Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Chapter 1:

Thinking Object Oriented

Object Oriented Programming

- Programming model which is based upon the concept of objects
- Programs are organized as cooperative collections of objects, each of which represents an instance of some class
- Importance is given to data rather than the algorithm
- Data and functions are tied together
- Data protection or security of data is achieved by data hiding using access specifiers
- Uses bottom up approach in program design

Characteristics of Object Oriented Programming

1. Classes
2. Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Data Abstraction
7. Message Passing

Characteristics of Object Oriented Programming

1. Object :

- Real world entities
- Object is entity that has characteristics and behavior
- Object is instance of a class

-If Ram is a Student than

- He has characteristics like name, address, registrationNumber, marks
- He has behavior like studying, taking exam, doing assignment

Characteristics of Object Oriented Programming

2. Classes :

- A group of objects that share common characteristics and behavior

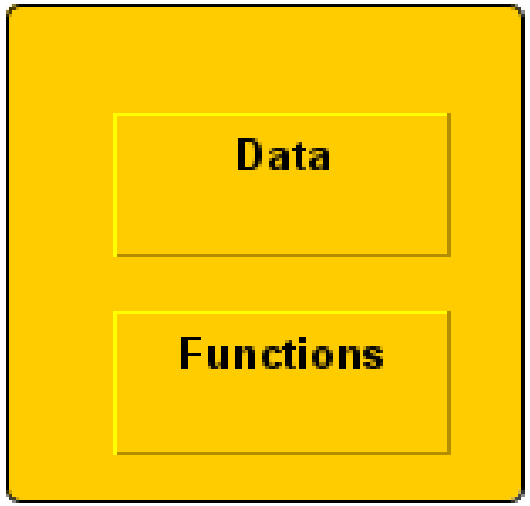
Example: Student

Student
Data : name rollNumber registrationNumber emailAddress Function : study() doAssignment() takeExam()

Characteristics of Object Oriented Programming

3. Encapsulation :

- Binding together the data and functions
- Prevents the data from unauthorized access



Characteristics of Object Oriented Programming

4. Inheritance :

- Ability to create new classes from the already existing class
- The existing class is called **base class** and the new class is called **derived class**
- Property of base class is inherited by derived class
- Derived class has data and functions of its parent class as well as its own data and functions
- Inheritance supports reusability

Characteristics of Object Oriented Programming

5. Data Abstraction:

- Abstraction refers representing only essential features or information and hiding background details

6. Polymorphism:

- Polymorphism means having many form
- Polymorphism can be achieved through function overloading and operator overloading

Characteristics of Object Oriented Programming

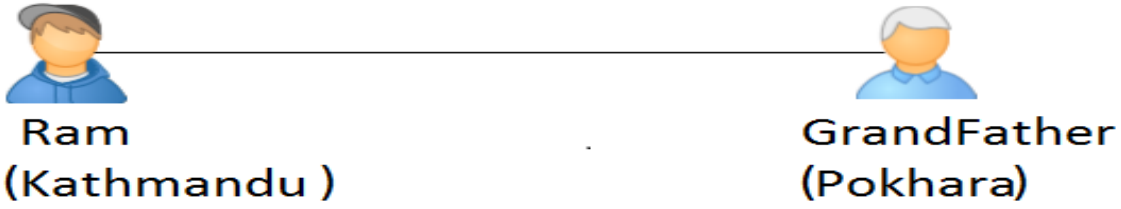
7. Message Passing :

- Objects communicate with each other by sending and receiving information to each other to perform certain task
- Message for an object is a request for execution of a procedure
- Invoke the function in receiving object that generates the desired result

Object.functionName(information)

A way of viewing the world:

Ram who lives in Kathmandu wants to send flowers to his grandfather who is in Pokhara.



A way of viewing the world:

- Ram logins to “FlowersNepal” application.
- Ram tells FlowersNepal kinds of flowers and address to which the flowers needs to be delivered
- Ram makes payment and can be assured that flowers will be delivered as expected

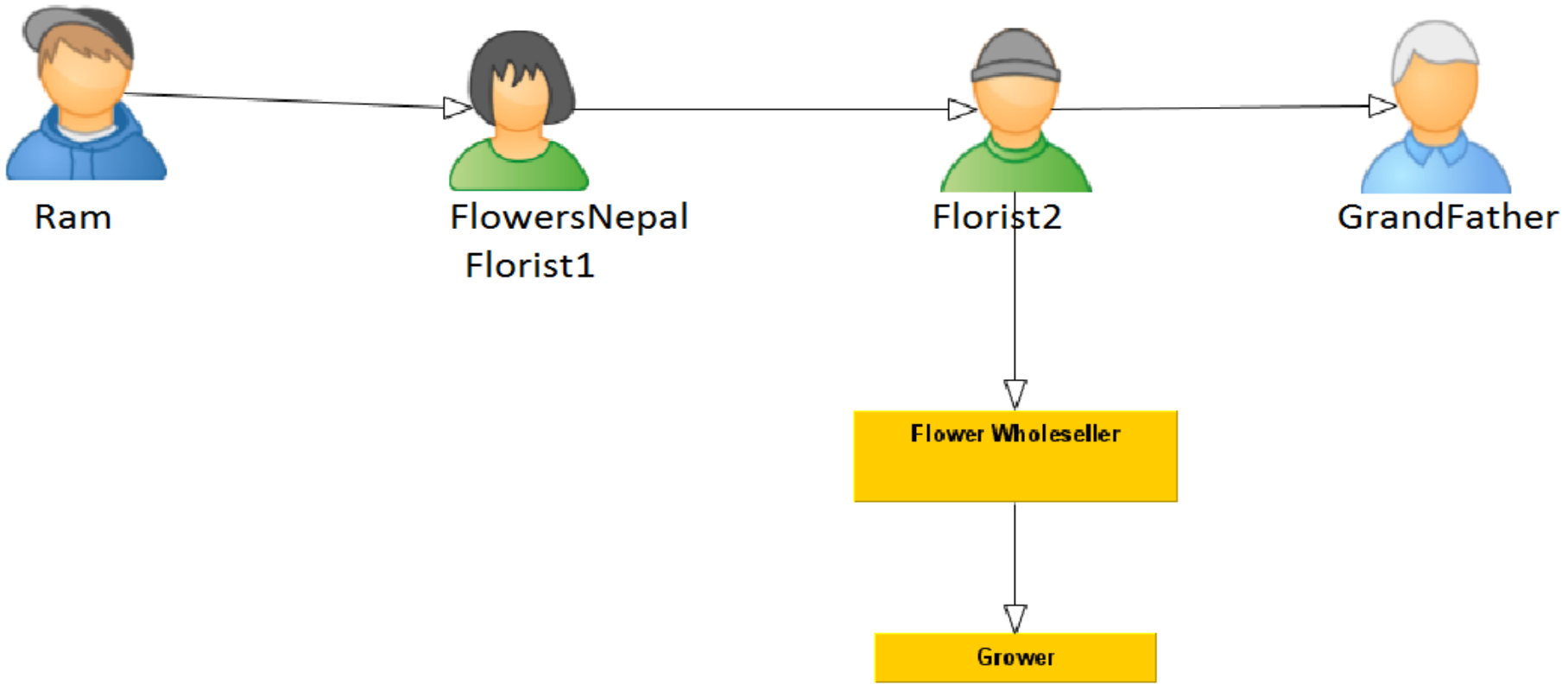
To solve the problem, the mechanism we used to was find an appropriate agent(FlowersNepal) and passed the agent a message containing request.

After that it is the responsibility of FlowersNepal to satisfy the request.

A way of viewing the world:

- FlowersNepal may have some sets of operations to do this. But Ram doesn’t need to know what methods or operations.
- This information is usually hidden
- Now, FlowersNepal will send a message to a florist in Pokhara with another message.
- Florist in Pokhara may have obtained the flowers from flower wholesaler who in turn had interactions with flower growers
- Florist in Pokhara passes the flowers to a delivery person with another message and so on.

Community of agents helping delivery of flowers



A way of viewing the world:

Message and methods:

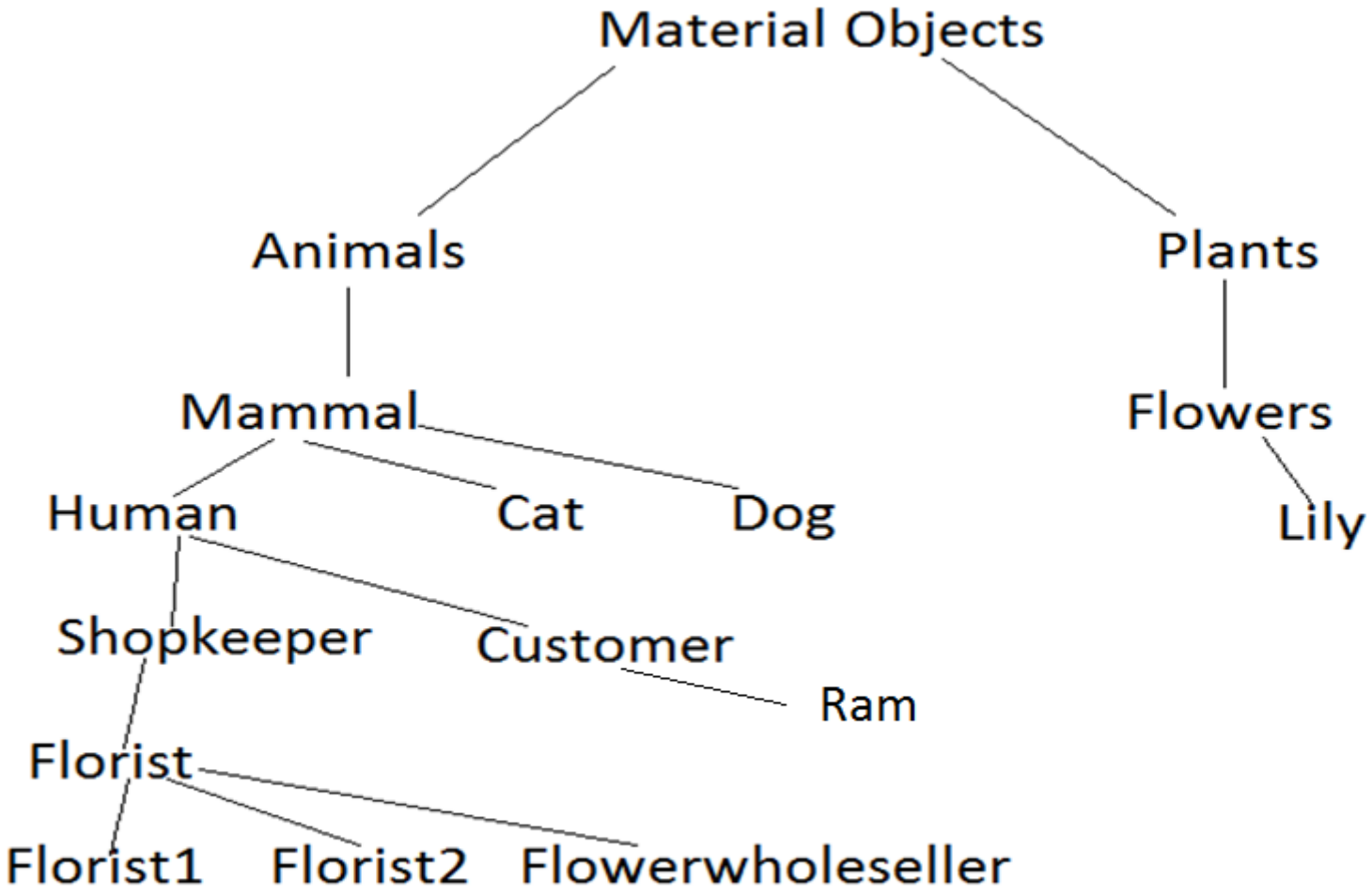
The chain of message that started from request given by Ram to “FlowersNepal” to solve Ram’s problem. This request lead to other requests which lead to more requests until flowers reached Grandfather.

Next principle of OOP is transmitting of a message to an agent(an object) responsible for action

A way of viewing the world:

- Ram requests for action and he only wants desired outcome that is flowers sent to GrandFather.
- FlowersNepal can choose any technique to achieve desired objective and it will not get any interference from Ram.
- By discussing the problem in terms of responsibilities, we increase the level of abstraction. This permits greater independence between objects which is important to solve complex problems

Class hierarchy



Alan Kay's Definition of Object Oriented Programming

1. Everything is an object.
2. Computation is performed by objects communicating with each other through message passing.
3. Each object has its own memory.
4. Every object is an instance of a class. A class simply represents a grouping of similar objects.
5. The class is the repository for behavior associated with an object. That is, all the objects that are instances of same class can perform the same actions.
6. Classes are organized into a singly rooted tree structure called inheritance hierarchy.

Computation as Simulation

- In discrete event driven simulation the user creates computer models of various elements of the simulation, describes how they will interact with one another and sets them moving.
- This is almost identical to Object Oriented Programming in which the user describes what the various entities in the program are, how will they interact with one another and finally set them in motion.
- Thus OOP we have the view that computation is simulation.

Coping with Complexity
The non-linear behavior of complexity

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

- The reason for this non-linear behavior is complexity which are
    1. the interconnection between software components are complicated
    2. a large amount of information has to be communicated among various members of the programming team

Abstraction

- Abstraction is used to control the complexity
- Abstraction means showing only essential attributes and hiding unnecessary information
- It is hiding unwanted details and showing only most essential information

Abstraction Mechanism

1. Procedure and function
2. Block Scope
3. Modules
4. Abstract Data Type
5. Object

Abstraction Mechanism

1. Function

-one of the first abstraction mechanism used in programming language

- function allows to write a set of code to perform a task and when ever you need to perform that task, we can simply call the function rather than writing same code multiple times

-Libraries of function gave the first possibility of information hiding where one programmer could write a set of procedures which could be used by many others.

-The other programmer would only know the necessary interface and not know the exact details of implementation.

- A function abstraction works by allowing the programmer to call a function written by the another programmer without necessarily understanding how it is implemented.

1. Procedure and function

```
int a,b;
void input()
        {
        ……
        }
 void add()
        {
        ……
        }
```

Abstraction Mechanism

2. Block Scope:

- Block scope determines the visibility or accessibility of the block
- When we use { }, it is a block. So, if we declare variables in the block scope, it means those variables exist only within the corresponding block

2. Block Scope:

Nesting of function

- In nesting of function, one function is nested within another function

```
void add(int x,int y)
        {
                …
        }
 void input()
        {
                int a,b;
                add(a,b);
        }
```

Abstraction Mechanism

3. Modules :

- Large programs can be broken into modules
- The programmers can think about the implementation of a piece of a program without full knowledge of the rest of the programs
- The rest of the programs needs to be understood only abstractly

3. Modules :

David Parnas' principle for modules are

i. The development module must be provided with all information needed to complete the module and nothing more.

ii. The developer of a module must provide the intended user with all the information to use the module correctly and nothing more.

Abstraction Mechanism

4. Abstract Data Type:

- Abstract data type is user defined data type or class whose properties is defined by set of values and set of operations
- ADT helps us to group data members and member functions
- It can decide which data members will be visible to which parts of programs
- It prevents user from directly accessing the private members

Abstract data type

5. Object

- Objects provide abstraction by hiding internal implementation details
- We just need to know which methods of the object are available to call and which input parameters we need to pass do specific operation
- We don’t need to understand how the method is implemented and what kinds of operations the method has to perform to create the expected result
- It has message passing feature

End of Chapter 1

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Introduction of Class:

- Class is user defined data type that is used to specify data representation and methods for manipulating the data in one package.
- The data and functions within a class are called members of the class.
- When you define a class, you define blueprint for a data type.
- Class defines what an object of class will contain and what operations can be performed in it.
- The data members and member functions can be grouped in private, public or protected section.

Class:

- Syntax for defining a class;

```
class class_name
{
    access specifier:
        variable_declaration;
        function_declaration;
    access specifier:
        variable_declaration;
        function_declaration;
};
```

- Example:

```
class Student
{
    private:
        char name[20];
        int roll;
        int marks;
    public:
        void getDetails();
        void display();
};
```

Data hiding in C++

- Data hiding is technique used in object oriented programming to hide object details i.e. data members to limit access to data and prevent them from unwanted manipulation
- There are three access specifiers:
  i. private
  ii. public
  iii. protected

i. Private access specifier:

- If the class members are declared private, then they can be accesses by member functions of that class only.
- Data members are made private to prevent direct access from outside the class.

ii. Public access specifier:

- If the class members are declared public then they can be accessed from anywhere in the program.
- Member functions are usually public which is used to manipulate the data present in the class.

iii. Protected access specifier:

It is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any derived class of that class.

Creating the object:

Syntax:

```
class_name object_name;

Student s1; //this statement creates a variable s1 of type Student
            //the variable s1 is object of class Student

Student s1,s2,s3,s4; // defining multiple objects of class Student
```

Creating the object:

```
class Student
{
    private:
            char name[20];
            int roll;
            int marks;
    public:
            void getDetails();
            void display();
    }s1,s2;
```

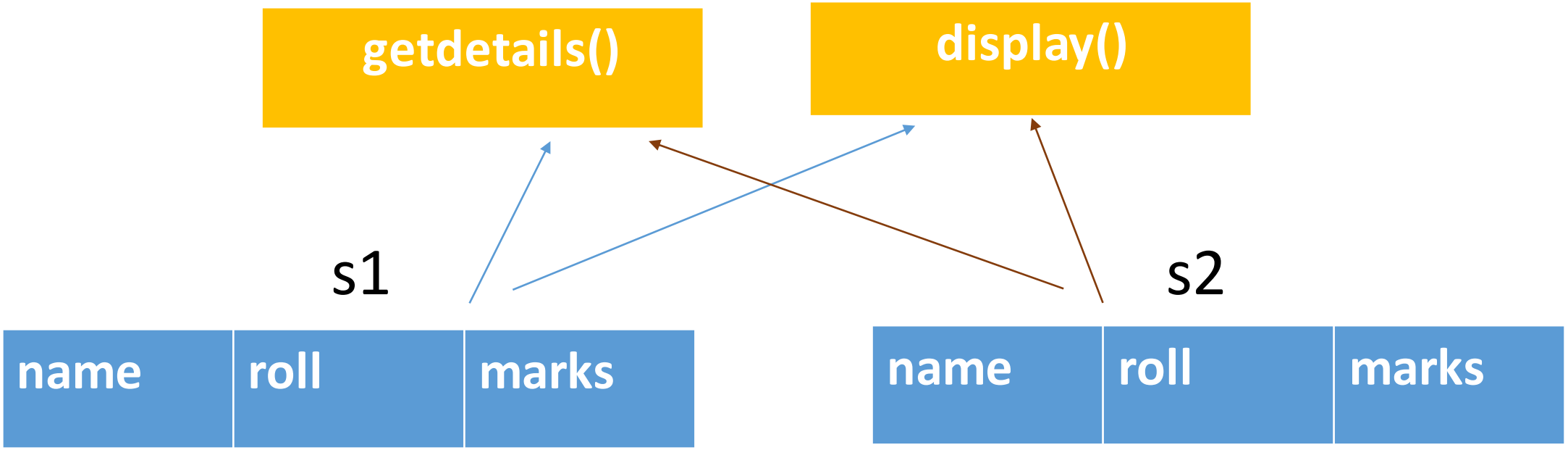
Creating the object:

```
class Student
{
    private:
        char name[20];
        int roll;
        int marks;
    public:
        void getDetails();
        void display();
    };
```

```
void main()
    {
        Student s1,s2;
    ...
    }
```

Creating the object:

Student s1,s2;



Accessing class members:

- The private members of a class can be accessed only through members of same class
- The public members of a class can be accessed from outside the class using object name and dot operator

Syntax for accessing public data member:

object_name.data_member;

Syntax for accessing public member functions:

object_name.function_name(arguments);

Accessing class members: Example

```
class Student
{
    private:
        char name[20];
        int roll;
        int marks;
    public:
        void getDetails();
        void display();
    };
```

```
void main()
{
        Student s1;
        s1.getDetails();
        s1.display();

        s1.roll=14578; //invalid as
roll is private member
```

Accessing class members: Example

```
class Student
{
    private:
        char name[20];
        int marks;
    public:
        int roll;
        void getDetails();
        void display();
    };
```

```
void main()
{
    Student s1;
    s1.getDetails();
    s1.display();
```

s1.roll=14578; valid as roll is public member

Defining member function:

1. Inside the class body
2. Outside the class body

<u>1. Defining member function inside class body</u>

If we define member function inside class definition, it is inline function

Defining member function: inside class body

```
class Student
{
    private:
        char name[20];
        int roll;
        int marks;
    public:
        void getDetails()
        {
        cout<<"Enter details";
        cin>>name>>roll>>marks;
        }
        void display()
        {
        cout<<"The name is "<<name<<endl;
        cout<<"Roll is "<<roll<<endl;
        cout<<"Marks is "<<marks<<endl;
        }
};
```

Defining member function: outside class body

- The function prototype is defined within the class body and its detail definition is written outside class body
- If we define function outside class body, we use scope resolution **::** operator

Syntax:

```
return_type class_name :: function_name(arguments)
{
        function body
}
```

Defining member function: outside class body

```
class Student
{
    private:
            char name[20];
            int roll;
            int marks;
    public:
            void getDetails();
            void display();
    };
```

```
void Student :: getDetails()
 {
  cout<<"Enter details";
  cin>>name>>roll>>marks;
  }
 void Student :: display()
 {
cout<<"The name is "<<name<<endl;
cout<<"Roll is "<<roll<<endl;
cout<<"Marks is "<<marks<<endl;
}
```

Example Program:

// class and object example program
//member function defined inside class body

```
#include<conio.h>
#include<iostream.h>

class Student
{
        private:
                char name[20];
                int roll;
                int marks;
        public:
        void getDetails()
        {
                cout<<"Enter name of student : ";
                cin>>name;
                cout<<"Enter roll number : ";
                 cin>>roll;
                 cout<<"Enter marks obtained :";
                  cin>>marks;
        }
void display()
        {
        cout<<endl<<endl<<"The Details are :"<<endl;
        cout<<"The name is "<<name<<endl;
        cout<<"Roll is "<<roll<<endl;
        cout<<"Marks is "<<marks<<endl;
        }
};
```

Program continued…

```
void main()
{
  Student S1,S2;
   S1.getDetails();
   cout<<endl;
   S2.getDetails();
   cout<<endl;
   S1.display();
   cout<<endl;
   S2.display();
   getch();
}
```

Note: Compile and run this program.

Example Program:

```
// class and object example program
//member function defined outside class body
#include<conio.h>
#include<iostream.h>

class Student
{
        private:
                char name[20];
                int roll;
                int marks;
        public:
                void getDetails();      //member function declaration inside class.
                void display();

};
```

Program continued…

```
void Student::  getDetails()              //member  function definition  outside class
{
                    cout<<endl<<"Enter  name of student  : ";
                    cin>>name;
                    cout<<"Enter  roll number  : ";
                     cin>>roll;
                     cout<<"Enter  marks obtained  :";
                    cin>>marks;
}
void Student  :: display()                  //member  function  definition  outside class
{
                    cout<<endl<<endl<<"The  Details are :"<<endl;
                    cout<<"The  name is "<<name<<endl;
                    cout<<"Roll is "<<roll<<endl;
                    cout<<"Marks  is "<<marks<<endl;
}
```

Program continued…

```
void main()
{
  Student S1,S2;     //object creation
   S1.getDetails();  //public member function call
   S2.getDetails();
   S1.display();
   S2.display();
   getch();
}
```

Note:Compile and run the program.

Practice:

Q1. Design a class called Person that contains appropriate members for storing name, age, gender, telephone number. Write member functions that can read and display these data.

Q2. Write a program to represent a Circle that has member functions to perform following tasks.

- Calculate area of circle
- Calculate perimeter of the circle

Q3. Create a class Point that represents a three dimensional coordinate system. Each object of Point should have coordinates x,y,z and methods to assign coordinates to the objects. Add a method to calculate the distance from origin and to the point (x,y,z). Define member functions outside the class body.

Q1. Design a class called Person that contains appropriate members for storing name, age, gender, telephone number. Write member functions that can read and display these data. Input details of 3 Person and display their details.

Nesting of member functions:

- A member function of a class can be called only by an object of that class using dot operator
- However, a member function can be called by using its name inside another member function of same class
- This is known as nesting of member function

Nesting of member function : Example

```
class Student
{
        private:
                char name[20];
                int roll;
                int marks;
        public:
                void studentDetails();
                void display();
};
```

```
void Student:: studentDetails()
{
..
 display();
}


 void main()
{
        Student S;
        S.studentDetails();
        …
}
```

Array of Objects:

Syntax for declaring array of objects:

```
class_name  object_name[size];
```

Example:

```
Student S[50];
```

Array of Objects : Example Program

Student S[4];

S[0]	name	roll	marks
S[1]	name	roll	marks
S[2]	name	roll	marks
S[3]	name	roll	marks

Example Program:

```
#include<conio.h>
#include<iostream.h>

class Student
{
        private:
                char name[20];
                int roll;
                int marks;

        public:
                void getDetails(int);
                void display();

};
```

Program continued…

```
void Student:: getDetails(int x)
{
        cout<<endl<<"Enter details of student "<<x<<" : "<<endl;
        cout<<"Name          : ";
         cin>>name;
        cout<<"Roll number     : ";
        cin>>roll;
        cout<<"Marks obtained :";
        cin>>marks;
        cout<<endl<<endl;
}
void Student :: display()
{
        cout<<"The name is "<<name<<endl;
        cout<<"Roll is "<<roll<<endl;
        cout<<"Marks is "<<marks<<endl<<endl;
}
```

Program Continued…

```
void main()
{
        Student S[50];
        int i,n;
        cout<<"Enter number of Students : ";
        cin>>n;
        cout<<endl<<endl<<"Enter details of students"<<endl;
        for(i=0;i<n;i++)
        {
                S[i].getDetails(i+1);
        }
        cout<<endl<<endl<<"Details of student are :"<<endl;
        for(i=0;i<n;i++)
        {
                S[i].display();
        }
  getch();
}
```

Practice:

Q1. Write a program to define a class named Employee with its data members name, salary, id number. Read records of n number of employees and display the records.

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Output Operator/ Insertion Operator /put Operator:

- The operator << (two less than sign) is called insertion operator
- It inserts data that follows it into the computer screen
- cout is predefined object of ostream class
- The object cout is used with insertion operator to display desired data in the screen

Syntax :

```
cout<< data or message to be displayed
```

Example:

```
cout<<“Hello world”;
cout <<x;   // x is a variable
cout<<500
```

Output Operator/ Insertion Operator :

- We can display multiple data using single insertion operator and single cout object, which is called cascaded output operation4

Syntax:

cout<< data or message to be displayed<< data or message to be displayed;

Example:

Int x=10,y=100;

cout<<“ The value of x is “<<x<<“ and the value of y is “<<y;

Input Operator/Extraction Operator/get Operator:

- The operator >> (two greater than sign) is known as extraction operator
- It extracts data from input device and store into a variable
- cin is predefined object of istream class
- The object cin is used with extraction operator to read data from input device

Syntax:

```
cin>>variable;
```

Example:

```
cin>>a ;
```

Input Operator/Extraction Operator:

- We can use cin to take more than one value as input.

Syntax:

```
cin>>variable1>>variable2>>variable3;
```

- iostream header file contains the declarations for cout ,cin,>> operator ,<< operator
- Header file iostream must be included in the program that use input/ output statements

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        cout<<“Hello World”;
        getch();
}
```

Example Program:

```
#include<iostream.h>
#include<conio.h>
        void main()
        {
        int x,y,sum;
        cout<<“Enter two numbers”;
        cin>>x;             //cin>>x>>y;
        cin>>y;
        sum=x+y;
        cout<<“The sum of  “<<x<<“and”<<y<<“ is “<<sum;
        getch();
        }
```

Enumeration:

- C++ allows programmers to create their own data type called enumerated type

Syntax:

```
enum enumerated_datatype
{
        Enumerator1, Enumerator2,….
};
```

Example:

```
enum days
{
        sunday, monday, tuesday, wednesday, thursday, friday
};
```

Enumeration:

```
#include<iostream.h>
#include<conio.h>

void main()
{
        enum days
                {
                sun,mon,tues,wed,thur,fri,sat
                };
  days day1,day2;
  day1=sun;
  day2=thur;

  cout<<"day 1= "<< day1 <<endl<<"day 2= "<< day2;
  getch();
}
```

Practice:

Q1. Write a program to define a enumerated data type Month with name of 12 months. Assign first month as 1 and display the integer value assigned to the months.

Inline Function:

- Inline function is a function that is expanded in line when it is called
- When the inline function is called whole code of the inline function gets inserted at the point of inline function call

Syntax:

```
inline returnType functionName( parameters)
{
        function body;
}
```

Program to calculate area of rectangle using inline function.

