Workbook

Object Oriented Concepts & Programming (SE – 201)

Name	
Roll No	
Batch	
Year	
Department	

Department of Computer Science & Software Engineering
NED University of Engineering & Technology

Workbook

Object Oriented Concepts & Programming (SE – 201)

Prepared by

Mrs. Nazish Irfan
I.T Manager, CS&SE

Approved by

Chairman
Department of Computer Science & Software Engineering

Department of Computer Science & Software Engineering
NED University of Engineering & Technology

SE-201: Object Oriented Concept Programming
Table of Contents

Department of Computer Science & Software Engineering
NED University of Engineering & Technology

Lab # 1

Object:

Introduction to Object Oriented Programming. Coding, compiling and debugging a simple C++ program.

.

Theory:

Introduction.

The fundamental idea behind object-oriented languages is to combine into a single unit both *data* and the *functions that operate on that data*. Such a unit is called an *object.* An object's functions, called *member functions* in C++, typically provide the only way to access its data.

To read a data item in an object, member function in the object is called, which in turn will access the data and return the value. The data cannot be accessed directly. The data is *hidden*, so it is safe from accidental alteration. *Data encapsulation* and *data hiding* are key terms in object-oriented languages. This simplifies writing, debugging, and maintaining the program.

A C++ program typically consists of a number of objects, which communicate with each other by calling one another's member functions. Member functions are also called *methods* in some other object-oriented (OO) languages. Also, data items are referred to as *attributes* or *instance variables.*

Characteristics of Object Oriented Languages.

The major elements of object-oriented languages in general, and C++ in particular are:

- Objects
- Classes
- Inheritance.
- Reusability
- Polymorphism and Overloading.

C++ and C.

C++ is derived from the C language. Almost every correct statement in C is also a correct statement in C++, although the reverse is not true. The most important elements added to C to create C++ are concerned with classes, objects and Object-Oriented Programming. However, C++ has many other new features as well, including an improved approach to input/output (I/O) and a new way to write comments.

A Basic C++ Program.

```
#include <iostream>
using namespace std;
void main()
{
   char course_code[10];
   cout<<"Welcome to OOP for Engineers"<<endl;
   cout<<"Input the course code for this lab:"<<endl;
   cin>>course_code;
   cout<<"The course code for this lab is:"<<course_code<<endl;
}
```

Exercise:

1. Write a program that displays the squares, cubes and fourth powers of the numbers 1 through 5. Save the file in your directory as question1.cpp

2. Find errors in these lines of codes. Locate them and give the corrections.

```
#include <iostream>
void main()
{
int number1=0,number2=8, value(0);
value=number1+number2
cout>>number1<<"+"<<number2<<"="<<value<<endl;
}
```

Lab # 2

Object:

Introduction to Unified Modeling Language (UML).

.

Theory:

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.

The UML is a very important part of developing object oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

Types of UML Diagrams

Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction.

UML diagrams commonly created in visual modeling tools include:

- **Use Case Diagram** displays the relationship among actors and use cases.
- **Class Diagram** models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.
- **Interaction Diagrams**
    - **Sequence Diagram** displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).
    - **Collaboration Diagram** displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.
- **State Diagram** displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions.
- **Activity Diagram** displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing.
- **Physical Diagrams**.
    - **Component Diagram** displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time.
    - **Deployment Diagram** displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.

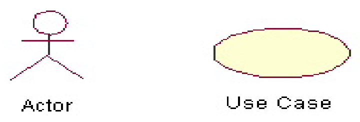
Exercise:

1. List the advantages of UML.

Lab # 3

Object:
To understand different types of UML Diagrams.

Theory:
Use Case Diagram is a set of scenarios that describing an interaction between a user and a system. A use case diagram displays the relationship among actors and use cases. The two main components of a use case diagram are use cases and actors.

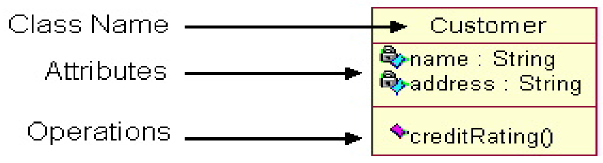


An actor is represents a user or another system that will interact with the system you are modeling. A use case is an external view of the system that represents some action the user might perform in order to complete a task.

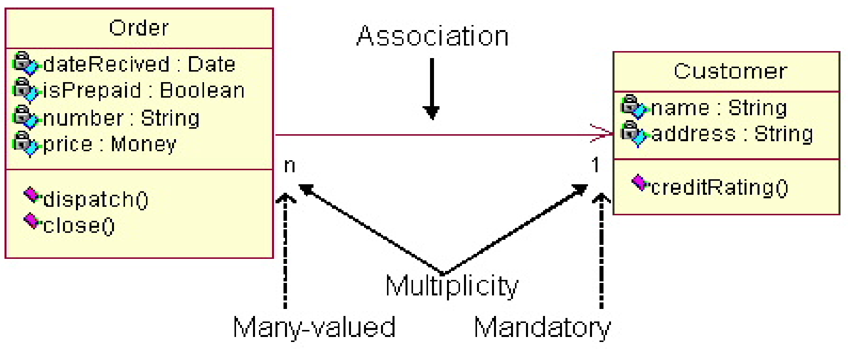
When to Use: Use Cases Diagrams.
Use cases are used in almost every project. They are helpful in exposing requirements and planning the project. During the initial stage of a project most use cases should be defined, but as the project continues more might become visible.

