CHAPTER 2

Inheritance and Polymorphism

Objectives ...

After learning this chapter you will be able:

- To understand the concept of Inheritance.
- To study how to define Base class and Derived class.
- To learn about different Types of Inheritance.
- To study Abstract class.
- To learn Polymorphism.

2.1 INHERITANCE [S-19, W-22]

- Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behavior and overriding these with capabilities the new classes requires.
- Inheritance is the process by which one object can acquire the properties of another object. Inheritance is the process of creating new classes from an existing class.
- The existing class is known as base class and the newly created class is called as a derived class. The derived class inherits all capabilities of the base class. A derived class is more specific than its base class and represents a smaller group of objects.

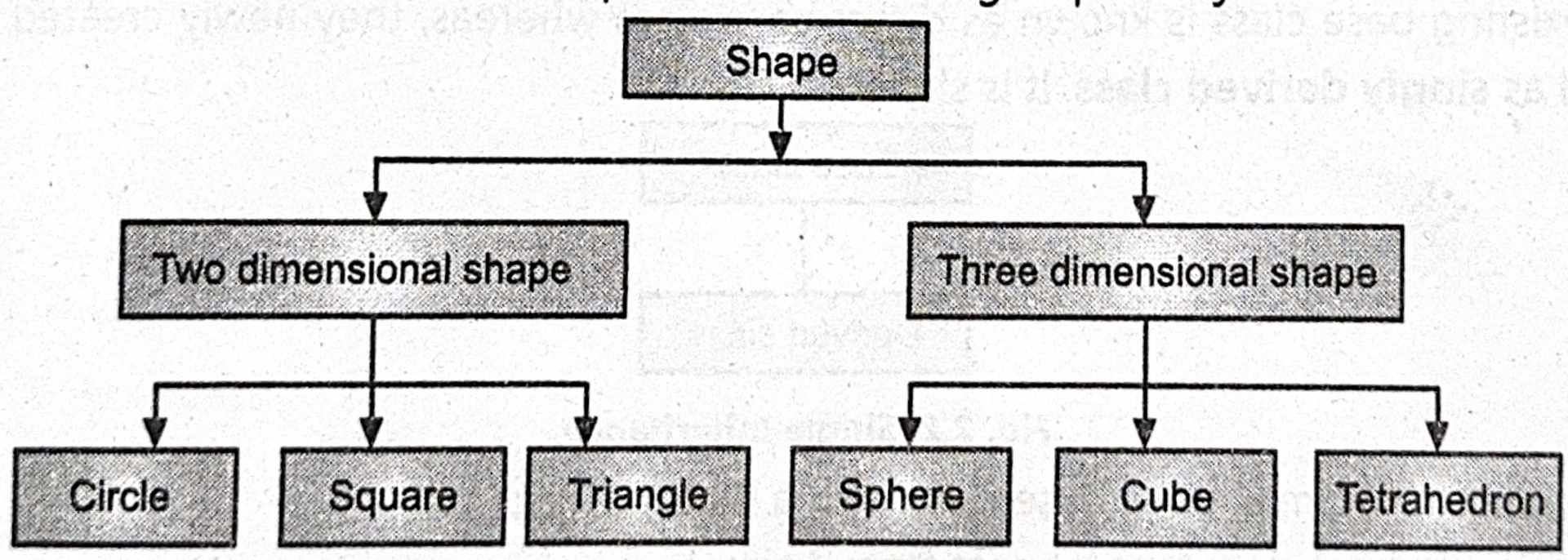

Fig. 2.1: 'Shape' Hierarchy

- For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class. Hence, inheritance supports the concept of classification; with the help of classification, an object need to define only those qualities that make it unique within its class.
- The inheritance makes it possible for one object to be a specific instance of a more general case. Example of inheritance hierarchy of shape is shown in Fig. 2.1.
- The main advantages of inheritance are: [W-23]
  1. It helps to implement software reusability. Once the hierarchy is done, the specific data to the particular class can be added as per the specification of the derived classes in hierarchy. Reusing existing code saves time and money.
  2. It increases the reliability of the code.
  3. Inheritance can also help in the original conceptualization of a programming problem and in the overall design of the program.
  4. It is useful to avoid redundancy, leading to smaller models that are easier to understand.
  5. It adds some enhancements to the base class.
  6. We can create new class based on base class without modifying base class.

2.2 TYPES OF INHERITANCE [S-18, 23]

- There are many ways in which we can create derived class. There are five types of inheritance as follows:
  1. Single inheritance
  2. Multilevel inheritance
  3. Multiple inheritance
  4. Hierarchical inheritance and
  5. Hybrid inheritance

2.2.1 Single Inheritance [S-18, W-23]

- When one base class is derived by one child class, then it is known as single inheritance. The existing base class is known as direct base class whereas, they newly created class is called as **singly derived class**. It is shown in Fig. 2.2.

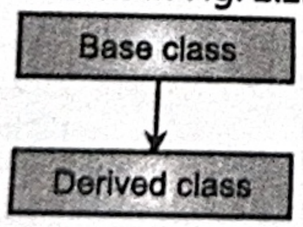

Fig. 2.2: Single Inheritance

- ***For Example,*** Animal is the base class with a method eat().
  Cat is the derived class that inherits from Animal.
  Using myCat, you can access both the base class method (eat) and the derived class method (meow).

Program 2.1: Program to illustrate single inheritance.

```
#include <iostream>
using namespace std;
// Base class
class Animal
{
   public:
   void eat()
   {
      cout << "This animal eats food." << endl;
   }
};
// Derived class
class Cat : public Animal
{
   public:
   void meow()
   {
      cout << "The cat meows." << endl;
   }
};
int main()
{
   Cat myCat;                    //create object
   myCat.eat();                  // Call function from base class
   myCat.meow();          // Call function from derived class
   return 0;
}
```

Output:

```
This animal eats food.
The Cat meowss.
```

Example 1: Suppose a college want to store information of undergraduate (UG) and postgraduate students (PG). Both are having registration no., name, and address. Each postgraduate student also have additional attributes along with above attributes are placement company name. It can show as in Fig. 2.3.

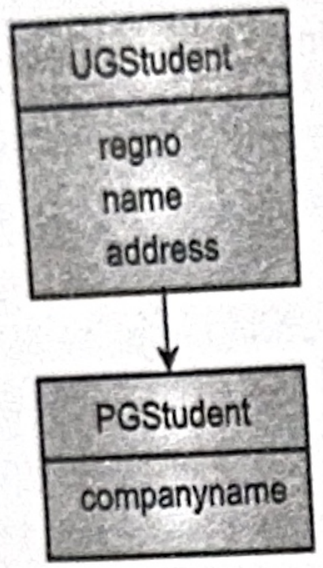

Fig. 2.3

Program 2.2:

```
#include<iostream>
using namespace std;
class student
{
    protected:
        int regno;
        char name[20];
        char addr[30];
    public:
        void getdata()
        {   cout<<"enter registration no, name and address";
            cin>>regno>>name>>addr;
        }
        void display()
        {
            cout<<"registration no.= "<<regno;
            cout<<"name = "<<name;
            cout<<"address = "<<addr;
        }
};   //end of base class
class PGstudent: public student
{
  private:
    char compname[20];
```



```
  public:
    void get_PG_data()
    {
        getdata();
        cout<<"\nEnter the company name=";
        cin>>compname;
    }
    void dispPG()
    {
        cout<<"\n registration no. ="<<regno;
        cout<<"\n name ="<<name;
        cout<<"\n address ="<<addr;
        cout<<"\n company name = "<<compname;
    }
};
int main()
{
  student s;
  PGstudent ps;
  s.getdata();
  s.display();
  ps.get_PG_data();
  ps.dispPG();
}
```

Output:

```
Enter registration no, name and address
101 Ram Pune
registration no = 101   name = Ram     addr = Pune
Enter registration no., name & address
401 Sita Pune
Enter the company name = zenex
Registration no = 401
name = Sita
address = Pune
Company name = Zenex
```