```
#include<iostream.h>
#include<conio.h>
        inline int area(int l,int b){
                return (l*b);
        }
    void main()
    {
            int len,bre,result;
            cout<<“Enter values of length and breadth”;
            cin>>len>>bre;
            result=area(len,bre);
            cout<<endl<<“The area of rectangle is “<<result;
            getch();
    }
```

Inline function:

Compiler can ignore the request for inlining in following cases:

- If function contains loop
- If function contains static variables
- If function is recursive
- If the function is large

Practice:

Q1. Write a program using inline function to calculate the square of a number.

Q2. Write a program to calculate volume of a cube. (vol=side*side*side)

Q3. Write a program using inline functions to calculate the multiplication and division of two user input numbers.

Default Argument:

- In C++, we can call function without specifying all its arguments by assigning default values for some arguments

Syntax for declaring function with default argument:

```
returnType functionName( argument1, argument2=default_value, argument3=default_value);
```

Default argument

Function prototype:

```
void test(int a,int b,int c=0,int d=10);
```

Function call:

```
test(w,x,y,z);
```

Function prototype:

```
void test(int a,int b,int c=0,int d=10);
```

Function call:

```
test(w,x,y);
```

Default Argument:

Function prototype:

```
void test(int a,int b,int c=0,int d=10);
```

Function call:

```
test(w,x);
```

During calling of function, arguments from calling function to called function are copied from left to right

Default Argument:

Function prototype:

```
void test(int a=5,int b=2,int c=0,int d=10);
```

Function call:

```
test();
```

Default Argument:

Function Prototype:

```
returntype functionName (argument1, argument2=defaultValue, argument3=defaultValue);
```

default values must be defined from right to left

- int testFunction(int a=5, int b=10, int c=20) ; //valid
- int testFunction(int a, int b=10, int c=20) ; //valid

- int testFunction(int a=5, int b, int c=20) ; //invalid
- int testFunction(int a, int b=10, int c) ; //invalid
- int testFunction(int a=5, int b, int c) ; //invalid

Example:

```
void add(int a, int b, int c=0, int d=0);
void main()
        {
                int x,y,z,w;
                /*ask user and input values
            for x, y and z*/
                add(x,y);
                add(x,y,z);
                add(x,y,z,w);

        getch();
        }
void add( int a, int b, int c, int d)
        {
                int sum=a+b+c+d;
                cout<<endl<<"The sum
                is  "<<sum;
        }
```

Practice:

Q1. Write a program to calculate simple interest using default value of r=1.5%. Ask the user for principal amount and time [SI=PTR/100]

Function Overloading:

- Function overloading is a feature in C++ where two or more functions can have the same name but can do different operations
- Two or more functions can have same name but different in their argument list
- We can overload function by either making
    i. type of arguments different
    ii. making number of argument different

- The correct function is invoked whose argument list matches the arguments in the function call

Function Overloading:

Example:

```
int calculate(int ); //function1
 float calculate(float ); //function2
 void calculate( int, int ); //function3
 int calculate(int, int, int); //function4
 float calculate(int ,float); //function5
 float calculate(float, int); //function6
```

Function Overloading:

- Example:
- Declaration:

```
int add(int a,int b); //function1
int add(int a, int b, int c); //function2
double add(double x , double y); //function3
double add(int p ,double q); //function4
double add(double a, int b); //function5
```

- Function call:

```
cout<<add(5,10);
cout<<add(100,100.5);
cout<<add(5.5,10);
cout<<add(12,23,56);
cout<<add(10.5,5.6);
```

```
#include<iostream.h>
#include<conio.h>

int add(int a,int b)
{
        cout<<"Function adds two integer"<<endl;
        return (a+b);
}

int add(int a,int b, int c)
{
        cout<<endl<<"Function adds three integer"<<endl;
        return (a+b+c);
}
double add(double a,double b)
{
        cout<<endl<<"Function adds two double values"<<endl;
        return (a+b);
}
```

```
double add(double a,int b)
{
        cout<<endl<<"Function adds double and int"<<endl;
        return (a+b);
}
double add(int a,double b)
{
        cout<<endl<<"Function adds int and double"<<endl;
        return (a+b);
}
void main()
{
        cout<<"Result is "<<add(2,10)<<endl<<endl;
         cout<<"Result is "<<add(10,10,20)<<endl;
        cout<<"Result is "<<add(10.5,20.52)<<endl;
        cout<<"Result is "<<add(10.5,30)<<endl;

        cout<<"Result is "<<add(35,70.5);

  getch();
}
```

Practice:

Q1. Write a program to find maximum of 2 numbers and maximum of 3 numbers using same function name, maximum().

Q2. Write a program to find the volume of 3 objects :cube, cylinder and

Rectangular box using same function name, volume().

Q3. Write a program to find the area of cube, cylinder and rectangle using concept of function overloading.

Reference Variable:

- Reference variable is another name for an already existing variable
- When a variable is declared as reference, it becomes an alternative name for an existing variable
- A variable can be declared as reference by putting ‘&’ in the declaration

Syntax:

```
datatype &referenceVariableName= VariableName;
int a;
int &ref=a;
```

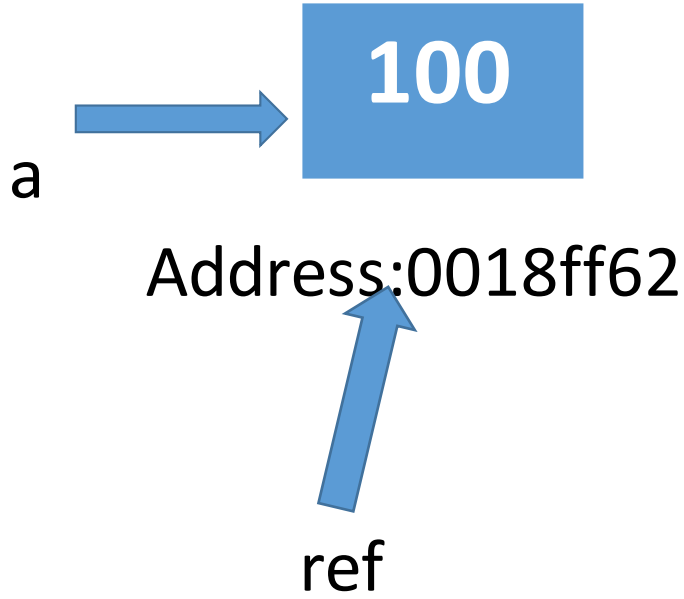
Reference Variable:

Syntax:

```
datatype &referenceVariableName= VaraibleName;
```

Example:

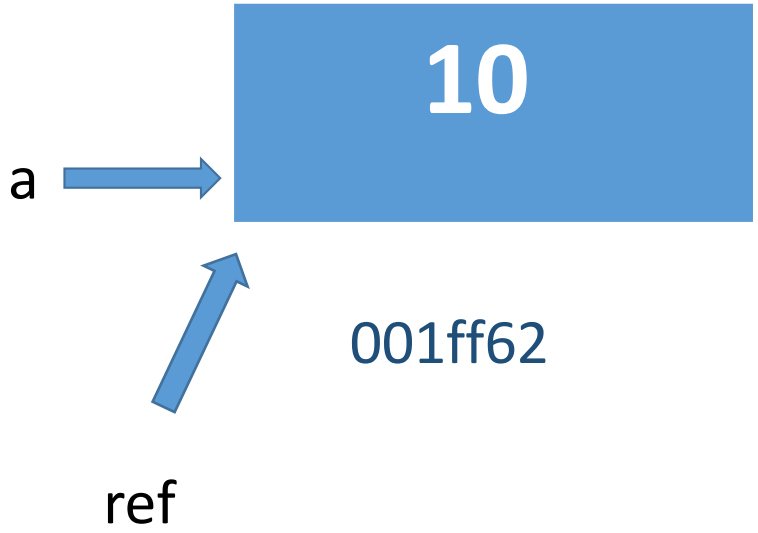
```
int a=100;
```




```
int &ref=a;
```

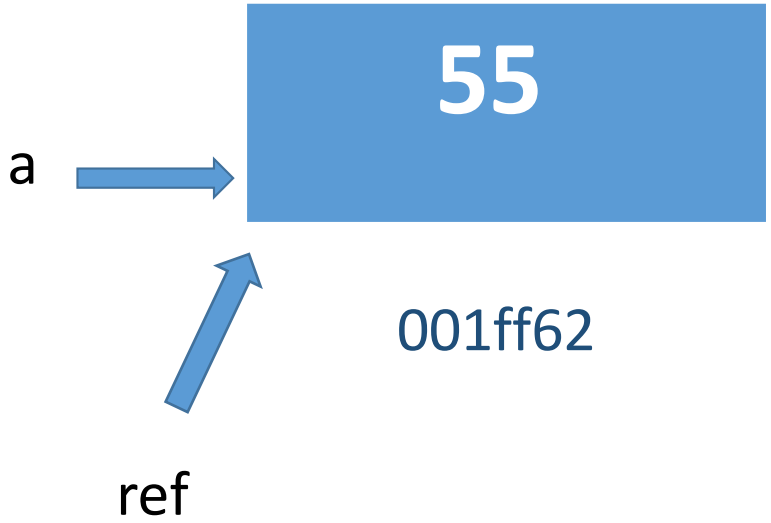
Example:

```
#include<iostream.h>
void main()
{
        int a=10;
        int &ref=a;
        cout<<"a="<<a<<endl;
         ref=55;
         cout<<"a="<<a<<endl;
        a=99;
         cout<<"ref="<<ref<<endl;
        cout<<"address of ref ="<<&ref<<endl;
         cout<<"address of a ="<<&a<<endl;

}
```



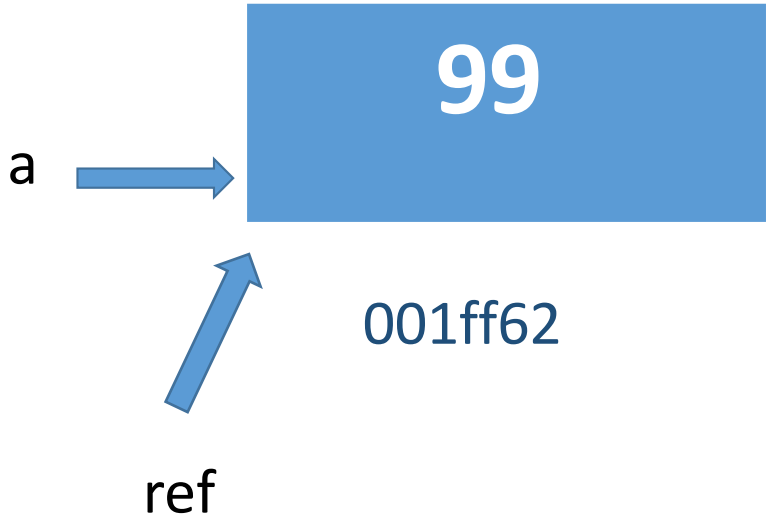
Example:

```
void main()
{
        int a=10;
        int &ref=a;
        cout<<"a="<<a<<endl;
        ref=55;
        cout<<"a="<<a<<endl;
        a=99;
        cout<<"ref="<<ref<<endl;
        cout<<"address of ref ="<<&ref<<endl;
        cout<<"address of a ="<<&a<<endl;

}
```



Example:

```
void main()
{
        int a=10;
        int &ref=a;
        cout<<"a="<<a<<endl;
         ref=55;
         cout<<"a="<<a<<endl;
        a=99;
         cout<<"ref="<<ref<<endl;
        cout<<"address of ref ="<<&ref<<endl;
         cout<<"address of a ="<<&a<<endl;

}
```



Call by Reference:

Function Prototype:

```
void testFunction( int &ref1, int &ref2);
```

Function Call:

```
testFunction  (   a       ,      b  );
```

When you pass parameters by reference, a new storage location is not created for these parameters. The reference parameters represent the same memory location as the actual parameters that are supplied to the method.

Here, ref1 will be alternative name of a and ref2 will be alternative name of b.

Example Program:

```
#include<iostream.h>
#include<conio.h>
void swap(int &x, int &y);
void main ()
{
        int a = 10;
         int b = 20;
        cout << "Before swap, value of a :" <<a << endl;
        cout << "Before swap, value of b :" <<b << endl;
        swap(a, b);
        cout << "After swap, value of a :" << a << endl;
         cout << "After swap, value of b :" << b << endl;
        getch();
}
```

```
void swap(int &x, int &y)
{
        int temp;
         temp = x;
        x = y;
        y = temp;
}
```

Practice:

Q1. Write and test the following computeSphere() function that returns the volume “v” and surface area “s” of a sphere with the given radius.

```
void computeSphere( float &v, float &s, float r)
```

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Friend Function:

- Private and protected members cannot be accessed from outside the class and it can be accessed only by the member function of same class
- But if a non- member function is defined as a friend function then, it can access private and protected data of class
- For accessing the private or protected data, the declaration of a friend function should be made inside the body of a class (no matter public or private or protected)

Characteristics of friend function:

1. A friend function can be declared in private or public or protected section of the class.
2. Since it is not a member function of class in which it is declared as friend, it is not called using object of class.
3. The friend function is called directly as normal function without the use of an object.
4. It is not a member function but can access the private and protected members of the class.
5. A friend function cannot access the class members directly. It needs to make use of class object and dot operator to access the class members.
6. It usually takes object as argument.

Degrades the importance of data hiding hence should be used only when necessary.

Friend Function:

We declare a friend function using friend keyword inside the body of the class.

Syntax for declaration of friend function:

```
class className
{
....
friend returnType  functionName(arguments);
...
};
```

Q1. Write a program to define a class Number that has two integer numbers as data members. Use friend function to find which one is greater among the two data members.

```
#include<conio.h>
#include<iostream.h>
class Number
{
        int num1,num2;
  public:
        void setNum();
         void displayNum();
    friend void maximum(Number N);
        //friend function declaration
};
```

```
void Number:: setNum()
{
        cout<<"Enter two integer
        numbers :";
         cin>>num1>>num2;
}
void Number:: displayNum()
{
        cout<<endl<<"The numbers
are”<<num1 <<" and "<<num2<<endl;
}
```

Q1.continued

```
void maximum(Number N)
{
if(N.num1>N.num2)
        cout<<endl<<N.num1<<" is greater than"<<N.num2<<endl;
 else
        cout<<endl<<N.num2<<" is greater than "<<N.num1<<endl;
}
```

```
void main()
{
        Number obj;
        obj.setNum();
        obj.displayNum();
        maximum(obj);
getch();
}
```

Example:

Q2. Write a program to define a class Time with data members hour, minute and second. Define a friend function to add two times.

Q3. Write a program to add two complex numbers using friend function.

Q4. Write a program to compare data of two different classes using friend function.

```
#include<conio.h>
#include<iostream.h>

class One; //forward declaration
class Two
{
        private:
                int num2;
        public:
                void setNum();
                void displayNum();
    friend void maximum(One a, Two b);
};
```

```
class One
{
        private:
                int num1;
  public:
                void setNum();
                void displayNum();
    friend void maximum(One a, Two b);
};
```

Q4. Program continued…

```
void One:: setNum()
{
        cout<<"Enter a number for class One";
         cin>>num1;
}
void One:: displayNum()
{
        cout<<endl<<"The data value of num in class One is "<<num1<<endl;
}
```

Q4. Program continued…

```
void Two:: setNum()
{
        cout<<"Enter a number for class Two";
         cin>>num2;
}
void Two:: displayNum()
{
        cout<<endl<<"The data value of num in class Two is "<<num2<<endl;
}
```

Q4. Program continued…

```
void maximum(One X, Two Y)
{
  if(X.num1>Y.num2)
            cout<<X.num1<<" is greater than "<<Y.num2;
   else
            cout<<Y.num2<<" is greater than "<<X.num1;
}
```

```
void main()
{
   One a;
   Two b;
   a.setNum();
   cout<<endl;
   b.setNum();
   cout<<endl;
   a.displayNum();
   b.displayNum();
   cout<<endl;
   maximum(a,b);
   getch();
}
```

Practice:

Q5. Write a program to swap the private data of two classes using friend function.

Q6. Write a program which has two classes named Practical and Theory which have data members to store practical and theory marks respectively. Use a non member function to calculate total marks (i.e practical marks of Practical class and theory marks of Theory class)

Friend Function:

Member function of one class can be friend function of another class. In this case, friend function is declared using scope resolution operator.

Syntax:

```
class One;
class Two
{
        ...
        int functionTwo(One obj); //member function of class Two
};
class One
{
        ...
        friend int Two::functionTwo(One obj); //functionOne() of class One is friend function of class Two
};
```

Friend Function:

```
class One;
class Two
{
        …
        int functionTwo(One obj); //member function
//of class Two
};
class One
{
        …
        friend int   Two::functionTwo(One obj);
//functionOne() of class One is friend function of class Two
};
```

Here, functionTwo() is member function of class Two and it is friend function of class One. So, it can access private and protected members of class One.

Example:

Q7. Write a program to swap contents of variables of two different classes where swapping function is member function of one class and friend function of another class.

Q8. Write a program which has two classes named Practical and Theory, which have data members to store practical and theory marks respectively. Define a class Marks that has member function to calculate total marks (i.e practical marks of Practical class plus theory marks of Theory class). Use the concept “member function of one class is friend of another class”.

Q7. Program:

```
#include<iostream.h>
#include<conio.h>

class One; //forward declaration
class Two
{
        private:
                int num2;
         public:
                void setNum();
                void displayNum();
                 void swap(One &a);
};
class One
{
        private:
                int num1;
          public:
                void setNum();
                 void displayNum();
                 friend void Two :: swap(One &a);
};
```

Q7. Program continued

```
void One:: setNum()
{
        cout<<"Enter a number for class One :";
         cin>>num1;
}
void One:: displayNum()
{
        cout<<endl<<"The  data value of num in class One is : "<<num1<<endl;
}
void Two:: setNum()
{
        cout<<"Enter a number for class Two :";
         cin>>num2;
}
void Two:: displayNum()
{
        cout<<endl<<"The  data value of num in class Two is : "<<num2<<endl;
}
void Two ::swap(One &X)
{
        int temp;
        temp=X.num1;
        X.num1=num2;
        num2=temp;
}
```

Q7. Program continued

```
void main()
{
  One a;
  Two b;
  a.setNum();
  cout<<endl;
  b.setNum();
  cout<<endl<<endl;
  cout<<"Before swapping :"<<endl;
  a.displayNum();
  b.displayNum();
  cout<<endl;
  b.swap(a);
  cout<<endl;
  cout<<"After swapping :"<<endl;
  a.displayNum();
  b.displayNum();

  getch();
}
```

Friend Class:

- A class can also be declared as friend of another class.
- A **friend class** is a **class** that can access the private and protected members of a **class** in which it is declared as **friend**.
- Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
- When a class **XYZ** declares another class **ABC** as its friend, then friend class **ABC** can access private and protected members of the class **XYZ**.

Friend Class:

- Syntax:

```
class ABC;
class XYZ
{
        ...
        friend class ABC;
};
class ABC
{
        ...
};
```

*When a class **XYZ** declares another class **ABC** as its friend, then friend class **ABC** can access every members of the class **XYZ** including private and protected members.*

Example Program:

```
#include<iostream.h>
#include<conio.h>

class ABC;
class XYZ
{
        int a,b;
        public:
        void setData()
        {
        cout<<"Enter two numbers :";
        cin>>a>>b;
        }
    friend class ABC;
};
class ABC
{
        public:
        void display(XYZ x)
        {
        x.setData();
        cout<<endl<<"The first number is "<<x.a<<endl;
        cout<<"The second number is "<<x.b<<endl;
        }
};
void main()
{
        ABC obj1;
        XYZ obj2;
        obj1.display(obj2);
         getch();
}
```

Friend Class:

Q1. Write a program which has three classes named Practical and Theory and Marks, which have data members to store practical marks, theory marks and total marks respectively. Calculate total marks (i.e practical marks of Practical class plus theory marks of Theory class). Use the concept of friend class.

Friend Function Practice:

Q1. Create classes called Class1 and Class2 with each having one private member. Add member function to set a value (say setValue) on each class. Add one more function max() that is friendly to both classes, max() function should compare two private member of two classes and show maximum among them. Create one-one object of each class and set a value on them. Display the maximum number among them.

Friend Function Practice:

Q2. Create a class called Mountain with data members name, height and location. Define a member function to initialize the data members. A friend function cmpHeight() to compare height of two objects and member function displayInfo() to display information of mountain. In main create two objects of the class mountain and display the information of mountain with greatest height.

End of Chapter 2

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Static Data Members:

- Static data members are class members that are declared using **static** keyword
- Static data members have following characteristics:

i. Only one copy of that static member is created for entire class and it is shared by all the objects of that class, no matter how many objects are created

ii. Its lifetime is entire program however scope is within the class

iii. Static data members are used to maintain values common to the entire class

Static Data Members:

Syntax for declaring static data member:

```
static data_type data_member_name;
```

Syntax for static member definition:

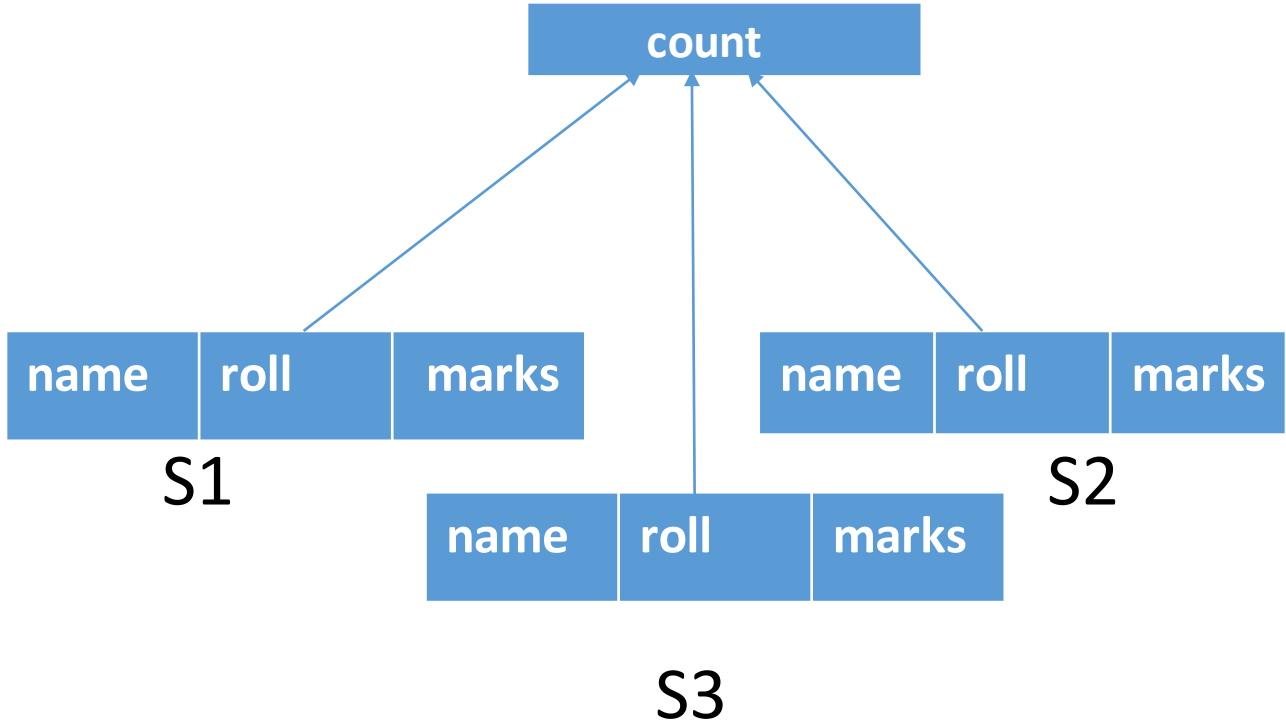
```
datatype class_name::static_data_member_name;
```

OR

```
datatype class_name::static_data_member_name=initial value;
```

Example:

```
class Student
{
char name[20];
int roll;
int marks;
static int count; // static data member
                  // declaration
...
};
int Student:: count; //static member definition
```



Example Program:

Compile and run the program of next page.

```
class Test
{
private:
        int num;
        static int count; //static member variable
 public:
        void getDetails()
         {
        num=++count;
          }
         void displayNum()
         {
        cout<<“Number:"<<num<<endl;
         }
        void displayCount()
         {
        cout<<"count : "<<count<<endl;
        }
};
int Test::count; //static data member definition
```

```
void main()
{
  Test t1,t2,t3;
  t1.getDetails();
  cout<<endl;
   t1.displayCount();
   t2.displayCount();
   t3.displayCount();
   cout<<endl;
   t2.getDetails();
   t1.displayCount();
   t2.displayCount();
   t3.displayCount();
   cout<<endl;
   t3.getDetails();
   t1.displayCount();
   t2.displayCount();
   t3.displayCount();
   t1.displayNum();
   t2.displayNum();
   t3.displayNum();
 getch();
}
```

Static Member Function:

- A class can also have static member functions
- We use static keyword to declare a static member function
- Syntax for declaring static member function:

```
static return_type functionName (parameters);
```

Static Member Function:

Characteristics of static member function:

1. A static member function is independent of any particular object of class
2. Static member function can be called even if no object of a class exist
3. It is accessed by using class name and scope resolution operator(::)

   Syntax:

   ```
   className:: functionName();
   ```

1. Static member function can only access static data members and other static member functions declared in same class
2. It can’t access non static members

```
class Test
{
private:
        int code;
        static int count; //static member variable
 public:
        void setCode()
         {
        code=++count;
          }
         void showCode()
         {
        cout<<"object number :"<<code<<endl;
         }
          static void showCount() //static member function
         {
        cout<<"count : "<<count<<endl;
        }
};
```

```
int Test::count; //static data member definition

void main()
{
        Test t1,t2;
        t1.setCode();
         t2.setCode();
        Test::showCount(); // accessing static function
        Test t3;
         t3.setCode();
        Test::showCount();
        t1.showCode();
        t2.showCode();
         t3.showCode();
 getch();
}
```

code= ++count;

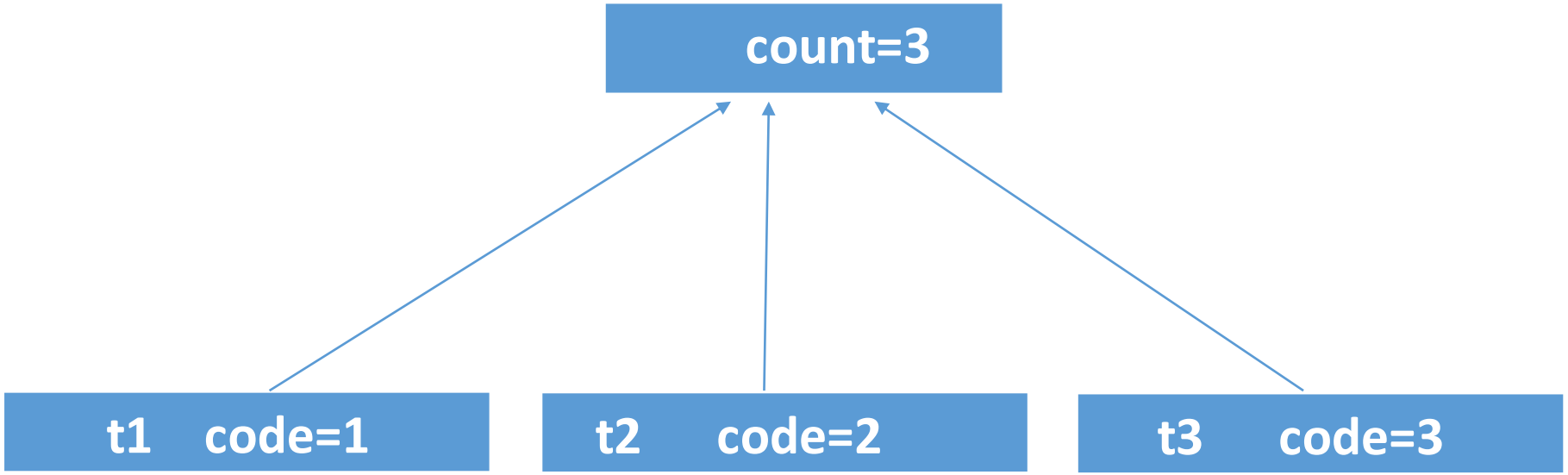
is executed whenever setCode() function is invoked and current value of count is assigned to code.

Since each object has its own copy of code, the value contained in code represents a unique number of its object.

```
void main(){
        Test t1,t2;
        t1.setCode();
         t2.setCode();
        Test::showCount();
        Test t3;
         t3.setCode();
        Test::showCount();
        t1.showCode();
        t2.showCode();
         t3.showCode();
}
```

Output of the program:

```
count=2
count=3
object number=1
object number=2
object number=3
```


count=3
t1 code=1
t2 code=2
t3 code=3

Practice:

Q1. Write a program to count number of students using data member and display the result using static member function. Use appropriate data members and member functions.

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Passing Object as Argument:

- An object can be passed to a function as an argument like normal variables
- When an object of a class is passed to a function, all data members of the class are passed in function

Syntax: Declaration of function with object as argument

```
return_type functionName(className object_name);
```

calling the function:

```
object.functionName(object_name);
```

Passing Integer as Argument:

While passing integer as argument in function

function declaration:

```
void testFunction(int x, int y );
```

function call:

```
testFunction(a,b);//a and b are integer variables
```

Passing Object as Argument:

While passing object as argument in function

function declaration:

```
void exampleFunction( class_name obj,class_name obj2);
```

function call:

```
object.exampleFunction(objectX,objectY);//objectX
//and  objectY are objects
```

Q. Define a class named Complex with data members real and img. Use appropriate member function of the class which accepts two objects of the class and add them.

```
#include<iostream.h>
#include<conio.h>
class Complex
{
        private:
        int real,img;
  public:
        void getData();
        void addComplex(Complex C1, Complex C2);
         void display();
};
void Complex::getData()
{
        cout<<endl<<"Real part : ";
        cin>>real;
        cout<<"imaginary part :";
        cin>>img;
}
void Complex::display()
{
if(img>0)
        cout<<real<<"+j"<<img;

  else
        cout<<real<<"-j"<<(-1)*img;
}
```

Program continued

```
void Complex::addComplex(Complex C1,Complex  C2)
{
        real=C1.real+C2.real;
        img=C1.img+C2.img;
}
```

```
void main()
{
  Complex first,second,result;
  cout<<"Enter first complex number";
  first.getData();
  cout<<"Enter second complex number";
  second.getData();
  cout<<"First complex number is :";
  first.display();
  cout<<endl<<"Second complex number is :";
  second.display();
  cout<<endl<<" The sum is : "<<endl;
  result.addComplex(first,second);
  result.display();
  getch();
}
```

Practice:

Q1. Write a program to define class Distance with data members feet and inches of appropriate type. Define member function of the class which accepts two objects of the class and adds them.

Q2. Write a program to define a class named Time with data members hours, minutes and seconds. Use appropriate member function of the class which accepts two objects of the class and adds them.

Q2. Write a program to define a class named Time with data members hours, minutes and seconds. Use appropriate member function of the class which accepts two objects of the class and add them.

