members of a structure?

The members of structure variable is accessed using a dot (.) operator.

Suppose, you want to access age of structure variable p1 and assign it 25 to it. You can perform this task by using following code below:

```
p1.age = 25;
```

Example: C++ Structure

C++ Program to assign data to members of a structure variable and display it.

```
#include <iostream>
using namespace std;
struct Person
{
    char name[50];
    int age;
    float salary;
};
int main()
{
    Person p1;
    cout << "Enter Full name: ";
    cin.get(p1.name, 50);
    cout << "Enter age: ";
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;
    cout << "Displaying Information." << endl;
    cout << "Name: " << p1.name << endl;
    cout << "Age: " << p1.age << endl;
    cout << "Salary: " << p1.salary;
    return 0;
}
```

Output

```
Enter Full name: Ram karki
```

```
Enter age: 24
```

```
Enter salary: 25000.35
```

```
Displaying Information.
```

```
Name: Ram karki
```

```
Age: 24
```

```
Salary: 25000.35
```

To read the text containing blank space, `cin.get` function can be used. This function takes two arguments.

First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.

Specifying a Class

A class serves as a blue print or a plan or a template for an object. It specifies what data and what functions will be included in objects of that class. Once a class has been defined, we can create any number of objects belonging to that class. An object is an instance of a class. A class binds the data and its associated functions together. It allows the data (and function) to be hidden, if necessary, from external use.

Example:

Class Car	Class Computer
Properties Company, Model Color, Capacity	Properties Brand, Price, Screen Resolution HDD size, RAM size
Action Speed(), Acceleration(), Break()	Action Processing(), Display(), Printing()

Declaration of class

Class declaration describes the type and scope of its members. The general form of class declaration is:

```
class class_name
{
private:
variable declaration;
function declaration;
public:
variable declaration;
function declaration;
};
```

Example:

```
class student
{
private:
char name[20];
int roll;
public:
void getinfo();
void display();
};
```

- Here, **class** is C++ keyword; **class_name** is name of class defined by user.
- The body of class enclosed within braces and terminated by a semicolon.
- The class body contains the declaration of variables and functions. The variables declared inside the class are known as data members and the functions are known as member functions. These functions and variables are collectively called as class members.
- Keyword private and public within class body are known as visibility labels. i.e. they specify which of the members are private and which of them are public.
- The class members that have been declared as private can be accessed only from within the class but public members can be accessed from outside of the class also. Using private declaration, data hiding is possible in C++ such that it will be safe from accidental manipulation.
- The use of keyword private is optional. By default, all the members are private. *Usually, the data within a class is private and the functions are public.*
- C++ provides a third visibility modifier, protected, which is used in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.
- **The binding of a data and functions together into a single class-type variable is referred to as encapsulation.**

```
#include<iostream>
using namespace std;
class student
{
private:
char name[20];
int roll;
public:
void getinfo()
{
cout<<"Enter name"<<endl;
cin>>name;
cout<<"Enter Rollno"<<endl;
cin>>roll;

}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll No:"<<roll<<endl;
}
};
```

```
int main()
{
student st;
st.getinfo();
st.display();
return 0;
}
```

Objects:

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when an object is created memory is allocated. To use the data and access functions defined in the class, we need to create objects.

Syntax for creating an object is

class_name object_name;

e.g.

Student st; // here Student is assumed to be a class name

The statement `student st;` creates a variable `st` of type `Student`. This variable `st` is known as object of the class `Student`. Thus, class variables are known as objects.

Key differences between a class and its objects

- Once we have defined a class, it exists all the time a program is running whereas objects may be created and destroyed at runtime.
- For single class there may be any number of objects.
- A class has unique name, attributes, and methods. An object has identity, state, and behavior.

Accessing the class member

The data member functions of class can be accessed using the **dot(.)** operator with the object. For example if the name of object is `st` and you want to access the member function with the name `getInfo()` then you will have to write `st.getInfo();`

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by **Access modifiers in C++**.

```

class Student
{
private:
int age;
public:
int rollno;
getInfo();
displayInfo();
.....
}
.....
int main()
{
Student st;
st.age=25; // error, age is private
st.rollno=5; //ok, roll number is public
st.getInfo(); //Ok
.....
}

```

Defining Member functions

Member function can be defined in two places:

- Outside the class definition
- Inside the class definition

Defining Member function inside the class definition	Defining Member function Outside the class definition
In this case, a member function is defined at the time of its declaration. The function declaration is replaced by the function definition inside the class body.	<p>In this technique, only declaration is done within class body. The member function that has been declared inside a class has to be defined separately outside the class. The syntax for defining the member function is as</p> <pre> return_type class_name::function_name(argument_list) { //function body }</pre> <p>Here, membership label class-name :: tells the compiler that the function function_name belongs to the class class-name. The symbol :: is called scope resolution operator.</p>

<pre>class Item { private: int number; float cost; public: void getdata() { } void display() { } };</pre>	<pre>class Item { private: int number; float cost; public: void getdata(); void display(); }; void Item::getdata() { } void Item::display() {</pre>
<pre>#include<iostream> using namespace std; class Item { private: int number; float cost; public: void getdata() { cout<<"Enter the Item number"<<endl; cin>>number; cout<<"Enter the cost of item"<<endl; cin>>cost; } void display() { cout<<"Item number ="<<number<<endl; cout<<"Cost of item ="<<cost<<endl; } };</pre>	<pre>#include<iostream> using namespace std; class Item { private: int number; float cost; public: void getdata(); void display(); }; void Item::getdata() { cout<<"Enter the Item number"<<endl; cin>>number; cout<<"Enter the cost of item"<<endl; cin>>cost; } void Item::display() { cout<<"Item number ="<<number<<endl; cout<<"Cost of item ="<<cost<<endl; }</pre>

<pre>int main() { Item x; x.getdata(); x.display(); return 0; }</pre>	<pre>int main() { Item x; x.getdata(); x.display(); return 0; }</pre>
<p>In this example, the functions getdata() and putdata() are defined within body of class item.</p>	<p>In this example, getdata() and putdata() functions are member functions of class item. They are declared within class body and they are defined outside the class body. While defining member functions, item:: (membership identity label) specifies that member functions belong to the class item. Other are similar to normal function definition.</p>

Some characteristics of the member functions

- Several different classes can use the same function name (function overloading). The membership level will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so.
- A member function can call another member function directly, without using the dot operator.
- A private member function cannot be called by other function that is not a member of its class.

Nesting of member function

A member function can be called by using its name inside another member function of the same class is known as nesting of member functions.

Example:

```
#include<iostream>
using namespace std;
class Largest
{
private:
int a,b;
public:
void getnumber();
int compare();
void display();
};
void Largest::getnumber()
{
cout<<"Enter two number"<<endl;
cin>>a>>b;
}
int Largest::compare()
{
if (a>b)
return a;
else
return b;
}
void Largest::display()
{
cout<<"Largest number="<<compare();
}
int main()
{
Largest l;
l.getnumber();
l.display();
return 0;
}
```

Assignment:

Write a program that will demonstrate nested member function. The program that uses the following properties - class name- Addnumber, integer type private member variable- a, b; three member function -inputnumber(), int add(), show(). add() function will add the inputted value a, b like $\text{int } x=a+b$. show() function will call the add() function as a nested function and will display the addition value y as output.

Accessing private member function

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

A private member function can only be called by another function that is member of its class. Even an object cannot invoke a private function using the dot operator.

Consider the class defined below

```
class sample
{
private:
    int m;
void read(); // private member function
public:
    void update();
};
```

If s1 is an object of sample, then

```
s1.read(); // Won't work ;
```

Object cannot access private members. However, the function read() can be called by the function update() to update the value of m.

```
void sample::update()
{
    read(); // simple call ;no object used
}
```

Example

```
#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 updation="<<m<<endl;
}

int main()
{
    sample s1;
    s1.update();
    s1.display();
    return 0;
}

```

Assignment

Write a program that will demonstrate private member function. The program that uses the following properties - class name- sum, integer type private member variable- x, y. One private member function getnumber(), two public member function - int addition(), display(). addition() function will call private member function getnumber() for input number x, y and add the inputted value x, y like int z=x+y. display() function will display the addition value z as output.

Array of object

An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of object. As we know, an array is a collection of similar date type, we can also have arrays of variables of type class. Such variables are called array of object. First we define a class and then array of objects are declared. An array of objects is stored inside the memory in the same way as multidimensional array.

```

#include<iostream>
using namespace std;
class Employee
{
    char name[25];
    int age;
    float salary;
public:
    void getdata();
    void display();
};

```

```

void Employee::getdata()
{
    cout<<"Enter Employee Name:";
    cin>>name;
    cout<<"Enter Employee Age :";
    cin>>age;
    cout<<"Enter Employee Salary:";
    cin>>salary;
}
void Employee::display()
{
    cout<<"Name:"<<name<<endl;
    cout<<"Age:"<<age<<endl;
    cout<<"Salary:"<<salary<<endl;
}
int main()
{
    int i;
    Employee e[10];      //Creating Array of 10 employees
    for(i=0;i<10;i++)
    {
        cout<<"Enter details of "<<i+1<<" Employee"<<endl;
        e[i].getdata();
    }
    for(i=0;i<10;i++)
    {
        cout<<"Details of Employee"<<i+1<<endl;
        e[i].display();
    }
    return 0;
}

```

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

Student
Name
Roll
Address
Percentage
input()
display()

input():to input initial values

display():to display the record of students who passed

Note:-45% is pass percentage

#include<iostream>

using namespace std;

```

class student
{
private:
char name[20],address[20];
int roll;
float per;
public:
void input()
{
cout<<"Enter the name";
cin>>name;
cout<<"Enter the address";
cin>>address;
cout<<"Enter the roll";
cin>>roll;
cout<<"Enter the percentage";
cin>>per;
}
void display()
{
if(per>=45)
{
cout<<"Name="<<name<<endl;
cout<<"Roll="<<roll<<endl;
cout<<"Address="<<address<<endl;
cout<<"Percentage="<<per<<endl;
}
}
};

int main()
{
int i;
student st[5];
for(i=0;i<5;i++)
{
cout<<"Enter the information of student"<<i+1<<endl;
st[i].input();
}
for(i=0;i<5;i++)
{
st[i].display();
}
return 0;
}

```

Object as function arguments

Like any other data type, an object can be used as function argument.

This can be done in two ways:

- A copy of the entire object is passed to the function (*pass-by-value*)
- Only the address of the object is transferred to the function(*pass-by-reference*)

Let us illustrate use of objects as function arguments with the help of program which performs the addition of time in the hour and minute format.

Program to perform addition of time in hours and minutes format by passing object as argument

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

```

int main()
{
time t1,t2,t3;
t1.gettime(2,45);
t2.gettime(3,30);
t3.sum(t1,t2);
cout<<"First time=";
t1.display();
cout<<"Second time=";
t2.display();
cout<<"Total time=";
t3.display();
return 0;
}

```

Since the member function sum () is invoked by the object t3, with the object t1 and t2 as arguments, it can directly access the hours and minutes variables of t3. But, the member of t1and t2 can be accessed by using the dot operator (like t1.hours and t1.minutes). Therefore, inside the function sum(), the variable hours and minutes refers to t3, t1.hours and t1.minutes refers to t1 and t2.hours and t2.minutes refers to t2.

Output:

*First time=2 Hours and 45 Minutes
 Second time=3 Hours and 30 Minutes
 Total time=6 Hours and 15 Minutes*

Another Example

Program to perform addition of two complex numbers by passing object as arguments.

```

#include<iostream>
#include<conio.h>
using namespace std;
class complex
{
private:
int real, imag;
public:
void getcomplex();
void addcomplex(complex, complex);
void display();
};

```

```

void complex:: getcomplex()
{
cout<<"Enter real part:"<<endl;
cin>>real;
cout<<"Enter imaginary part:"<<endl;
cin>>imag;
}
void complex::display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
void complex::addcomplex( complex c1, complex c2)
{
real=c1.real+c2.real;
imag=c1.imag+c2.imag;
}
int main()
{
complex c1,c2,result;
cout<<"For first complex number"<<endl;
c1.getcomplex();
cout<<"For second complex number"<<endl;
c2.getcomplex();
result.addcomplex(c1,c2); //objects first and second passed as argument
cout<<"sum of two complex number=";
result.display();
getch();
}

```

Output:

```

For first complex number
Enter real part:
2
Enter imaginary Part:
4
For second complex number
Enter real part:
1
Enter imaginary Part:
8
Sum of two complex number=3+12i

```

Returning Object as Arguments

A function cannot only receive object as arguments but also return them. The example in program below illustrates how an object can be created (within function) and returned to another function.

Addition of complex number by returning object as argument

```
#include<iostream>
using namespace std;
class complex
{
private:
int real;
int imag;
public:
void getdata();
complex addcomplex(complex,complex);
void display();
};

void complex::getdata()
{
cout<<"Enter the real part:"<<endl;
cin>>real;
cout<<"Enter the imaginary part :"<<endl;
cin>>imag;
}

void complex::display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
complex complex::addcomplex( complex c1, complex c2)
{
complex temp;
temp.real=c1.real+c2.real;
temp.imag=c1.imag+c2.imag;
return temp;
}
```

```

int main()
{
complex c1,c2,c3,result;
cout<<"For first complex number"<<endl;
c1.getdata();
cout<<"For second complex number"<<endl;
c2.getdata();
result=c3.addcomplex(c1,c2);
cout<<"sum of two complex number=";
result.display();
return 0;
}

```

What changes need to be done above program if we change the function call statement as `c3=c1.addcomplex(c2)`

```

#include<iostream>
using namespace std;
class complex
{
private:
int real;
int imag;
public:
void getdata();
complex addcomplex(complex);
void display();
};
void complex::getdata()
{
cout<<"Enter the real part:"<<endl;
cin>>real;
cout<<"Enter the imaginary part :"<<endl;
cin>>imag;
}
void complex::display()
{
cout<<real<<"+ "<<imag<<"i"<<endl;
}
complex complex::addcomplex( complex c2)
{
complex temp;
temp.real=real+c2.real;
temp.imag=imag+c2.imag;
return temp;
}

```

```

int main()
{
complex c1,c2,c3;
cout<<"For first complex number"<<endl;
c1.getdata();
cout<<"For second complex number"<<endl;
c2.getdata();
c3=c1.addcomplex(c2);
cout<<"sum of two complex number=";
c3.display();
return 0;
}

```

WAP to perform the addition of distance with feet and inches returning object as argument.

```

#include<iostream>
using namespace std;
class dist
{
private:
int feet;
int inch;
public:
void getdata();
dist add(dist,dist);
void display();
};

void dist::getdata()
{
cout<<"Enter feet:"<<endl;
cin>>feet;
cout<<"Enter inch:"<<endl;
cin>>inch;
}
void dist::display()
{
cout<<feet<<"feet and "<<inch<<"inches"<<endl;
}

```

```

dist dist::add(dist d1, dist d2)
{
dist sum;
sum.inch=d1.inch+d2.inch;
sum.feet=sum.inch/12;
sum.inch=sum.inch%12;
sum.feet=sum.feet+d1.feet+d2.feet;
return sum;
}

int main()
{
dist d1,d2,d3,result;
cout<<"Enter information of first distance"<<endl;
d1.getdata();
cout<<"Enter information of second distance"<<endl;
d2.getdata();
result=d3.add(d1,d2);
cout<<"sum of distance=";
result.display();
return 0;
}

```

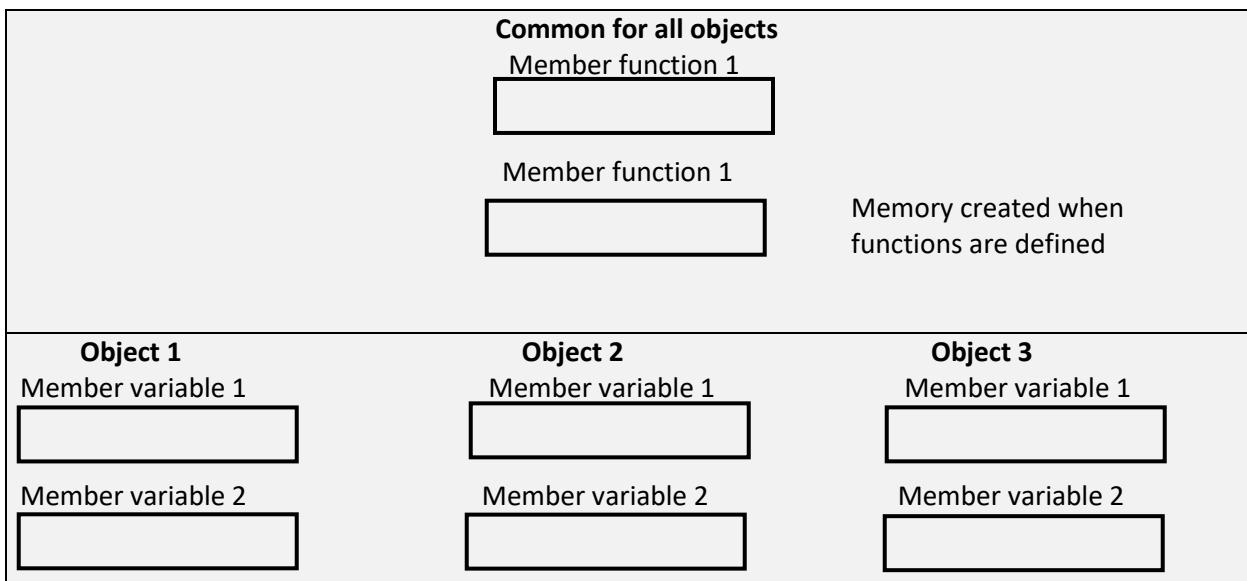
Assignment:

- WAP to perform the addition of time in hours, minutes and seconds format.
- WAP to perform the addition of time using the concept of returning object as argument.
- WAP to create two distance objects with data members feet, inches and a function call by one object passing second object as function argument and return third object adding two objects. *Hint:d3=d1.adddistance(d2);*

Memory allocation for objects

For each object memory spaces for member variables is allocated separately, because the member variables will hold different data value for different objects.

The member functions are created and placed in the memory space only once when they are defined as a part of class specification. Since, all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when objects are created.



Friend function

The private member of a class cannot be accessed from outside the class i.e. a nonmember function cannot access to the private data of a class. But may be in some situation one class wants to access private data of second class and second wants to access private data of first class, or may be an outside function wants to access the private data of a class. However, we can achieve this by using friend functions.

The non-member function that is “friendly” to a class, has full access rights to the private members of the class.

To make an outside function friendly to a class, we have to declare this function as a friend function. Declaration of friend function should be preceded by the keyword **friend**.

The friend function declares as follows:

```
class ABC
{
-----
public :
-----
friend void xyz(void); //declaration
};
```

We give a keyword **friend** in front of any members function to make it friendly.

```

void xyz(void) //function definition
{
//function body
}

```

It should be noted that a function definition does not use friend keyword or scope resolution operator (::). A function can be declared as friend in any number of classes.

Characteristics of a Friend Function:

- It is not in the scope of the class to which it has been declared as friend. That is why, it cannot be called using object of that class.
- It can be invoked like a normal function without the help of any object.
- It cannot access member names (member data) directly and has to use an object name and dot membership operator with each member name.(eg.A.x)
- It can be declared in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments

Program to illustrate the use of friend function

```

#include <iostream>

using namespace std;
class sample
{
private:
int a;
int b;
public:
void setvalue( )
{
a=25 ;
b=40 ;
}
friend float mean (sample s) ;
};
float mean (sample s)
{
return float(s.a+s.b)/2 ;
}
int main( )
{
sample x ;
x.setvalue( );
cout<<"Mean value=<<mean(x)<<endl;
return 0 ;
}

```

Create classes called class1 and class2 with each of having one private member .Add member function to set a value(say setvalue) one each class. Add one more function max() that is friendly to both classes. max() function should compare two private member of two classes and show maximum among them. Create one-one object of each class and then set a value on them. Display the maximum number among them.[PU:2015 fall,2016 fall]

```
#include <iostream>
using namespace std;
class class2;
class class1
{
private:
int x;
public:
void setvalue (int num)
{
x=num ;
}
friend void max (class1, class2);
};

class class2
{
private:
int y ;
public:
void setvalue(int num)
{
y=num ;
}
friend void max(class1, class2) ;
};

void max(class1 m, class2 n)
{
if (m.x>=n.y)
cout<<"Maximum value="<<m.x ;
else
cout<<"Maximum value="<<n.y ;
}
```

```
int main( )
{
class1 p;
class2 q;
p.setvalue(10) ;
q.setvalue(20) ;
max(p, q);
return 0;
}
```

WAP using friend function to Add Data Objects of two Different Classes

OR

Addition of two private data of different classes using friend fuction.

```
#include <iostream>
using namespace std;
class ABC;      // Forward declaration
class XYZ
{
private:
int x;
public:
void setdata (int num)
{
x=num ;
}
friend void add (XYZ,ABC);
};
class ABC
{
private:
int y ;
public:
void setdata (int num)
{
y=num ;
}
friend void add(XYZ,ABC) ;
};
```

```

void add (XYZ m, ABC n)      // Definition of friend
{
cout<<"Sum ="<<(m.x+n.y);
}
int main( )
{
XYZ p;
ABC q;
p.setdata(15) ;
q.setdata(20) ;
add(p,q);
return 0;
}

```

Output:

Sum=35

Alternative solution

```

#include <iostream>
using namespace std;
class ABC; // Forward declaration
class XYZ
{
public:
int x;
void getdata ()
{
cout<<"Enter Value of class XYZ"<<endl;
cin>>x;
}
friend void add (XYZ,ABC);
};
class ABC
{
int y;
public:
void getdata ()
{
cout<<"Enter the value of class ABC"<<endl;
cin>>y;
}
friend void add(XYZ,ABC) ;
};

```

```

void add (XYZ m, ABC n)
{
cout<<"Sum ="<<(m.x+n.y);
}
int main( )
{
XYZ p;
ABC q;
p.getdata();
q.getdata();
add(p,q);
return 0;
}

```

Swapping Private data of classes

```

#include <iostream>
using namespace std;
class ABC; // Forward declaration
class XYZ
{
private:
int x;
public:
void getdata ()
{
cout<<"Enter Value of class XYZ"<<endl;
cin>>x;
}
void display()
{
cout<<"value1="<<x<<endl;
}
friend void swap (XYZ &,ABC &);
};
class ABC
{
private:
int y ;
public:
void getdata ()
{
cout<<"Enter the value of class ABC"<<endl;
cin>>y;
}

```

```

void display()
{
cout<<"value2="<<y<<endl;
}
friend void swap(XYZ &,ABC & );
};
void swap (XYZ &m, ABC &n) // Definition of friend
{
int temp;
temp=m.x;
m.x=n.y;
n.y=temp;
}

int main( )
{
XYZ p;
ABC q;
p.getdata();
q.getdata();
cout<<"Value before swapping"<<endl;
p.display();
q.display();
swap(p,q);
cout<<"Value after swapping"<<endl;
p.display();
q.display();
return 0;
}

```

Friend class

We can declare all the member functions of one class as the friend functions of another class. In such cases, the class is called friend class. A friend class can use all the data member of a class for which it is friend. For example

```

class B;
class A
{
.....
friend class B;
};

```

Here, class B can use all the data member of class A.

Example:

```
#include<iostream>
using namespace std;
class B; //forward declaration
class A
{
int x,y;
public:
void getdata()
{
cout<<"Enter the value of x and y:"<<endl;
cin>>x>>y;
}
friend class B;
};

class B
{
int z;
public:
void getdata()
{
cout<<"Enter the value of z:";
cin>>z;
}
void sum(A t)
{
cout<<"Sum of x,y and z ="<<(t.x+t.y+z)<<endl;
}
};

int main()
{
A p;
B q;
p.getdata();
q.getdata();
q.sum(p);
return 0;
}
```

Inline functions

When function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving register, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code.

A function is made inline by using a keyword **inline** before the function definition. Inline functions must be defined before they are called.

Example

```
#include<iostream>
#include<conio.h>
using namespace std;
inline int max(int a, int b)
{
    return (a>b)?a:b;
}
int main()
{
    int x,y;
    cout<<"Enter x and y:"<<endl;
    cin>>x>>y;
    cout<<"Maximum value is:"<<max(x,y);
    getch();
    return 0;
}
```

Note:

Some of the situations where inline expansion may not work are

- For functions returning values, if a loop, a switch or a goto exists.
- For functions not returning values, if a return statement exists.
- If function contain static variables.
- If inline functions are recursive.

Static data member

A data member of class can be qualified as static using the prefix static. A static member variable has following characteristics. They are:

- Static member variable of a class is **initialized to zero** when the first object of its class is created.
- **Only one copy** of that member is created for the entire class and is shared by all the objects of that class, no matter how many object are created.
- It is **visible only within the class** but its **life time is the entire program**.

The type and scope of each static member variable must be defined outside the class definition.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects.

Example:

```
#include <iostream>
using namespace std;
class item
{
    static int count ; // static member variable
    int number ;
public:
    void getdata (int a)
    {
        number=a;
        count++ ;
    }
    void displaycount()
    {
        cout<<"count:"<<count<<endl;
    }
};
int item ::count; //static member definition
```

```

int main()
{
item x,y,z ; //count is initialized to zero
x.displaycount() ; // display count
y.displaycount();
z.displaycount();
x.getdata(10) ; //getting data into object x
y.getdata(20) ; //getting data into object y
z.getdata(30) ; //getting data into object z
cout<<"After reading data"<<endl ;
x.displaycount() ; // display count
y.displaycount();
z.displaycount();
return 0;
}

```

Output:

```

Count:0
Count:0
Count:0
After reading data
Count:3
Count:3
Count:3

```

Static Member functions

Like static member variables, we can also have static member functions.

A member function that is declared static has the following properties.

- A static function can have access to only other static members (functions or variables) declared in the same class.
- A static member function can be called using the class name(instead of objects) as follows.
class-name::function name;

```

#include<iostream>
using namespace std;
class test
{
int code;
static int count;
public:
void setcode()
{
code=++count;
}
void showcode()
{
cout<<"code="<<code<<endl;
}
static void showcount()
{
cout<<"count="<<count<<endl;
}
};
int test::count;
int main()
{
    test t1,t2;
    t1.setcode();
    t2.setcode();
    test::showcount();
    test t3;
    t3.setcode();
    test::showcount();
    t1.showcode();
    t2.showcode();
    t3.showcode();
    return 0;
}

```

Output:

```

Count=2
Count=3
Code=1
Code=2
Code=3

```

Reference Variable

A reference variable provides an alias(alternative name) for a previously defined variable.
A reference variable can be created as follows:

data-type &reference-name = variable name;

eg float total = 100;
float &sum = total; // creating reference variable for ‘total’.

Example:

```
#include<iostream>
using namespace std;
int main()
{
    float total=100;
    float &sum=total;
    cout<<"Total="<<total<<endl;
    cout<<"Sum="<<sum<<endl;
    total=total+100;
    cout<<"Total="<<total<<endl;
    cout<<"Sum="<<sum<<endl;
    return 0;
}
```

Output:

```
Total=100
Sum=100
Total=200
Sum=200
```

In the above example, we are creating a reference variable ‘sum’ for an existing variable ‘total’. Now these can be used interchangeably. Both of these names refer to same data object in memory. If the value is manipulated and changed using one name then it will change for another also. Eg- the statement

total = total + 100; will change value of ‘total’ to 200. And it will also change for ‘sum’. So the statements

*cout<<total;
cout<<sum; both will print 200. This is because both the variables use same data object in memory.*

- A reference variable must be initialized at the time of declaration, since this will establish correspondence between the reference and the data object which it names.
- The symbol & is not an address operator here. The notation float & means reference to float type data.

Major application of reference variable is in passing arguments to function. Consider the following example.

```
#include<iostream>
using namespace std;
void fun(int &x);
int main()
{
int m;
m=10;
fun(m);
cout<<"Updated value=<<m;
return 0;
}
void fun(int &x)
{
x=x+10;
}
```

When the function call fun(m) is executed, the following initialization occurs:

```
int &x=m;
```

Thus x becomes an alias of m after executing the statement.

```
fun(m);
```

such function calls are known as call by reference. Since the variables x and m are aliases, when the function increments x, m is also incremented. The value of m becomes 20 after the function is executed.

Default argument

C++ allows a function to assign a parameter the default value in case no argument for that parameter is specified in the function call. Default values are specified when the function is declared. Default value is specified in a manner syntactically similar to a variable initialization. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

For example:

```
float amount(float principal, int period, float rate=0.15);
```

The above prototype declares a default value of 0.15 to the argument rate.

A subsequent function call like

```
value=amount(5000,7); // one argument missing
```

Passes the value 5000 to principal and 7 to period and then lets a function use default value of 0.15 for rate,

The call

```
Value=amount(5000,5,0.12); // no missing argument
```

Passes the explicit value of 0.12 to rate.

Note:

- Default argument are useful in situations where some arguments always have the same value. Eg. bank interest may remain the same for all customers for a particular period of deposit.
- Only the trailing arguments can have default values and therefore we must add defaults from right to left.
- We cannot provide a default value to a particular argument in the middle of an argument list.

```
int add (int a, int b = 5, int c); // illegal  
int add (int a = 5, int b, int c = 6); // illegal  
int add (int a , int b , int c = 5); // legal  
int add (int a = 5, int b = 6, int c = 7); // legal
```

Example:

```
#include<iostream>
#include<conio.h>
using namespace std;
int add(int a=1, int b=2, int c=3);
int main()
{
    int x=5,y=10,z=15;
    cout<<"Sum="<<add(x,y,z)<<endl;
    cout<<"Sum="<<add(x,y)<<endl;
    cout<<"Sum="<<add(x)<<endl;
    cout<<"Sum="<<add()<<endl;
    getch();
    return 0;
}
int add(int a, int b, int c)
{
    return (a+b+c);
}
```

Output:

```
Sum=30
Sum =18
Sum =10
Sum =6
```

State and Behavior of Object

State

- State tells us about the type or the value of that object.
- It tells, “**What the objects have?**”
Example: Student have a first name, last name, age, etc...
- An objects state is defined by the attributes (i.e. data members or variables) of the object.
- In OOP state is defined as data members.
- It is determined by the values of its attributes.

Behavior

- Behavior tells us about the operations or things that the object can perform.
- It tells “**What the objects do**”
- Example Student attend a course “OOP”, “C programming” etc...
- An objects behavior is defined by the methods or action (i.e. Member functions) of the object.
- In OOP behaviors are defined as member function
- It determines the actions of an object.

Let us consider another example

Imagine a dog. This is an example of an object.

A state can be on or off. If it's asleep, its state is off. If it's awake, the state is on.

Behavior: If the dog is awake, what is it doing? Examples of behavior might be barking, sniffing, eating or drinking, pawing, jumping, playing, to name a few.

Responsibility of Object

An object must contain the data (attributes) and code (methods) necessary to perform any and all services that are required by the object. This means that the object must have the capability to perform required services itself or at least know how to find and invoke these services.

Rather than attempt to further refine the definition, take a look at an example that illustrates this responsibility concept.

Consider standalone application when looking for an example to illustrate object responsibility. One of my favorite examples is that of a generic paint program that allows a user to select a shape from a pallet and then drag it to a canvas, where the shape is dropped and displayed. From the user's perspective, this activity is accomplished by a variety of mouse actions. First, the user hovers over a specific shape with the mouse (perhaps a circle), right clicks, drags the shape to a location on the canvas and then lifts his/her finger off the mouse.

This last action causes the shape to be placed at the desired location. In fact, what is really happening when our user's finger is lifted from the mouse button? We can use this action to illustrate our object responsibility concept.

Let's assume that when the finger is lifted from the mouse button that the resulting message from this event is simply: draw. Consider the mouse as an object and that the mouse is only responsible for itself as well. The mouse can, and should only do mouse things. In this context, when the finger is lifted from the mouse button, the mouse thing to do is to draw. But draw what? Actually, the mouse doesn't care. It is not the mouse's responsibility to know how to draw anything. Its responsibility is to signal when to draw, not what to draw.

If this is the case (remember that a circle was selected), how does the right thing get drawn? The answer is actually pretty straightforward; the circle knows how to draw itself. The circle is responsible for doing circle things—just like the mouse is responsible for doing mouse things. So, if you select a circle, the circle object that is created knows everything possible about how to use a circle in the application.

This model has a major implication. Because the mouse does not need to know anything about specific shape objects, all it needs to do to draw a shape is to send the message draw. Actually, this will take the form of a method called draw(). Thus, more shapes can be added without having to change any of the mouse's behavior. The only constraint is that any shape object installed in the application must implement a draw() method. If there is no draw() method, an exception will be generated when the mouse mouse-up action attempts to invoke the object's non-existent draw() method. In this design methodology, there is a de facto contract between the mouse class and the shape class and these contracts are made possible by the powerful object-oriented design technique called polymorphism.

Previous Board Exam Questions

- 1) Declare a C++ structure (Program) to contain the following piece of information about cars on a used car lot: [PU:2013 spring]
 - i. Manufacturer of the car
 - ii. Model name of the car
 - iii. The asking price for the car
 - iv. The number of miles on odometer
- 2) Differentiate between class and structure. Explain them with example.[PU:2010 spring]
- 3) What sorts of shortcomings of structure are addressed by classes? Explain giving appropriate example.[PU:2014 fall]
- 4) Differentiate between structure and class. Why Class is preferred over structure? Support your answer with suitable examples.[PU:2016 fall]
- 5) Explain the various access specifiers used in C++ with an example.
[PU:2010 fall][PU:2016 spring]
- 6) What is information hiding? What are the access mode available in C++ to implement different levels of visibility? Explain through example.[PU:2014 spring][PU:2016 fall]

- 7) What is data hiding? How do you achieve data hiding in C++? Explain with suitable program. **[PU:2019 fall]**
- 8) What is encapsulation? How can encapsulation enforced in C++? Explain with suitable example code. **[PU:2017 spring]**
- 9) What are the common typed of function available in C++? Define the 3 common types of functions in C++ with a program. **[PU:2015 spring]**
- 10) What is a function? Discuss the use of friend function taking into consideration the concept of data hiding in object oriented programming. **[PU:2009 spring]**
- 11) Does friend function violate the data hiding? Explain briefly. **[PU:2017 fall]**
- 12) "Friend function breaches the encapsulation." Justify. Also mention the use of friend function. **[PU:2015 spring]**
- 13) Where do you use friend function? **[PU:2014 spring]**
- 14) What are the merits and demerits of friend function? **[PU:2009 fall]**
- 15) Private data and function of a class cannot be accessed from outside function. Explain how is it possible to access them with reference of an example. **[PU:2018 fall]**
- 16) What are the advantages of using friend function? List different types of classes and explain any two. **[PU:2010 spring]**
- 17) What are the advantages and disadvantages of using friend function? Explain with example program. **[PU:2018 fall]**
- 18) What is inline function? Explain its importance with the help of example program. **[PU:2015 fall]**
- 19) What is the role of static data in C++ classes? Give example. **[PU:2006 spring]**
- 20) What do you mean by static member of a class? Explain the characteristic of static data member. **[PU:2013 fall][PU:2017 fall]**
- 21) When and how do we make use of static data members of a class? Differentiate between virtual functions, friend functions and static member functions. **[PU:2013 spring]**
- 22) What are the static data member and static member functions? Show their significance giving examples. **[PU:2014 fall]**
- 23) Write short notes on:
 - Friend function
[PU:2006 spring][PU:2013 spring] [PU:2017 spring][PU:2005 fall]
 - Inline function**[PU:2009 fall][PU:2010 spring][PU:2013 spring]**
 - Reference variable
 - Default argument

Programs

1. Declare a C++ structure (Program) to contain the following piece of information about cars on used car lot. [PU:2013 spring]
 - i. Manufacturer of the car
 - ii. Model name of the car
 - iii. The asking price of car
 - iv. The number of miles on odometer
2. Create a class called Employee with three data members (empno , name, address), a function called readdata() to take in the details of the employee from the user, and a function called displaydata() to display the details of the employee. In main, create two objects of the class Employee and for each object call the readdata() and the displaydata() functions. [PU:2005 fall]
3. Create a class called student with three data members (stdnt_name[20],faculty[20],roll_no), a function called readdata() to take the details of the students from the user and a function called displaydata() to display the details of the students. In main, create two objects of the class student and for each object call both of the functions. [PU:2010 fall]
4. Modify the **Que.no 2** for 20 students using array of object.
5. WAP to perform the addition of time in hours, minutes and seconds format.
6. WAP to perform the addition of time using the concept of returning object as argument.
7. WAP to create two distance objects with data members feet, inches and a function call by one object passing second object as function argument and return third object adding two objects. *Hint:d3=d1.adddistance(d2);*
8. Create a class called Rational having data members nume and deno and using friend function find which one is greater.
9. WAP to add the private data of three different classes using friend function.
10. Write a program to find the largest of four integers .your program should have three classes and each classes have one integer number.[PU:2014 spring]
11. WAP to swap the contents of two variables of 2 different classes using friend function.
12. WAP to add two complex numbers of two different classes using friend function.
13. WAP to add complex numbers of two different classes using friend class.
14. Using class write a program that receives inputs principle amount, time and rate. Keeping rate 8% as the default argument, calculate simple interest for three customers.[PU:2019 fall]

15. Create a new class named City that will have two member variables CityName (char[20]), and DistFromKtm (float). Add member functions to set and retrieve the CityName and DistanceFromKtm separately. Add new member function AddDistance that takes two arguments of class City and returns the sum of DistFromKtm of two arguments. In the main function, Initialize three city objects . Set the first and second City to be pokhara and Dhangadi. Display the sum ofDistFromKtm of Pokhara and Dhangadi calling AddDistance function of third City object. **[PU: 2010 Spring]**

16. Create a class called Volume that uses three Variables (length, width, height) of type distance (feet and inches) to model the volume of a room. Read the three dimensions of the room and calculate the volume it represent, and print out the result .The volume should be in (feet3) form ie. you will have to convert each dimension into the feet and fraction of foot. For instance , the length 12 feet 6 inches will be 12.5 ft)

[PU: 2009 spring]

17. WAP to read two complex numbers and a function that calls by passing references of two objects rather than values of objects and add into third object and returns that object.

