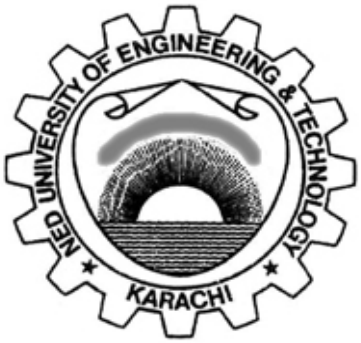
Workbook

Object Oriented Programming
(CT – 251)



Name _____

Roll No _____

Batch _____

Year _____

Department _____

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

Workbook

Object Oriented Programming
(CT – 251)

Prepared by

Dr. Saman Hina
Associate Professor, CS&IT

Approved by

Chairman
Department of Computer Science & Information Technology

Table of Contents

11.	Introduction to design patterns and its benefits		
12.	The Adapter Pattern		
13.	The Bridge Pattern		
14.	Reviewing the concepts of object oriented software development and design patterns case study.		

Lab # 1

Object:

Introduction to Object Oriented Programming with C++, .Net Framework and Visual Studio 2010.

.

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. C++ programs differ from C programs in other ways, including how I/O is performed and what headers are included. Also, most C++ programs share a set of common traits that clearly identify them as C++ programs. Before moving on to C++'s object-oriented constructs, an understanding of the fundamental elements of a C++ program is required.

.

The New C++ Headers

When you use a library function in a program, you must include its header. This is done using the #include statement. Standard C++ still supports C-style headers for header files that you create and for backward compatibility. However, Standard C++ created a new kind of header that is used by the Standard C++ library.

The new-style C++ headers are an abstraction that simply guarantee that the appropriate prototypes and definitions required by the C++ library have been declared. Since the new-style headers are not filenames, they do not have a .h extension. They consist solely of the header name contained between angle brackets. For example, here are some of the new-style headers supported by Standard C++.

<iostream> <fstream> <vector> <string>

Namespaces

When you include a new-style header in your program, the contents of that header are contained in the std namespace. A namespace is simply a declarative region. The purpose of a namespace is to localize the names of identifiers to avoid name collisions. Elements declared in one namespace are separate from elements declared in another. Originally, the names of the C++ library functions, etc., were simply put into the global namespace (as they are in C). However, with the advent of the new-style headers, the contents of these headers were placed in the std namespace.

I/O Operators

cout << “This is output. \n”;

This statement introduces new C++ features, cout and <<. The identifier cout is a predefined object that represents output stream in C++. The standard output stream represents the screen. The operator << is called the insertion operator. It inserts (or sends) the contents of the variable on its right to the object on its left.

cin<<num1;

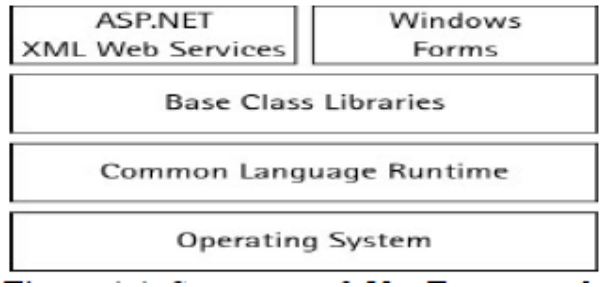
The statement is an input statement and causes program to wait for the user to type in number. The identifier cin is a predefined object in C++ that corresponds to the standard input stream. Here stream represents the keyboard. The operator >> is known as extraction or get from operator. In general, you can use cin >> to input a variable of any of the basic data types plus strings.