```
#include<conio.h>
#include<iostream.h>
class Time
{
private:
        int hour,minute,second;
  public:
        void setTime();
         void displayTime();
         void addTime(Time, Time);
};
```

Program Continued…

```
void Time::setTime()
{
   cout<<"Enter hour :";
   cin>>hour;
   cout<<"Enter minute :";
   cin>>minute;
   cout<<"Enter second :";
   cin>>second;
}
```

Program Continued…

```
void Time::displayTime()
{
cout<<hour<<":"<<minute<<":"<<second<<endl;
}


void Time::addTime(Time  t1,Time t2)
{
  second=t1.second+t2.second;
  minute=second/60;
  second=second%60;

  minute=minute+t1.minute+t2.minute;
  hour=minute/60;
  minute=minute%60;

  hour=hour+t1.hour+t2.hour;
}
```

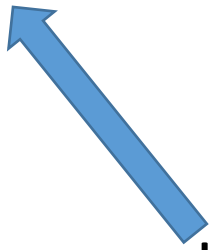
Program Continued…

```
void main()
{
Time time1,time2,time3;
  cout<<"Enter first time"<<endl;
  time1.setTime();
  cout<<endl<<"Enter second time"<<endl;
  time2.setTime();
  cout<<endl<<"The first time is : ";
  time1.displayTime();
  cout<<endl<<"The second time is : ";
  time2.displayTime();

  time3.addTime(time1,time2);

  cout<<endl<<"The resultant time after addition is : ";
  time3.displayTime();

  getch();
}
```

Returning Integer from function:

Syntax:

```
int function_name(arguments);
```



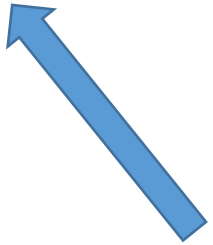
return_type

Returning Object from function:

A function can return object to the calling function as it can return integer, float or character values.

Syntax:

class_name function_name(arguments);



return_type

Example:

Q1. Write a program to define class Complex with data members real and img. Define a member function to add two complex numbers and return the sum of them.

```
#include<iostream.h>
#include<conio.h>

class Complex
{
        private:
        int real,img;
  public:
        void getData();
        Complex addComplex(Complex C2);
        void display();
};
```

Program continued…

```
void Complex::getData()
{
        cout<<endl<<"Real part : ";
        cin>>real;
        cout<<"imaginary part :";
        cin>>img;
}
void Complex::display()
{
        if(img>0)
        {
                cout<<real<<"+j"<<img;
        }
        else
        {
        cout<<real<<"-j"<<(-1)*img;
        }
}
```

Program continued…

```
Complex Complex::addComplex(Complex   C2)
{
        Complex C3;
        C3.real=real+C2.real;
        C3.img=img+C2.img;
        return C3;
}
```

Program continued…

```
void main()
{
        Complex first,second,result;
         cout<<"Enter first complex number";
        first.getData();
        cout<<"Enter second complex number";
        second.getData();
        cout<<"First complex number is :";
        first.display();
        cout<<endl<<"Second complex number is :";
        second.display();
        cout<<endl<<" The sum is : "<<endl;
        result=first.addComplex(second);
        result.display();
  getch();
}
```

Practice:

Q2. Write a program to define class Distance with data members feet and inches of appropriate type. Define member function of the class to adds two distances and return the sum in main program.

Q3. Write a program to define a class named Time with data members hours, minutes and seconds. Use appropriate member function of the class which accepts two objects of the class and adds them and returns the result.

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor : Er. Rabina Chaudhary

Chapter 3:Message, Instance and Initialization

Message Passing:

- A request for an object to perform its operation is called message
- A message for an object is a request for execution of a procedure and therefore will invoke a function in the object that generates the desired results
- Message passing involves specifying
  - name of the object
  - name of the function
  - information to be sent

Message Passing:

Syntax for message passing:

objectName . functionName (attributes/ information);

Example:

objectA . setData(20,30);

object message information

Initialization in C++:

```
class Test
        {
                int x,y;
                public:
                void setData()
                {
                        cout<<"Enter values of x and y";
                        cin>>x>>y;
                }
        };
void main()
{
        Test objOne;
        objOne.setData(); /*objOne.setData() invokes the member function setData() which assigns initial value of private data members x and y of objOne. */
}
```

Initialization in C++:

```
class Test
        {
                int x,y;
                public:
                void setData(int a,int b)
                {
                        x=a;
                        y=b;
                }
        };
void main()
{
        Test objOne;
        objOne.setData(10,20); /*objOne.setData() invokes the member function setData(int,int) and
passes values to assign initial value of private data members x and y of objOne. */
}
```

Constructor:

- A constructor is a member function of a class which initializes object of the class
- Constructor is automatically called when object of the class is created
- Constructor are useful for initializing the values of data of object

Constructor has following characteristics:

1. Constructor name should be same as its class name
2. Constructor does not have any return type not even void
3. It is called automatically when an object is created
4. It should be declared in public section
5. It cannot be static

Constructor:

Syntax for declaration of constructor:

```
class ClassName
{
        ...// data members
        public:
        ClassName(); //constructor
        … //other member functions
};
```

Constructor:

- Example:

```
 class Construct
{
        int x,y;
        public:
        Construct()
        {
          x=10;
           y=20;
        }
};

 void main()
{
        Construct  obj1 ;
        …
}
```

Types of Constructor:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

1. Default Constructor:

- A constructor without any argument is called default constructor
- It has no parameters
- If we do not specify a constructor explicitly, the compiler generates a default constructor implicitly

Default Constructor:Example

```
#include<iostream.h>
#include<conio.h>

 class Construct
        {
                int x,y;
                public:
                Construct()
                {
                        x=0;
                        y=0;
                }
                void display()
                {
                cout<<“Value of x=“<<x<<endl<<“Value of y =“<<y;
                }
        };
```

```
void main()
 {
        Construct obj;
        obj.display();
        getch();
 }
```

2. Parameterized Constructor:

1. Constructor with arguments is parameterized constructor
2. Here, we pass arguments while defining object of the class which are assigned to data members of the class
3. It is used to initialize data members of different objects with different initial values when they are created

Parameterized Constructor: Example

```
#include<iostream.h>
#include<conio.h>
  class Construct
        {
        int x,y;
        public:
                Construct()  //default constructor
                {
                x=0;
                y=0;
                }
                Construct(int a,int b) //parameterized constructor
                {
                        x=a;
                        y=b;
                }
                void display()
                {
                cout<<"Value of x="<<x<<endl;
              cout<<"Value  of y ="<<y<<endl;
                }
        };
```

```
void main()
  {
        Construct obj1(100,200), obj2(500,600) ;
        obj1.display();
        obj2.display();
        getch();
  }
```

Practice:

Q1. Write a program to define a class Rational. Use constructor to initialize the values of data members of object (i.e num and denum) and display the rational number. Use appropriate member functions as required.

2.2 Constructor with default arguments

- Parameterized constructor with default values in arguments

Example:

```
class Construct
        {
        int a,b,c;
        public:
                Construct();
                Construct(int x,int y=20, int z=30);
                void display();
        };
```

```
Construct:: Construct()
        {
                a=0;
                b=0;
                c=0;
        }
Construct::Construct(int x,int y, int z)
        {
                a=x;
                b=y;
                c=z;
        }
 void Construct:: display()
        {
        cout<<endl<<"Value of a="<<a;
        cout<<endl<<"Value of b="<<b;
        cout<<endl<<"Value of c="<<c;
        }
```

```
 void main()
{
        Construct obj1(1,2,3);
        Construct obj2(120,89);
        obj1.display();
        obj2.display();

        getch();
}
```

Practice:

Q1. Define a class Bank. Calculate simple interest using default value of rate=1.5%. Use constructor to initialize the object and appropriate member functions to display and calculate the result.

Q2. Define a class Complex. Write a program to add two complex numbers. Use constructor to initialize the objects.

Q3. . Define a class Complex. Write a program to add two complex numbers using friend function. Use constructor to initialize the objects.

3. Copy Constructor:

- Copy constructor is a constructor which creates an object as a copy of existing object of same class
- Copy constructor is used to create an object by initializing it with an object of the same class, which has been already created
- It uses reference to an object of same type (class) as argument

Copy Constructor:

Syntax for declaration of copy constructor:

```
class Construct
{
        …
        public:
        …
        Construct(Construct &); //copy constructor
};
```

Copy Constructor:

```
#include<iostream.h>
#include<conio.h>

 class Construct
        {
        int a,b,c;
         char name[20];
        public:
                Construct();                //default constructor
                Construct(int x,int y, int z);  //parameterized constructor
                Construct(Construct &);         //Copy constructor
                void display();
        };
Construct:: Construct()
        {

                a=0;
                b=0;
                c=0;
        }
```

```
Construct::Construct(int x,int y, int z)
        {
                a=x;
                b=y;
                c=z;
        }
Construct::Construct(Construct &obj)
{
        a=obj.a;
        b=obj.b;
        c=obj.c;
}
 void Construct:: display()
        {
        cout<<endl<<"Value of a="<<a;
        cout<<endl<<"Value of b="<<b;
        cout<<endl<<"Value of c="<<c;
        }
```

```
void main()
{
        Construct obj1(10,20,30);     //parameterized constructor called
        Construct obj2;               //Default constructor called

        cout<<"Object 1 :"<<endl;
        obj1.display();

        cout<<endl<<endl<<endl;
        cout<<"Object 2: "<<endl;
        obj2.display();

        cout<<endl<<endl<<endl;
        cout<<"Object 3 : "<<endl;
        Construct obj3(obj1);         //copy Constructor called
        obj3.display();

        getch();
}
```

Can Constructor be Overloaded?

- Yes, Constructor can be overloaded in same way as functions can be overloaded
- A class can have more than one constructor where each constructor take different set of parameters
- Constructor can be overloaded as class can have more than one constructor having same name but different signature
- When we define an object of a class, correct version of the constructor is called on the basis of passed arguments

Write an example program to illustrate constructor overloading.

Practice:

- Q1. Create a class called Mountain with data members name, height and location. Use constructor that initialize the members to the values passed to it as parameters. A function cmpHeight() to compare height of two objects and function displayInfo() to display information of mountain. In main create two objects of the class mountain and display the information of mountain with greatest height.

Destructor:

- A special member function of the class that is called when an object of the class is destroyed
- A constructor initializes the object, whereas a destructor destroys the object
- Destructor de allocates the memory allocated by object
- It is invoked automatically when the object goes out of the scope

Characteristics of destructor:

1. Destructor has same name as class and is preceded by a tilde(~)
2. Destructor doesn’t have any argument
3. Destructor doesn’t have any return type, not even void
4. It is called automatically whenever an instance of the class to which it belongs , goes out of scope
5. It is defined in public section of class

Destructor:

Syntax:

```
class className
{
        public:
        ~className ()
                {
                        destructor code
                }
                ...
};
```

Example:

```
#include<iostream.h>
#include<conio.h>
class ABC
        {
            public:
                ABC(){
                cout<<“This is a constructor”<<endl;
                }
                ~ABC(){
                cout<<“This is a destructor”<<endl;
                }
        };
void main()
        {
                {
                        ABC obj1, obj2;
                }
        getch();
        }
```

Destructor:

Objects are destroyed in reverse direction of creation

Example program:

```
#include<iostream.h>
#include<conio.h>
class ABC
{
        int id;
        public:
        ABC()                   //default constructor
          {
                id=0;
          }
        ABC(int a)              //parameterized constructor
          {
                id=a;
                cout<<endl<<"Constructor called for object : "<<id<<endl;
        }
        ~ABC()                  //destructor
        {
                cout<<endl<<"Destructor called for object :"<<id<<endl;
        }
};
```

Program continued

```
void main()
        {

                {
                        cout<<"**************Block  1****************"<<endl;
                        ABC obj1(1),obj2(2);
                                {
                                        cout<<endl<<endl<<"***************Block  2***************"<<endl;
                                        ABC obj3(3),obj4(4);
                                        cout<<endl<<endl<<endl<<endl;
                                        cout<<endl<<endl<<"*********End  of block 2**********"<<endl;
                                }
                        cout<<endl<<endl<<"***********End  of block 1***********"<<endl;
                }
                getch();
        }
```

Can destructor be overloaded?

- Destructor cannot be overloaded
- There can be only one destructor in a class
- Destructor doesn’t take any argument nor has a return type
- So multiple destructors with different signatures are not possible in a class
- Hence, destructor overloading is not possible

What will be the output of this program?

```
 class Car
{
        static int count;
        public:
        Car()
        {
                count++;
                cout<<"There are "<<count<<" number of
cars"<<endl;
        }
        ~Car()
        {
                count--;
        cout<<"There are "<<count<<" number of cars"<<endl;
        }
};
```

Program continued

```
 void main()
{
 cout<<“We are in main”<<endl;
Car c1,c2,c3,c4;
{
        cout<<“We are in block1”<<endl;
        Car c5;
}
{
        cout<<“We are in block2”<<endl;
        Car c6;
}
 cout<<“We have re-entered main”<<endl;
}
```

Practical Questions:

Q1. Imagine a tollbooth at a bridge. Carspassing by the booth are expected to pay five-rupee toll. Mostly they do, but sometimes a car go by without paying. The tollbooth keeps track of the number of cars that have gone by and of the total amount of money collected.

Model this tollbooth with class (say TollBooth). The two data members to hold number of cars and amount of money collected. The constructor may initialize both these to 0. A member function (say payingCar() increments the car total and adds 5 to the cash total, while another member function (say noPayCar()) increments the car total but adds nothing to the cash total. Finally member function display() displays the two totals.

Hint: allow user to press a particular key to indicate a paying car and another key to indicate non paying car.

Q2. Define a class to represent a bank account. Include the following members:

Data Members:

i) Name of the account holder

ii) Account number

iii) Balance Amount is the account

Member functions:

i) Open an account

ii) Deposit and withdraw money

iii) Display account information

Write a program to test this class for 10 costumers. Make use of all three types of constructors (whenever appropriate).

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Memory Allocation:

- Static Memory Allocation
- Dynamic Memory Allocation

Static Memory Allocation: In static memory allocation, memory is allocated in stack and the memory allocated is automatically freed or de allocated by the compiler itself when the scope of the allocated memory finishes.

Example:

```
void main()
{
        int num;            // variable
        int arrayEg[5];     //  array
                            ...
}
```

```
class Student
{
        ...
};
void main()
{
        Student obj;        // object
        Student objArray[10]; //array of object
        ...
}
```

Dynamic Memory Allocation:

- In dynamic memory allocation, the memory is allocated in heap
- While allocation of memory on heap, we need to delete the memory manually as memory is not de allocated or freed by the compiler itself even if the scope of allocated memory finishes
- We allocate and de allocate memory dynamically using **new** and **delete** operator respectively.

new Operator:

- The new operator is used for allocating memory dynamically
- new operator requests for the memory allocation in heap

Syntax to allocate memory using new operator:

```
pointer_variable = new datatype ;
```

Example:

```
int *ptr = new int;
```

new Operator:

- We can also initialize the memory using new operator.

Syntax:

```
pointer_variable = new datatype (value);
```

Example:

```
int *ptr = new int (50);
```

delete Operator:

- delete operator is used to de allocate the memory allocated by new operator

Syntax:

delete pointer_variable;

Syntax: To free dynamically allocated array pointed by pointer variable

delete [] pointer_variable:

Example Program:

```
//Dynamically allocate memory for an integer variable.
#include <iostream.h>
#include <conio.h>
        void main()
        {
                int *ptr1=NULL;
                ptr1 = new int ;     //declaring a integer variable dynamically
                *ptr1 = 100;
                 int *ptr2 = new int (1234);

                cout << "Value at address pointed by pointer variable 1 :"<< *ptr1 << endl;
                cout << "Value at address pointed by pointer variable 2 :"<< *ptr2 << endl;
          delete ptr1;    //memory pointed by ptr1 is released
          delete ptr2;    //memory pointed by ptr2 is released

        getch();
        }
```

new Operator:

- Allocate block of memory using new operator:

  Syntax:

  pointer_variable = new datatype [size];

  Example:

  int *ptr = new int [10];

Delete Operator :

Syntax: To free dynamically allocated block of memory or array pointed by pointer variable

delete [] pointer_variable:

Example Program:

```
//Dynamically allocate memory for an integer array.
 #include <iostream.h>
 #include <conio.h>
void main()
{
                int *ptr=NULL;
                int n;
                cout << "Enter the size of array :";
                 cin >> n;
                ptr = new int [n];                //dynamically allocate memory for n integers
                 cout << endl << "Enter "<< n << " integer numbers :" << endl;
                 for(int i=0; i<n ; i++)
                {
                        cin >> *( ptr + i );
                }
                cout << endl << endl << endl ;
                 cout << "The values in array are : "<<endl;
                 for(int i=0; i<n ; i++)
                {
                        cout << *( ptr + i ) << endl;
                }
                delete [] ptr;        //deallocate memory pointed by ptr
                getch();
                }
```

```
//Dynamically allocating memory for object:Example Program
#include <iostream>
#include <conio>

class Student {
                char name[50];
                 int roll;

  public:
                void setInfo()
                  {
                cout<< "Enter name of student : ";
                  cin >>name;
                  cout << "Enter roll of student : ";
                  cin >>roll;
                 }
                void display()
                {
                cout << endl<<"Name= " << name << endl;
                 cout << endl<<"Roll= " << roll << endl;
                }

};
```

//Program continued.

```
int main() {

        Student *ptr1 = new Student; // dynamically declare Student object
        cout<<endl<<"Enter information of student :"<<endl;
        ptr1->setInfo();   // call setInfo() function

        cout<<endl<<"Information of student :"<<endl;
        ptr1->display();   // call display() function


        // pointed memory is released
        delete ptr1;

        getch();
        return 0;
}
```

//Dynamically allocating memory for array of objects: Example Program

```
#include<iostream.h>
#include<conio.h>

class Student {
        int roll;
        char name[50];
        public:
        void setInfo() {
                cout<<"Enter name :";
                cin>>name;
                cout<<"Enter roll number :";
                cin>>roll;
                cout<<endl<<endl;
        }

        void getInfo() {
                cout << "Name        = "<< name <<endl;
                cout << "Roll number = " << roll << endl<<endl;
        }
};
```

//Program continued.

```
int main()
{
        int n;
        cout<<"Enter number of students :";
        cin>>n;
        Student* ptr = new Student[n];     // dynamically declare Student object array
        for(int i=0;i<n;i++)
        {
                (ptr+i)->setInfo();   // accessing class member through pointer, calling setInfo() function
        }
        cout<<endl<<endl<< "Information of Students are : "<<endl;
        for(int i=0;i<n;i++)
        {
                (ptr+i)->getInfo();       // accessing class member through pointer ,calling getInfo() function
        }

        delete []ptr;  // block of pointed memory is released

  getch();
  return 0;
}
```

Note:

- When new operator is used to allocate memory for a class’s object, the object’s constructor is called after the memory is allocated.
- When delete operator is used to deallocate memory for a class’s object, the object’s destructor is called before the objects memory is de allocated.

Example Program:

```
#include <iostream>
#include <conio>

class Test {
        int a,b;
        public:
                Test()
                {
                a=0;
                b=0;
                }
                Test(int x,int y)
                {
                a=x;
                b=y;
                }
void display()
        {
         cout << endl<<"a= " << a << endl;
        cout << endl<<"b= " << b << endl;
        }
  ~Test()
        {
        cout<<endl<<endl<<"Hey this is a
        destructor "<<endl<<endl;
        }
};
```

//Program Continued.

```
int main() {

        // dynamically declare Test objects
        Test *ptr1 = new Test;              // allocates memory for Test object and calls default constructor
        Test *ptr2 = new Test(100,200);     //allocates memory for Test object and calls parameterized consructor

        cout<<endl<<"Information of first object"<<endl;
        ptr1->display();
        cout<<endl<<"Information of second object"<<endl;
        ptr2->display();
   delete ptr1; //address pointed by ptr1 is released
   delete ptr2; //address pointed by ptr2 is released

   getch();
   return 0;
}
```

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Chapter 4: Object Inheritance and Reusability

Inheritance:

- Inheritance allows us to create new class from already existing class
- Inheritance is a process where the child class acquires properties and functionality of its parent class
- The Class that inherits another class is referred as child class, derived class or subclass
- The already existing class through which new class is inherited is referred as parent class, base class or super class

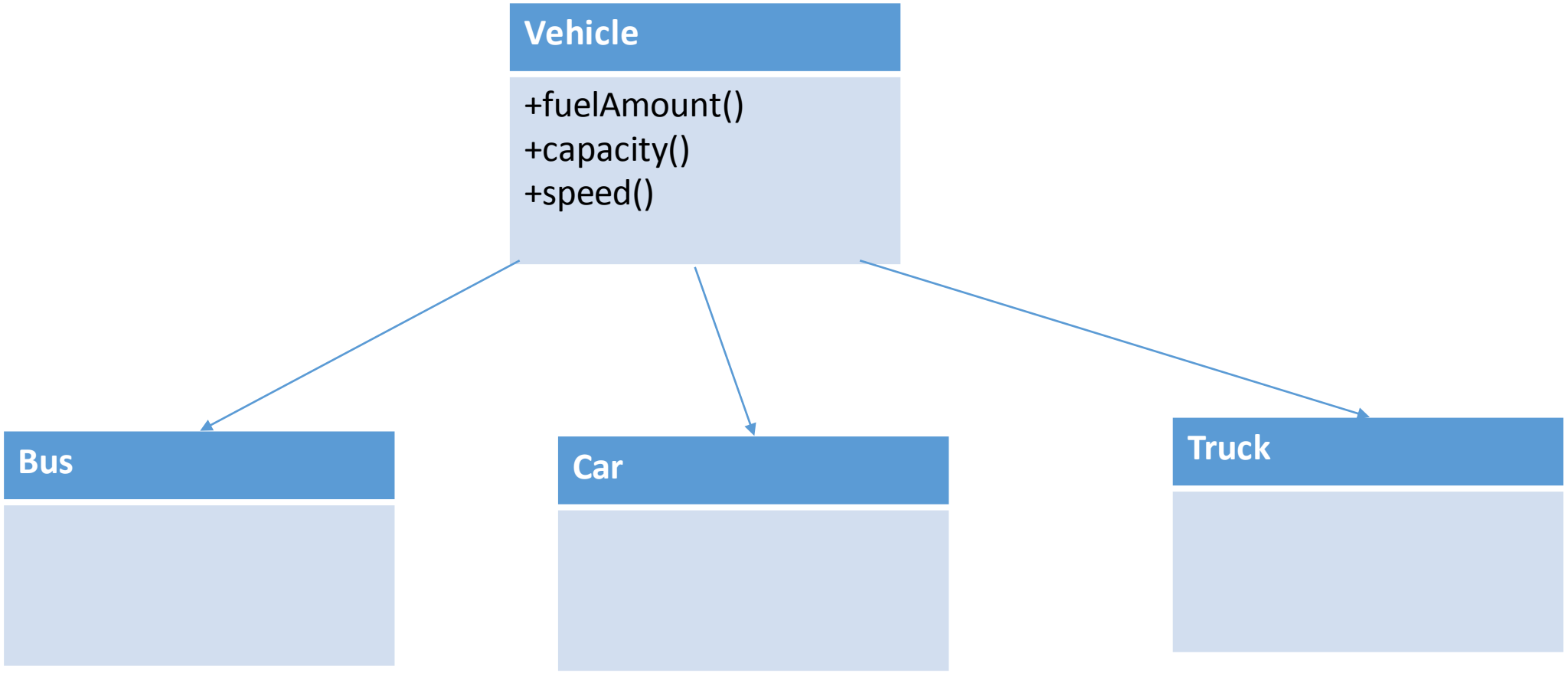
Without using inheritance:

Bus
+fuelAmount() +capacity() +speed()

Car
+fuelAmount() +capacity() +speed()

Truck
+fuelAmount() +capacity() +speed()

Using inheritance:



Inheritance:

Syntax for defining a derived class is :

```
class baseClassName
        {
        …       //code for base class
        };
class derivedClassName : visibility_mode baseClassName
        {
        …        //code specific for derived class
        };
```

Inheritance:

Example:

