OBJECT ORIENTED PROGRAMMING (PC-CS-203A)

UNIT-II

Prepared By:
Er. Tanu Sharma
(A.P. CSE Deptt., PIET)

Syllabus (Unit-2)

Friend Function and Friend Classes, This Pointer, Dynamic Memory Allocation and De-allocation (New and Delete), Static Class Members, Constructors, parameter Constructors and Copy Constructors, Deconstructors

Introduction of inheritance, Types of Inheritance, Overriding Base Class Members in a Derived Class, Public, Protected and Private Inheritance, Effect of Constructors and Deconstructors of Base Class in Derived Classes.

FRIEND FUNCTIONS

- If two or more classes are required to share a particular function, C++ allows a common function to be made friendly with both the classes.

- This function can access the private data of these classes.

- This function need not be a member of these classes.

- This function can be declared as a **friend** of the class.

- Function declaration should be preceded by keyword **friend**.

- Friend function can be defined anywhere in the program like a normal C++ function.
- Function definition does not use keyword friend or scope resolution operator ::.
- A function can be declared as friend in any number of classes.
- A friend function can access the private members of the class.

Characteristics of friend function

- It is not in the scope of the class to which it has been declared as a friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.

Characteristics of friend function

- It cannot access the member names directly and has to use an object name and dot membership operator with each member name.
- It can be declared either in private or public part of the class without affecting its meaning.
- Usually, it has objects as arguments.

Example program

```
class sample
{
        int a;
        int b;
   public:
        void setvalue() {a=25; b=40; }
        friend float mean(sample s);
};
float mean(sample s)
{
        return float(s.a + s.b)/2.0;
}

int main()
{
        sample X;       // object X
        X.setvalue();
        cout << "Mean value = " << mean(X) << "\n";

        return 0;
}
```

```
Mean value = 32.5
```

Friend Class

- Members function of one class can be friend function of another class.
- Then they are defined using scope resolution operator.
- In the example below:

    fun1() is a member of **class X** and **friend** of **class Y.**

```
class X
{
        .....
        .....
        int fun1();
        .....
};
```

```
class Y
{
        .....
        .....
        friend int X :: fun1();
        .....
};
```

- All the member functions of one class can be declared as friend functions of another class.
- The class is thus called a **friend class.**

```
class Z
{
      .....
      friend class X;    // all member functions of X are
                         // friends to Z
};
```

Example Program

A FUNCTION FRIENDLY TO TWO CLASSES

```
#include <iostream>

using namespace std;

class ABC;      // Forward declaration
//----------------------------------------------------------//
class XYZ
{
        int x;
  public:
        void setvalue(int i) {x = i;}
        friend void max(XYZ, ABC);
};
//----------------------------------------------------------//
class ABC
{
        int a;
  public:
        void setvalue(int i) {a = i;}
        friend void max(XYZ, ABC);
};
```

```
//------------------------------------------------------------//
void max(XYZ m, ABC n)        // Definition of friend
{
        if(m.x >= n.a)
                cout << m.x;
        else
                cout << n.a;
}
//------------------------------------------------------------//
int main()
{
        ABC abc;
        abc.setvalue(10);
        XYZ xyz;
        xyz.setvalue(20);
        max(xyz, abc);

        return 0;
}
```

This pointer

- A unique keyword that is used to represent an object that invokes a member function.
- It points to the object for which the function was called.
- Automatically passed to a member function when it is called.
- Acts as an implicit argument to all the member functions.
- E.g.

  return *this;

  - The above statement inside a member function definition, will return the object that invoked the function.
  - Returns the invoking object as a result.

```
person & person :: greater(person & x)
{
    if x.age > age
        return x;                    // argument object
    else
        return *this;               // invoking object
}
```

- Invoking the function by the call:

  max = a.greater(b);

  – The function will return object **b** if **age of person b is greater than of a**

  – Else it will return object a using this pointer.

this POINTER

```
#include <iostream>
#include <cstring>

using namespace std;

class person
{
        char name[20];
        float age;
  public:
        person(char *s, float a)
        {
        strcpy(name, s);
        age = a;
}
person & person :: greater(person & x)

{
        if(x.age >= age)
                return x;
        else
                return *this;
}
```

```
        void display(void)
        {
                cout << "Name: " << name << "\n"
                     << "Age:  " << age << "\n";
        }
};

int main()
{
        person P1("John", 37.50),
               P2("Ahmed", 29.0),
               P3("Hebber", 40.25);

        person P = P1.greater(P3);          // P3.greater(P1)
        cout << "Elder person is: \n";
        P.display();

        P = P1.greater(P2);                 // P2.greater(P1)
        cout << "Elder person is: \n";
        P.display();

        return 0;
}
```

The output of Program

```
Elder person is:
Name: Hebber
Age:  40.25
Elder person is:
Name: John
Age:  37.5
```

Swapping private data of classes

```
#include <iostream>

using namespace std;

class class_2;

class class_1
{
        int value1;
  public:
        void indata(int a) {value1 = a;}
        void display(void) {cout << value1 << "\n";}
        friend void exchange(class_1 &, class_2 &);
};

class class_2
{
        int value2;
  public:
        void indata(int a) {value2 = a;}
        void display(void) {cout << value2 << "\n";}
        friend void exchange(class_1 &, class_2 &);
};
```

```
void exchange(class_1 & x, class_2 & y)
{
        int temp = x.value1;
        x.value1 = y.value2;
        y.value2 = temp;
}

int main()
{
        class_1 C1;
        class_2 C2;

        C1.indata(100);
        C2.indata(200);

        cout << "Values before exchange" << "\n";
        C1.display();
        C2.display();

        exchange(C1, C2);          // swapping

        cout << "Values after exchange " << "\n";
        C1.display();
        C2.display();

        return 0;
}
```