Program 2.3: A base class contains the data of employee name, empno and sex. The derived class contains basic salary. The derived class has been declared as an array of class objects. The program illustrates the single inheritance.

```
#include <iostream>
using namespace std;
class base
{
    private:
        char name[20];
        int empno;
        char sex;
    public:
        void get();
        void disp();
};
class derived : public base
{
    private:
        float bsalary;
    public:
        void get();
        void disp();
};
void base::get()
{
    cout << "\nEnter name: ";
    cin >> name;
    cout << "Enter Employee ID: ";
    cin >> empno;
    cout << "Enter Sex (M/F): ";
    cin >> sex;
}
void base::disp()
{
    cout << name << "\t" << empno << "\t" << sex << "\t";
}
```



```
void derived::get()
{
    base::get();
    cout << "Enter Basic Salary: ";
    cin >> bsalary;
}
void derived::disp()
{
    base::disp();
    cout << bsalary << endl;
}
int main()
{
    derived ob[10];
    int n;
    cout << "How many employees? ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        cout << "\nEnter details for employee " << (i + 1) << ":\n";
        ob[i].get();
    }
    cout << "\nName\tEmpNo\tSex\tSalary\n";
    for (int i = 0; i < n; i++)
    {
        ob[i].disp();
    }
    return 0;
}
```

Output:

```
How many employees? 2

Enter name: Prajakta
Enter Employee id: 101
Enter Sex: f
Enter basic salary: 10000

Enter name: Pranjal
Enter Employee id: 102
Enter Sex: m
Enter basic salary: 40000

name        Empno        sex      bsalary
name =abc   Emp no. is =101   sex=f    basic salary = 10000
name =xyz   Emp no. is =102   sex=m    basic salary = 40000
```

2.2.2 Multilevel Inheritance

[S-23]

- When a base class is derived by a child class which further derived by another child class, then it is known as Multilevel Inheritance.
- The class which provides link between two classes is known as intermediate base class. This is illustrate in Fig. 2.4.

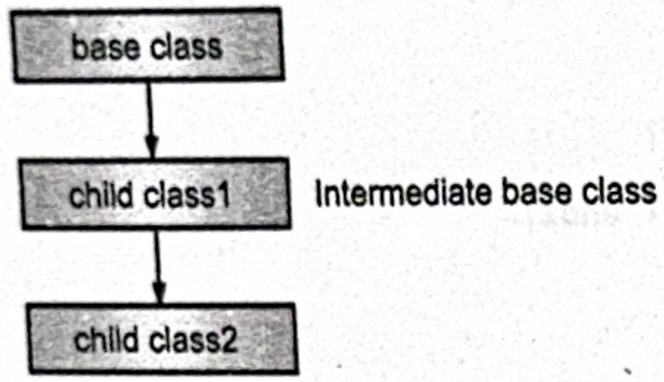

Fig. 2.4: Multilevel Inheritance

Practically, we can have more than two levels also.

- **Syntax of multilevel inheritance:**

```
class base
{
    .......
    .......
};
class child1: public base
{
    .......
    .......
};
class child2: public child1
{
    .......
    .......
};
```

Program 2.4:

```
#include <iostream>
using namespace std;
class Animal                    // Base class
{
    public:
    void eat()
    {
        cout << "Animal eats food." << endl;
    }
};
```



```
// Derived class from Animal
class Mammal : public Animal
{
    public:
    void walk()
    {
        cout << "Mammal walks on land." << endl;
    }
};
// Derived class from Mammal (Multilevel Inheritance)
class Cat : public Mammal
{
    public:
    void meow()
    {
        cout << "Cat meows." << endl;
    }
};
int main()
{
    Cat myCat;
    // Accessing members from all levels of inheritance
    myCat.eat();    // From Animal
    myCat.walk();   // From Mammal
    myCat.meow();   // From Cat
    return 0;
}
```

Output:

```
Animal eats food.
Mammal walks on land.
Cat meows.
```

Example 2: Consider an example of bank. In a bank, different customers have savings account. Some customers may have taken loan from the bank. So the bank always maintains information about bank depositors and borrowers. This can be shown using the Fig. 2.5.

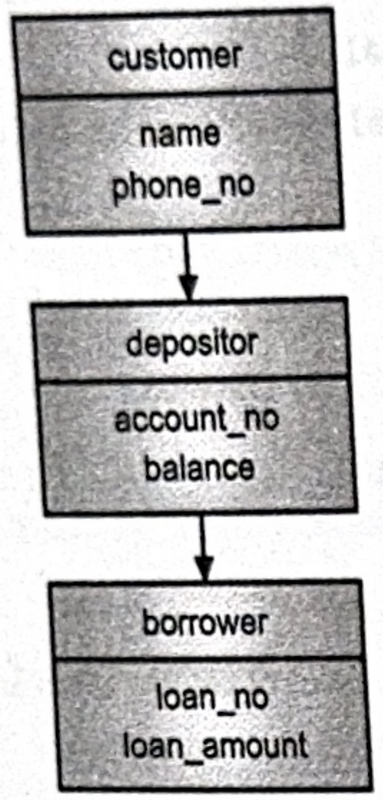

Fig. 2.5: Example of Bank

Program 2.5: Program to illustrate use of multilevel inheritance.

```
#include<iostream>
using namespace std;
class customer        // base class
{
    protected:
        char name[30];
        int phone_no;
}; // end of base class
class depositor: public customer      // child class1
{
    private:
        int account_no;
        float balance;
    public:
        void get_depositor_data( )
        {
            cout<<"\n Enter name & phone no ";
            cin>>name>>phone_no;
            cout<<"\n Enter account no and balance ";
            cin>>account_no>>balance;
        }
```



```
        void display_depositor_data( )
        {
            cout<<"\n Name = "<<name;
            cout<<"\n Phone no = "<<phone_no;
            cout<<"\n Account no = "<<account_no;
            cout<<"\n Balance = "<<balance;
        }
}; // end of child class1
class borrower: public depositor                // child class2
{
    protected:
        int loan_no;
        float loan_amount;
    public:
        void get_loan_data( )
        {
            cout<<"\n Enter loan no and loan amount";
            cin>>loan_no>>loan_amount;
        }
        void display_loan_data( )
        {
            cout<<"\n Loan no = "<<loan_no;
            cout<<"\n Loan amount = "<<loan_amount;
        }
}; // end of child class2
int main( )
{
    int choice;
    cout<<"1-Read & display depositor information "<<endl;
    cout<<"2-Read & display depositor & borrower information"<<endl;
    cout<<"Enter your choice = ";
    cin>>choice;
    switch (choice)
    {
        case 1:
            depositor d;
            d.get_depositor_data( );
            d.display_depositor_data( );
            break;
```

```
        case 2:
            borrower b;
            b.get_depositor_data( );
            b.get_loan_data( );
            b.display_depositor_data( );
            b.display_loan_data( );
            break;
    }
}  // end of main()
```

Output:

```
1-Read & display depositor information
2-Read & display depositor & borrower information
Enter your choice = 1
Enter name & phone no Prajakta 98776677
Enter account no and balance 123456789 10000
Name = Prajakta
Phone no = 98776677
Account no = 123456789
Balance = 10000
```

- In the program 2.5, though the data members of depositor class are private, they can be accessed indirectly by the borrower class through the member functions of the depositor class indirectly.

Program 2.6: Consider the example of student data. Base class is student. Class test is intermediate base class and result is child class. The class result inherits the details of marks obtained in the test and the roll no. of student from base class through multilevel inheritance.

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



```
        void display()
        {
            cout<<"roll="<<roll;
        }
};
class test: public student
{
    protected:
        int mark1, mark2;
    public:
        void getmark(int a, int b)
        {
            mark1 = a;
            mark2 = b;
        }
        void dispmark()
        {
            cout<<"mark1 = \n"<<mark1<<"mark2 ="<<mark2;
        }
};
class result: public test
{
    private:
        int total;
    public:
        void displayall()
        {
            total = mark1 + mark2;
            display();
            dispmark();
            cout<<"total="<<total;
        }
};
```

```
int main()
{
    result r;
    r.get(10);
    r.getmark(60,75);
    r.displayall();
}
```

Output:

```
roll = 10
mark1 = 60
mark2 = 75
total = 135
```

2.2.3 Multiple Inheritance [W-18, 22]

- When more than one base class is inherited by a derived class, then such type of inheritance is called as multiple inheritance. Fig. 2.6 illustrates the point.