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;
}
```

Introduction to dot net framework

.NET is a collection of tools, technologies, and languages that all work together in a framework to provide the solutions that are needed to easily build and deploy truly robust enterprise applications. These .NET applications are also able to easily communicate with one another and provide information and application logic, regardless of platforms and languages.



The first thing that you should notice when looking at this diagram is that the .NET Framework sits on top of the operating system. Presently, the operating systems that can take the .NET Framework include Windows XP, Windows 2000, and Windows NT. There has also been a lot of talk about .NET being ported over by some third-party companies so that a majority of the .NET Framework could run on other platforms as well.

At the base of the .NET Framework is the Common Language Runtime (CLR). The CLR is the engine that manages the execution of the code.

The next layer up is the .NET Framework Base Classes. This layer contains classes, value types, and interfaces that you will use often in your development process. Most notably within the .NET Framework Base Classes is ADO.NET, which provides access to and management of data.

The third layer of the framework is ASP.NET and Windows Forms. ASP.NET should not be viewed as the next version of Active Server Pages after ASP 3.0, but as a dramatically new shift in Web application development. Using ASP.NET, it's now possible to build robust Web applications that are even more functional than Win32 applications of the past.

The second part of the top layer of the .NET Framework is the Windows Forms section. This is where you can build the traditional executable applications that you built with Visual Basic 6.0 in the past. There are some new features here as well, such as a new drawing class and the capability to program these applications in any of the available .NET languages.

Introduction to visual studio 2010

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft for creating, documenting, and debugging programs written in a verity of .net programming languages.

It is used to develop console and graphical user interface applications along with Windows Forms applications, web sites, web applications, and web services in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

Exercise:

1. Write a program that displays the squares, cubes and fourth powers of the numbers 1 through 5.

2. Write a program to implement four function calculator. Allow user to provide input and select the operation to be performed.

Lab # 2

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 # 3

Object:

The UML – Unified Modeling Language.

Theory:

The UML – Unified Modeling Language is a visual language (meaning a drawing notation with semantics) used to create models of programs. The UML include several diagrams for each process area. Some diagrams are for analysis, some for design and others for implementation and deployment. Each diagram shows the relationship among the different sets of entities, depending on the purpose of the diagram.

The UML is useful for communication among the team members about what is required to be done. The UML gives us tools to understand better requirements rather than moving to development phase straight away. In this context, we will cover two diagrams;

1. The Class diagram
2. The Sequence diagram

The Class Diagram

This is the most basic and commonly used UML diagram. It not only describes classes but also shows the relationships among them. Different types of relationships possible are;

- When one class is a “kind of” another class; is-a relationship.
- When there are associations between two classes;
  - ✓ One class “contains” another class: has-a relationship (Aggregation)
  - ✓ One class “uses” another class: the uses-a relationship (Dependency)
  - ✓ One class “creates” another class (Composition)

The Sequence Diagram

Sequence diagrams shows how objects interact with each other. These are most common type of interaction diagrams. 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.

Exercise:

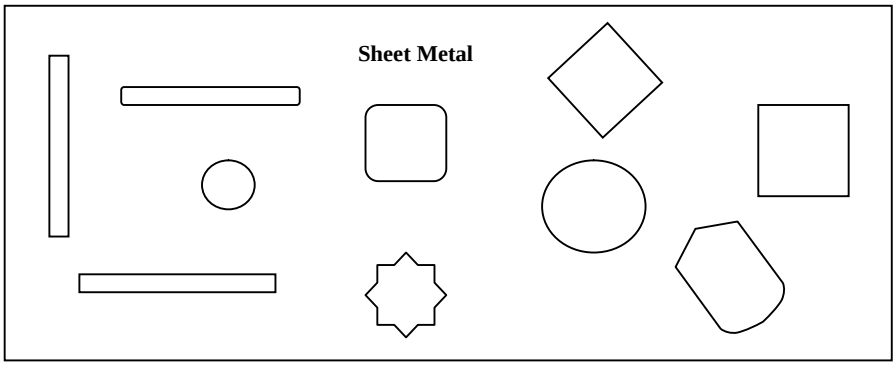
1. Draw a class diagram and sequence diagram for a computer tool that need to extract information from the two different versions of the CAD/CAM system. Both versions are not compatible with each other. This information extraction is required so that an expert system could use it in a particular way. The expert system needed this information to control the manufacturing of the part. Because the expert system was difficult to modify and would have a longer lifespan than the current version of the CAD/CAM system, this information extracting tool should be written in a way that it could easily be adapted to new revisions of the CAD/CAM system. The vocabulary of this system is described in Table 1 and Table 2.

Table 1 CAD/CAM Terminology

Term	Description
Geometry	The description of how a piece of sheet metal looks: the location of each of the features and their dimensions and the external shape of the sheet metal.
Part	The piece of sheet metal itself and the geometry of each part is to be stored.
Dataset or Model	The set of records in the CAD/CAM database that stores the geometry of a part.
NC machine and NC set	Numerically controlled (NC) machine. A special manufacturing tool that cuts metal using a variety of cutting heads that are controlled by a computer program. Usually the computer program is fed the geometry of the part. This computer program is composed of commands called the NC set.

Table 2 Features found in a Piece of sheet metal

Shape	Description
Slot	Straight cuts in the metal of constant width that terminate with either squared or rounded edges. Slots may be oriented to any angle.
Hole	Circles cut into the sheet metal. Typically they are cut with drill bits of varying width.
Cutout	Squares with either squared or rounded edges. These are cut by a high-powered punch hitting the metal with great impact.
Special	Performed shapes that are not slots, holes, or cutouts.
Irregular	Anything else. They are formed by using a combination of tools.



2. Use example case study from real-life and draw class diagram.

Lab # 4

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; }
};

int 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;
return 0;
}
```