Output

```
Values before exchange
100
200
Values after exchange
200
100
```

Returning Objects

```
#include <iostream>

using namespace std;

class complex                    // x + iy form
{
        float x;                         // real part
        float y;                         // imaginary part
    public:
        void input(float real, float imag)
        { x = real; y = imag; }
        friend complex sum(complex, complex);

        void show(complex);
};

complex sum(complex c1, complex c2)
{
        complex c3;              // objects c3 is created
        c3.x = c1.x + c2.x;
        c3.y = c1.y + c2.y;
        return(c3);              // returns object c3
}
```

```
void complex :: show(complex c)
{
     cout << c.x << " + j" << c.y << "\n";
}

int main()
{
     complex A, B, C;

     A.input(3.1, 5.65);
     B.input(2.75, 1.2);


     C = sum(A, B);        // C = A + B

     cout << "A = "; A.show(A);
     cout << "B = "; B.show(B);
     cout << "C = "; C.show(C);

     return 0;
}
```

Output

```
A = 3.1 + j5.65
B = 2.75 + j1.2
C = 5.85 + j6.85
```

const Member Functions

- If a member function does not alter any data in the class, it can be declared as a **const** member function.
  - Error message will be generated if **const** member function try to alter any data value.
- **const** is appended to function prototypes in declaration as well as definition.

  void mul (int, int) const;

  double get_balance () const;

Constructors

- A constructor is a special member function which initializes the object of its class when created.
- This is also known as **automatic initialization** of objects.
- The name of constructor is same as the name of the class.
- The constructor is invoked whenever object of its associated class is created.

Declaration and Definition of Constructor

```
// class with a constructor

class integer
{
        int m, n;
    public:
        integer(void);              // constructor declared
        .....
        .....
};
integer :: integer(void)            // constructor defined
{
      m = 0; n = 0;
}
```

Characteristics of Constructors

- Should be declared in public section.
- Invoked automatically when objects are created.
- Do not have return types, hence cannot return any value.
- Cannot be inherited, but derived class can call the base class constructor.
- Can have default arguments.
- Cannot be virtual.

Types of Constructors

- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Dynamic Constructor.

Default Constructor

- A constructor with no arguments is called default constructor.
- If default constructor is not defined, compiler automatically supplies default constructor.

```
A::A(){}                //default constructor
A::A(){m=0; n=o;}       //default constructor
```

Parameterized Constructors

- Constructors that can take arguments are called parameterized constructors.
- To call a parameterized constructor, arguments are passed to the constructor function when an object is declared.
- This can be done in two ways:
  - Calling constructors explicitly.
  - Calling constructors implicitly.

```
class integer
{
        int m, n;
   public:
        integer(int x, int y);   // parameterized constructor
        .....
        .....
};
integer :: integer(int x, int y)
{
       m = x; n = y;
}
```

Calling Parameterized constructor

- Explicit call

  integer int1=integer (0,100);
- Implicit call

  integer int1 (0,100);
- Parameters of a constructor can be of any type except that of the class to which it belongs.
- Constructor can accept a reference to its own class as a parameter.
- Such a constructor is called **copy constructor**.

```
Class A
{
      .....
      .....
   public:
      A(A&);
};
```

```
class integer
{
        int m, n;
  public:
        integer(int, int);                    // constructor declared

        void display(void)
        {
             cout << " m = " << m  << "\n";
             cout << " n = " << n  << "\n";
        }
};

integer :: integer(int x, int y)      // constructor defined
{
      m = x;  n = y;
}

int main()
{
    integer int1(0,100);                 // constructor called implicitly

    integer int2 = integer(25, 75);  // constructor called explicitly

    cout << "\nOBJECT1" << "\n";
    int1.display();

    cout << "\nOBJECT2" << "\n";
    int2.display();

    return 0;
}
```

Copy Constructors

- It is a special constructor for creating a new object as a copy of an existing object.
- A copy constructor takes a reference to an object of the same class as itself as an argument.
- The process of initializing through a copy constructor is known as copy initialization.
- General form:

  constructor-name (classname & objectname)
- Eg.

```
integer (integer & i)
{
m=i.m;
n=i.n;
}
```

- To invoke the copy constructor and copies the value of i1 into i2, following statements can be used:

```
integer  i2(i1);
integer i2=i1;
```

Multiple Constructors

```
class integer
{
        int m, n;
    public:
        integer(){m=0; n=0;}                    // constructor 1
        integer(int a, int b)
        {m = a; n = b;}                         // constructor 2
        integer(integer & i)
        {m = i.m;  n = i.n;}                    // constructor 3
};
```

- These constructors can be invoked by the following statements:

  integer i1;

  integer i2(20,40);

  integer i3(i2);

Overloaded Constructors

- The process of sharing the same name by two or more functions is called function overloading.
- Similarly, when more than one constructor is defined in a class, we can say that the constructor is overloaded.

```
#include <iostream>

using namespace std;

class code
{
      int id;
   public:
      code(){ }                              // constructor
      code(int a) { id = a;}                 // constructor again
      code(code & x)                         // copy constructor

      {
            id = x.id;                       // copy in the value
      }
      void display(void)
      {
            cout << id;
      }
};
```

```
int main()
{
    code A(100);  // object A is created and initialized
    code B(A);    // copy constructor called
    code C = A;   // copy constructor called again

    code D; // D is created, not initialized
    D = A;        // copy constructor not called

    cout << "\n id of A: "; A.display();
    cout << "\n id of B: "; B.display();
    cout << "\n id of C: "; C.display();
    cout << "\n id of D: "; D.display();

    return 0;
}
```

Output

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

Note:

An argument can only be pass by reference to the copy constructor and not by value.