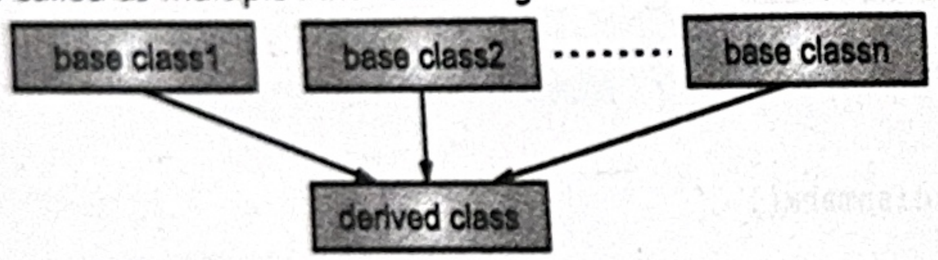

Fig. 2.6: Multiple Inheritance

- **For example,** A child inherits the features from father and intelligence from mother.

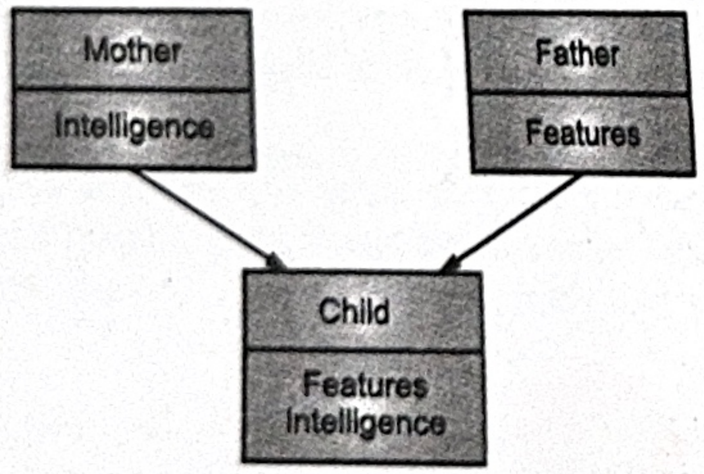

Fig. 2.7: Multiple Inheritance

- ***Syntax of Multiple Inheritance:***

1. **Multiple inheritance with all public derivation:**

```
class base1
{
    .....
    .....
    .....
};
```



```
class base2
{
    .....
    .....
    .....
};
class derived: public base1, public base2
{
    .....
    .....
    .....
};
```

Here access specifier may be public or private and the base classes are separated by commas.

2. **Multiple inheritance with all private derivation:**

```
class base1
{
    .....
    .....
    .....
};
class base2
{
    .....
    .....
    .....
};
class derived: private base1, private base2
{
    .....
    .....
    .....
};
```

3. **Multiple inheritance with all mixed derivation:**

```
class base1
{
    ......
    ......
    ......
};
class base2
{
    ......
    ......
    ......
};
class base3
{
    ......
    ......
    ......
};
class derived: pubic base1, public base2, private base3
{
    ......
    ......
    ......
};
```

Let us discuss the program how multiple inheritance works.

Program 2.7: Animal and Pet are two independent base classes. And Cat is a derived class that inherits from both.

```
#include <iostream>
using namespace std;
// First base class
class Animal
{
    public:
    void eat()
    {
        cout << "Animal eats food." << endl;
    }
};
```



```
// Second base class
class Pet
{
    public:
    void play()
    {
        cout << "Pet plays with owner." << endl;
    }
};
// Derived class inheriting from both Animal and Pet
class Cat : public Animal, public Pet
{
    public:
    void meow()
    {
        cout << "Cat meows." << endl;
    }
};
int main()
{
    Cat myCat;
    // Accessing functions from both base classes
    myCat.eat();    // From Animal
    myCat.play();   // From Pet
    myCat.meow();   // From Cat
    return 0;
}
```

Output:

```
Animal eats food.
Pet plays with owner.
Cat meows.
```

Program 2.8: With private class.

- Cat inherits from Animal and Pet privately.
- eat() and play() become private in Cat, so you can't call them directly from main().

```
#include <iostream>
using namespace std;
// First base class
class Animal
{
    public:
    void eat()
    {
        cout << "Animal eats food." << endl;
    }
};
// Second base class
class Pet
{
    public:
    void play()
    {
        cout << "Pet plays with owner." << endl;
    }
};
// Derived class inherits privately from both
class Cat : private Animal, private Pet
{
    public:
    void meow()
    {
        cout << "Cat meows." << endl;
    }
    // Public method to access base class methods
    void showBehaviors()
    {
        eat();   // Call Animal's method
        play();  // Call Pet's method
    }
};
```



```
int main()
{
    Cat myCat;
    myCat.meow();              // Allowed
    myCat.showBehaviors();     // Indirect access to base class functions
    // The following lines will cause errors if uncommented:
    // myCat.eat();            // ERROR: eat() is private in Cat
    // myCat.play();           // ERROR: play() is private in Cat
    return 0;
}
```

Output:

```
Cat meows.
Animal eats food.
Pet plays with owner.
```

Program 2.9: Program for multiple inheritance.

```
#include<iostream>
using namespace std;
class basex
{
    protected:
        int x;
    public:
        void get_x (int a) {x = a;}
};
class basey
{
    protected:
        int y;
    public:
        void get_y (int b) {y = b;}
};
class derived: public basex, public basey
{
    public:
        void display()
        {
            cout<<"x = " <<x<<"\n";
            cout<<"y = "<<y<<"\n";
            cout<<"x+y = "<<x+y<<"\n";
        }
};
```

```
int main()
{
    derived ob;
    ob.get_x(50);
    ob.get_y(60);
    ob.display();
}
```

Output:

```
x = 50
y = 60
x + y = 110
```

Program 2.10: Consider an example to print bio-data of a diploma student having personal and academic information. Program to illustrate use of multiple inheritance.

```
#include<iostream>
#include<cstring>
using namespace std;
class personal                          // base class1
{
    private:
        char name [30], email_id[50];
    protected:
    void get_personal_info( )
    {
        cout<<"\n Enter name & email id of a person = ";
        cin>>name>>email_id;
    }
    void display_personal_info( )
    {
        cout<<"\n Name = "<<name;
        cout<<"\n Email_id = "<<email_id;
    }
};  // end of base class1
class academic                          // base class 2
{
  private:
    float tenth_marks;
    char tenth_class[20];
```



```
protected:
    void get_academic_info( )
    {
        cout<<"\n Enter tenth marks";
        cin>>tenth_marks;
    }
    void find_class( )
    {
        if (tenth_marks>=70)
            strcpy (tenth_class, "Distinction");
        else
        if (tenth_marks>=60)
            strcpy (tenth_class, "First class");
        else
        if (tenth_marks>=50)
            strcpy (tenth_class, "Second class");
        else
        if (tenth_marks>=40)
            strcpy (tenth_class, "Pass class");
        else
            strcpy (tenth_class, "Fail");
    }
    void display_academic_info( )
    {
        cout<<"\n Tenth marks = "<<tenth_marks;
        find_class( );
        cout<<"\n Tenth class = ";
        cout<<tenth_class;
    }
}; // end of base class2
class bio_data: private personal, private academic      // Derived class
{
  public:
    void display_biodata( );
    void get_info( );
}; // end of derived class
  // Member functions of derived class
```

```
void bio_data::display_biodata( )
{
    cout<<"BIO DATA "<<endl;
    cout<<"Personal info = "<<endl;
    display_personal_info( );
    cout<<"Academic info= "<<endl;
    display_academic_info( );
}
void bio_data::get_info( )
{
    get_personal_info( );
    get_academic_info( );
}
int main( )
{
    bio_data b;
    // First read info
    b.get_info( );
    // Now print biodata
    b.display_biodata( );
}
```

Output:

```
Enter name & email id of a person = Prajakta abc@yahoo
Enter tenth marks 80.5
BIO DATA
Personal info =
Name = Prajakta
Email_id = abc@yahoo
Academic info=
Tenth marks = 80.5
Tenth class = Distinction
```

Ambiguity in Multiple Inheritance: [W-18]

- One common cause of ambiguity in multiple inheritance is when two or more base classes have methods with the same name, while there is no method of that name in the derived class. In this case objects of the derived class have no way of knowing which of the parent methods is to be executed. The scope resolution operator can be used to resolve the ambiguity.