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Classes are composed of three things: a name, attributes, and operations. Below is an example of a class

\

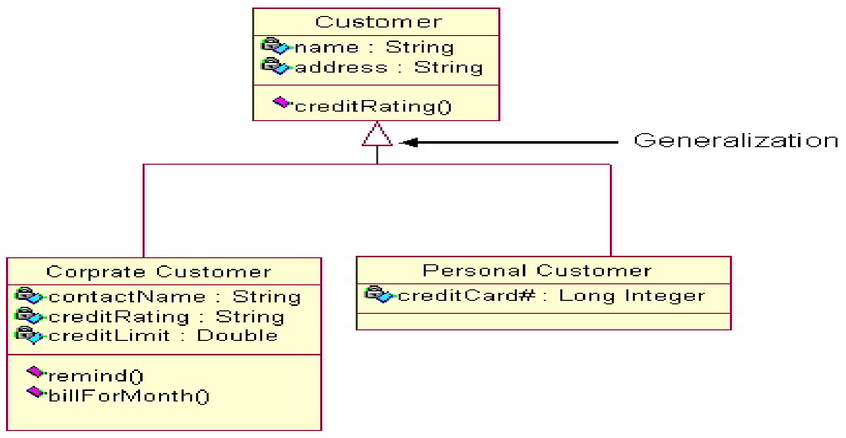


Class diagrams also display relationships such as containment, inheritance, associations and others. The association shows the relationship between instances of classes.



\

A generalization is used when two classes are similar, but have some differences.

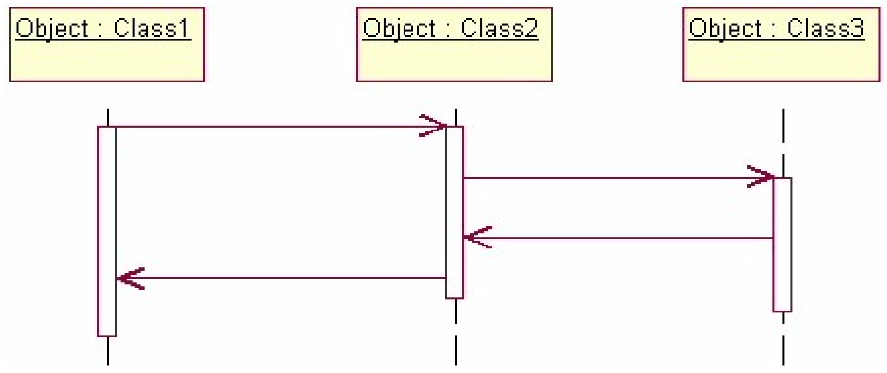


When to Use: Class Diagrams.
Class diagrams are used in nearly all Object Oriented software designs. Use them to describe the Classes of the system and their relationships to each other.

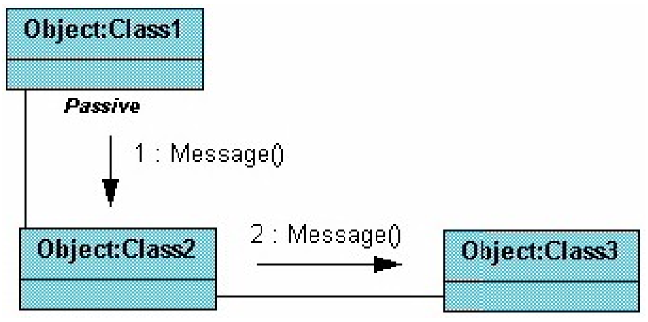
Interaction Diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task. The two kinds of interaction diagrams are **sequence** and **collaboration** diagrams. This example is only meant as an introduction to the UML and interaction diagrams.

When to Use: Interaction Diagrams.
Interaction diagrams are used when you want to model the behavior of several objects in a use case. They demonstrate how the objects collaborate for the behavior. Interaction diagrams do not give in depth representation of the behavior.

Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass. The diagrams are read left to right and descending. The example below shows an object of class 1 start the behavior by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message.

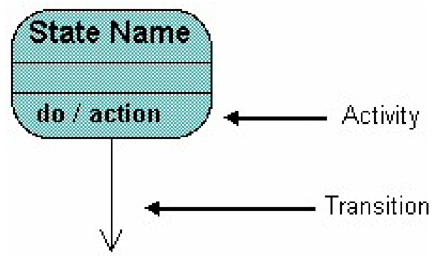


Collaboration diagrams are also relatively easy to draw. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects. There are many acceptable sequence numbering schemes in UML. A simple 1, 2, 3... format can be used, or for more detailed and complex diagrams a 1, 1.1 ,1.2, 1.2.1... scheme can be used.