CHAPTER 3

Message, instance and initialization

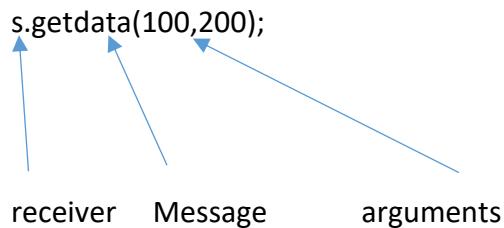
Message Passing

Message passing (sometimes also called method lookup) means the dynamic process of asking an object to perform a specific action.

- A message is always given to some object, called the receiver.
- The action performed in response to the message is not fixed but may be differ, depending on the class of the receiver .That is different objects may accept the same message the and yet perform different actions.

There are three identifiable parts to any message-passing expression. These are

- 1) **Receiver:** the object to which the message is being sent
- 2) **Message selector:** the text that indicates the particular message is being sent.
- 3) **Arguments** used in responding the message.



Example of message passing

```
#include<iostream>
#include<conio.h>
using namespace std;
class student
{
int roll;
public:
void getdata(int x)
{
roll=x;
}
void display()
{
cout<<"Roll number="<<roll;
}
};
```

```

int main()
{
student s;
s.getdata(325); //objects passing message
s.display(); //objects passing message
getch();
return 0;
}

```

Message passing vs procedure calls

The message passing and procedure calls are differ in following aspects.

Message passing

- In a message passing there is a designated receiver for that message; the receiver is some object to which message is sent.
- Interpretation of the message (that is method is used to respond the message) is determined by the receiver and can vary with different receivers. That is, different object receive the same message and yet perform different actions.
- Usually, the specific receiver for any given message will not be known until run time, so determination of which method cannot be made until then. Thus, we say there is a late binding between the message (function or procedure name) and the code fragment (method) used to respond to the message.

Procedure calls

- In a procedure call, there is no designated receiver.
- Determination of which method to invoke is very early (compile-time or link time) binding of a name to fragment in procedure calls.

Constructor

- A constructor is a special member function which initializes the objects of its class. It is called special because its name is same as that of the class name.
- The constructor is automatically executed whenever an object is created. Thus, a constructor helps to initialize the objects without making a separates call to a member function.
- It is called constructor because it constructs values of data members of a class.

Characteristics of constructor

- A constructor has same name as class name.
- They are invoked automatically when the objects are created.
- They should be declared in the public section.
- They do not have return types.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make ‘implicit calls’ to the new and delete operators when a memory allocation is required.

Types of Constructor:

1. Default constructor

- A constructor that accepts no arguments (parameters) is called the default constructor.
- If a class does not include any constructor, the compiler supplies a default constructor.

The default Constructor is declared and defined as follows:

```
class Complex
{
private:
int real,imag;
public:
Complex
{
real=0;
imag=0;
}
.....
.....
};
```

When a class contains a constructor like the defined above, it is guaranteed that an object by class will be initialized automatically. For example, the declaration

Complex c1; not only creates the object **c1** of type **Complex** but also initializes its data members real and imag to zero. There is no need to write any statement to invoke the constructor function (as we do with normal member functions).

Program:

```
#include <iostream>
using namespace std;
class Complex
{
private:
int real,imag ;
public:
Complex()
{
    real=0;
    imag=0;
}
void display( )
{
    cout<<"Real part="<<real<<endl;
    cout<<"Imaginary part="<<imag<<endl ;
}
};
int main( )
{
Complex c1;
c1.display();
return 0;
}
```

Output:

```
Real part=0
Imaginary Part=0
```

2. Parameterized Constructor

The constructor that can take arguments are called parameterized constructor.

Arguments are passed when the objects are created. This can be done in two ways.

- by calling the constructor explicitly
- by calling the constructor implicitly

For example, the constructor used in the following example is the parameterized constructor and takes two arguments both of type int.

```
class Complex
{
private:
int real,imag;
public:
Complex (int r, int i) //parameterized constructor
{
real=r;
imag=i;
}
.....
.....
};
int main()
{
Complex c1(5,2); //implicit call
Complex c2 =Complex(7,4); //explicit call
.....
}
```

Example:

```
#include <iostream>
using namespace std;
class Complex
{
private:
int real,imag;
public:
Complex(int r,int i)
{
    real=r;
    imag=i;
}
void display( )
{
cout<<"Real part="<<real<<endl;
cout<<"Imaginary part="<<imag<<endl ;
}
};

int main( )
{
Complex c1(5,2);
c1.display();
return 0;
}
```

Output:

```
Real part=5
Imaginary part=2
```

3) Copy Constructor

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

For example, the statement

```
Complex c2(c1);
```

Creates new object c2 and performs member-by-member copy of c1 into c2. Another form of this statement is

```
Complex c2 = c1;
```

The process of initializing through assignment operator is known as copy initialization.

A copy constructor takes reference to an object of the same class as its argument. For example,

```
Complex (Complex &x)
{
real=x.real;
Imag=x.imag;
}
```

Remember: *We cannot pass the argument by value to a copy constructor*

When no copy constructor is defined, the compiler supplies its own copy constructor.

```
#include<iostream>
using namespace std;
class Complex
{
private:
int real, imag ;
public:
Complex( )
{
}
Complex (int r,int i)
{
real=r ;
imag=i ;
}
Complex (Complex &x)
{
real=x.real ;
imag=x.imag ;
}
void display( )
{
cout<<"Real part=" <<real <<endl;
cout<<"Imaginary part=" <<imag <<endl;
}
};
```

```

int main( )
{
Complex c1(5,10) ;
Complex c2(c1) ; // copy constructor called
Complex c3;
c3=c1;
cout<<"Details of c1"<<endl;
c1.display( ) ;
cout<< "Details of c2"<<endl;
c2.display( ) ;
cout<< "Details of c3"<<endl;
c3.display( ) ;
return 0;
}

```

Constructor Overloading

We can have more than one constructor in a class with same name and different number of arguments. This concept is known as Constructor Overloading.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

```

#include<iostream>
using namespace std;
class Complex
{
int real,imag;
public:
Complex() //Default Constructor
{
real=0;
imag=0;
}
Complex(int r,int i) //Parameterized Constructor
{
real=r;
imag=i;
}

```

```

Complex(Complex &x) //Copy Constructor
{
real=x.real;
imag=x.imag;
}

void display()
{
cout<<"Real part:"<<real<<endl;
cout<<"Imaginary part"<<imag<<endl;
}
};

int main()
{
Complex c1;
Complex c2(5,2);
Complex c3(c2);
c1.display();
c2.display();
c3.display();
return 0;
}

```

In the above example of class Complex, we have defined three constructors. The first one is invoked when we don't pass any arguments. The second gets invoked when we supply two argument, while the third one gets invoked when an object is passed as an argument.

Example

```

Complex c1;
This statement would automatically invoke the first one.
Complex c2(102,300); invokes 2nd constructor
Complex c3(c2); invokes 3rd constructor

```

Constructor with default argument

It is possible to define constructor with default arguments.

```
#include<iostream>
using namespace std;
class test
{
int a,b,c;
public:
test(int x=1,int y=2,int z=3);
void display();
};
test::test(int x,int y,int z)
{
a=x;b=y;c=z;
}
void test::display()
{
cout<<"a="<Output:
a=1b=2 c=3
a=5b=10 c=15
a=6b=2 c=3
```

Constructors

- Destructor is a member function that destroys the object that have been created by a constructor.
- Destructor name is similar to class name but preceded by a tilde sign (~).
- They are also defined in the public section.
- Destructor never takes any argument, nor does it return any value. So, they cannot be Overloaded.

- Destructor gets invoked, automatically, when an object goes out of scope (i.e. exit from the program, or block or function).

Example:

A destructor for the class test will look like

```

~test()
{
    -----
    -----
}

#include<iostream>
using namespace std;
class test
{
    static int count;
public:
    test()
    {
        count++;
        cout<<"object:"<<count<<"created"<<endl;
    }
    ~test()
    {
        cout<<"object:"<<count<<"destroyed"<<endl;
        count--;
    }
};

int test::count;
int main()
{
    cout<<"Inside the main block"<<endl;
    test t1;
    {
        //Block 1
        cout<<"Inside Block 1"<<endl;
        test t2,t3;
        cout<<"Leaving Block 1"<<endl;
    }
    cout<<"Back to main Block"<<endl;
    return 0;
}

```

Output:

```
Inside the main block
object:1created
Inside Block 1
object:2created
object:3created
Leaving Block 1
object:3destroyed
object:2destroyed
Back to main Block
object:1destroyed
```

New and Delete Operator

New

- Dynamic memory is allocated using operator **new**.
- **new** is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated.

Syntax: pointer-variable = new data-type;

This expression is used to allocate memory to contain **one single element of type data-type**.

For Example: int *ptr;
 ptr=new int;

Similarly,

Syntax: pointer-variable = new data-type [size];

This one is used to allocate **a block (an array) of elements of type data-type**, where size is an integer value representing the number of elements.

For Example:

```
int *ptr;
*ptr = new int [5];
```

Delete

- In most cases, memory allocated dynamically is only needed during specific periods of time within a program. Once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory.
- **delete** operator free the memory allocated to the variable i.e. it deletes the variables memory.

Syntax is:

```
delete pointer-variable;  
delete [size] pointer-variable;
```

- The first statement releases the memory of a single element allocated using new
- The second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).

Note: The size specifies the number of elements in the array to be freed. The problem with this form is that programmer should remember the size of the array. Recent version of c++ do not require the size to be specified. for example,
`delete[] ptr;`

Example Program for new & delete operator

```
#include <iostream>  
using namespace std;  
int main ()  
{  
    int i,size;  
    int *ptr;  
    cout << "How many numbers would you like to Enter? ";  
    cin >>size;  
    ptr= new int[size];  
    for (i=0; i<size; i++)  
    {  
        cout << "Enter number:" << i+1 << endl;  
        cin >> ptr[i];  
    }  
    cout << "You have entered:" << endl;  
    for (i=0; i<size; i++)  
    {  
        cout << ptr[i] << endl ;  
    }  
    delete[] ptr;  
    return 0;  
}
```

Memory Mapping, Allocation and Recovery

Stack & Heap Memory

Stack

- It's a region of computer's memory that stores temporary variables created by each function (including the main() function).
- The stack is a "Last in First Out" data structure and limited in size. Every time a function declares a new variable, it is "pushed" (inserted) onto the stack. Every time a function exits, all of the variables pushed onto the stack by that function, are freed or popped (that is to say, they are deleted).
- Once a stack variable is freed, that region of memory becomes available for other stack variables.
- A key to understanding the stack is the notion that when a function exits, all of its variables are popped off (Removed) of the stack (and hence lost forever).

Advantages

- Memory is managed to store variables automatically. We don't have to allocate memory by hand, or free it once we don't need it any more.
- CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.

Limitations

- Limit (varies with OS) on the size of variables that can be stored on the stack.

Heap

- The heap is a region of computer's memory that is not managed automatically and is not as tightly managed by the CPU.
- Dynamic memory allocation is a way for running programs to request memory from the operating system when needed. This memory does not come from the program's limited stack memory -- instead, it is allocated from a much larger pool of memory managed by the operating system called the heap. It is a more free-floating region of memory (and is larger).
- To allocate memory on the heap, you must use **new** operator in C++. Once you have allocated memory on the heap, you are responsible for using **delete** to de-allocate that memory once you don't need it any more.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of computer).
- Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap.

- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

Stack Vs Heap

Stack	Heap
1) Only Local variables can be accessed.	1) Variables can be accessed globally
2) Limit on stack size dependent on OS.	2) Does not have a specific limit on memory size.
3) It can access faster.	3) Relatively slower access.
4) Don't have to explicitly de-allocate variables	4) We must allocate and de-allocate variable explicitly.(for allocating variable new and for freeing variables delete operator is used.
5) Variables cannot be resized.	5) Variables can be resized using new operator
6) Space is managed efficiently by CPU, memory will not become fragmented.	6) No guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed.
7) Main issue is shortage of memory.	7) Main issue is Memory fragmentation.
8) It is used for static memory allocation, meaning memory is allocated at compile time before the program executes.	8) It is used for dynamic memory allocation meaning the memory can be allocated and freed in random order.

When to use the Heap or stack?

We should use heap when we require to allocate a large block of memory. For example, you want to create a large size array or big structure to keep that variable around a long time then you should allocate it on the heap.

However, If we are working with relatively small variables that are only required until the function using them is alive. Then we need to use the stack, which is faster and easier.

Previous Board Exam Questions

- 1) What is message passing? Describe with example. [PU:2014 fall]
- 2) What is the difference between message passing and function call? Explain the basic message formalization. [PU:2006 spring]
- 3) Differentiate message passing and procedure call with suitable example. What are the possible memory errors in programming. [PU:2014 spring]
- 4) Explain the following term with suitable examples. Agents ,Responsibility, Messages and methods. [PU:2009 fall]
- 5) Explain message passing formalism with syntax in C++. What is stack Vs heap memory allocation? [PU:2016 spring]
- 6) What does constructor mean? Explain different types of constructor with suitable examples. [PU:2005 fall]
- 7) What is constructor? Explain copy constructor with suitable examples. [PU:2009 fall]
- 8) What is constructor? Write an example of Copy constructor and explain each line of code. [PU:2017 spring]
- 9) “A constructor is a special member function that automatically initializes the objects of its class”, support this statement with a program of all types of constructors. Also enlist the characters of constructors. [PU:2015 spring]
- 10) What is constructor? Is it mandatory to use constructor in class. Explain [PU:2006 spring]
- 11) Differentiate between constructor and destructor. Can there be more than one destructor in a program for destroying a same object. Illustrate your answer. [PU:2010 fall]
- 12) What are constructors and destructors? Explain their types and uses with good illustrative example? What difference would be experienced if the features of constructors and destructors were not available in C++. [PU:2009 spring]
- 13) What is de-constructor? can you have two destructors in a class? Give example to support your reason. [PU:2014 spring]
- 14) Discuss the various situations when a Copy constructor is automatically invoked. How a default constructor can be equivalent to a constructor having default arguments. [PU:2013 spring]
- 15) Can we have more than one destructor in a class? Write a Program to add two complex numbers using the concept of constructor. [PU:2015 spring]
- 16) What do you mean by dynamic constructor? Explain its application by a program to compute complex numbers. [PU:2016 fall]
- 17) Differentiate methods of argument passing in constructor and destructor. [PU:2017 fall]

- 18) Why destructor function is required in class? Can a destructor accept arguments?
[PU:2017 spring]
- 19) What is constructor? Can constructor can be overloaded? If yes how that is possible with reference of an example? [PU:2018-fall][PU:2017 fall]
- 20) Discuss about stack vs heap storage allocation. [2010 spring]
- 21) What do you mean by stack vs Heap? Explain the memory recovery. Explain the use of new and delete operator. [PU:2009 spring]
- 22) Explain and contrast memory recovery, stack and heap with suitable example.[PU:2013 fall]
- 23) What are the advantages of dynamic memory allocation? Explain with suitable example. [PU:2016 spring]
- 24) What is memory recovery? How does stack differ from heap memory allocation.
[PU:2005 fall]
- 25) Write a short notes on:
- Copy constructor[PU:2014 spring]
 - Stack vs Heap based Allocation
 - Memory Recovery[PU:2015 spring]

1) WAP to find the area of rectangle using the concept of constructor

```
#include<iostream>
using namespace std;
class Rectangle
{
private:
    float length,breadth,area;
public:
    Rectangle(float l,float b)
    {
        length = l;
        breadth = b;
    }
    void calc( )
    {
        area= length * breadth;
    }
    void display( )
    {
        cout << "Area = " << area << endl;
    }
};
int main()
{
    Rectangle r(2,4);
    r.calc();
    r.display();
    return 0;
}
```

2) WAP to initialize name, roll and marks of student using constructor and display the obtained information.

```
#include<iostream>
#include<string.h>
using namespace std;
class Student
{
private:
    char name[20];
    int roll;
    float marks;
```

```
public:  
Student(char n[],int r,float m)  
{  
strcpy(name,n);  
roll=r;  
marks=m;  
}  
void display()  
{  
cout<<"Name:"<<name<<endl;  
cout<<"Roll:"<<roll<<endl;  
cout<<"Marks:"<<marks<<endl;  
}  
};  
int main()  
{  
char name[20];  
int roll;  
float marks;  
cout<<"Enter name"<<endl;  
cin>>name;  
cout<<"Enter marks"<<endl;  
cin>>marks;  
cout<<"Enter Rollno"<<endl;  
cin>>roll;  
Student st(name,marks,roll);  
st.display();  
return 0;  
}
```

- 3) Create a class student with data members (name, roll, marks in English, Maths and OOPs(in 100), total, average) a constructor that initializes the data members to the values passed to it as parameters, A function called calculate () that calculates the total of the marks obtained and average. And function print() to display name, roll no, total and average.**

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
    private:
        int roll;
        float me,mm,mo,total,avg;
        char name[30];
    public:
        student(char n[],int r,float e,float m,int o)
        {
            strcpy(name,n);
            roll=r;
            me=e;
            mm=m;
            mo=o;
        }
        void calculate()
        {
            total= me+mm+mo;
            avg=total/3;
        }
        void print()
        {
            cout<<"Name="<<name<<endl;
            cout<<"Roll No="<<roll<<endl;
            cout<<"Total="<<total<<endl;
            cout<<"Average="<<avg<<endl;
        }
};
```

```

int main()
{
    char name[30];
    int roll;
    float meng,mmath,moops;
    cout<<"Enter the name"<<endl;
    cin>>name;
    cout<<"Enter the roll"<<endl;
    cin>>roll;
    cout<<"Enter the marks in english"<<endl;
    cin>>meng;
    cout<<"Enter the marks in maths"<<endl;
    cin>>mmath;
    cout<<"Enter marks in oops"<<endl;
    cin>>moops;
    student st(name,roll,meng,mmath,moops);
    st.calculate();
    st.print();
    return 0;
}

```

- 4) Write a Program to add two complex number using the concept of
Constructor/Constructor overloading.
[PU:2015 spring]**

```

#include<iostream>
using namespace std;
class complex
{
private:
int real,imag;
public:
complex()
{
real=2;
imag=4;
}
complex(int r,int i)
{
real=r;
imag=i;
}

```

```

void addcomplex(complex c1,complex c2)
{
    real=c1.real+c2.real;
    imag=c1.imag+c2.imag;
}
void display()
{
    cout<<real<<"+"<<imag<<"i"<<endl;
}
};
int main()
{
    complex c1;
    complex c2(10,20);
    complex c3;
    cout<<"First complex number=";
    c1.display();
    cout<<"Second complex number=";
    c2.display();
    c3.addcomplex(c1,c2);
    cout<<"sum of complex number=";
    c3.display();
    return 0;
}

```

- 5) Create a class person with data members Name, age, address and citizenship number.
Write a constructor to initialize the value of the person. Assign citizenship number if
the age of the person is greater than 16 otherwise assign value zero to citizenship
number. Also create a function to display the values.[PU:2013 fall]**

```

#include<iostream>
#include<string.h>
using namespace std;
class person
{
private:
    char name[20];
    char address[20];
    int age;
    long int citizno;

```

```

public:
person(char n[],int a,char addr[],long int c)
{
strcpy(name,n);
age=a;
strcpy(address,addr);
citno=c;
}

void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Address:"<<address<<endl;
cout<<"Age:"<<age<<endl;
cout<<"Citizenship no:"<<citno<<endl;
}
};

int main()
{
char name[20];
char address[20];
int age;
long int citno;
cout<<"Enter the name"<<endl;
cin>>name;
cout<<"Enter the address"<<endl;
cin>>address;
cout<<"Enter the age"<<endl;
cin>>age;
if (age>16)
{
cout<<"Enter the citizenship number"<<endl;
cin>>citno;
person p(name,age,address,citno);
p.display();
}
else
{
person p(name,age,address,0);
p.display();
}
return 0;
}

```

- 6) Create a class Mountain with data members name, height, location, a constructor that initializes the members to the values passed it to its parameters, a function called CmpHeight() to compare two objects and DispInf() to display the information of mountain. In main create two objects of the class mountain and print the information of the mountain which is greatest height.[PU:2016 spring]**

```
#include<iostream>
#include<string.h>
using namespace std;
class Mountain
{
private:
char name[20],location[20];
float height;
public:
Mountain(char n[],float h,char l[])
{
strcpy(name,n);
height=h;
strcpy(location,l);
}

void DispInf()
{
    cout<<"Name:"<<name<<endl;
    cout<<"Height:"<<height<<endl;
    cout<<"Location:"<<location<<endl;
}
friend void CmpHeight(Mountain,Mountain);
};

void CmpHeight(Mountain m1,Mountain m2)
{
    if(m1.height>m2.height)
    {
        m1.DispInf();
    }
    else
    {
        m2.DispInf();
    }
}
```

```

int main()
{
char name1[20],name2[20];
float height1,height2;
char location1[20],location2[20];
cout<<"Enter Name,Height and location of first mountain"<<endl;
cin>>name1>>height1>>location1;
cout<<"Enter Name,Height and location of second mountain"<<endl;
cin>>name2>>height2>>location2;
Mountain m1(name1,height1,location1);
Mountain m2(name2,height2,location2);
cout<<"Information of mountain with greatest height"<<endl;
CmpHeight(m1,m2);
return 0;
}

```

- 7) Create a class time constructor having hour, minute and second as an arguments is use to take two time data from user. The add function that takes two class objects as arguments add them respectively then display the aggregate result?
(Apply 60 second =1 minutes and 60 minutes =1 hour)**

[PU: 2017 spring]

```

#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;
}
void display()
{
cout<<hr<<"."<<min<<"."<<sec<<endl;
}
void sum(time,time);
};

```

```

void time::sum(time t1,time t2)
{
    sec=t1.sec+t2.sec;
    min=sec/60;
    sec=sec%60;
    min=min+t1.min+t2.min;
    hr=min/60;
    min=min%60;
    hr=hr+t1.hr+t2.hr;
}
int main()
{
int hr1,min1,sec1,hr2,min2,sec2;
cout<<"Enter first hour,minutes and seconds"<<endl;
cin>>hr1>>min1>>sec1;
cout<<"Enter second hour,minutes and seconds"<<endl;
cin>>hr2>>min2>>sec2;
time t1(hr1,min1,sec1);
time t2(hr2,min2,sec2);
time t3;
cout<<"The 1st time is"<<endl;
t1.display();
cout<<"The 2nd time is"<<endl;
t2.display();
t3.sum(t1,t2);
cout<<"The resultant time is"<<endl;
t3.display();
return 0;
}

```

- 8) Create a class called employee with data member Code, Name, address, salary. Create a constructor to initialize the member of the class. Also create the another constructor so that we can create an object from another object. Define member function display() to display the information of the class.[PU:2018 fall]**

```
#include<iostream>
#include<string.h>
using namespace std;
class employee
{
    private:
        int code;
        char name[20];
        char address[20];
        float salary;
    public:
        employee(int c, char n[],char addr[],float s)
        {
            code=c;
            strcpy(name,n);
            strcpy(address,addr);
            salary=s;
        }
        employee(employee &e)
        {
            code=e.code;
            strcpy(name,e.name);
            strcpy(address,e.address);
            salary=e.salary;
        }
        void display()
        {
            cout<<"Code="<<code<<endl;
            cout<<"Name="<<name<<endl;
            cout<<"Address="<<address<<endl;
            cout<<"Salary="<<salary<<endl;
        }
};
```

```

int main()
{
    int code;
    char name[20];
    char address[20];
    float salary;
    cout<<"Enter Code:"<<endl;
    cin>>code;
    cout<<"Enter Name:"<<endl;
    cin>>name;
    cout<<"Enter Address:"<<endl;
    cin>>address;
    cout<<"Enter Salary:"<<endl;
    cin>>salary;
    employee e1(code,name,address,salary);
    employee e2(e1);
    cout<<"Values in object e1"<<endl;
    e1.display();
    cout<<"Values in object e2"<<endl;
    e2.display();
    return 0;
}

```

- 9) Write a simple program using of dynamic memory allocation which should include calculation of marks of 3 subjects of n students and displaying the result as pass or fail & name, roll. Pass mark is 45 out of 100 in each subject**

```

#include<iostream>
using namespace std;
class student
{
private:
char name[20];
int roll;
float m1,m2,m3;
public:
void getdata()
{
cout<<"enter the name"<<endl;
cin>>name;
cout<<"enter the roll"<<endl;
cin>>roll;
cout<<"Enter marks of 3 subjects"<<endl;
cin>>m1>>m2>>m3;
}

```

```

void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll:"<<roll<<endl;
if(m1>=45&&m2>=45&&m3>=45)
cout<<"Result:Pass"<<endl;
else
cout<<"Result:Fail"<<endl;
}
};

int main()
{
    int n,i;
    student *ptr;
    cout<<"Enter the number of students:";
    cin>>n;
    ptr= new student[n];
    for(i=0;i<n;i++)
    {
        cout<<"Enter the information of student"<<i+1<<endl;
        ptr[i].getdata();
    }
    for(i=1;i<=n;i++)
    {
        cout<<"Information of student"<<i<<endl;
        ptr[i].display();
    }
    delete[] ptr;
    return 0;
}

```

10) Write a simple program explaining the use of dynamic memory allocation which should include calculation of marks of 5 subjects of students and displaying the result of pass or fail.(Pass marks is 45 out of 100 in each subject) [PU: 2010 fall 3(b)]

```
#include<iostream>
using namespace std;
class student
{
private:
float sub1,sub2,sub3,sub4,sub5;
public:
void getdata()
{
cout<<"Enter marks of 5 subjects"<<endl;
cin>>sub1>>sub2>>sub3>>sub4>>sub5;
}
void display()
{
if(sub1>=45&&sub2>=45&&sub3>=45&&sub4>=45&&sub5>=45)
cout<<"student is passed"<<endl;
else
cout<<"student is failed"<<endl;
}
};
int main()
{
int n,i;
student *ptr;
cout<<"Enter the number of students:";
cin>>n;
ptr= new student[n];
for(i=0;i<n;i++)
{
cout<<"Enter marks of student"<<i+1<<endl;
ptr[i].getdata();
ptr[i].display();
}
delete[] ptr;
return 0;
}
```

11) WAP to find the sum of n numbers Entered by user using Dynamic Memory Allocation in C++.

```
#include <iostream>
using namespace std;
int main ()
{
    int i,size,sum=0;
    int *ptr;
    cout << "How many numbers would you like to Enter?" << endl;
    cin >> size;
    ptr= new int[size];
    for (i=0; i<size; i++)
    {
        cout << "Enter number" << i+1 << endl;
        cin >> ptr[i];
    }
    cout << "You have entered:" << endl;
    for (i=0; i<size; i++)
    {
        cout << ptr[i] << endl;
    }
    for (i=0; i<size; i++)
    {
        sum=sum+ptr[i];
    }
    cout << "sum of numbers=" << sum << endl;
    delete[] ptr;
    return 0;
}
```

CHAPTER 4

Object Inheritance and Reusability

- In OOPs, the mechanism of deriving new class from old one is called inheritance.
- The old class is referred to as (base class /parent class/super class) and new class is called as (derived class/child class /sub class).
- The derived class inherits some or all of the data and functions from base class so that so that we can reuse the already written and tested code in our program.

Example:

The properties of class named Person like name, age, address, phone number can be reusable in Class Employee. So, inheritance supports reusability and minimizes coding redundancy.

Visibility Modifier (Access Specifier):

Visibility Modifier (Access Specifier)	Accessible from own class	Accessible from derived class	Accessible from objects outside class
public	yes	yes	yes
private	yes	no	no
protected	yes	yes	no

Defining Derived Class (Specifying Derived Class):

General syntax :

```
class derived_class_name : visibility_mode base_class_name
{
//members of derived class
};
```

- The colon (:) indicates that the derived_class_name is derived from the base_class_name.
- The visibility_mode is optional, if present, may be either private or public. The default visibility mode is private.
- Visibility mode specifies whether the features of the base class are privately derived or publicly derived or derived on protected.

Base Class Visibility	Derived Class Visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

```

class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A  // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};

```

While any derived_class is inherited from a base_class, following things should be understood:

1. When a base class is privately inherited by a derived class, only the public and protected members of base class can be accessed by the member functions of derived class. This means no private member of the base class can be accessed by the objects of the derived class. Public and protected member of base class becomes private in derived class.
2. When a base class is publicly inherited by a derived class the private members are not inherited, the public and protected are inherited. The public members of base class becomes public in derived class whereas protected members of base class becomes protected in derived class.
3. When a base class is protectedly inherited by a derived class, then public members of base class becomes protected in derived class ; protected members of base class becomes protected in the derived class, the private members of the base class are not inherited to derived class but note that we can access private member through inherited member function of the base class.

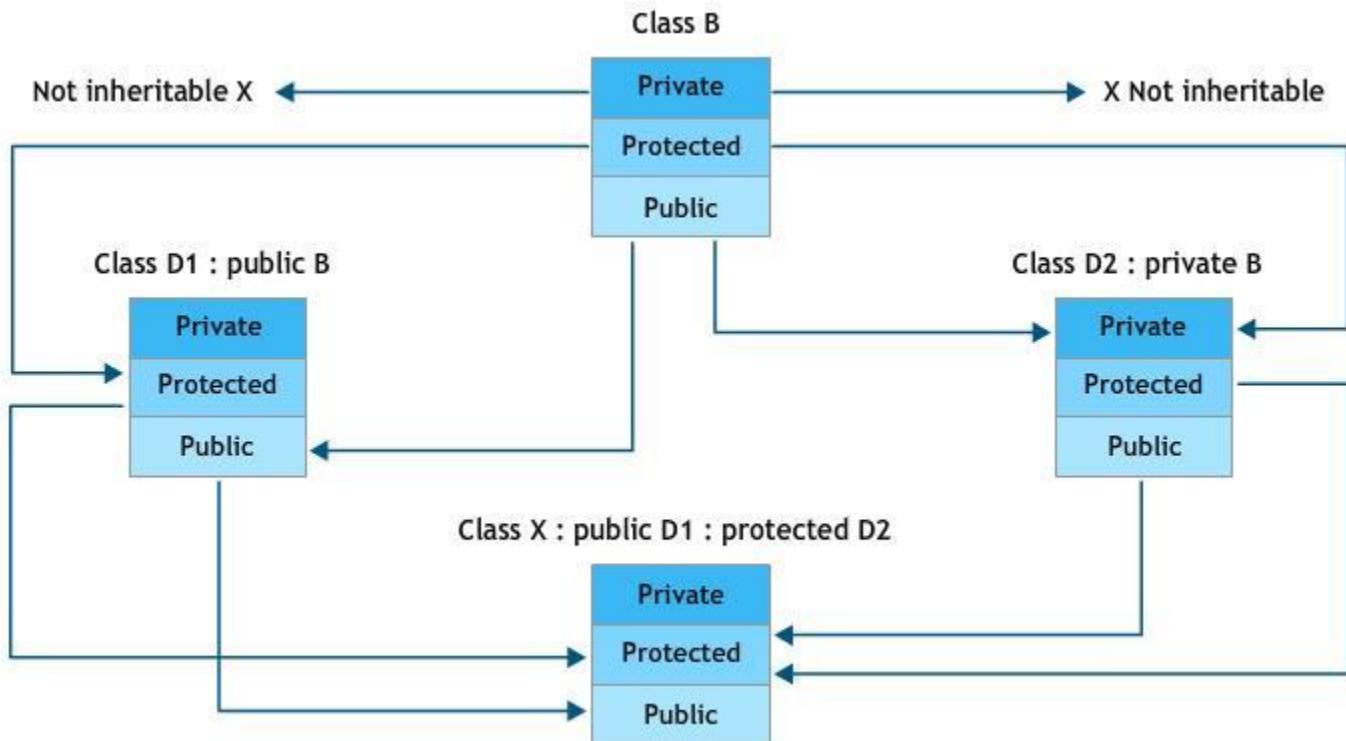


Fig: Effect of inheritance on visibility of members

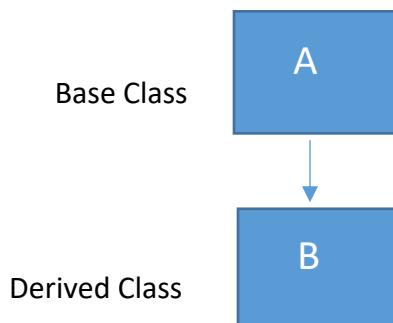
Types of Inheritance:

There are five types of inheritance:

- 1) Single Inheritance
- 2) Multiple inheritance
- 3) Hierarchical Inheritance
- 4) Multilevel Inheritance
- 5) Hybrid Inheritance

1) Single Inheritance

When a class is derived from only one base class, then it is called single inheritance.



General form:

```
class A           // base class
{
.....
};

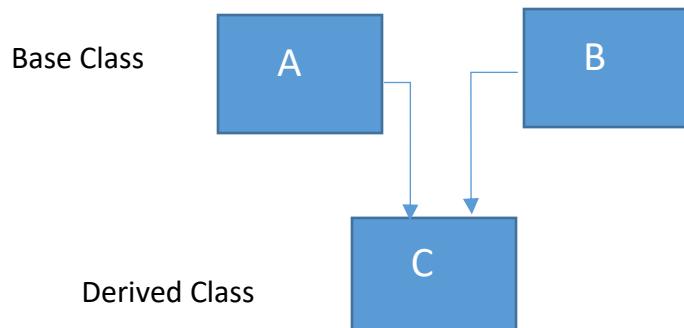
class B : public A // derived class B
{
.....
};
```

Here in, Single Inheritance

- One base class and one derived class only involved
- Class A is base class.
- B is derived class.
- B is not used as base class again
- No further extension of derived class
- All data members in protected and public of class A can be accessed using object of class B.

2) Multiple Inheritance

When a class is derived from two or more base classes, it is called multiple inheritance.



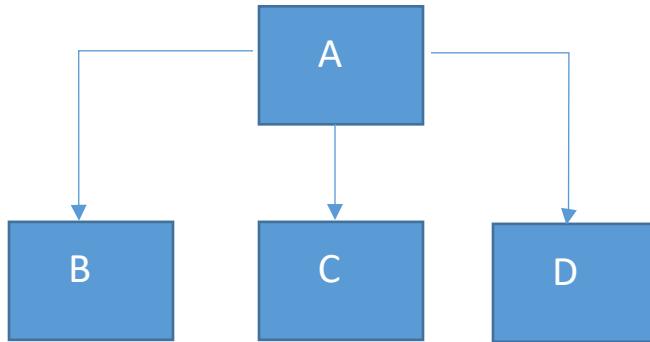
General form

```
class A  
{  
};  
Class B  
{  
};  
class C:public A, public B  
{  
};
```

In General form there are 3 classes A, B and C. Class A and B are base classes and C is derived class.

All data members in protected and public of class A & B can be accessed using object of class C. Here two base classes A and B have been created and a derived class C is created from both base classes.

- 3) **Hierarchical Inheritance:** When two or more than two classes are derived from one base class, it is called hierarchical inheritance.

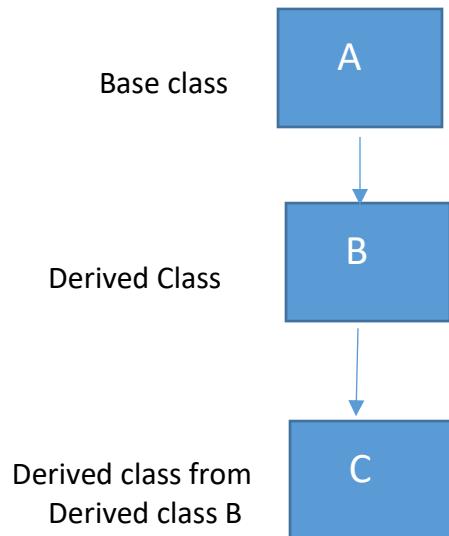


General form

```
class A  
{  
};  
Class B: public A  
{  
};  
Class C: public A  
{  
};  
Class D: public A  
{  
};
```

In General form three classes B and C and D are derived from same base class A. All data members in protected and public of class A can be accessed using object of classes B, C, D.

- 4) **Multilevel Inheritance:** The mechanism for deriving a class from another derived class is known as multilevel inheritance.



General form

```
Class A
{
    .....
};

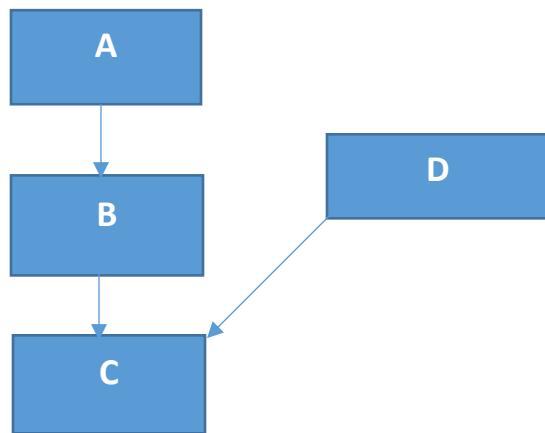
Class B: public A
{
    .....
};

Class C: public B
{
    .....
};
```

In General form there are 3 classes A,B and C. Class A is base class and Class B is derived class from Class A and class C is derived class from class B. All data members in protected and public of class A can be accessed using object of classes B, C and of class B can be accessed by class C.

5) Hybrid Inheritance:

Combination of one or more types of inheritance is known as hybrid inheritance.