- Another element of ambiguity in multiple inheritance arises when two or more derived classes inherit the same member from a common base class. If now another derived class inherits from these classes, two copies of the original member function could be inherited.
- Fig. 2.8 shows one possible scenario that leads to inheritance of multiple copies.
- One possible solution to this duplicity is the use of the scope resolution operator. Another solution of this problem is to declare the base classes to be virtual at the time of they are inherited.

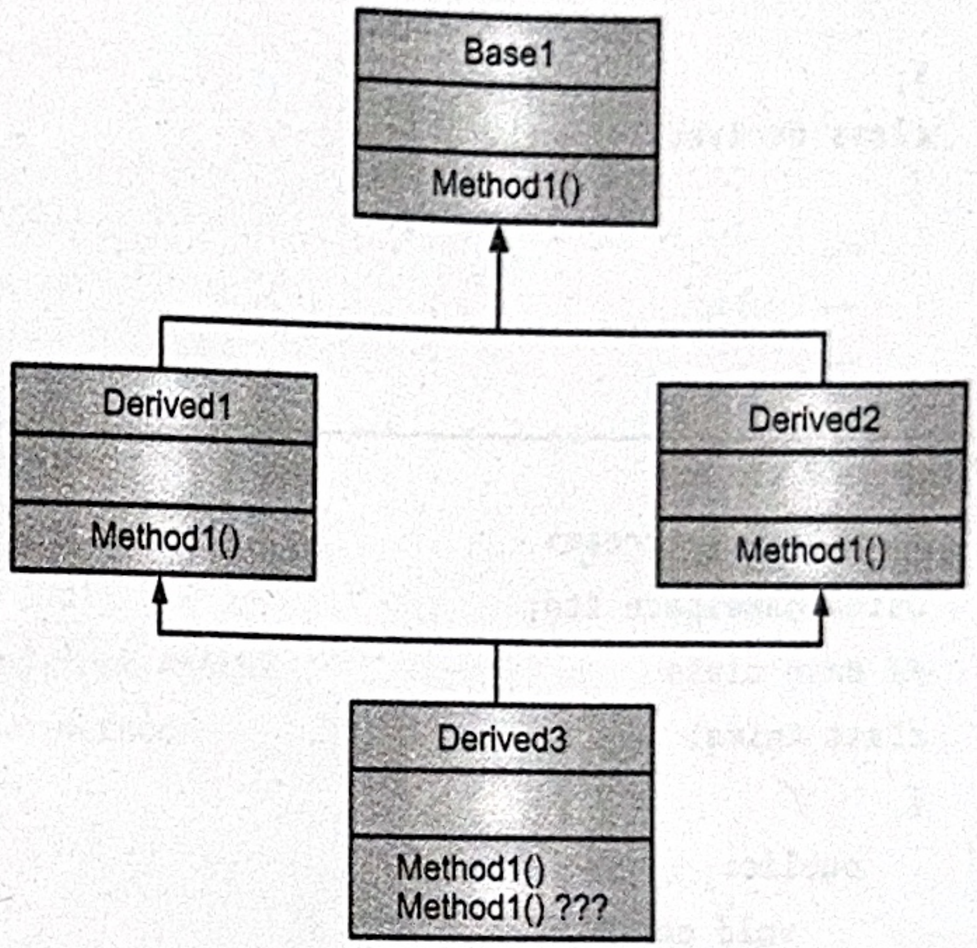

Fig. 2.8: Inheritance of Multiple Copies

Advantages of Multiple Inheritance:

1. Derived class typically represents a combination of its base classes.
2. It has rich semantics
3. It has ability to directly express complex structures.

2.2.4 Hierarchical Inheritance

- When one base class is inherited by more than one derived classes, it is known as hierarchical inheritance. In this, many programming problems can be cast into a hierarchy where certain features of one level are shared by many other below that level. The Fig. 2.9 illustrates this point.

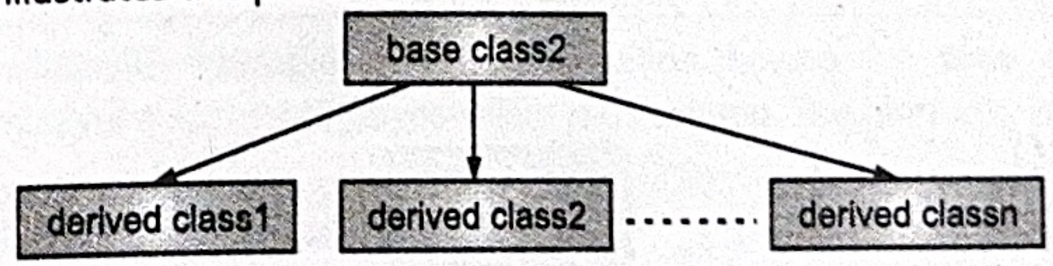

Fig. 2.9: Hierarchical Inheritance

- **Syntax of Hierarchical Inheritance:**

```
class base
{
    ......
    ......
    ......
};
```

```
class derived1: public base
{
    .....
    .....
    .....
};
class derived2: public base
{
    .....
    .....
    .....
};
```

Program 2.11:

```
#include <iostream>
using namespace std;
// Base class
class Animal
{
    public:
        void eat()
        {
            cout << "Animal eats food." << endl;
        }
};
// Derived class 1
class Dog : public Animal
{
    public:
    void bark()
    {
        cout << "Dog barks." << endl;
    }
};
// Derived class 2
class Cat : public Animal
{
  public:
```



```
  void meow()
  {
     cout << "Cat meows." << endl;
  }
};
int main()
{
  Dog myDog;
  Cat myCat;
  cout << "Dog Object:" << endl;
  myDog.eat();   // Inherited from Animal
  myDog.bark();  // Dog's own method
  cout << "Cat Object :" << endl;
  myCat.eat();   // Inherited from Animal
  myCat.meow();  // Cat's own method
  return 0;
}
```

Output:

```
Dog Object:
Animal eats food.
Dog barks.
Cat Object:
Animal eats food.
Cat meows.
```

Example 3: Consider an example of an organisation having full time and part time employees. Depending on it, the employee salary is calculated. The hierarchy is:

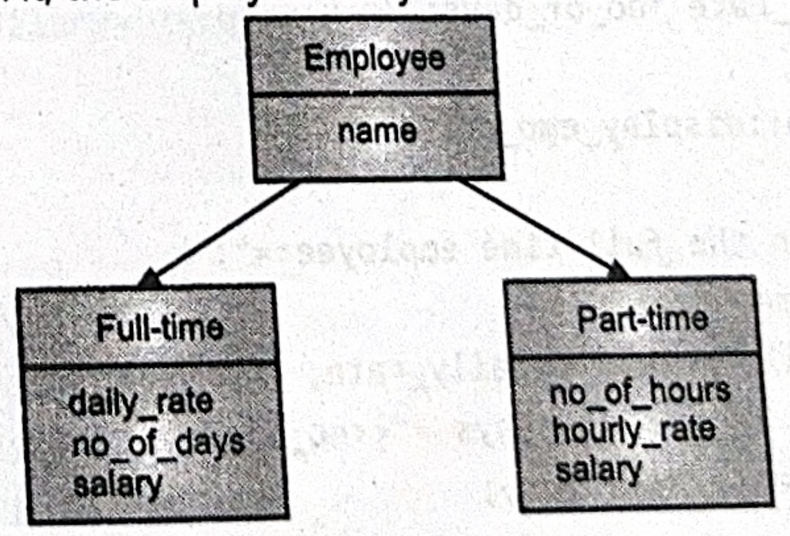

Fig. 2.10: Hierarchical Inheritance

Program 2.12: Program to illustrate use of hierarchical inheritance.

```
#include<iostream>
using namespace std;
class employee   // base class
{
   protected:
      char name[30];
}; // end of base class
class full_time_emp: protected employee
{
   protected:
      float daily_rate, salary;
      int no_of_days;
   public:
      void read_emp_data( );
      void display_emp_data( );
      void calculate_salary( );
}; // end of derived class1
void full_time_emp::read_emp_data( )
{
   cout<<endl<<"Enter name of employee =";
   cin>>name;
   cout<<endl<<"Enter working rate for one day = ";
   cin>>daily_rate;
   cout<<endl<<"Enter no. of working days of employee=";
   cin>>no_of_days;
}
void full_time_emp::calculate_salary( )
{
   salary = daily_rate *no_of_days;
}
void full_time_emp::display_emp_data( )
{
   cout<<endl<<"For the full time employee:=";
   cout<<endl<<"Name="<<name;
   cout<<endl<<"Daily rate ="<<daily_rate;
   cout<<endl<<"No. of working days ="<<no_of_days;
   cout<<endl<<"Salary ="<<salary;
}
```



```
class part_time_emp:protected employee      // derived class2
{
   protected:
      float hourly_rate, salary;
      int no_of_hours;
   public:
      void read_emp_data( );
      void display_emp_data( );
      void calculate_salary( );
}; // end of derived class 2
void part_time_emp::read_emp_data( )
{
   cout<<endl<<"Enter name of employee=";
   cin>>name;
   cout<<endl<<"Enter working rate for one hour =";
   cin>>hourly_rate;
   cout<<endl<<"Enter no. of working hours for employee =";
   cin>> no_of_hours;
}
void part_time_emp::calculate_salary( )
{
   salary = hourly_rate * no_of_hours;
}
void part_time_emp::display_emp_data( )
{
   cout<<endl<<"For the part time employee";
   cout<<endl<<"Name ="<<name;
   cout<<endl<<"Hourly rate ="<<hourly_rate;
   cout<<endl<<"No. of working hours="<<no_of_hours;
   cout<<endl<<"Salary="<<salary;
}
int main( )
{
   int choice;
   cout<<endl<<"You have following choices";
   cout<<endl<<"1-Read & display information of full time employee";
   cout<<endl<<"2-Read & display information of part time employee";
   cout<<endl<<"Enter your choice=";
   cin>>choice;
```

```
switch(choice)
{
    case 1:
            full_time_emp ft;
            // First read data
            ft.read_emp_data( );
            ft.calculate_salary( );
            // Now display data
            ft.display_emp_data( );
            break;
    case 2:
            part_time_emp pt;
            // First read data
            pt.read_emp_data( );
            pt.calculate_salary( );
            // Now display data
            pt.display_emp_data( );
            break;
    }
}
```

Output:

```
You have following choices
1-Read & display information of full time employee
2-Read & display information of part time employee
Enter your choice=1

Enter name of employee = Prajakta
Enter working rate for one day = 250
Enter no. of working days of employee=30

For the full time employee:=
Name=Prajakta
Daily rate =250
No. of working days =30
Salary =7500
```


- In the program 2.12, both derived classes have same names for the member functions. You can also use different names.
- Also in the above program, you can use array of objects to store information about more than one employee.

2.2.5 Hybrid Inheritance

[W-18, S-22]

- Sometimes, we may require to combine two or more types of inheritance to design a program; the result is known as Hybrid Inheritance. Fig. 2.11 illustrates this point.

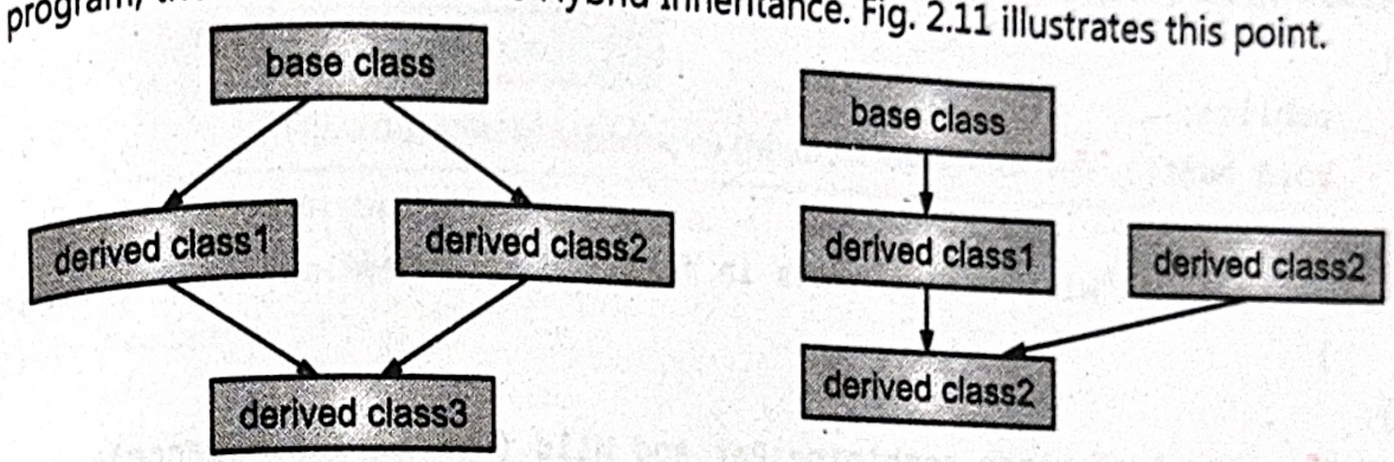

Fig. 2.11: Hybrid Inheritance

Program 2.13:

- One base class (Animal)
- Two derived classes (Pet, Wild) from Animal
- A final class (Cat) that inherits from **both** Pet and Wild

```
#include <iostream>
using namespace std;
// Base class
class Animal
{
    public:
    void eat()
    {
        cout << "Animal eats food." << endl;
    }
};
// Derived from Animal using virtual inheritance
class Pet : virtual public Animal
{
    public:
```

```
    void play()
    {
        cout << "Pet plays with owner." << endl;
    }
};
// Another class derived from Animal using virtual inheritance
class Wild : virtual public Animal
{
    public:
    void hunt()
    {
        cout << "Wild animal hunts in forest." << endl;
    }
};
// Final derived class combining Pet and Wild (Hybrid Inheritance)
class Cat : public Pet, public Wild
{
    public:
    void meow()
    {
        cout << "Cat meows." << endl;
    }
};
int main()
{
    Cat myCat;
    myCat.eat();   // From Animal (resolved by virtual inheritance)
    myCat.play();  // From Pet
    myCat.hunt();  // From Wild
    myCat.meow();  // Cat's own method
    return 0;
}
```

Output:

```
Animal eats food.
Pet plays with owner.
Wild animal hunts in forest.
Cat meows.
```


Example 4: Consider an example where we have three classes' student, test and result. But some authority wants that some weightage should be given for sports to finalize the result. This is shown in Fig. 2.12.

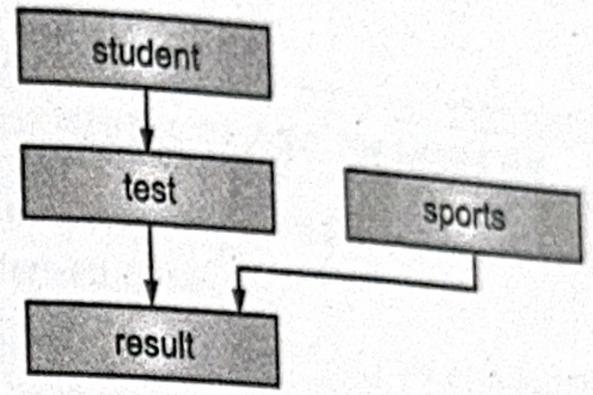

Fig. 2.12: Hybrid Inheritance (Multiple and Multilevel)

Program 2.14: Program for hybrid inheritance.

```
#include<iostream.h>
class student
{
    protected:
        int rollno;
    public:
        void get (int x)
        { rollno = x; }
        void display()
        { cout<<rollno; }
};
class test: public student
{
    protected:
        int mark1, mark2;
    public:
        void getmark(int a, int b)
        {
            mark1 = a;
            mark2 = b;
        }
        void dispmark()
        { cout<<mark1<<"  "<<mark2; }
};
```

```
class sport
{
    protected:
        int smark;
    public:
        void getsm(int s)
        { smark = s; }
        void disps()
        { cout<<smark<<"\n"; }
};
class result: public test, public sport
{
    private:
        int total;
    public:
        void displaytotal()
        {
            total = mark1 + mark2 + smark;
            display();
            dispmark();
            disps();
            cout<<"total = "<<total;
        }
};
void main()
{
    result r;
    r.get(10);
    r.getmark(50, 60);
    r.getsm(70);
    r.displaytotal();
}
```


Output:

```
10    50    60    70
total = 180
```

2.3 BASE CLASS AND DERIVED CLASS EXAMPLES [S-19, W-23]

- In C++, a class that is inherited is referred to as a base class. The class that does the inheriting is called the derived class.
- A derived class can be defined by specifying its relationship with the base class. The pictorial representation of single inheritance is shown in Fig. 2.13.