Constructor with default arguments

- Constructor can also be defined with default arguments.
- Eg:
  - complex (float real, float imag=0);
  - Default value of imag=0;

complex c(5.0);

  - Assigns value 5.0 to real and 0.0 to imag (default value).

complex c (2.0,3.0);

  - Assigns value 2.0 to real and 3.0 to imag.

Dynamic Initialization of Objects

- Class objects can also be initialized dynamically.
- The initial value of object may be provided during run time.
- Advantages:
  - Various initialization formats can be provided using overloaded constructors.
  - Provides flexibility of using different formats of data at run time.

```
// Long-term fixed deposit system
#include <iostream>
using namespace std;
class Fixed_deposit
{
        long int P_amount;      // Principal amount
        int      Years;         // Period of investment
        float    Rate;          // Interest rate
        float    R_value;       // Return value of amount
  public:
        Fixed_deposit(){ }
        Fixed_deposit(long int p, int y, float r=0.12);
        Fixed_deposit(long int p, int y, int r);
        void display(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for(int i = 1; i <= y; i++)
            R_value = R_value * (1.0 + r);
}
```

```
Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;

        for(int i=1; i<=y; i++)
            R_value = R_value*(1.0+float(r)/100);
}

void Fixed_deposit :: display(void)
{
        cout << "\n"
            << "Principal Amount = " << P_amount << "\n"
            << "Return Value     = " << R_value  << "\n";
}
```

```
int main()
{
        Fixed_deposit FD1, FD2, FD3;  // deposits created

        long int p;                   // principal amount

  int     y;                    // investment period, years
  float   r;                    // interest rate, decimal form
  int     R;                    // interest rate, percent form

  cout << "Enter amount,period,interest rate(in percent)"<<"\n";
  cin >> p >> y >> R;
  FD1 = Fixed_deposit(p,y,R);

  cout << "Enter amount,period,interest rate(decimal form)" << "\n";
  cin >> p >> y >> r;
  FD2 = Fixed_deposit(p,y,r);

  cout << "Enter amount and period" << "\n";
  cin >> p >> y;
  FD3 = Fixed_deposit(p,y);

  cout << "\nDeposit 1";
  FD1.display();

  cout << "\nDeposit 2";
  FD2.display();

  cout << "\nDeposit 3";
  FD3.display();

  return 0;
}
```

Dynamic Constructor

- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- The memory is allocated with **new** operator.

```
class String
{
        char *name;
        int  length;
  public:
        String()            // constructor-1
        {
                length = 0;
                name = new char[length + 1];
        }

        String(char *s)  // constructor-2
        {
                length = strlen(s);
        name = new char[length + 1];    // one additional
                                        // character for \0
        strcpy(name, s);
        }

        void display(void)
        {cout << name << "\n";}
        void join(String &a, String &b);
};
```

```
void String :: join(String &a, String &b)
{
        length = a.length + b.length;
        delete name;
        name = new char[length+1];              // dynamic allocation

        strcpy(name, a.name);
        strcat(name, b.name);
};

int main()
{
        char *first = "Joseph ";
        String name1(first), name2("Louis "),name3("Lagrange"),s1,s2;

        s1.join(name1, name2);
        s2.join(s1, name3);
        name1.display();
        name2.display();
        name3.display();
        s1.display();
        s2.display();

        return 0;
}
```

The output of Program 6.5 would be:

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange
```

Destructor

- Used to destroy the objects that have been created by a constructor.
- Destructor is a member function whose name is the same as the class name.
- It is preceded by a tilde (~).
- Does not take any argument.
- Does not return any value.
- Invoked implicitly by the compiler upon exit from the program (or block or function).
- When **new** is used to allocate memory in constructor, **delete** should be used to free that memory.

```
matrix :: ~matrix()
{
      for(int i=0; i<d1; i++)
      delete p[i];
      delete p;
}
```

IMPLEMENTATION OF DESTRUCTORS

```
#include <iostream>

using namespace std;

int count = 0;

class alpha
{
  public:
      alpha()
      {
           count++;
           cout << "\nNo.of object created " << count;
      }

      ~alpha()
      {
           cout << "\nNo.of object destroyed " << count;
           count--;
      }
};
```

```
int main()
{
    cout << "\n\nENTER MAIN\n";

    alpha A1, A2, A3, A4;
    {
        cout << "\n\nENTER BLOCK1\n";
        alpha A5;
    }

    {
        cout << "\n\nENTER BLOCK2\n";
        alpha  A6;
    }
    cout << "\n\nRE-ENTER MAIN\n";

    return 0;
}
```

The output of a sample run of Program

```
ENTER MAIN

No.of object created 1
No.of object created 2
No.of object created 3
No.of object created 4