```
Class A  
{  
};  
Class D  
{  
};  
Class B: public A  
{  
};  
Class C: public B, Public D  
{  
};
```

Description: In General form there are 4 classes A, B, C and D. Classes A & D are base classes and Class B is derived class from Class A and class C is derived class from classes B & D. All data members in protected and public of class A can be accessed using object of classes B, C and of classes B&D can be accessed by class C.

Program:

Single Inheritance: Public

```
#include<iostream>
using namespace std;
class A
{
private:
int x;
protected:
int y;
public:
int z;
void get_xyz()
{
    cout<<"Enter the value of x,y,z"<<endl;
    cin>>x>>y>>z;
}
void show_xyz()
{
    cout<<"x="<<x<<"y="<<y<<"z="<<z<<endl;
}
};
class B:public A
{
private:
int k,sum;
public:
void get_k()
{
cout<<"Enter the value of k"<<endl;
cin>>k;
}
void show_k()
{
cout<<"k="<<k<<endl;
}
void addition()
{
    sum=y+z+k;
}
```

```

void display()
{
cout<<"Sum="<<sum<<endl;
}
};

```

```

int main()
{
B b1;
b1.get_xyz();
b1.get_k();
b1.show_xyz();
b1.show_k();
b1.addition();
b1.display();
}

```

Single Inheritance: Private

```

#include<iostream>
using namespace std;
class A
{
private:
int x;
protected:
int y;
public:
int z;
void get_xyz()
{
    cout<<"Enter the value of x,y,z"<<endl;
    cin>>x>>y>>z;
}
void show_xyz()
{
    cout<<"x="<<x<<"y="<<y<<"z="<<z<<endl;
}
};

class B:private A
{
private:
int k,sum;
public:

```

```
void getdata()
{
get_xyz();
cout<<"Enter the value of k"<<endl;
cin>>k;
}
void showdata()
{
show_xyz();
cout<<"k="<<k<<endl;
}
void addition()
{
    sum=y+z+k;
}
void display()
{
cout<<"y+z+k="<<sum<<endl;
}
};

int main()
{
B b1;
b1.getdata();
b1.showdata();
b1.addition();
b1.display();
}
```

Example of Multiple Inheritance:

```
#include<iostream>
using namespace std;
class M
{
protected:
    int m;
public:
    void get_m(int x)
    {
        m=x;
    }
};
class N
{
protected:
int n;
public:
    void get_n(int y)
    {
        n=y;
    }
};

class P:public M,public N
{
public:
    void display()
    {
        cout<<"m= "<<m<<endl;
        cout<<"n= "<<n<<endl;
        cout<<"m*n="<<m*n<<endl;
    }
};

int main()
{
P p;
p.get_m(10);
p.get_n(20);
p.display();
return 0;
}
```

Example of multilevel Inheritance

```
#include<iostream>
using namespace std;
class student
{
protected:
int roll;
public:
void getnum(int x)
{
roll=x;
}

void putnum()
{
cout<<"Roll number="<<roll<<endl;
}
};

class test:public student
{
protected:
float sub1,sub2;
public:
void getmarks(float x,float y)
{
sub1=x;
sub2=y;
}

void putmarks()
{
cout<<"Marks in sub1="<<sub1<<endl;
cout<<"Marks in sub2="<<sub2<<endl;
}
};
```

```

class result:public test
{
    private:
        float total;
    public:
        void display()
    {
        total=sub1+sub2;
        putnum();
        putmarks();
        cout<<"Total="<<total<<endl;
    }
};

int main()
{
    result r;
    r.getnum(10);
    r.getmarks(75.5,80);
    r.display();
    return 0;
}

```

Example of Hierarchical Inheritance

```

#include<iostream>
using namespace std;
class A
{
protected:
    int x,y;
public:
    void init()
    {
        x=20;y=10;
    }
};

```

```

class B:public A
{
private:
int sum ;
public:
void add( )
{
sum=x+y ;
cout<< "x+y=" << sum << endl;
}
};

class C: public A
{
int diff;
public:
void sub()
{
diff=x-y;
cout<< "x-y=" << diff << endl;
}
};

class D: public A
{
int m;
public:
void mul( )
{
m=x*y ;
cout<< "x*y=" << m << endl;
}};
int main()
{
B b1;
C c1;
D d1;
b1.init();
b1.add();
c1.init();
c1.sub() ;
d1.init();
d1.mul();
return 0;
}

```

Example of Hybrid Inheritance

```
#include<iostream>
using namespace std;
class A
{
protected:
int x;
public:
void assignx( )
{
x=20;
}
};
class B: public A
{
protected:
int y ;
public:
void assigny()
{
y=40;
}
};
class C :public B
{
protected:
int z;
public:
void assignz( )
{
z=60;
}
};
class D
{
protected :
int k ;
public:
void assignk( )
{
k=80;
}
};
```

```
class E :public D, public C
{
private:
int total;
public:
void output()
{
total=x+y+z+k;
cout<< "x+y+z+k=" << total << endl;
}
};

int main( )
{
E e1;
e1.assignx();
e1.assigny();
e1.assignz();
e1.assignk();
e1.output();
return 0;
}
```

Ambiguity in Multiple inheritance

When two or more than two base classes have a function of identical name and when class inherits from multiple base classes then ambiguity occurs.

Let us consider the following case:

```
class M
{
public:
void display()
{
cout<<"Class M"<<endl;
}
};

class N
{
public:
void display()
{
cout<<"Class N"<<endl;
}
};
```

Which display function is used by the derived class when we inherit these two classes?
We can solve this problem by redefining the member in the derived class using **resolution operator** with the function as shown below.

```
class P:public M,public N
{
public:
void display()
{
M::display();
}
};
```

Now we can use the derived class as follows:

```
int main()
{
P p;
p.display();
return 0;
}
```

Output:

Class M

Ambiguity in single Inheritance

Ambiguity may also arise in single inheritance applications. For example, Let us consider the following situation.

```
class A
{
public:
void display()
{
cout<<"class A"<<endl;
}
};

class B:public A
{
public:
void display()
{
cout<<"class B"<<endl;
}
};
```

In this case, the function in the derived class overrides the inherited function and therefore simple call to display() by B type object will invoke function defined in B only. However, we may invoke the function defined in A by using the scope resolution operator to specify the class.

```
int main()
{
B b;
b.display(); //derived class object
b.A::display(); //invokes display() in A
b.B::display(); //invokes display() in B
return 0;
}
```

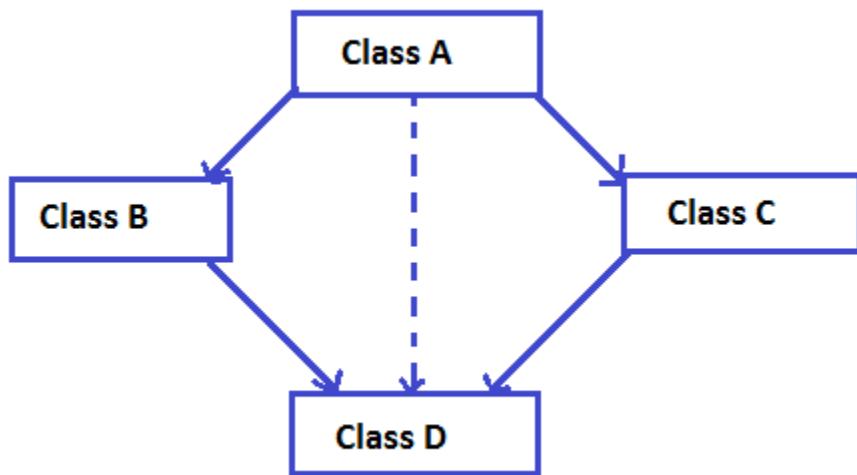
Output:

Class B
Class A
Class B

Virtual Base class

- During the time of hybrid inheritance when there is a hierarchical inheritance at the upper level and multiple inheritance at lower level, ambiguity occurs due to the duplication of data from multiple path at the grand child class. How this kind of ambiguity is resolved? Explain with suitable example.[PU:2013 fall]
- Does ambiguity occurs in hybrid inheritance .If yes? How can you remove this? Explain with example.[PU:2018 fall]
- Under what condition virtual base class is created? Explain it with suitable example .[PU:2017 fall]

Let us consider the following situation:



In the above example, both Class B and Class C inherit properties of Class A, they both have single copy of Class A. However Class D inherit both Class B and Class C, therefore Class D have two copies of Class A, one from Class B and another from Class C. This introduces ambiguity and should be avoided..

The duplication of inherited members due to these multiple paths can be avoided by making common base class as virtual class while declaring the direct or intermediate base classes.

By making a class virtual only one copy of that class is inherited though we may have many inheritance paths between the virtual base class and a derived class .we can specify a base class inheritance by using the keyword virtual.

To remove multiple copies of Class A from Class D, we must inherit Class A in Class Band Class C as virtual class.

```
class A
{
-----
};

class B:virtual public A
{
//parent1
-----
};

class C:public virtual A
{
//parent2
-----
};

class D:public B, public C{
//child
-----
};
```

Program

```
#include<iostream>
using namespace std;
class A
{
    public:
        int a;
};
class B : virtual public A
{
    public:
        int b;
};
class C : virtual public A
{
    public:
        int c;
};

class D : public B, public C
{
    public:
        int d;
};

int main()
{
    D obj;
    obj.a = 10;    //value of obj.a is replaced by 100 due to only one copy of data member of class A
    obj.a = 100;   // is available in class D
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout<< "A=" << obj.a << endl;
    cout<< "B=" << obj.b << endl;
    cout<< "C=" << obj.c << endl;
    cout<< "D=" << obj.d << endl;
    return 0;
}
```

Constructor and Destructor in Inheritance

- Compiler automatically calls constructor of base class and derived class automatically when derived class object is created.
- If we declare derived class object in inheritance constructor of base class is executed first and then constructor of derived class.
- If derived class object goes out of scope or deleted by programmer the derived class destructor is executed first and then base class destructor

Illustration of when base and derived class constructor and destructor functions are executed

```
#include <iostream>
using namespace std;
class A
{
public:
A( )
{
cout<< "Constructor in base class"<<endl ;
}
~A( )
{
cout<< "Destructor in base class"<<endl ;
}
};

class B:public A
{
public:
B()
{
cout<< "Constructor in derived class"<<endl ;
}
~B()
{
cout<< "Destructor in derived class"<<endl ;
}
};

int main( )
{
    B obj;
    return 0;
}
```

In Above Program Class A is base class with one constructor and destructor, Class B is derived from class A having a constructor and destructor. In **main()** Object of derived class i.e. class B is declared. When class B's object is declared constructor of base class is executed followed by derived class constructor. At end of program destructor of derived class is executed first followed by base class destructor

Output:

```
Constructor in base Class  
Constructor in derived class  
Destructor in derived Class  
Destructor in base class
```

Constructor in derived class

- If the **base class does not have any constructors taking arguments** (parameterized constructors), the **derived class need not have a constructor** function.
- If the **base class contains a constructor with one or more arguments** then it is **mandatory for the derived class to have a constructor** and pass the arguments to the base class constructors.
- When both the **derived and base classes contain constructors** then the **base class constructors is execute first** and then the constructor in the derived class is executed.
- In case of **Multiple Inheritance**
 - The **base classes are constructed in the order** in which they appear in the declaration of the derived class
- In case of **Multilevel Inheritance**
 - The constructors will be executed **in the order of inheritance**
- The constructors for virtual base classes are invoked before any non-virtual base classes.

Method of inheritance	Order of Execution
Class B:public A{ };	A(); base constructor (first) B(); derived constructor(second)
Class A:public B, public C{ }	B(); base constructor (first) C(); base constructor(second) A();derived (last)
Class A: public B, virtual public C{ }	C(); virtual base (first) B(); ordinary base constructor(second) A();derived (last)

Argument passing mechanism for supplying initial values to the bases classes constructors:

- The **constructor of the derived class receives the entire list of values as its arguments and passed them to the base class constructors** in the order in which they are declared in the base class.
- The **base class constructors are called and executed first** before executing the statements in the body of the derived constructor.

General form of defining a derived constructor:

Derived-constructor(Arglist1, Arglist2,.....ArglistN, ArglistD):

base1(Arglist1),

base2(arglist2)

.....

baseN(arglistN)

{

//Body of the derived constructor

}

The header line of derived constructor function contains two parts separated by a colon (:)
The first part provide the declaration of arguments that are passed to the derived constructor
and the second part lists the function calls to the base constructors

Here, base1(Arglist1), base2(Arglist2)..... baseN (ArglistN) are function calls to base class
constructors and Arglist1, Arglist2..... ArglistN are the actual parameters that are passed to the
base constructors . Here, ArglistD provides the parameters that are necessary to initialize the
members of the derived class itself.

Example:

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

```

class alpha
{
int x;
public:
alpha(int a)
{
x=a;
cout<<"Alpha is initialized" << endl;
}
void showa()
{
cout<<"x=" << x << endl;
}
};

class beta
{
int y;
public:
beta(int b)
{
y=b;
cout<<"Beta is initialized" << endl;
}
void showb()
{
cout<<"y=" << y << endl;
}
};

class gamma:public beta,public alpha
{
int z;
public:
gamma(int a,int b,int c):alpha(a),beta(b)
{
z=c;
cout<<"Gamma is initialized" << endl;
}

void showg()
{
cout<<"z=" << z << endl;
}
};

```

```
int main()
{
gamma g(5,10,15);
g.showa();
g.showb();
g.showg();
return 0;
}
```

Output:

```
Beta is initialized
Alpha is initialized
Gamma is initialized
```

Here, **beta** is initialized first although it appears second in the derived constructor *as it has been declared first in the derived class header line*

```
class gamma: public beta, public alpha
{
```

```
}
```

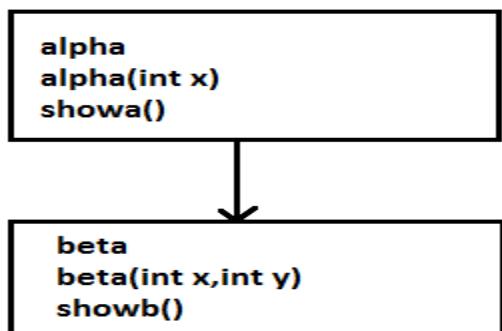
If we change the order to

```
class gamma: public alpha, public beta
{
```

```
}
```

then alpha will be initialized first

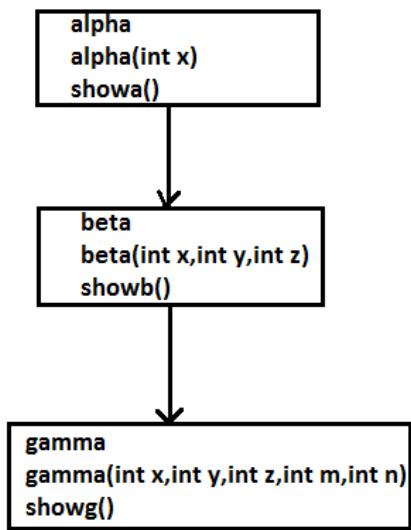
1. Write a complete program with reference to given below.



```
#include<iostream>
using namespace std;
class alpha
{
private:
int a;
public:
alpha(int x)
{
a=x;
}
void showa()
{
cout<<"value of a="<<a<<endl;
}
};
class beta:public alpha
{
private:
int b;
public:
beta(int x,int y):alpha(x)
{
b=y;
}
void showb()
{
cout<<"value of b="<<b<<endl;
}
};

int main()
{
beta b1(3,4);
b1.showa();
b1.showb();
return 0;
}
```

2. Write a complete program with reference to given below.



```
#include<iostream>
using namespace std;
class alpha
{
private:
int a;
public:
alpha(int x)
{
a=x;
}
void showa()
{
cout<<"value of a="<<a<<endl;
}
};
```

```

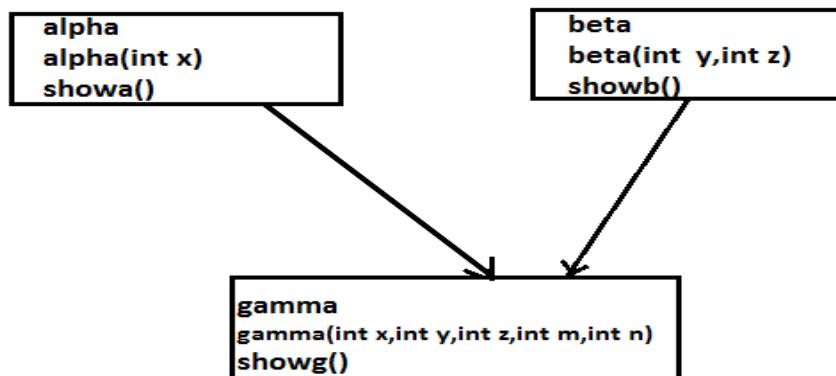
class beta:public alpha
{
private:
int b,c;
public:
beta(int x,int y,int z):alpha(x)
{
b=y;
c=z;
}
void showb()
{
cout<<"value of b="<<b<<endl;
cout<<"value of c="<<c<<endl;
}
};

class gamma:public beta
{
private:
int d,e;
public:
gamma(int x,int y,int z,int m,int n):beta(x,y,z)
{
d=m;
e=n ;
}
void showc()
{
cout<<"value of d="<<d<<endl;
cout<<"value of e="<<e<<endl;
}
};

int main()
{
gamma g1(5,6,7,10,20);
g1.showa();
g1.showb();
g1.showc();
return 0;
}

```

3. Write a complete program with reference to given below.



```
#include<iostream>
using namespace std;
class alpha
{
private:
int a;
public:
alpha(int x)
{
a=x;
}
void showa()
{
cout<<"value of a="<<a<<endl;
}
};
class beta
{
private:
int b,c ;
public:
beta(int y,int z)
{
b=y;
c=z;
}
```

```

void showb()
{
cout<<"value of b="<<b<<endl;
cout<<"value of c="<<c<<endl;
}
};

class gamma:public alpha,public beta
{
private:
int d,e;
public:
gamma(int x,int y,int z,int m,int n):alpha(x),beta(y,z)
{
d=m;
e=n    ;
}
void showc()
{
cout<<"value of d="<<d<<endl;
cout<<"value of e="<<e<<endl;
}
};

int main()
{
gamma g1(5,6,7,10,20);
g1.showa();
g1.showb();
g1.showc();
return 0;
}

```

Initialization list in constructors

C++ supports another method of initializing the class objects .This method uses what is known as initialization list in the constructor function. This takes the following form.

```
constructor (arglist):initialization- section
{
assignment-section
}
```

- **assignment-section:** It is body of the constructor function and used to initialize value to its data members.
- **Initialization-section:** This section is used to provide initial value to the base constructors and also initialize its own class member.
- We can use either of the section to initialize the data members of constructor in class.
- The initialization section basically contains a list of initializations separated by commas. This list is known as initialization list.

Consider the simple example:

```
class alpha
{
int a;
int b;
public:
alpha(int x,int y):a(x),b(2*y)
{
}
int main()
{
alpha a1(2,3);
}
```

This program will initialize a to 2 and b to 6. Now data members are initialized ,just by using the variable name followed by initialization value enclosed in the parenthesis(like a function call).

Any of the parameters of the argument list may be used as the initialization value and the item list may be in any order.

```
alpha(int x,int y):b(x),a(x+y)

{ }
```

In this case a will be initialized to 5 and b to 2.

Data members are initialized in the order of the declaration, independent of the order of the initialization list.

```
alpha(int x,int y):a(x),b(a*y){}
```

Here the data member a has been declared first so value of a is initialized first and its value is used to initialize b.

However, the following will not work:

```
alpha(int x,int y):b(x),a(b*y)
{
}
```

Because the value of b is not available to which is to be initialized first.

The following statements are also valid:

```
alpha(int x,int y):a(x)
{
b=y
}
```

OR

```
alpha(int x,int y)
{
a=x;
b=y;
}
```

Program:

```
#include<iostream>
using namespace std;
class alpha
{
private:
int a,b,c;
public:
alpha(int x,int y,int z):a(x),b(2*y)
{
c=z;
}
void showa()
{
cout<<"a="<<a<<endl;
cout<<"b="<<b<<endl;
cout<<"c="<<c<<endl;
}
};

class beta:public alpha
{
private:
int d,e;
public:
beta(int x,int y,int z,int m,int n):alpha(x,y,z),d(2+m)
{
e=n;
}
void showb()
{
cout<<"d="<<d<<endl;
cout<<"e="<<e<<endl;
}
};
int main()
{
beta b1(5,10,15,20,25);
b1.showa();
b1.showb();
return 0;
}
```

Subclass, Subtype and Principle of Substitutability

If we consider the relationship of a data type associated with a parent class to the data type associated with a child class, the following observations can be made.

- Instances of the child class must possess all data members associated with the parent class.
- Instances of the child class must implement, through inheritance at least (if not explicitly overridden), all functionality defined for parent class. (*They can also define new functionality*)
- The instance of a child class can mimic the behavior of parent class and should be indistinguishable from an instance of parent class if substituted in similar situation.

The Principle of Substitutability

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

The term subtype is used to refer to a subclass relationship in which the principle of substitutability is maintained to distinguish such forms from the general subclass relationship, we may or may not satisfy this principle.

In the example below the class B object is replaced by the object of class A without any error. This is achieved when a child class is inherited from a base class publicly. In the same example, if the mode of inheritance is made private the base class object ‘a’ can’t be replaced by the parent class object ‘a’.

EXAMPLE:

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

class A
{
public:
void display()
{
cout << "class A";
}
};
```

```

class B : public A
{
public:
void display()
{
cout << "class B";
}
};

void test(A a)
{
a.display();
}

int main()
{
B b;
test(b); // b substituted for object of A.
return 0;
}

```

Containership/Composition

When a 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”.

```

class A
{
.....
};

class B
{
.....
A obj1;
.....
};

```

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

Program to illustrate the concept of containership

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

class Employee
{
int eid;
float salary;
public:
void getdata()
{
cout<<"Enter id and salary of employee"<<endl;
cin>>eid>>salary;
}
void display()
{
cout<<"Emp ID:"<<eid<<endl;
cout<<"Salary:"<<salary<<endl;
}
};

class Company
{
char cname[20];
char department[20];
Employee e;
public:
void getdata()
{
e.getdata();
cout<<"Enter company name and Department:"<<endl;
cin>>cname>>department;
}
void display()
{
e.display();
cout<<"Company name:"<<cname<<endl;
cout<<"Department:"<<department<<endl;
}
};
```

```

int main()
{
Company c;
c.getdata();
c.display();
return 0;
}

```

In above example class company contains the object of another class employee. As we know “company **has a** employee” sounds logical .so there exists has a relationship between company and employee.

IS A RULE/HAS A RULE OR (IS A /HAS A RELATIONSHIP)

As we know, One of the advantages of an Object Oriented programming language is code reuse, however, There are two ways we can do code reuse either by the implementation of inheritance (IS-A relationship), or object composition (HAS-A relationship).

IS A Relationship	HAS A relationship
1. If the first concept is a specialized instance of the second one then there exists an “is-a” relationship between them.	1. If the second concept is a component of the first and the two are not in any sense the same thing then there exists a “has a” relationship between them.
2. Example: Car is a vehicle	2. Example: car has a engine.
3. “IS A” Relationship refers to Inheritance.	3. “HAS A” Relationship refers to composition.
4. ‘Is-a’ relationship asserts the instance of the subclass must be more specialized form of the superclass.	4. In ‘has-a’ relationship class contains object of another class as its member data to make it as a whole.
5. class Vehicle { }; class Car: Public Vehicle { }; class Bus: Public Vehicle { };	5. class engine{ }; class car { engine e; // a car is composed of an engine };

Inheritance VS. composition

Inheritance	Composition
<p>1. In Inheritance derived classes inherit attributes and methods from their parent class.</p> <p>2. It exhibits “is-a” relationship.</p> <p>3. Example: Car is a vehicle</p> <pre>4. class Vehicle { }; class Car: Public Vehicle { }; class Bus: Public Vehicle { }</pre>	<p>1. In composition a class contains object of another class as its member data.</p> <p>2. It exhibits “has-a” relationship.</p> <p>3. Example: Car has a engine</p> <pre>4. class engine { }; class car { engine e; // a car is composed of an engine };</pre>
<p>5. Inheritance leads tight coupling between superclass and subclass.</p> <p>6. So it is harder to maintain in future.</p>	<p>5. In Composition relationship between class can be decoupled easily .so it is can be easily maintained in future.</p>
<p>7. Inheritance permits polymorphism</p>	<p>6. Composition does not permit polymorphism.</p>
<p>Program to illustrate concept of inheritance</p> <pre>#include<iostream> using namespace std; class first { private: int a; public: void geta() { cout<<"Enter value of a"<<endl; cin>>a; } void puta() { cout<<"a="<<a<<endl; } };</pre>	<p>Program to illustrate concept of composition</p> <pre>#include<iostream> using namespace std; class first { private: int a; public: void geta() { cout<<"Enter value of a"<<endl; cin>>a; } void puta() { cout<<"a="<<a<<endl; } };</pre>

<pre> class second:public first { private: int b; public: void getb() { cout<<"Enter value of b"<<endl; cin>>b; } void putb() { cout<<"b="<<b<<endl; } }; int main() { second s; s.getb(); s.putb(); return 0; } </pre>	<pre> class second { private: int b; first f; public: void getb() { f.getb(); cout<<"Enter value of b"<<endl; cin>>b; } void putb() { f.putb(); cout<<"b="<<b<<endl; } }; int main() { second s; s.getb(); s.putb(); return 0; } </pre>
--	--

Software reusability

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

In OOP, the concept of inheritance provide the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.

Reusability can be provided through the concept of composition also. In composition one class contains the object of another classes to make it as a whole.

Advantages of reusability.

- Re usability enhanced reliability.
- Re usability saves the programmer time and effort.
- As the existing code is reused, it leads to less development and maintenance costs.
- Extensibility as we can extend the already made classes by adding some new features.
- Inheritance makes the sub classes follow a standard interface.

Difficulty in software reusability

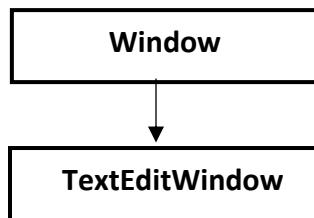
- As the number of projects and developers increases, it becomes harder to reuse software. It's a challenge to effectively communicate the details and requirements for code reuse.
- As the number of projects and developers grows, it's difficult to share libraries of reusable code.

Forms of Inheritance

1. Subclassing for specialization

The new class is a specialized form of the parent class but satisfies the specifications of the parent class in all relevant aspects.

Example:

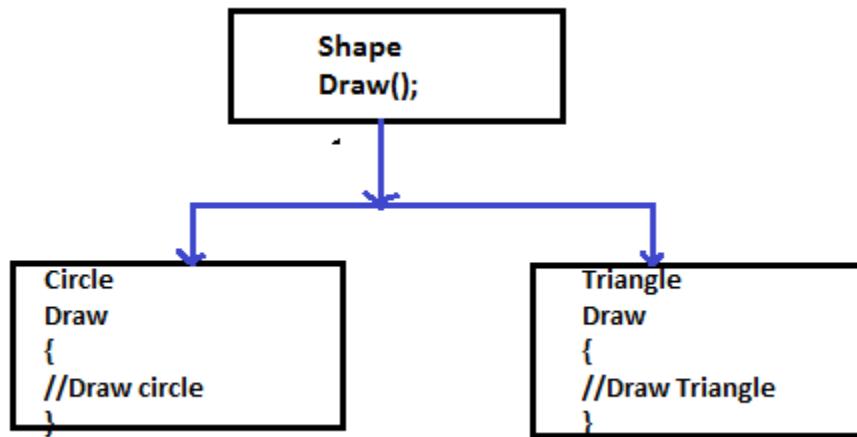


- Window class provides the general windowing operations (moving, resizing, iconification and so on).
- The specialized subclass TextEditWindow inherits the window operations and in addition provides the facilities that allow window to display textual material and user to edit the text values.

2. Subclassing for specification

- Here class maintains the common interface (they implement the same methods).
- The parent class can be a combination of implemented operations and operations that are deferred to the child classes.
- There is no interface change as the child implements the behavior described but not implemented in parent.

Example:



3. Subclassing for Construction

Here a class can often inherit almost all of its desired functionality from a parent class, perhaps changing only names of the methods used to interface to the class or modifying the arguments in a certain fashion.

4. Subclassing for Generalization

- It is opposite to the subclassing for specialization.
- Subclass extends the behavior of parent class to create more general kind of object.
- It is applicable when we build on a base of existing classes that we do not wish to or cannot modify.

Consider a graphics display system in which a class Window has been defined for displaying on a simple black-and-white background. We could create a subtype ColoredWindow that lets the background color to be something other than white by adding an additional field to store the color and overriding the inherited window display code that specifies the background be drawn in that color.

5. Subclassing for extension

- The child class adds the new functionality to the parent class but does not change any inherited behavior.
- Subclassing for generalization modifies or expands on the existing functionality of an object whereas subclassing for extension add totally new abilities.

6. Subclassing for limitation

- The child class restricts the use of some of the behaviors inherited from the parent class.
- Subclassing for limitation occurs when the behavior of subclass is smaller or more restrictive than the behavior of the parent class.

For example ,an existing class library provides the double-ended queue or **deque**, data structure. Elements can be added or removed from either end of the **deque**, but a programmer wishes to write stack class, enforcing the property that elements can be added to removed from only one end of the stack.

7. Subclassing for variance

Subclassing for variance is employed when two or more classes have similar implementations but do not seem to possess any hierarchical relationships between the abstract concept represented by the classes.

Example

Code required to control the mouse may be nearly identical to the code required to control the graphics tablet. Conceptually there is no reason why class mouse should be made the subclass of class GraphicsTablet.

A better alternative is to factor out the common code into an abstract class, say PointingDevice and to have both classes inherit from this parent classes.

8 .Subclassing for Combination

Here, the subclass represents the combination of features from two or more parent classes .Eg. A teaching assistant may have characteristic of both teacher and student. This is multiple inheritance.

SUMMARY OF FORMS OF INHERITANCE

Inheritance is used in variety ways according to user's requirements. The Following are forms of inheritance.

- 1) **Sub classing for specialization:** The child class is the special case of the parent class; in other words the child class is a subtype of parent class.
- 2) **Sub classing for specification:** The parent class defines behavior that is implemented in child class but not in parent class.
- 3) **Sub classing for construction:** The child class makes use of behavior provided by the parent class but is not a subtype of parent class.
- 4) **Sub classing for generalization:** The child class modifies or overrides some of the methods of the parent class but is not a subtype of parent class.
- 5) **Sub classing for extension:** The child class adds new functionality to the parent class but does not change any inherited behavior.
- 6) **Sub classing for limitation:** The child class restricts the use of some of the behavior inherited from parent class.
- 7) **Sub classing for variance:** The child class and parent class are variants of each other and the class-subclass relationship is arbitrary.
- 8) **Sub classing for combination:** The child class inherits features from more than one parent class. This is multiple inheritance.

Advantages and Disadvantages of Inheritance

Advantages (Pros /Merits)

- 1) Inheritance strongly supports reusability. The data members and member functions that are defined at parent class can be reused in base class. So, there is no need to define the member again.
- 2) Reusability enhanced reliability. The base class code will be already tested and debugged.
- 3) Base class logic can be extend as per business logic of the derived class.
- 4) Eliminates duplication of code.
- 5) Reduces development and maintenance costs as well saves time and efforts.
- 6) Program structure is short and concise which is more reliable.
- 7) Codes are easy to debug. Inheritance allows the program to capture the bugs easily
- 8) With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.
- 9) It is easy to partition the work in a project based on objects.
- 10) Object oriented systems can be easily upgraded from small to large system.
- 11) Software complexity can be easily managed.

Disadvantages (Cons/Demerits)

- 1) Main disadvantage of using inheritance is that the two classes (base and inherited class) get tightly coupled. This means one cannot be used independent of each other.
- 2) A change in base class will affect all the child classes.
- 3) It increases the time and efforts take to jump through different levels of the inheritance.so it reduces execution speed.
- 4) Often, data members in the base class are left unused which may lead to memory wastage.
- 5) Inappropriate usage of inheritance will cause complexity in program
- 6) If the methods in the super class are deleted then it is very difficult to maintain the functionality of the child class which has implemented the super class's method.
- 7) Also with time, during maintenance adding new features both base as well as derived classes are required to be changed. If a method signature is changed then we will be affected in both cases (inheritance & composition)

- 1) WAP to concatenate two strings (name and address of a person) using the concept of containership.[2014 fall]

```
#include<iostream>
#include<string.h>
using namespace std;
class first
{
char name[20];
public:
void setn(char n[])
{
strcpy(name,n);
}
char* getn()
{
return (name);
}
};

class second
{
char address[20];
public:
void seta(char a[])
{
strcpy(address,a);
}
char* geta()
{
return (address);
}
};

class concat
{
private:
first f;
second s;
public:
void getinfo(char n[],char a[])
{
f.setn(n);
s.seta(a);
}
```

```

void join()
{
strcat(f.getn(),s.getn());
cout<<"name+address:<<f.getn();
}
};

int main()
{
concat c;
c.getinfo("Ram","kathmandu");
c.join();
return 0;
}

```

Alternative solution:

```

#include<iostream>
#include<string.h>
using namespace std;
class first
{
string name;
public:
void setn(string n)
{
name=n;
}
string getn()
{
return (name);
}
};

```

```

class second
{
string address;
public:
void seta(string a)
{
    address=a;
}
string geta()
{
    return (address);
}
};

class concat
{
private:
string tn,ta;
first f;
second s;
public:
void getinfo( string a,string b)
{
f.setn(a);
s.seta(b);
}
void join()
{
tn=f.getn();
ta=s.geta();
tn=tn+ta;
cout<<"name+address:<<tn;
}
};

int main()
{
concat c;
c.getinfo("Ram","Kathmandu");
c.join();
return 0;
}

```

- 2) Create a class person with data members name, age and address. Create another class teacher with data members Qualification and department .Also create another class Student with data members program and semester. Both class are inherited from class person. Every class has at least one constructor which uses base class constructor. Create member function showdata() in each to display the information of the class member.[PU:2018 fall,2014 spring]

```
#include<iostream>
#include<string.h>
using namespace std;
class Person
{
private:
char name[20];
int age;
char address[20];
public:
Person(char n[],int a,char addr[])
{
    strcpy(name,n);
    age=a;
    strcpy(address,addr);
}
void showdata()
{
cout<<"Name:"<<name<<endl;
cout<<"Age:"<<age<<endl;
cout<<"Address:"<<address<<endl;
}
};
class Teacher:public Person
{
char qualification[20];
char department[20] ;
public:
Teacher(char n[],int a,char addr[],char q[],char d[]):Person(n,a,addr)
{
strcpy(qualification,q);
strcpy(department,d);
}
```

```

void showdata()
{
cout<<"Qualification:"<<qualification<<endl;
cout<<"Department:"<<department<<endl;
}
};

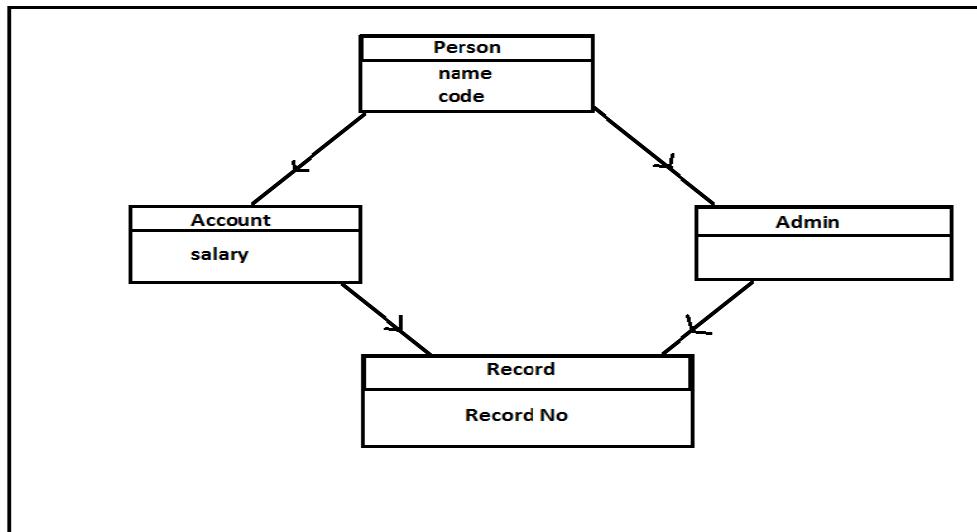
class Student:public Person
{
private:
char program[20];
int sem;
public:
    Student(char n[],int a,char addr[],char p[],int s):Person(n,a,addr)

    {
        strcpy(program,p);
        sem=s;
    }
    void showdata()
    {
        cout<<"Program:"<<program<<endl;
        cout<<"Semester:"<<sem<<endl;
    }
};

int main()
{
Teacher t("Hari",32,"Kathmandu","Master","Civil");
Student s("Ram",24,"Pokhara","Computer",2);
cout<<"Information of Teacher is"<<endl;
t.Person::showdata();
t.showdata();
cout<<"Information of Student is"<<endl;
s.Person::showdata();
s.showdata();
return 0;
}

```

3) Consider the class network of the following figure:



The class **Record** derives the information from both **Account** and **Admin** classes and in turn derive information from the class **Person**. Define all the four classes with at least one parameterized constructor and 'void display' method in each class. In main() function create a object of class '**Record**' and initialize all data members and display them.[PU:2015 spring]

```
#include<iostream>
#include<string.h>
using namespace std;
class Person
{
private:
char name[20];
int code;
public:
Person(char n[],int c)
{
strcpy(name,n);
code=c;
}
void display()
{
cout<<"Name="<<name<<endl;
cout<<"Code="<<code<<endl;
}
};
```

```

class Account: virtual public Person
{
private:
float salary;
public:
Account(char n[],int c,float s):Person(n,c)
{
salary=s;
}
void display()
{
cout<<"salary="<<salary<<endl;
}
};

class Admin:virtual public Person
{
private:
int year;
public:
Admin(char n[],int c,int y):Person(n,c)
{
year=y;
}

void display()
{
cout<<"No of experience year="<<year<<endl;
}
};

class Record:public Account,public Admin
{
private:
int recno;
public:
Record(char n[],int c,float s,int y,int r):Account(n,c,s),Admin(n,c,y),Person(n,c)
{
recno=r;
}
void display()
{
cout<<"Record no="<<recno<<endl;
}
};

```