Copy Constructor.

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

class Distance                                  //English Distance class
{
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 << '\"'; }
};
```

```
int 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;
return 0;
}
```

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 # 5

Object:
To understand the fundamentals of function overloading.

Theory:

Function overloading.
In C++, function overloading is achieved by creating two or more functions that share the same name but their parameter declarations are different. Each redefinition of the function must use either:

- Different types of parameters
- Different number of parameters

1. ***Different types of parameters:***

In the following example, there are two definitions of function print(). The first definition accepts an integer value whereas the second definition accepts a double value. Here, the function overloading is achieved by using arguments with different data type.

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

class example
{
public:
    void print (int a)
    {
        cout<<"Integer value:"<<a;
    }

    void print (double b)
    {
        cout<<"Double value:"<<b;
    }
};
void main()
{
    example one;
    one.print(5);
    one.print(5.5);
    getch();
}
```

2. ***Different number of parameters:***

In this example, the function add() has two definitions. Both of these definitions accept integer arguments so there is no difference in data type. However, the first definition accepts two arguments and the second definition accepts three arguments. Here, the function overloading is achieved by using the different number of arguments.

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

class example
{
private:
    int sum;
public:
    void add (int a, int b)
    {
        sum=a+b;
        cout<<"Two arguments"<<sum;
    }