State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.

When to Use: State Diagrams.
Use state diagrams to demonstrate the behavior of an object through many use cases of the system. Only use state diagrams for classes where it is necessary to understand the behavior of the object through the entire system. Not all classes will require a state diagram and state diagrams are not useful for describing the collaboration of all objects in a use case.
State diagrams are other combined with other diagrams such as interaction diagrams and activity diagrams.



All state diagrams being with an initial state of the object. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.

Activity diagrams describe the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed. Activity diagrams can show activities that are conditional or parallel.

When to Use: Activity Diagrams.
Activity diagrams should be used in conjunction with other modeling techniques such as interaction diagrams and state diagrams. The main reason to use activity diagrams is to model the workflow behind the system being designed. Activity Diagrams are also useful for:

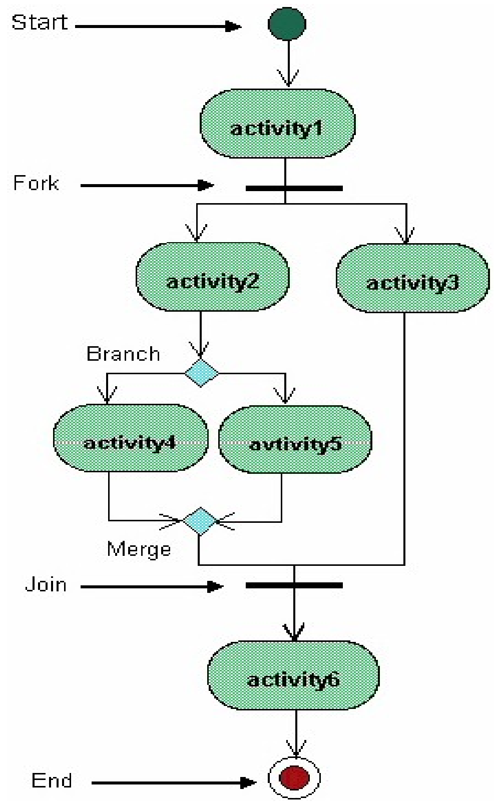
- analyzing a use case by describing what actions need to take place and when they should occur;
- describing a complicated sequential algorithm;
- modeling applications with parallel processes.

However, activity diagrams should not take the place of interaction diagrams and state diagrams. Activity diagrams do not give detail about how objects behave or how objects collaborate.

How to Draw: Activity Diagrams.
Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities.

- A **fork** is used when multiple activities are occurring at the same time.
- The **branch** describes what activities will take place based on a set of conditions.
- All branches at some point are followed by a merge to indicate the end of the conditional behavior started by that branch.
- After the merge all of the parallel activities must be combined by a join before transitioning into the final activity state.

The diagram below shows a fork after activity1. This indicates that both activity2 and activity3 are occurring at the same time. After activity2 there is a branch.

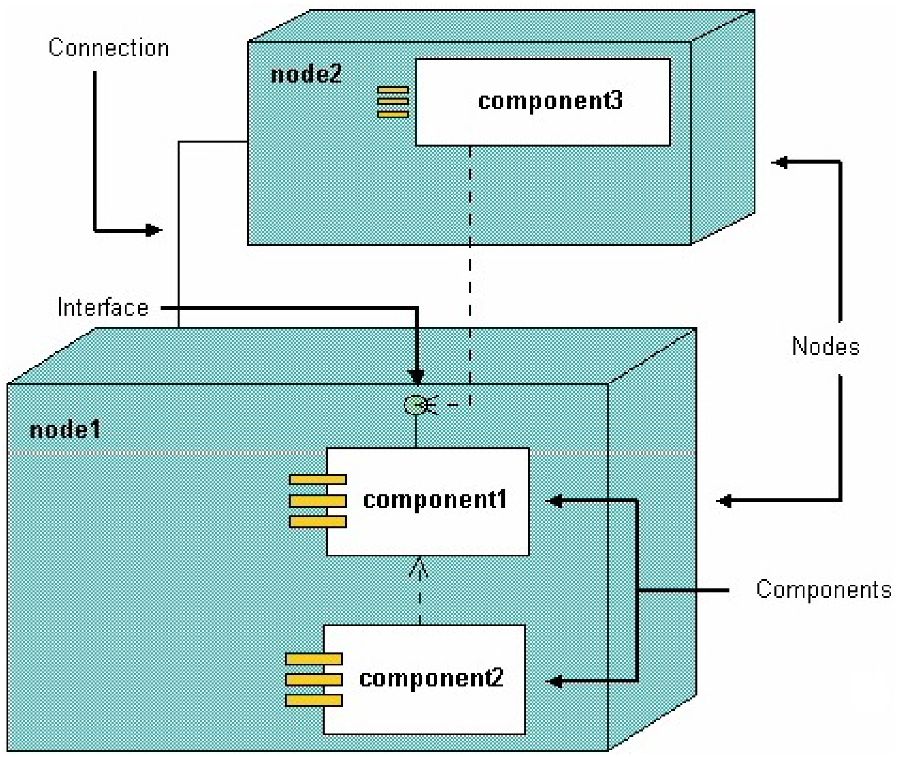


Physical Diagrams are of two types: **deployment diagrams** and **component diagrams.** Deployment diagrams show the physical relationship between hardware and software in a system. Component diagrams show the software components of a system and how they are related to each other. These relationships are called dependencies.