```
class ABC
{
        code for base class ABC
};
class XYZ : public ABC
{
        code specific for derived class XYZ
};
```

Visibility modes/ Inheritance mode:

Visibility modes specifies whether the features of the base class is derived in private, public or protected mode

Types:

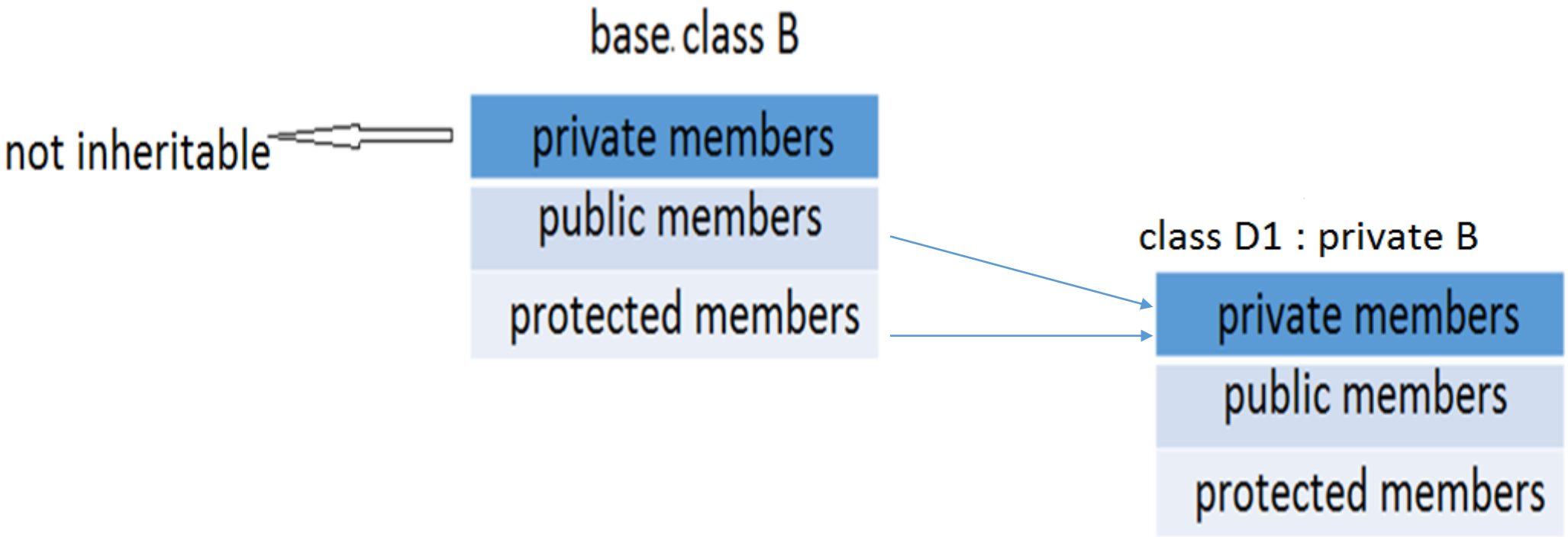
1. Private mode
2. Public mode
3. Protected mode

1. Private mode:

- When the base class is inherited by derived class in private mode, both the public member and protected members of base class will become private member in derived class.

Note: The private members of base class are not directly accessible in derived class however, they can be accessed using public or protected methods of base class in the derived class.

1. Private mode:

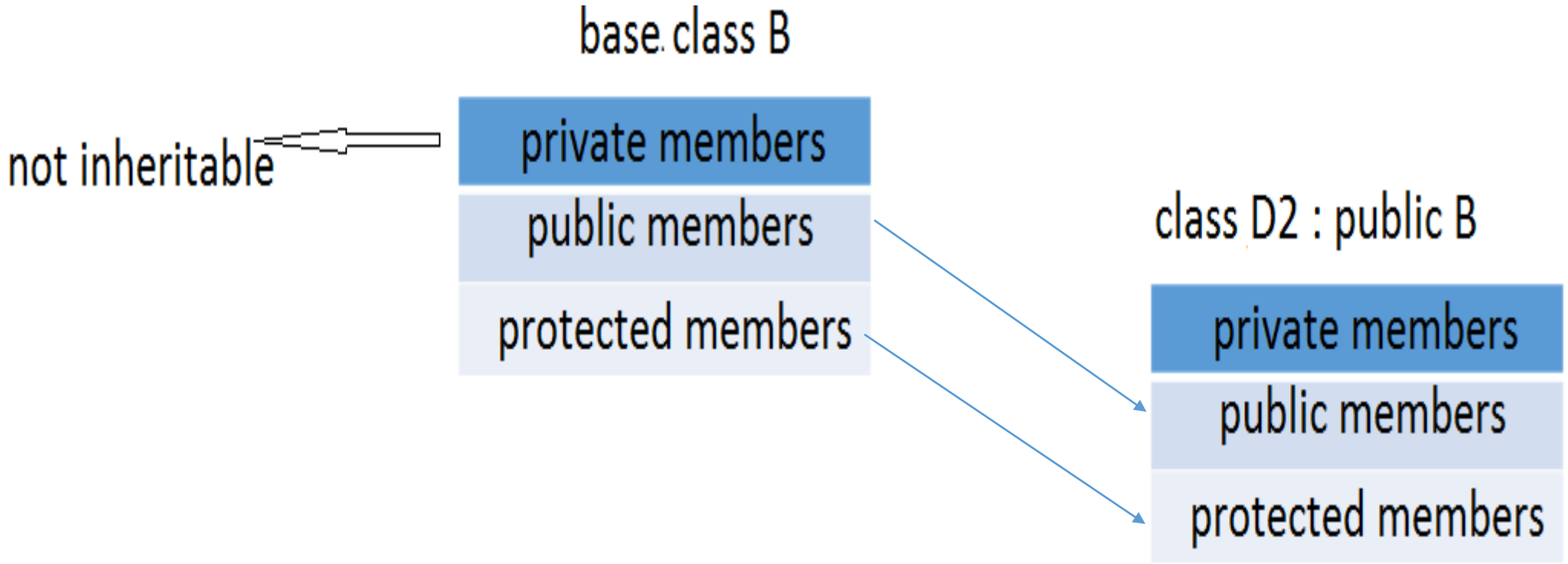


2. Public mode:

- When the base class is inherited by derived class in public mode, public members of base class will become public members and protected members will become protected members for derived class

Note: The private members of base class are not directly accessible in derived class however, they can be accessed using public or protected methods of base class in the derived class.

2. Public mode:



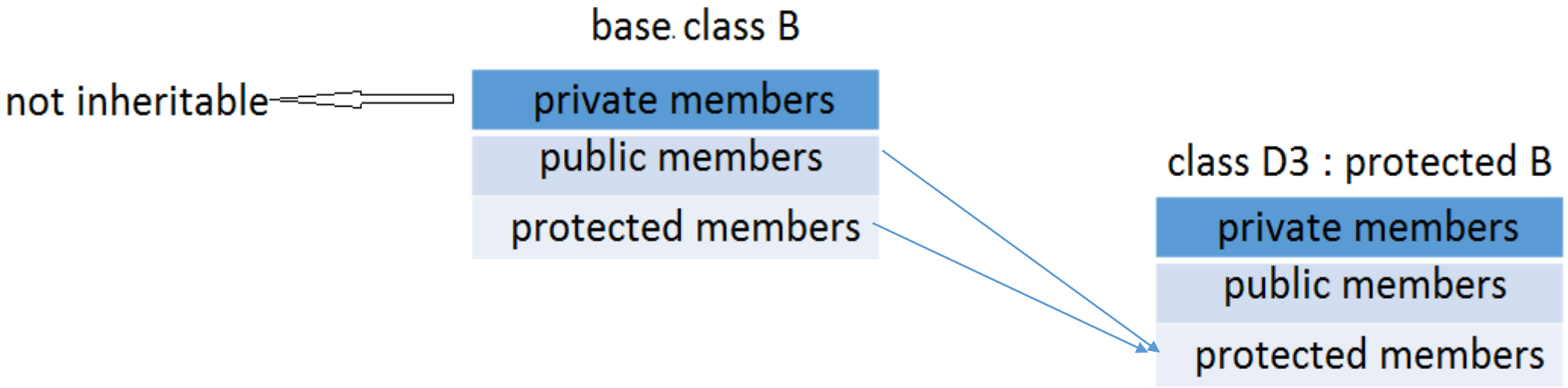
3. Protected mode:

- When the base class is inherited by derived class in protected mode, public members and protected members of base class will become protected members for derived class

Note: The private members of base class are not directly accessible in derived class however, they can be accessed using public or protected methods of base class in the derived class.

Note: Protected members can be accessed within same class and its derived classes only

3. Protected mode:



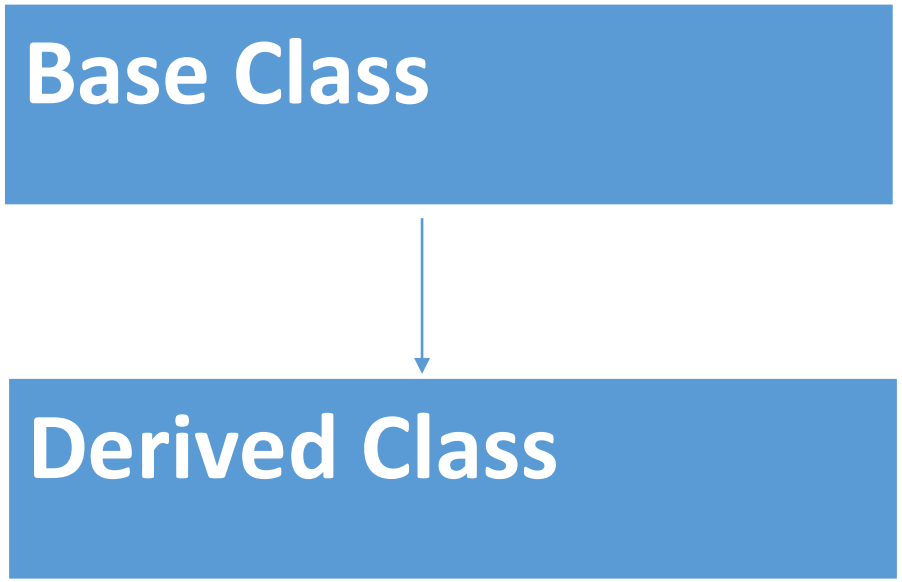
Types of Inheritance:

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance
5. Hybrid inheritance

1. Single Inheritance:

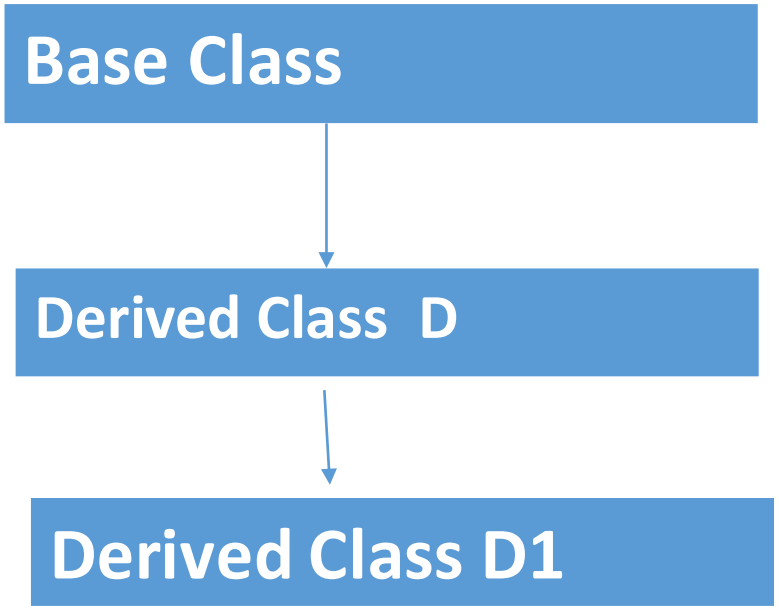
- When a single child class is being inherited by a single parent class, it is called single inheritance
- i.e. one derived class with only one base class

1. Single Inheritance:



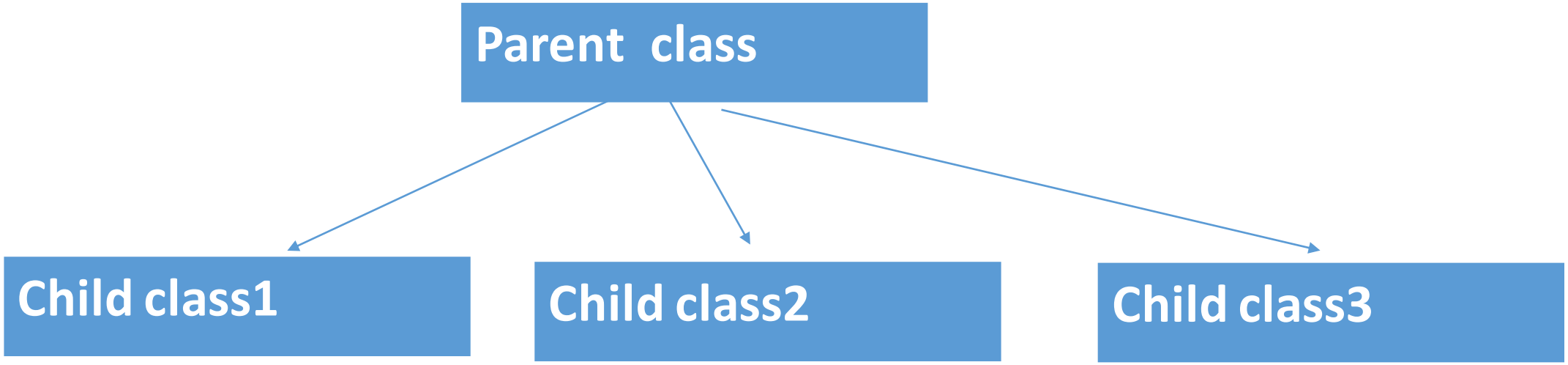
2. Multilevel Inheritance:

- In multilevel inheritance, a class is derived from another derived class.
- The base class of a derived class is derived class of another base class



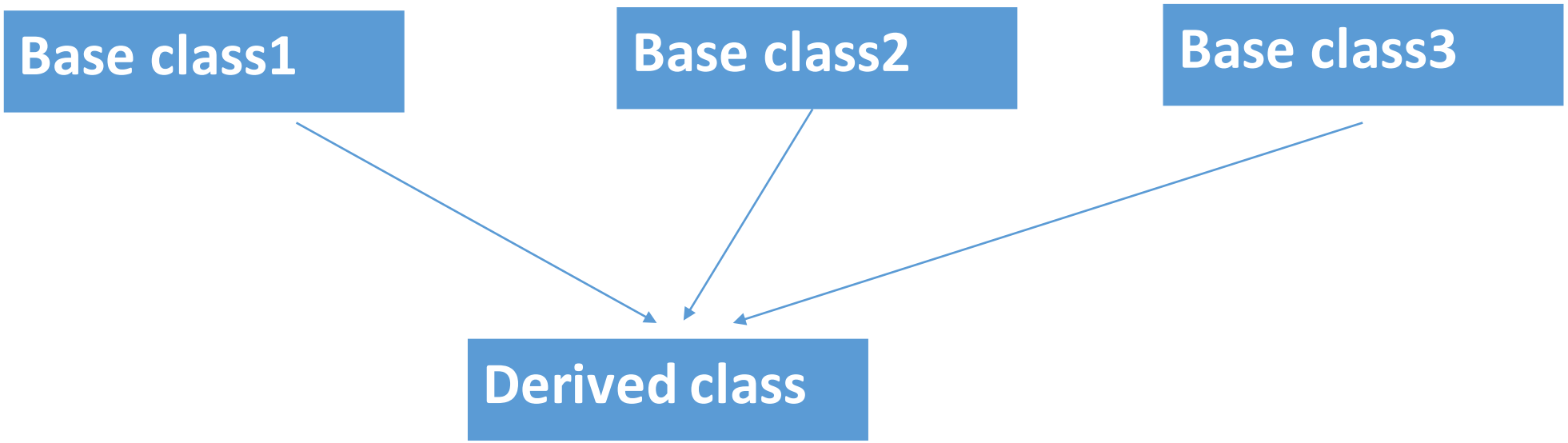
3. Hierarchical Inheritance:

- More than one child class is inherited from a base class in hierarchical inheritance



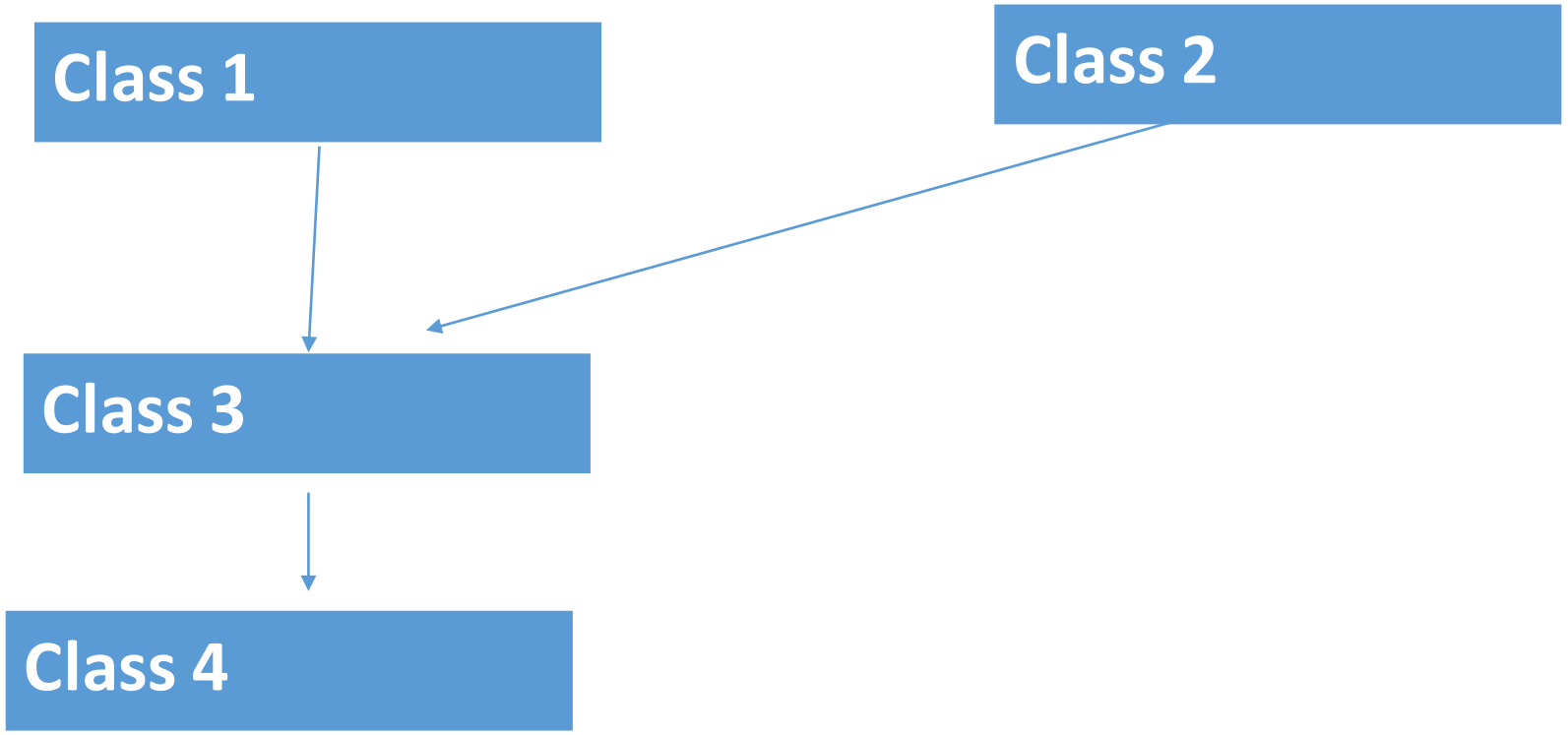
4. Multiple Inheritance:

- In this type of inheritance a child class is derived from more than one base class



5. Hybrid Inheritance:

- The inheritance which involves more than one form of inheritance is called hybrid inheritance



1. Single Inheritance:

- When a single child class is being inherited by a single parent class, it is called single inheritance
- i.e. one derived class with only one base class

Syntax to define a derived class using single inheritance:

```
class Derived_Class_Name : Inheritance_Mode Base_Class_Name
{
        class body
};
```

Example Program : using public inheritance mode

```
#include<iostream.h>
#include<conio.h>



class ABC
{
        private:
                int a;
         protected:
                int b;
        public:
                int c;
                void setValue();
                void displayValue();
                int getA()
                {
                        return a;
                }
};
```

```
class XYZ: public ABC
{
        private:
                int x;
        public :
                void calculate();
                void displayResult();
};
```

```
void ABC :: setValue()
{
        cout << "Enter value of a, b and c:";
        cin >> a >> b >> c;
}
void ABC :: displayValue()
{
        cout << "Value of a : "<<a <<endl ;
        cout << "Value of b : "<<b<<endl;
        cout << "Value of c : "<<c <<endl;
}
```

```
void XYZ :: calculate()
{
        x = getA() + b + c;
}

void XYZ :: displayResult()
{
        cout<<endl<<"Result  after addition is  : "<< x <<endl;
}
```

```
void main()
{
        XYZ obj;
        obj.setValue();
        obj.displayValue();
        obj.calculate();
        obj.displayResult();

                // obj.b =200;  //protected members are not accessible outside class.
        obj.c =100;    //public members are accessible outside class
        cout << endl <<"After updating the value of public member c :"<<endl;
        obj. displayValue();

getch();
}
```

Example Program : using private inheritance mode

```
#include<iostream.h>
#include<conio.h>



class ABC
{
        private:
                int a;
         protected:
                int b;
        public:
                int c;
                void setValue();
                void displayValue();
                int getA()
                {
                        return a;
                }
};
```

```
class XYZ: private ABC        // replace visibility mode with  protected
{
        private:
                int x;
        public :
                void calculate();
                void displayResult();
};
```

```
void ABC :: setValue()
{
        cout << "Enter value of a, b and c:";
        cin >> a >> b >> c;
}
void ABC :: displayValue()
{
        cout << "Value of a : "<<a <<endl ;
        cout << "Value of b : "<<b<<endl;
        cout << "Value of c : "<<c <<endl;
}
```

```
void XYZ :: calculate()
{
        setValue();          //accessing setValue() of Base class ABC
        x = getA() + b + c;
}

void XYZ :: displayResult()
{
        displayValue();     //accessing displayValue() of Base class ABC
        cout<<endl<<"Result after addition is  : "<< x <<endl;
}
```

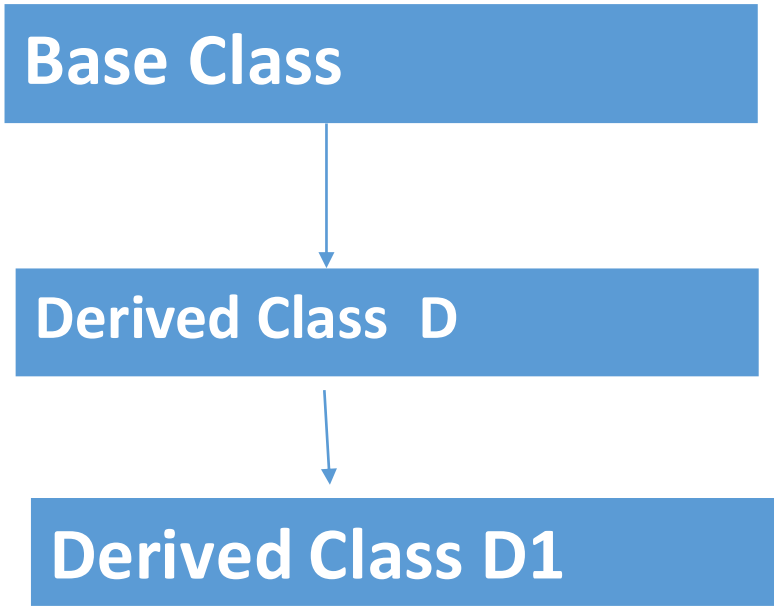
```
void main()
{
        XYZ obj;

        /* obj.setValue();
        obj.displayValue();          cannot be accessed outside class as XYZ has inherited ABC's properties
in private mode              */


        obj.calculate();
        obj.displayResult();

                // obj.b =200;  //private members are not accessible outside class.
                  //obj.c =100;    //private members are accessible outside class

getch();
}
```

Practice:

Q1. Write a program to create a class named Person which has name and age as data members and member functions to read and display its data. Create another class Student derived from class Person to use the features of base class.

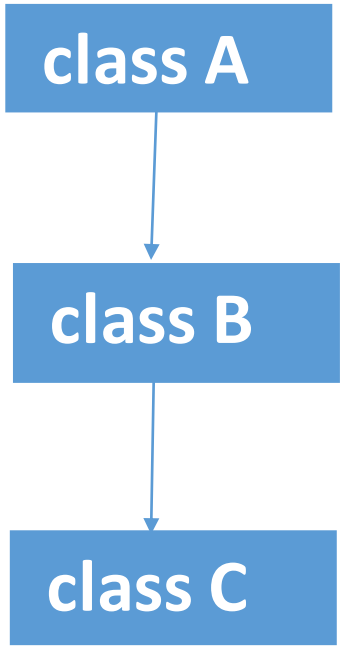
2. Multilevel Inheritance:

- In multilevel inheritance, a class is derived from another derived class.
- The base class of a derived class is derived class of another base class



2. Multilevel Inheritance:

```
class A
        {
                body of class A
        };
class B : visibility_mode A
        {
                body of class B
        };
class C : visibility_mode B
        {
                body of class C
        };
```



Example Program:

```
#include <iostream.h>
#include <conio.h>
class Base
{
        protected:
                int a;
        public:
                void setData ()
                {
                        cout << "Enter the value of a = ";
                        cin >> a;
                }
};
```

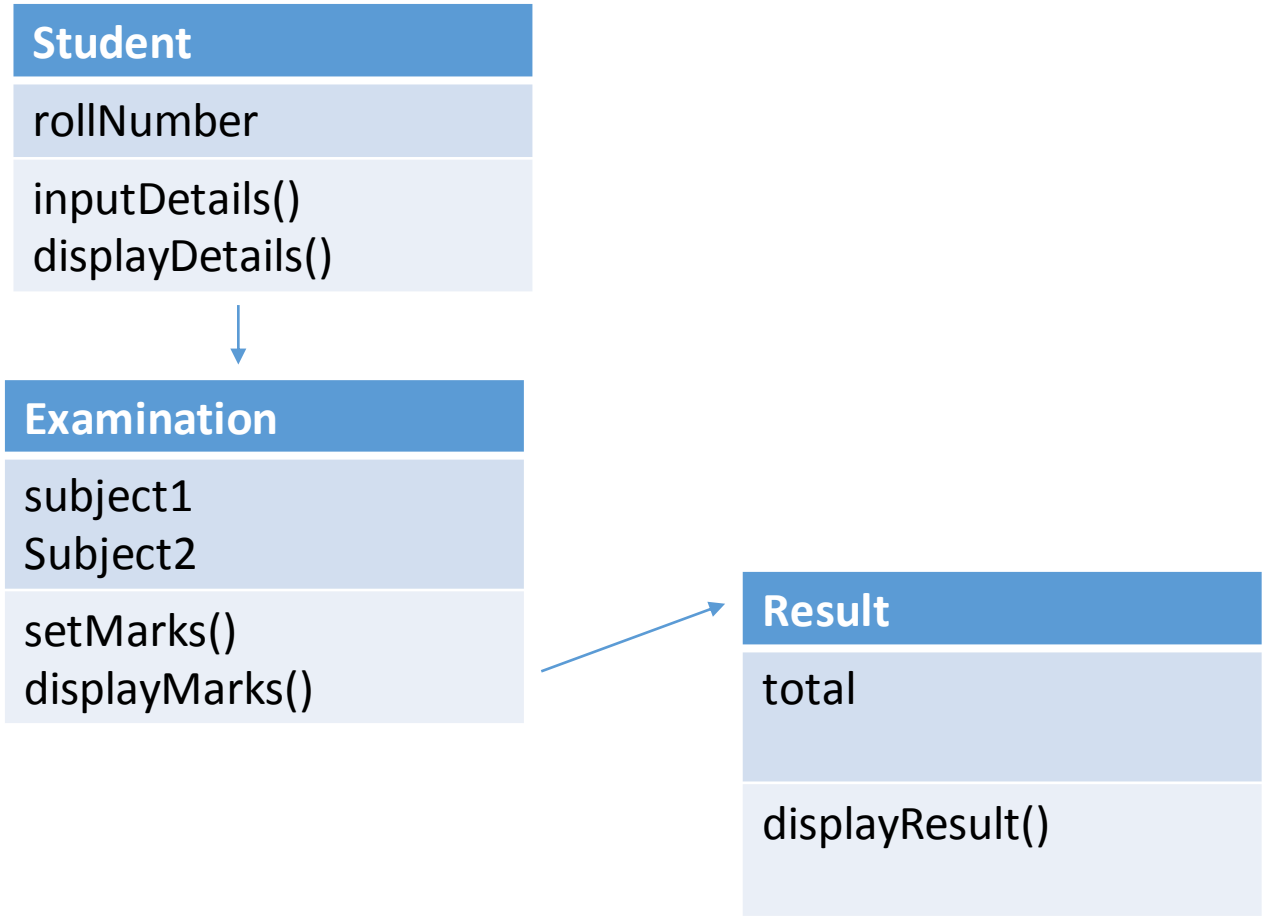
```
class DerivedOne : public Base
{
        protected:
                int b;
        public:
                void readData ()
                {
                        cout << "Enter the value of b = ";
                        cin >> b;
                }
};
```

```
class DerivedTwo : public DerivedOne
{
        private:
                int c;
        public:
                void input()
                {
                        cout << "Enter the value of c = ";
                         cin >> c;
                }
                void product()
                {
                        cout << "Product = " << a * b * c;      //accessing members a and b of base class
                }
};
```

```
int main ()
{
        DerivedTwo obj;
        obj.setData();       //accessing member of Base class
        obj.readData();    //accessing member of DerivedOne
        obj.input();
        obj.product();
        getch();
        return 0;
}
```

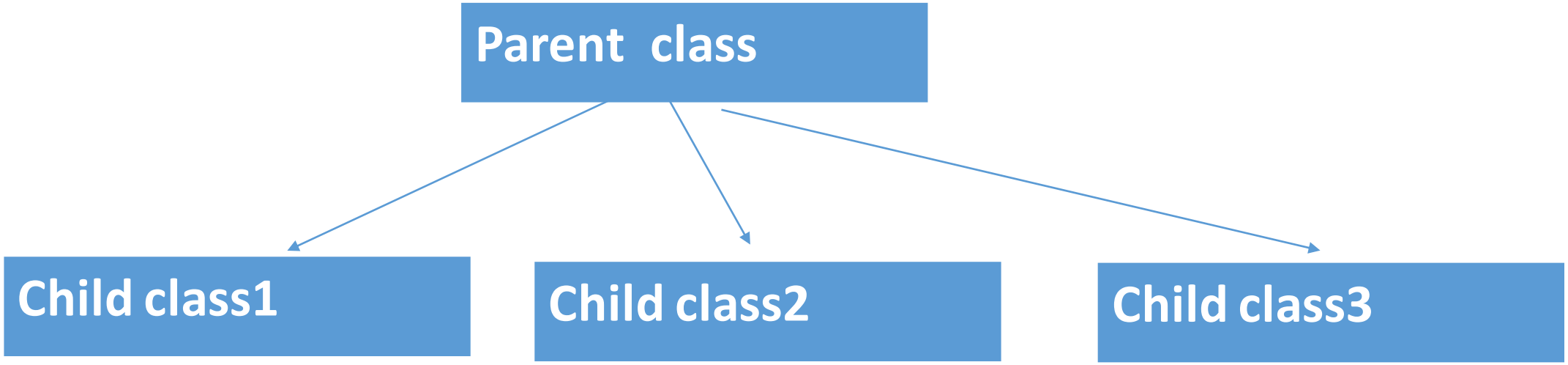
Practice:

Q1. Define a class Student which has roll number and member functions to input and display roll number. Derive a class Examination from class Student which has marks of two subjects and member functions to initialize and display marks. Again derive a class Result from class Examination and calculate total and display the result.

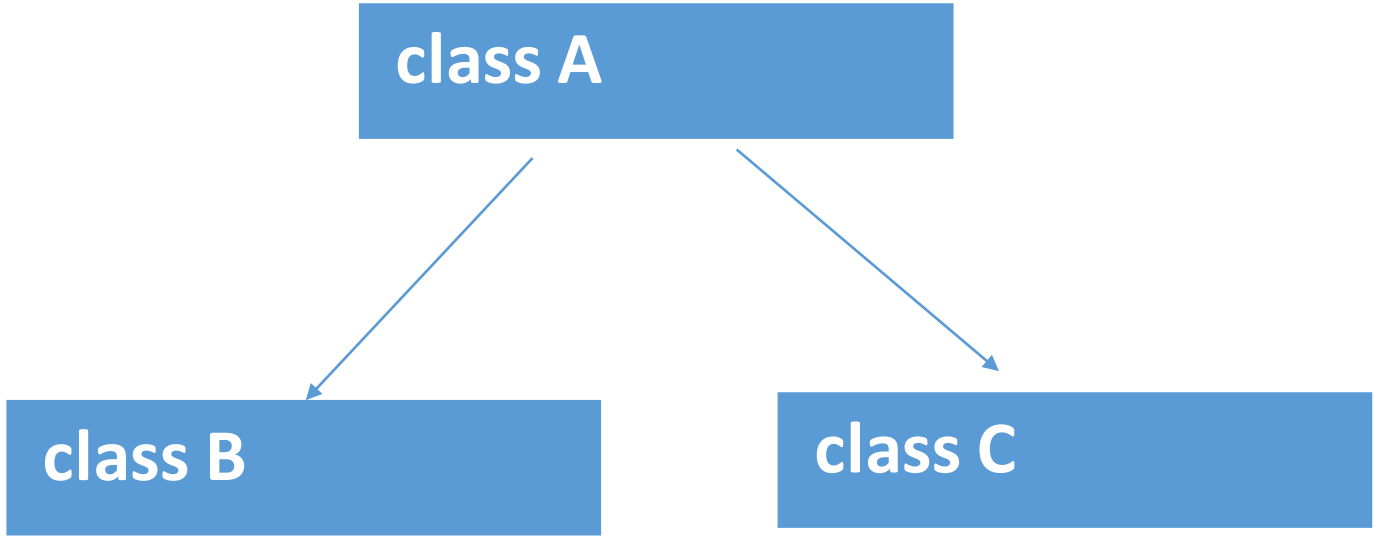


3. Hierarchical Inheritance:

- More than one child class is inherited from a base class in hierarchical inheritance



```
class A
        {
                ...
        };
class B : visibility_mode A
        {
                ...
        };
class C : visibility_mode A
        {
                ...
        }
```



Example Program:

```
#include <iostream.h>
#include <conio.h>
class Base          //single base class
{
  protected:
                    int a,b;
          public:
                    void setData ()
                    {
                    cout << "Enter the value of a : ";
                    cin >> a;
                    cout << "Enter the value of b : ";
                    cin >>b;
                    }
};
```

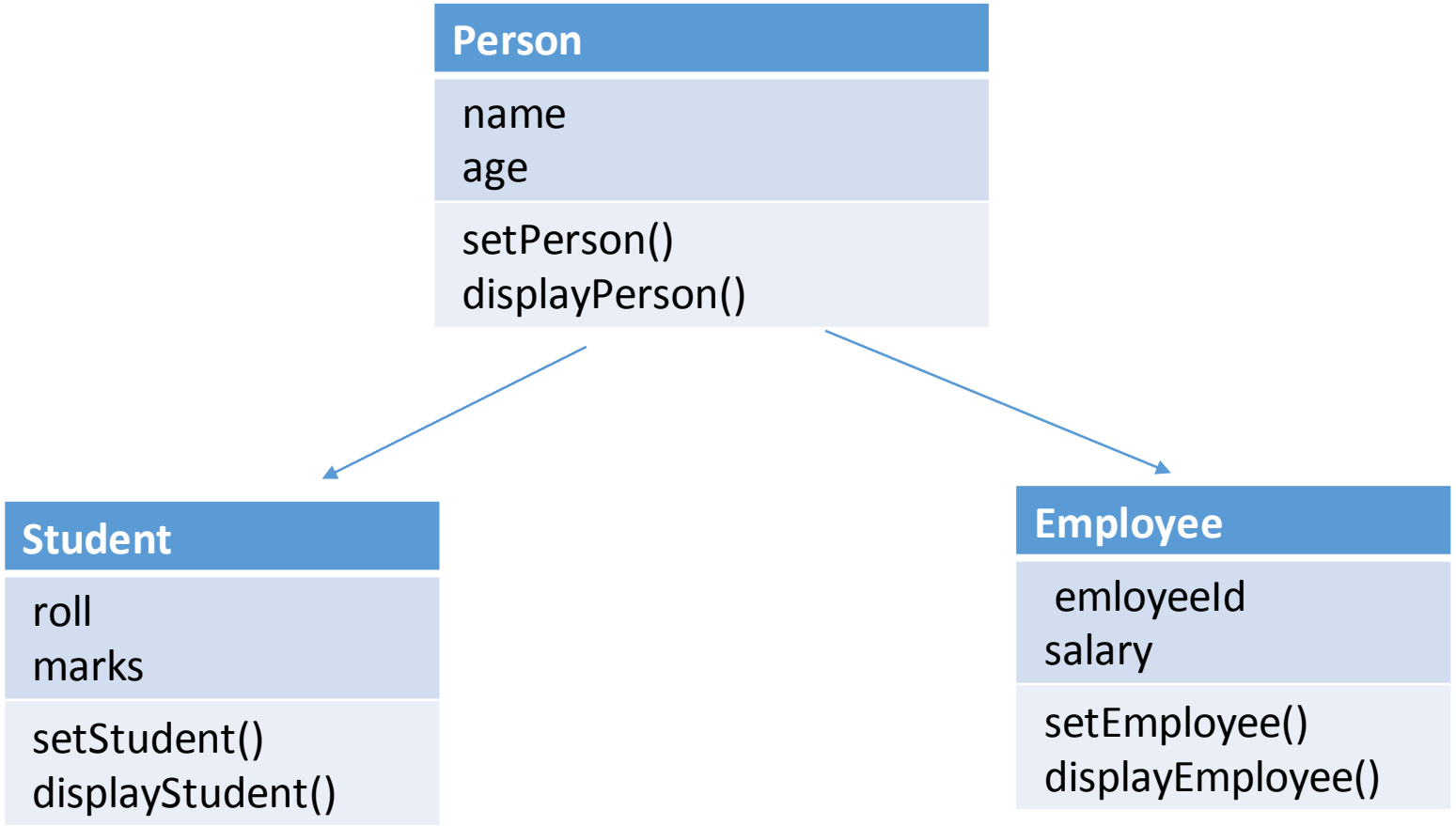
```
class DerivedOne : public Base        //DerivedOne is derived from class Base
{
        protected:
                int c;
        public:

                void add()
                {
                        c=a+b;                          //access the member of base class
                        cout << a << "+" << b <<"="<<c<<endl;
                }
};
```

```
class DerivedTwo : public Base            //DerivedTwo is also derived from class Base
{
        private:
                int c;
        public:

                void product()
                {
                        c=a*b;
                        cout << a < < "*" << b <<"="<<c<<endl;
                }
};
```

```
int main ()
{
        DerivedOne obj1;   //Object of DerivedOne class
        DerivedTwo obj2;    //Object of DerivedTwo class

        obj1.setData();         // call member function of Base
        obj1.add();

        obj2.setData();         // call member function of Base
        obj2.product();

        getch();
        return 0;
}
```

Practice:

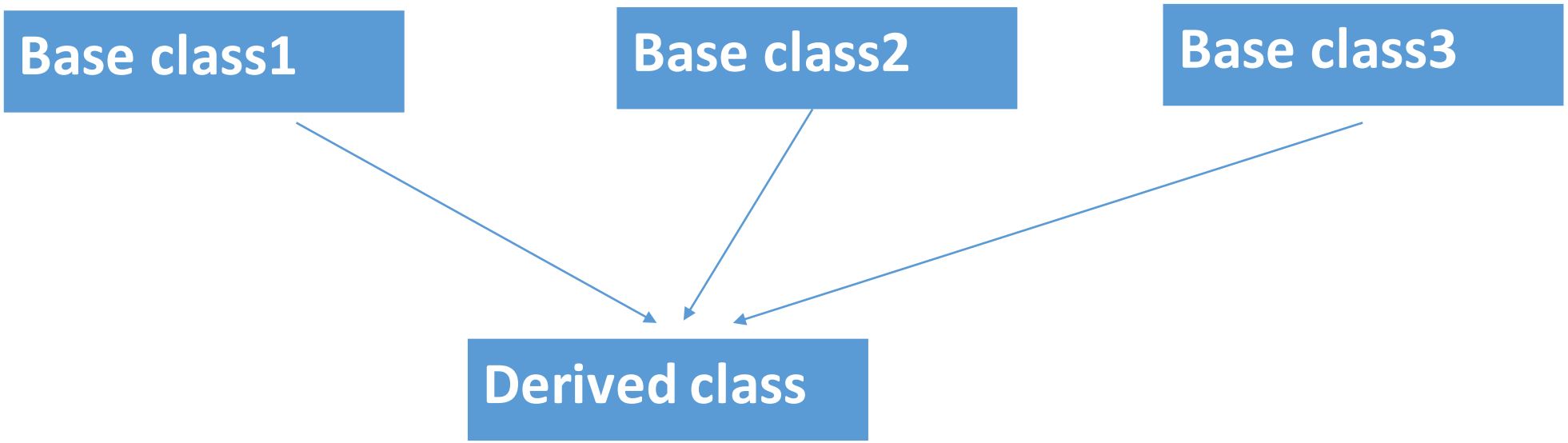
Q1. Write a program to use hierarchical inheritance.



4. Multiple Inheritance:

- In this type of inheritance a child class is derived from more than one base class

i.e. a child class inherits properties of two or more base classes.

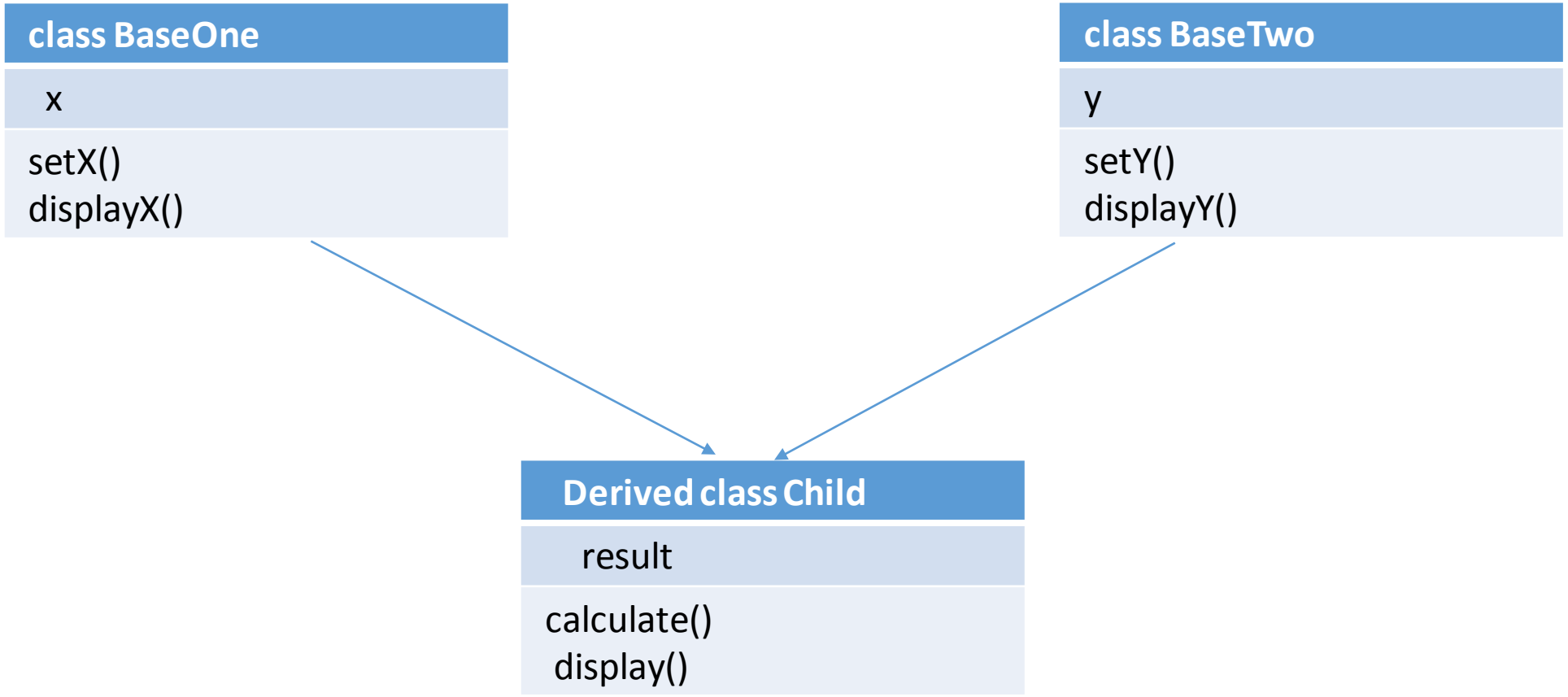


4. Multiple Inheritance:

Syntax to define a derived class using multiple inheritance:

```
class Base_Class_One
        {
        …
        };
class Base_Class_Two
        {
                …
        };
class Derived_Class : visibility_mode Base_Class_One , visibility_mode Base_Class_Two
        {
          …
        };
```

Example :


class BaseOne
x
setX()
displayX()
class BaseTwo
y
setY()
displayY()
Derived class Child
result
calculate()
display()

```
#include <iostream.h>
#include <conio.h>

class BaseOne
{
        protected :
                int x;
        public :
                void setX();
                void displayX();
};

class BaseTwo
{
        protected :
                int y;
        public :
                void setY();
                void displayY();
};
```

```
void BaseOne:: setX()
{
        cout << "Enter value of x :";
         cin >> x ;
}

void BaseOne :: displayX()
{
        cout << endl <<"The value of x is : " <<x <<endl;
}
void BaseTwo:: setY()
{
        cout << "Enter value of y :";
        cin >> y ;
}

void BaseTwo :: displayY()
{
        cout << endl <<"The value of y is : " <<y <<endl;
}
```

```
class Child : public BaseTwo, public BaseOne
{
        private:
        int result;
  public:
        void calculate();
        void display();
};
```

```
void Child :: calculate()
{
        result = x + y;        //accessing x and y inherited from its base classes
}
void Child :: display()
{
        cout << endl <<"The result after addition is :"<<result <<endl;
}

void main()
{
        Child obj;
        obj.setX();
        obj.setY();
        obj.displayX();
        obj.displayY();
        obj.calculate();
        obj.display();

  getch();
}
```

Ambiguity in Multiple Inheritance:

- Ambiguity arises in multiple inheritance when more than one base class have same function name which is not overridden in derived class.
- If we try to call the function using the object of the derived class, compiler shows error because compiler doesn't know which function to call.

Ambiguity in Multiple Inheritance: Example program

```
#include <iostream.h>
#include <conio.h>

class BaseOne
{
        protected :
                int x;
        public :
                void set();
                void display();
};

class BaseTwo
{
        protected :
                int y;
        public :
                void set();
                void display();
};
```

```
void BaseOne:: set()
{
        cout << "Enter value of x :";
        cin >> x ;
}

void BaseOne :: display()
{
        cout << endl <<"The value of x is : " <<x <<endl;
}
void BaseTwo:: set()
{
        cout << "Enter value of y :";
        cin >> y ;
}
```

```
void BaseTwo :: display()
{
        cout << endl <<"The value of y is : " <<y <<endl;
}

void Child :: calculate()
{
        result = x + y;
}
void Child :: displayResult()
{
        cout << endl <<"The result after addition is :"<<result <<endl;
}
```

```
class Child : public BaseTwo, public BaseOne
{
        private:
        int result;
  public:
        void calculate();
        void displayResult();
};
```

```
void main()
{
        Child obj;
        obj.set();
                        /*compiler throws an error as there is ambiguity
                                about which base class's set() to call  */
        obj.set();
        obj.display();
                        / *compiler throws an error as there is ambiguity
                         about which base class's display() to call   */
        obj.display();
        obj.calculate();
        obj.displayResult();

  getch();
}
```

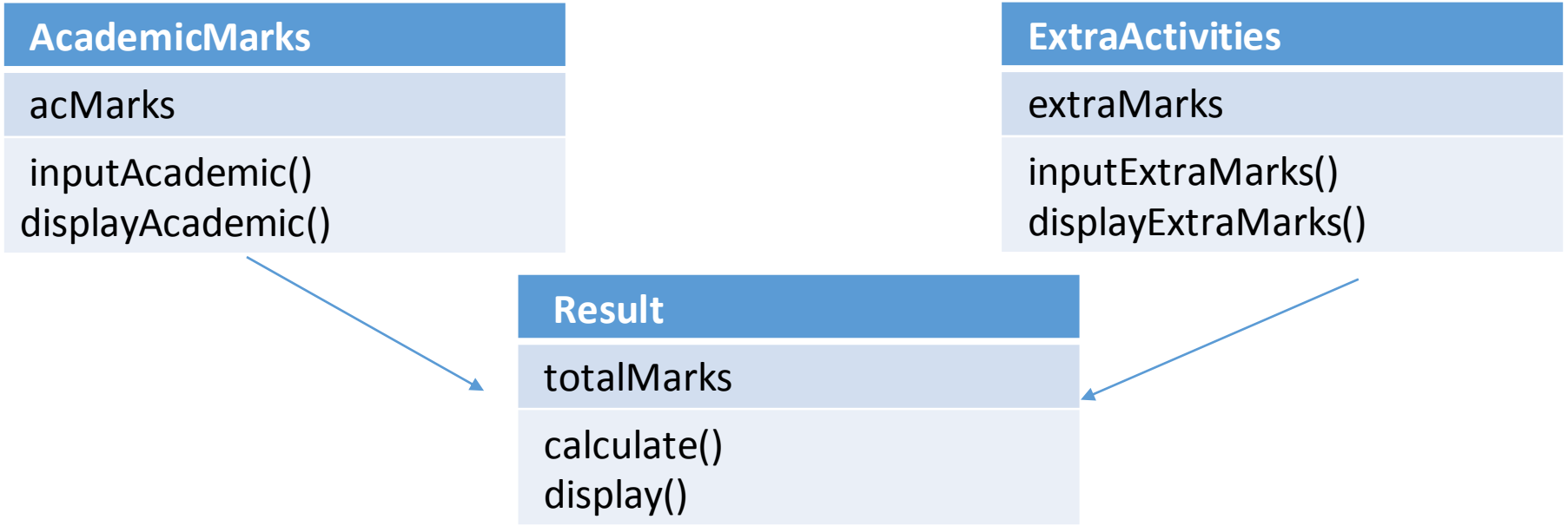
Solution of ambiguity in multiple inheritance:

The ambiguity can be resolved by using the class_name and scope resolution operator (::) to specify the class name of member function.

```
void main()
{
        Child obj;
        obj.BaseOne::set();             //calls BaseOne's set()
        obj.BaseTwo::set();             //calls BaseTwo's set()
        obj.BaseOne::display();         //calls BaseOne's display()
        obj.BaseTwo::display();         //calls BaseOne's display()
        obj.calculate();
        obj.displayResult();

  getch();
}
```

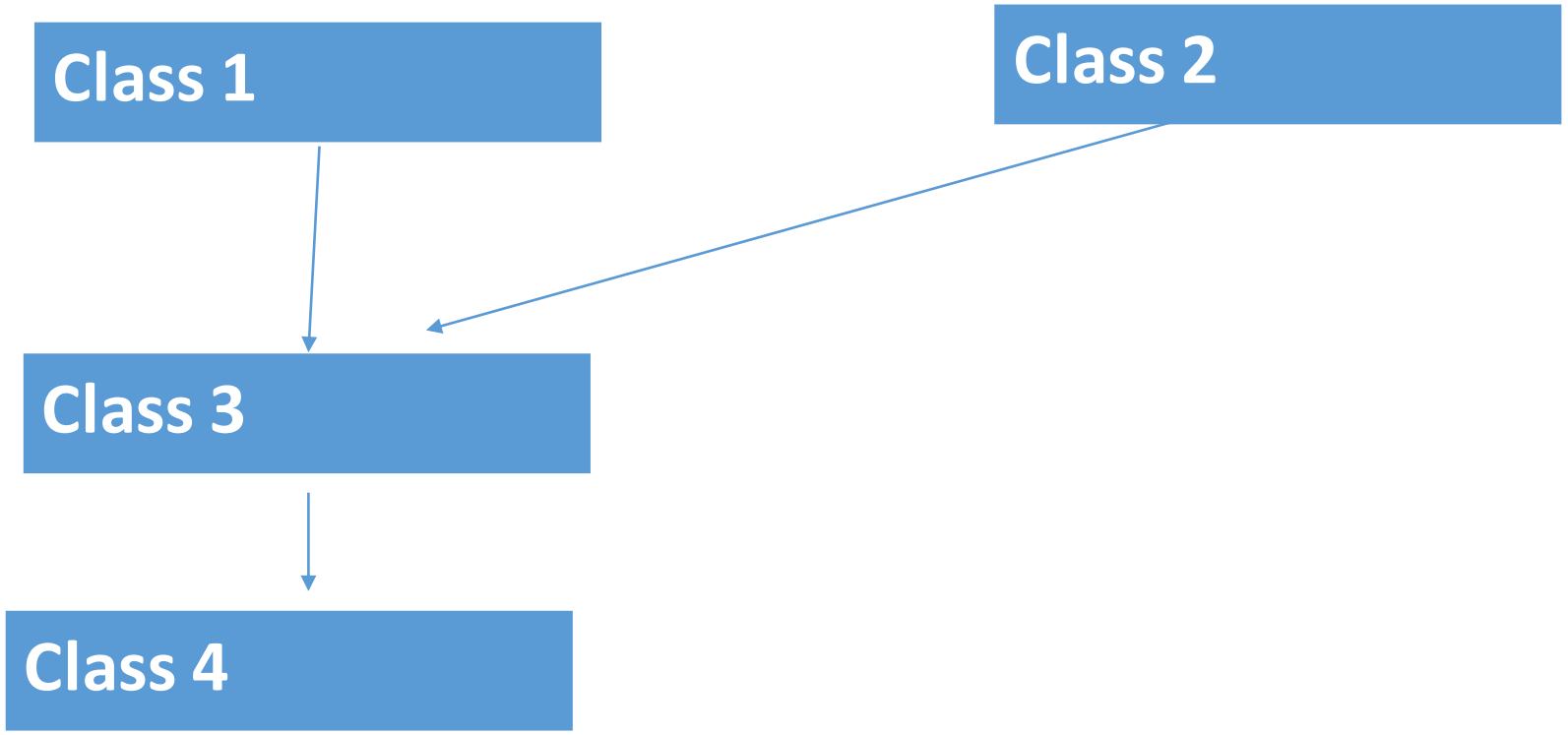
Practice:

Q1. Write a program to define class AcademicMarks and ExtraActivities which have data members for academic marks and extra activities marks respectively. Define member functions to initialize the marks. Derive a class Result from AcademicMarks and ExtraActivities and calculate the total marks obtained.



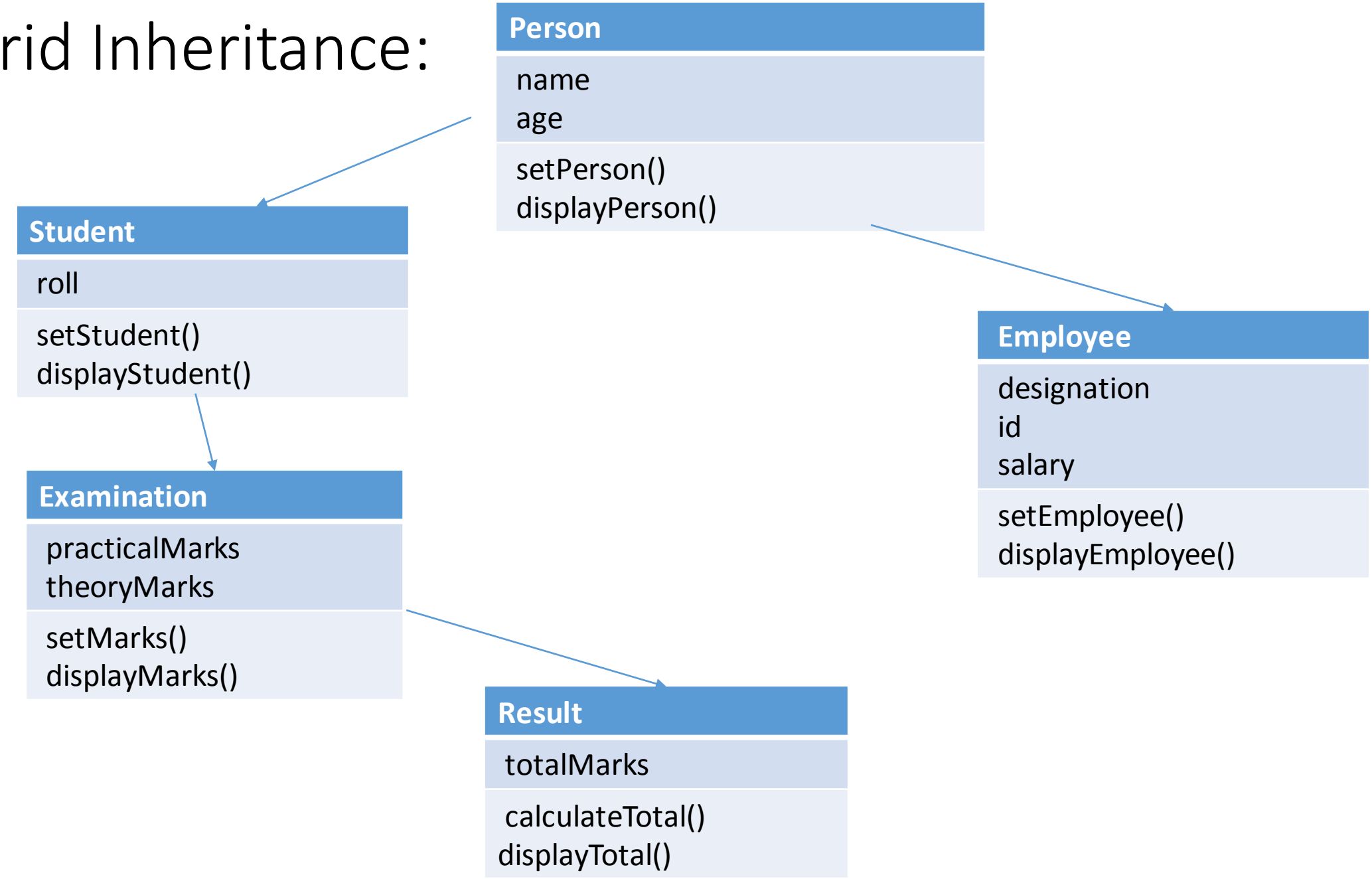
5. Hybrid Inheritance:

- The inheritance which involves more than one form of inheritance is called hybrid inheritance



Hybrid Inheritance:

Q1.



```
#include<iostream.h>
#include<conio.h>
class Person
{
  protected:
                    char name[20];
                     int age;
  public:
          void setPerson()
          {
                    cout << "Enter name : ";
                    cin >> name ;
                    cout << "Enter age : ";
                    cin >> age;
          }
          void displayPerson()
          {
                    cout<< endl << "Name : "<< name <<endl << "Age : "<< age <<endl;
          }
};
```

```
class Employee : public Person
{
        private :
                char designation[20];
                int id;
                int salary;
        public :
                void setEmployee()
                {
                        cout << "Enter designation : ";
                        cin >> designation;
                        cout << "Enter id : ";
                        cin >> id;
                        cout << "Enter salary : " ;
                        cin >> salary;
                }
                void displayEmployee()
                {
                        cout << "Designation :"<<designation << endl << "Id : "<<id <<endl <<"Salary :" << salary <<endl;
                }
};
```

```
class Student : public Person
{
        protected:
                int roll;
        public:
                void setStudent()
                {
                        cout<< endl <<"Enter roll number :";
                          cin >> roll;
                 }
                void displayStudent()
                {
                        cout <<endl <<"Roll number :"<<roll<<endl;
                }
};
```

```
class Examination : public Student
{
        protected:
                float practicalMarks, theoryMarks;
        public:
                void setMarks()
                {
                        cout << "Enter practical marks :";
                        cin >> practicalMarks;
                        cout << "Enter theory marks :";
                        cin >> theoryMarks;
                }
                void displayMarks()
                {
                        cout << "Practical Marks :"<<practicalMarks << endl << "Theory Marks :" <<theoryMarks <<endl;
                }
};
```

```
class Result : public Examination
{
        float totalMarks;
  public:
        void calculateTotal()
        {
                totalMarks = practicalMarks + theoryMarks;
        }
        void displayTotal()
        {
                cout << "Total Marks :"<<totalMarks<<endl;
        }
};
```

```
void main()
{
        Employee e;
         cout <<"Enter Information of Employee "<<endl;
        e.setPerson();
        e.setEmployee();
        cout << endl << endl << "Information of Employee : "<<endl;
        e.displayPerson();
        e.displayEmployee();

        Result s;
        cout << endl << endl << "Enter Information of Student "<<endl;
        s.setPerson();
        s.setStudent();
        s.setMarks();
        cout << endl << endl << "Information of Student "<<endl;
        s.displayPerson();
        s.displayStudent();
        s.displayMarks();
        s.calculateTotal();
        s.displayTotal();

  getch();
}
```

Practical:

Q1. Create a class called **Student** with two data member to represent name and age of the student. Use member function to read and print these data. From this class, derive a class called **Boarder** with data member to represent room number. Derive another class called **DayScholar** from the class Student with data member to represent address and bus number of the student. In both derived classes, use member function to read and print the respective data.

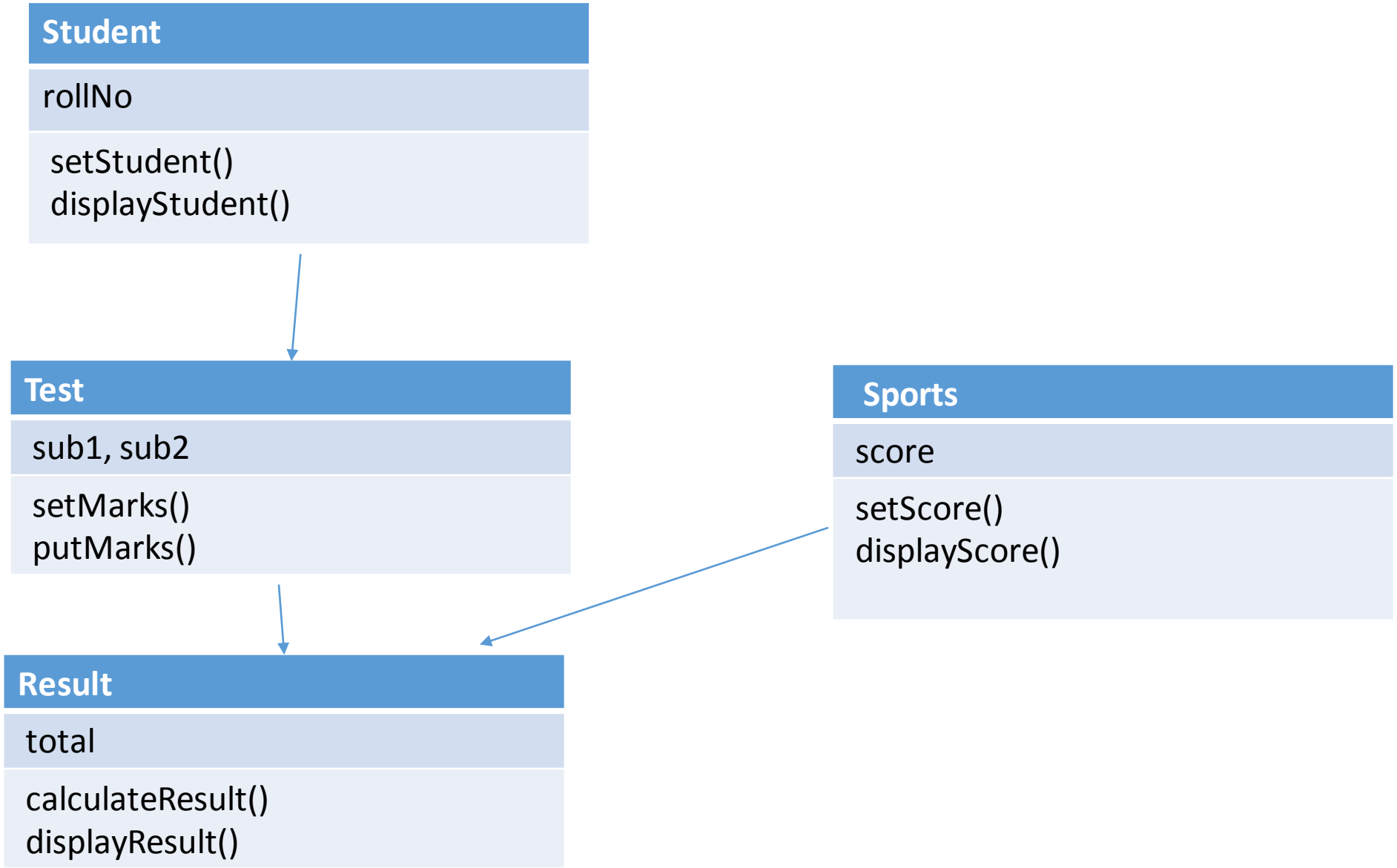
Practical:

Q2. Create a class called Rectangle with data member to represent length, breadth and area. Use appropriate member functions to read length and breadth from user to calculate area of the rectangle. Next create a derived class called Box from the class Rectangle. Use appropriate member function to read height and to calculate the volume of the box.

Practical:

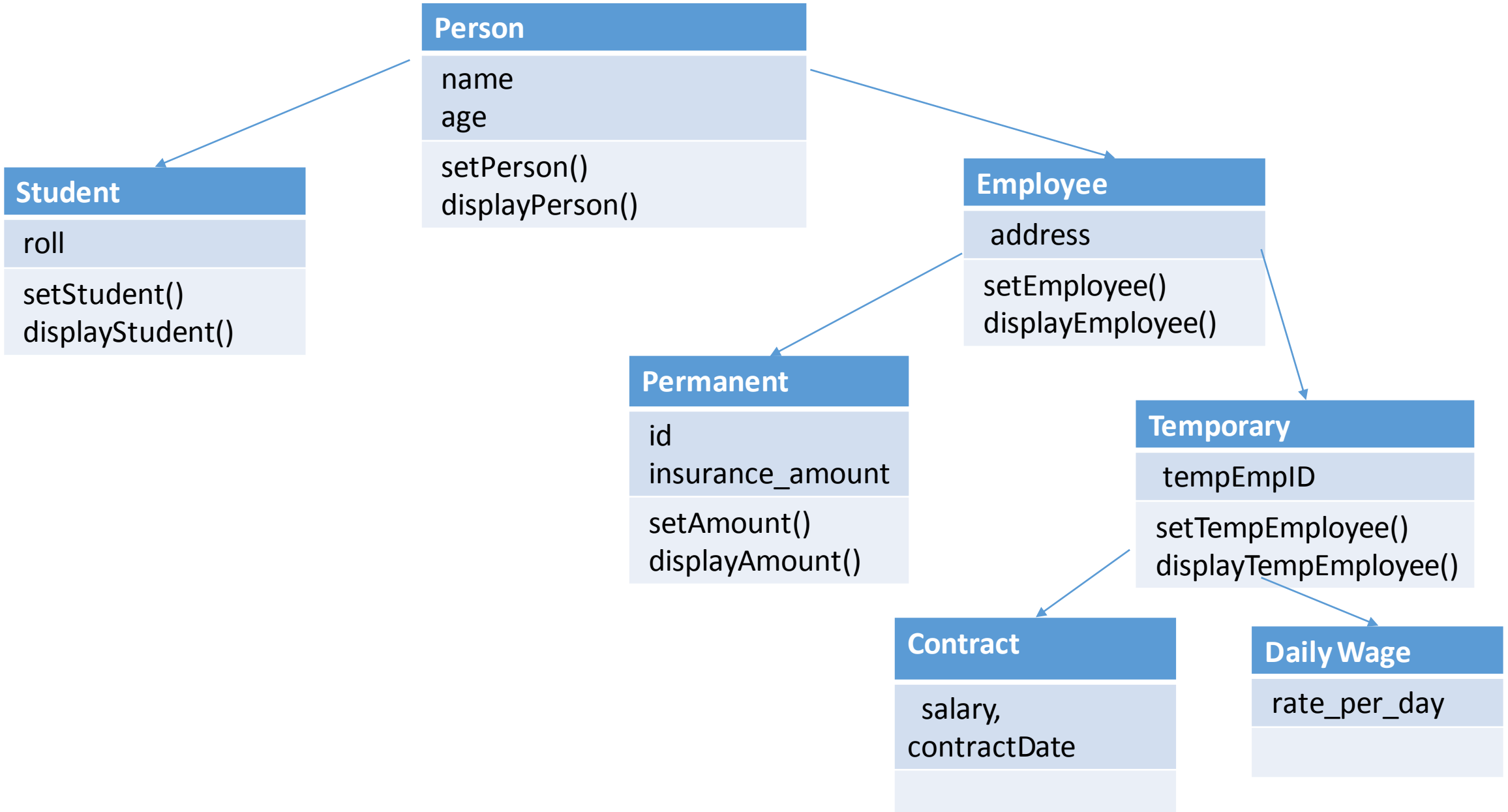
Q3. Write a base class that ask the user to enter a complex number and derived adds the complex number of its own with the base. Finally make third class that is friend of derived and calculate the difference of base complex number and its own complex number.

Q4. Write a program to represent following classes with following data members.



Q5. Create a class **Polygon** with data members to represent two dimensions and appropriate member functions to read its data member. Derive two classes **Rectangle** and **Triangle** from above class. Define member function area() in each derived class to calculate area of corresponding figure.

Q6. Write a program to represent following classes.



Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Chapter 4: Inheritance

Function Overriding:

- Two or more functions having same name and same signature (number, type and sequence of parameters) but one defined in base class and other defined in derived class
- When the function is called with the help of object of derived class, by default derived class gives priority to the function defined within it.
- So base class function is overridden by derived class version

Example program:

```
#include<iostream.h>
#include<conio.h>
class Base
{
        public :
                void display()
                {
                        cout<<"This is base "<<endl;
                }
};

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

void main()
{
        Base bobj;
        cout<<"Calling from Base class's object"<<endl;
        bobj.display();
        Derived dobj;
        cout<<endl<<endl<<"Calling from Derived class's object"<<endl;
        dobj.display();
        dobj.display();
        getch();
}
```

Function Overriding :

- We use base class name and scope resolution operator to call base class version using object of derived class

Example program:

```
#include<iostream.h>
#include<conio.h>
class Base
{
        public :
                void display()
                {
                        cout<<"This  is base "<<endl;
                }
};

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

```
void main()
{
        Base bobj;
        cout<<"Calling from Base class's object"<<endl;
        bobj.display();
        Derived dobj;
        cout<<endl<<endl<<"Calling  from Derived
class's  object"<<endl;
        dobj.Base::display();
        dobj.display();
        getch();
}
```

Function Overriding:

- Two or more functions having same name and same signature (number, type and sequence of parameters) but one defined in base class and other defined in derived class
- In function overriding, the compiler doesn’t know about the correct form of function to be called in compile time. The correct form of function is selected to call on the basis of content of calling object at runtime or while the program is running.
- The code associated with the function call is not known until program execution, hence called late binding/ dynamic binding/ dynamic linkage.

Virtual Function:

- In C++, function overriding is achieved using virtual function.
- A virtual function is a member function which is declared within a base class and overridden by derived class
- When we use same function name with same signature in base class and derived class, the function in base is declared as virtual function
- When the class containing virtual function is inherited, the derived class redefines virtual function to perform task respective to derived class

Virtual Function:

- Base class pointer can point object of same class and also point object of its derived classes
- The correct version of function is called on the basis of type of object pointed by base class pointer at particular time
- By making the base class pointer point to different class’s object, we can execute different version of virtual function
- If the derived class doesn’t redefine virtual function, the derived class inherits its immediate base class virtual function definition
- Base class pointer can point to object of any of its descendant class

Without implementing Virtual Function : Compile and run this program

```
#include<iostream.h>
#include<conio.h>

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

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

```
void main()
{
        Base *bptr, bobj;
        bptr=&bobj;
        cout<<"Calling from Base class's pointer
        that points to Base's object"<<endl;
        bptr->display();

        Derived dobj;
        bptr=&dobj;
        cout<<endl<<endl<<"Calling from Base class's
        pointer that points to Derived's object"<<endl;
        bptr->display();
        getch();
}
```

implementing Virtual Function : Compile and run this program

```
#include<iostream.h>
#include<conio.h>

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

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

```
void main()
{
        Base *bptr, bobj;
        bptr=&bobj;
        cout<<"Calling from Base class's pointer
        that points to Base's object"<<endl;
        bptr->display();

        Derived dobj;
        bptr=&dobj;
        cout<<endl<<endl<<"Calling from Base class's
        pointer that points to Derived's object"<<endl;
        bptr->display();
        getch();
}
```

Working of Virtual Function:

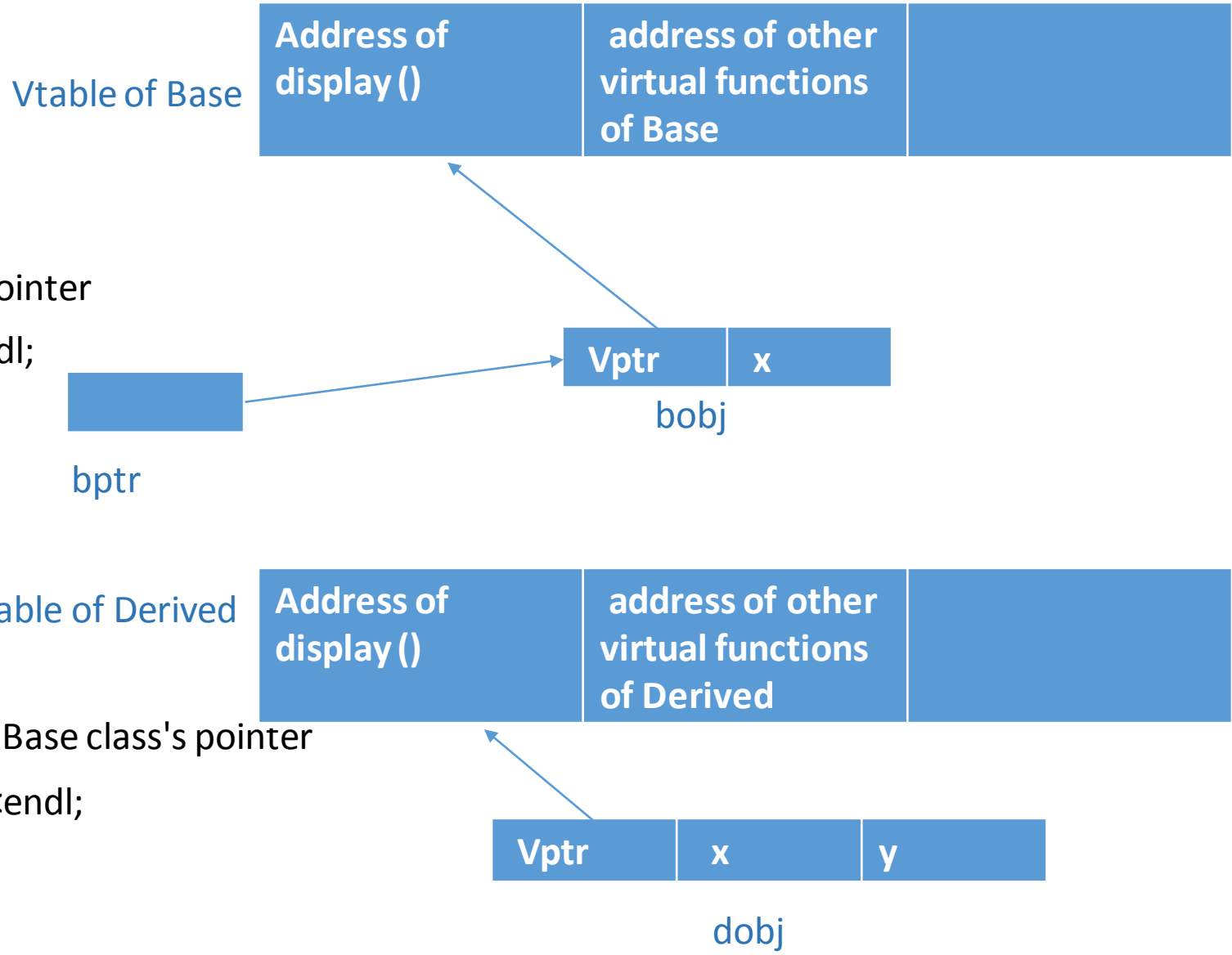
```
class Base{
        int x;     Vptr
        public :
        virtual void display()
        {
        cout<<"This is base "<<endl ;
        }
        void testFunction()
        {
        cout<<"This is Test in base "<<endl;
        }
        ...
};

class Derived: public Base{
        public :
        int y;
        void display()
        {
        cout<<"This is derived "<<endl;
        }
        void testFunction()
        {
        cout<<"This is Test in base "<<endl;
        }
        ...
};
```

Address of display	address of other virtual functions of Base	

Address of display	address of other virtual functions of Derived	

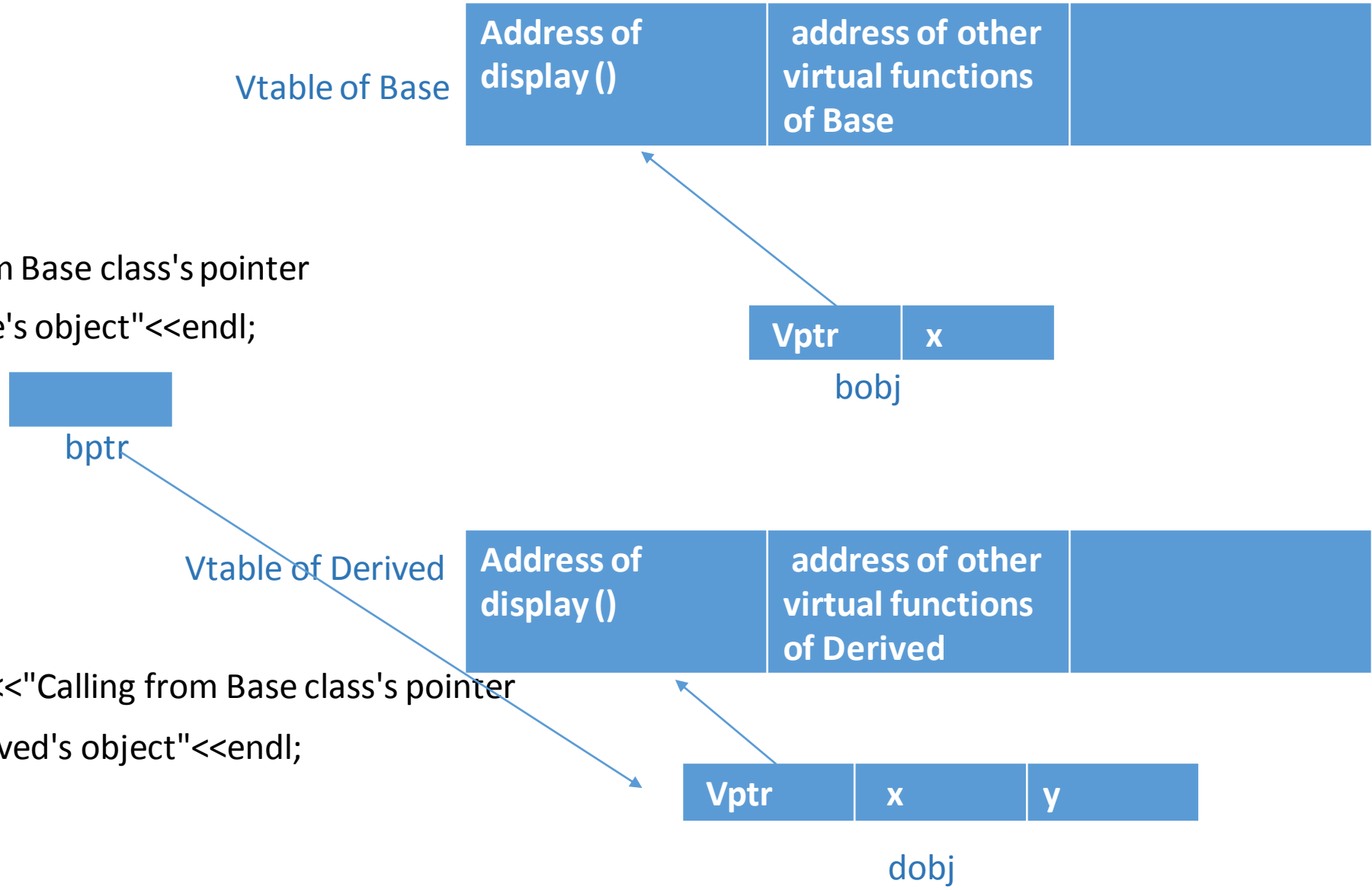
Working of Virtual Function:

```
void main()
{
        Base *bptr, bobj;
         bptr=&bobj;
        cout<<"Calling from Base class's pointer
         that points to Base's object"<<endl;
        bptr->display();

        Derived dobj;
        bptr=&dobj;
        cout<<endl<<endl<<"Calling from Base class's pointer
         that points to Derived's object"<<endl;
        bptr->display();
        getch();
}
```



Working of Virtual Function:

```
void main()
{
        Base *bptr, bobj;
         bptr=&bobj;
        cout<<"Calling from Base class's pointer
         that points to Base's object"<<endl;
        bptr->display();

        Derived dobj;
        bptr=&dobj;
        cout<<endl<<endl<<"Calling from Base class's pointer
         that points to Derived's object"<<endl;
        bptr->display();
        getch();
}
```



Virtual Constructor:

- Constructor cannot be virtual because when constructor of a class is executed, virtual pointer is not defined yet.

Virtual Destructor:

- A virtual destructor is used to free up the memory space allocated by the derived class object while deleting instance of the derived class using base class pointer.

Pure Virtual function:

- Virtual function which has no body is known as pure virtual function
- It is a virtual function for which we do not need to write any function definition, we only need to declare it

Syntax to declare pure virtual function:

```
virtual returnType functionName()=0;
```

- It is used when function doesn’t have any use in the base class but must be implemented by all its derived class

Pure Virtual function:

- Pure virtual is declared in base class and it has no function definition relative to the base class
- Each derived class define the virtual function as per its requirement

Abstract Class:

- It is a class that has at least one pure virtual function
- The classes inheriting the abstract class must provide the definition for the pure virtual function otherwise the subclass would become an abstract class itself
- Abstract class cannot be instantiated, but pointers of abstract class can be created
- Abstract class can have normal functions and variables along with a pure virtual function

Programs:

1. Define a class Person with attributes and behaviors that are common to Student and Teacher, Person class doesn’t have any implementation of the functions, so make it abstract class. Derive class Student and Teacher from class Person and provide the implementation of the inherited functions in each derived class.

Object Oriented Programming in C++

Nepal College of Information Technology

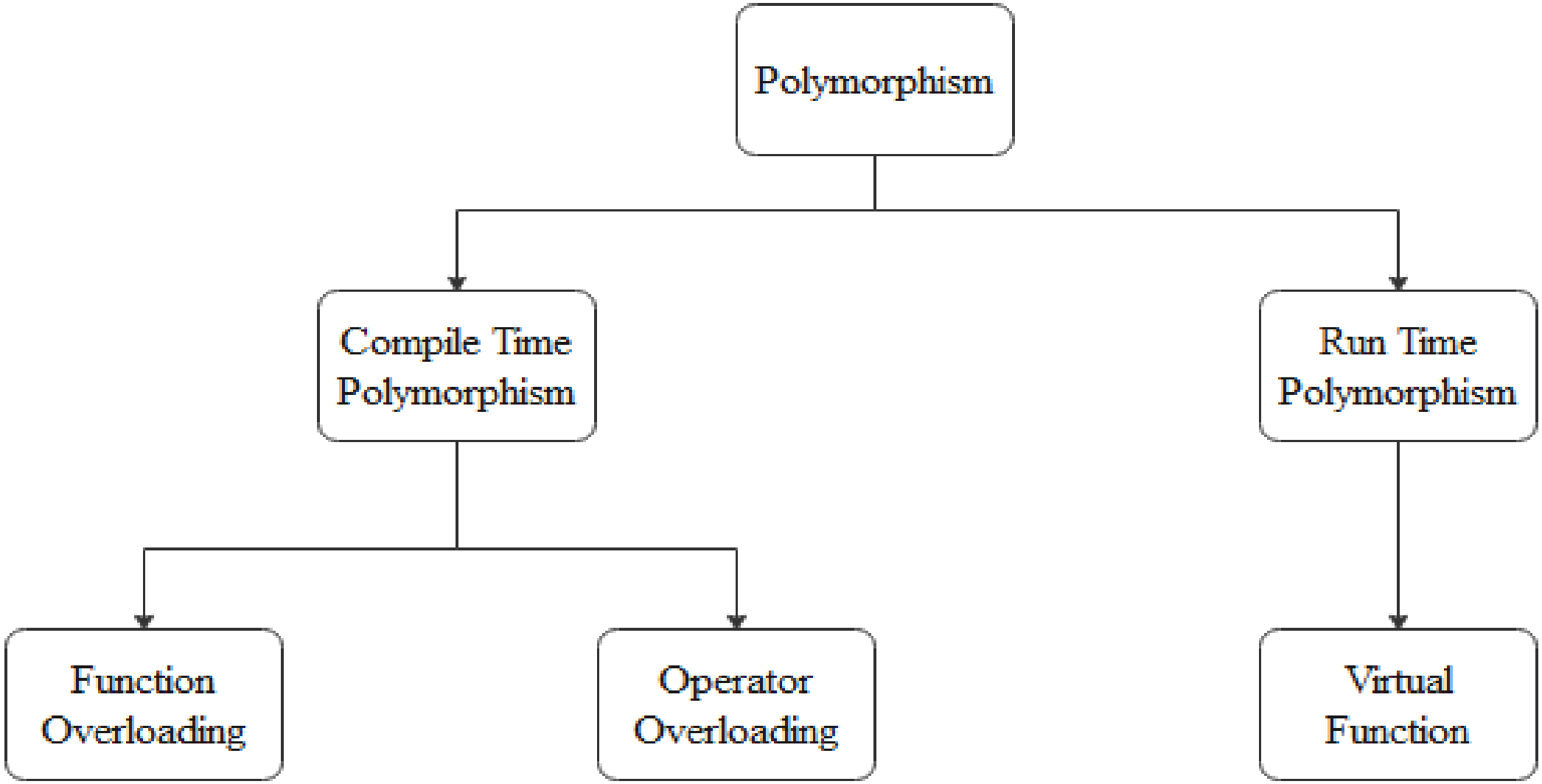
Course Instructor : Er. Rabina Chaudhary

Chapter 5: Polymorphism

Polymorphism:

- “One name multiple forms”
- Polymorphic in Greek: poly means many and morphos means forms

In programming languages, a polymorphic object is any entity such as a variable or function argument that is permitted to hold values of different types during the course of execution


Polymorphism
Compile Time Polymorphism
Run Time Polymorphism
Function Overloading
Operator Overloading
Virtual Function

Compile-Time Polymorphism

- Compile time polymorphism refers to binding of function on the basis of their signature (name, type, sequence of parameters)
- Also called early binding as the calls are already bound to the proper type of functions during the compilation of the program
- Overloaded functions and operators support Compile Time Polymorphism
- Example :
  - void calculateArea (int);
  - void calculateArea(int , int);
  - When the function calculateArea () is invoked, the passed argument determine which one is to be executed
  - This resolution takes place at compile time.

Function Overloading:

1. Write a program to calculate volume of rectangular box, sphere and cylinder. Use calculateVolume() function to calculate the result. Use concept of function overloading.

Runtime Polymorphism :

- A function exhibits runtime polymorphism/ dynamic polymorphism if it exists in various forms and the resolution to different function calls are made during execution time
- Runtime polymorphism is achieved using virtual function

Deferred Methods:

- A deferred method (sometimes called abstract method and in C++ called pure virtual method) is a virtual function without a body
- The virtual function of base class are not used, it is just a place holder and all useful activity is defined as part of the code provided by the child classes

Pure Polymorphism:

- In inheritance where classes have hierarchical relationship where base class have derived classes, all of objects of derived class can be pointed at by a base class pointer.
- By accessing the virtual function through base pointer, C++ selects the appropriate function definition at runtime.

This is a form of polymorphism called **pure polymorphism**.

Explain with an example program implementing the concept of function overriding using virtual function.

Compile time polymorphism using Operator Overloading:

- Operator overloading is one of the many exciting features of C++ language.
- C++ makes the user-defined data types behave in much the same way as the built in types.
- For example, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types.
- This means that C++ has the ability to provide the operator with a special meaning for a data type.
- The mechanism of giving such special meanings to an operator is known as operator overloading.

Operator Overloading:

- a= b+c; works only with basic data types like int and float.
- using operator overloading, we can make this statement legal for user-defined data type also (like object).
- The use of + operator for two objects, means giving special meaning to + operator.
- The statement like C3.add_complex(C1,C2) can be converted into statement like C3=C1+C2 using operator overloading.

Rules on Operator overloading:

1. Only existing operators can be overloaded, new operators cannot be created
2. The precedence and associativity of operators can’t be changed while overloading
3. The overloaded operators follow the syntax rules of the original predefined operators.
4. The overloaded operators must have at least one user-defined type operand

Defining operator overloading:

- An operator overloading is defined with the help of a special function, called **operator function**, which describes the task to be done by the operator.
- syntax for operator function is:

```
return_type operator op(argument list);
```

where,

- operator is keyword
- op is any exisiting valid operator to be overloaded.
  - operator op is function name.

Defining operator overloading:

- Syntax for defining operator function:

```
return_type className :: operator existing_operator( argument/s)
{
        //task of operator
}
```

- Operator function may be either member function or friend function.

this pointer:

- this pointer holds the address of current calling object when calling a member function

  objectA. functionX();

  Here this pointer holds the address of objectA

  - Whenever a non static member function is called, this pointer is passed as hidden argument implicitly

Program to use this pointer to display address of object

```
#include<iostream.h>
#include<conio.h>

class Complex
{
        int real,img;

        public:
                void getAddress()
                {
                cout<<"The address of object using this pointer"<<this;
                 }
};
```

```
void main()
{
        Complex c1,c2;
        cout<<"For c1"<<endl;
        c1.getAddress();
        cout<<endl<<"Address of object directly is "<<&c1<<endl;

        cout<<endl<<endl<<"For c2"<<endl;
        c2.getAddress();
        cout<<endl<<"Address of object directly is "<<&c2<<endl;
        getch();
}
```

Example:

```
#include<iostream.h>
#include<conio.h>

class Complex
{
        int real,img;
         public:
        Complex( int real, int img)
        {
                this->real=real;
                this->img=img;
        }
        void display()
        {
                cout<<"Real part : "<<this->real<<endl
                <<"Imaginary part : "<<img<<endl;
        }
};
```

```
void main()
{
        Complex obj(10,20);
         obj.display();
        getch();
}
```

Overloading Unary Operators : using member function

- The operator function defined as member function of a class has no arguments for unary operators
- The member function takes one less argument because member function is called using object of the class such that the calling object acts as implicit argument
- The operator that operates only one operand to perform its operation is called unary operators.
    - Unary plus(+)
    - Unary minus (-)
    - Increment operator(++)
    - Decrement operator(--)

Operator function as member function:

- Syntax for declaration:

```
return_type operator unary_operator()
{
    task of unary_operator
}
```

- The operator function is invoked using,

```
unary_operator   object_of_class;
object_of_class  unary_operator;
```

which is equivalent to:

```
object_of_class . operator unary_operator( );
```

Example Program to overload unary minus operator:

```
#include<iostream.h>
#include<conio.h>

class Complex
{
        int real,img;
  public:
        Complex( int r, int i)
    {
        real=r;
      img=i;
    }
```

```
    void operator - ()
    {
        real = - real;
        img  = - img;
    }
    void display()
    {
        cout<<"Real part : "<<real<<endl
        <<"Imaginary part : "<<img<<endl;
    }
};
```

```
void main()
{
        Complex obj(10,20);
        obj.display();
        -obj ;    //obj.operator - ();

        cout<<endl<<"Complex  number after negation is :"<<endl;

        obj.display();
  getch();
}
```

Overloading Unary operator using friend function:

- The operator function defined as friend function will have one argument
- Friend function is called without object of the class thus all arguments have to be passes explicitly

Example Program to overload unary minus operator and perform negation operation on Complex type

```
#include<iostream.h>
#include<conio.h>

class Complex
{
        int real,img;
  public:
        Complex( int r, int i)
        {
                real=r;
                img=i;
        }
```

```
void display()
    {
        cout<<"Real part : "<<real<<endl
        <<"Imaginary part : "<<img<<endl;
    }
 friend void operator –(Complex C);
};
 void operator - (Complex C)
    {
        C.real = - C.real;
        C.img  = - C.img;
    }
```

```
void main()
{
        Complex obj(10,20);
        obj.display();
        -obj ;    //operator - (obj);

        cout<<endl<<"Complex  number after negation is :"<<endl;

        obj.display();
  getch();
}
```

Practical:

1. Write a program to overload unary - operator and the operator function returns object

   such that complexOne = -complexTwo;

   - where complexOne and complexTwo are complex numbers

   a. Using operator function as member function
   b. Using operator function as friend function

Practical:

2. Define a class Distance with data members kilometer and meter. Increase the distance by one using increment operator. Write a program to overload increment operator(++) in prefix notation.