    void add (int a, int b, int c)
    { sum=a+b+c;
        cout<<"Three arguments:"<<sum;
    }
};
void main()
{
    example one;
    one.add(2,3);
    one.add(3,5,1);
    getch();
}
```

Exercise:

1. Create a class **tollbooth** at a bridge. Cars passing by the booth are expected to pay a 10 PKR toll. Mostly they do, but sometimes a car goes by without paying. The tollbooth keeps track of the number of cars that have gone by and the total amount of money collected. Model this tollbooth class. The two data items are :
    a) Integer variable to the number of cars, and
    b) Integer variable to hold the total amount of money collected

The class has the following member functions:

a) A constructor that initializes both the data items to 0
b) A member function called **payingCar()** that increments the car total and adds 10 to total amount.
c) A member function called **noPayCar()** that increments the car total only.
d) A member function called **display()** to display both the data items.

Include a program to test this class. This program should allow the user to push "**1**" key to count a paying car, and a "**0**" key to count a non-paying car. Pushing the**"2"** key should cause the program to print out the total cars and total cash and then exit.

Lab # 6

Object:
To understand the fundamentals of Operator Overloading.

Theory:

Operator Overloading.
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

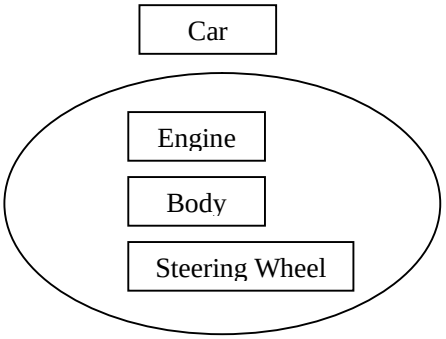
Object:
To understand the fundamentals of inheritance as opposed to composition.

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.

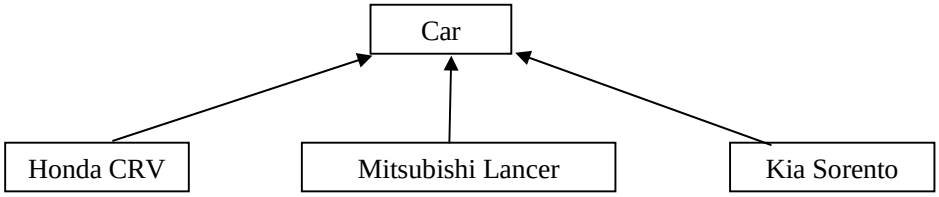
A Honda CRV is a car.
A Mitsubishi Lancer is also a car.
A Kia Sorento is yet another car.

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.

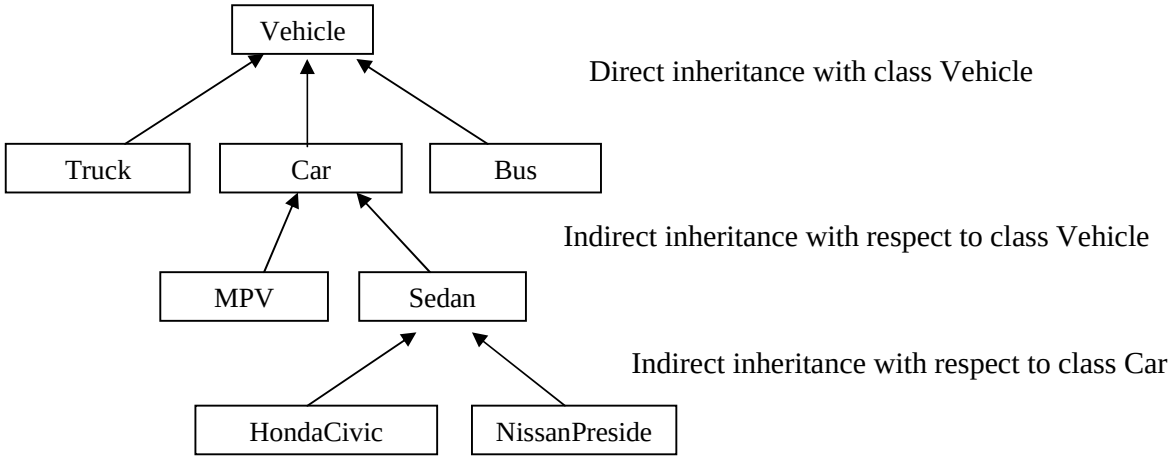
Difference between inheritance and composition.

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



- **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
                }
};

int 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();             //oops, popped one too many...
cout << endl;
return 0;
}
```

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/behaviour** 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"; }
};

int 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()
return 0;
}
```

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); }
  };

int 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;
return 0;
}
```

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.

Lab # 11

Object:

Introduction to Design Patterns and understanding the Facade Pattern.

Theory:

Design patterns are part of the cutting edge of object-oriented technology. Object-oriented analysis tools, books, and seminars are incorporating design patterns. The most commonly stated reasons for studying design patterns are because patterns enable us to;

- **Reuse Solutions –** By reusing already established designs, get a head start on your problems and avoid designing solution from scratch. You should get the benefits from other's experiences and do not have to reinvent solutions. Experienced designers reuse solutions that have worked in the past.
- **Establish Common Terminology –** Communication and teamwork require a common base vocabulary and a common view point of the problem. Design patterns provide a common point of reference during the analysis and design phase of a project.