ENTER BLOCK1

No.of object created 5
No.of object destroyed 5

ENTER BLOCK2

No.of object created 5
No.of object destroyed 5

RE-ENTER MAIN

No.of object destroyed 4
No.of object destroyed 3
No.of object destroyed 2
No.of object destroyed 1
```

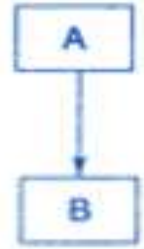
INHERITANCE

Inheritance

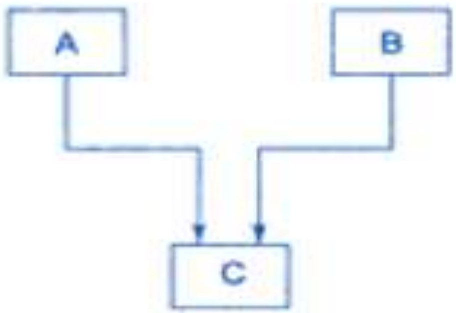
- The process of deriving new class from an old one is called **Inheritance** (or **derivation**).
  - i.e. creating new classes, **reusing** the properties of existing classes.
- The old class is called **base class** and the new class is called **derived class** or **subclass**.
- The derived class inherits some or all of the traits from the base class.

Types of Inheritance

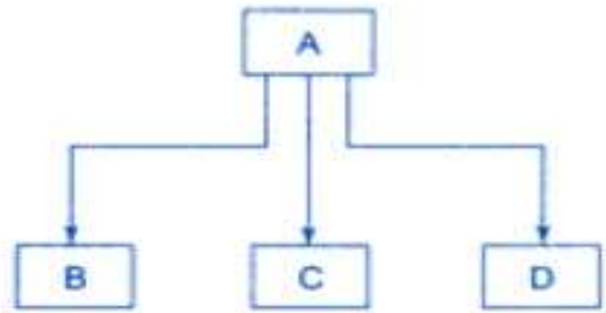
- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance



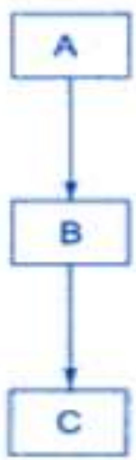
(a) Single inheritance



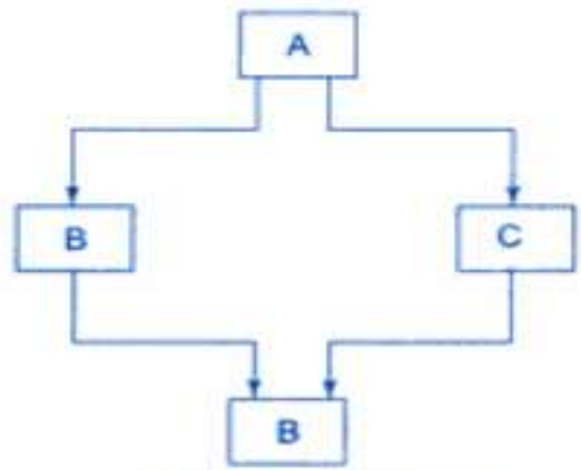
(b) Multiple inheritance



(c) Hierarchical inheritance



(d) Multilevel inheritance



(e) Hybrid inheritance

Defining Derived Classes

```
class derived-class-name  : visibility-mode base-class-name
{
        .....//
        .....//   members of derived class
        .....//
};
```

- When deriving a class, visibility-mode is optional.
- May be either private, protected or public.
- If not specified then it is private by default.

Examples:

```
class ABC: private XYZ          // private derivation
{
        members of ABC
};

class ABC: public XYZ           // public derivation
{
        members of ABC
};