Overloading Binary Operator:

- Binary operator: Operator that requires two operands for its operations
- Example:
  - binary plus (+)
  - Binary minus (-)
  - Multiplication (*)
  - Division(/)
  - Equality operator(==)
  - Greater than(>)

  etc

Overloading binary operator: using member function:

- The operator function defined as member function of a class takes one argument for binary operators

- Syntax for defining operator function as member function

```
return_type operator binaryOperator(object_of_class)
{
        body of function
}
```

Example program to overload binary operator: Add two Complex numbers C3=C1+C2 : using member function

```
#include<iostream.h>
#include<conio.h>

class Complex
{
        int real,img;
public:
        Complex()
         {
        }
        Complex (int real, int img)
        {
                this->real=real;
                this->img=img;
        }
        Complex operator + (Complex c)
        {
                Complex cobj;
                cobj.real=real+c.real;
                cobj.img=img+c.img;
                return cobj;
        }
```

```
void display()
  {
            cout<<endl<<"Real part is "<<real<<endl<<"Imaginary part is "<<img;
  }
};

void main()
{
            Complex C3,C1(10,20),C2(30,40);
            C3=C1+C2;  //C3=C1.operator + (C2); /*left operand of binary operator is used as calling object to call operator function
                                                and right argument is passed as argument to the function*/
            cout<<"For C1"<<endl;
            C1.display();
            cout<<endl<<endl<<endl<<"For  C2"<<endl;
            C2.display();
            cout<<endl<<endl<<endl<<"For  C3"<<endl;
            C3.display();

  getch();
}
```

Overloading binary operator: using friend function:

- The operator function defined as friend function of a class takes two argument for binary operators
- Friend function is called without object of the class thus all arguments have to be passes explicitly

Example program to overload binary operator: Add two Complex numbers C3=C1+C2; using friend function

```
#include<iostream.h>
#include<conio.h>

class Complex
{
        int real,img;
public:
        Complex()
        {
        }
        Complex (int real, int img)
        {
                this->real=real;
                this->img=img;
        }
        void display()
        {
                cout<<endl<<"Real part is "<<real<<endl<<"Imaginary part is "<<img;
        }
        friend Complex operator + (Complex c1,Complex c2);
};
```

```
Complex operator + (Complex c1,Complex c2)
        {
                Complex cobj;
                cobj.real=c1.real+c2.real;
                cobj.img=c1.img+c2.img;
                return cobj;
        }


void main()
{
        Complex C3,C1(10,20),C2(30,40);
        C3=C1+C2;  //C3=operator + (C1,C2);
        cout<<"For C1"<<endl;
        C1.display();
        cout<<endl<<endl<<endl<<"For C2"<<endl;
        C2.display();
        cout<<endl<<endl<<endl<<"For C3"<<endl;
        C3.display();

  getch();
}
```

Practical :

1. Write a program to define a class Time with data members hours, minutes and seconds. Overload binary plus operator(+) to add two times.
    a. Using operator function as member function
    b. Using operator function as friend function

2. Write a program to overload binary plus operator(+) to concatenate two strings.
    a. Using operator function as member function
    b. Using operator function as friend function

3. Write a program to overload equality operator (==) to check if two strings are identical or not.
    a. Using operator function as member function
    b. Using operator function as friend function

Practical:

4. Define a class Distance with data members kilometer and meter. Add 5 to the distance object by overloading binary plus operator.
5. Write a program to overload multiplication operator (*) to multiply each element of 3*3 matrix by 9.
6. Define a class Complex with data members real and img. Write a program to overload binary (+) operator to add two Complex numbers, binary minus (–) operator to subtract two Complex number and multiplication operator (*) to multiply two Complex numbers

Data Type Conversion:

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

1. Conversion of Basic type to another basic type:

Program to illustrate use of implicit conversion from float to int

```
#include<iostream.h>
#include<conio.h>
        void main()
        {
                float a=100.5657;
                int b;
                b=a;
                cout<<b;
                getch();
        }
```

1. Conversion of Basic type to another basic type:

Program to illustrate use of explicit conversion from float to int

```
#include<iostream.h>
#include<conio.h>
        void main()
        {
                int a=100;
                float b=float(a)/3;
                cout<<b;
                getch();
        }
```

2. Conversion from basic type to user-defined type:

- Constructor can be used to convert basic data type to user defined data type
    - Distance d; // user defined type

      int meter = 50; // basic type

      d=meter; //error

Program to convert duration in seconds into object of a class Time which has second, minute and hour.

```
#include<iostream.h>
#include<conio.h>

class Time
{
        int minute,second,hour;
  public:
        Time()
        {
                minute=0;
                second=0;
                hour=0;
        }
Time(int sec)
    {
            hour=sec/3600;
            minute=(sec%3600)/60;
            second=(sec%3600)%60;
    }

void display()
    {
            cout<<hour<<":"<<minute<<":"<<second<<endl;
    }
};
```

```
void main()
{
        Time T;        //Time T=3700 or Time T(3700)
        int duration=3700;
        T=duration;    // T=Time(duration);
        T.display();
  getch();
}
```

2. Conversion from basic type to user-defined type:

- Overloaded assignment operator (=) can also be used to convert basic type to class type

Program to convert duration in seconds into object of a class Time which has second, minute and hour. {basic type to class type conversion by overloading assignment '=' operator}

```
#include<iostream.h>
#include<conio.h>

class Time
{
        int minute,second,hour;
  public:
        Time()
        {
                minute=0;
                second=0;
                hour=0;
        }
void operator = (int sec)
    {
                hour=sec/3600;
                minute=(sec%3600)/60;
                second=(sec%3600)%60;
    }

void display()
    {
                cout<<hour<<":"<<minute<<":"<<second<<endl;
    }
};
```

```
void main()
{
        Time T;        //Time T=3700 or Time T(3700)
        int duration=3700;
        T=duration;    // T.operator = (duration);
        T.display();
  getch();
}
```

Conversion from basic type to user-defined type : Practical:

Q1. Write a program to convert meters in value into object of a class Distance which has feet and inch as members

1 meter = 3.28034 feet

3. Conversion from user defined type to basic type

- We define casting operator to convert class type to basic type
- We use casting operator function also called as conversion function

- Syntax to define an overloaded casting operator function

```
operator type_name()
{
        function body
        return(data);
}
```

  - operator float () converts class type to float
  - operator int() converts class type to int

Note:

1. The operator function must be a member of the class
2. The operator function must not specify a return type even though it return the value
3. The operator function must not have any arguments
4. Being a member function, it is invoked by an object and values of that object are used inside the function. Hence no argument is passed to conversion function

Example Program to convert Time type to seconds, which is int type

```
#include<iostream.h>
#include<conio.h>

class Time
{
        int minute, second, hour;
  public:
        Time()
         {
                minute=0;
                second=0;
                hour=0;
        }
        Time(int h,int m,int s)
        {
                minute=m;
                second=s;
                hour=h;
        }
```

```
 operator int ()
    {
                int duration;
                duration=hour*3600.+minute*60+second;
                return duration;
    }

    void display()
    {
                cout<<hour<<":"<<minute<<":"<<second<<endl;
    }
};
void main()
{
                Time T(1,1,40);       //Time T=3700 or Time T(3700)
                int duration;
                duration=T;   //duration=T.operator  int();
                T.display();
                cout<<"The  duration  in seconds is "<<duration;
                getch();
}
```

Conversion from user defined type to basic type : Practical :

Q1.Define a class Distance expressed in feet and inches. Write a program to convert distance into single integer value meter.

1 meter = 3.28034 feet

4. Conversion from One class type to another class type

objectOfDestination = objectOfSource

- It can be performed using either
  - One argument constructor
  - Conversion function

4.1 : Conversion Routine in Destination class

- When the conversion routine is in destination class, we use one argument constructor.

Example Program : class to class conversion using example of Dollar to Rupees conversion.

```
#include<iostream.h>
#include<conio.h>

class Dollar
{
        double dol;
  public:
        Dollar()
        {
                dol=0;
        }
        Dollar(int d)
        {
                dol=d;

        }
        void displayDollar()
        {
                cout<<endl<<"$"<<dol<<endl;
        }
        double getDollar()
        {
                return dol;
        }
};
```

```
class Rupees
{
                double rs;
  public:
  Rupees()
    {
                rs=0;
    }

   Rupees(Dollar dollar)
    {
                rs=120.311*(dollar.getDollar());
    }
   void displayRupees()
     {
         cout<<"Rs."<<rs<<endl;
     }
};
```

```
void main()
{
                Dollar d(154);
                Rupees r;
                r=d;      //r=Rupees(d);
                r.displayRupees();
                d.displayDollar();
  getch();
}
```

Practical:

Q1. Define a class to represent rectangular co-ordinates and another class to represent polar co-ordinates. Convert polar co-ordinates into rectangular co-ordinate value using **conversion routine in destination class**.

note: Polar co-ordinate has radius and angle

Rectangular co-ordinates has x-co-ordinate and y-co-ordinate

x-co-ordinate=radius *cos(angle)

y-co-ordinate=radius *sin(angle)

Practical:

2. Define a class to represent rectangular co-ordinates and another class to represent polar co-ordinates. Convert rectangular co-ordinates into polar coordinate value using **conversion routine in destination class**.

note: Polar co-ordinate has (radius, angle)Rectangular co-ordinates has (x-coordinate and y-co-ordinate)

radius = sqrt(x-coordinate^2+y-coordinate^2)

angle = tan^{-1} (y-coordinate / x-coordinate)

4.2 : Conversion routine in Source class:

- If the source class handles the conversion activity, it is performed using a conversion function

syntax for conversion function is:

```
operator destination_className()
{
        //body of function
}
```

Example Program: class to class conversion using example of Dollar to Rupees conversion.

```
#include<iostream.h>
#include<conio.h>
class Rupees
{
  double rs;
  public:
            Rupees()
            {
                        rs=0;
            }
            Rupees(double r)
            {
                        rs=r;
            }

            void displayRupees()
            {
                        cout<<"Rs"<<rs<<endl;
            }
};
```

```
class Dollar
{
  double dol;
  public:
                Dollar()
                {
                        dol=0;
                }
                Dollar(int d)
                {
                        dol=d;

                }
                void displayDollar()
                {
                        cout<<endl<<"$"<<dol<<endl;
                }
                double getDollar()
                {
                        return dol;
                }
```

```
 operator Rupees()
    {

                double rupees;
                rupees=dol*120.11;
                Rupees r(rupees);
                return r;
    }
};




void main()
{
                Dollar d(154);
                Rupees r;
                r=d;   //r=d.operator  Rupees();
                r.displayRupees();
                d.displayDollar();
                getch();
}
```

Practical:

Q1. Define a class to represent rectangular co-ordinates and another class to represent polar co-ordinates. Convert polar co-ordinates into rectangular co-ordinate value using **conversion routine in source class**.

note: Polar co-ordinate has radius and angle

Rectangular co-ordinates has x-co-ordinate and y-co-ordinate

x-co-ordinate=radius *cos(angle)

y-co-ordinate=radius *sin(angle)

Practical:

2. Define a class to represent rectangular co-ordinates and another class to represent polar co-ordinates. Convert rectangular co-ordinates into polar coordinate value using **conversion routine in source class**.

note: Polar co-ordinate has (radius, angle)Rectangular co-ordinates has (x-coordinate and y-co-ordinate)

radius = sqrt(x-coordinate^2+y-coordinate^2)

angle = tan^{-1} (y-coordinate / x-coordinate)

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Chapter 6: Template and generic programming

Generic Programming:

- A new concept that enables us to define generic classes and functions and thus provides support for generic programming
- Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures

Template:

- C++ allows defining general classes or functions to handle different data types
- The generic family of classes and functions is called template

- In template, we define function with general type of arguments such that we can use the function for any data type
- A template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the function

Example:

a template for a function multiply() multiplies int, float, double types

Template:

- In template, we define class with general type data member such that we can use the class for any data type
- When an object of a specific type is defined for actual use the template definition for that class is substituted with the required data type

Example:

- a class template for an array class enables us to create array of various data types such as int or a float

Types of Template:

1. Function template
2. Class Template

Function Template:

- Function template is used to define a function to handle arguments of different types
- Function templates are special functions that can operate with generic types
- In C++ this can be achieved using template parameters
- A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

Without using function template:

```
#include<iostream.h>
#include<conio.h>
void swap(int a, int b)
{
        int temp;
        temp=a;
        a=b;
        b=temp;
        cout<<"After  swaping a="<<a<<endl<<"   b="<<b<<endl<<endl;
}

void swap(float a,float b)
{
        float temp;
        temp=a;
        a=b;
        b=temp;
        cout<<"After  swaping a="<<a<<endl<<"   b="<<b<<endl<<endl;
}
```

```
void swap(char a, char b)
{
        char  temp;
        temp=a;
        a=b;
        b=temp;
        cout<<"After swaping a="<<a<<endl<<"    b="<<b<<endl<<endl;
}

void main()
{
        int r=100,s=200;
        swap(r,s);
        float x=100.2323,y=121.333;
        swap(x,y);
        char a='a',b='b';
        swap(a,b);
getch();
}
```

Defining function template:

1. Defining general data type:

   template < typename templateName>

   2. Defining a function with general datatype

   returnType functionName (argument_with_template_name)
   {
   body of function with type template_name if appropriate
   }

Example Program:

```
#include<iostream.h>
#include<conio.h>

template <typename T> //defining template
void swap(T a, T b) //using generic type T wherever appropriate in function template
{
        T temp;
        temp=a;
        a=b;
        b=temp;
        cout<<"After swaping a="<<a<<endl<<"   b="<<b<<endl<<endl;
}
void main()
{
        int r=100,s=200;
        swap(r,s);
         swap<float>(102.560,45.1212);
        char a='a',b='b';
        swap(a,b);
getch();
}
```

Practice:

1. Write a program to find maximum among 3 integers, 3 double values and 3 characters using function template
2. Write a program to sort integer array, character array and float array using function template

Function template with multiple parameters:

- More than one generic data type in function template
- They are separated by comma separated list

1. Defining general data type

```
template < typename template_name1, typename template_name1,…>
```

2. Defining function with general data type

```
returntype functionName (arguments with template_name1, arguments with template_name2, …)
{
        body of function with type template
}
```

Example Program:

```
#include<iostream.h>
#include<conio.h>

template <typename T1,typename T2>
void testFunction( T1 x, T2 y)
{
        cout<<endl<<x<<endl<<y<<endl<<endl;
}

void main()
{
        testFunction(123,124.656);
         testFunction(10.234,'z');
        testFunction(12,'r');
        getch();
}
```

Practical:

Q. Define a function that calculates the memory allocated for integer, double, float, char using function template with multiple parameters.

Overloading function templates:

- A template function may be overloaded either by template function or ordinary function of its name
- In normal function overloading, the function may have the same definition, but with different arguments types or numbers or sequence
- In case of templates, the overloading happens only on the basis of number of parameters

Example program: Function template overloading

```
#include<iostream.h>
#include<conio.h>

template <typename T>
T maximum( T x, T y)
{
        T z= (x>y)?x:y;
        return z;
}
template <typename T>
T maximum ( T x, T y , T z)
{
        if( x>y && x>z)
                return x;
        else if(y>z)
                return y;
        else
                return z;
}
```

```
template <typename T>
void display(T x)
{
		cout<<endl<<x<<"is maximum"<<endl;
}


void main()
{
		int max=maximum(123,1246,1234);
		display(max);
		char Cmax=maximum('a','b');
		display(Cmax);
  getch();
}
```

Template function can also be overloaded by non- template function:

- Template function can be overloaded with ordinary or non-template function
- In this case, the compiler gives priority to ordinary function with exact match, if the exact match is not with the ordinary function, then compiler prefers the template function

Template function can also be overloaded by non- template function: Example

```
#include<iostream.h>
#include<conio.h>

int  maximum(  int  x, int y)
{
                cout<<endl<<"normal  function  is called"<<endl ;
                int z= (x>y)?x:y;
                return  z;
}
template  <typename  T>
T maximum(  T x, T y)
{
                cout<<endl<<"template  function  with two  parameters  is called"<<endl;
                T z= (x>y)?x:y;
 return  z;
}
```

```
template <typename T>
T maximum ( T x, T y , T z)
{
        cout<<endl<<"template function  with three parameters is called"<<endl;
        if( x>y && x>z)
                return x;
        else if(y>z)
                return y;
        else
                return z;
}
template <typename T1>
void display(T1 x)
{
        cout<<endl<<"template function is called to display maximum"<<endl;
        cout<<endl<<x<<" is maximum"<<endl;
}
```

```
void main()
{
        int max=maximum(123,1246,1234);
        display(max);
        cout<<endl<<endl<<"For maximum among two integers"<<endl;
        max=maximum(10,20); /*calling non-template function (ordinary
                                function)*/
        display(max);
        cout<<endl<<endl;
        char Cmax=maximum('a','b');
        display(Cmax);
  getch();
}
```

Practical:

Q1. Write a program to overload template function to swap two values and three values. The values to be swapped must be integer, char and double. Also implement template overloading with ordinary function.

Q2. Write a function template to calculate average and multiplication of numbers

Q3. Calculate roots of quadratic equation by using function template

Non type template argument:

- A template non-type parameter is a special type of parameter that does not substitute for a type, but is instead replaced by a value

Example:

```
#include<iostream.h>
#include<conio.h>

template <typename T, int size> //size is non-type parameter

void testArray(T x[])
{
        int i;
        cout<<endl<<"Enter values of array elements "<<endl;
        for(i=0;i<size;i++)
        {
        cin>>x[i];
        }
        cout<<endl<<"The array is :";
        for(i=0;i<size;i++)
        {
                cout<<x[i]<<",";
        }
}
```

```
void main()
{
        int x[5];
        double y[10];
        char z[10];
        cout<<endl<<"For integer array"<<endl;
        testArray<int, 5>(x);   //generic type T is replaced by int
                                //non-type parameter, size, is replaced by 5
         cout<<endl<<endl<<"For double array"<<endl;
        testArray<double, 10>(y);       //generic type T is repalced by double
                                        //non-type size is replaced by 10
        cout<<endl<<endl<<"For character array"<<endl;
        testArray<char, 5>(z);
getch();
}
```

Class Template:

- A class template is a kind of class which has members of template type
- Class template lets us define the behavior of a class without actually knowing what data type will be handled by the operations of the class
- Template class provide the mechanism for creating applications with generic type which are common applications such as linked list, stacks ,queues, etc.

General form for class template:

Step 1. defining template

```
template<typename T>
```

Step 2: Defining class with members of template type

```
class className
{
        class members with data type  T whenever appropriate
}
```

Step 3: Defining object

```
class_name <specific data type for T> objectName;
```

Class Template:

- Each object of template class must be of some specific size. Therefore, the data type must be specified before creating an object of template class

Example Program:

```
#include<iostream.h>
#include<conio.h>

template <typename T>
class Rectangle
{
        T length, breadth;
        public:
                Rectangle();
                Rectangle (T x, T y)
                {
                length=x;
                breadth=y;
                }
                T area();
};
```

```
template <typename T>
Rectangle<T>::Rectangle()
    {
            length=0;
            breadth=0;
    }
template <typename T>
  T Rectangle<T> :: area()
    {
            return (length*breadth);
    }

void main()
{
            Rectangle <int> obj1(5,89);
            Rectangle <double>  obj2(12.34,56.78);
            cout<<obj1.area()<<endl;
            cout<<obj2.area()<<endl;
            cout<<sizeof(obj1)<<endl<<sizeof(obj2);
  getch();
}
```

Class Template with multiple parameters:

- We can define class template with more than one generic data type
- They are declared as comma separated list in template specification

Step 1. defining template

```
template<typename T1, typename T2,…>
```

Step 2: Defining class with members of template type

```
class className
{
        class members with data type  T1, T2, … whenever appropriate
}
```

Step 3: Defining object

```
class_name <specific data type for T1, specific data type for T2, …> objectName;
```

Example Program:

```
#include<iostream.h>
#include<conio.h>

template <typename T1, typename T2>
class Rectangle
{
  T1 length;
  T2 breadth;
  public:
                    Rectangle();
    Rectangle (T1 x, T2 y)
    {
                    length=x;
      breadth=y;
    }
    T1 area();
};
```

```
template <typename T1,typename T2>
Rectangle<T1,T2>::Rectangle()
    {
                length=0;
                breadth=0;
    }
template <typename T1,typename T2>
  T1 Rectangle<T1,T2> :: area()
    {
                return (length*breadth);
    }

void main()
{
  Rectangle <int,double> obj1(5,89.12);
  Rectangle <double,double> obj2(12.34,56.78);
  Rectangle <int,int> obj3(1,2);
  cout<<obj1.area()<<endl;
  cout<<obj2.area()<<endl;
  cout<<obj3.area()<<endl;
  cout<<sizeof(obj1)<<endl<<sizeof(obj2);
  getch();
}
```

Non-type template argument:

```
template <class T, int size>
 class className
        {   T a[size];
            …
        };
```

This template supplies the size of the array as an argument

So we can make objects as

```
 className <int, 10>   a1;  // array of 10 integers
 className <float, 10> a2; // array of 10 floats
className <char, 20>  a3;  // string of size 20
```

Member function of class template defined outside class:

Syntax:

```
template<typename T>
returntype  className<T> :: functionName(argument_list)
{
        body of function
}
```

Practical:

Q1. Define a class Array with general type data member and appropriate member functions to input, display and sort the array. Use this class as to perform operation on integer array, character array and double array.

Q2. Define a class Box with l, b and h as general type data members and appropriate functions to calculate volume of a box. Use template class for the above operations.

Practical:

3. Define two classes “Polar” and “Rectangle” to represent points in polar and rectangle system. Use conversion routines to convert from one system to another system using template.

Implementing stack data structure :

Step 1 : start

Step 2 : Include all the header files which are used in the program

Step 3 :Define a class to represent Stack with following class members

- Step 3.1 : Create a one dimensional array with fixed size (int stack[SIZE])
- Step 3.2 : Declare all the functions used in stack implementation
- Step 3.3: Define a integer variable 'top' and initialize with '-1'. (int top = -1) in constructor

Step 4: In main function, make the function call to insert element to stack and remove element from stack as per the requirement

Step 5: stop

push(value) - Insert value into the stack

- In a stack, push() is a function used to insert an element into the stack.
- New element is always inserted at top position.
- Push function takes one integer value as parameter and inserts that value into the stack

Step 1 - Check whether stack is FULL. (top == SIZE-1)

Step 2 - If it is FULL, then display "Stack is full and insertion is not possible”

Step 3 - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

pop() : remove element from stack

- In a stack, the element is always deleted from top position.

Step 1 - Check whether stack is empty. (top == -1)

Step 2 - If stack is empty, then display "Stack is EMPTY"

Step 3 - If it is not empty, then remove stack[top] and decrement top value by one (top--).

Program implementing stack operation using class template:

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
template <typename T, int max>
class Stack
{
        T stk[max];
        int top;
        public:
        Stack()
        {
                top = -1;
        }

        void push(T data);

        void pop();

        void show();
};
```

```
template<typename  T, int max>
void Stack<T,max> ::push(T data)
 {
                if (top == (max -1))
                                cout << "stack is full\n";
                else
                {
                top++;
                stk[top]= data;
                }
 }
template<typename  T, int max>
void Stack<T,max>:: pop()
 {
                if (top == -1)
                                cout << "stack is empty\n";
                else
                {
                                T data = stk[top];
                                cout<<endl<<data  << " is popped  out"<<endl;
                                top--;
                }
}
```

```
template<typename T, int max>
void Stack<T,max>:: show()
        {
                for(int i=top;i>=0;i--)
                cout << "\nstack["<< i <<"]" << stk[i];
        }
```

```
void main()
{
                    Stack <char ,5>s;
                    int choice;
                    char data;
 while(1)
 {
                    cout<<endl<<endl<<"***Menu***"<<endl <<"1. push data "<<endl<<"2. Pop data"<<endl<<"3. Display"<<endl<<"4. exit"<<endl;
                    cout<<"Enter your choice : ";
                    cin>>choice;
                    switch(choice)
                    {
                    case 1:          cout<<"Enter data to be pushed into stack :  " ;
                                     cin>>data;
                                     s.push(data);
                                     break;
                    case 2:          s.pop();
                                     break;
                    case 3:          s.show();
                                     break;
                    case 4: exit(0);
                    default : cout<<"Please enter valid choice"<<endl;
                    }
}

 }
```

Implementation of stack using standard template library:

```
#include<iostream>
#include<stack>

using namespace std;

int main()
{
        stack<int> stackobj;
        stackobj.push(41);
        stackobj.push(42);
        stackobj.push(45);
        stackobj.push(50);
        while (!stackobj.empty()) {
        cout << endl << stackobj.top()<<endl;
        stackobj.pop();
  }
        cout<<endl;
        cout<<endl;

        stackobj.push(541);

        stackobj.push(50);
        stackobj.pop();
        while (!stackobj.empty()) {
        cout << " " << stackobj.top();
        stackobj.pop();
  }
 return 0;
}
```

Implementing Stack using standard template library:

```
#include<iostream>
#include<stack>
using namespace std;
int main(){
        stack<int> stackobj;
        stackobj.push(41);
        stackobj.push(42);
        stackobj.push(45);
        stackobj.push(50);
        while (!stackobj.empty()) {
        cout << endl << stackobj.top()<<endl;
        stackobj.pop();
  }
        cout<<endl;
        cout<<endl;
        stackobj.push(541);

        stackobj.push(50);
        stackobj.pop();
        while (!stackobj.empty()) {
                cout << " " << stackobj.top();
                stackobj.pop();
        }
  return 0;
}
```

Standard Template Library:

Standard template library is a set of C++ template classes to provide template classes and functions that implement many popular and commonly used algorithms and data structures like vectors, stack, queues, lists , etc.

- Components of STL
  - Containers
  - Algorithms
  - Iterators

Containers:

- Containers are used to manage collections of objects of a certain kind
- There are several different types of containers like queue, stack, list, etc

Algorithms:

- Algorithms acts on containers
- They provides means by which we will perform operations on contents of containers
- Algorithm library contains built in functions that performs complex algorithms on the data structures

Iterators:

- Iterators are used to point to the containers
- Acts as bridge between containers and algorithm

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Chapter 6: Template and generic programming

Generic Programming:

- A new concept that enables us to define generic classes and functions and thus provides support for generic programming
- Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures

Template:

- C++ allows defining general classes or functions to handle different data types
- The generic family of classes and functions is called template

- In template, we define function with general type of arguments such that we can use the function for any data type
- A template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the function

Example:

a template for a function multiply() multiplies int, float, double types

Template:

- In template, we define class with general type data member such that we can use the class for any data type
- When an object of a specific type is defined for actual use the template definition for that class is substituted with the required data type

Example:

- a class template for an array class enables us to create array of various data types such as int or a float

Types of Template:

1. Function template
2. Class Template

Function Template:

- Function template is used to define a function to handle arguments of different types
- Function templates are special functions that can operate with generic types
- In C++ this can be achieved using template parameters
- A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

Without using function template:

Defining function template:

1. Defining general data type:

   template < typename templateName>

   2. Defining a function with general datatype

   returnType functionName (argument_with_template_name)
   {
   body of function with type template_name if appropriate
   }

Example Program:

Practice:

1. Write a program to find maximum among 3 integers, 3 double values and 3 characters using function template
2. Write a program to sort integer array, character array and float array using function template

Function template with multiple parameters:

- More than one generic data type in function template
- They are separated by comma separated list

1. Defining general data type

```
template < typename template_name1, typename template_name1,…>
```

2. Defining function with general data type

```
returntype functionName (arguments with template_name1, arguments with template_name2, …)
{
        body of function with type template
}
```

Example Program:

```
#include<iostream.h>
#include<conio.h>

template <typename T1,typename T2>
void testFunction( T1 x, T2 y)
{
        cout<<endl<<x<<endl<<y<<endl<<endl;
}

void main()
{
        testFunction(123,124.656);
         testFunction(10.234,'z');
        testFunction(12,'r');
        getch();
}
```

Practical:

Q. Define a function that calculates the memory allocated for integer, double, float, char using function template with multiple parameters.

Overloading function templates:

- A template function may be overloaded either by template function or ordinary function of its name
- In normal function overloading, the function may have the same definition, but with different arguments types or numbers or sequence
- In case of templates, the overloading happens only on the basis of number of parameters

Example program: Function template overloading

```
#include<iostream.h>
#include<conio.h>

template <typename T>
T maximum( T x, T y)
{
        T z= (x>y)?x:y;
        return z;
}
template <typename T>
T maximum ( T x, T y , T z)
{
        if( x>y && x>z)
                return x;
        else if(y>z)
                return y;
        else
                return z;
}
```

```
template <typename T>
void display(T x)
{
		cout<<endl<<x<<"is maximum"<<endl;
}


void main()
{
		int max=maximum(123,1246,1234);
		display(max);
		char Cmax=maximum('a','b');
		display(Cmax);
  getch();
}
```

Template function can also be overloaded by non- template function:

- Template function can be overloaded with ordinary or non-template function
- In this case, the compiler gives priority to ordinary function with exact match, if the exact match is not with the ordinary function, then compiler prefers the template function

Template function can also be overloaded by non- template function: Example

```
#include<iostream.h>
#include<conio.h>

int  maximum(  int  x, int y)
{
                cout<<endl<<"normal  function  is called"<<endl ;
                int z= (x>y)?x:y;
                return  z;
}
template  <typename  T>
T maximum(  T x, T y)
{
                cout<<endl<<"template  function  with two  parameters  is called"<<endl;
                T z= (x>y)?x:y;
 return  z;
}
```

```
template <typename T>
T maximum ( T x, T y , T z)
{
        cout<<endl<<"template function  with three parameters is called"<<endl;
        if( x>y && x>z)
                return x;
        else if(y>z)
                return y;
        else
                return z;
}
template <typename T1>
void display(T1 x)
{
        cout<<endl<<"template function is called to display maximum"<<endl;
        cout<<endl<<x<<" is maximum"<<endl;
}
```

```
void main()
{
        int max=maximum(123,1246,1234);
        display(max);
        cout<<endl<<endl<<"For maximum among two integers"<<endl;
        max=maximum(10,20); /*calling non-template function (ordinary
                                function)*/
        display(max);
        cout<<endl<<endl;
        char Cmax=maximum('a','b');
        display(Cmax);
  getch();
}
```

Practical:

Q1. Write a program to overload template function to swap two values and three values. The values to be swapped must be integer, char and double. Also implement template overloading with ordinary function.

Q2. Write a function template to calculate average and multiplication of numbers

Q3. Calculate roots of quadratic equation by using function template

Non type template argument:

- A template non-type parameter is a special type of parameter that does not substitute for a type, but is instead replaced by a value

Example:

```
#include<iostream.h>
#include<conio.h>

template <typename T, int size> //size is non-type parameter

void testArray(T x[])
{
        int i;
        cout<<endl<<"Enter values of array elements "<<endl;
        for(i=0;i<size;i++)
        {
        cin>>x[i];
        }
        cout<<endl<<"The array is :";
        for(i=0;i<size;i++)
        {
                cout<<x[i]<<",";
        }
}
```

```
void main()
{
        int x[5];
        double y[10];
        char z[10];
        cout<<endl<<"For integer array"<<endl;
        testArray<int, 5>(x);   //generic type T is replaced by int
                                //non-type parameter, size, is replaced by 5
         cout<<endl<<endl<<"For double array"<<endl;
        testArray<double, 10>(y);       //generic type T is repalced by double
                                        //non-type size is replaced by 10
        cout<<endl<<endl<<"For character array"<<endl;
        testArray<char, 5>(z);
getch();
}
```

Class Template:

- A class template is a kind of class which has members of template type
- Class template lets us define the behavior of a class without actually knowing what data type will be handled by the operations of the class
- Template class provide the mechanism for creating applications with generic type which are common applications such as linked list, stacks ,queues, etc.

General form for class template:

Step 1. defining template

```
template<typename T>
```

Step 2: Defining class with members of template type

```
class className
{
        class members with data type  T whenever appropriate
}
```

Step 3: Defining object

```
class_name <specific data type for T> objectName;
```

Class Template:

- Each object of template class must be of some specific size. Therefore, the data type must be specified before creating an object of template class

Example Program:

Class Template with multiple parameters:

- We can define class template with more than one generic data type
- They are declared as comma separated list in template specification

Step 1. defining template

```
template<typename T1, typename T2,…>
```

Step 2: Defining class with members of template type

```
class className
{
        class members with data type  T1, T2, … whenever appropriate
}
```

Step 3: Defining object

```
class_name <specific data type for T1, specific data type for T2, …> objectName;
```

Example Program:

Non-type template argument:

```
template <class T, int size>
 class className
        {   T a[size];
            …
        };
```

This template supplies the size of the array as an argument

So we can make objects as

```
 className <int, 10>   a1;  // array of 10 integers
 className <float, 10> a2; // array of 10 floats
className <char, 20>  a3;  // string of size 20
```

Member function of class template defined outside class:

Syntax:

```
template<typename T>
returntype  className<T> :: functionName(argument_list)
{
        body of function
}
```

Practical:

Q1. Define a class Array with general type data member and appropriate member functions to input, display and sort the array. Use this class as to perform operation on integer array, character array and double array.

Q2. Define a class Box with l, b and h as general type data members and appropriate functions to calculate volume of a box. Use template class for the above operations.

Practical:

3. Define two classes “Polar” and “Rectangle” to represent points in polar and rectangle system. Use conversion routines to convert from one system to another system using template.

Implementing stack data structure using class template:

Step 1 : start

Step 2 : Include all the header files which are used in the program

Step 3 :Define a class to represent Stack with following class members

- Step 3.1 : Create a one dimensional array with fixed size (int stack[SIZE])
- Step 3.2 : Declare all the functions used in stack implementation
- Step 3.3: Define a integer variable 'top' and initialize with '-1'. (int top = -1) in constructor

Step 4: In main function, make the function call to insert element to stack and remove element from stack as per the requirement

Step 5: stop

push(value) - Insert value into the stack

- In a stack, push() is a function used to insert an element into the stack.
- New element is always inserted at top position.
- Push function takes one integer value as parameter and inserts that value into the stack

Step 1 - Check whether stack is FULL. (top == SIZE-1)

Step 2 - If it is FULL, then display "Stack is full and insertion is not possible”

Step 3 - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

pop() : remove element from stack

- In a stack, the element is always deleted from top position.

Step 1 - Check whether stack is empty. (top == -1)

Step 2 - If stack is empty, then display "Stack is EMPTY"

Step 3 - If it is not empty, then remove stack[top] and decrement top value by one (top--).

Program implementing stack operation using class template:

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
template <typename T, int max>
class Stack
{
        T stk[max];
        int top;
        public:
        Stack()
        {
                top = -1;
        }

        void push(T data);

        void pop();

        void show();
};
```

```
template<typename  T, int max>
void Stack<T,max> ::push(T data)
 {
                if (top == (max -1))
                                cout << "stack is full\n";
                else
                {
                top++;
                stk[top]= data;
                }
 }
template<typename  T, int max>
void Stack<T,max>:: pop()
 {
                if (top == -1)
                                cout << "stack is empty\n";
                else
                {
                                T data = stk[top];
                                cout<<endl<<data  << " is popped  out"<<endl;
                                top--;
                }
}
```

```
template<typename T, int max>
void Stack<T,max>:: show()
        {
                for(int i=top;i>=0;i--)
                cout << "\nstack["<< i <<"]" << stk[i];
        }
```

```
void main()
{
                    Stack <char ,5>s;
                    int choice;
                    char data;
 while(1)
 {
                    cout<<endl<<endl<<"***Menu***"<<endl <<"1. push data "<<endl<<"2. Pop data"<<endl<<"3. Display"<<endl<<"4. exit"<<endl;
                    cout<<"Enter your choice : ";
                    cin>>choice;
                    switch(choice)
                    {
                    case 1:          cout<<"Enter data to be pushed into stack :  " ;
                                     cin>>data;
                                     s.push(data);
                                     break;
                    case 2:          s.pop();
                                     break;
                    case 3:          s.show();
                                     break;
                    case 4: exit(0);
                    default : cout<<"Please enter valid choice"<<endl;
                    }
}

 }
```

Standard Template Library:

Standard template library is a set of C++ template classes to provide template classes and functions that implement many popular and commonly used algorithms and data structures like vectors, stack, queues, lists , etc.