The Gang of four[1] suggests a few strategies for creating good object-oriented designs. In particular, they suggest the following:

1. Design to interfaces.
2. Favor aggregation over inheritance.
3. Find what varies and encapsulate it.

The Façade Pattern

According to the Gang of Four, the intent of the Façade pattern is to:

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Key Features

Intent- You want to simplify how to use an existing system. You need to define your own interface.

[1] Gamma, Helm, Johnson and Vlissides (Writers of *Design Patterns: Elements of Reusable Object-Oriented Software*) .

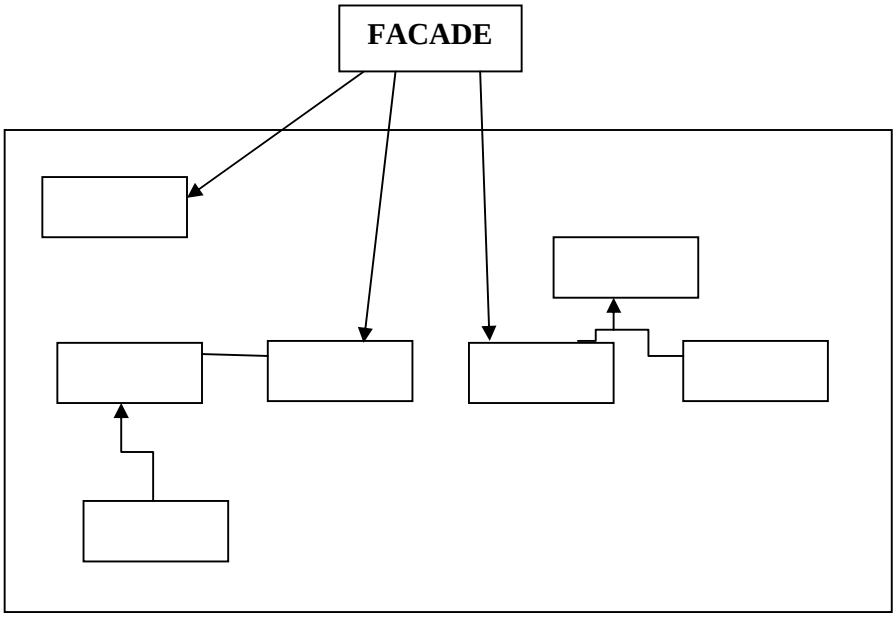
Problem- You need to use only a subset of a complex system. Or you need to interact with the system in a particular way.

Solution- the Façade pattern presents a new interface for the client for the existing system to use.

Participants and Collaborators- It presents a simplified interface to the client that makes it easier to use.

Consequences- the Façade simplify the use of the required subsystem. However, because the Façade is not complete, certain functionality may be unavailable to the client.

Implementation- Define a new class (or classes), that has the required interface. Have this new class use the existing system.



Exercise:

Work in group of three people. Consider any example of a system that illustrates the need of Façade. Explain key features of Façade pattern with respect to your example system and also draw diagram. Justify the use of Façade pattern in your example.

Lab # 12

Object:

Understanding the Adapter Pattern.

Theory:

The Adapter Pattern

Convert the interface of a class into another interface that the clients expect. Adapter let classes work together that could not otherwise because of incompatible interfaces.

Intent- Match an existing object beyond your control to a particular interface.