When to Use: Physical Diagrams.
Physical diagrams are used when development of the system is complete. Physical diagrams are used to give descriptions of the physical information about a system.

Many times the deployment and component diagrams are combined into one physical diagram. A combined deployment and component diagram combines the features of both diagrams into one diagram. The deployment diagram contains nodes and connections. A node usually represents a piece of hardware in the system. A connection depicts the communication path used by the hardware to communicate and usually indicates a method such as TCP/IP.

The component diagram contains components and dependencies. Components represent the physical packaging of a module of code. The dependencies between the components show how changes made to one component may affect the other components in the system. Dependencies in a component diagram are represented by a dashed line between two or more components. Component diagrams can also show the interfaces used by the components to communicate to each other. The combined deployment and component diagram below gives a high level physical description of the completed system.



Exercise:

1. Draw all the UML diagrams for an ATM Machine.

Lab # 4

Object:

To understand the concepts of classes and to build classes and create object.

Theory:

A class can be viewed as a customized 'struct' that **encapsulates data** and **function**. Format of a class definition:

```
        class your_class_name
        {       member_access_specifier:
                        data members;
                member_access_specifier:
                        member_functions();
        };
```

When cin and cout are used to perform I/O, actually objects are created from istream and ostream respectively that has been defined in iostream header file.

Building a Class

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

class smallobj                          //declare a class
{
private:
        int somedata;                   //class data
public:
        void setdata(int d)             //member function to set data
                { somedata = d; }
        void showdata()                 //member function to display data
                {
                cout << "Data is " << somedata << endl; }
};

void main()
{
smallobj s1, s2;                //define two objects of class smallobj
s1.setdata(1066);               //call member function to set data
s2.setdata(1776);
s1.showdata();                  //call member function to display data
s2.showdata();
}
```

Exercise:

1. Create a class DM which stores and displays the values of distances. DM stores distances in meters and centimeters. The value of distance should be entered by the user.

Lab # 5

Object:
To understand the concept of Constructors and Destructors.

Theory:

a) Constructor is a member function that is automatically called when we create an object, while destructor is automatically called when the object is out of scope.

b) Constructor is used to initialize the data members according to the desired value.

c) In a class definition, if both constructor and destructor are not provided, the compiler **will automatically provide default constructor** for you. Hence during the instantiation of the object, the data member will be initialized **to any value**.

d) There are 3 types of constructor:
   a. Default constructor
      - This type of constructor does not have any arguments inside its parenthesis.
      - Is called when creating objects without passing any arguments.

   b. Constructor
      - This type of constructor accepts arguments inside its parenthesis
      - Is called when creating objects without passing any arguments

   c. Copy constructor
      - Passing object as an argument to a function.
      - Return object from a function.
      - Initializing an object with another object in a declaration statement

e) By providing many definitions for constructor, you are actually implementing '**function overloading**'; i.e. functions of same name but different parameters (not return type) and function definition.

f) Another function is called automatically when an object is destroyed. Such a function is called a *destructor*.

g) A destructor has the same name as the constructor but is preceded by a tilde.

h) Destructors do not have a return value. They also take no arguments.

i) The most common use is to deallocate memory that was allocated for the object by the constructor.

Constructor.

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

```
class Counter
{
private:
        int count; //count
public:
        Counter() : count(0)                    //constructor
                { /*empty body*/ }

        void inc_count()                        //increment count
                { count++; }
        int get_count()                         //return count
                { return count; }
};

void main()
{
Counter c1, c2;                                 //define and initialize
cout << "\nc1=" << c1.get_count();              //display
cout << "\nc2=" << c2.get_count();
c1.inc_count();                                 //increment c1
c2.inc_count();                                 //increment c2
c2.inc_count();                                 //increment c2
cout << "\nc1=" << c1.get_count();              //display again
cout << "\nc2=" << c2.get_count();
cout << endl;
}
```

Copy Constructor.

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

class Distance
{
private:
        int feet;
        float inches;
public:
        Distance() : feet(0), inches(0.0)       //constructor (no args)
                { }
        Distance(int ft, float in) : feet(ft), inches(in)  //constructor (two args)
                { }
        void getdist()                          //get length from user
                {
                cout << "\nEnter feet: "; cin >> feet;
                cout << "Enter inches: "; cin >> inches;
                }
        void showdist()                         //display distance
                {
                cout << feet << "\'–" << inches << '\"'; }
};

void main()
{
Distance dist1(11, 6.25);                    //two–arg constructor
Distance dist2(dist1);                       //one–arg constructor
Distance dist3 = dist1;                      //also one–arg constructor

cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
}
```