class ABC: XYZ                  // private derivation by default
{
        members of ABC
};
```

Accessing members of base class

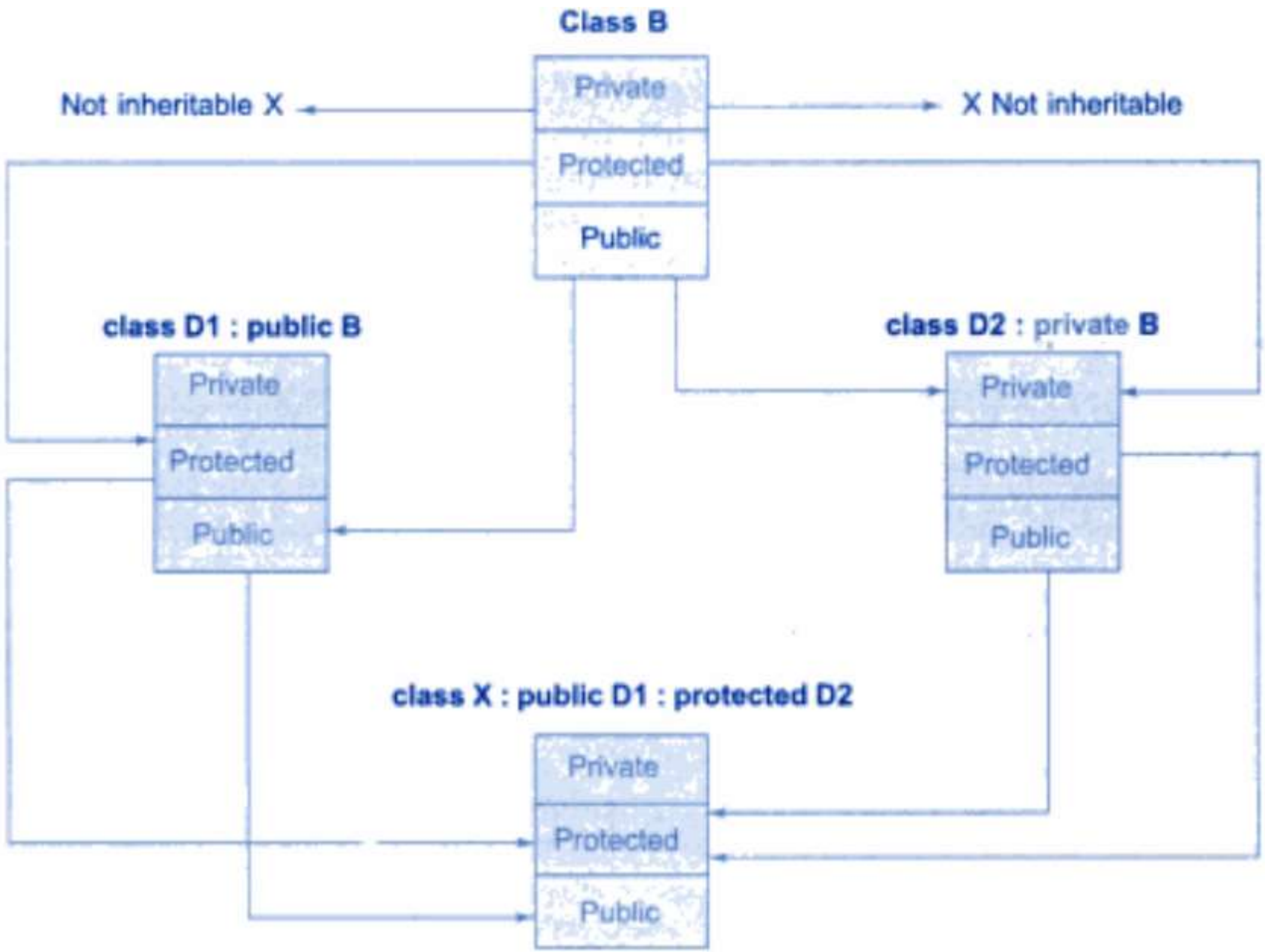
- When a base class is privately inherited by derived class:
  - public members of base class become private members of derived class.
  - Can only be accessed by member functions of derived class.
  - Cannot be accessed by objects of derived class.
- When a base class is publicly inherited:
  - public members of base class become public members of derived class.
  - Can be accessed by objects of derived class.
- Private members cannot be inherited in both the cases.
- Derived class can add its own data members and member functions also.

Making a private member inheritable

- By using a another visibility modifier, protected, we can inherit the private members of the class.
- Protected members are accessible to:
  - the members functions of the class to which it belongs
  - any class immediately derived from it.
  - Cannot be accessed by the function outside these two classes.

```
class alpha
{
  private:                    // optional
      .....                   // visible to member functions
      .....                   // within its class
  protected:
      .....                   // visible to member functions
      .....                   // of its own and derived class
  public:
      .....                   // visible to all functions
      .....                   // in the program
};
```

Effect of Inheritance on visibility of members



Visibility of inherited members

Base class visibility		Derived class visibility		
		Public derivation	***Private derivation***	***Protected derivation***
Private	⟶	Not inherited	Not inherited	Not inherited
Protected	⟶	Protected	Private	Protected
Public	⟶	Public	Private	Protected

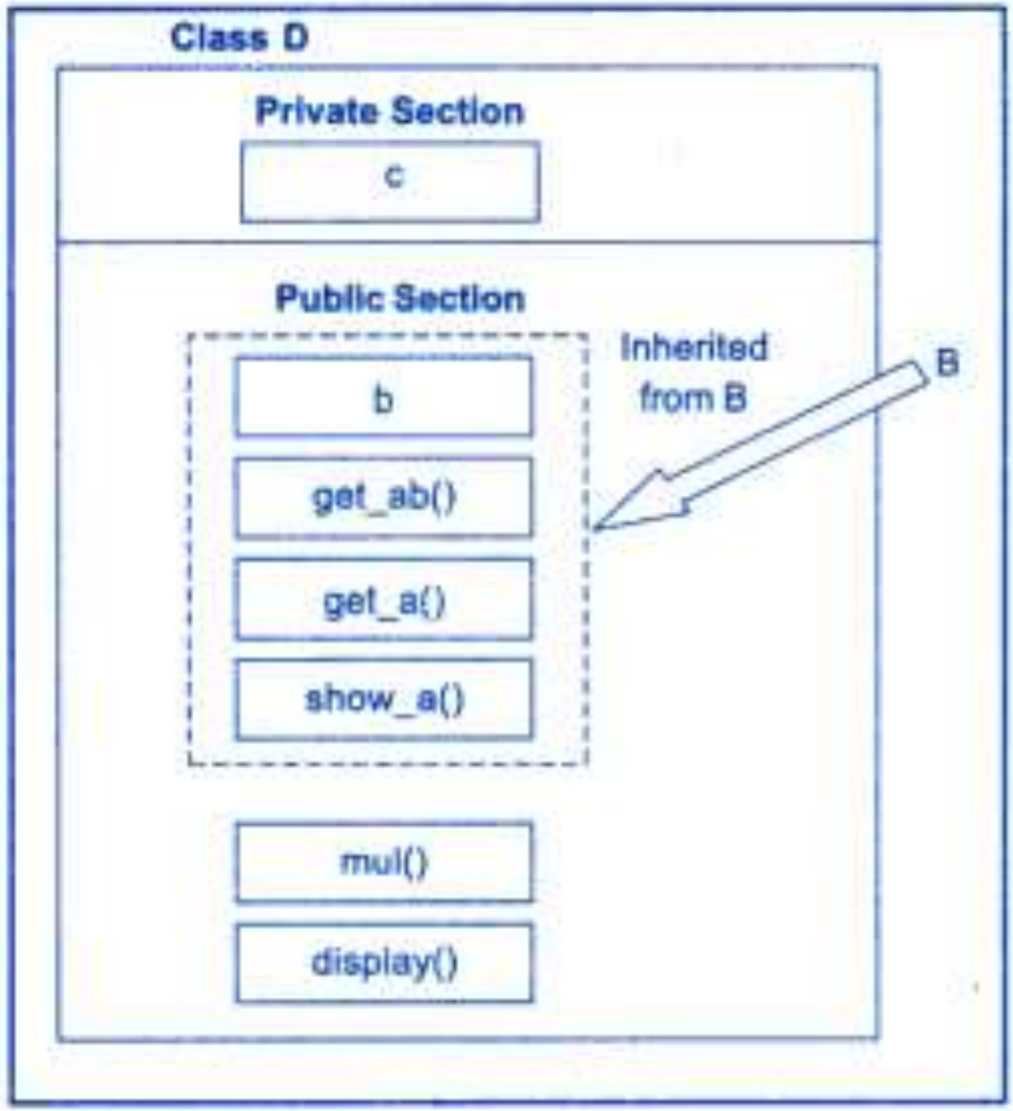
Single Inheritance (public derivation)