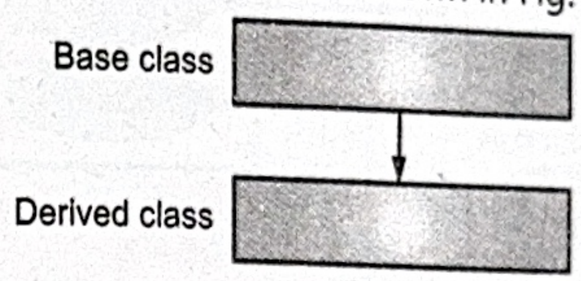

Fig. 2.13: Base Class and Derived Class

- Some example of inheritance is shown in Table 2.1.

Table 2.1: Examples of inheritance

Base Class	Derived Class
Shape	Circle Triangle Rectangle
Employee	Teacher Clerk
Media	Book Tape
Student	Undergraduate Postgraduate

- When a class inherits another, the member of the base class becomes members of the derived class.
- The **syntax** of derived class definition is as follows:

```
class <derived_class_name>: <access modifier> <base_class_name>
{
    // body of class
};
```

It contains:

- Keyword class.
- Derived class name which is user define name for new class.
- The : (colon) indicates that the derived_class_name is derived from the base_class_name.
- The access modifier is optional. If no access modifier is present, the access modifier is private by default. If written, then it must be public, private or protected.
- base_class_name is the existing class name.

For example,

```
class base
{
    ..............
    ..............
};
class derived1: private base          //private derivation
{
    ..............
    ..............
};
class derived2: public base           //public derivation
{
    ..............
    ..............
};
class derived3: protected base        //protected derivation
{
    ..............
    ..............
};
class derived: base                   //by default private derivation
{
    ..............
    ..............
};
```


2.3.1 Public Derivation

- When the access modifier for a base class is **public**, all public members of the base class become public members of the derived class and all protected members of the base class becomes protected members of the derived class.
- The members of derived class can access members of the base class if the base class members are public or protected. They can not access private members.
- The Fig. 2.14 shows the base and derived class relationship.

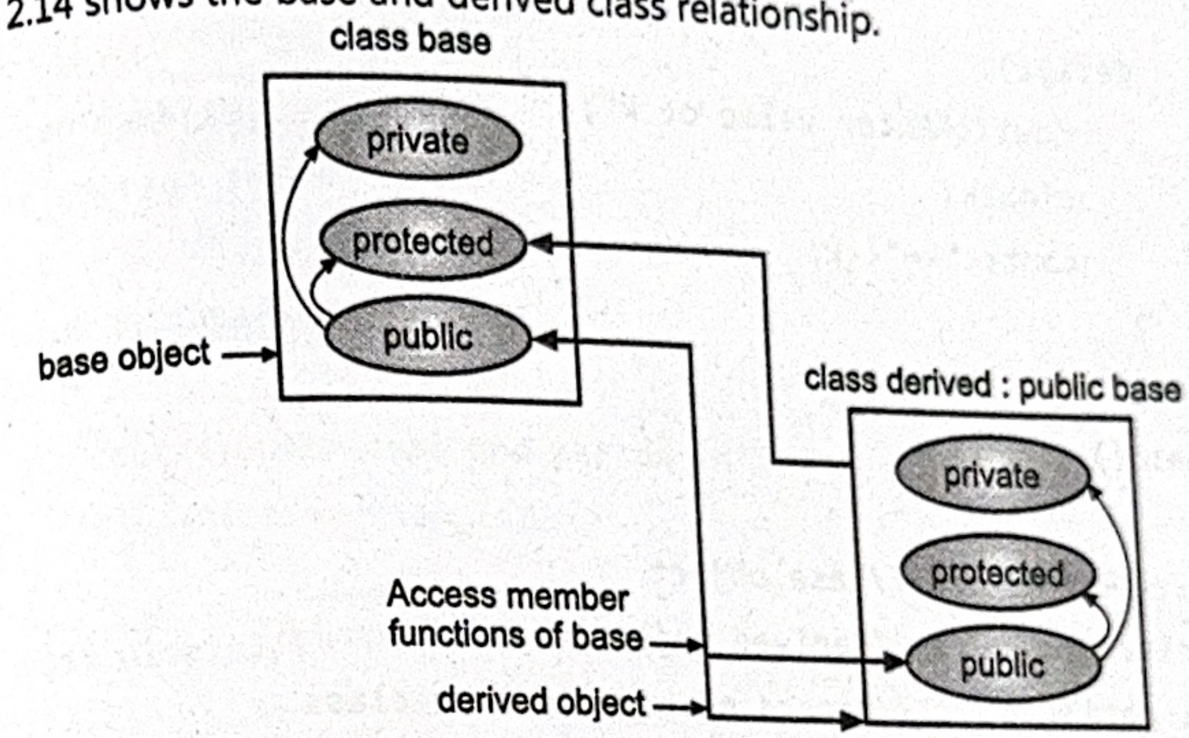

Fig. 2.14: Public Derivation

Program 2.15: Program to demonstrate public derivation.

```
#include<iostream>
using namespace std;
class base
{
   int i, j;
   public:
      void getij()
      {
         cout<<"Enter value of i and j";
         cin>>i;
         cin>>j;
      }
      void show()
      {
         cout<<"i="<<i<<"j="<<j<<"\n";
      }
};
```

```
class derived: public base
{
    int k;
    public:
        void getdata()
        {
            getij();
                cout<<"Enter value of k";
                cin>>k;
                cout<<"k="<<k;
            }
    };
    int main()
    {
        base ob1;        //base object
        derived ob2;     //derived object
        ob1.getij();     //access members of base class
        ob1.show();      //display values of i and j
        ob2.getdata();   //access members of derived class
        ob2.show();      //access members of base class
    }
```

Output:

```
Enter value of i and j 2 3
i=2 j=3
Enter value of i and j 5 6
Enter value of k 7
k=7 i=5 j=6
```

- *In this program, the base class members are directly access through derived class, because the derived through derived class and derived class is publically derived from base class.*
- *In inheritance some of the base class data elements and member functions are inherited into the derived class which extends the capability of the existing classes and useful in incremental program development. If the function names of the base and derived class are same then use scope resolution operation (::) and show the scope of the function i.e. to which the class that function belongs.*

Program 2.16: Write a program to read data members of a base class i.e. name, person id and the derived class members are height and weight. Display the data of person.

```
#include<iostream>
using namespace std;
class base
{
    private:
        char name[20];
        int person_id;
    public:
        void getdata()
        {
            cout<<"Enter name and person_id";
            cin>>name>>person_id;
        }
        void display()
        {
            cout<<"name:"<<name<<"  "<<"person_id:"<<person_id;
        }
};
class derived: public base
{
    private:
            int height, weight;
    public:
        void getdata()
        {
            base:: getdata();
            cout<<"Enter height and weight";
            cin>>height>>weight;
        }
        void display()
        {
            base::display();
            cout<<"height:"<<height<<" "<<"weight:"<<weight;
        }
};
```

```
int main()
{
    derived obj;
    obj.getdata();
    obj.display();
}
```

Output:

```
Enter name and person_id prajakta 11
Enter height and weight 6 80
name: prajakta  person_id:11 height:6  weight:80
```

2.3.2 Private Derivation [S-23]

- When the base class is inherited by using the private access modifier, all public and protected members of the base class become private members of the derived class.
- The Fig. 2.15 shows the private derivation of inheritance.