Destructor.

```
class Foo
{
private:
int data;
public:
Foo() : data(0)                          //constructor (same name as class)
{ }
~Foo()                                   //destructor (same name with tilde)
{ }
};
```

Exercise:

1. Create a class tollbooth. The two data items are a type int to hold the total number of cars and a type double to hold the total amount of money collected. A constructor initializes both these to 0. A member function called payingCar( ) increments the car total and adds 0.50 to the cash total. Another member function displays the two totals.

Lab # 6

Object:
To understand the fundamentals of operator and function overloading.

Theory:
Operator overloading is needed to make operation with user defined data type, i.e., classes, can be performed concisely. If you do not provide the mechanism of how these operators are going to perform with the objects of our class, you will get this error message again and again

In C++, an operator is *just* one form of function with a special naming convention. As such, it can have return value as well as having some arguments or parameters. Recall the general format of a function prototype:

```
return_type function_name(type_arg1, type_arg2, …);
```

An operator function definition is done similarly, with an exception that an operator's function name must take the form of **operatorX**, where **X** is the symbol for an operator, e.g. +,-, /, etc. For an instance, to declare **multiplication operator ***, the function prototype would look like

```
return_type operator*( type arg1, type arg2)
```

Most of us have been using the operators such as +, -, *, & etc. and pay little attention about what actually is happening. We can easily add an integer with another integer (int + int), or add an integer with a floating-type number (int + float) using the very same operator symbol + without questioning anything. How do they work? Hence, what we can say here is that, operator + has been **overloaded** to perform on various types of built-in data types. We just use them all these while without having to think of the details of how it works with different built-in data types.

Limitations on operator overloading.
Although C++ allows us to overload operators, it also imposes restrictions to ensure that operator overloading serves its purpose to enhance the programming language itself, without compromising the existing entity. The followings are the restrictions:

- Cannot change the original behavior of the operator with built in data types.
- Cannot create new operator
- Operators =, [], () and -> can only be defined as members of a class and not as global functions
- The arity or number of operands for an operator may not be changed. For example, addition, +, may not be defined to take other than two arguments regardless of data type.
- The precedence of the operator is preserved, i.e., does not change, with overloading

- Operators that can be overloaded:

```
+     -     *     /     %     ^     &     |     ~     !
=     <     >     +=    -=    *=    /=    %=    ^=
|=
<<    >>    >>=   <<=   ==    !=    <=    >=    &&
||
++    --    ->*   ,     ->    []    ()    new   delete
new[] delete[]
```

- Operators that cannot be overloaded:

```
.     .*    ::    :?    sizeof
```

The (=) assignment operator and the (&) address operator are not needed to be overloaded since both can work automatically with whatever data types. (=) assignment operator creates a bit-by-bit copy of an object while the (&) address operator returns a memory address of the object. The exception comes when we deal with classes containing pointers as members. In this case, the assignment operator needs to be overloaded explicitly

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

class Distance
{
private:
        int feet;
        float inches;
public:                                                     //constructor (no args)
        Distance() : feet(0), inches(0.0)
                { } //constructor (two args)
        Distance(int ft, float in) : feet(ft), inches(in)
                { }
        void getdist() //get length from user
                {
                cout << "\nEnter feet: "; cin >> feet;
                cout << "Enter inches: "; cin >> inches;
                }
        void showdist() const //display distance
                { cout << feet << "\'-" << inches << '\"'; }

        Distance operator + ( Distance ) const;             //add 2 distances
};
//add this distance to d2
Distance Distance::operator + (Distance d2) const          //return sum
        {
        int f = feet + d2.feet;                             //add the feet
        float i = inches + d2.inches;                       //add the inches
        if(i >= 12.0)                                       //if total exceeds 12.0,
           {                                                //then decrease inches
                i -= 12.0;                                  //by 12.0 and
                f++;                                        //increase feet by 1
           }                                        //return a temporary Distance
        return Distance(f,i);                               //initialized to sum
}

void main()
{
Distance dist1, dist3, dist4;                               //define distances
dist1.getdist();                                            //get dist1 from user
Distance dist2(11, 6.25);                                   //define, initialize dist2
dist3 = dist1 + dist2;                                      //single '+' operator
dist4 = dist1 + dist2 + dist3;                              //multiple '+' operators

cout << "dist1 = "; dist1.showdist(); cout << endl;
cout << "dist2 = "; dist2.showdist(); cout << endl;
cout << "dist3 = "; dist3.showdist(); cout << endl;
cout << "dist4 = "; dist4.showdist(); cout << endl;
return 0;
}
```

Function overloading.
Function overloading means having more than one function with exactly the same function name and each function has different parameters and different return type. For an instance,