```
#include <iostream>

using namespace std;

class B
{
        int a;                      // private; not inheritable
    public:
        int b;                      // public; ready for inheritance
        void get_ab();
        int  get_a(void);
        void show_a(void);
};

class D : public B                  // public derivation
{
         int c;
    public:
         void mul(void);
         void display(void);
};
//------------------------------------------------------------
void B :: get_ab(void)
{
         a = 5; b = 10;
}
int B :: get_a()
{
         return a;
}
```

```
void B :: show_a()
{
        cout << "a = " << a << "\n";
}
void D :: mul()
{
        c = b * get_a();
}
void D :: display()
{
        cout << "a = " << get_a() << "\n";
        cout << "b = " << b << "\n";
        cout << "c = " << c << "\n\n";
}
//-----------------------------------------------
int main()
{
        D d;

        d.get_ab();
        d.mul();
        d.show_a();
        d.display();

        d.b = 20;
        d.mul();
        d.display();

        return 0;
}
```

Given below is the output of Program :

```
a = 5
a = 5
b = 10
c = 50

a = 5
b = 20
c = 100
```



SINGLE INHERITANCE : PRIVATE

```
#include <iostream>

using namespace std;

class B
{
        int a;              // private; not inheritable
      public:
        int b;              // public; ready for inheritance
        void get_ab();
        int  get_a(void);
        void show_a(void);
};

class D : private B                 // private derivation
{
        int c;
    public:
        void mul(void);
        void display(void);
};
```

```
void B :: get_ab(void)
{
      cout << "Enter values for a and b:";
      cin >> a >> b;
}

int B :: get_a()
{
      return a;
}

void B :: show_a()
{
       cout << "a = " << a << "\n";
}

void D :: mul()
{
       get_ab();
       c = b * get_a();          // 'a' cannot be used directly
}

void D :: display()
{
       show_a();                 // outputs value of 'a'
       cout << "b = " << b << "\n"
            << "c = " << c << "\n\n";
}
```

```
int main()
{
     D d;

     // d.get_ab();  WON'T WORK
     d.mul();
     // d.show_a();  WON'T WORK
     d.display();
    // d.b = 20;     WON'T WORK; b has become private
    d.mul();
    d.display();

    return 0;
}
```

The output of Program 8.2 would be:

```
Enter values for a and b:5 10
a = 5
b = 10
c = 50
Enter values for a and b:12 20
a = 12
b = 20
c = 240
```

Multilevel Inheritance

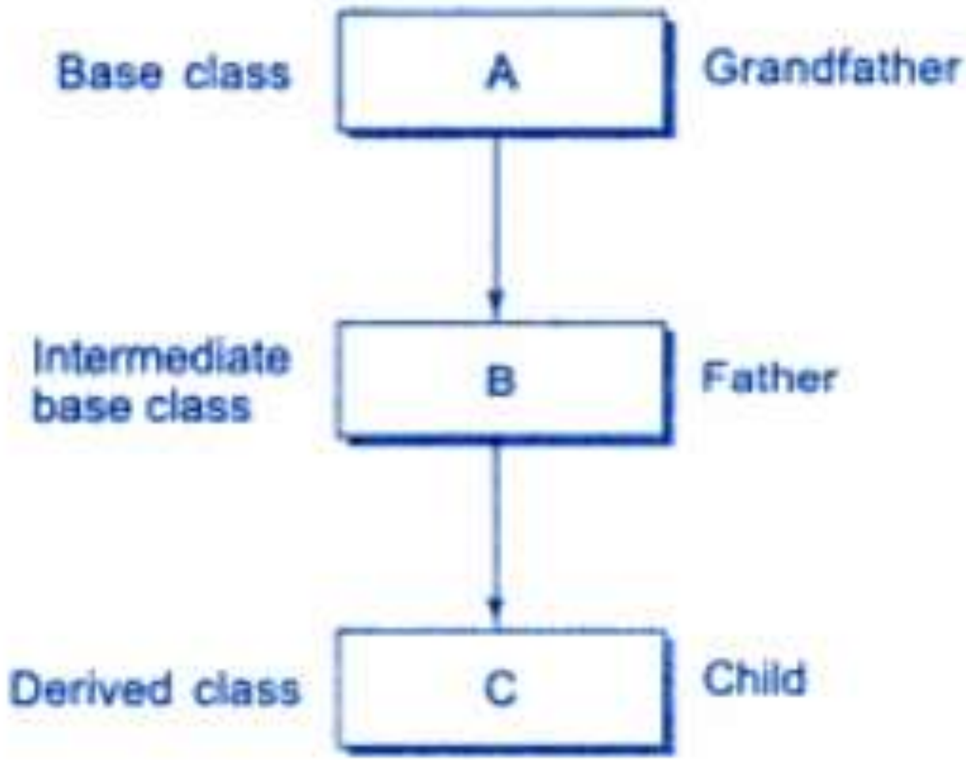


Fig. 8.7 ⇔ *Multilevel inheritance*

- A derived class with multilevel inheritance is derived as follows:

```
class A {……};
class B: public A {……};
class C: public B {……};
```

- This process can be extended to any number of levels.

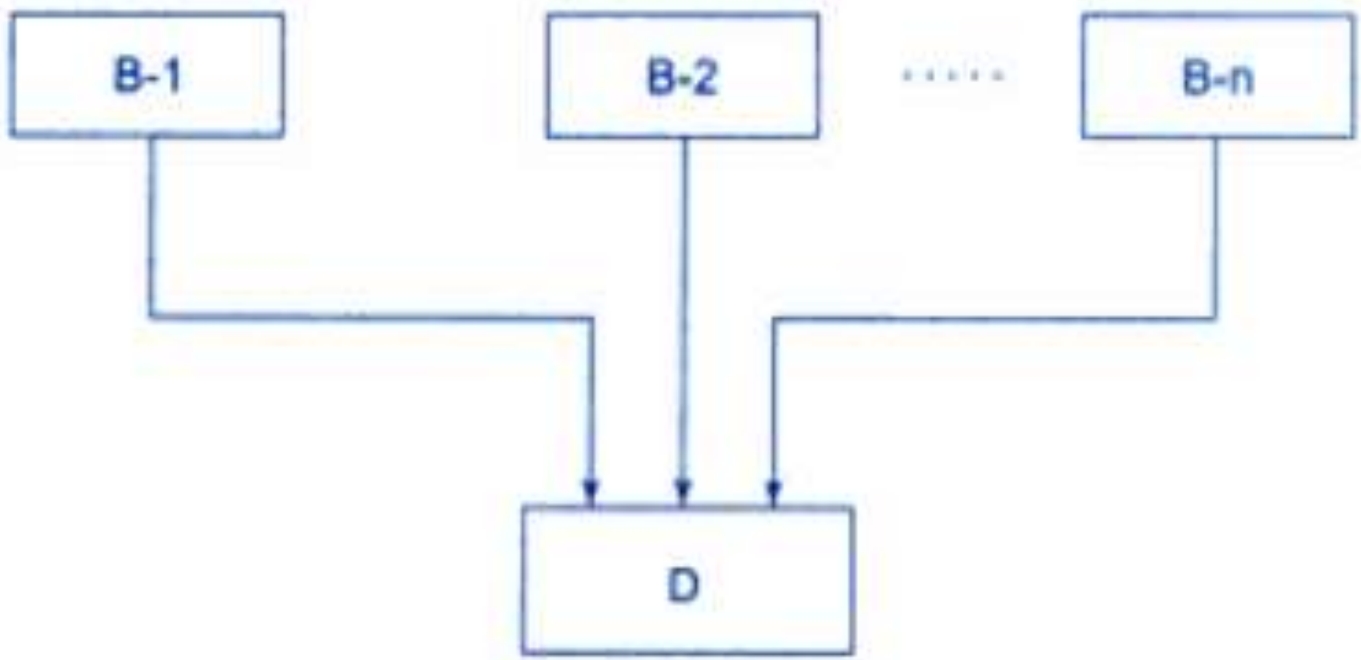
```
class student
{
  protected:
          int roll_number;
  public:
          void get_number(int);
          void put_number(void);
};

void student :: get_number(int a)
{
          roll_number = a;
}

void student :: put_number()
{
          cout << "Roll Number: " << roll_number << "\n";
}

class test : public student                    // First level derivation
{
  protected:
      float sub1;
      float sub2;
  public:
        void get_marks(float, float);
        void put_marks(void);
};

void test :: get_marks(float x, float y)
{
     sub1 = x;
     sub2 = y;
}
```

```
void test :: put_marks()
{
        cout << "Marks in SUB1 = " << sub1 << "\n";
        cout << "Marks in SUB2 = " << sub2 << "\n";
}

class result : public test        // Second level derivation
{
      float total;                // private by default
  public:
      void display(void);
};

void result :: display(void)
{
        total = sub1 + sub2;
        put_number();
        put_marks();
        cout << "Total = " << total << "\n";
}

int main()
{
        result student1;          // student1 created

        student1.get_number(111);
        student1.get_marks(75.0, 59.5);

        student1.display();

     return 0;
}
```

Program 8.3 displays the following output:

```
Roll Number: 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5
```

Multiple Inheritance



Multiple Inheritance

- In multiple inheritance, a class can inherit the attributes of two or more classes.
- It allows to combine the features of several existing classes to define a new class.
- Syntax:

```
class D: visibility B-1, visibility B-2 .....
{
  ........
  ........
  ........
}
```

Program: Multiple Inheritance

```
class M
{
  protected:
          int m;
  public:
          void get_m(int);
};

class N
{
  protected:
      int n;
  public:
      void get_n(int);
};

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

```
void M :: get_m(int x)
{
      m = x;
}

void N :: get_n(int y)
{
      n = y;
}

void P :: display(void)
{
      cout << "m = " << m << "\n";
      cout << "n = " << n << "\n";
      cout << "m*n = " << m*n << "\n";
}

int main()
{
      P p;

      p.get_m(10);
      p.get_n(20);
      p.display();

      return 0;
}
```

The output of Program

```
m = 10
n = 20
m*n = 200
```

Ambiguity resolution in Inheritance

- In case of multiple inheritance, if we try to call a function with object of derived class having same name in more than one base class, compiler shows error.
- This situation is called ambiguous .
- This can be solved using scope resolution operator.