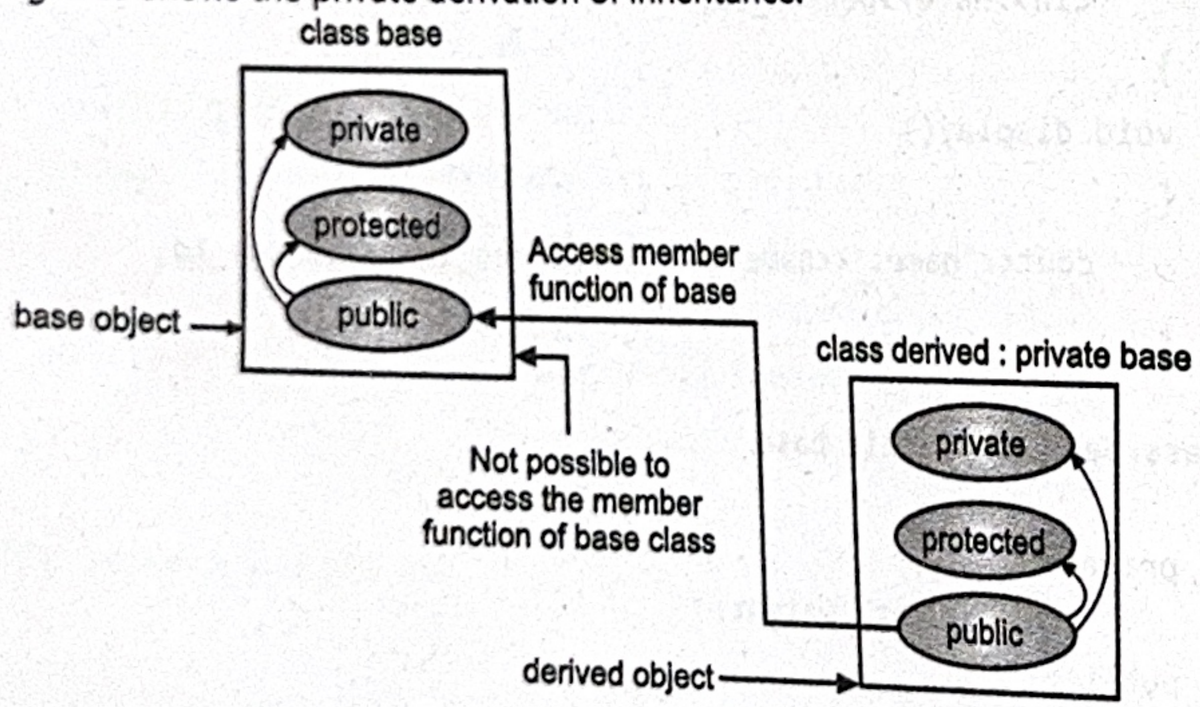

Fig. 2.15: Private Derivation

Program 2.17: Program for private derivation.

```
#include <iostream>
using namespace std;
class Engine
{
    public:
        Engine(int nc)
        {
            cylinder = nc;
        }
```



```
        void start()
        {
            cout << getCylinder() <<" cylinder engine started" << endl;
        }
        int getCylinder()
        {
            return cylinder;
        }
    private:
        int cylinder;
};
class Car: private Engine
{
    public:
        Car(int nc = 4): Engine(nc) { }
        void start()
        {
            cout << "car with " << Engine::getCylinder() <<
            "cylinder engine started" << endl;
            Engine:: start();
        }
};
int main( )
{
    Car c(8);
    c.start();
    return 0;
}
```

Output:

```
car with 8 cylinder engine started
8 cylinder engine started
```

- In this program, derived class object cannot access the functions of base class because they become private. But the public functions of base class access by public functions of derived class.

2.3.3 Protected Members

- The protected keyword provides greater flexibility in the inheritance mechanism. When a member of a class is declared as protected, that member is not accessible by other, non-member elements of program.
- Protected members are as good as private members within the class. Protected member differs from private members when a protected member is inherited. Using the access modifier protected in base class, we could make available the data member in the derived class and restrict its access in other classes or in main().

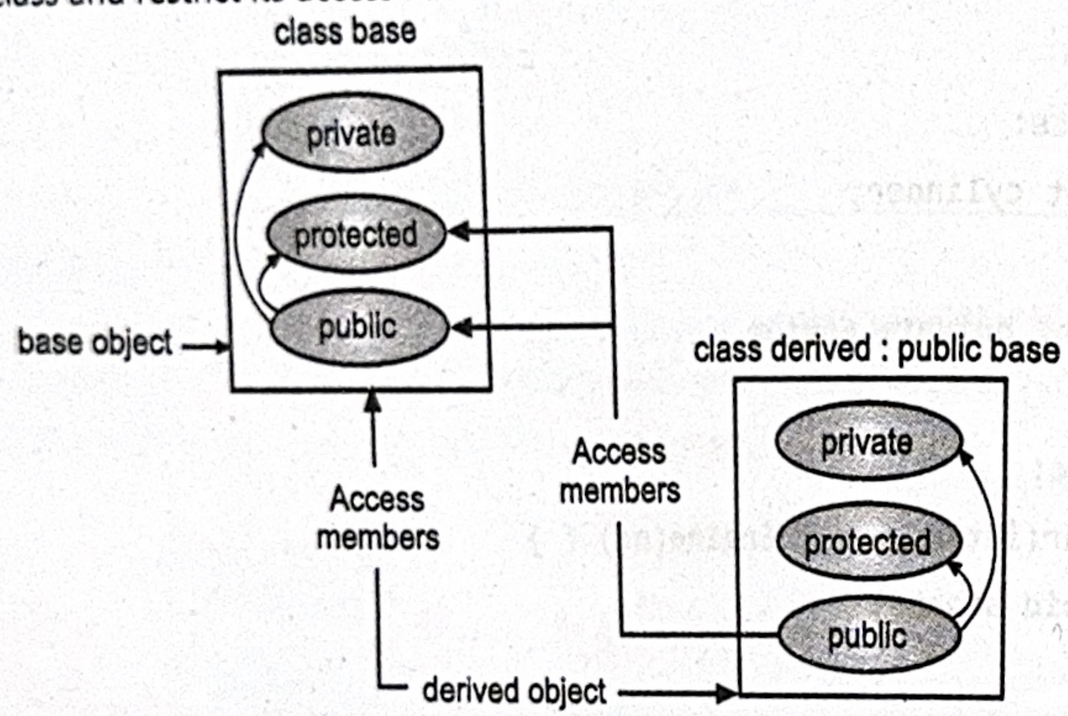

Fig. 2.16: Protected members of base class can be access in derived class

- Data member declare as protected within a base class is accessible by the member within its class and any class immediately derived from it. The Fig. 2.16 shows the public derivation with protected data members.
- If the base class is inherited as public, then the base class protected member becomes protected member of derived class and therefore, can be accessed by derived class. Thus, by using **protected**, we can create class members that are private to their class but can still be inherited and accessed by a derived class.

Program 2.18: Program using protected members.

```
#include<iostream>
using namespace std;
class base
{
   protected:
      int i, j;     // private to base, but accessible by derived class
   public:
      void getij()
      {
         cout<<"Enter value of i and j";
         cin>>i>>j;
      }
```



```
      void show()
      {
         cout<<"i="<<i<<" "<<"j="<<j<<"\n";
      }
};
class derived1: public base
{
   int k;
   public:
      void getdata()
      {
         getij();
            k=i*j;
      }
      void showall()
      {
         cout<<"k="<<k;
      }
}; //end of class derived1
class derived2: public derived1
{
   int x;
   public:
      void get()
      {
         getij();
         x=i*j;
      } // can access i and j since public inheritance
      void showx()
      {
         cout<<"x="<<x;

      }
}; //end of class derived2
```

```
int main()
{
    derived1 ob1;
    derived2 ob2;
    ob1.getdata();
    ob1.showall();
    ob2.get();
    ob2.showx();
}
```

Output:

```
Enter value of i and j 10 20
k=200
Enter value of i and j 15 10
x=150
```

- In the above program, if derived1 class inherits base class privately, then protected member of base becomes private members of derived1 class which cannot inherited in derived2 class. So, when a protected member is inherited in public mode, it becomes protected in the derived class and therefore it is accessible by the member function of the derived class. It is also inherited further.
- A protected member inherited in the private mode becomes private in the derived class and cannot be inherited further.

2.3.4 Protected Derivation

- C++ allows us to use protected as access modifier. When access modifier is protected, all public and protected members of the base class become protected members of the derived class.

Program 2.19: Program for protected derivation.

```
#include<iostream>
using namespace std;
class base
{
    protected:
        int i, j;    //private to base but accessible in derived
    public:
        void getij()
        {
            cin>>i>>j;
        }
```



```
        void show()
        {
            cout<<i<<" "<<j<<"\n";
        }
};
class derived: protected base
{
    int k;
    public:
        void getdata()
        {
            getij();
            k = i*j;   // derived may access base protected data members
        }
        void showall()
        {
            cout<<k<<"  ";
            show();
        }
};
int main()
{
    derived ob;
    //ob.getij();     //illegal, getij() is protected member of derived
    ob.getdata();     // correct as public member of derived
    ob.showall();     // correct as public member of derived
    //ob.show();      // illegal
}
```

Output:

```
2  3
6  2  3
```

- In this program, getij() and show() are public members of base, but they become protected members of derived. Using derived object is not accessible.