```
        int square ( int );
        double square ( double);
```

are some examples of **function overloading**. We have two functions with the same function name, square and different type of argument, i.e. int and double. Try the following example:

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

int square(int s)
{ return s*s; }

double square(double s)
{ return s*s; }
```

```
void main()
{
  cout<<"Calling square function with INTEGER argument"<<endl;
  cout<<"Square of 5 = "<<square(5)<<endl<<endl;
  cout<<"Calling square function with DOUBLE argument"<<endl;
  cout<<"Square of 6.2 = "<<square(6.2)<<endl<<endl;
 }
```

Exercise:

1. Write a program to concatenate two strings by overloading the + operator.

Lab # 7

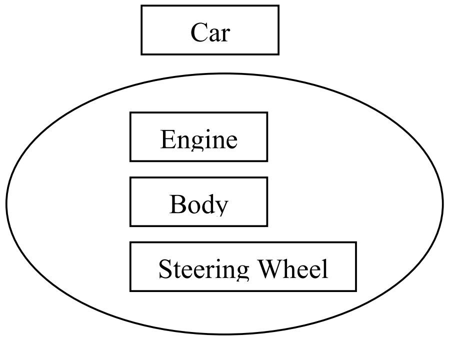
Object:
To understand the fundamentals of inheritance as opposed to composition.

Theory:
INHERITANCE VS COMPOSITION.
Initially, OOP is proposed as a means to increase the efficiency of software developments. One of its benefits is promoting software/program reuse. Once a class is developed reliably, i.e., after went through all sorts of possible verification processes, it can be used everywhere. It can be used right away, without needing to "reinvent the wheel".

There are two forms of software reuse in OOP. The first one is composition which is a "**has a(n)**" relationship. For instance, we consider a car. A car consists of engine, tires, body, steering wheel, etc. So, we can have

A car has an engine.
A car has a steering wheel.
A car has a body.

Note that in the above relationship, the enclosing entity is the car object. It encloses engine, steering wheel and body objects. It is clear from logical relationship that a car **is not** an engine or a steering wheel. Engine, Body and Steering wheel are part of class Car.



On the other hand, inheritance relationship is also called an "is a(n)" relationship. For instance, again we consider car as example.

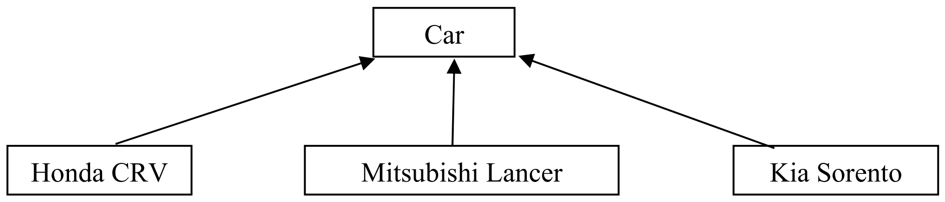
A Honda CRV is a car.
A Mitsubishi Lancer is also a car.
A Kia Sorento is yet another car.

From the above example, we have the common denominator/type/class: **a car**. Honda CRV, Mitsubishi Lancer and Kia Sorento are specifics examples of car.

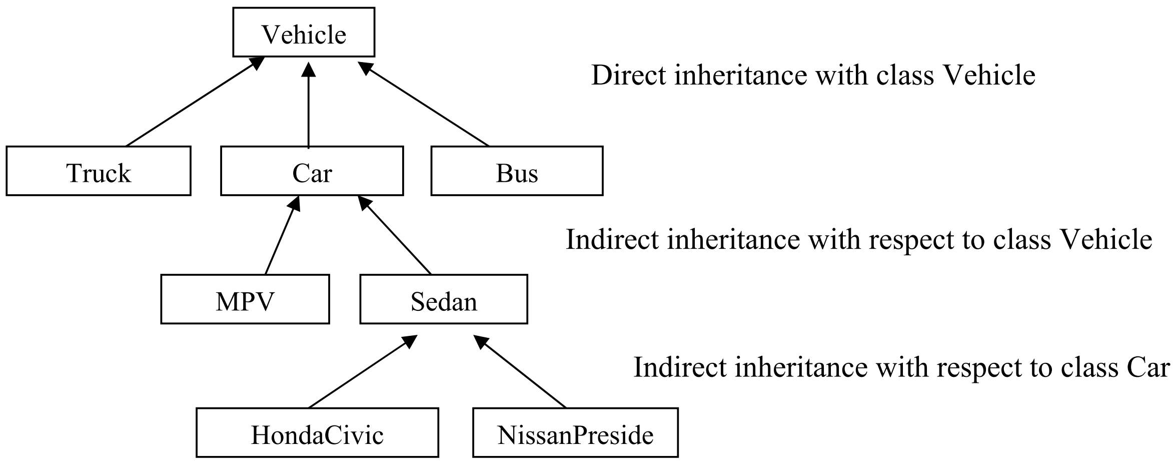
Difference between inheritance and composition.

a. Composition does not enhance the existing classes. The enclosing class is just like a client of the existing classes.

b. Inheritance enhances the existing classes. It inherits all existing classes (data and function) members and enhances the existing classes by adding new members (data and function).