```

int main()
{
char name[20];
int code,year,recno;
float salary;
cout<<"Enter person name and code"<<endl;
cin>>name>>code;
cout<<"Enter salary"<<endl;
cin>>salary;
cout<<"Enter number of year of experience"<<endl;
cin>>year;
cout<<"Enter Record no"<<endl;
cin>>recno;
Record r1(name,code,salary,year,recno);
r1.Person::display();
r1.Account::display();
r1.Admin::display();
r1.display();
return 0;
}

```

- 4) Write a base class that ask the user to enter time(hour,minute and second) and derived class adds the time of its own with the base.Finally make third class that is friend of derived and calculate the difference of base time and is own time.[PU:2017 fall]**

```

#include<iostream>
using namespace std;
class time1
{
protected:
int hr1,min1,sec1;
public:
void getdata1()
{
cout<<"Enter hour,minute and second for base class"<<endl;
cin>>hr1>>min1>>sec1;
}
void display1()
{
cout<<hr1<<"hours"<<min1<<"minutes"<<sec1<<"seconds"<<endl;
}
};

```

```

class time2:public time1
{
private:
int hr2,min2,sec2;
int hr,min,sec;
public:
void getdata2()
{
cout<<"Enter hour,minute and second for derived class" << endl;
cin>>hr2>>min2>>sec2;
}
void display2()
{
cout<<hr2<<"hours" << min2 << "minutes" << sec2 << "seconds" << endl;
}
void addtime()
{
sec=sec1+sec2;
min=sec/60;
sec=sec%60;
min=min+min1+min2;
hr=min/60;
min=min%60;
hr=hr+hr1+hr2;
cout<<"sum of time:" ;
cout<<hr<<"hours" << min << "minutes" << sec << "seconds" << endl;
}
friend class time3;
};
class time3
{
private:
int hr,min,sec;
int hr3,min3,sec3;
public:
void getdata3()
{
cout<<"Enter hour,minute and second for friend class" << endl;
cin>>hr3>>min3>>sec3;
}

```

```

void timediff(time2 t)
{
if(sec3>t.sec1)
{
--t.min1;
t.sec1=t.sec1+ 60;
}
sec = t.sec1-sec3;
if(min3> t.min1)
{
--t.hr1;
t.min1=t.min1+ 60;
}
min = t.min1-min3;
hr = t.hr1-hr3;
if(hr<0)
{
hr=hr*(-1);
}
cout<<"Time Difference:";
cout<<hr<<"hours"<<min<<"minutes"<<sec<<"seconds"<<endl;
}
};

int main()
{
time2 t2;
time3 t3;
t2.getdata1();
t2.getdata2();
cout<<"Time in base class:";
t2.display1();
cout<<"Time in derived class:";
t2.display2();
t2.addtime();
t3.getdata3();
t3.timediff(t2);
return 0;
}

```

IMPORTANT QUESTIONS FROM THIS CHAPTER

INHERITANCE

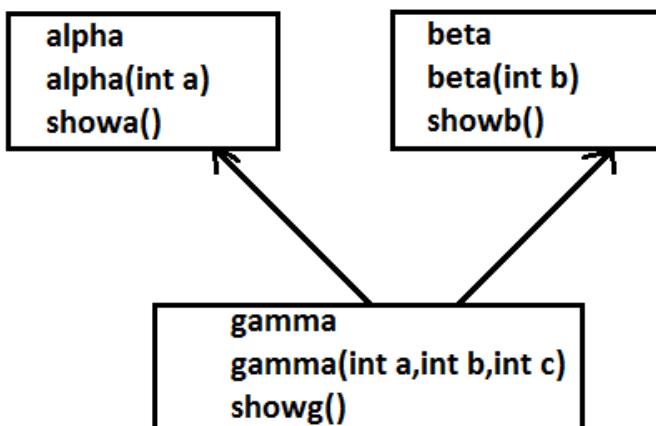
1. “Inheritance allows us to create a hierarchy of classes. Justify this statement. Discuss private and public inheritance.[PU:2016 spring]
2. How does visibility mode control the access of members in the derived class? Explain with an example.[PU 2017 spring]
3. Explain hybrid inheritance with example.[2009 spring]
4. What is hybrid Inheritance. Explain any three pros and three cons of inheritance. [PU: 2010 fall]
5. How inheritance support reusability features of OOP? Explain with example.[PU:2010 spring]
6. When base class and derived class have the same function name what happens when derived class object calls the function?[PU 2017 fall]
7. Explain how inheritance support Reusability? Describe the syntax of multiple and multilevel inheritance?[PU:2015 fall]
8. Inheritance supports characteristic of OOP. Justify your answer. Explain ambiguity that occurs in multiple inheritance.[PU:2017 spring]
9. “Ambiguity is essential evil” ”,Explain by example how it can effectively solve in complex programming?[PU: 2015 spring]
10. Explain why multiple inheritance is dangerous?
11. During the time of hybrid inheritance when there is hierarchical inheritance at the upper level and multiple inheritance at lower level, ambiguity occurs due to the duplication of data from multiple path at the grand child class. How this kind of ambiguity is resolved? Explain with suitable example?
12. Does ambiguity occurs in hybrid inheritance? If yes, how can you remove this? Explain with an example.[PU 2018 fall]
13. Under what condition virtual base class is created? Explain with suitable example.[PU:2017 fall,2019 fall,2014 fall]
14. How are arguments are sent to base constructors in multiple inheritance ?Who is responsibility of it.[PU:2013 spring]
15. How does inheritance influence working of constructors and destructors? Class ‘Y’ has been derived from class ‘X’ .The class ‘Y’ does not contain any data members of its own. Does the class ‘Y’ require constructors? If yes why.[PU:2013 spring]
16. What is containership? How does it differ from inheritance, describe how an object of a class that contain object of another classes are created.[PU:2013 fall]
17. How composition differs from inheritance?
18. Explain how composition provide reusability?[PU:2018 fall]
19. Compare and contrast composition and inheritance?[PU:2015 fall]

20. Distinguish between subclass and subtype in light of principle of substitutability. Support your answer with suitable example.[PU:2006 spring,2016 spring]
21. Differentiate between

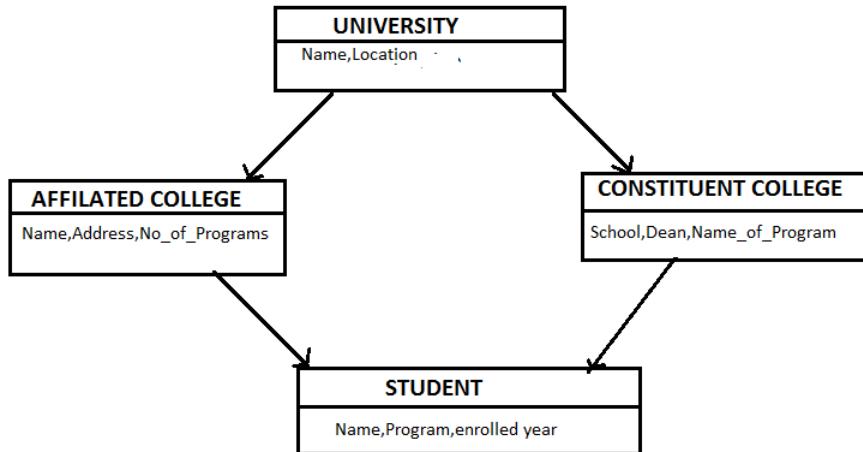
- subclass and subtype.
 - Is a rule and has a rule
22. State principle of substitutability .Explain sub-classing for specialization, generalization.
List out disadvantages of inheritance.[PU:2016 fall]
23. What is inheritance? What are the different forms of inheritance?
[PU: 2016 spring,PU:2015 spring]
24. Differentiate between is a rule and has a rule with suitable example.
[PU:2015 fall,2014 spring]
- 25. Write a short notes on:**
- Containership[PU:2010 fall]
 - Subclass-subtype
 - Software reusability[PU:2005 fall]
 - Is a rule and has a rule[PU:2009 fall,2016 fall, 2016 spring,2015 spring]
 - Hybrid inheritance[PU:2006 spring]ii
 - Inheritance and substitutability
 - Generalization[PU:2013 spring]

Programming Questions:

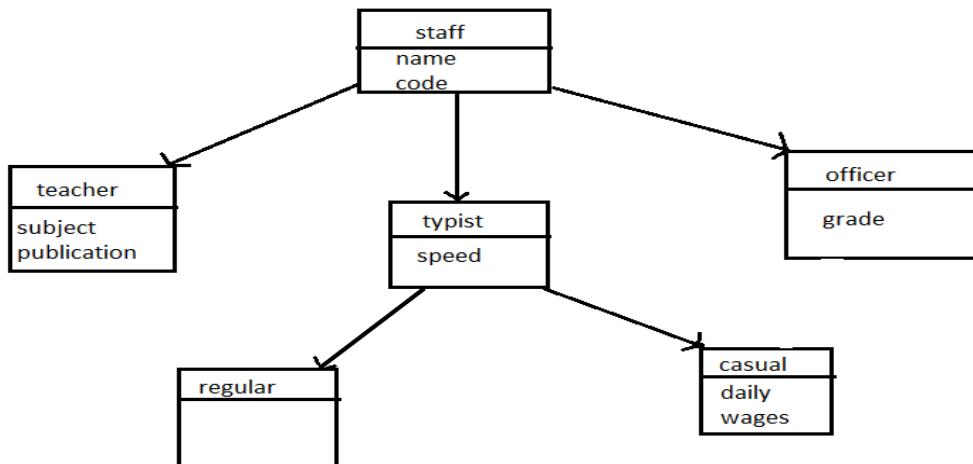
- 1) WAP to enter information of n students and then display is using the concept multiple inheritance,[PU: 2015 fall]
- 2) Write a complete program with reference to the given figure.



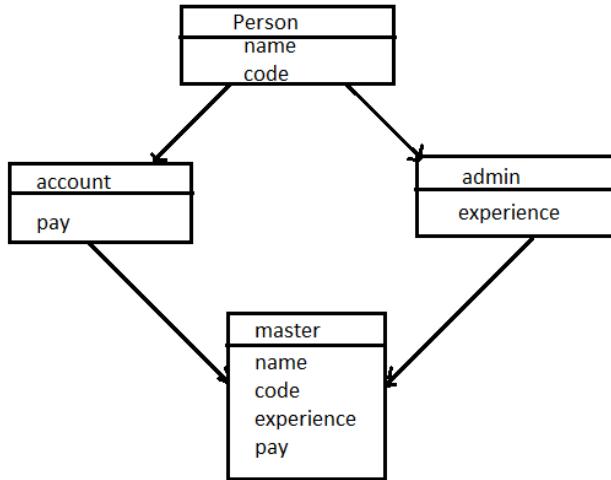
- 3) The following figure shows the minimum information required for each class. Write a program by realizing the necessary member functions to read and display information of individual object. Every class should contain at least one constructor and should be inherited from other classes as well.[PU:2019 fall]



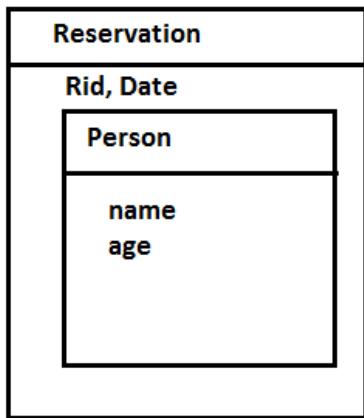
- 4) An educational institution wishes to maintain a database of its employees. The database is divided into a number of classes whose hierarchical relationship are shown below. The figure also shows minimum information requires for each class. Specify all the classes and define functions to create database and retrieve individual information when required.



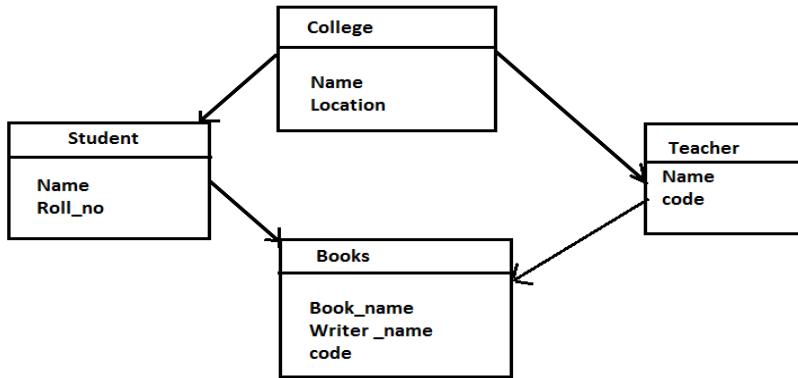
- 5) The following figure shows minimum information required for each class.
- Write a Program to realize the above program with necessary member functions to create the database and retrieve individual information



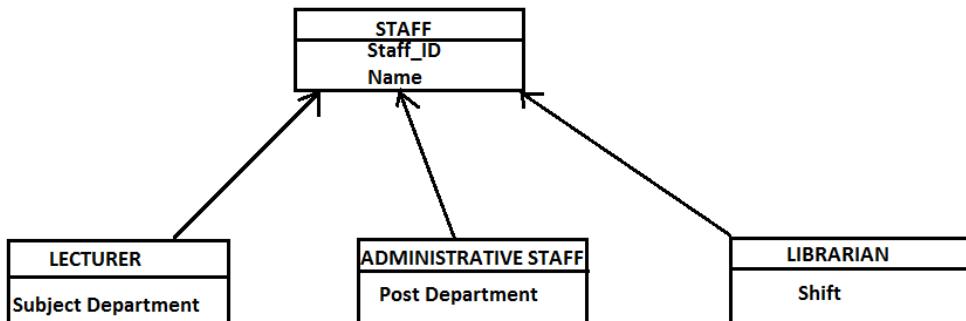
- ii) Rewrite the above program using constructor on each class to initialize the data members.
- 6) Write a program that allow you to book a ticket for person and use two classes PERSON, RESERVATION. Class RESERVATION is composite class/ container class.



- 7) The following figure shows the minimum information required for each class. Write a program to realize the above program with necessary member functions to create the database and retrieve individual information .Every class should contain at least one constructor and should be inherited to other classes as well.[PU:2010 spring][PU 2009 fall]



- 8) Develop a complete program for an institution, which wishes to maintain a database of its staff. The database is divided into number of classes whose hierarchical relationship is shown in the following diagram. specify all classes and define constructors and functions to create database and retrieve the individual information as per requirements.



- 9) Develop a complete program for an institution which wishes to maintain a database of its staff. Declare a base class STAFF which include staff_id and name. Now develop a records for the following staffs with the given information below.

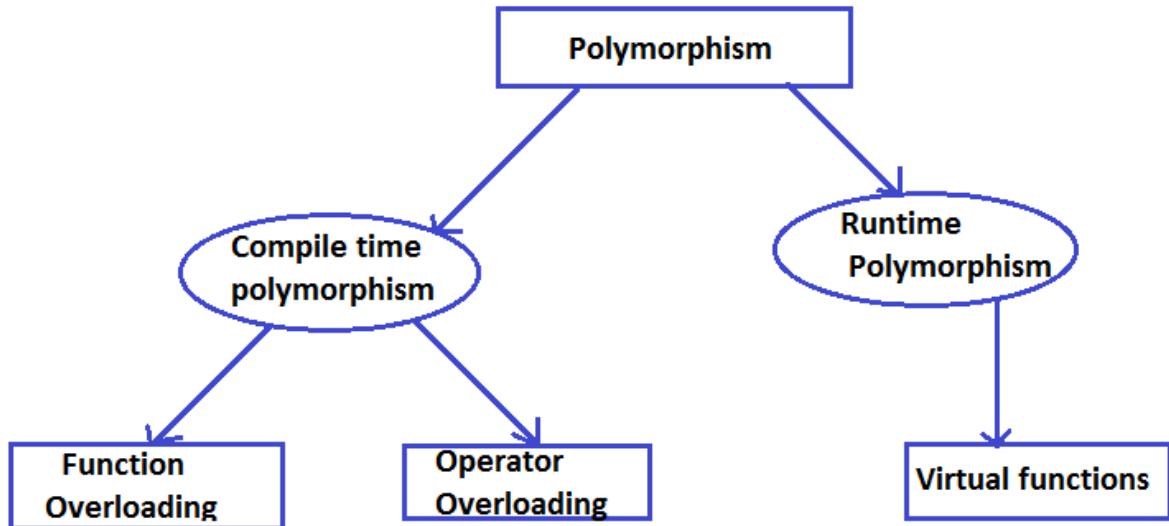
- i. Lecturer(subject,department)
- ii. Administrative staff (Post,department)

CHAPTER 5

POLYMORPHISM

Introduction:

- Polymorphism means ‘One name-multiple forms’
- Greek word “**poly**” means many and “**morphos**” means form.
- C++ polymorphism means that a call to member function will cause a different function to be executed depending on the type of object invokes the functions.
- There are two types of polymorphism. They are compile time polymorphism and runtime polymorphism.



Compile time Polymorphism

- Compile time polymorphism means that an object is bound to 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 called early binding or static binding or static linking.
- Compile time polymorphism is achieved in two ways.
 - Function overloading
 - Operator overloading

Runtime Polymorphism

- The selection of appropriate function is done dynamically at runtime.
- Thus it is not known which function will be invoked till an object actually makes a function call during program execution.
- This mechanism is also called late binding or dynamic binding.
- Runtime polymorphism can be achieved with the help of virtual functions.

Function Overloading

- The method of using same function name but with different parameter list along with different data type to perform the variety of different tasks is known as function overloading.
- The correct function to be invoked is determined by checking the number and type of arguments but not function return type.

```
#include<iostream>
using namespace std;
class calcarea
{
public:
int area(int);
int area(int,int);
float area(float);
};
int calcarea::area(int s)
{
    return(s*s);
}
int calcarea::area(int l,int b)
{
    return(l*b);
}
float calcarea::area(float r)
{
    return(3.14*r*r);
}
int main()
{
calcarea c1,c2,c3;
cout<<"Area of square="<<c1.area(5)<<endl;
cout<<"Area of Rectangle="<<c2.area(5,10)<<endl;
cout<<"Area of Circle="<<c3.area(2.5f)<<endl;
return 0;}
```

Operator overloading

- The mechanism of adding special meaning to an operator is called operator overloading.
- It provides a flexibility for the creation of new definitions 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 types.

Things to be understand

- Usually Operations can perform only on basic data type. Example int a,b,c;
c=a+b; But if we declare a class complex {} and complex c1,c2,c3 ;
c3=c1+c2;is not possible because object is user defined data type.
- Operator overloading helps to define usage of operator for user defined data type i.e. objects.
- After overloading operands used with operator are objects instead of basic data types.

Operators that cannot be overloaded

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

- Class member access operators(. , .*)
- Scope resolution Operator ()::
- Sizeof operator(**sizeof**)
- Conditional Operator(?:)

General form of Operator function

The general form of operator function

```
return_type operator op(arglist)
{
    function body //task defined
}

OR

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

Where,

- **return_type** is the type of value returned by the specified operation.
- **op** is the operator being overloaded.
- **Operator op** is function name, where **operator** is keyword.

Note:

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

This is because the object used to invoke the member function is passed implicitly and therefore available for member function. This is not the case with friend functions.

Arguments may be passed either by value or by reference. Operator functions are declared in class using prototype as follows.

```
vector operator+(vector);    //vector addition
vector operator-();          //unary minus
friend vector operator+(vector,vector);    //vector addition
friend vector operator-(vector);          //unary minus
vector operator-(vector &a);            //subtraction
int operator==(vector);                //comparison
friend int operator==(vector,vector);   //comparison
```

vector is a datatype of class and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements(as in mathematics).

The process of overloading involves the following steps:

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 a **friend** function.
3. Define operator function to implement the required operations.

Operator overloaded function can be invoked using expression such as:

	In case of member function	In case of friend function
For unary operators	op x or x op (eg.++x, or x++) <i>x.operator op();</i>	op x or x op (eg.++x, or x++) <i>operator op(x)</i>
For binary operators	x op y eg x+y <i>x.operator op(y);</i>	x op y eg x+y <i>operator op(x,y)</i>

Note:

- Here **op** represents the operator being overloaded.
- **x** and **y** represents the object.

Overloading Unary Operators

Unary operators operates on single operand. Some of unary operators are

- Increment operator (++)
- Decrement operator (--)
- Unary minus operator(-)

Eg. `++a` ; Here a is only one operand.

Overloading unary minus Operator

Overloading unary minus using member function

```
#include<iostream>
using namespace std;
class space
{
private:
int x;
int y;
int 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;
```

Overloading unary minus using friend function

```
#include<iostream>
using namespace std;
class space
{
private:
int x;
int y;
int z;
public:
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<<endl;
    cout<<"y="<<y<<endl;
    cout<<"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;
}

```

**WAP to overload unary (-) minus operator so that the statement $s2 = -s1$ can be returned when $s1$ and $s2$ are of type space that represent 3 dimensional coordinate system.
(using friend function)**

```

#include<iostream>
using namespace std;
class space
{
private:
    int x;
    int y;
    int 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;
}
void space::display()
{
cout<<"x="<<x<<endl;
cout<<"y="<<y<<endl;
cout<<"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();
cout<<"-s:"<<endl;
s2=-s1;
s2.display();
return 0;
}

```

Write a simple program to overload (unary++) operator. [PU:2016 spring]

OR

Overloading for prefix increment (++) operator by 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;
}
```

Output

Count=3

Count =4

Count =4

Unary operator overloading for postfix increment 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++(int);
};
counter counter::operator ++(int)
{
counter temp;
temp.count=count++;
return temp;
}
int main( )
{
counter c1,c2;
c1.getdata(5);
c1.showdata();
c2=c1++;
c1.showdata();
c2.showdata();
return 0;
}
```

Output:

```
count =5
count =6
count =5
```

Overloading for prefix increment (++) operator with no return type

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

Output

```
Count=3
Count=4
```

Unary operator overloading for postfix increment(++) Operator with no return type

```
#include<iostream>
using namespace std;
class counter
{
private:
int count;
public:
void getdata (int x)
{
count=x;
}
void showdata( )
{
cout<<"count ="<<count<<endl;
}
void operator++(int);
};
void counter::operator ++(int)
{
count++;
}
int main( )
{
counter c1;
c1.getdata(5);
c1.showdata();
c1++;
c1.showdata();
return 0;
}
```

Output :

```
Count =5
Count=6
```

Overloading for prefix increment (++) operator using friend function

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

Output:

Count=5

Count=6

**Write a program to generate Fibonacci series using operator overloading of (++) operator.
Which type of overloading is it.[PU:2009 fall]**

```
#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<<",";
}
void operator++()
{
a=b;
b=c;
c=a+b;
};
int main()
{
    int n,i;
    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.

Overloading + operator

Write a program to add two complex number using binary operator overloading.

[PU:2013 fall]

```
#include<iostream>
using namespace std;
class complex
{
private:
    int real,imag;
public:
    complex()
    {
    }
    complex(int r,int i)
    {
        real=r;
        imag=i;
    }
    void display()
    {
        cout<<real<<"+"<<imag<<endl;
    }
    complex operator +(complex);
};
complex complex::operator +(complex c2)
{
    complex temp;
    temp.real=real+c2.real;
    temp.imag=imag+c2.imag;
    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;
}

```

Overloading Binary operator (+) using friends

```

#include <iostream>
using namespace std;
class complex
{
private:
int real ;
int imag ;
public:
complex()
{
}
complex(int r, int i)
{
real=r;
imag=i;
}
void display()
{
cout<<real<< "+i" << imag << endl;
}
friend complex operator+ (complex,complex) ;
};

```

```
complex operator + (complex c1, complex c2)
{
    complex temp ;
    temp.real=c1.real+c2.real;
    temp.imag=c1.imag+c2.imag;
    return temp;
}
```

```
int main()
{
    complex c1(5,3);
    complex c2(3,4);
    complex c3;
    c3=c1+c2;
    cout<<"c1=";
    c1.display();
    cout<<"c2=";
    c2.display();
    cout<<"c3=";
    c3.display();
    return 0;
}
```

Write a program to overload the arithmetic operators(+,-,* ./)

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

```

Arithmetic Arithmetic::operator/(Arithmetic b)
{
    Arithmetic temp;
    temp.num=num/b.num;
    return temp;
}

int main()
{
    Arithmetic a,b,c;
    a.getdata();
    b.getdata();
    c=a+b;
    cout<<"Additon of two objects="<<endl;
    c.putdata();
    cout<<"Multiplication of two objects="<<endl;
    c=a*b;
    c.putdata();
    cout<<"Substraction of two objects="<<endl;
    c=a-b;
    c.putdata();
    cout<<"Division of two objects="<<endl;
    c=a/b;
    c.putdata();
    return 0;
}

```

WAP to overload two binary operator '+' and '-' so that statement

c3=c1+c2;
c3=c1-c2; exists

OR

Write a program to find the sum and difference of any two complex number by overloading '+' and '-' operator .[PU:2019 fall]

```
#include<iostream>
using namespace std;
class complex
{
private:
    int real,imag;
public:
    complex()
    {
    }
    complex(int r,int i)
    {
        real=r;
        imag=i;
    }
    void display()
    {
        cout<<real<<"+"<<imag<<endl;
    }
    friend complex operator +(complex c1,complex c2);
    friend complex operator -(complex c1,complex c2);
};
complex operator +(complex c1,complex c2)
{
    complex temp;
    temp.real=c1.real+c2.real;
    temp.imag=c1.imag+c2.imag;
    return temp;
}
complex operator -(complex c1,complex c2)
{
    complex temp;
    temp.real=c1.real-c2.real;
    temp.imag=c1.imag-c2.imag;
    return temp;
}
```

```

int main()
{
complex c1(15,10);
complex c2(10,5);
complex c3;
c3=c1+c2;
cout<<"After overloading + operator"<<endl;
c3.display();
c3=c1-c2;
cout<<"After overloading - operator"<<endl;
c3.display();
return 0;
}

```

Write a class Time with three integer attributes hour, minute and second. Include the following responsibilities in class

- i. Default and parameterized constructor
- ii. Display method to display time in hour, minute and second format.
- iii. Appropriate function overload to realize addition of two time objects with ‘+’ operator[PU:2013 spring]

```

#include<iostream>
using namespace std;
#include <conio.h>
class Time
{
private:
    int hrs,mins,secs;
public:
    Time()
    {
        hrs=0, mins=0; secs=0;
    }
    Time(int h,int m,int s)
    {
        hrs=h;
        mins=m;
        secs=s;
    }
    void display()
    {
        cout<< hrs<< ":"<<mins<< ":"<< secs<<endl;
    }
    Time operator+(Time);
};

```

```

Time Time::operator+(Time t1)
{
    Time temp;
    temp.secs=secs+t1.secs;
    temp.mins=temp.secs/60;
    temp.secs=temp.secs%60;
    temp.mins=temp.mins+mins+t1.mins;
    temp.hrs=temp.mins/60;
    temp.mins=temp.mins%60;
    temp.hrs=hrs+temp.hrs+t1.hrs;
    return temp;
}

int main()
{
    Time t1(5,25,45);
    Time t2(2,35,10);
    Time t3;
    t3 = t1 + t2;
    cout<<"First time=";
    t1.display();
    cout << "Second time=";
    t2.display();
    cout << "Sum =" ;
    t3.display();
    return 0;
}

```

Alternative solution:

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

Time operator+(Time t1,Time t2)
{
    Time temp;
    temp.secs=t1.secs+t2.secs;
    temp.mins=temp.secs/60;
    temp.secs=temp.secs%60;
    temp.mins=temp.mins+t1.mins+t2.mins;
    temp.hrs=temp.mins/60;
    temp.mins=temp.mins%60;
    temp.hrs=temp.hrs+t1.hrs+t2.hrs;
    return temp;
}
```

```

int main()
{
    Time t1(5,25,45);
    Time t2(2,35,15);
    Time t3;
    t3 = t1 + t2;
    cout<<"First time=";
    t1.display();
    cout << "Second time=";
    t2.display();
    cout << "Sum =";
    t3.display();
    return 0;
}

```

Write a program to overload ‘+=’ operator to add distance of two objects.

```

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

```

```

void dist::operator+=(dist d)
{
feet+=d.feet;
inch+= d.inch;
if(inch>=12)
{
inch-=12;
feet++;
}
}
int main()
{
dist d1(5,9);
dist d2(10,5);
cout <<"first distance=";
d1.display();
cout <<"second distance=";
d2.display();
d1 += d2;
cout <<"After addition d1=";
d1.display();
return 0;
}

```

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[])
{
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("pradip");
stringc s2("Paudel");
stringc s3;
s1.display();
s2.display();
s3=s1+s2;
s3.display();
return 0;
}

```

Write a program to overload == 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;
    }
    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 are not equal";
    }
    return 0;
}

```

WAP to overload > operator.

```

#include<iostream>
using namespace std;
class maximum
{
private:
int x;
public:
maximum(int a)
{
    x=a;
}

```

```

void operator >(maximum m2)
{
if(x>m2.x)
cout<<"Maximum number is "<<x;
else
cout<<"Maximum number is "<<m2.x;
}
};

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 eg 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 x=3.14159

m=x;

```

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 conversion of user defined data type.

There are three possible type conversions are:

- Conversion from basic type to class type
- Conversion from class type to basic type
- 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.

Example1:

```
#include<iostream>
using namespace std;
class time
{
int hours;
int minutes;
public:
time(int t)
{
hours=t/60;
minutes=t%60;
}
void display()
{
cout<<"Hours="<<hours<<endl;
cout<<"Minutes="<<minutes<<endl;
}
};
int main()
{
int duration=65;
time t1=duration;
t1.display();
return 0;
}
```

After conversion, the hours members of t1 contains the value 1 and minutes member contains the value 5, denoting 1 hours 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();
}
```

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 megabytes 38 kilobytes 177 bytes.

```

#include<iostream>
using namespace std;
class memory
{
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;
}

```

Class to Basic type

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

```

operator typename()
{
.....
//function statemets
.....
}

```

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 specify return type.
- It must not have any arguments.

Example:

```
#include<iostream>
using namespace std;
class item
{
private:
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;
}
```

Example 2:

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

```
#include <iostream>
using namespace std;
class dist
{
int feet;
float inches;
public:
dist(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( )
{
dist 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 to another class type

Define type conversion. Explain with conversion from one class type to another class type.

[PU: 2016 fall]

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

Example:

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

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 source class and X is known as destination class.

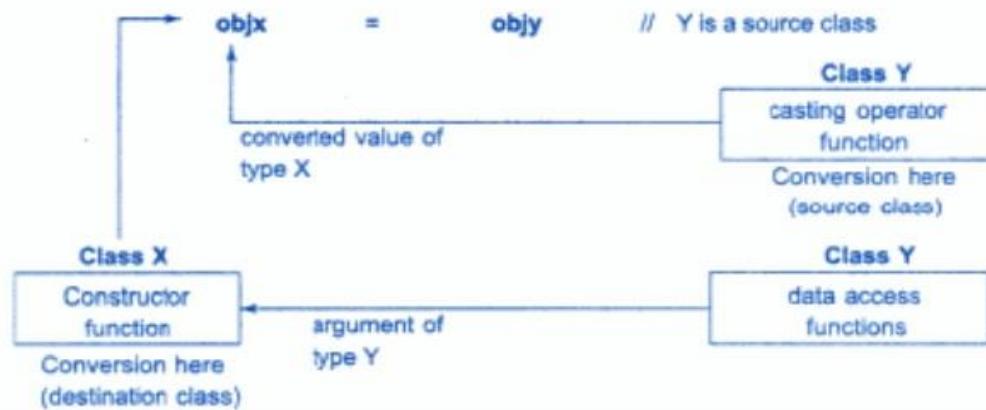
Such conversions between objects of different classes can be carried out by either a constructor or a conversion function.

Note:

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

One class to Another class type

- When to use constructor and type conversion function?



Conversion Routine in source class (*conversion function in source class*)

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;
cout<<"Polar coordinates=";
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()
    {
        radius=0.0;
        angle=0.0;
    }

```

```

Polar(float r,float a)
{
    radius=r;
    angle=a;
}
void display()
{
    cout<<"(" <<radius <<"," <<angle <<")" <<endl;
}
};

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;
}
operator Polar()
{
    float a=atan(yco/xco);
    float r=sqrt(xco*xco+yco*yco);
    return Polar(r,a);
}
};

```

```

int main()
{
    Rectangle r(7.07107,7.07107);
    Polar p;
    p=r;
    cout<<"Rectangular coordinates=";
    r.display();
    cout<<"Polar coordinates=";
    p.display();
    return 0;
}

```

Conversion Routine in destination class

Conversion from polar to rectangle (using conversion routine in Rectangle)

```

#include <iostream>
#include <math.h>
using namespace std;
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;
    }
    float getr()
    {
        return radius;
    }
}

```

```

float geta()
{
return angle;
}
};

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

Rectangle(Polar p)
{
float r=p.getr();
float a=p.geta();
xco=r*cos(a);
yco=r*sin(a);
}
};

int main()
{
Polar p(10.0,0.785398);
Rectangle r;
r=p;
cout<<"Polar coordinates=";
p.display();
cout<<"Rectungular coordinates=";
r.display();
return 0;
}

```

Conversion from rectangle to polar (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;
}
float getx()
{
    return xco;
}
float gety()
{
    return yco;
}
};
```

```

class Polar
{
private:
    float radius;
    float angle;
public:
    Polar()
    {
        radius=0.0;
        angle=0.0; }
    void display()
    {
        cout<<"(" <<radius <<"," <<angle <<")" <<endl;
    }

    Polar(Rectangle r)
    {
        float x=r.getx();
        float y=r.gety();
        angle=atan(y/x);
        radius=sqrt(x*x+y*y);
    }
};

int main()
{
    Rectangle r(7.07107,7.07107);
    Polar p;
    p=r;
    cout<<"Rectangular coordinates=";
    r.display();
    cout<<"Polar coordinates=";
    p.display();
    return 0;
}

```

Pointer to objects (object pointer)

The pointer pointing to objects are referred to as object pointer.

Declaration

```
Class_name *object_pointer_name;
```

```
Eg. student *ptr;
```

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

Initialization

```
object_pointer_name=&object;
```

```
Eg. ptr=&st;
```

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

Note: When accessing members of class using 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 student Name"<<endl;
cin>>name;
cout<<"Enter student Rollno"<<endl;
cin>>roll;
}
```

```

void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Rollno:"<<roll<<endl;
}
};

int main()
{
student st;
student *ptr;
ptr=&st;
ptr->getdata();
ptr->display();
return 0;
}

```

Creating objects at runtime using object pointers

Object pointers are also used to create the object at runtime by using it with **new** operator as follows:

Eg. item *ptr=new item;

This statement allocates the 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 example, the statement

```
item *ptr=new item[10];
```

Creates memory space for an array of 10 objects of item.

Program:

```

#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; //&ptr[0]
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

Can you derive a pointer from a base class? Explain with 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 single pointer variable can be made to point to objects belonging to different classes. For example, if **B** is a base class and **D** is the derived class from **B**, then a pointer declared as a pointer to **B** can also be a pointer to **D**. consider the following declarations.

```
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 the class **B**.

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.

Program:

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

```

class D:public B
{
public:
int y;
void display()
{
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
}
};

int main ()
{
    B b1;
    B *bptr; //base pointer
    bptr=&b1; //base address
    bptr->x = 10; //access B via base pointer
    cout<<"bptr points to base object"<<endl;
    bptr->display();

    D d1;
    bptr=&d1; //address of derived object
    bptr->x=10; //access D via base pointer
    //bptr->y = 20; wont work
    cout<<"bptr now points to derived object"<<endl;
    bptr->display(); //access to base class function

    D *dptr; //derived type pointer
    dptr=&d1;
    dptr->y=20;
    cout<<"dptr is a derived type pointer"<<endl;
    dptr->display();

    cout<<"using ((D*)bptr)"<<endl;
    ((D*)bptr)->y=30;
    ((D*)bptr)->display();
    return 0;
}

The statements,
dptr->display();
((D*)bptr)->display(); //cast bptr to D type

```

Displays the contents of **derived** object.

This shows that, although a base pointer cannot directly access the members defined by a derived class. But it can be made to point any number of derived objects, so that we can access all the members of derived class.

Alternative program for pointer to Derived objects:

```
#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 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)bprr)"<<endl;
((D*)bprr)->display();
return 0;
}

```

this pointer

- **this** pointer stores the address of current calling object.
- 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. The 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.

Application:

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;
}
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 points to.

For example, the statement

return *this;

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

Example:

```

#include<iostream>
#include<string.h>
using namespace std;
class person
{
char name[20];
float age;
public:
person()
{
}
person (char n[],float a)
{
strcpy(name,n);
age=a;
}

person greater(person x)
{
    if(x.age>=age)
    {
        return x;
    }
    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<<"Elder person is:"<<endl;
p3.display();
return 0;
}

```

Things to be understand:

In addition to the explicit parameters in their argument lists, every class member function (method) receives an additional hidden parameter, the "this" pointer. The "this" pointer addresses the object on which the method was called. Thus, this pointer is used as a pointer to the class object instance by the member function.

Each object maintains its own set of data members, but all objects of a class share a single set of methods. This is, a single copy of each method exists within the machine code that is the output of compilation and linking. A natural question is then if only a single copy of each method exists, and its used by multiple objects, how are the proper data members accessed and updated. The compiler uses the "this" pointer to internally reference the data members of a particular object.

The member functions of every object have access to a pointer named this, which points to the object itself. When we call a member function, it comes into existence with the value of this set to the address of the object for which it was called. The this pointer can be treated like any other pointer to an object

Virtual Functions

- Virtual means existing in appearance but not in reality.
- Virtual function is declared by using a keyword 'virtual' preceding 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 by using the 'virtual' function. When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer. In this way runtime polymorphism can also achieved.

```
#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;
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 the 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. That is to say. We cannot use a pointer to derived class to access an object of base type.

1. **What is virtual function? When and how to we make function virtual? Explain with suitable example. PU:2010 spring]**

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

When we use the same function name in both base and derived classes, the function in 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 used 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 versions 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 determined at run time.