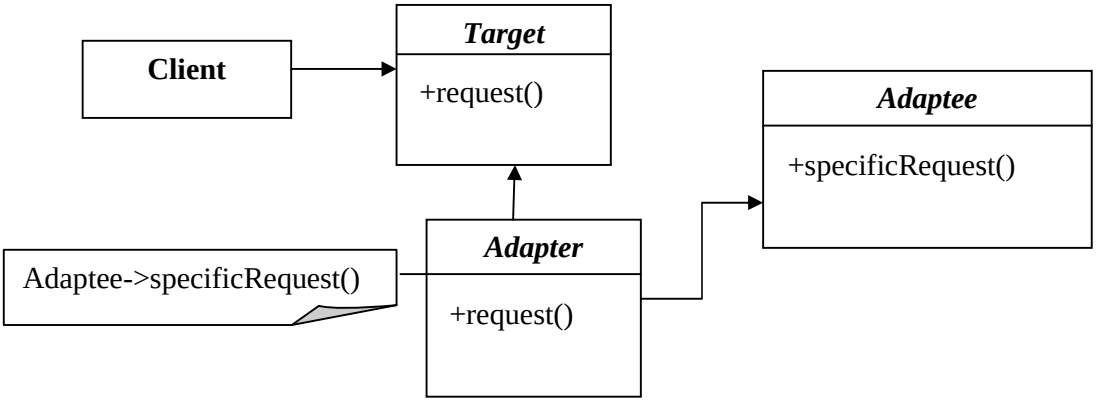
Problem- A system has the right data and behavior but the wrong interface. Typically used when you have to make something a derivative of an abstract class we are defining or already have.

Solution- The Adapter provides a wrapper with the desired interface.

Participants and Collaborators – The Adapter adapts the interface of an Adaptee to match that of the Adapter's Target (the class it derives from). This allows the Client to use the Adaptee as if it were a type of Target.

Consequences- The Adapter Pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.

Implementation- Contain the existing class in another class. Have the containing class match the required interface and call the methods of the contained class.



Exercise:

Work in group of three people. Consider any example of a system that illustrates the need of Adapter Pattern. Explain key features of Adapter pattern with respect to your example system and also draw diagram. Justify the use of Adapter pattern in your example.

Lab # 13

Object:

Understanding the Bridge Pattern.

Theory:

The Bridge Pattern

According to the Gang of four, the intent of the Bridge pattern is to;

"Decouple an abstraction from its implementation so that the two can vary independently"

Intent- Decouple a set of implementations from the set of objects using them.

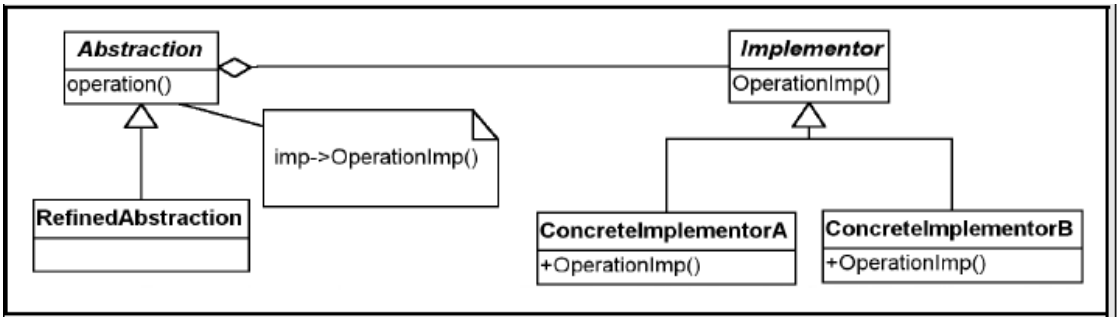
Problem- The derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes.

Solution- Define an interface for all implementations to use and have the derivations of the abstract class use that.

Participants and Collaborators- Abstraction defines the interface for the objects being implemented. **Implementor** defines the interface for the specific implementation classes. Classes derived from **Abstraction** use classes derived from **Implementor** without knowing which particular **ConcreteImplementor** is in use.