Important Terminologies


Vehicle
Truck
Car
Bus
MPV
Sedan
HondaCivic
NissanPreside
Direct inheritance with class Vehicle
Indirect inheritance with respect to class Vehicle
Indirect inheritance with respect to class Car

- **base class**: a class which is inherited from by other classes. E.g., class Vehicle is the base class for all other classes. Class Sedan is base class for classes HondaCivic and class NissanPresident.
- **derived class**: a class which inherits from base class(es). E.g., classes Truck, Car and Bus are derived classes from base class Vehicle.
- **direct inheritance**: inheritance relationship where the derived class inherit directly from its base class. E.g., class Car and class Vehicle have direct inheritance relationship.
- **indirect inheritance**: inheritance relationship where the derived class inherit indirectly from its base class. E.g., class HondaCivic and class Car have indirect inheritance relationship
- **single inheritance**: a derived class inherits from a single base class. E.g., all examples in Fig. 5.3 are single inheritance
- **multiple inheritance**: a derived class inherits from more than one base classes.

Types of inheritance.
There are three types of inheritance, namely private, protected and public inheritances.

Base class member access specifier	Types of inheritance		
	public	protected	Private
Public	Public in derived class. Can be accessed directly by any non-static member functions.	Protected in derived class. Can be accessed directly by all non-static member functions.	Private in derived class. Can be accessed directly by all non-static member functions.
Protected	Protected in derived class. Can be directly accessed by all non-static member functions.	Protected in derived class. Can be directly accessed by all non-static member functions.	Private in derived class. Can be accessed directly by all non-static member functions.
Private	Hidden in derived class. Can be accessed by non-static member functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non-static member functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non-static member functions through public or protected member functions of the base class.

General Syntax.
General syntax for single inheritance:

Derived_class_name : type_of_inheritance Base_class_name{};

General syntax for multiple inheritance:

**Derived_class_name : type_of_inheritance1 Base_class_name1,
type_of_inheritance2 Base_class_name2,…{};**

Example.

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


class Counter                                   //base class
{
protected:                                      //NOTE: not private
        unsigned int count; //count
public:
        Counter() : count(0)                    //no-arg constructor
                { }
        Counter(int c) : count(c)                //1-arg constructor
                { }
        unsigned int get_count() const          //return count
                { return count; }
        Counter operator ++ ()                  //incr count (prefix)
                { return Counter(++count); }
};

class CountDn : public Counter                  //derived class
{
public:
        Counter operator -- ()                  //decr count (prefix)
                { return Counter(--count); }
};

void main()
{
CountDn c1;                                     //c1 of class CountDn
cout << "\nc1=" << c1.get_count();              //display c1
++c1; ++c1; ++c1;                               //increment c1, 3 times
cout << "\nc1=" << c1.get_count();              //display it
--c1; --c1;                                     //decrement c1, twice
cout << "\nc1=" << c1.get_count();              //display it
cout << endl;
}
```

Exercise:

1. A hospital wants to create a database regarding its indoor patients. The information to store include:
   a) Name of patient
   b) Date of admission
   c) Disease
   d) Date of discharge

   Create a base class to store the above information. The member functions should include functions to enter the information and display a list of all patients. Create a derived class to store the age of patients. List the information about the age of all the patients.

Lab # 8

Object:
To understand member function overriding.

Theory:
Member functions in a derived class can be overriden—that is, have the same name as—those in the base class. This is done so that calls in your program work the same way for objects of both base and derived classes.

```
#include <iostream>
#include <process.h>                                //for exit()
using namespace std;

class Stack
{
protected:                                          //NOTE: can't be private
        enum { MAX = 3 };                           //size of stack array
        int st[MAX];                                //stack: array of integers
        int top;                                    //index to top of stack
public:
        Stack()                                     //constructor
                { top = -1; }
        void push(int var)                          //put number on stack
                { st[++top] = var; }
        int pop()                                   //take number off stack
                { return st[top--]; }
};

class Stack2 : public Stack
{
public:
        void push(int var)                          //put number on stack
                {
                if(top >= MAX-1)                    //error if stack full
                   { cout << "\nError: stack is full"; exit(1); }
                Stack::push(var);                   //call push() in Stack class
                }
        int pop()                                   //take number off stack
                {
                if(top < 0)                         //error if stack empty
                        {
                        cout << "\nError: stack is empty\n";
                        exit(1); }
                return Stack::pop();                //call pop() in Stack class
                }
};
```