Program:

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

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

```
#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 derived"<<endl;
}
void show()
{
cout<<"Show derived"<<endl;
}
};
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 the base  
Display base  
Show base  
bptr points to derived  
Display base  
Show derived
```

Note:

When bptr is made to point the object d (ie. bptr=&d1),
the statement
bptr->display();
Calls only the function associated with the B. (ie.B::display())

This is because 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 first the base pointer has address of the base class object, then its content is changed to contain the address of the derived object.

3. How can you achieve runtime polymorphism in C++? Discuss with 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 type of parameters in both base and derived classes. The function in the base class is declared as virtual using the keyword virtual.

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

```

#include<iostream>
using namespace std;
class base
{
public:
virtual void show()
{
cout<<"Base Class Show function"<<endl;
}
};

class derived:public base
{
public:
void show()
{
cout<<"Derived Class Show function "<<endl;
}
};

int main()
{
base b1,*bptr;
derived d1;
bptr=&b1;
bptr->show();
bptr=&d1;
bptr->show();
return 0;
}

```

Output:

Base Class Show function
Derived Class Show function

A bookshop sells both books and video tapes. 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;
}
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:
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 media:" << endl;
m[1] = &t;
m[1]->display();
return 0;
}

```

Pure virtual function (Deferred methods/Abstract methods)

- A virtual function will become pure virtual function when we append "=0" at the end of declaration of 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()
{
    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;
}

```

Abstract class

- 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 class can be created for selecting the proper virtual function.
- An abstract class is designed to act only as base class .It is a design concept in program development and provides a base upon which program is built.
- Sometimes implementation of all function 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 implementation of function draw() in Shape, but we know every derived class must have implementation of draw().In this case concept of abstract class is used.

Example:

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

class square:public shape
{
public:
void draw()
{
    cout<<"Implementing method to draw square"<<endl;
}
};

class circle:public shape
{
public:
void draw()
{
    cout<<"Implementing method to draw circle"<<endl;
}
};

int main()
{
    shape *bptr;
    square s;
    circle c;
    bptr=&s;
    bptr->draw();
    bptr=&c;
    bptr->draw();
    return 0;
}
```

Function overriding (Overriding base class members)

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be overridden, and this mechanism is called Function Overriding.

Requirements for Overriding a Function

- Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
- Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

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

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

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

int main()
{
B b;
b.display();
return 0;
}
```

Here, the function `display()` is overridden.

- If the function is invoked from an object of the derived class, then the function in the derived is executed.
- If the function is invoked from an object of the base class, then the base class member function is invoked.

Virtual Constructors

A constructors cannot be virtual due to the following reasons.

- To create an object of the constructor of the object class must be of the same type as class. But it is not possible with virtually implemented constructor.
- At the time of calling constructor, the virtual table would not have been to create any function calls.

Whereas destructor can be virtual.

Let's first see what happens when we do not have a virtual Base class destructor.

```
class Base
{
public:
~Base()
{
    cout << "Base class destructor" << endl;
}
};

class Derived:public Base
{
public:
~Derived()
{
    cout << "Derived class destructor" << endl;
}
};

int main()
{
    Base* ptr = new Derived; //Base class pointer points to derived class object
    delete ptr;
}

Output:
Base class Destructor
```

In the above example, delete ptr will only call the Base class destructor, which is undesirable because, then the object of Derived class remains un-destructed, because its destructor is never called. Which leads to memory leak situation.

To make sure that the derived class destructor is mandatory called, we make base class destructor as virtual

```
class Base
{
public:
virtual ~Base()
{
    cout << "Base class destructor" << endl;
}
};
```

Output:..

```
Derived class Destructor
Base class Destructor
```

When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behavior.

Previous Old Question from this chapter

- 1) What is polymorphism? Differentiate between compile time and runtime polymorphism with program in each.
- 2) How can polymorphism be achieved during compile time and during runtime? Explain with examples in C++. [PU:2013 spring]
- 3) What are the advantages of using runtime polymorphism over compile time polymorphism. How does overloading differ from overriding. Explain. [PU:2014 spring]
- 4) Explain importance of polymorphism with a suitable example. [PU:2009 fall]
- 5) What is the difference between static binding and runtime binding? Explain with suitable code. [PU:2013 fall]
- 6) How can you use operator overloading in C++? Give syntax
- 7) What do you mean by operator overloading? How do you overload the + operator in main program ($c3=c1+c2$). so that $c3$ can store a complex number obtained by adding $c2$ and $c2$. [PU: 2006 spring]
- 8) What is polymorphism? How operator overloading is used to support polymorphism. Explain it by overloading '+' operator to concatenate two strings. [PU:2017 fall]
- 9) What is type casting? [PU:2018 fall]
- 10) Define type conversion .Explain with example conversion from one class type to another class type. [PU:2016 fall]
- 11) Can you derive a pointer from base class? Explain with suitable example. [PU:2014 spring]

- 12) How does polymorphism play constructive role in application development? Which type of polymorphism is essential for the computation of distance among two cities from the specific location. The unit of measurements are feet and inch.(Also use standard unit if essential) [PU:2015 spring]
- 13) What is pure virtual function? What is the difference between compile time and runtime polymorphism? Which is best and why?
- 14) What is virtual function? When do we make a function virtual and when we make function pure virtual ? Explain with suitable example.[PU:2010 spring]
- 15) What is compile time and runtime polymorphism? How can you achieve runtime polymorphism in C++?Explain deferred methods.
- 16) When do you use virtual function? How does it provides runtime polymorphism. Explain with suitable example.[PU:2016 fall]
- 17) Discuss the role of virtual function in c++ to cause dynamic polymorphism. Show with example the how it is different from the compile time polymorphism.[PU:2006 spring]
- 18) What is virtual function? When do we make a function virtual? Explain with suitable example.[PU: 2014 spring]
- 19) How can you define pure virtual function in C++? The pure virtual function do nothing but it is defined in base class why?[PU:2015 spring]
- 20)** What is an abstract class? How does it differ from virtual base class? Explain .[PU:2006 spring]
- 21) When base class and derived class have same function name ,what happens when the derived class object class the functions? Differentiate overloading and overriding.[PU:2017 fall]
- 22) Describe overriding. How do you differentiate function overloading from function overriding. Explain with suitable example.[PU:2016 spring]
- 23) Why is ‘this’ Pointer is widely used than object pointer? Write a program to implement pure polymorphism.[PU:2019 fall]
- 24) Define role of this pointer and pure abstract class in object oriented programming to create multiple object with suitable program.[PU:2015 spring]

Write a short notes on:

- Virtual function vs friend function[PU:2016 spring]
- Virtual function vs pure virtual function[PU:2016 spring]
- Deferred methods[PU:2006 spring]
- Virtual functions[2015 fall]
- Operator overloading [PU:2016 fall]
- Pure polymorphism[Pu: 2017 spring]
- Overriding [PU:2018 fall]
- Virtual Destructor[PU:2013 fall]

Program

1. Write a program to generate Fibonacci series using operator overloading of `++` operator.
Which type of overloading is it.[PU:2009 fall]
2. Write a program to add two complex number using operator overloading.
[PU:2010 spring]
3. Write a program to add two complex number using binary operator overloading.[PU:2013 fall]
4. Write a program with class fibo to realize the following code snippet.

```
fibo f=1;
for (i=1;i<=10;i++)
{
    ++f;
    f.display();
}
```

[Hint: overload `++` operator and conversion technique.] [PU: 2014 fall]
5. Design a soccer player class that includes three integer fields:a player's jersey number,number of goals,number of assists and necessary constructors to initialize the data members.Overload the `>` operator (Greater than) .One player is considered greater than another if the sum of goals plus assists is greater than that of others.Create an array of 11 soccer players,then use the overloaded `>` operator to find the greater total of goal plus assists.[PU:2015 fall]
6. Write a simple program to overload unary `++` operator.[PU: 2016 spring]
7. Make a class called memory with member data to represent bytes, kilobytes and megabytes.Read the value of memory in bytes from the user as basic types and display the result in user defined memory type. Like for m (basic type) = 108766, your program should display as: 1 megabyte 38 kilobytes 177 bytes. [Hint: Use basic to user defined data conversion method.]
8. Write a program that converts object that represents 24 hrs times to 12 hrs times and vice versa.
9. Write a program to create a class age with attributes YY, MM, DD and convert the object of class age to basic data type int days.
10. Write a program finding area of square ,rectangle, triangle. Use function overloading technique.
11. Write a program to implement vector addition using operator overloading
 - i. Using Friend Function
 - ii. Without using Friend Function(using member function)

12. A bookshop sells both books and video tapes. 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).
13. Create a base class student. Use the class to store name, dob, rollno and includes the member function getdata(), discount(). Derive two classes PG and UG from student. make dispresult() as virtual function in the derived class to suit the requirement.
14. Define two class names 'Polar' and 'Rectangle' to represent points in polar and Rectangle systems. Use conversion Routines to convert from one system to another system.
15. Write a complete program to convert the polar coordinates into rectangular coordinates.
16. (hint: polar-coordinates(radius,angle) and rectangular co-ordinates (x,y) where $x=r\cos(\text{angle})$ and $y=r\sin(\text{angle})$)
17. Write a program to read a height of person in feet and inches and convert it into meter using user defined to class type conversion method. 1 meter=3.28084 feet, 1 feet=12 inch [PU:2018 fall]
18. Define two classes named 'Polar' and 'Rectangle' to represent points in polar and rectangle systems. Use conversion routines to convert from one system to another system. [PU:2005 fall]
19. Create a class Rectangle with xco and yco as data members and use appropriate function to initialize them and display them. Now create another class polar with radius and angle as data members and member functions to initialize them and display the data. Now use conversion function in source class to convert rectangular object to polar object and vice-versa.
20. Create a abstract class shape with two members base and height, a member function for initialization and a pure virtual function to compute area(). Derive two specific classes ,Triangle and Rectangle which override the function area ().use these classes in main function and display the area of triangle and rectangle. [PU:2009 spring]

CHAPTER 6

Templates and Generic Programming

Templates

- **Template** new concept that enables us to define *generic classes* and *functions* and thus provides support for generic programming.
- **Generic programming** is an approach where **generic types** are used as **parameters** in algorithms so that they work for a variety of suitable data types and data structures.
- A template can be used to create a family of classes or functions .For example, a class template for an array class enables us to create array of various data types such as **int** array or a float array. We can define template for a function, say **mul()**,that would help us create various versions of **mul()** for multiplying int, float and double type values.
- A template *can be* considered as a kind of a macro .When an object of a specific type is defined for actual use the template definition for that class is substituted with the required data type. since, a template is defined with a **parameter** that would be replaced by a specified data type at the time of actual use of the class or function. So a templates are also called a **parameterized classes or functions**.

There are two types of templates:

1. Function Templates
2. Class Templates

CLASS TEMPLATE

We can create class templates for generic class operations. Sometimes, we need a class implementation that is same for all classes, only the data types used are different. Normally, we would need to create a different class for each data type or create different member variables and functions within a single class. This promotes the coding redundancy and will be hard to maintain, as a change is one class/function should be performed on all classes/functions. However, class templates make it easy to reuse the same code for all data types.

The general format of class template is

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

The prefix template <class T> tells the compiler that we are going to declare a template and use T as the type name in the declaration. T may be substituted by any data type including the user defined types.

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

Syntax for creating object

The syntax for creating an object of a template class is

Class_name <datatype> objectname;

Example:

- *For integer data example <int> e1; Here e1 is object of class example*
- *For float data example <float> e2; Here e2 is object of class example*
- *For char data example <char> e3; Here e3 is object of class example*

WAP to add two integer and float using class templates.

```
#include<iostream>
using namespace std;
template<class T>
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 values="<<endl;
s1.add();
cout<<"sum of float values="<<endl;
s2.add();
return 0;
}

```

WAP to perform 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:
sample(T x,T y)
{
a=x;
b=y;
}
void calculate()
{
s=a+b;
p=a*b;
cout<<"sum="<<s<<endl;
cout<<"product="<<p<<endl;
}
};
int main()
{
sample <int> s1(5,8);
sample <float> s2(3.5,8.9);
cout<<"for integer values:"<<endl;
s1.calculate();
cout<<"for float values:"<<endl;
s2.calculate();
return 0;
}

```

WAP to find the maximum number between two numbers using class template.

```
#include<iostream>
using namespace std;
template<class T>
class compare
{
private:
T a,b;
public:
compare(T x,T y)
{
a=x;
b=y;
}
T max()
{
return (a>b)?a:b;
}
};
int main()
{
compare <int> c1(5,6);
compare <float> c2(4.9,32.4);
cout<<"Maximum value among two integer ="<<c1.max()<<endl;
cout<<"Maximum value among two float ="<<c2.max()<<endl;
return 0;
}
```

Create a class template to find the scalar product of vectors of integers and vectors of floating point number.[PU:2009 spring]

```
#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*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 values=<<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 values=<<m*n;
    return 0;
}

```

Class template with multiple parameters

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

template<class T1,class T2>

```

class classname
{
.....
..... (Body of the class)
.....
};

```

Example:

```

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

```

```

void show()
{
cout<<a<<" "<<b<<endl;
}
};

int main()
{
Test<float,int> test1(1.23,123);
Test<int,char> test2(100,'W');
test1.show();
test2.show();
return 0;
}

```

WAP to perform sum of two integer, two float one integer and one float using class template

```

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

```

```

int main()
{
    sample <int,int> s1(5,8);
    sample <float,float> s2(3.5,8.9);
    sample <int,float> s3(3,8.9);
    cout<<"sum of two integer values="<<endl;
    s1.add();
    cout<<"sum of two float values="<<endl;
    s2.add();
    cout<<"sum of one integer and one float values="<<endl;
    s3.add();
    return 0;
}

```

Using default data types in class definition.

```

#include<iostream>
using namespace std;
template <class T1=int,class T2=int>
class Test
{
private:
    T1 a;
    T2 b;
public:
    Test(T1 x,T2 y)
    {
        a=x;
        b=y;
    }
    void show()
    {
        cout<<a<<" "<<b<<endl;
    }
};
int main()
{
    Test <float,int> t1(1.52,8);
    Test <int,char> t2(7,'a');
    Test <> t3(5,10);
    t1.show();
    t2.show();
    t3.show();
    return 0;
}

```

WAP to perform sum and product of one integer and one float using class template

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

WAP to perform sum and product of one integer and one float using class template

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

Define a class called stack and implement generic methods to push and pop elements from stack.[PU:2015 fall]

```
#include <iostream>
using namespace std;
#define max 10
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;
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 << "popped top" << endl;
s.pop();
s.show();
return 0;
}

```

Function definition outside the class template

The member function of the class template is defined outside the class in the following form.

```

template <class template_type>
class class_name
{
private:
template_type variable_name;
//.....
public:
return_type function_name (template_type arg);
//.....
}

```

```

template<class      template_type>
return_type  class_name <template_type>::function_name(template_type  arg)
{
    //body of function template
}

```

Program to find the sum of array elements using class template.

```
#include<iostream>
using namespace std;
const int size=5;
template<class T>
class Array
{
private:
T arr[size];
public:
void get_array();
T find_sum();
};
template <class T>
void Array<T>::get_array()
{
    for(int i=0;i<size;i++)
    {
        cin>>arr[i];
    }
}
template <class T>
T Array<T>::find_sum()
{
    T sum=0;
    for(int i=0;i<size;i++)
    {
        sum=sum+arr[i];
    }
    return sum;
}
int main()
{
Array <int> a1;
cout<<"Enter integer numbers"<<endl;
a1.get_array();
cout<<"sum of integer numbers="<<a1.find_sum();
Array <float> a2;
cout<<"Enter floating numbers"<<endl;
a2.get_array();
cout<<"sum of floating numbers="<<a2.find_sum();
return 0;
}
```

Function template

- Function template can be used to create a family of functions with different argument types.
- A single function template can work with different data types.
- Any type of function argument is accepted by the function.

The general format of function template is:

```
template <class T>
return_type  function_name(arguments of type T)
{
.....
//body of function with type T
//wherever appropriate
.....
}
```

1. Program to display largest among two numbers using function templates.

```
#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;
    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 value"<<endl;
    return 0;
}
```

2. Create a function template to swap two values.[PU:2018 fall]

```
#include <iostream>
using namespace std;
template <class T>
void swapvar(T &x,T &y)
{
    T temp;
    temp=x;
    x=y;
    y=temp;
}
int main()
{
    int i1=10,i2=20;
    float f1=5.5,f2=8.2;
    cout <<"Before swapping" << endl;
    cout <<"i1=" <<i1 <<" " <<"i2=" <<i2 << endl;
    cout <<"f1=" <<f1 <<" " <<"f2=" <<f2 << endl;
    swapvar(i1,i2);
    swapvar(f1,f2);
    cout <<"After swapping" << endl;
    cout <<"i1=" <<i1 <<" " <<"i2=" <<i2 << endl;
    cout <<"f1=" <<f1 <<" " <<"f2=" <<f2 << endl;
    return 0;
}
```

3. Write a function templates to calculate the average and multiplication of numbers.

[PU 2012 spring]

```
#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 << endl;
    cout <<"Product=" <<pro << endl;
}
```

```

int main()
{
    int i1=10,i2=20;
    float f1=9.4,f2=4.3;
    cout<<"Calculation for integer values" << endl;
    calculate(i1,i2);
    cout<<"Calculation for float values" << endl;
    calculate(f1,f2);
    return 0;
}

```

4. Write a function template to calculate the sum and average of numbers.[PU:2009 fall]

```

#include<iostream>
using namespace std;
template<class T>
void calculate(T a,T b)
{
    T sum,avg;
    sum=a+b;
    avg=(a+b)/2;
    cout<<"Average=" << avg << endl;
    cout<<"Sum=" << sum << endl;
}
int main()
{
    int i1=10,i2=20;
    float f1=9.4,f2=4.3;
    cout<<"Calculation for integer values" << endl;
    calculate(i1,i2);
    cout<<"Calculation for float values" << endl;
    calculate(f1,f2);
    return 0;
}

```

5. Create a templates to find the sum of two integers and floats.

[PU:2014 fall] [PU:2016 spring] [PU:2017 spring]

```
#include<iostream>
using namespace std;
template <class T>
void sum(T x,T y)
{
T s;
s=x+y;
cout<<"s="<
```

6. Write a program to find the sum of integer and float array using function templates.

```
#include<iostream>
using namespace std;
template<class T>
T sum(T a[],int size)
{
    T s=0;
    for(int i=0;i<size;i++)
    {
        s=s+a[i];
    }
    return s;
}
```

```

int main()
{
    int x[5]={10,20,30,40,50};
    float y[3]={1.1,2.2,3.3};
    cout<<"Integer array element sum="<<sum(x,5)<<endl;
    cout<<"Float array element sum="<<sum(y,3)<<endl;
    return 0;
}

```

Note:

A function generated from a function template is called template function. Program Demonstrates the use of template functions in nested form for implementing bubble sort algorithm.

Bubble sort using Template functions (Using of template functions using nested form)

```

#include<iostream>
using namespace std;
template<class T>
void bubble(T arr[],int n)
{
    for(int i=0;i<n-1;i++)
    {
        for(int j=0;j<n-i-1;j++)
        {
            if(arr[j]>arr[j+1])
            {
                swap(arr[j],arr[j+1]);
            }
        }
    }
}

template <class X>
void swap(X &a,X &b)
{
    X temp;
    temp=a;
    a=b;
    b=temp;
}

```

```

int main()
{
    int i,j;
    int x[5]={50,10,30,20,40};
    float y[5]={1.1,5.5,3.3,4.4,2.2};
    bubble(x,5);
    bubble(y,5);
    cout<<"Sorted x-array"<<endl;
    for(i=0;i<5;i++)
    {
        cout<<x[i]<<endl;
    }
    cout<<"Sorted y-array "<<endl;
    for(i=0;i<5;i++)
    {
        cout<<y[i]<<endl;
    }
    return 0;
}

```

Write a program to find the minimum elements in an array using function template.

```

#include<iostream>
using namespace std;
template<class T>
T find_min(T arr[],int n)
{
    T min=arr[0];
    for(int i=0;i<n;i++)
    {

        if(arr[i]<min)
        {
            min=arr[i];
        }
    }
    return min;
}

```

```

int main()
{
int i,j,imin;
float fmin;;
int x[6]={50,10,30,20,40,70};
float y[5]={1.1,5.5,3.3,4.4,2.2};
imin=find_min(x,6);
fmin=find_min(y,5);
cout<<"Minimum value in integer array=<<imin<<endl;
cout<<"Minimum value in float array=<<fmin<<endl;
return 0;
}

```

Write a program to find the maximum elements in an array using function template.

```

#include<iostream>
using namespace std;
template<class T>
T find_max(T arr[],int n)
{
    T max=arr[0];
    for(int i=0;i<n;i++)
    {
        if(arr[i]>max)
        {
            max=arr[i];
        }
    }
    return max;
}

int main()
{
int imax;
float fmax;;
int x[6]={50,10,30,20,40,70};
float y[5]={1.1,5.5,3.3,4.4,2.2};
imax=find_max(x,6);
fmax=find_max(y,5);
cout<<"Maximum value in integer array=<<imax<<endl;
cout<<"Maximum value in float array=<<fmax<<endl;
return 0;
}

```

Function templates with multiple parameters

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

```
Template<classs T1,class T2,.....>
return_type function_name(arguments of types T1,T2. . . . .)
{
.....
..... (Body of function)
.....
}
```

Program to illustrate the concept of two generic data types.

```
#include<iostream>
using namespace std;
template <class T1,class T2>
void display(T1 x,T2 y)
{
    cout<<x<<" "<<y<<endl;
}
int main()
{
    cout<<"calling function template with integer and string type parameters"<<endl;
    display(199,"pokhara");
    cout<<"calling function template with float and integer type parameters"<<endl;
    display(12.34,5);
    return 0;
}
```

Write a program to add two integers; two floats and one integer and one float numbers respectively. Display the final result in the float.[PU:2005 fall]

```
#include<iostream>
using namespace std;
template <class T1,class T2>
float sum(T1 x,T2 y)
{
    return(x+y);
}
```

```

int main()
{
int i1,i2,i3;
float f1,f2,f3;
float s1,s2,s3;
cout<<"Enter two integer values" << endl;
cin>>i1>>i2;
s1=sum(i1,i2);
cout<<"Enter two float values" << endl;
cin>>f1>>f2;
s2=sum(f1,f2);
cout<<"Enter Integer and Float" << endl;
cin>>i3>>f3;
s3=sum(i3,f3);
cout<<"Sum of two integers = "<<s1<< endl;
cout<<"Sum of two float = "<<s2<< endl;
cout<<"Sum of one integer and one float= "<<s3<< endl;
}

```

Overloading of template functions

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

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

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions. Program below shows how a template function is overloaded with explicit function.

Template function with Explicit Function

```
#include<iostream>
#include<string.h>
using namespace std;
template <class T>
void display(T x)
{
cout<<"Overloaded Template Display1:"<<x<<endl;
}

template <class T1,class T2>
void display(T1 x,T2 y)
{
cout<<"Overloaded Template Display2:"<<x<<" "<<y<<endl;
}
void display(int x)
{
cout<<"Explicit display:"<<x<<endl;
}

int main()
{
display(100);
display(12.34);
display(100,15.4);
display('C');
return 0;
}
```

Output:

```
Explicit display:100
Overloaded Template Display1:12.34
Overloaded Template Display2:100,12.34
Overloaded Template Display1:C
```

Merit and demerit of using template in C++

Merit

- C++ templates enable us to define a family of functions or classes that can operate on different types of information.
- We can use templates in situations that result in duplication of the same code for multiple types. For example, we can use function templates to create a set of functions that apply the same algorithm to different data types.
- Deliver fast, efficient, and robust code
- Easy to use
- When we use templates in combination with STL – it can drastically reduce development time.

Demerit

- Historically, some compilers exhibited poor support for templates. So, the use of templates could decrease code portability.
- Automatically generated source code can become overwhelmingly huge.
- Compile-time processing of templates can be extremely time consuming.
- It can be difficult to debug code that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime.
- Templates are in the headers, which require a complete rebuild of all project pieces when changes are made.
- Many compilers lack clear instructions when they detect a template definition error. This can increase the effort of developing templates, and has prompted the development of Concepts for possible inclusion in a future C++ standard.
- No information hiding. All code is exposed in the header file. No one library can solely contain the code .
- Though STL itself is a collection of template classes, templates are not used to write conventional libraries.

Exception handling

- Exceptions are runtime anomalies or unusual condition that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory space or disk space.
- The exception handling is a mechanism to detect and report an “exceptional circumstances “at runtime, so that appropriate action can be taken. It provides a type-safe, integrated approach, for coping with the unusual predictable problems that arise while executing a program.

Types of Exceptions

- Exceptions are basically of two types namely, *synchronous* and *asynchronous* exceptions.
- Errors such as “*out of range index*” and “*overflow*” belongs to synchronous type exceptions.
- The errors that are caused by the events beyond the control of program (such as keyboard interrupts) are called *asynchronous* exceptions.
- The exception handling mechanism in C++ can handle only synchronous exceptions.

The exception handling mechanism suggests a separate error handling code that performs the following tasks.

1. Find the problem (Hit the exception)
2. Inform the error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take the corrective action (Handle the exception)

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

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

1. The keyword **try** is used to **preface** a block of statements (surrounded by 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 defined by the keyword **catch**, catches the exception thrown by the throw statement in the try block and handles it appropriately.

```

try
{
.....
//block of statements which detects and throw an exceptions

throw exception;
}
catch(type arg ) //catch the exception
{
// Block of statements that handles the exceptions
}

```

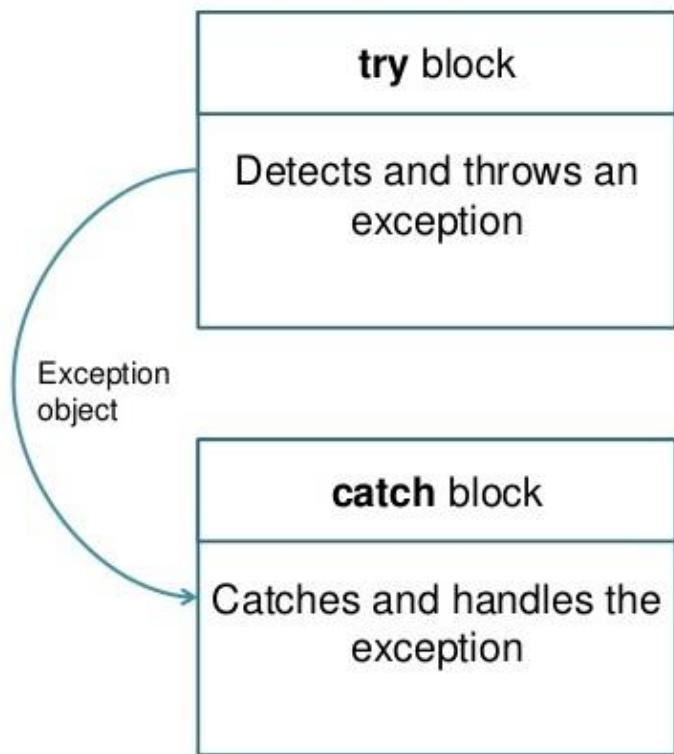


Figure : The block throwing exception

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 arg type in the catch statement, then the catch block is executed for handling the exception. If they do not 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 blocks That is ,catch block is skipped.

Program:

```
#include<iostream>
using namespace std;
int main()
{
int a,b,x;
cout<<"Enter values 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); //throws an object
}
}
catch(int i) //catches an exception
{
cout << "Exception caught:DIVIDE BY ZERO" << endl;
}
cout << "END";
return 0;
}
```

Output:

First Run:

```
Enter values of a and b
20 15
Result(a/x)=4
END
```

Second Run:

```
Enter values of a and b
10 10
Exception caught: DIVIDE BY ZERO
END
```

Multiple catch statements

It is possible that a program segment 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 conditions in **switch** statement) as shown below.

```
try
{
//try block
}
catch(type1 arg)
{
//Catch block1
}

catch(type2 arg)
{
//catch block2
}
.....
catch(typeN arg)
{
Catch block N
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match 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.

It is possible that arguments of several catch statements match the type of an exception. In such cases the first handler that matches the exception type is executed.

Example of program where multiple catch statements are used to handle various types of exceptions.

Write a program that catches multiple exceptions.[PU:2016 spring]

```

#include<iostream>
using namespace std;
void test(int x)
{
    try
    {
        if(x==1)
            throw x;
        else if(x==0)
            throw 'x';
        else if (x==-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<<"x==1"<<endl;
    test(1);
    cout<<"x==0"<<endl;
    test(0);
    cout<<"x== -1"<<endl;
    test(-1);
    cout<<"x==2"<<endl;
    test(2);
    return 0;
}

```

Output:

```
Testing multiple catches
```

```
x==1
```

```
Caught an integer
```

```
End of try-catch system
```

```
x==0
```

```
Caught a character
```

```
End of try-catch system
```

```
x==-1
```

```
Caught a double
```

```
End of try-catch system
```

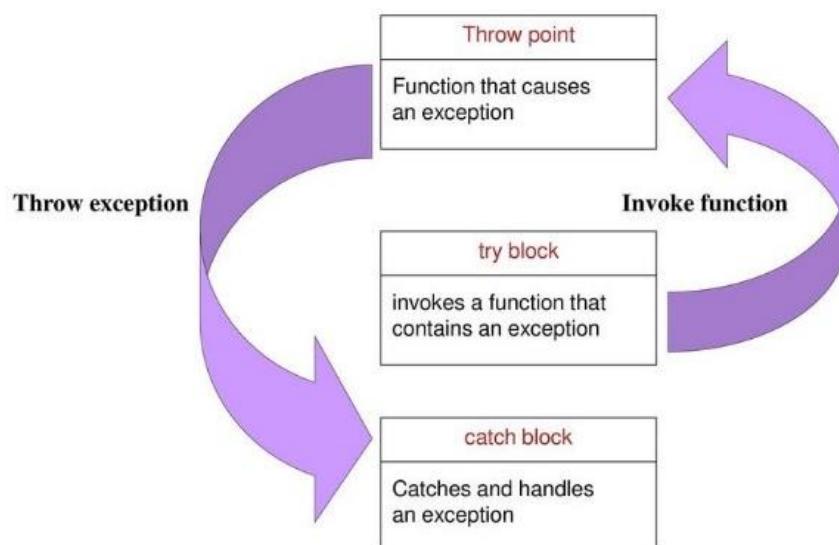
```
x==2
```

```
End of try-block
```

```
End of try-catch system
```

Functions throwing an exception

Most often, exceptions are thrown by functions that are invoked from within **try** blocks. The point at which the throw is executed is called throw point. Once an exception is thrown to the catch block, control cannot return to the throw point. This kind of relationship is as shown in figure.



The general format of code for this kind of relationship is as shown in figure.

```
type function(arg list)
{
.....
.....
throw(object);
.....
.....
}
.....
.....
try
{
.....
..... Invoke function here
.....
}
catch(type arg)
{
.....
..... Handles exception here
.....
}
....,
```

Note:

Catch block must be immediately after try block without any code between them.

Invoking function that generates exception

```
#include<iostream>
using namespace std;
void divide(int x,int y,int z)
{
    cout<<"We are inside the function"<<endl;
    if((x-y)!=0)
    {
        int r=z/(x-y);
        cout<<"Result="<<r<<endl;
    }
    else
    {
        throw(x-y);// throw point;
    }
}
int main()
{
    try
    {
        cout<<"We are inside the try block"<<endl;
        divide(10,20,30);
        divide(10,10,20);
    }
    catch(int i)
    {
        cout<<"caught an exception"<<endl;
    }
    return 0;
}
```

Output:

```
We are inside the try block
We are inside the function
Result=-3
We are inside the function
Caught an exception
```

Rethrowing an exception

A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any arguments as shown below.

throw;

This causes the current exception to be thrown to the next enclosing **try/catch** sequence and is caught by a catch statement listed after that enclosing try block. Program demonstrates how an exception is rethrown and caught.

```
#include<iostream>
using namespace std;
void divide(double x,double y)
{
    cout<<"Inside function"<<endl;
    try
    {
        if(y==0)
            throw y;
        else
            cout<<"Division="<<x/y<<endl;
    }
    catch(double)
    {
        cout<<"caught double inside function"<<endl;
        throw;
    }
    cout<<"End of function"<<endl;
}

int main()
{
    cout<<"Inside main"  <<endl;
    try
    {
        divide(10.5,2.0);
        divide(20.0,0.0);
    }
    catch(double)
    {
        cout<<"caught double inside main"<<endl;
    }
    cout<<"End of main"<<endl;
    return 0;
}
```

Output:

```
Inside main
Inside function
Division=5.25
End of function

Inside function
Caught double inside function
Caught double inside main
End of main
```

When an exception is rethrown, it will not be caught by the same **catch** statement or any other catch in that group. Rather, it will be caught by an appropriate **catch** in the outer **try/catch** sequence only.

A catch handler itself may detect and throw an exception. Here again, the exception thrown will not be caught by any catch statements in that group. It will be passed on to the next outer try/catch sequence for processing.

Standard Template Library

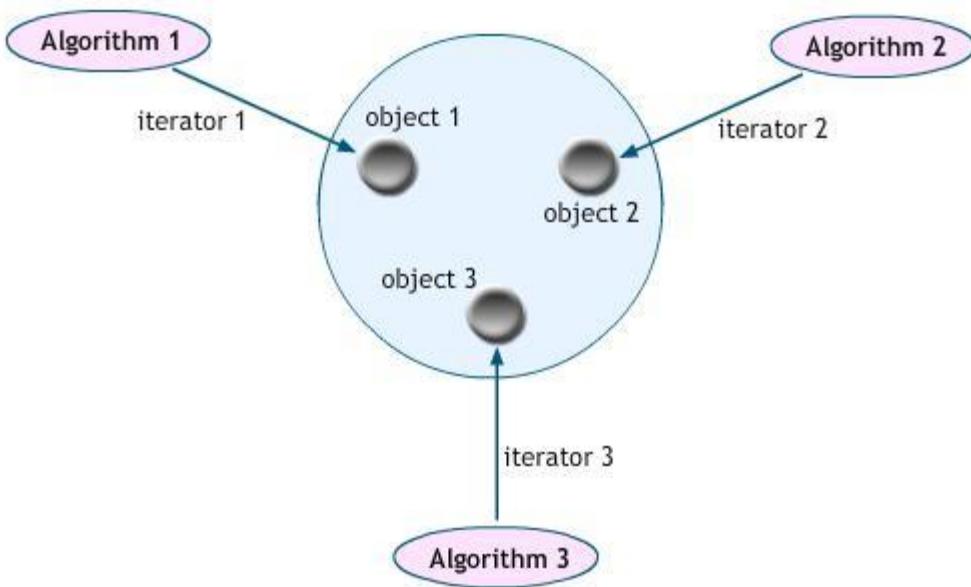
- In order to help the C++ users in generic programming, Alexander Stepanov and Meng Lee of Hewlett-Packard developed a set of general purpose templated class (data structure) and functions (algorithms) that could be used as a standard approach of storing and processing of data. The collection of these generic classes and functions is called the Standard Template Library (STL).
- It helps to save C++ users' time and effort there by helping to produce high quality programs.

Components of STL:

The STL contains several components .But at its core are three key components. They are

1. Containers
2. Algorithms
3. Iterators

These three components work in conjunction with one another to provide support to a variety of programming solutions. The relationship between the three components is as shown in figure below. Algorithms employ iterators to perform operation stored in containers.



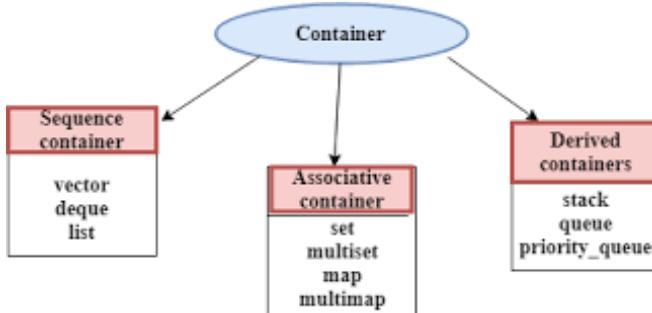
- A **container** is an object that actually stores data. It is a way in which data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data.
- An **algorithm** is a procedure used to process the data contained in the containers. The STL includes many different kinds of algorithms to provide support to tasks such as initializing, searching, coping, sorting and merging. Algorithms are implemented by templates functions.
- An **iterator** is an object (like pointer) that points to an element in a container. We can use iterators to move through the contents of containers. It is handled just like pointer and can be incremented and decremented. Iterator connect algorithm with containers and play a key role in the manipulation of data stored in the containers.

Features of STL

It helps in saving time, efforts, load fast, high quality programming because STL provides well written and tested components, which can be reuse in our program to make our program more robust.

Containers

As we know containers is an object that actually stores data. The STL defines ten containers which are grouped in three categories as shown in fig.



1. Sequence containers

- Sequence containers stores elements in liner sequence.
- Elements of these containers can be accessed using an iterator.

The STL provides three types of sequence containers.

- Vector
- List
- Deque

2. Associative containers

Associative containers are designed to support direct access to element using keys. They are not sequential. There are four types of associative containers.

- Set
- Multiset
- Map
- Multimap

All these containers store data in a structure called tree which facilitates fast searching,deletion and insertion. However these are very slow for random access and inefficient for sorting.

3. Derived containers

The STL provides three derived containers namely stack, queue, and priority queue. These are also called containers adaptors.

Stack Queues and priority queues can be created from different sequence containers. The derived containers do not support iterators and therefore we cannot use them for data manipulation. However, they support two member functions **pop()** and **push()** for implementing deleting and inserting operations.