Summary about Access Specifiers:

Table 2.2: Access in a Derived Class

Base class members	Derived Class		
	Public derivation	Private derivation	Protected derivation
Private	not accessible	not accessible	not accessible
Protected	protected	private	protected
Public	public	private	protected

Name	Constructor has the same name as class name.	Destructor also has the same name as class name but with a tilde (~) prefixed to its name.
Use	Constructor is used for initializing the values to the data members of the class.	Destructor is used when the object is destroyed or goes out of scope.
In numbers	There can be multiple constructors in the class.	There is always a single destructor in the class.
Overloading	Constructors can be overloaded.	Destructor cannot be overloaded.
Return type	Constructors never have a return type even void.	A destructor neither accepts any parameter nor has a return type (not even void).
Types	In C++, program we can use different types of constructors like default constructor, parameterized constructor, copy constructor etc.	In C++, program a single destructor is used to destroy all the objects of a class created either using default, parameterized or copy constructor.

Example:	`class Student`
	`{`
	`private:`
	`int marks;`
	`char grade;`
	`public:`
	`Student(int m, char g)`
	`{`
	`marks= m;`
	`grade= g;`
	`}`
	`void show()`
	`{`
	`cout<<"Marks ="<<marks<<endl;`
	`cout<<"Grade = "<<grade<<endl;`
	`}`
	`};`
		`class test`
		`{`
		`private:`
		`int a;`
		`public:`
		`test()`
		`{`
		`cout<<"object is created :"<<endl<<endl;`
		`}`
		`~test()`
		`{`
		`cout<<"Object is destroyed"<<endl<<endl;`
		`}`
		`};`

2.6 ABSTRACT CLASS

[S-23, W-22]

- A class that contains at least one pure virtual function is said to be *abstract*. A pure virtual function is a member function i.e., declared in an abstract class, but *defined in a derived class*.
- An abstract class does not create any object and it contains one or more *functions for* which there is no definition (i.e., a pure virtual function).
- We can create pointers and references to an abstract class. This allows abstract classes to support runtime polymorphism, which depends upon base class pointers and references to select the proper virtual function.
- An abstract class is used only to derive other classes. For example: Shape is abstract base class which derives square, circle and rectangle.

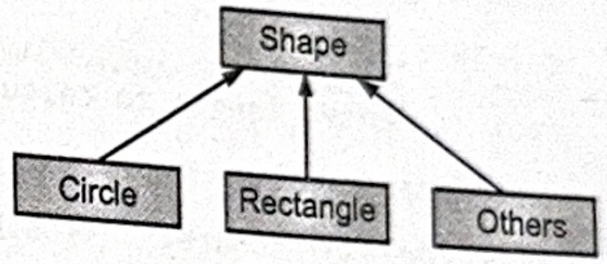

Fig. 2.22

Program 2.34: Following program shows use of abstract classes.

```
#include<iostream>
using namespace std;
// Abstract class
class Shape
{
    protected:
        float l;
    public:
        void getData()
        {
            cin >> l;
        }
        // virtual Function
        virtual float calculateArea() = 0;
};
class Square: public Shape
{
    public:
        float calculateArea()
        {   return l*l;   }
};
```

```
class Circle: public Shape
{   public:
        float calculateArea()
        { return 3.14*l*l; }
};
int main()
{
    Square s;
    Circle c;
    cout << "Enter length to calculate the area of a square: ";
    s.getData();
    cout<<"Area of square: " << s.calculateArea();
    cout<<"\nEnter radius to calculate the area of a circle: ";
    c.getData();
    cout << "Area of circle: " << c.calculateArea();
    return 0;
}
```

Output:

```
Enter length to calculate the area of a square: 4
Area of square: 16
Enter radius to calculate the area of a circle: 5
Area of circle: 78.5
```

2.7 POLYMORPHISM [S-19, W-23]

- Polymorphism means "one name, many forms". Polymorphism is the process of defining a number of objects at different classes into group and call the method to carry out the operation of the objects using different function calls.
- There are two types of polymorphism:

1. Compile time polymorphism: [S-18]

- This is also called as **early** or **static** binding. Selection of an appropriate function for a particular call at the compile time itself. For example, function overloading and operator overloading. Function overloading is also referred to as functional polymorphism. The same function can perform a wide variety of tasks.

- The same function can handle different data types. When many functions with the same name but different argument lists are defined, then the function to be invoked corresponding to a function call is known during compile time.
- When the source code is compiled, the functions to be invoked are bound to the compiler during compile time, as to invoke which function depending upon the type and number of arguments. Such a phenomenon is referred to early binding, static linking or compile time polymorphism.

2. Run-time polymorphism:

- This is also called as **dynamic binding** or **late binding**. Sometimes, a situation occurs where function name and prototype is same in both the base class and in the derived class. Compiler does not know what to do, which function to call. In this class appropriate member function is selected to run time. For example, virtual function.

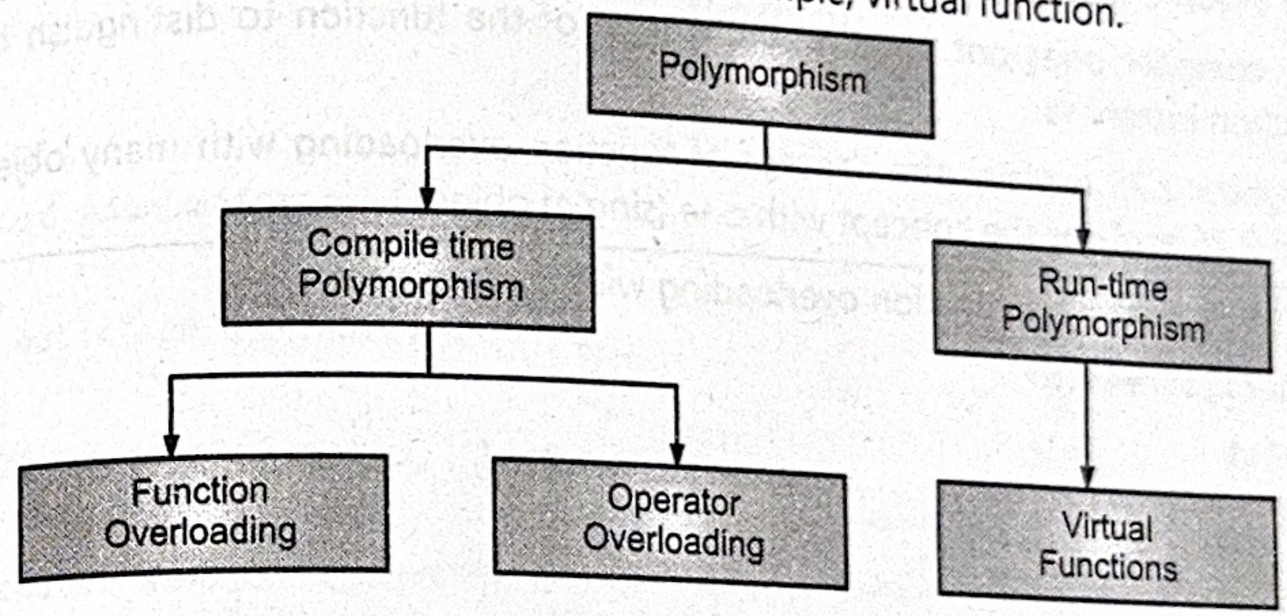

Fig. 2.23: Types of Polymorphism

2.8 FUNCTION OVERLOADING [W-18, 22; S-18, 22]

- C++ provides the facility of function overloading in which we can define multiple functions with the same name but the types of argument are different.
- For example, we can have a function add with two floating point numbers as argument, another function adds with 3 integers, one add function with 4 doubles and so on. But function name is similar. This is called as **Function Overloading**.
- An overloaded function is a function with the same name as another function, but with different parameter types.
- The function overloading can be performed using single and many objects. This is shown in Fig. 2.24.