- Components of STL
  - Containers
  - Algorithms
  - Iterators

Containers:

- Containers are used to manage collections of objects of a certain kind
- There are several different types of containers like queue, stack, list, etc

Algorithms:

- Algorithms acts on containers
- They provides means by which we will perform operations on contents of containers
- Algorithm library contains built in functions that performs complex algorithms on the data structures

Iterators:

- Iterators are used to point to the containers
- Acts as bridge between containers and algorithm

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Chapter 4:Inheritance

Subclass, Subtype and Substitutability

The relationship of the data type associated with a parent class to the data type associated with a child class gives rise to the following arguments:

i. Instances of the subclass must possess all data areas associated with parent class

ii. Instances of the subclass must implement thru inheritance at least all functionality defined for the parent class (They can also define new functionality)

iii. An instance of a child class can mimic the behavior of the parent class and should be indistinguishable from an instance of the parent class if substituted in a similar situation

Principal of Substitutability :

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

Example program for Principal of Substitutability :

```
#include<iostream.h>
#include<conio.h>
class A
{
        public:
        void printMsg()
         {
                cout <<" I am A!!";
        }
};
class B : public A
{
        public:
        void printMsg()
        {
        cout <<" I am B!!";
        }
};
```

Example program for Principal of Substitutability :

```
void testingA(A  a)
{
        a.printMsg();
}
void testingFunc()
{
        B b;
        cout<<"  Passing B"<<endl;
        testingA(b);        // b substituted for object of A.
}

void main()
{
        testingFunc();
        getch();
}
```

Subclass:

- A class which inherits the characteristics and behavior of another class is called a subclass.

Subtype:

- Subtype is subclass relationship in which the principle of substitutability is maintained

Forms of Inheritance:

Description of categorization of the uses of inheritance

1. Inheritance for Specialization
2. Inheritance for Specification
3. Inheritance for Construction
4. Inheritance for Generalization
5. Inheritance for Extension
6. Inheritance for Limitation
7. Inheritance for Variance
8. Inheritance for Combination

1. Inheritance for Specialization:

- Child class is specialized form of Parent class
- Child class satisfies the specification of Parent in all relevant respects
- Child class inherits the properties of the Parent class and has its own additional features also
- Child class can override methods from the Parent in order to specialize the Child class in some way

2. Inheritance for Specification:

- The Parent class is just used to provide behavior or which function should be available but doesn’t provide its implementation
- Parent class is abstract class and just provides specification for child class
- The derived class have the implementation of the function respective to their requirement

3. Inheritance for Construction:

- If the Parent class is used as a source for the behavior but the child has no is-a relationship to the parent, then the child is using inheritance for construction

4. Inheritance for Generalization:

- Subclass extends the behavior of Superclass to create more general kind of object but doesn’t override any method with completely new features

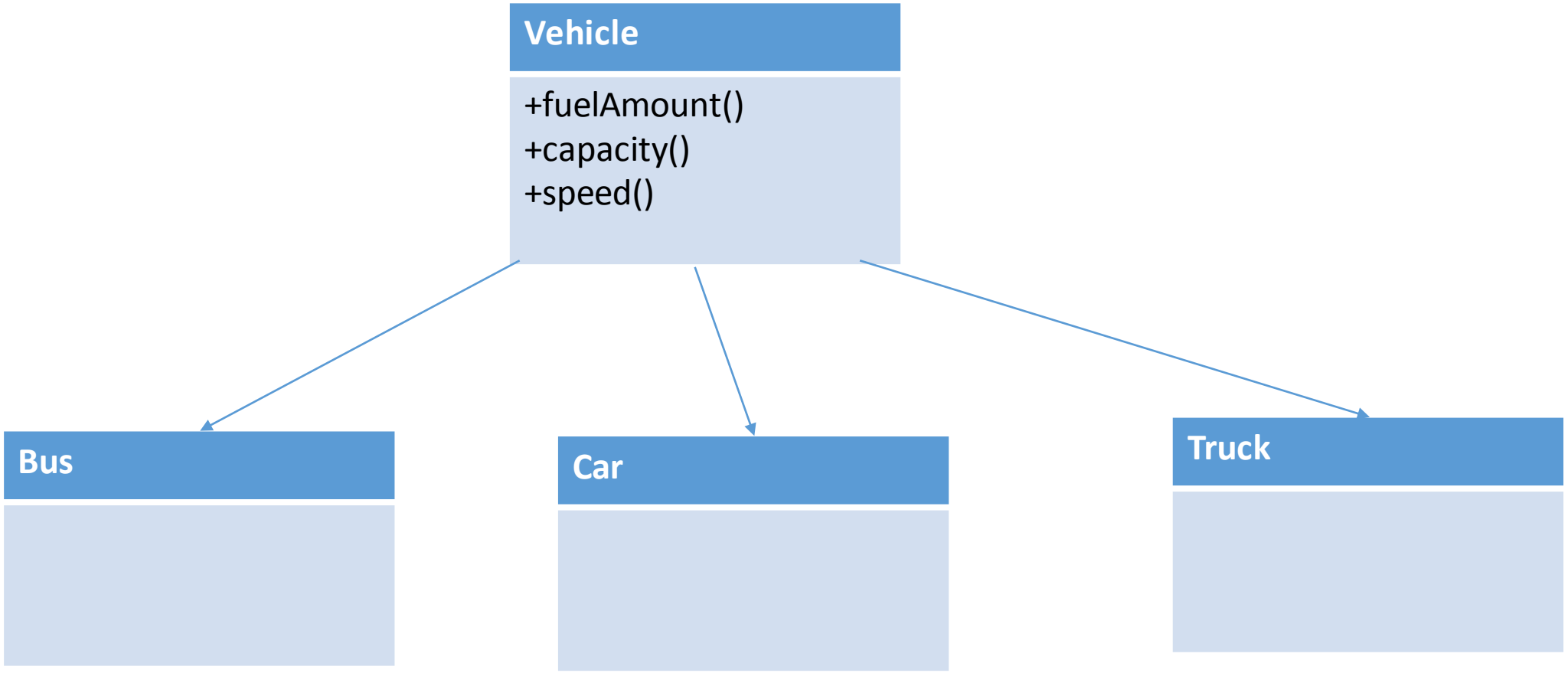
Without using inheritance:

Bus
+fuelAmount() +capacity() +speed()

Car
+fuelAmount() +capacity() +speed()

Truck
+fuelAmount() +capacity() +speed()

Using inheritance for Generalization:



5. Inheritance for Extension:

- Extension simply adds new methods in the Child to those of the Parent

Inheritance for Limitation:

- Inheritance for limitation is used when the behavior of subclass is smaller or more restrictive than the behavior of the Parent class
- The programmer can override the undesired methods inherited from Parent class so that they can limit the features
- The methods override existing methods and eliminate or limit their functionality

7. Inheritance for Variance:

- Used when two or more classes have similar implementations but do not seem to possess any hierarchical relationships between concepts represented by the classes
- e.g. code required to control a mouse and a tablet are identical
- In this case one of the two classes can be chosen as the parent by inheriting the common code and overriding the device specific code

8. Inheritance for Combination:

- The subclass represents a combination of features from two or more parent classes
- The ability of a class to inherit from two or more parent classes is known as multiple inheritance

Advantages of inheritance:

- Software Reusability : The child class need not rewrite the behavior inherited from its parent class
- Code Sharing : many user or projects can use the same class, code sharing through inheritance
- Software Components : Inheritance facilitates building of reusable software components
- Rapid Prototyping : usage of reusable components cuts down time required for coding which can be spent on understanding the new and unusual portions of the system. Thus software can generated quickly and early leading to a style of programming called Rapid Prototyping.
- Information Hiding : Programmer reusing a software component needs only to understand the functionality of the component and its interface to reuse it

IS – A relationship:

- In OOP “is-a” relationship is a relation between two classes where one class is a specialized form of second class
- X is a specialized instance of concept Y if the assertion in plain English “X is a Y” sounds correct or logical
- Examples of inheritance satisfying the “is-a” relationship
  - student is a person, apple is a fruit, car is a vehicle
- The concept of “is-a” relationship is represented by Inheritance

HAS - A relationship:

- If the one concept is a component of another concept then there exist “has-a “ relationship
- the two concepts are not the same thing
- e.g. a car has a engine , fire station has fire fighter
- Also called a “part of” relationship e.g. an engine is a part of car

- Testing the relationship:
  - Form a simple sentence “ X has a Y”,
  - if the result sounds reasonable then the relationship hold

Example of “Car has a engine”:

```
class Engine
        {
        };
class Car
        {
        Engine e;    // a car is composed of an engine
        };
```

Example program of “has-a “ relationship:

```
#include<iostream.h>
#include<conio.h>
class Engine
{
        public:
            void startEngine()
                {
                        cout<<" Engine started"<<endl;
                }

        void stopEngine()
        {
                cout<<" Engine Stopped"<<endl;
        }
};
```

Example program of “has-a “ relationship:

```
class Car
{
        Engine e;
        public:
        void startCar()
        {
                e.startEngine();
                cout<<endl<<" The car is moving"<<endl;
        }
        void stopCar()
        {
                e.stopEngine();
                cout<<endl<<" The car has stopped"<<endl;
        }
};
```

Example program of “has-a “ relationship:

```
void main()
{
        Car car;
        car.startCar();
        cout<<endl<<endl<<endl;
        car.stopCar();

        getch();
}
```

Has-a relationship:

Here, since one object is composed of or contained in another one, this relationship is also called containership.

This relationship between classes is also referred to as aggregation.

Q1. Define a class **Address** that has attributes country, province number, municipality name and ward number. Define a class Student with attributes relative to student. Write a program to use aggregation to represent “Student has Address”.

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor : Er. Rabina Chaudhary

Errors:

- Syntax error: occurs due to typing mistake in syntax or writing wrong syntax

- Semantic error: program having logical errors is compiled and executed successfully but doesn’t produce expected output
- Linker error: program is compiled successfully but error occur during linkage phase example when function prototype is written and that function is called in the program, but there is no function definition

Run time errors:

- Errors which occur when the program is running (during execution time)
- The program having runtime error is compiled successfully but the error occurs while execution of the program
- A program crash in unexpected way while running is caused by run time error

Exception handling :

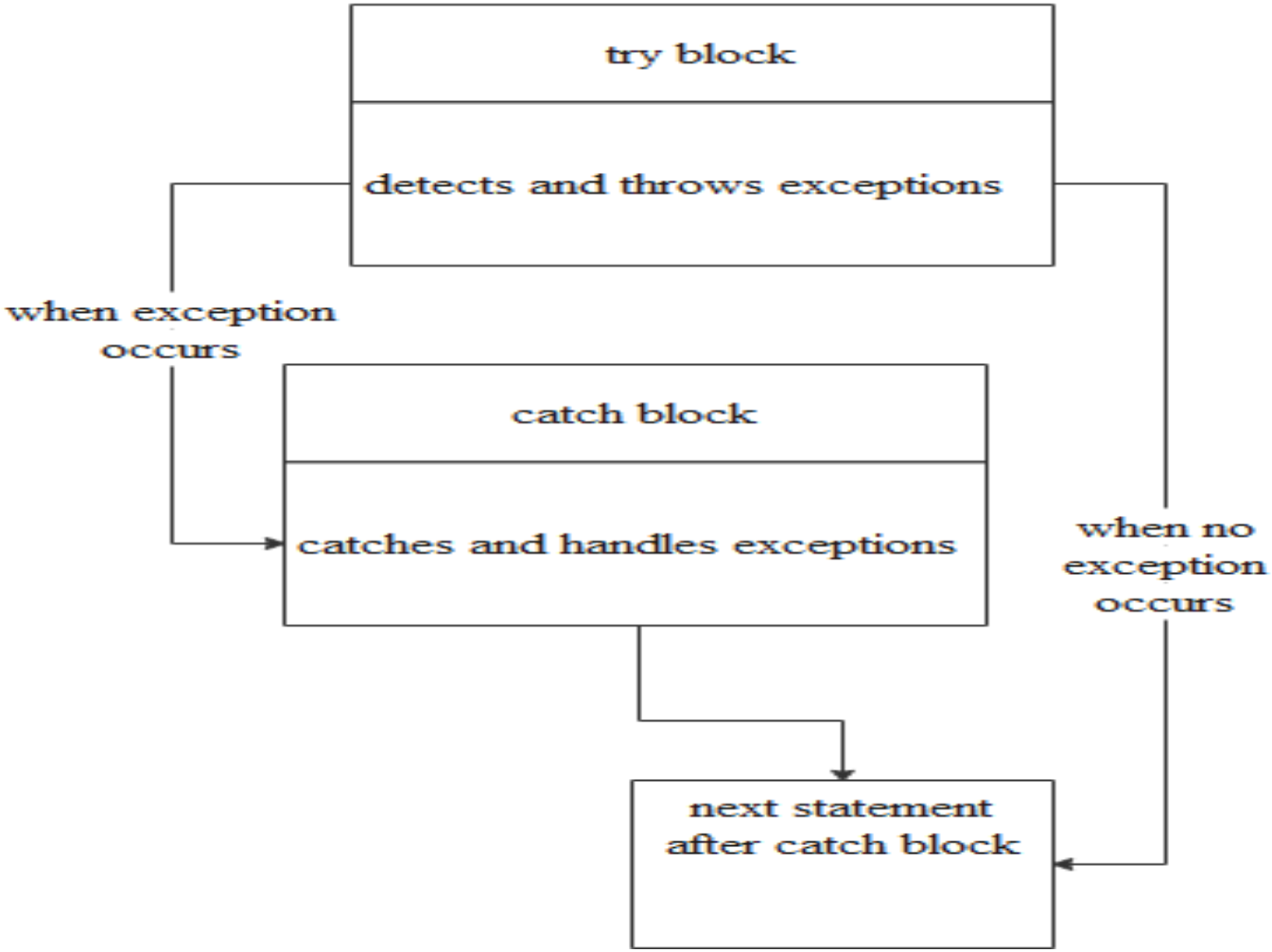
- C++ has mechanism to handle run time error called as exception handling mechanism
- Exception handing mechanism is responsible to detect and report exceptional circumstances so that appropriate action can be taken
- Exceptional handling mechanism performs the following tasks

1. Find the problem (hit the exception)
2. Inform that error has occurred (throw the exception)
3. Receive the error information (catch the exception)
4. Take corrective actions (handle the exception)

- Error handling code consists of two segments

1. One segment to detect errors and throw exceptions
2. The other segment to catch the exception and take appropriate actions

Exception handling mechanism consists of three key words



General format of try, throw and catch:

```
…
 try
  {
        …
        throw exception;
        …
  }
 catch( type argument)
  {
        //block of statements that handles the exception
  }
```

try block:

- try block is a block of code for which particular exceptions will be analyzed and detected
- try block encloses the block of code that we want to analyze for exception
- If the try block detects exceptions, the exception is thrown to catch block using throw key word and the program control also is transferred to catch block
- If no error is found in try block, the control never goes into catch block

Syntax:

```
try
{
        //code to be monitored for execution
        throw exception;
}
```

catch block:

- Syntax

```
catch (type argument)
{
        //block of statements that handles the exception
}
```

- Type is the type of exception that it handles and argument is parameter name
- Catch statement catches an exception whose type matches with the type of catch argument
- When the exception is caught, the code in the catch block is executed

throw statement:

- An exception is thrown to a particular catch block using throw keyword
- When the exception is thrown, it will be caught by the catch statement associated with the try block
- When the try block throws an exception, the program control is transferred to the catch block

Program to handle division by zero exception.

```
#include<iostream.h>
#include<conio.h>

void main()
{
        char error[]="ERROR!!! division by zero!!!";
        float num1,num2,result;
        cout<<"Enter two numbers :";
        cin>>num1>>num2;
  try
  {
        if(num2!=0)
        {
        result =num1/num2;
        cout<<"The result after division is "<<result<<endl;
        }
        else
                throw error;
  }
catch(char e[] )
  {
                cout<<"The denominator must be a non zero"<<endl<<e<<endl;
  }
  cout<<endl<<"This is last statement "<<endl;
  getch();
}
```

Multiple Catch Block:

- It is possible for a program segment to have more than one condition to throw an exception
- In this case, we can associate more than one catch statement with a try statement
- To handle different types of situation, we can specify two or more catch blocks, each catching different types of exceptions
- When exception is thrown, each catch statement is inspected in order and first one whose type matches with that of exception is executes and other catch blocks are ignored and execution continues after the try catch blocks

Multiple catch block

```
try
        {
                //try block
        }
catch(type1 argument)
        {
                //catch block1
        }
catch(type2 argument)
        {
                //catch block2
        }
catch(typeN argument)
        {
                //catch blockN
        }
```

Example program:

```
#include<iostream>
using namespace std;
int main()
{
        int array[5]={1,2,3,4,5};
        int deno,size;
        try
        {
                cout<<"Enter size of array!!!";
                cin>>size;
                if(size>5)
                {
                throw 1;
                }

                cout<<"Enter denominator value :";
                cin>>deno;
                if(deno==0)
                {
                throw 'a';
                }
                for(int i=0;i<size;i++)
                {
                cout<<"the result is "<<array[i]/deno<<endl;
                }
        }
```

Program continued…

```
catch(int a)
        {
                cout<<"Exception Caught!!!"<<a<<endl;
        }
catch(const char a[])
        {
                cout<<"Exception Caught!!!"<<a<<endl;
        }

        return 0;

}
```

Catching all types of exceptions by single catch block:

- We can use a single catch block to catch all types of exception

Syntax to define catch block to catch all types of exception:

```
catch(…)
{
        //statements to process all the exceptions
}
```

Example program:

```
#include<iostream>
using namespace std;
int main()
{
        int array[5]={1,2,3,4,5};
        int deno,size;
        try
        {
                cout<<"Enter size of array!!!";
                cin>>size;
                if(size>5)
                {
                throw 1;
                }

                cout<<"Enter denominator value :";
                cin>>deno;
                if(deno==0)
                {
                throw 'a';
                }
                for(int i=0;i<size;i++)
                {
                cout<<"the result is "<<array[i]/deno<<endl;
                }
        }
```

Program Continued…

```
catch(...)
{
cout<<"Exception Caught"<<endl;
}

return 0;
}
```

Object Oriented Programming in C++

Nepal College of Information Technology

Course Instructor: Er. Rabina Chaudhary

Chapter 7 : Object Oriented Design

Responsibility implies non-interference:

- Responsibility implies a degree of independence or non-inference
- Example: when you send flowers to your grandfather at Pokhara, it was not necessary to think how the request would be served. The florist, having taken the responsibility for this service, is free to operate without inference from the customer

- In Conventional programming one portion of code is dependent on many other portions of the code of system
- Such dependencies arise through use of global variables, pointer variables or inappropriate use of and dependencies of implementation details on other portion of code

- Responsibility driven design make the portion of code as independent as possible
- Responsibility driven design support information hiding ans it becomes more important when one move from programming in small to programming in large

Programming in Small:

- Code is developed by a single programmer or very small number of programmers
- A single individual can understand all the aspects of a project

Programming in large:

- Software system is developed by a large team
- Team consists of people with many different skills
- A single individual may not necessarily understand all aspects of the project
- Major problem in software development is team management and communication of information between different parts of the project

Role of behavior in OOP: why begin with behavior

- Object oriented design process begins with the analysis of behavior
- Behavior can be described from the moment an idea is conceived and can be described in terms meaning to both the programmers and the client

Responsibility Driven Design:

- Developed by Rebecca Wirfs- Brock
- It is an object oriented design technique that is driven by a emphasis on behavior at all levels of development
- In object oriented design, first we establish who is responsible to for which action of the system
- The responsible class to carry out an action may communicate with other classes to complete its responsibility

- Responsibility is anything that a class knows or does

- Responsibility is anything that a class knows or does
- Example:

class Student might have knowledge for its data members : registration number, roll number, name, address, faculty

and it may perform actions like attend class, take exam, view result, pay fee

Collaborator:

- Collaborator is another class that is used to interact with a class
- Collaborator for a class is also a class which helps to perform first class’s action

Class Responsibility Collaborator:

- CRC card:

➢ Identify class

➢ Identify list of responsibility

➢ List collaboration

Class Responsibility Collaborator card:

- Consists of three sections:

1. Class : a class represents a collection of similar objects
2. Responsibility: it is something that a class knows or does
3. Collaborator: another class that is a class interacts with to fulfill its responsibilities

CRC card:

component	
Responsibilities	Collaborators

CRC card for Student:

Student	
Responsibilities name roll number registration number attend class fill examination form take exam request for transcript pay fee borrow book return book	**Collaborators** ExamDepartment AccountDepartment Library

CRC card for Library:

Library	
Responsibilities book details lend book receive book track record	Collaborators Student

Identification of Component:

- The complex system is decomposed into smaller units
- A component is simply an entity that can perform tasks which fulfills some responsibilities
- A component can be function, structures or class or collection of other components
- The identification of component takes place during the process of imagining the execution of a working system

Documentation:

- Two document that are essential part of any software system
    1. user manual
    2. system design documentation

Coupling and Cohesion:

- If any services provided by component is done within the component then it is cohesive
- If the component cannot do all the work itself and interacts with other components to do the work then it is coupling
- It is desirable to reduce the amount of coupling as much as possible.
- Coupling is not good because of dependency in the components, if problem in one, the problem occurs in another too

Interface and Implementation:

- It is possible for one programmer to know how to use a component developed by another programmer, without needing to know how the component is implemented
- The interface view is the side seen by other programmers. It describes what a software component can perform
- The implementation view is the side seen by the programmer working on a particular component. It describes how a component is performing a task

Formalize the interface:

- Component with only one behavior can be made into function
- Components with many tasks are implemented as classes
- Names given to each of the responsibilities identified on the CRC card for each component are mapped into function names along with function name, the argument to be passed to the function is identified
- If the component requires some data to perform a specific task, the source of the data, either through argument or global value or maintained internally by the component, must be clearly identified

Coping with names:

- The selection of useful name is important
- Names should be meaning and according to the context of the problem
- Use pronounceable name
- Use capitalization or underscore to mark beginning of a new word within a name

  use BankAccount, Bank_account rather than bankaccount

- Avoid names with multiple interpretations
- Avoid digits within a name (0 can be misread as O, 2 can be misread as Z, 5 can be misread as S]
- Name functions and variables that yield Boolean values so they describe the interpretation of true and false value

  StudentIsPresent clearly indicates that a true value and student is present

  StudentStatus does not give clear indication

- Once the names have been developed for each activity, the CRC cards for each component are redrawn with the names and formal arguments

Designing the Representation:

- The design teams are divided into groups, where each group is responsible for one or more software components

- The description of behavior must be transformed into algorithm

Implementing Components:

- After designing each software subsystem, the next step is to implement the desired activities in a computer language

Integration of Components:

- Once the software subsystem are individually designed and tested, they can be integrated into a final product

Sequence diagram:

- Sequence diagrams is used to model the interactions between objects in a single use case
- They illustrate how the different parts of a system interact with each other to carry out a function.

Sequence diagram:

- **Lifeline**

A lifeline represents an individual participant in the Interaction

- **Activations**

A thin rectangle on a lifeline) represents the period during which an element is performing an operation

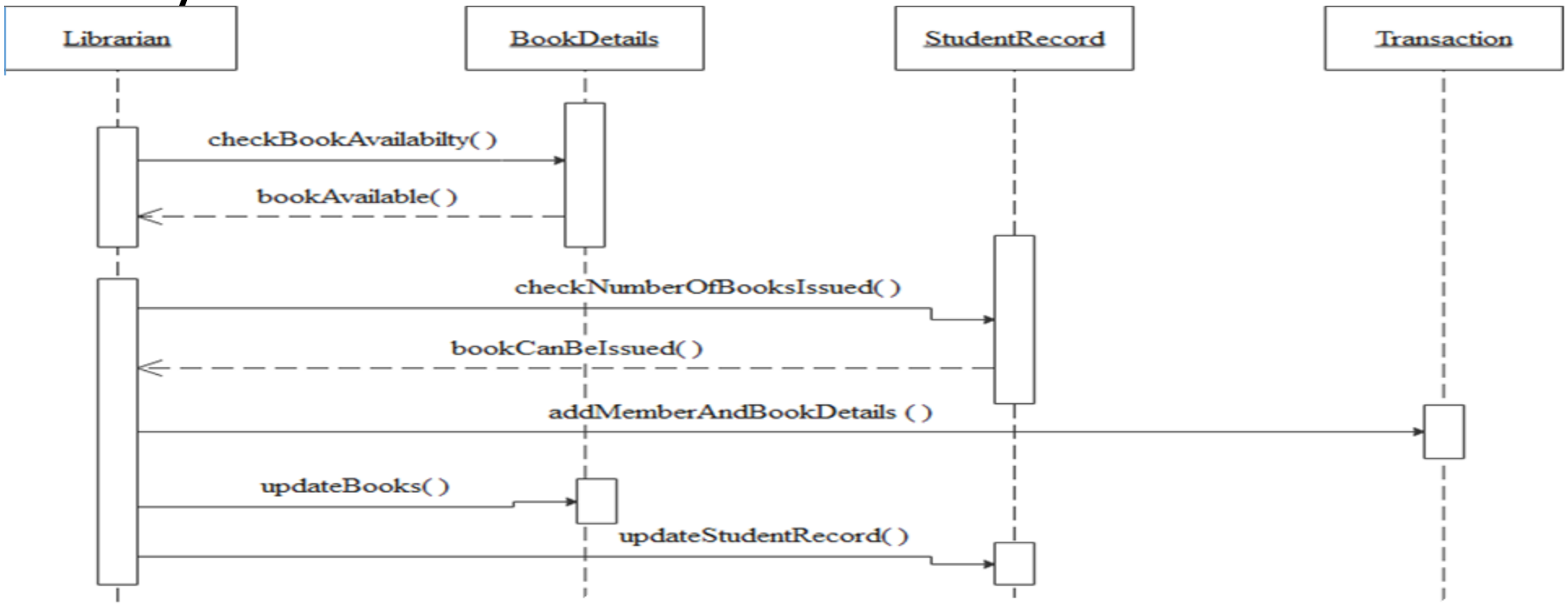
- **Call Message**

A message defines a particular communication between Lifelines of an Interaction

- **Return Message**

Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message

Sequence diagram of borrowing book from library:



Practice:

Q1. Design a CRC card of ATM card reader

Q2. Design a sequence diagram to withdraw money from ATM