Containers supported by STL

Container	Description	Header file	Iterator
Vector	A dynamic array. Allows insertions and deletions at back. Permits direct access to any element.	<vector>	Random access
List	A bidirectional, linear list. Allows insertions and deletions anywhere.	<list>	Bidirectional
Deque	A double-ended queue. Allows insertions and deletions at both ends. Permit direct access to any element.	<deque>	Random access
Set	An associate container for storing unique sets. Allows rapid lookup.(No duplicates are allowed)	<set>	Bidirectional
multiset	An associate container for storing non- unique sets. (Duplicates allowed)	<set>	Bidirectional
Map	An associate container for storing unique key/value pairs. Each key is associated with only one value(one-to-one mapping).Allows key-based lookup.	<map>	Bidirectional
multimap	An associate container for storing key/value pairs in which one key may be associated with more than one value.(one-to-many mapping) .Allows key-based lookup.	<map>	Bidirectional
Stack	An standard stack. Last-in-first-out(LIFO)	<stack>	No iterator
Queue	A standard queue. First-in-first-out(FIFO)	<queue>	No iterator
Priority-queue	A priority queue. The first element out is always the highest priority element.	<queue>	No iterator

Previous old Questions from this chapter

- 1) What is generic programming? How would you convert rectangular classes into templates.[PU:2006 spring]
- 2) What do you mean by generic programming? Illustrate with the example of function template.[PU:2015 spring]
- 3) What are the advantages of Generic programming? Explain with suitable example.
- 4) What is function template?
- 5) What is template? List merit and demerit of using template in C++.
- 6) What is generic and templates. [PU:2016 spring]
- 7) What is template ?List the merit and demerit of using a template in C++. [PU:2013 fall]

- 8) What is template ?Explain different types of template used in C++. [PU:2014 spring]
- 9) What are the advantages of using template functions. Write a program to illustrate a template function with two arguments.[PU:2017 fall]
- 10) With an example explain the concept of generic programming.
- 11) What is exception handling? Discuss briefly.[PU:2005 fall]
- 12) What is exception? Explain the method of exception handling in C++?
- 13) What is exception? Define the type of exceptions. Explain about Exception handling mechanism in C++. [PU:2013 spring]
- 14) What is exception? What is the syntax for exception handling in C++.Write a program that catches multiple exceptions.[PU:2016 spring]
- 15) Write a short notes on:
 - Templates[PU:2010 fall]
 - Container classes[PU:2005 fall]
 - Exception handling[PU:2009 spring][PU:2014 spring][PU:2014 fall] [2018 fall]
 - Exception mechanism[PU:2017 spring]
 - Standard Template Library(STL)

Programs

1. Write a program using template to add two integers, two floats and one integer and one float numbers respectively. Display the final result in float.[PU:2005 fall]
2. Write a function template to calculate the sum and average of numbers.[PU:2009 fall]
3. Create a template function to swap two values.[PU:2018 fall]
4. Write a function template to calculate the average and multiplication of numbers.
5. Create a templates to find the sum of two integers and floats.[PU:2014 fall] .[PU:2016 spring] .[PU:2017 spring]
6. Create a template class stack to show push and pop operation on stack.[PU:2010 spring]
7. Define a class called stack and implement generic methods to push and pop the elements from the stack.[PU:2015 fall]
8. Define class called stack and implement generic methods to push and pop the elements from stack.[PU:2015 fall]
9. Write a program to illustrate the template class with two generic types.
10. Write a program to illustrate the overloading of template functions.
11. Define two classes named ‘Polar’ and ‘rectangle’ to represent points in polar and rectangle systems. Use conversion routines to convert from one system to another system using template.[PU:2013 fall]
12. How can we compute the roots of quadratic equations by using function template? Explain with examples.

CHAPTER 7

Object Oriented Design

7.1 Responsibility implies non-interface

- When we make an object (be it a child or a software system) responsible for specific actions, we expect a certain behavior, at least when the rules are observed.
- Responsibility implies a degree of independence or noninterference.
- If we tell a child that she is responsible for cleaning her room, we do not normally stand over her and watch while that task is being performed—that is not the nature of responsibility. Instead, we expect that, having issued a directive in the correct fashion, the desired outcome will be produced.
- In case of conventional programming we tend to actively supervise the child while she's performing a task.
- In case of OOP we tend to handover to the child responsibility for that performance
- Conventional programming proceeds largely by doing something to something else—modifying a record or updating an array. Thus, one portion of code in a software system is often intimately tied by control and data connections to many other sections of the system. Such dependencies can come about through the use of global variables, through use of pointer values or simply through inappropriate use of and dependence on implementation details of other portions of code.
- A responsibility-driven design attempts to cut these links, or at least make them as unobtrusive as possible.

One of the major benefits of object-oriented programming occurs when software subsystems are reused from one project to the next. For example, a simulation manager might work for both a simulation of balls on a billiards table and a simulation of fish in a fish tank. This ability to reuse code implies that the software can have almost no domain-specific components; it must totally delegate responsibility for domain-specific behavior to application-specific portions of the system. The ability to create such reusable code is not one that is easily learned—it requires experience, careful examination of case studies (paradigms, in the original sense of the word), and use of a programming language in which such delegation is natural and easy to express.

7.2 Programming in small and Programming in large

The difference between the development of individual projects and of more sizable software systems is often described as programming in the small versus programming in the large.

Programming in the small characterizes projects with the following attributes:

- Code is developed by a single programmer, or perhaps by a very small collection of programmers. A single individual can understand all aspects of a project, from top to bottom, beginning to end.
- The major problem in the software development process is the design and development of algorithms for dealing with the problem at hand.

Programming in the large, on the other hand, characterizes software projects with features such as the following:

- The software system is developed by a large team, often consisting of people with many different skills. There may be graphic artists, design experts, as well as programmers. Individuals involved in the specification or design of the system may differ from those involved in the coding of individual components, who may differ as well from those involved in the integration of various components in the final product. No single individual can be considered responsible for the entire project, or even necessarily understands all aspects of the project.
- The major problem in the software development process is the management of details and the communication of information between diverse portions of the project.

7.3 Role of Behavior in OOP

Earlier software development methodologies (those popular before the advent of object-oriented techniques) concentrated on ideas such as characterizing the basic data structures or the overall structure of function calls, often within the creation of a formal specification of the desired application. But structural elements of the application can be identified only after a considerable amount of problem analysis.

Similarly, a formal specification often ended up as a document understood by neither programmer nor client. But behavior is something that can be described almost from the moment an idea is conceived, and (often unlike a formal specification) can be described in terms meaningful to both the programmers and the client.

7.4 Responsibility-Driven Design

Responsibility-Driven Design (RDD), developed by Rebecca Wirfs-Brock, is an object-oriented design technique that is driven by an emphasis on behavior at all levels of development.

It focuses on:

- What action is object responsible for?
- What information does the object share?

- RDD focuses on what action must get accomplished and which object will accomplish them.
- The RDD approach focuses on modeling object behavior and identifying patterns of communication between objects.
- One objective of object oriented design is first to establish who is responsible for each action to be performed.
- The design process consists of finding key objects during their role and responsibilities and understanding their pattern of communication.
- RDD initially focuses on what should be accomplished not how. RDD tries to avoid dealing with details.
- If any particular action is to happen, someone must be responsible for doing it. No action takes place without an agent.
- One of the technique is use of index cards to represent individual class. such cards are known as index cards.

7.4 Case study in RDD

7.4.1 The interactive Intelligent Kitchen Helper

Brief Description

The Interactive Intelligent Kitchen Helper (IIKH) is a PC-based application that will replace the index-card system of recipes found in the average kitchen. But more than simply maintaining a database of recipes, the kitchen helper assists in the planning of meals for an extended period, say a week. The user of the IIKH can sit down at a terminal, browse the database of recipes, and interactively create a series of menus. The IIKH will automatically scale the recipes to any number of servings and will print out menus for the entire week, for a particular day, or for a particular meal. And it will print an integrated grocery list of all the items needed for the recipes for the entire period.

7.4.2 Working through scenarios

Initial specifications are almost always ambiguous and unclear on anything except the most general points. So, the first task is to refine the specification.

The specifications for the final application will may change during the creation of the software system, so it is important that, the design be developed to easily accommodate change and that potential changes be noted as early as possible.

Equally important, at this point very high level decisions can be made concerning the structure of the eventual software system. In particular, the activities to be performed can be mapped onto components.

In order to uncover the fundamental behavior of the system, the design team first creates a number of scenarios. That is, the team acts out the running of the application just as if it already possessed a working system. An example scenario is as below.

7.4.3 Identification of components

- The engineering of software is simplified by the identification and development of software components.
- A component is simply an abstract entity that can perform tasks—that is, fulfill some responsibilities.
- At this point, it is not necessary to know exactly the eventual representation for a component or how a component will perform a task. A component may ultimately be turned into a function, a structure or class, or a collection of other components.
- At this level of development there are just two important characteristics:
 - A component must have a small well-defined set of responsibilities.
 - A component should interact with other components to the minimal extent possible.

7.5 CRC Cards (Class Responsibility Collaborators)

CRC stands for **Class Responsibility Collaborators**.

CRC cards are a brainstorming tool used in the design of object-oriented software.

It was first introduced by Kent Beck and Ward Cunningham.

The cards are arranged to show the flow of messages among instances of each class.

CRC Card is divided into three sections as shown in figure.

Class Name	
Responsibilities	Collaborators

A class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities.

Let us illustrate these concept with an example of **Student CRC card**

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

Seminar	
Name Seminar number Fees Add Student Drop student Instructor	Student Professor

Professor	
Name Address Email address Salary Provides information Instructing seminar	Seminar

Transcript	
Student name Marks obtained Enrolled year Calculate final grade	Student

Fig: CRC cards for class student

Things to be understand:

Here, *Student* is used as class name across the top of a CRC card .We use name of a class as a singular noun because each class represents a generalized version of a singular object.

As we know responsibility is anything that a class knows or does. In this example, students have names, addresses, and phone numbers. These are the things a student knows. Students also enroll in seminars and request transcripts. These are the things a student does.

Sometimes a class has a responsibility to fulfill, but not have enough information to do it. In above example, as you see students enroll in seminars. To do this, a student needs to know if a spot is available in the seminar and, if so, he then needs to be added to the seminar. However, students only have information about themselves (their names and so forth), and not about seminars. What the student needs to do is collaborate/interact with the card labeled Seminar to sign up for a seminar. Therefore, Seminar is included in the list of collaborators of Student. In this way we can draw CRC cards.

7.5.1 Give components a physical Representation

While working through scenarios, it is useful to assign CRC cards to different members of the design team. The member holding the card representing a component records the responsibilities of the associated software component, and acts as the "surrogate" for the software during the scenario simulation. He or she describes the activities of the software system, passing "control" to another member when the software system requires the services of another component.

The physical separation of the cards encourages an intuitive understanding of the importance of the logical separation of the various components, helping to emphasize the cohesion and coupling (which we will describe shortly).

7.5.2 What/Who Cycle

The identification of components takes place during the process of imagining the execution of a working system.

Often this proceeds as a cycle of what/who questions.

- First, the design team identifies what activity needs to be performed next.
- This is immediately followed by answering the question of who performs the action.

In this manner, designing a software system is much like organizing a collection of people, such as a club. Any activity that is to be performed must be assigned as a responsibility to some component.

The secret to good object-oriented design is to first establish an agent for each action.

7.5.3 Documentation

Two documents should be essential parts of any software system: the user manual and the system design documentation. Work on both of these can commence even before the first line of code has been written.

The User Manual

The user manual describes the interaction with the system from the user's point of view.

Before any actual code has been written, the mindset of the software team is most similar to that of the eventual users.

The system Design Documentation

- The design documentation records the major decisions made during software design, and should thus be produced when these decisions are fresh in the minds of the creators.
- Gives the initial picture of the larger structure hence should be carried out early in the development cycle.
- CRC cards are one aspect of the design documentation, but many other important decisions are not reflected in them. Arguments for and against any major design alternatives should be recorded, as well as factors that influenced the final decisions.

7.6 Interaction Diagram /Sequence diagram

- Used for describing their dynamic interactions during the execution of a scenario.
- In the diagram, time moves forward from the top to the bottom.
- Each component is represented by a labeled vertical line.
- A component sending a message to another component is represented by a horizontal arrow from one line to another.
- Similarly, a component returning control and perhaps a result value back to the caller is represented by an arrow.
- The commentary on the right side of the figure explains more fully the interaction taking place.
- With a time axis, the interaction diagram is able to describe better the sequencing of events during a scenario. For this reason, interaction diagrams can be a useful documentation tool for complex software systems.

Figure shows the beginning of an interaction diagram for the interactive kitchen helper.

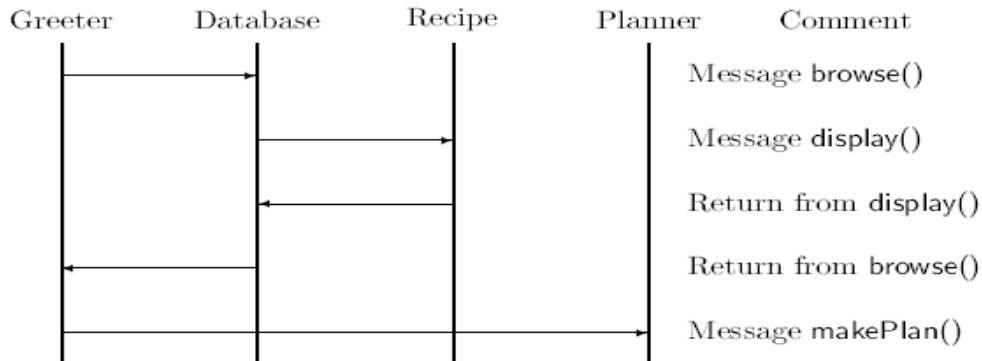


Figure 0-1: An Example interaction diagram.

Note:

Some authors use two different arrow forms, such as a solid line to represent message passing and a dashed line to represent returning control.

Sequence diagram is a sub-category of Interaction diagram

7.7 Software components

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

In object oriented programming, a component is a reusable program building block that can be combined with other components to form an application. Examples of components include: a single button in graphical interface, a simple interest calculator, an interface to a database manager.

Each component is characterized by:

1) Behavior and state

- The behavior of a component is the set of actions it can perform. The complete description of all the behavior for a component is sometimes called the protocol. For the Recipe component this includes activities such as editing the preparation instructions, displaying the recipe on a terminal screen, or printing a copy of the recipe.
- The state of a component represents all the information held within it at a given point of time. For our Recipe component the state includes the ingredients and preparation instructions. Notice that the state is not static and can change over time. For example, by editing a recipe (a behavior) the user can make changes to the preparation instructions (part of the state).

2) Instances and classes

- In the real application there will probably be many different recipes. However, all of these recipes will perform in the same manner i.e. the behavior of each recipe is the same.
- But the state (individual list of ingredients and instructions for preparation) differs between individual recipes.
- The term class is used to describe a set of objects with similar behavior.
- An individual representative of a class is known as an instance.
- Note that behavior is associated with a class, not with an individual. That is, all instances of a class will respond to the same instructions and perform in a similar manner.
- On the other hand, state is a property of an individual. We see this in the various instances of the class Recipe. We see this in the various instances of the class Recipe .They can all perform the same actions (editing, displaying, printing) but use different data values.

3) Coupling and cohesion

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

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

4) Interface and Implementation-Parnas's Principles

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

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

These ideas were captured by computer scientist David Parna's in a pair of rules, known as **Parnas's principles**:

- The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide no other information.
- The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with no other information.

7.8 Formalizing the Interface

- The first step in this process is to formalize the patterns and channels of communication.
- A decision should be made as to the general structure that will be used to implement each component. A component with only one behavior and no internal state may be made into a function. Components with many tasks are probably more easily implemented as classes. Names are given to each of the responsibilities identified on the CRC card for each component, and these will eventually be mapped onto method names. Along with the names, the types of any arguments to be passed to the function are identified.
- Next, the information maintained within the component itself should be described. All information must be accounted for. If a component requires some data to perform a specific task, the source of the data, either through argument or global value, or maintained internally by the component, must be clearly identified.

7.8.1 Coming up with names

The selection of useful names is extremely important, as names create the vocabulary with which the eventual design will be formulated. Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem

General Guidelines for choosing names:

- Use pronounceable names. As a rule of thumb, if you cannot read a name out loud, it is not a good one.
- Use capitalization (or underscores) to mark the beginning of a new word within a name, such as "CardReader" or "Card_reader," rather than the less readable "cardreader."
- Abbreviations should not be confusing. Is a "TermProcess" a terminal process, something that terminates processes, or a process associated with a terminal?
- Avoid names with several interpretations. Does the empty function tell whether something is empty, or empty the values from the object?
- Avoid digits within a name. They are easy to misread as letters (0 as O, 1 as I, 2 as Z, 5 as S).
- Name functions and variables that yield Boolean values so they describe clearly the interpretation of a true or false value. For example, "Printer-IsReady" clearly indicates that a true value means the printer is working, whereas "PrinterStatus" is much less precise
- Names for operations that are costly and infrequently used should be carefully chosen as this can avoid errors caused by using the wrong function.

Once names have been developed for each activity, the CRC cards for each component are redrawn, with the name and formal arguments of the function used to elicit each behavior identified. An example of a CRC card for the Date is shown in Figure .

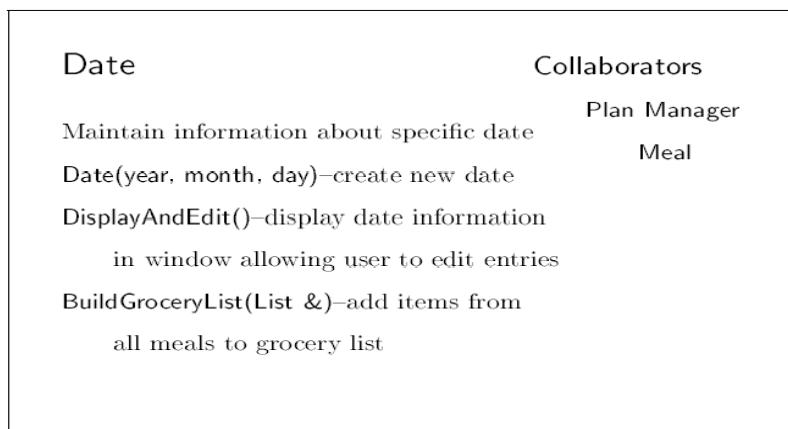


Figure: Revised CRC card for the Date component.

7.9 Designing the Representation

At this point, the design team can be divided into groups, each responsible for one or more software components. The task now is to transform the description of a component into a software system implementation. The major portion of this process is designing the data structures that will be used by each subsystem to maintain the state information required to fulfill the assigned responsibilities.

Once data structures have been chosen, the code used by a component in the fulfillment of a responsibility is often almost self-evident. A wrong choice can result in complex and inefficient programs.

It is also at this point that descriptions of behavior must be transformed into algorithms. These descriptions should then be matched against the expectations of each component listed as a collaborator, to ensure that expectations are fulfilled and necessary data items are available to carry out each process.

7.10 Implementing components

- If the previous steps were correctly addressed, each responsibility or behavior will be characterized by a short description. The task at this step is to implement the desired activities in a computer language.
- As one programmer will not work on all aspects of a system, programmer will need to master are understanding how one section of code fits into a larger framework and working well with other members of a team.
- There might be components that work in back ground (facilitators) that needs to be taken into account.
- An important part of analysis and coding at this point is:
 - i. Characterizing and documenting the necessary preconditions a software component requires to complete a task.
 - ii. Verifying that the software component will perform correctly when presented with legal input values.

7.11 Integration of components

- Once software subsystems have been individually designed and tested, they can be integrated into the final product. This is often not a single step, but part of a larger process. Starting from a simple base, elements are slowly added to the system and tested, using stubs-simple dummy routines with no behavior or with very limited behavior-for the as yet unimplemented parts.
- Testing of an individual component is often referred to as **unit testing**.
- Next, one or the other of the stubs can be replaced by more complete code. Further testing can be performed until it appears that the system is working as desired. (This is sometimes referred to as **integration testing**. The application is finally complete when all stubs have been replaced with working components.)
- Testing during integration can involve the discovery of errors, which then results in changes to some of the components. Following these changes the components should be once again tested in isolation before an attempt to reintegrate the software, once more, into the larger system.
- Re-executing previously developed test cases following a change to a software component is sometimes referred to as **regression testing**.

7.12 Maintenance and Evolution

The term software maintenance describes activities subsequent to the delivery of the initial working version of a software system. A wide variety of activities fall into this category.

- Errors, or bugs, can be discovered in the delivered product. These must be corrected, either in updates or corrections to existing releases or in subsequent releases.
- Requirements may change, perhaps as a result of government regulations or standardization among similar products.
- Hardware may change. For example, the system may be moved to different platforms, or input devices, such as a pen-based system or a pressure-sensitive touch screen, may become available. Output technology may change-for example, from a text-based system to a graphical window-based arrangement.
- User expectations may change. Users may expect greater functionality, lower cost, and easier use. This can occur as a result of competition with similar products. Better documentation may be requested by users.

A good design recognizes the inevitability of changes and plans an accommodation for them from the very beginning.

Previous Old Questions from this chapter

- 1) Explain Responsibility implies non-interface. Explain with example.
 - 2) Explain the terms:
 - Responsibility implies non-interface
 - Programming in small and Programming in large[PU:2014 fall]
 - 3) What are the difference between programming in small and programming in large?[PU:2010 spring]
 - 4) What is Role behavior of OOP? Along with figure and an example of CRC card .Explain its significance in object oriented Design.[PU:2006 spring][PU:2015 fall]
 - 5) Write a short notes on:
 - CRC cards [PU:2005 fall] [PU:2010 fall][PU:2013 fall][PU:2016 fall]
 - Interface and Implementation [PU:2009 fall]
 - Programming in small and large [PU:2013 spring]
 - Responsibility Driven Design (RDD)[PU:2015 fall][PU:2014 spring][PU:2016 spring]
 - Cohesion and coupling
 - 6) Differentiate between **[PU:2010 fall]**
- OR**
- Explain and contrast the following. **[PU:2015 spring]**
- Programming in small and Programming in large
 - Interface and implementation
 - 7) What do you mean by responsibility driven design? Also explain what is meant by CRC card.[PU:2010 spring]
 - 8) What do you mean by software component? Explain Integration of components with suitable example scenario to support your answer.[PU:2010 spring]
 - 9) What are the advantages of adopting RDD? Explain with the example of suitable example.[PU:2009 spring]
 - 10) Do you find any advantages of adopting Responsibility Driven Design? Explain with help of suitable example.
 - 11) Explain in brief about interface and implementation. How different components of designed software can be represented and integrated? Discuss in brief.[PU:2013 fall][PU:2017 fall]
 - 12) How are object oriented Programs are designed and developed according to the concept of RDD? Describe entire process in brief.[PU:2013 spring]
 - 13) What do you mean by RDD? What is the use of CRC card?[PU:2014 fall]
 - 14) What is software component? Explain the integration of component with real world example. [PU:2016 fall]
 - 15) What are the different aspects of software components?[PU:2016 spring]
 - 16) Explain in brief about interface and implementation.
 - 17) Draw CRC cards of students.[PU:2017 spring]
 - 18) Explain CRC card and sequence diagram with suitable example.[PU:2019 fall]
 - 19) Differentiate between the concept of computation as simulation and Responsibility implies non-interface. [PU:2017 spring]

Path follower Robot senses the path it needs to follow through its sensors. Based on the data received through its sensors, the robot make use of its actuators (Robotic Wheels) to steer itself forward. For the above mentioned systems, identify as many components (Collaborating objects) as you can draw CRC card for least three of them .Show the interaction between these components through interaction diagram.[PU:2015 fall 6b]

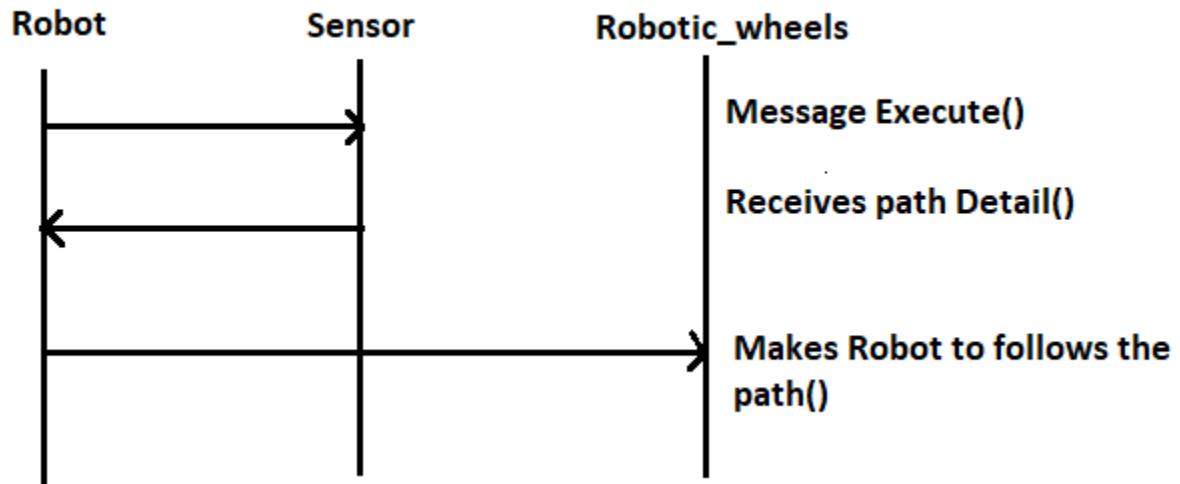
Three CRC cards are given below.

Robot	
Senses the path Follows the sensors Steer itself forward	Sensor Robotic_Wheels

Sensors	
Senses the path Knows which path to go Gives information to Robotic wheels	Robotic_Wheels

Robotic_Wheels	
Makes Robot move Receives information from sensor Follows the path	Robot Sensors

Interaction diagram for above CRC cards



CHAPTER-1 (THEORY SOLUTION)

1. Explain the notation “Everything is an object” in an object oriented Programming.[PU:2017 spring]

An entity that has state and behavior is known as an object .If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior .If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

In object oriented programming problem is analyzed in terms of object and nature of communication between them. Program object should be chosen such that they match closely with real-world objects. In object oriented programming object's state is stored as data member and behavior is defined using functions (methods).

When a program is executed, the object interact by sending message to one another. For example, if “Customer” and “account” are two objects in a program then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code (methods) to manipulate data. Objects can interact without having to know details of each other’s data or code. It is sufficient to know the type of message accepted, and the type of response returned by objects.

So, from above discussion we can say that “Everything is an object” in an object oriented Programming.

2. With the help of object oriented programming, explain how can object oriented programming cope in solving complex problem. [PU:2014 spring][PU:2018 fall]

When the software sizes grows large, due to the interconnections of software components the complexity of software system increases. The interconnections of software components means the dependence of one portion of code on another portion. Due to complexity, it is difficult to understand program in isolation as well it makes difficult to divide a task between a several programs.

The abstraction mechanism is used to manage complexity. The Abstraction is mechanism displays only essential information and hiding the details.

Abstraction is often combined with a division into components. For example, we divided the automobile into the engine and the transmission. Components are carefully chosen so that they can encapsulate certain key features and interact with another component through simple and fixed interface.

The division of components means we can divide large task into smaller problems that can then be worked on more less or less independently of each other. It is a responsibility of a developer of a component to provide a implementation that satisfies the requirement of interface. So, set of procedure written by one programmer can be used by many other programmers without knowing the exact details of implementation. They needed only the necessary interface.

From above discussion we can conclude that, breaking down the system in a way such that component can be reuse in solution of different problems just by knowing only the interfaces which must utilize. The implementation details can be modified in future without affecting the program. So program can also easily maintained. In this way object oriented programming cope in solving complex problem.

3. What influence is an object oriented approach said to have on software system design? What is your own opinion? Justify through example.[PU:2009 fall]

Object oriented approach contributes greater programmer productivity, better quality of software and lesser maintenance cost.

The influence of an object oriented approach is discussed below with certain characteristic of OOP which justifies that object oriented approach leads to better software design.

- It is easy to model a real system as programming objects in OOP represents real objects. the objects are processed by their member data and functions. It is easy to analyze the user requirements.
- Complexity that arises due to interconnections of software components can be managed by abstraction. So, set of procedure written by one programmer can be used by many other programmers without knowing the exact details of implementation. They needed only the necessary interface.
- It modularize the programs. Modular programs are easy to develop and can be distributed independently among different programmers.
- Large problems can be reduced to smaller and more manageable problems. It is easy to partition the work in a project based on objects.
- It makes software easier to maintain. Since the design is modular, part of the system can be updated in case of issues without a need to make large-scale changes.
- Elimination of redundant code due to inheritance, that is, we can use the same code in a base class by deriving a new class from it. The reusability feature of OOP lowers the cost of software development.
- Reuse also enables faster development. Object-oriented programming languages come with rich libraries of objects, and code developed during projects is also reusable in future projects.

- It provides a good framework for code libraries where the supplied software components can be easily adapted and modified by the programmer. This is particularly useful for developing graphical user interfaces.
- In OOP, data can be made private to a class such that only member functions of the class can access the data. This principle of data hiding helps the programmer to build a secure program that cannot be invaded by code in other part of the program.
- With the help of polymorphism, the same function or same operator can be used for different purposes. This helps to manage software complexity easily.

4. What is the significance of forming abstractions while designing an object oriented System. Explain with a example.[PU:2015 fall]

Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car.

The significance of forming abstractions while designing an object oriented system can be listed as follows:

- It manage complexity that arises due to interconnections of software components.
- Set of procedure written by one programmer can be used by many other programmers without knowing the exact details of implementation. They needed only the necessary interface.
- Data abstraction increases the reusability of the code by avoiding any chances of redundancy.
- It increases the readability of the code as it eliminates the possibility of displaying the complex working of the code.
- With the implementation of classes and objects, comes enhanced security. Since data abstraction is a method of implementing classes and objects any denying access to other classes of accessing the data members and member functions of the base class.
- Helps the user to avoid writing the low level code.
- It separates the entire program into code and implementation making it more comprehensible.

- Can change internal implementation of class independently without affecting the user level code.
- Helps to increase security of an application or program as only important details are provided to the user.
- To increase security of an application or program as only important details are provided to the user.

5. What are the mechanism of data abstraction? What is the use of abstraction mechanism in C++? Explain with example.[PU:2019 fall]

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation

Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC separates the implementation details from the external interface.

Data Abstraction can be achieved in following ways:

- Abstraction using classes
- Abstraction in header files
- Abstraction using access specifiers

Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

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

For example;

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

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

Example:

```
#include <iostream>
using namespace std;
class AbstractionExample
{
private:
int a, b;
public:
void setdata(int x, int y)
{
a = x;
b = y;
}
void display()
{
cout<<"a = " <<a << endl;
cout<<"b = " << b << endl;
}
};
int main()
{
AbstractionExample obj;
obj.setdata(10, 20);
obj.display();
return 0;
}
```

In above program ,By making data members private, we have hidden them from outside world. These data members are not accessible outside the class. The only way to set and get their values is through the public functions.

The main use of abstraction mechanism in C++ are as follows:

- To manage complexity that arises due to interconnections of software components using abstraction.
- Set of procedure written by one programmer can be used by many other programmers without knowing the exact details of implementation. They needed only the necessary interface. The program can use the function `sort_name()` to sort names in alphabetical order without knowing whether implementation uses bubble sort, merge sort, quick sort algorithms.
- To change the Internal implementation without affecting the user level code.
- To avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- To increase the readability of the code as it eliminates the possibility of displaying the complex working of the code.
- To increase security of an application or program as only important details are provided to the user.

CHAPTER 2 (THEORY SOLUTION)

- 1. What sorts of shortcomings of structure are addressed by classes? Explain giving appropriate example.[PU:2014 fall]**

The shortcoming of structure that are addressed by classes are as follows:

- 1) Class can create a subclass that will inherit parent's properties and methods, whereas Structure does not support the inheritance.
- 2) Structure members can be easily accessed by the structure variables in their scope but class members doesn't due to feature of data hiding.
- 3) Classes support polymorphism (late binding), whereas structure doesn't.
- 4) Also we can use "this" pointers in classes due to which you don't have to explicitly pass an object to a member function.
- 5) The structure data type cannot be treated like built in types while performing arithmetic operations. But it is valid in class using the concept of operator overloading.

For example:

```
struct complex
{
    int real;
    int imag;
}
struct complex c1,c2,c3;
c3=c1+c2; Is illegal .
```

- 2. Differentiate between structure and class. Why Class is preferred over structure?
Support your answer with suitable examples.[PU:2016 fall]**

Structure	Class
1. A structure is a collection of variables of different data types under a single unit.	1. A class is a user-defined blueprint or prototype from which objects are created.
2. To define structure we will use "struct" keyword.	2. To define class we will use "class" keyword.
3. A structure has all members public by default.	3. A class has all members private by default.

4. A Structure does not support inheritance.	4. A Class can inherit from another class. Which means class supports inheritance.
5. Structures are good for small and isolated model objects.	5. Classes are suitable for larger or complex objects.
6. Structure is a value type and its object is created on the stack memory.	6. Class is a reference type and its object is created on the heap memory.
7. Function member of the struct cannot be virtual or abstract	7. Function member of the class can be virtual or abstract.

Due to the following reasons class is preferred over structure.

- Structures in C++ doesn't provide data hiding where as a class provides data hiding
- A Structure is not secure and cannot hide its implementation details from the end user while a class is secure and can hide its programming and designing details.
- Classes support polymorphism (late binding), whereas structure doesn't.
- Class support inheritance but structure doesn't.
- "this" pointer will works only with class.
- Class lets us to use constructors and destructors.

3. Explain the various access specifier used in C++ with example.

Access Modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++.They are:

1) Public:

All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

2) Private:

The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

```
#include<iostream>
using namespace std;
class Rectangle
{
private:
float length,breadth,area;
public:
void setdata(float l,float b)
{
length=l;
breadth=b;
}
void disp_area()
{
area=length*breadth;
cout<<"Area="<<area<<endl;
}
};
int main()
{
Rectangle r;
r.setdata(10.5,6);
r.disp_area();
return 0;
}
```

Here, member function disp_area() is public so it is accessible outside class .But data member length ,breadth and area being private they are not accessible outside class. Only member function of that class disp_area() can access it.

3) Protected

class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

Example:

```
#include <iostream>
using namespace std;
class A
{
protected:
int a, b;
public:
void setdata(int x, int y)
{
a = x;
b = y;
}
};
class B : public A
{
public:
void display()
{
cout << "value of a=" << a << endl;
cout << "value of b=" << b << endl;
cout << "Sum of two protected variables a and b = "<< a + b << endl;
}
};
int main()
{
B obj;
obj.setdata(4, 5);
obj.display();
return 0;
}
```

In above example , protected variable a and b is accessed by the derived class B.

From above discussion we can conclude that:

Access Specifier	Accessible from own class	Accessible from derived class	Accessible from objects outside class
public	yes	yes	yes
private	yes	no	no
protected	yes	yes	no

**4. What is data hiding/Information hiding? How do you achieve data hiding in C++?
Explain with suitable program.[PU:2019 fall]**

Data Hiding means protecting the data members of a class from an illegal or unauthorized access from outside class. It ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.

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

```
#include<iostream>
using namespace std;
class Square
{
private:
int num;
public:
void getdata()
{
cout << "Enter the number" << endl;;
cin >> num;
}
void display()
{
cout << "Square of a given number= " << num * num << endl;
}
};
int main()
{
Square obj;
obj.getdata();
obj.display();
return 0;
}
```

In the above example, the variable “num” is private. Hence this variable can be accessed only by the member function of the same class and is not accessible from anywhere else. Hence outside the classes will be unable to access this variable which is called data hiding. In this way data hiding can be achieved.

5. What is encapsulation? How can encapsulation enforced in C++? Explain with suitable example code. [PU:2017 spring]

Encapsulation is a process of combining data members and functions in a single unit called class.

In encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.

Encapsulation can be enforced by declaring all the variables in the class as private and writing public methods in the class to set and get the values of variable.

Example:

```
#include<iostream>
using namespace std;
class Square
{
private:
int num;
public:
void setnum(int x)
{
num=x;
}
int getnum()
{
return (num*num);
}
};
int main()
{
Square obj;
obj.setnum(5);
cout<<"Square of a number="<<obj.getnum()<<endl;
return 0;
}
```

In the above program the Class square is encapsulated as the variables are declared as private. The get methods getnum() is set as public, this method is used to access these variables. The setter method like setnum() is also declared as public and is used to set the values of the variables.

6. Where do you use friend function? What are the merits and di-merits of friend function?

In some situation non-member function need to access the private data of class or one class wants to access private data of second class and second wants to access private data of first class. This can achieve by using friend functions.

The non-member function that is “friendly” to a class, has full access rights to the private members of the class.

For example when we want to compare two private data members of two different classes in that case you need a common function which can make use of both the private variables of different class. In that case we create a normal function and make friend in both the classes, as to provide access of theirs private variables.

Merits of friend function:

- It can access the private data member of class from outside the class
- Allows sharing private class information by a non-member function.
- It acts as the bridge between two classes by operating on their private data's.
- It is able to access members without need of inheriting the class.
- It provides functions that need data which isn't normally used by the class.

Demerits of friend function:

- It violates the law of data hiding by allowing access to private members of the class from outside the class.
- Breach of data integrity
- Conceptually messy
- Runtime polymorphism in the member cannot be done.
- Size of memory occupied by objects will be maximum

7. Does friend function violate the data hiding? Explain Briefly.[PU:2017 fall]

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

Hence, friend function is not a member function of the class but can access private members of that class. So that's why friend function violate data hiding.

Let's illustrate the given concept with the following example.

```
#include<iostream>
using namespace std;
class sample
{
private:
int a,b;
public:
void setdata(int x,int y)
{
a=x;
b=y;
}
friend void sum(sample s);
};
void sum(sample s)
{
cout<<"sum="<<(s.a+s.b)<<endl;
}
int main()
{
sample s;
s.setdata(5,10);
sum(s);
return 0;
}
```

Here in class named sample there are two private data members a and b. The function sum() is not a member function but when it is declared as friend in class sample then sum() can access the private data a and b. which shows that friend function violate the concept of data hiding.

8. What are the limitations of static member functions

Limitations of static member functions:

- It doesn't have a "this" pointer.
- A static member function cannot directly refer to static members.
- Same function can't have both static and non static types.
- It can't be virtual.
- A static member function can't be declared as const or volatile.