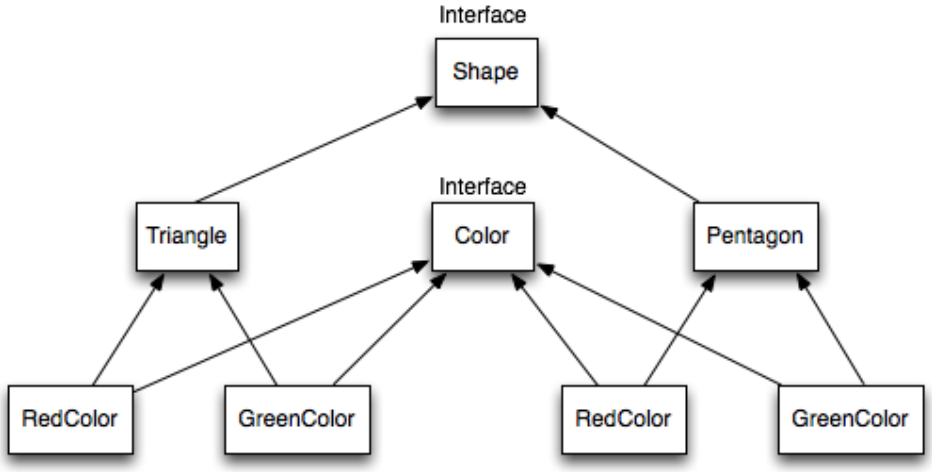
Consequences- The decoupling of the implementations from the objects that use them increases extensibility. Client objects are not aware of implementation issues.

Implementation- 1) Encapsulate the implementations in an abstract class. 2) Contain a handle to it in the base class of the abstraction being implemented.



Exercise:

Work in group of three people and apply bridge pattern to the following scenrio. Also justify the usage of bridge pattern with the help of its key features.



Lab # 14

Object:

Reviewing the concepts of object oriented software development.

Theory:

The Object Oriented Paradigm

The object-oriented paradigm is centered on the concept of the object. Everything is focused on objects. Objects have traditionally been defined as data with methods (the object-oriented term for functions). The advantage of using objects is that I can define things that are responsible for themselves. Objects inherently know what type they are. The data in an object allows it to know what state it is in. and the code in the object allows it to function properly (that is, do what it is supposed to do).

The best way to think about an object is to think of something with responsibilities. A good design rule is that objects should be responsible for themselves and should have those responsibilities clearly defined. Remember that objects have data to tell the object about itself and methods to implement required functionality. Many methods of an object will be identified as callable by other objects. The collection of these methods is called the **object's public interface**.

Consider the classroom example of previous lab; you could write the **Student** object with the method **gotoNextClassroom()**. There is no need to pass any parameters because each **Student** object would be responsible for itself. That is, it would know:

- What it needs to be able to move
- How to get any additional information it needs to perform this task

Initially, there was only one kind of student – a regular student who goes from one class to class. Note that there would be many of these "regular students" in the classroom (system). There should have an object of each student, to track the state of each student easily and independently of other students. However it is insufficient to require each Student object to have its own set of methods to tell it what it can do and how to do it, especially for the tasks that are common to all students.

A more efficient approach would be to have a set of methods associated with all students that each one could use or tailor to his or her own needs. 'General student' can be defined that contain definition of all these common methods. Then it can have

all manner of specialized student, each of whom has to keep track of his or her own private information.

In object oriented terms, this general student is called a *class*. A class is a definition of the behavior of an object. It contains a complete description of the following:

- The data elements the object contains.
- The methods the object can do.
- The way these data elements and methods can be accessed.

To get an object, we have to tell the program that we want a new object of this type. This new object is called an instance of the class. Creating instances of a class is called *instantiation*.

Exercise:

In the example explained in this lab, there is an assumption that any type of student is allowed into the collection (either regular student or graduate student). The problem is how do we manage the collection to refer to its constituents? In coding, this collection will actually be an array or something, of some type of object. If the collection were named something like **RegularStudent**, we will not be able to put graduate students into the collection. If collection is just a group of objects, how can it assures that wrong type of object is not included (that is, something that doesn't do "Go to your next class")?

Work in group of 3-5 students to find a solution to this problem.