```
void main()
{
Stack2 s1;
s1.push(11);                              //push some values onto stack
s1.push(22);
s1.push(33);
cout << endl << s1.pop();                 //pop some values from stack
cout << endl << s1.pop();
cout << endl << s1.pop();
cout << endl << s1.pop();
cout << endl;
}
```

Exercise:

1. Imagine a publishing company that markets both book and audiocassette versions of its works. Create a class publication that stores the title (a string) and price (type float) of a publication. From this class derive two classes: book, which adds a page count (type int); and tape, which adds a playing time in minutes (type float). Each of these three classes should have a getdata() function to get its data from the user at the keyboard, and a putdata() function to display its data. Write a main() program to test the book and tape classes by creating instances of them, asking the user to fill in data with getdata(), and then displaying the data with putdata().

Lab # 9

Object:
To learn polymorphism and its implementation using virtual functions.

Theory:
A **POLYMORPHIC** function is one that has the **same name** for different classes of the same family, but has **different implementations/behavior** for the various classes. In other words, polymorphism means **sending the same message (invoke/call member function)** to **different objects** of different classes in a hierarchy.

To enable polymorphism, the first thing that we need to do is declaring the overridden function as **virtual** one in the base class declaration. Once it is declared virtual, it is also **virtual** in the derived classes implicitly. However, it is a good practice to declare the function to be virtual as well in the derived classes

There are 3 pre-requisite before we can apply virtual functions:
1. Having a hierarchy of classes/implementing inheritance
2. Having functions with same signatures in that hierarchy of classes, but each function in each class is having different implementation (function definition)
3. Would like to use base-class pointer that points to objects in that hierarchy.

To implement virtual functions, you will just place virtual keyword at the function prototype as shown below:

virtual return_type functionName(argument list);

Advantages.
- It allows objects to be more independent, though belong to the same family
- New classes can be added to the family without changing the existing ones and they will have the basic same structure with/without added extra feature
- It allows system to evolve over time, meeting the needs of a ever-changing application

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

class Base                              //base class
{
public:
        virtual void show()             //virtual function
                { cout << “Base\n”; }
};

class Derv1 : public Base               //derived class 1
{
public:
        void show()
```

```
                { cout << “Derv1\n”; }
};

class Derv2 : public Base                    //derived class 2
{
public:
        void show()
                { cout << “Derv2\n”; }
};

void main()
{
Derv1 dv1;                                   //object of derived class 1
Derv2 dv2;                                   //object of derived class 2
Base* ptr;                                   //pointer to base class
ptr = &dv1;                                  //put address of dv1 in pointer
ptr->show();                                 //execute show()
ptr = &dv2;                                  //put address of dv2 in pointer
ptr->show();                                 //execute show()
}
```

Exercise:

1. Consider the following class definition

```
Class father
{
protected:
    int age;
public:
    father (iny x)
            { age = x;}
    virtual void iam()
            { cout << "I AM FATHER, my age is ….." << age<<endl;}
};
```

Derive two classes son and daughter from the above and for each define iam() to write a similar but appropriate message. Write a main() that creates objects of the three classes and then calls iam() for them. Declare pointer to father. Successively, assign addresses of objects of the two derived classes to this pointer and in each case, call iam() through pointer to demonstrate polymorphism.

Lab # 10

Object:

To understand the concept of an abstract base class

Theory:

A base class which does not have the implementation of one or more of its virtual member functions is said to be an **abstract base class**. In other words, an abstract base class is a base class without the definition of one or more of its virtual member functions. It serves as a framework in a hierarchy and the derived classes will be more specific and detailed.

Further, a virtual member function which does not have its implementation/definition is called a **"pure" virtual** member function. An abstract base class is said to be incomplete –missing some of the definition of its virtual member functions– and can not be used to instantiate objects. However, it still can be used to instantiate pointer that will be used to send messages to different objects in the inheritance hierarchy to affect polymorphism.

Pure virtual member function declaration:
virtual return_type functionName(argument_list) = 0;

	Abstract base class	**Concrete base class**
Data member	Yes	Yes
Virtual function	Yes	Yes
Pure virtual function	Yes	No
Object instantiation	No	Yes
Pointer instantiation	Yes	Yes

Example

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

class CPolygon
{
protected:
           int width, height;
public:
          void set_values (int a, int b)
                    { width=a; height=b; }
          virtual int area (void) =0;
  };

class CRectangle: public CPolygon
{
public:
          int area (void)
               { return (width * height); }
  };
```

```
class CTriangle: public CPolygon
{
public:
        int area (void)
            { return (width * height / 2); }
  };

Void main ()
{
CRectangle rect;
CTriangle trgl;
CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
cout << ppoly1->area() << endl;
cout << ppoly2->area() << endl;
}
```

Exercise:

1. Create a base class called shape. Use this class to store two double type values that could be used to computer the area of the figures. Derive two specific classes called triangle and rectangle from the base shape. Add to the base class a member function get_data () to initialize base class data members and another function display_area () to computer and display the area of figures. Make display_area () as virtual and redefine this function in derived class to suit their requirements. Using these three classes, design a program that will accept the dimensions of a triangle or a rectangle interactively and displays the area.