9. Data hiding Vs Encapsulation

Data hiding	Encapsulation
1) Data Hiding means protecting the members of a class from an illegal or unauthorized access.	1) Encapsulation means wrapping the implementation of data member and methods inside a class.
2) Data hiding focus more on data security.	2) Encapsulation focuses more on hiding the complexity of the system.
3) The data under data hiding is always private and inaccessible.	3) The data under encapsulation may be private or public.
4) When data member of class are private only member function of that class can access it.	4) When implementation of all the data member and methods inside a class are encapsulated, the method name can only describe what action it can perform on an object of that class.
5) Data hiding is a process as well as technique.	5) Encapsulation is a sub-process in data hiding.

10. Abstraction Vs Data hiding

Abstraction	Data hiding
1) Abstraction is a mechanism of expressing the necessary properties by hiding the details.	1) Data Hiding means protecting the members of a class from an illegal or unauthorized access
2) It's purpose is to hide complexity.	2) It's purpose is to achieve encapsulation.
3) Class uses the abstraction to derive a new user-defined datatype	3) Data hiding is used in a class to make its data private.
4) It focuses on observable behavior of data.	4) It focuses on data security.

Tutorial-1

Program

1. Declare a C++ structure (Program) to contain the following piece of information about cars on used car lot. [PU:2013 spring]
 - i. Manufacturer of the car
 - ii. Model name of the car
 - iii. The asking price of car
 - iv. The number of miles on odometer

```
#include<iostream>
using namespace std;
struct car
{
    char name[50];
    char model[50];
    long int price;
    int miles;
};
int main()
{
    car c;
    cout<<"Enter manufacturer of car"<<endl;
    cin.getline(c.name,50);
    cout<<"Enter the model name of car"<<endl;
    cin.getline(c.model,50);
    cout<<"Enter the price of car"<<endl;
    cin>>c.price;
    cout<<"Enter number of miles on odometer"<<endl;
    cin>>c.miles;
    cout<<"Information of cars is:"<<endl;
    cout<<"Manufacturer name:"<<c.name<<endl;
    cout<<"Model name of car:"<<c.model<<endl;
    cout<<"Price of car:"<<c.price<<endl;
    cout<<"Number of miles on odometer:"<<c.miles<<endl;
    return 0;
}
```

2. Create a class called Employee with three data members (empno , name, address), a function called readdata() to take in the details of the employee from the user, and a function called displaydata() to display the details of the employee. In main, create two objects of the class Employee and for each object call the readdata() and the displaydata() functions. [PU:2005 fall]

```

#include<iostream>
using namespace std;
class Employee
{
private:
int empno;
char name[20];
char address[20];
public:
void readdata();
void displaydata();
};
void Employee::readdata()
{
cout<<"Enter the Employee number"<<endl;
cin>>empno;
cout<<"Enter the Employee name"<<endl;
cin>>name;
cout<<"Enter the address"<<endl;
cin>>address;
}
void Employee::displaydata()
{
cout<<"Employee number ="<<empno<<endl;
cout<<"Employee name ="<<name<<endl;
cout<<"Employee address ="<<address<<endl;
}
int main()
{
Employee e1,e2;
cout<<"Enter the information of first employee"<<endl;
e1.readdata();
cout<<"Enter the information of second employee"<<endl;
e2.readdata();
cout<<"Information of first employee is "<<endl;
e1.displaydata();
cout<<"Information of second employee is "<<endl;
e2.displaydata();
return 0;
}

```

3. Create a class called student with three data members

(stdnt_name[20],faculty[20],roll_no), a function called readdata() to take the details of the students from the user and a function called displaydata() to display the details the of the students. In main, create two objects of the class student and for each object call both of the functions. [PU:2010 fall]

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

class student
{
private:
char stdnt_name[20];
char faculty[20];
int roll_no;
public:
void readdata();
void displaydata();
};

void student::readdata()
{
cout<<"Enter student name"<<endl;
cin>>stdnt_name;
cout<<"Enter student faculty"<<endl;
cin>>faculty;
cout<<"Enter student roll number"<<endl;
cin>>roll_no;
}

void student::displaydata()
{
cout<<"Name="<<stdnt_name<<endl;
cout<<"Roll no="<<roll_no<<endl;
cout<<"Faculty="<<faculty<<endl;
}
```

```

int main()
{
student st1,st2;
cout<<"Enter the information of first student"<<endl;
st1.readdata();
cout<<"Enter the information of second student"<<endl;
st2.readdata();
cout<<"Information of first student is:"<<endl;
st1.displaydata();
cout<<"Information of second student is:"<<endl;
st2.displaydata();
return 0;
}

```

4. Modify the **Que.no 3** for 20 students using array of object.

```

#include<iostream>
using namespace std;
class student
{
private:
char stdnt_name[20];
char faculty[20];
int roll_no;
public:
void readdata();
void displaydata();
};
void student::readdata()
{
cout<<"Enter student name"<<endl;
cin>>stdnt_name;
cout<<"Enter student faculty"<<endl;
cin>>faculty;
cout<<"Enter student roll number"<<endl;
cin>>roll_no;
}

```

```

void student::displaydata()
{
    cout<<"Name="<<stdnt_name<<endl;
    cout<<"Roll no="<<roll_no<<endl;
    cout<<"Faculty="<<faculty<<endl;
}
int main()
{
    student st[20];
    int i;
    for(i=0;i<20;i++)
    {
        cout<<"Enter the information of student:"<<i+1<<endl;
        st[i].readdata();
    }
    for(i=0;i<20;i++)
    {
        cout<<"Information of student:"<<i+1<<endl;
        st[i].displaydata();
    }
    return 0;
}

```

5. WAP to perform the addition of time in hours, minutes and seconds format.

```

#include<iostream>
using namespace std;
class time
{
private:
    int hours,minutes,seconds;
public:
    void gettime(int h,int m,int s)
    {
        hours=h;
        minutes=m;
        seconds=s;
    }

```

```

void display()
{
    cout<<hours<<"hours"<<minutes<<"minutes"<<seconds<<"seconds"<<endl;
}

void sum(time t1,time t2)
{
    seconds=t1.seconds+t2.seconds;
    minutes=seconds/60;
    seconds=seconds%60;
    minutes=minutes+t1.minutes+t2.minutes;
    hours=minutes/60;
    minutes=minutes%60;
    hours=hours+t1.hours+t2.hours;
}
};

int main()
{
    time t1,t2,t3;
    t1.gettime(2,45,35);
    t2.gettime(3,30,40);
    t3.sum(t1,t2);
    t3.display();
    return 0;
}

```

- 6. WAP to perform the addition of time using the concept of returning object as argument.**

```

#include<iostream>
using namespace std;
class time
{
private:
    int hours,minutes,seconds;
public:
    void gettime(int h,int m,int s)
    {
        hours=h;
        minutes=m;
        seconds=s; }

```

```

void display()
{
    cout<<hours<<"hours"<<minutes<<"minutes"<<seconds<<"seconds"<<endl;
}
time sum(time t1,time t2)
{
    time t;
    t.seconds=t1.seconds+t2.seconds;
    t.minutes=t.seconds/60;
    t.seconds=t.seconds%60;
    t.minutes=t.minutes+t1.minutes+t2.minutes;
    t.hours=t.minutes/60;
    t.minutes=t.minutes%60;
    t.hours=t.hours+t1.hours+t2.hours;
    return t;
}
int main()
{
    time t1,t2,t3,result;
    t1_gettime(2,45,35);
    t2_gettime(3,30,40);
    result=t3.sum(t1,t2);
    result.display();
    return 0;
}

```

7. WAP to create two distance objects with data members feet, inches and a function call by one object passing second object as function argument and return third object adding two objects. Hint: $d3=d1.adddistance(d2);$

```

#include<iostream>
using namespace std;
class dist
{
private:
    int feet;
    int inch;
public:
    void getdata();
    dist add(dist);
    void display();
};

```

```

void dist::getdata()
{
    cout<<"Enter feet:"<<endl;
    cin>>feet;
    cout<<"Enter inch:"<<endl;
    cin>>inch;
}
void dist::display()
{
    cout<<"The sum of distances=" ;
    cout<<feet<<"feet and "<<inch<<"inches"<<endl;
}
dist dist::add( dist d2)
{
    dist sum;
    sum.inch=inch+d2.inch;
    sum.feet=sum.inch/12;
    sum.inch=sum.inch%12;
    sum.feet=sum.feet+feet+d2.feet;
    return (sum);
}
int main()
{
    dist d1,d2,d3;
    cout<<"Enter first Distance"<<endl;
    d1.getdata();
    cout<<"Enter second Distance"<<endl;
    d2.getdata();
    d3=d1.add(d2);
    d3.display();
    return 0;
}

```

8. Create a class called Rational having data members nume and deno and using friend function find which one is greater.

```
#include<iostream>
using namespace std;
class Rational
{
    private:
        int nume,deno;
    public:
        void getdata()
        {
            cout<<"Enter the value of numerator"<<endl;
            cin>>nume;
            cout<<"Enter the value of denominator"<<endl;
            cin>>deno;
        }

        friend void max(Rational r);
};

void max(Rational r)
{
    if(r.nume>r.deno)
    {
        cout<<"Maximum value="<<r.nume<<endl;
    }
    else
    {
        cout<<"Maximum value="<<r.deno<<endl;
    }
}

int main()
{
    Rational r;
    r.getdata();
    max(r);
    return 0;
}
```

9. WAP to add the private data of three different classes using friend function.

```
#include<iostream>
using namespace std;
class B;
class C;

class A
{
private:
int x;
public:
void setdata(int num)
{
x=num;
}
friend void sum(A,B,C);
};

class B
{
private:
int y;
public:
void setdata(int num)
{
y=num;
}
friend void sum(A,B,C);
};

class C
{
private:
int z;
public:
void setdata(int num)
{
z=num;
}
friend void sum(A,B,C);
};
```

```

void sum(A m,B n,C o)
{
cout<<"sum="<<(m.x+n.y+o.z)<<endl;
}

int main()
{
A p;
B q;
C r;
p.setdata(5);
q.setdata(10);
r.setdata(15);
sum(p,q,r);
return 0;
}

```

10. Write a program to find the largest of four integers .your program should have three classes and each classes have one integer number.[PU:2014 spring]

```

#include<iostream>
using namespace std;
class Y;
class Z;

class X
{
private:
int a;
public:
void getdata()
{
cout<<"Enter first number"<<endl;
cin>>a;
}
friend void max(X,Y,Z);
};

```

```

class Y
{
private:
int b;
public:
void getdata()
{
cout<<"Enter second number"<<endl;
cin>>b;
}
friend void max(X,Y,Z);
};

class Z
{
private:
int c;
public:
void getdata()
{
cout<<"Enter third number"<<endl;
cin>>c;
}
friend void max(X,Y,Z);
};

void max(X m,Y n,Z o)
{
int d;
cout<<"Enter fourth number"<<endl;
cin>>d;
if(m.a>n.b&&m.a>o.c&&m.a>d)
{
cout<<"Largest number="<<m.a<<endl;
}
else if(n.b>o.c&&n.b>d)
{
cout<<"Largest number="<<n.b<<endl;
}
else if(o.c>d)
{
cout<<"Largest number="<<o.c<<endl;
}
else
{
cout<<"Largest number="<<d<<endl;
}
}

```

```

int main()
{
X p;
Y q;
Z r;
p.getdata();
q.getdata();
r.getdata();
max(p,q,r);
return 0;
}

```

11. WAP to add two complex numbers of two different classes using friend function.

```

#include<iostream>
using namespace std;
class complex2;
class complex1
{
private:
int real,imag;
public:
void getdata()
{
cout<<"Enter real and imaginary part"<<endl;
cin>>real>>imag;
}
void display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
friend void addcomplex(complex1,complex2);
};

```

```

class complex2
{
private:
int real,imag;
public:
void getdata()
{
cout<<"Enter real and imaginary part"<<endl;
cin>>real>>imag;
}
void display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
friend void addcomplex(complex1,complex2);
};

void addcomplex(complex1 c1,complex2 c2)
{
int real,imag;
real=c1.real+c2.real;
imag=c1.imag+c2.imag;
cout<<"sum of complex number="<<real<<"+"<<imag<<"i"<<endl;
}

int main()
{
complex1 c1;
complex2 c2;
c1.getdata();
c2.getdata();
cout<<"first complex number"<<endl;
c1.display();
cout<<"second complex number"<<endl;
c2.display();
addcomplex(c1,c2);
return 0;
}

```

12. WAP to add complex numbers of two different classes using friend class.

```
#include<iostream>
using namespace std;
class complex2;
class complex1
{
private:
int real,imag;
public:
void getdata()
{
cout<<"Enter real and imaginary part"<<endl;
cin>>real>>imag;
}
void display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
friend class complex2;
};
class complex2
{
private:
int real,imag;
public:
void getdata()
{
cout<<"Enter real and imaginary part"<<endl;
cin>>real>>imag;
}
void display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
void addcomplex(complex1 c1)
{
real=c1.real+real;
imag=c1.imag+imag;
}
};
```

```

int main()
{
complex1 c1;
complex2 c2;
cout<<"For first complex number"<<endl;
c1.getdata();
cout<<"For second complex number"<<endl;
c2.getdata();
c2.addcomplex(c1);
cout<<"sum of complex number="<<endl;
c2.display();
return 0;
}

```

13. Using class write a program that receives inputs principle amount, time and rate.

Keeping rate 8% as the default argument, calculate simple interest for three customers.[PU:2019 fall]

```

#include<iostream>
using namespace std;
class customer
{
private:
float principle,rate,si;
int time;
public:
void setdata(float p,int t,float r=8);
void display();
};
void customer::setdata(float p,int t,float r)
{
principle=p;
time=t;
rate=r;
}
void customer::display()
{
si=(principle*time*rate)/100;
cout<<"Simple Interest="<<si<<endl;
}

```

```

int main()
{
float p;
int t,i;
customer c[3];
for(i=0;i<3;i++)
{
cout<<"Enter principle and time for customer"<<i+1<<endl;
cin>>p>>t;
c[i].setdata(p,t);
c[i].display();
}
return 0;
}

```

14. Create classes called class1 and class2 with each of having one private member .Add member function to set a value(say setvalue) one each class. Add one more function max() that is friendly to both classes. max() function should compare two private member of two classes and show maximum among them. Create one-one object of each class and then set a value on them. Display the maximum number among them.[PU:2015 fall,2016 fall]

Solution: [see chapter2 page23]

15. WAP to swap private data of two different classes.

Solution: [see chapter2 page26]

16. Create a new class named City that will have two member variables CityName (char[20]),and DistFromKtm (float).Add member functions to set and retrieve the CityName and DistanceFromKtm separately. Add new member function AddDistance that takes two arguments of class City and returns the sum of DistFromKtm of two arguments. In the main function, Initialize three city objects .Set the first and second City to be pokhara and Dhangadi. Display the sum ofDistFromKtm of Pokhara and Dhangadi calling AddDistance function of third City object. [PU: 2010 Spring]

```
#include<iostream>
#include<string.h>
using namespace std;
class City
{
private:
char CityName[20];
float DistFromKtm;
public:
void setdata(char cname[],float d)
{
strcpy(CityName,cname);
DistFromKtm=d;
}
void display()
{
cout<<"City name=";<<CityName<<endl;
cout<<"Distance from ktm=";<<DistFromKtm<<endl;
}
float AddDistance(City c1,City c2)
{
return (c1.DistFromKtm+c2.DistFromKtm);
}
};
int main()
{
City c1,c2,c3;
c1.setdata("Pokhara",250);
c2.setdata("Dhangadi",150);
cout<<"Information of first city"<<endl;
c1.display();
cout<<"Information of second city"<<endl;
c2.display();
cout<<"sum of DistFromKtm of Pokhara and Dhangadi=";<<c3.AddDistance(c1,c2);
return 0;
}
```

17. Create a class called Volume that uses three Variables (length, width, height) of type distance (feet and inches) to model the volume of a room. Read the three dimensions of the room and calculate the volume it represent, and print out the result .The volume should be in (feet3) form ie. you will have to convert each dimension into the feet and fraction of foot. For instance , the length 12 feet 6 inches will be 12.5 ft)[PU: 2009 spring]

```
#include<iostream>
using namespace std;
class volume
{
private:
int lf,wf,hf,li,wi,hi;
float vol,length,width,height;
public:
void getdata();
void convertdim();
void display();
};
void volume::getdata()
{
cout<<"Enter the length of room in feet and inches" << endl;
cin>>lf>>li;
cout<<"Enter the width of room in feet and inches" << endl;
cin>>wf>>wi;
cout<<"Enter the Height of room in feet and inches" << endl;
cin>>hf>>hi;
}
void volume::convertdim()
{
length=lf+(float)li/12;
width=wf+(float)wi/12;
height=hf+(float)hi/12;
cout<<"The length of room =" << length << endl;
cout<<"The width of room =" << width << endl;
cout<<"The Height of room=" << height << endl;
}
void volume::display()
{
vol=length*width*height;
cout<<"Volume of Room =" << vol << endl;
}
```

```

int main()
{
volume v;
v.getdata();
v.convertdim();
v.display();
return 0;
}

```

- 18. WAP to read two complex numbers and a function that calls by passing references of two objects rather than values of objects and add into third object and returns that object.**

```

#include<iostream>
using namespace std;
class Complex
{
private:
int real,imag;
public:
void getdata();
void display();
Complex sum( Complex &c1,Complex &c2);
};
void Complex::getdata()
{
cout<<"Enter real part and imaginary part"<<endl;
cin>>real>>imag;
}
void Complex::display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
Complex Complex::sum( Complex &c1,Complex &c2)
{
Complex c3;
c3.real=c1.real+c2.real;
c3.imag=c1.imag+c2.imag;
return c3;
}

```

```

int main()
{
Complex c1,c2,c3,c4;
c1.getdata();
c2.getdata();
c4=c3.sum(c1,c2);
cout<<"sum=";
c4.display();
return 0;
}

```

- 19. WAP in C++ to calculate simple interest from given principal, time and rate. Set the rate to 15 % as default argument when rate is not provided.**

```

#include<iostream>
using namespace std;
float calculate(float principle,int time,float rate=15);
int main()
{
float p,r;
int t;
char ch;
cout<<"Enter principle and time"<<endl;
cin>>p>>t;
cout<<"Do you want to enter rate?"<<endl;
cin>>ch;
if(ch=='Y' | | ch=='y')
{
cout<<"Enter rate"<<endl;
cin>>r;
cout<<"Interest="<<calculate(p,t,r)<<endl;
}
else
{
cout<<"Interest="<<calculate(p,t)<<endl;
}
return 0;
}
float calculate(float principle,int time,float rate)
{
return(principle*time*rate)/100.0;
}

```

20. Create a class comparison with data members x and y and max and member function getdata() to read the value of x, y, largest() to find greater of two and print() to the greater number.

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

class comparison
{
private:
int x,y,max;
public:
void getdata()
{
cout<<"Enter two numbers"<<endl;
cin>>x>>y;
}
int largest()
{
max=(x>y)?x:y;
return max;
}
void print()
{
cout<<"Greater number="<<max<<endl;
}
};

int main()
{
comparison c1;
c1.getdata();
c1.largest();
c1.print();
return 0;
}
```

21. Create a class student with six data members (roll no, name, marks in English,math, science and total).Write a program init() to initializes necessary data members calctotal() and display().Create a program for one student (i.e one object only necessary)

```
#include<iostream>
using namespace std;
class student
{
private:
char name[20];
int roll;
float me,mm,ms,total;
public:
void init()
{
cout<<"Enter roll number"<<endl;
cin>>roll;
cout<<"Enter name"<<endl;
cin>>name;
cout<<"Enter Marks in English,Math and science"<<endl;
cin>>me>>mm>>ms;
}
void calctotal()
{
total=me+mm+ms;
}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"RollNo:"<<roll<<endl;
cout<<"Marks in English:"<<me<<endl;
cout<<"Marks in Math:"<<mm<<endl;
cout<<"Marks in science:"<<ms<<endl;
cout<<"Total:"<<total<<endl;
}
};
int main()
{
student st;
st.init();
st.calctotal();
st.display();
return 0; }
```

Tutorial solution (Chapter-3) Theory only

1. What is the difference between message passing and function call? Explain the basic message formalization. [PU:2006 spring]

Message Passing	Function Call
1) In a message passing there is a designated receiver for that message; the receiver is some object to which message is sent.	1) In function call, there is no designated receiver.
2) Interpretation of the message (that is method is used to respond the message) is determined by the receiver and can vary with different receivers.	2) Determination of which method to invoke is very early binding of a name to fragment in procedure calls
3) Message passing must involve name of the object, function name and information to be sent.	3) Simply function name and its arguments is used to make function call.
4) A message is always given to some object, called the receiver.	4) Name will be matched to a message to determine when the method should be executed Signature the combination of return type and argument types.
5) Example: st.getdata(2,5)	5) Example: getdata(2,5);

Explanation:

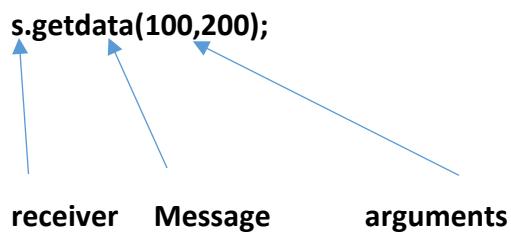
Message passing formalization

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

- A message is always given to some object, called the receiver.
- The action performed in response to the message is not fixed but may be differ, depending on the class of the receiver .That is different objects may accept the same message the and yet perform different actions.

There are three identifiable parts to any message-passing expression. These are

- 1) **Receiver:** the object to which the message is being sent
- 2) **Message selector:** the text that indicates the particular message is being sent.
- 3) **Arguments** used in responding the message.



```

st.getdata(100,200);
receiver Message arguments

```

Example of message passing

```

#include<iostream>
#include<conio.h>
using namespace std;
class student
{
int roll;
public:
void getdata(int x)
{
roll=x;
}
void display()
{
cout<<"Roll number="<<roll;
}
};

int main()
{
student s;
s.getdata(325); //objects passing message
s.display(); //objects passing message
getch();
return 0;
}

```

2. What are the possible memory errors in programming.[PU:2014 spring]

Memory errors occur very commonly in programming, and they can affect application stability and correctness. These errors are due to programming bugs. They can be hard to reproduce, hard to debug, and potentially expensive to correct as well. Applications that have memory errors can experience major problems.

Memory errors can be broadly classified into Heap Memory Errors and Stack Memory Errors. Some of the challenging memory errors are:

- 1. Invalid Memory Access in stack and heap:** This error occurs when a read or write instruction references unallocated or deallocated memory.
- 2. Memory leaks**
Memory leaks occur when memory is allocated but not released.
- 3. Mismatched Allocation/Deallocation**
This error occurs when a deallocation is attempted with a function that is not the logical counterpart of the allocation function used.
To avoid mismatched allocation/deallocation, ensure that the right de-allocator is called. In C++, new[] is used for memory allocation and delete[] for freeing up.
- 4. Missing allocation**
This error occurs when freeing memory which has already been freed. This is also called "repeated free" or "double free".
- 5. Uninitialized Memory Access**
This type of memory error will occur when an uninitialized variable is read in your application. To avoid this type of memory error, always initialize variables before using them.

3. Is it mandatory to use constructor in class. Explain?

Constructor is a 'special' member function whose task is to initialize the object of its class.

It is not mandatory to use constructor in a class. As we know, Constructor are invoked automatically when the objects are created.

So that object are initialized at the time of declaration. There is no need to separate function call to initialize the object. Which reduces coding redundancy and minimizes the programming error such as uninitialized memory access.

4. Differentiate between constructor and destructor.

Constructor	Destructor
1) Constructor is used to initialize the instance of the class.	1) Destructor destroys the objects when they are no longer needed.
2) Constructors is called when new instances of class is created.	2) Destructor is called when instances of class is deleted or released.
3) Constructor allocates the memory.	3) Destructor releases the memory.
4) Constructor can have arguments.	4) Destructor cannot have any arguments.
5) Overloading of constructor is possible.	5) Overloading of Destructor is not possible.
6) Constructor have same name as class name.	6) Destructor have same name as class name but with tilde (~) sign.
7) Syntax: Class_name(arguments) { //Body of the constructor }	7) Syntax: ~Class_name() { //Body of the destructor }

**5. Discuss the various situations when a copy constructor is automatically invoked.
How a default constructor can be equivalent to constructor having default arguments.**

Various situations when the copy constructor is called or automatically invoked are:

- When compiler generates the temporary object.
- When an object is constructed based on the object of the same class.
- When an object of the class is passed to function by values as an argument.
- When an object of the class is returned by value.

Constructor with default arguments is a parameterized constructor which has default values in arguments. Thus the arguments defined in such constructor are optional. We may or may not pass arguments while defining object of the class. When an object is created with no supplied values, the default values are used. By this way, constructor with default constructor is equivalent to default constructor.

Example:

```
#include<iostream>
using namespace std;
class Box
{
private:
float l,b,h;
public:
Box(float le=10,float br=5,float he=5)
{
    l=le;
    b=br;
    h=he;
}
void displaymember()
{
    cout<<"Length,breadth and height"<<l<<b<<h<<endl;
}
float getvolume()
{
    return (l*b*h);
}
};
int main()
{
    Box b;
    float vol;
    vol=b.getvolume();
    cout<<"Volume="<<vol<<endl;
    return 0;
}
```

- 6. What is de-constructor? can you have two destructors in a class? Give example to support your reason.[PU:2014 spring]**

De-constructor is a member function that destroys the object that have been created by a constructor.

Destructor doesn't take any arguments, which means destructor cannot be overloaded. So, that there will be only one destructor in a class.

Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

So, that there cannot be more than one destructor (two destructor) in a same class.

(Write example of destructor yourself)

7. Differentiate methods of argument passing in constructor and destructor.

A constructor is allowed to accept the arguments as the arguments can be used to initialize the data members of the class.

A destructor does not accept any arguments as its only work is to deallocate the memory of the object.

In class, there can be multiple constructors which are identified by the number arguments passed.

In class, there is only one destructor.

Constructors can be overload to perform different action under the name of the same constructor whereas, destructors cannot be overloaded.

Example:

```
#include <iostream>
using namespace std;
class test
{
private:
int a;
public:

test()
{
    a=10;
    cout<<"Default constructor is called"<<endl;
}

test(int x)
{
    a=x;
    cout<<"Parameterized constructor is called"<<endl;
}
~test()
{
    cout<<"Destructor is called"<<endl;
}
void display()
{
    cout<<"value of a="<<a<<endl;
}
};
```

```

int main()
{
    test t1;
    test t2(5);
    t1.display();
    t2.display();
    return 0;
}

```

- 8. What do you mean by stack vs Heap? Explain the memory recovery. Explain the use of new and delete operator. [PU:2009 spring]**

Stack

- It's a region of computer's memory that stores temporary variables created by each function (including the main() function).
- ☐The stack is a "Last in First Out" data structure and limited in size. Every time a function declares a new variable, it is "pushed" (inserted) onto the stack. Every time a function exits, all of the variables pushed onto the stack by that function, are freed or popped (that is to say, they are deleted).
- ☐Once a stack variable is freed, that region of memory becomes available for other stack variables.

Heap

- The heap is a region of computer's memory that is not managed automatically and is not as tightly managed by the CPU.
- We must allocate and de-allocate variable explicitly. (for allocating variable **new** and for freeing variables **delete** operator is used).
- It does not have a specific limit on memory size.

Memory recovery

Because in most languages objects are dynamically allocated, they must be recovered at run-time. There are two broad approaches to this:

- Force the programmer to explicitly say when a value is no longer being used:
`delete aCard; // C++ example`
- Use a garbage collection system that will automatically determine when values are no longer being used, and recover the memory.

Use of new and delete operator

new is used to allocate memory blocks at run time (dynamically). While, **delete** is used to de-allocate the memory which has been allocated dynamically.

Example of new and delete in C+

```
#include <iostream>
using namespace std;
int main ()
{
int i,n;
int *ptr;
int sum=0;
cout << "How many numbers would you like to Enter? ";
cin >>n;
ptr= new int[n];

for (i=0; i<n; i++)
{
cout << "Enter number:"<<i+1<<endl;
cin >> ptr[i];
}
for (i=0; i<n; i++)
{
sum=sum+ptr[i];
}
cout<<"sum of numbers="<<sum<<endl;
delete[] ptr;
return 0;
}
```

9. What are the advantages of dynamic memory allocation? Explain with suitable example. [PU:2016 spring]

The main advantage of using dynamic memory allocation is preventing the wastage of memory or efficient utilization of memory. Let us consider an example:

In static memory allocation the memory is allocated before the execution of program begins (During compilation). In this type of allocation the memory cannot be resized after initial allocation. So it has some limitations. Like

- Wastage of memory
- Overflow of memory

eg. int num[100];

Here, the size of an array has been fixed to 100. If we just enter to 10 elements only, then there will be wastage of 90 memory location and if we need to store more than 100

elements there will be memory overflow.

But, in dynamic memory allocation memory is allocated during runtime so it helps us to allocate and de-allocate memory during the period of program execution as per requirement. So there will be proper utilization of memory.

In c++ dynamic memory allocation can be achieved by using new and delete operator. **new** is used to allocate memory blocks at run time (dynamically). While, **delete** is used to de-allocate the memory which has been allocated dynamically.

Let us consider an example:

```
#include <iostream>
using namespace std;
int main ()
{
    int i,n;
    int *ptr;
    int sum=0;
    cout << "How many numbers would you like to Enter? ";
    cin >>n;
    ptr= new int[n];
    for (i=0; i<n; i++)
    {
        cout << "Enter number:"<<i+1<<endl;
        cin >> ptr[i];
    }
    for (i=0; i<n; i++)
    {
        sum=sum+ptr[i];
    }
    cout<<"sum of numbers="<<sum<<endl;
    delete[] ptr;
    return 0;
}
```

Here, in above example performs the sum of n numbers .But the value of n will be provided during runtime.so that memory allocation to store n numbers is done during program execution which results proper utilization of memory and preventing from wastage of memory or overflow of memory.

Dynamic Constructor

Dynamic constructor is used to allocate the memory to the objects at the run time. Memory is allocated at run time with the help of 'new' operator .This will enable the system to allocate right amount of memory for each object when objects are not of the same size, thus resulting in the saving of memory.

Example:

Program to find the sum of two Complex number using the concept of dynamic constructor

```
#include<iostream>
using namespace std;
class complex
{
int *real,*imag;
public:
complex()
{
real=new int;
imag=new int;
*real=0;
*imag=0;
}
complex(int r,int i)
{
real=new int;
*real=r;
imag=new int;
*imag=i;
}
void display()
{
cout<<*real<<"+"<<*imag<<endl;
}
void addcomplex(complex c1,complex c2)
{
*real=*c1.real+*c2.real;
*imag=*c1.imag+*c2.imag;
}
};
```

```

int main()
{
complex c1(5,10);
complex c2(2,4);
complex c3;
cout<<"First complex number=";
c1.display();
cout<<"Second complex number=";
c2.display();
cout<<"Sum =";
c3.addcomplex(c1,c2);
c3.display();
return 0;
}

```

WAP to concatenate two string using the concept of dynamic constructor.

```

#include<iostream>
#include<string.h>
using namespace std;
class stringc
{
char *name;
int length;
public :
    stringc()
    {
        length=0;
        name=new char[length + 1];
    }
    stringc(char s[])
    {
        length=strlen(s);
        name=new char[length+1];
        strcpy(name,s);
    }
    void join(stringc &a,stringc &b)
    {
        length=a.length+b.length;
        delete name;
        name=new char[length+1];
        strcpy(name,a.name);
        strcat(name,b.name);
    }
}

```

```

void display()
{
    cout<<name<<endl;
}

};

int main ()
{
stringc s1("Hello");
stringc s2("Nepal");
stringc s3;
s3.join(s1,s2);
s3.display();
return 0;
return 0;
}

```

Dynamic initialization of object

Dynamic initialization of object refers to initializing the objects at run time i.e. the initial value of an object is to be provided during run time. Dynamic initialization can be achieved using constructors and passing parameters values to the constructors. This type of initialization is required to initialize the class variables during run time.

Need of dynamic initialization

Dynamic initialization of objects is needed as

- It utilizes memory efficiently.
- Various initialization formats can be provided using overloaded constructors.
- It has the flexibility of using different formats of data at run time considering the situation.

Program example to explain the concept of dynamic initialization

```
#include <iostream>

using namespace std;

class simple_interest
{
    float principle , time, rate ,interest;

public:
    simple_interest (float a, float b, float c)
    {
        principle = a;
        time =b;
        rate = c;
    }
    void display ( ) {

        interest =(principle* rate* time)/100;
        cout<<"interest ="<<interest ;
    }
};

int main()
{
    float p,t,r;
    cout<<"principle amount, time and rate"<<endl;
    cin>>p>>t>>r;
    simple_interest s1(p,t,r);//dynamic initialization
    s1.display();
    return 0;
}
```

Chapter 3 Tutorial solution (Program Only)

- 1) WAP to input the name, age, and salary of employee and display the records and total number of employee using the concept of static data members.

```
#include<iostream>
using namespace std;
class Employee
{
private:
char name[20];
int age;
float salary;
static int count;
public:
void getinfo()
{
cout<<"Enter name of employee"<<endl;
cin>>name;
cout<<"Enter age"<<endl;
cin>>age;
cout<<"Enter salary"<<endl;
cin>>salary;
count++;
}
void dispinfo()
{
cout<<"Name:"<<name<<endl;
cout<<"Age:"<<age<<endl;
cout<<"Salary:"<<salary<<endl;
}
static void disptotal()
{
cout<<"Total number of employees are:"<<count<<endl;
}
};
int Employee::count;
```

```

int main()
{
Employee e1,e2,e3;
cout<<"Enter information of employees" << endl;
e1.getinfo();
e2.getinfo();
e3.getinfo();
cout<<"Details of employees are:" << endl;
e1.dispinfo();
e2.dispinfo();
e3.dispinfo();
Employee::disptotal();
return 0;
}

```

2) WAP to find area of rectangle using the concept of parameterized constructor.

```

#include<iostream>
using namespace std;
class rectangle
{
private:
int length,breadth,area;
public:
rectangle(int l,int b)
{
length=l;
breadth=b;
}
void calc()
{
area=length*breadth;
}
void display()
{
cout<<"Area of Rectangle=" << area << endl;
}
};

```

```

int main()
{
rectangle r1(10,5);
r1.calc();
r1.display();
return 0;
}

```

- 3) WAP to initialize name, roll and marks of student using constructor and display the obtained information.**

```

#include<iostream>
#include<string.h>
using namespace std;
class student
{
private:
char name[20];
int roll;
float marks;
public:
student(char n[],int r,float m)
{
strcpy(name,n);
roll=r;
marks=m;
}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll no:"<<roll<<endl;
cout<<"Marks:"<<marks<<endl;
}
};
int main()
{
student st("Ram",7,88.5);
st.display();
}

```

- 4) Create a class student with data members (name, roll, marks in English, Maths and OOPs(in 100), total, average), a constructor that initializes the data members to the values passed to its parameters, A function called calculate() that calculates the total of marks obtained and average. And function display() to display name, roll no, total and average.

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
private:
char name[20];
int roll;
float me,mm,mo,total,avg;
public:
student(char n[],int r,float e,float m,float o)
{
strcpy(name,n);
roll=r;
me=e;
mm=m;
mo=o;
}
void calculate()
{
total=me+mm+mo;
avg=total/3;
}
void print()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll no:"<<roll<<endl;
cout<<"Total:"<<total<<endl;
cout<<"Average:"<<avg<<endl;
}
};
```

```

int main()
{
char name[20];
int roll;
float meng,mmath,moops;
cout<<"Enter name"<<endl;
cin>>name;
cout<<"Enter roll"<<endl;
cin>>roll;
cout<<"Enter marks in English,Maths and OOPs"<<endl;
cin>>meng>>mmath>>moops;
student st(name,roll,meng,mmath,moops);
st.calculate();
st.print();
return 0;
}

```

- 5) **WAP to perform the addition of complex number using the concept of constructor.**

```

#include<iostream>
using namespace std;
class complex
{
private:
int real,imag;
public:
complex()
{
real=2;
imag=4;
}
complex(int r,int i)
{
real=r;
imag=i;
}
void addcomplex(complex c1,complex c2)
{
real=c1.real+c2.real;
imag=c1.imag+c2.imag;
}

```

```

void display()
{
    cout<<real<<"+"<<imag<<endl;
}
};

int main()
{
    complex c1;
    complex c2(10,20);
    complex c3;
    cout<<"First complex number="<<endl;
    c1.display();
    cout<<"Second complex number="<<endl;
    c2.display();
    c3.addcomplex(c1,c2);
    cout<<"sum="<<endl;
    c3.display();
    return 0;
}

```

6) WAP to copy complex number using the concept of constructor.

```

#include<iostream>
using namespace std;
class complex
{
private:
    int real,imag;
public:
    complex(int r,int i)
    {
        real=r;
        imag=i;
    }
    complex(complex &x)
    {
        real=x.real;
        imag=x.imag;
    }
}

```

```

void display()
{
    cout<<"Real part ="<<real<<endl;
    cout<<"Imaginary part "<<imag<<endl;
}
};

int main()
{
    complex c1(5,10);
    complex c2(c1);
    cout<<"Details of c1 "<<endl;
    c1.display();
    cout<<"Details of c2 "<<endl;
    c2.display();
    return 0;
}

```

7) WAP to concatenate strings of two objects using the concept of Dynamic Constructor.

```

#include<iostream>
#include<string.h>
using namespace std;
class strings
{
char *name;
int length;
public:
strings()
{
length=0;
name=new char[length+1];
}
strings (char *s)
{
length=strlen(s);
name=new char[length+1];
strcpy(name,s);
}
void display()
{
cout<<name<<endl;
}

```

```

void join(strings &a,strings &b)
{
length=a.length+b.length;
delete name;
name=new char[length+1];
strcpy(name,a.name);
strcat(name,b.name);
}
};

int main()
{
char name1[20];
char name2[20];
cout<<"Enter first string"<<endl;
cin>>name1;
cout<<"Enter second string"<<endl;
cin>>name2;
strings s1(name1);
strings s2(name2);
strings s3;
s3.join(s1,s2);
cout<<"String after concatenation"<<endl;
s3.display();
return 0;
}

```

- 8) WAP to perform the addition of two complex numbers using the concept of constructor overloading.**

Solution: (*Same as Que. No 5*)

- 9) Write a simple program using of dynamic memory allocation which should include calculation of marks of 3 subjects of n students and displaying the result as pass or fail and name, roll. Pass mark is 45 out of 100 in each subject.**

```

#include<iostream>
using namespace std;
class student
{
private:
char name[20];
int roll;
float m1,m2,m3;
public:
void getdata()
{
cout<<"Enter the name"<<endl;
cin>>name;
cout<<"Enter the roll"<<endl;
cin>>roll;
cout<<"Enter the marks in 3 subjects"<<endl;
cin>>m1>>m2>>m3;
}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll no:"<<roll<<endl;
if(m1>=45&&m2>=45&&m3>=45)
{
cout<<"Result=Pass"<<endl;
}
else
{
cout<<"Result=Fail"<<endl;
}
}
};

```

```

int main()
{
    int n,i;
    student *ptr;
    cout<<"Enter the number of students"<<endl;
    cin>>n;
    ptr=new student[n];
    for(i=0;i<n;i++)
    {
        cout<<"Enter the information of student"<<i+1<<endl;
        ptr[i].getdata();
    }
    for(i=0;i<n;i++)
    {
        cout<<"Information of student"<<i+1<<endl;
        ptr[i].display();
    }
    delete []ptr;
    return 0;
}

```

10) Write a program to find the difference of complex number using the concept of constructors. Input the values from main () and pass argument to the constructor.

```

#include<iostream>
using namespace std;
class complex
{
private:
int real,imag;
public:
complex()
{
}
complex(int r,int i)
{
real=r;
imag=i;
}
void diffcomplex(complex c1,complex c2)
{

```

```

real=c1.real-c2.real;
imag=c1.imag-c2.imag;
}
void display()
{
cout<<real<<"+"<<imag<<endl;
}
};

int main()
{
int real1,imag1,real2,imag2;
cout<<"Enter the real and imaginary part of first complex number"<<endl;
cin>>real1>>imag1;
cout<<"Enter the real and imaginary part of second complex number"<<endl;
cin>>real2>>imag2;
complex c1(real1,imag1);
complex c2(real2,imag2);
complex c3;
cout<<"First complex number="<<endl;
c1.display();
cout<<"Second complex number="<<endl;
c2.display();
c3.diffcomplex(c1,c2);
cout<<"Difference of complex number="<<endl;
c3.display();
return 0;
}

```

Inheritance Old question solution(Theory)

1. When base class and derived class have the same function name what happens when derived class object calls the function?[PU 2017 fall]

When the base class and derived class have the same function name, the derived class function overrides the function that is inherited from base class, which is known as function overriding. Now, when derived class object calls that overridden function, the derived class member function is accessed.

For example:

```
#include<iostream>
using namespace std;
class Base
{
public:
void display()
{
    cout<<"This is base class"<<endl;
}
};
class Derived:public Base
{
public:
void display()
{
    cout<<"This is derived class"<<endl;
}
};
int main()
{
Derived d;
d.display();
return 0;
}
```

Output:

This is derived class

In this example, the class derived inherits the public member function display() .When we redefine this function in derived class then the inherited members from base are overridden.so when derived class object calls the function display() it actually refers the function in derived class but not the inherited member function in base class.

**2. How inheritance support reusability features of OOP? Explain with example.
[PU:2010 spring]**

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

A derived class includes all features of the base class and then it can add additional features specific to derived class.

Example:

```
#include<iostream>
using namespace std;
class Sum
{
protected:
int a,b;
public:
void getdata()
{
    cout<<"Enter two numbers"<<endl;
    cin>>a>>b;
}
void display()
{
cout<<"a="<<a<<endl;
cout<<"b="<<b<<endl;
}
};
class Result:public Sum
{
private:
int total;
public:
void disp_result()
{
    total=a+b;
    cout<<"sum="<<total<<endl;
}
};
```

```
int main()
{
Result r;
r.getdata();
r.display();
r.disp_result();
return 0;
}
```

Here, In above example **Sum** is base class and **Result** is derived class. The data member **named a and b** and member function **display ()** of base class are inherited and they are used by using the object of derived class named **Result**. Hence, we can see that inheritance provides reusability.

3. How are arguments are sent to base constructors in multiple inheritance ?Who is responsibility of it.[PU:2013 spring]

In Multiple Inheritance arguments are sent to base class in the order in which they appear in the declaration of the derived class.

For example:

```
Class gamma: public beta, public alpha
{  
}
```

Order of execution

```
beta(); base constructor (first)  
alpha(); base constructor(second)  
gamma();derived (last)
```

The constructor of derived class is responsible to supply values to the base class constructor.

Program:

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

```

class alpha
{
int x;
public:
alpha(int a)
{
x=a;
cout<<"Alpha is initialized" << endl;
}
void showa()
{
cout<<"x=" << x << endl;
}
};

class beta
{
int y;
public:
beta(int b)
{
y=b;
cout<<"Beta is initialized" << endl;
}
void showb()
{
cout<<"y=" << y << endl;
}
};

class gamma:public beta,public alpha
{
int z;
public:
gamma(int a,int b,int c):alpha(a),beta(b)
{
z=c;
cout<<"Gamma is initialized" << endl;
}
void showg()
{
cout<<"z=" << z << endl;
}
};

```

```
int main()
{
gamma g(5,10,15);
g.showa();
g.showb();
g.showg();
return 0;
}
```

Output:

```
Beta is initialized
Alpha is initialized
Gamma is initialized
x=5
y=10
z=15
```

Here, **beta** is initialized first although it appears second in the derived constructor *as it has been declared first in the derived class header line*

```
class gamma: public beta, public alpha
{ }
```

If we change the order *to*

```
class gamma: public alpha, public beta
{ }
```

then alpha will be initialized first

- 4. Class ‘Y’ has been derived from class ‘X’ .The class ‘Y’ does not contain any data members of its own. Does the class ‘Y’ require constructors? If yes why.[PU:2013 spring]**

When Class ‘Y’ has been derived from class ‘X’ and class ‘Y’ does not contain any data members of its own then,

- It is mandatory have constructor in derived class ‘Y’, whether it has data members to be initialized or not, if there is constructor with arguments(parameterized constructor) in base class ‘X’.
 - It not necessary to have constructor in derived class ‘Y’ if base class ‘X’ does not contain a constructor with arguments (parameterized constructor).
- Derived class constructor is used to provide the values to the base class constructor.

Example

```
#include<iostream>
using namespace std;
class X
{
    private:
        int a;
    public:
        X (int m)
        {
            a=m;
        }
        void display()
        {
            cout<<"a="<<a<<endl;
        }
};
class Y:public X
{
public:
    Y(int b):X(b)
    {
    }
};
int main()
{
    Y obj(5);
    obj.display();
    return 0;
}
```

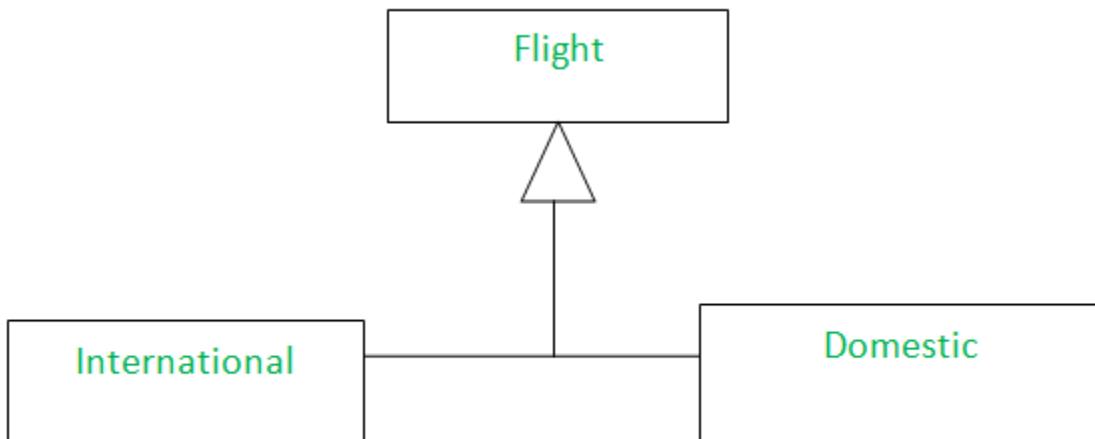
Here, in above example, class 'Y' does not have any data members but there is a parameterized constructor in class 'X' so, there is necessary to have constructor in class 'Y'.

5. Write a short notes on:

Generalization[PU:2013 spring]

Generalization is the process of taking out common properties and functionalities from two or more classes and combining them together into another class which acts as the parent class of those classes or what we may say the generalized class of those specialized classes. All the subclasses are a type of superclass. So we can say that subclass “is-A” superclass. Therefore Generalization is termed as “is-A relationship”.

Here is an example of generalization:



In this figure, we see that there are two types of flights so we made a flight class which will contain common properties and then we have an international and domestic class which are an extension of flight class and will have properties of flight as well as their own. Here flight is the parent/superclass and the other two are child/subclass. International Flight “is-a” flight as well as the domestic flight.

6. Explain how composition provide reusability.

In composition one class contains object of another class as its member data, which means members of one class are available in another class. Composition exhibits “has-a” relationship. For example, Company **has an** employee.

So, properties of one class can be used by another class independently.

Let us consider an example

```
#include<iostream>
using namespace std;
class Employee
{
int eid;
float salary;
public:
void getdata()
{
cout<<"Enter id and salary of employee"<<endl;
cin>>eid>>salary;
}
void display()
{
cout<<"Emp ID:"<<eid<<endl;
cout<<"Salary:"<<salary<<endl;
}
};
class Company
{
char cname[20];
char department[20];
Employee e;
public:
void getdata()
{
e.getdata();
cout<<"Enter company name and Department:"<<endl;
cin>>cname>>department;
}
void display()
{
e.display();
cout<<"Company name:"<<cname<<endl;
cout<<"Department:"<<department<<endl;
}
};
```

```
int main()
{
Company c;
c.getdata();
c.display();
return 0;
}
```

In above example class company contains the object of another class employee. As we know “company **has a** employee” sounds logical .so there exists a relationship between company and employee. Class company contains the object of class employee and members of class employee are used in class company which provides reusability.

Inheritance Program solution

1. WAP to enter information of n students and then display is using the concept of **multiple inheritance**.

```
#include <iostream>
#include <conio.h>
using namespace std;
class Student_info
{
char name[25];
int age;
public:
void get_info()
{
cout<<"Enter name and age of student"<<endl;
cin>>name>>age;
}
void disp_info()
{
cout<<"Name of student:"<<name<<endl;
cout<<"Age of student:"<<age<<endl;
}
};
class Faculty
{
char faculty[20];
public:
void get_faculty()
{
cout<<"Enter Faculty of student"<<endl;
cin>>faculty;
}
void disp_faculty()
{
cout<<"Faculty of student:"<<faculty<<endl;
}
};
```

```

class Attendance:public Faculty,public Student_info
{
int attendance;
public:
void get_attendance()
{
cout<<"Enter student's attendance percentage"<<endl;
cin>>attendance;
}

void disp_attendance()
{
cout<<"Student's attendance percentage ="<<attendance<<endl;
}
};

int main()
{
int i,n;
Attendance a[50];
cout<<"Enter Number of students"<<endl;
cin>>n;
for(i=0;i<n;i++)
{
cout<<"Enter information of student:"<<i+1<<endl;
a[i].get_info();
a[i].get_faculty();
a[i].get_attendance();
}
for(i=0;i<n;i++)
{
cout<<"Information of student:"<<i+1<<endl;
a[i].disp_info();
a[i].disp_faculty();
a[i].disp_attendance();
}
return 0;
}

```

2. WAP to enter information of n students and then display is using the concept of multilevel inheritance,[PU: 2015 fall]

```
#include <iostream>
#include <conio.h>
using namespace std;
class Student_info
{
char name[25];
int age;
public:
void get_info()
{
cout<<"Enter name and age of student"<<endl;
cin>>name>>age;
}
void disp_info()
{
cout<<"Name of student:"<<name<<endl;
cout<<"Age of student:"<<age<<endl;
}
};
class Faculty:public Student_info
{
char faculty[20];
public:
void get_faculty()
{
cout<<"Enter Faculty of student"<<endl;
cin>>faculty;
}
void disp_faculty()
{
cout<<"Faculty of student:"<<faculty<<endl;
}
};
class Attendance:public Faculty
{
int attendance;
public:
void get_attendance()
{
cout<<"Enter student's attendance percentage"<<endl;
cin>>attendance;
}
```

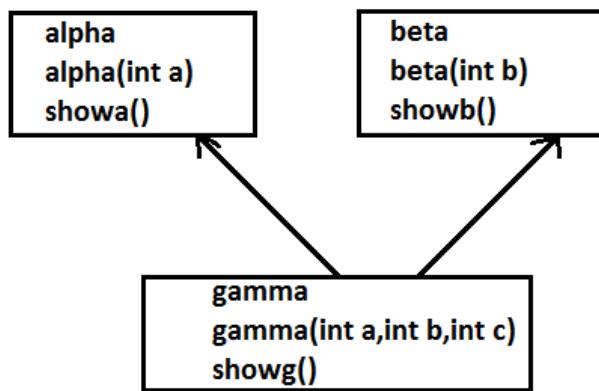
```

void disp_attendance()
{
cout<<"Student's attendance percentage ="<<attendance<<endl;
}
};

int main()
{
int i,n;
Attendance a[50];
cout<<"Enter Number of students"<<endl;
cin>>n;
for(i=0;i<n;i++)
{
cout<<"Enter information of student:"<<i+1<<endl;
a[i].get_info();
a[i].get_faculty();
a[i].get_attendance();
}
for(i=0;i<n;i++)
{
cout<<"Information of student:"<<i+1<<endl;
a[i].disp_info();
a[i].disp_faculty();
a[i].disp_attendance();
}
return 0;
}

```

3. Write a complete program with reference to the given figure.



```

#include<iostream>
#include<iostream>
using namespace std;
class alpha
{
int x;
public:
alpha(int a)
{
x=a;
}
void showa()
{
cout<<"x="<<x<<endl;
}
};
class beta
{
int y;
public:
beta(int b)
{
y=b;
}
void showb()
{
cout<<"y="<<y<<endl;
}
};
class gamma:public beta,public alpha
{
int z;
public:
gamma(int a,int b,int c):alpha(a),beta(b)
{
z=c;
}
void showg()
{
cout<<"z="<<z<<endl;
}
};

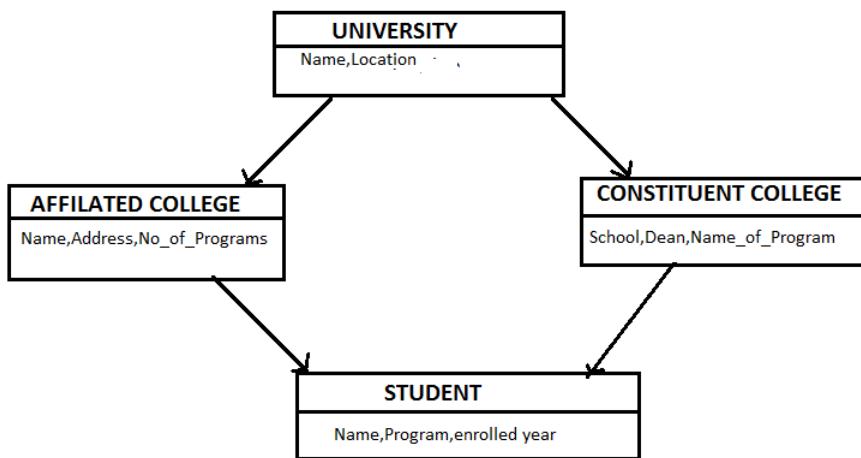
```

```

int main()
{
gamma g(5,10,15);
g.showa();
g.showb();
g.showg();
return 0;
}

```

4. The following figure shows the minimum information required for each class. Write a program by realizing the necessary member functions to read and display information of individual object. Every class should contain at least one constructor and should be inherited from other classes as well.[PU:2019 fall]



```

#include<iostream>
#include<string.h>
using namespace std;
class University
{
private:
char uname[20];
char location[20];
public:
University(char un[],char loc[])
{
strcpy(uname,un);
strcpy(location,loc);
}

```

```

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

class Affiliated_College:virtual public University
{
private:
char cname[20];
char address[20];
int no_of_programs;
public:
Affiliated_College(char un[],char loc[],char cn[],char addr[],int nop):University(un,loc)
{
strcpy(cname,cn);
strcpy(address,addr);
no_of_programs=nop;
}
void display()
{
cout<<"Affiliated College Name=" <<cname << endl;
cout<<"Address=" <<address << endl;
cout<<"Number of programs=" <<no_of_programs << endl;
}
};

class Constituent_College:virtual public University
{
private:
char school[20];
char dean[20];
char name_of_program[20];
public:
Constituent_College(char un[],char loc[],char sn[],char d[],char npro[]):University(un,loc)
{
strcpy(school,sn);
strcpy(dean,d);
strcpy(name_of_program,npro);
}
void display()
{
cout<<"School name=" <<school << endl;
cout<<"Dean name=" <<dean << endl;
cout<<"Name of Program=" <<name_of_program << endl;}}

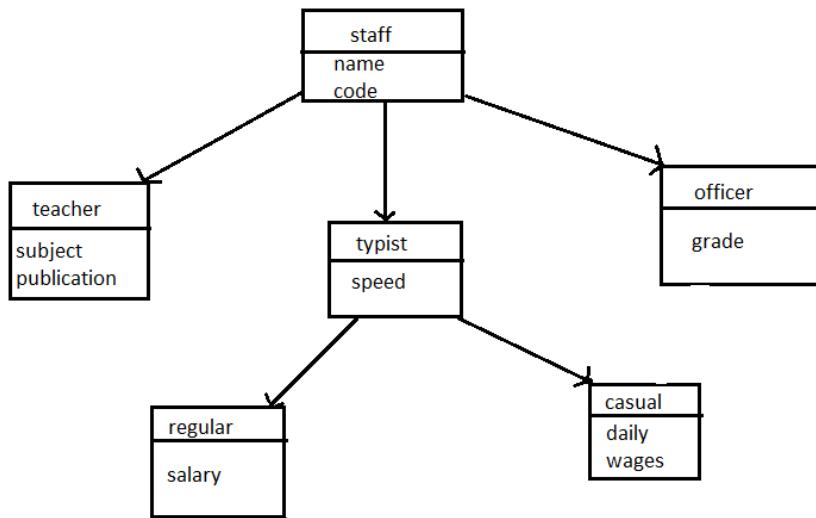
```

```

class Student:public Affiliated_College,public Constituent_College
{
private:
char student_name[20];
char program[20];
int enrolled_year;
public:
Student(char un[],char loc[],char cn[],char addr[],int nop, char sn[],char d[],char npro[],char stn[],char pro[],int
year):Affiliated_College(un,loc,cn,addr,nop),Constituent_College(un,loc,sn,d,npro),University(un,loc)
{
strcpy(student_name,stn);
strcpy(program,pro);
enrolled_year=year;
}
void display()
{
cout<<"Student name=" <<student_name << endl;
cout<<"Program=" <<program << endl;
cout<<"Enrolled year=" <<enrolled_year << endl;
}
};
int main()
{
Student
st("Pokhara","Dhungepatan","EEC","Sanepa",3,"Engineering","Bharat","Civil","Pradip","Civil",2018);
st.University::display();
st.Affiliated_College::display();
st.Constituent_College::display();
st.display();
return 0;
}

```

5. An educational institution wishes to maintain a database of its employees. The database is divided into a number of classes whose hierarchical relationship are shown below. The figure also shows minimum information required for each class. Specify all the classes and define functions to create database and retrieve individual information when required.



```

#include<iostream>
using namespace std;
class Staff
{
protected:
char name[20];
int code;
public:
void get_staff()
{
cout<<"Enter Name and code of staff" << endl;
cin>>name>>code;
}
void disp_staff()
{
cout<<"Name=" << name << endl;
cout<<"Code=" << code << endl;
}
};
  
```

```

class Teacher:public Staff
{
char subject[20],publication[20];
public:
void get_teacher()
{
cout<<"Enter Subject and Publication"<<endl ;
cin>>subject>>publication;
}
void disp_teacher()
{
cout<<"Subject="<<subject<<endl;
cout<<"Publication="<<publication<<endl;
}
};

class Officer:public Staff
{
char grade;
public:
void get_officer()
{
cout<<"Enter the grade"<<endl;
cin>>grade;
}
void disp_officer()
{
cout<<"Grade="<<grade<<endl;
}
};

class Typist:public Staff
{
protected:
int speed;
public:
void get_typist()
{
cout<<"Enter Speed"<<endl;
cin>>speed;
}
void disp_typist()
{
cout<<"Speed="<<speed<<endl;
}
};

```

```

class Regular:public Typist
{
int salary;
public:
void get_regular()
{
cout<<"Enter Salary"<<endl;
cin>>salary;
}
void disp_regular()
{
cout<<"Salary="<<salary<<endl;
}
};

class Casual:public Typist
{
int wages;
public:
void get_casual()
{
cout<<"Enter Daily wages"<<endl;
cin>>wages;
}
void disp_casual()
{
cout<<"Daily wages="<<wages<<endl;
}
};

int main()
{
Teacher t;
Officer o;
Regular r;
Casual c;

t.get_staff();
t.get_teacher();
t.disp_staff();
t.disp_teacher();

o.get_staff();
o.get_officer();
o.disp_staff();
o.disp_officer();

```

```

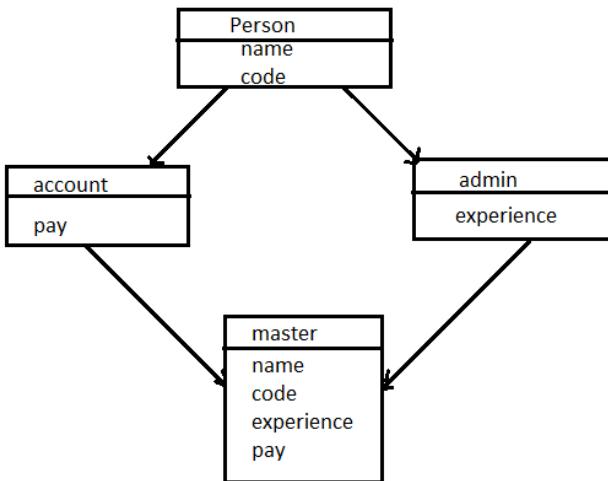
r.get_staff();
r.get_typist();
r.get_regular();
r.disp_staff();
r.disp_typist();
r.disp_regular();

c.get_staff();
c.get_typist();
c.get_casual();
c.disp_staff();
c.disp_typist();
c.disp_casual();
return 0;
}

```

6. The following figure shows minimum information required for each class.

Write a Program to realize the above program with necessary member functions to create the database and retrieve individual information



```

#include<iostream>
using namespace std;
class Person
{
protected:
    char name[20];
    int code;
};

```

```

class Account:virtual public Person
{
    protected:
        float pay;
};

class Admin:virtual public Person
{
    protected:
        int experience;
};

class Master:public Account,public Admin
{
public:
void getinfo()
{
    cout<<"Enter name of person"<<endl;
    cin>>name;
    cout<<"Enter code"<<endl;
    cin>>code;
    cout<<"Enter pay for person"<<endl;
    cin>>pay;
    cout<<"Enter experience"<<endl;
    cin>>experience;
}
void display()
{
    cout<<"Name:"<<name<<endl;
    cout<<"Code:"<<code<<endl;
    cout<<"Pay:"<<pay<<endl;
    cout<<"Experience:"<<experience<<"years"<<endl;
}
};

int main()
{
Master m;
m.getinfo();
m.display();
return 0;
}

```

7. Rewrite the above program using constructor on each class to initialize the data members.

```

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

```

```

class Person
{
    protected:
    char name[20];
    int code;
    public:
    Person(char n[],int c)
    {
        strcpy(name,n);
        code=c;
    }
};

class Account:virtual public Person
{
    protected:
    float pay;
    public:
    Account(char n[],int c,float p):Person(n,c)
    {
        pay=p;
    }
};

class Admin:virtual public Person
{
    protected:
    int experience;
    Admin(char n[],int c,int e):Person(n,c)
    {
        experience=e;
    }
};

class Master:public Account,public Admin
{
public:
Master(char n[],int c,float p,int e):Admin(n,c,e),Account(n,c,p),Person(n,c)
{
    strcpy(name,n);
    code=c;
    pay=p;
    experience=e;
}
}

```

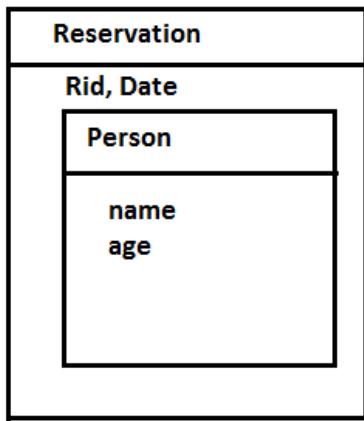
```

void display()
{
    cout<<"Name:"<<name<<endl;
    cout<<"Code:"<<code<<endl;
    cout<<"Pay:"<<pay<<endl;
    cout<<"Experience:"<<experience<<"years"<<endl;
}
};

int main()
{
Master m("Pradip",121,23000.56,2);
m.display();
return 0;
}

```

8. Write a program that allow you to book a ticket for person and use two classes PERSON, RESERVATION. Class RESERVATION is composite class/ container class.



```

#include<iostream>
using namespace std;
class Person
{
char name[20];
int age;
public:
void getdata()
{
cout<<"Enter name and age of person"<<endl;
cin>>name>>age;
}

```

```

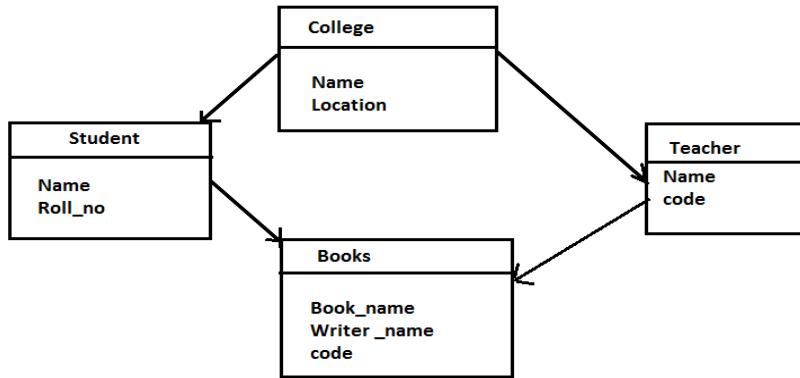
void display()
{
cout<<"Name="<<name<<endl;
cout<<"Age="<<age<<endl;
}
};

class Reservation
{
int rid;
int year,month,date;
Person p;
public:
void getdata()
{
p.getdata();
cout<<"Enter Reservation ID"<<endl;
cin>>rid;
cout<<"Date in terms of YY-MM-DD"<<endl;
cin>>year>>month>>date;
}
void display()
{
p.display();
cout<<"Reservation ID="<<rid<<endl;
cout<<"Reservation date="<<year<<"-"<<month<<"-"<<date<<endl;
}
};

int main()
{
Reservation r;
r.getdata();
r.display();
return 0;
}

```

- 9. The following figure shows the minimum information required for each class. Write a program to realize the above program with necessary member functions to create the database and retrieve individual information .Every class should contain at least one constructor and should be inherited to other classes as well.[PU:2010 spring][PU 2009 fall]**



```

#include<iostream>
#include<string.h>
using namespace std;
class College
{
private:
char cname[20];
char location[20];
public:
College(char cn[],char loc[])
{
strcpy(cname,cn);
strcpy(location,loc);
}
void display()
{
cout<<"College name="<<cname<<endl;
cout<<"College location="<<location<<endl;
}
};

class Student:virtual public College
{
private:
char sname[20];
int roll;
public:
Student(char cn[],char loc[],char sn[],int r):College(cn,loc)
{
strcpy(sname,sn);
roll=r;
}
}

```

```

void display()
{
cout<<"Student name"<<sname<<endl;
cout<<"Roll no"<<roll<<endl;
}
};

class Teacher:virtual public College
{
private:
char tname[20];
int tcode;
public:
Teacher(char cn[],char loc[],char tn[],int tc):College(cn,loc)
{
strcpy(tname,tn);
tcode=tc;
}
void display()
{
cout<<"Teacher name"<<tname<<endl;
cout<<"Teacher Code"<<tcode<<endl;
}
};

class Books:public Student,public Teacher
{
private:
char book_name[20];
char writer_name[20];
int book_code;
public:
Books(char cn[],char loc[],char sn[],int r, char tn[],int tc,char bn[],char wn[],int
bc):Teacher(cn,loc,tn,tc),Student(cn,loc,sn,r),College(cn,loc)
{
strcpy(book_name,bn);
strcpy(writer_name,wn);
book_code=bc;
}
void display()
{
cout<<"Book name"<<book_name<<endl;
cout<<"Writer name"<<writer_name<<endl;
cout<<"Book code"<<book_code<<endl;
}
};

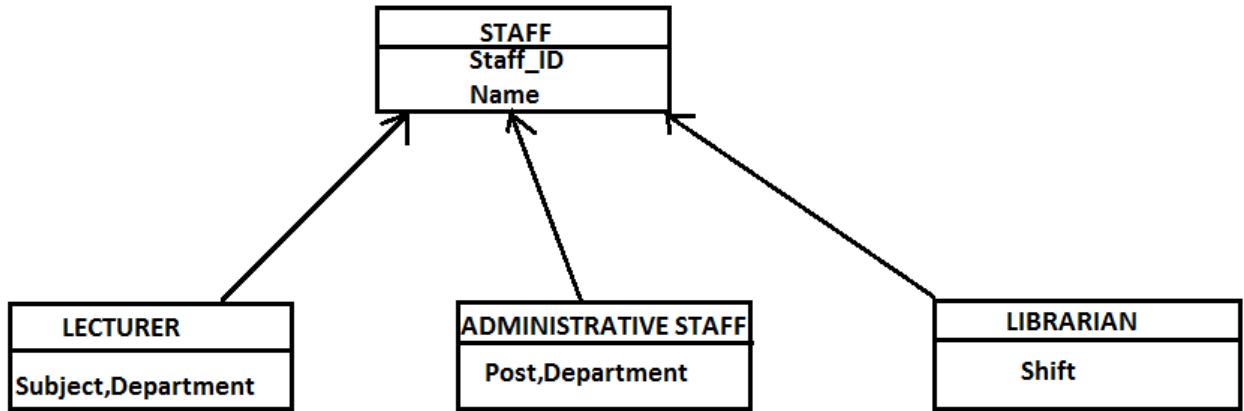
```

```

int main()
{
Books b("EEC","sanepa","Ram",47,"Pradip",151,"OOPs","Timothy Budd",53421);
b.College::display();
b.Student::display();
b.Teacher::display();
b.display();
return 0;
}

```

- 10. Develop a complete program for an institution, which wishes to maintain a database of its staff. The database is divided into number of classes whose hierarchical relationship is shown in the following diagram. specify all classes and define constructors and functions to create database and retrieve the individual information as per requirements.**



```

#include<iostream>
#include<string.h>
using namespace std;
class STAFF
{
private:
    int staff_id;
    char name[20];
public:
    STAFF(int sid,char sn[])
    {
        staff_id=sid;
        strcpy(name,sn);
    }
}

```

```

void display()
{
    cout<<"Staff ID:"<<staff_id<<endl;
    cout<<"Name:"<<name<<endl;
}
};

class LECTURER:public STAFF
{
char subject[20];
char department[20];
public:
    LECTURER(int sid,char sn[],char sub[],char dept[]):STAFF(sid,sn)
    {
        strcpy(subject,sub);
        strcpy(department,dept);
    }
    void display()
    {
        cout<<"Subject:"<<subject<<endl;
        cout<<"Department:"<<department<<endl;
    }
};

class ADMINISTRATIVE_STAFF:public STAFF
{
char post[20];
char department[20];
public:
ADMINISTRATIVE_STAFF(int sid,char sn[],char p[],char dept[]):STAFF(sid,sn)
    {
        strcpy(post,p);
        strcpy(department,dept);
    }
    void display()
    {
        cout<<"Post:"<<post<<endl;
        cout<<"Department:"<<department<<endl;
    }
};

```

```

class LIBRARIAN:public STAFF
{
char shift[20];
public:
    LIBRARIAN(int sid,char sn[],char s[]):STAFF(sid,sn)
    {
        strcpy(shift,s);
    }
    void display()
    {
        cout<<"Shift:"<<shift<<endl;
    }
};

int main()
{
LECTURER l(101,"Pradip Paudel","C programming","Computer");
l.STAFF::display();
l.display();
ADMINISTRATIVE_STAFF a(212,"Ekraj Acharaya","Admin","Administration");
a.STAFF::display();
a.display();
LIBRARIAN lib(314,"Sita sharma","Morning");
lib.STAFF::display();
lib.display();
return 0;
}

```

- 11. Develop a complete program for an institution which wishes to maintain a database of its staff. Declare a base class STAFF which include staff_id and name. Now develop a records for the following staffs with the given information below.**
- Lecturer(subject,department)**
- Administrative staff (Post,department)**

```

#include<iostream>
#include<string.h>
using namespace std;
class STAFF
{
    private:
        int staff_id;
        char name[20];
    public:
        void get_staff()
        {
            cout<<"Enter staff id"<<endl;
            cin>>staff_id;
            cout<<"Enter name"<<endl;
            cin>>name;
        }
        void display_staff()
        {
            cout<<"Staff ID:"<<staff_id<<endl;
            cout<<"Name:"<<name<<endl;
        }
};

class LECTURER:public STAFF
{
    char subject[20];
    char department[20];
public:
    void get_lecturer()
    {
        cout<<"Enter subject"<<endl;
        cin>>subject;
        cout<<"Enter department"<<endl;
        cin>>department;
    }
    void display_lecturer()
    {
        cout<<"Subject:"<<subject<<endl;
        cout<<"Department:"<<department<<endl;
    }
};

```

```

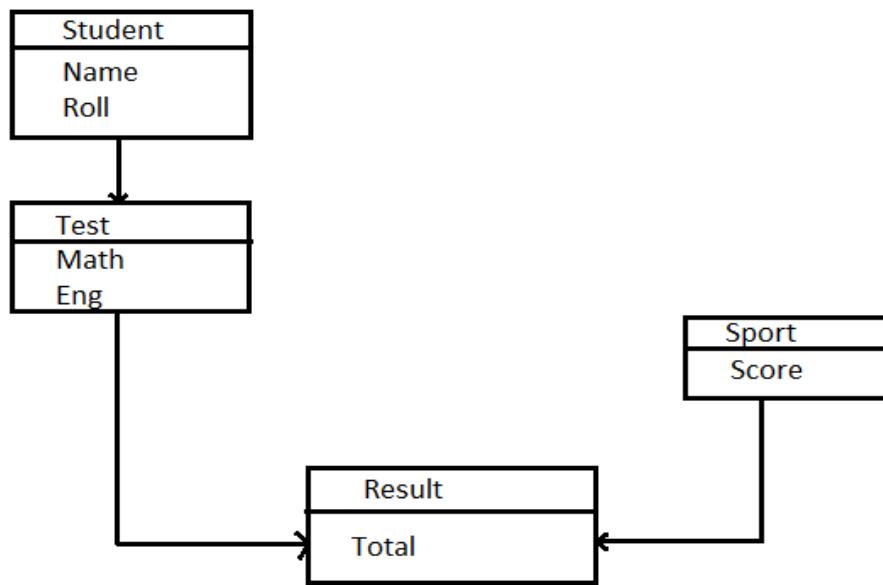
class ADMINISTRATIVE_STAFF:public STAFF
{
char post[20];
char department[20];
public:

    void get_administrative()
    {
        cout<<"Enter post"<<endl;
        cin>>post;
        cout<<"Enter department"<<endl;
        cin>>department;
    }
    void display_administrative()
    {
        cout<<"Post:"<<post<<endl;
        cout<<"Department:"<<department<<endl;
    }
};

int main()
{
LECTURER l;
l.get_staff();
l.get_lecturer();
l.display_staff();
l.display_lecturer();
ADMINISTRATIVE_STAFF a;
a.get_staff();
a.get_administrative();
a.display_staff();
a.display_administrative();
return 0;
}

```

12. Implement the below given in class diagram in C++. Assume necessary function yourself.



```
#include<iostream>
using namespace std;
class Student
{
private:
char name[20];
int roll;
public:
void get_student()
{
    cout<<"Enter name and roll no"<<endl;
    cin>>name>>roll;
}
void disp_student()
{
    cout<<"Name="<<name<<endl;
    cout<<"Rollno="<<roll<<endl;
}
```

```

class Test:public Student
{
protected:
float math,eng;
public:
void get_test()
{
    cout<<"Enter marks in math and english" << endl;
    cin>>math>>eng;
}
void disp_test()
{
    cout<<"marks in math=" << math << endl;
    cout<<"marks in english=" << eng << endl;
}
};

class Sport
{
protected:
float score;
public:
void get_sport()
{
    cout<<"Enter score" << endl;
    cin>>score;
}
void disp_sport()
{
    cout<<"Score=" << score << endl;
}
};

class Result:public Test,public Sport
{
private:
float total;
public:
void disp_total()
{
total=eng+math+score;
cout<<"Total=" << total << endl;
}
};

```

```
int main()
{
Result r;
r.get_student();
r.get_test();
r.get_sport();
r.disp_student();
r.disp_test();
r.disp_sport();
r.disp_total();
return 0;
}
```