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

class N
{
  public:
      void display(void)
      {
             cout << "Class N\n";
      }
};
class P : public M, public N
{
  public:
      void display(void)        // overrides display() of M and N
      {
            M :: display();
      }
};
```

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

Ambiguity in Single Inheritance

- Ambiguity can also arise in single inheritance if both the base class and derived class are having the same member function.
- The function in derived class always overrides the inherited base class function (if same names).
- Then, derived class function can be invoked using object of derived class.
- Scope resolution operator can be used to invoke base class function in such situation.

```cpp
class A
{
  public:
      void display()
      {
              cout << "A\n";
      }
};
class B : public A
{
  public:
     void display()
     {
              cout << "B\n";
     }
};
int main()
{
     B b;                        // derived class object
     b.display();                // invokes display() in B
     b.A::display();             // invokes display() in A
     b.B::display();             // invokes display() in B

      return 0;
}
```

This will produce the following output:

```
B
A
B
```

Hierarchical Inheritance

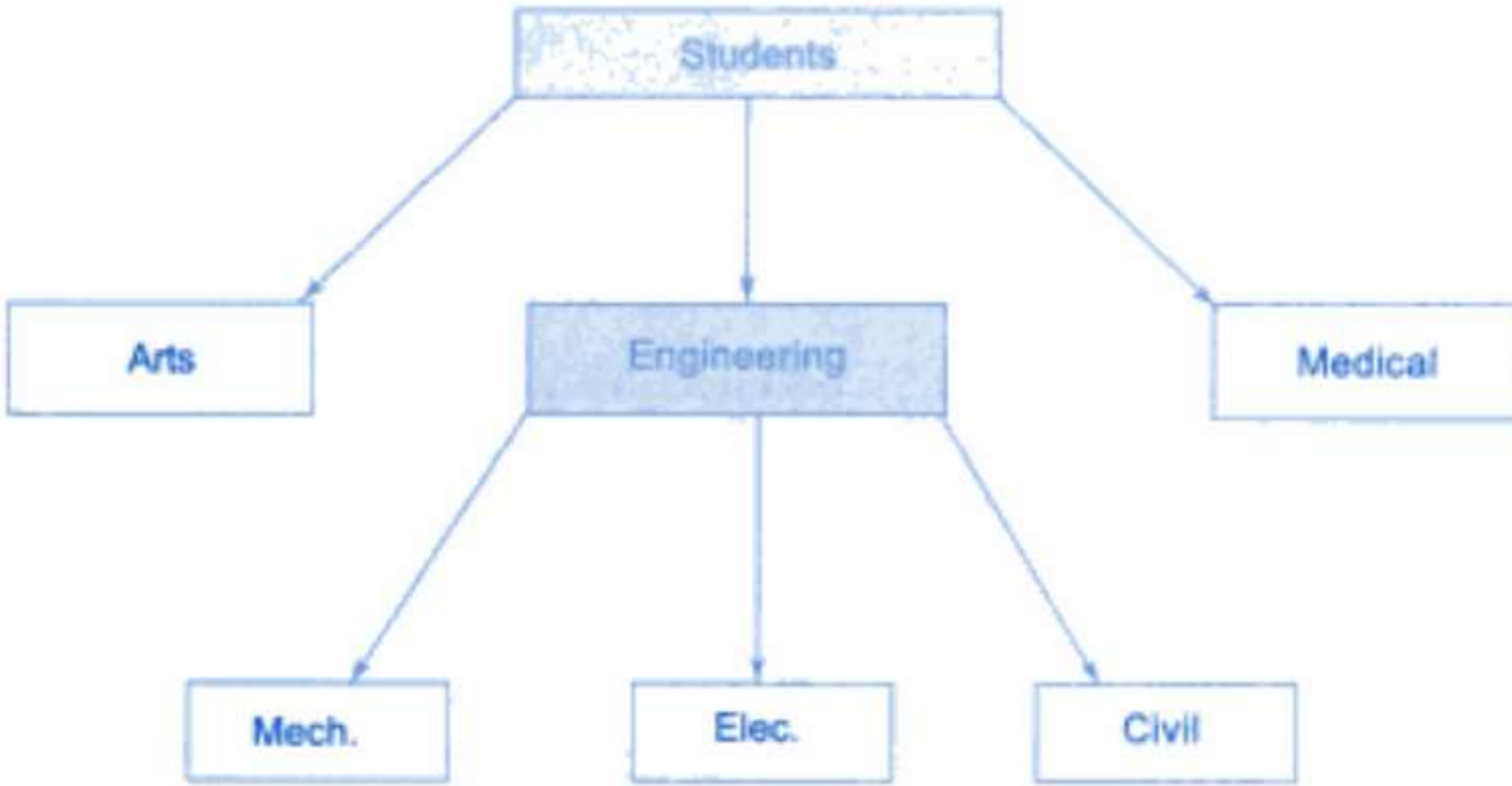


Fig. 8.9 ⇔ *Hierarchical classification of students*

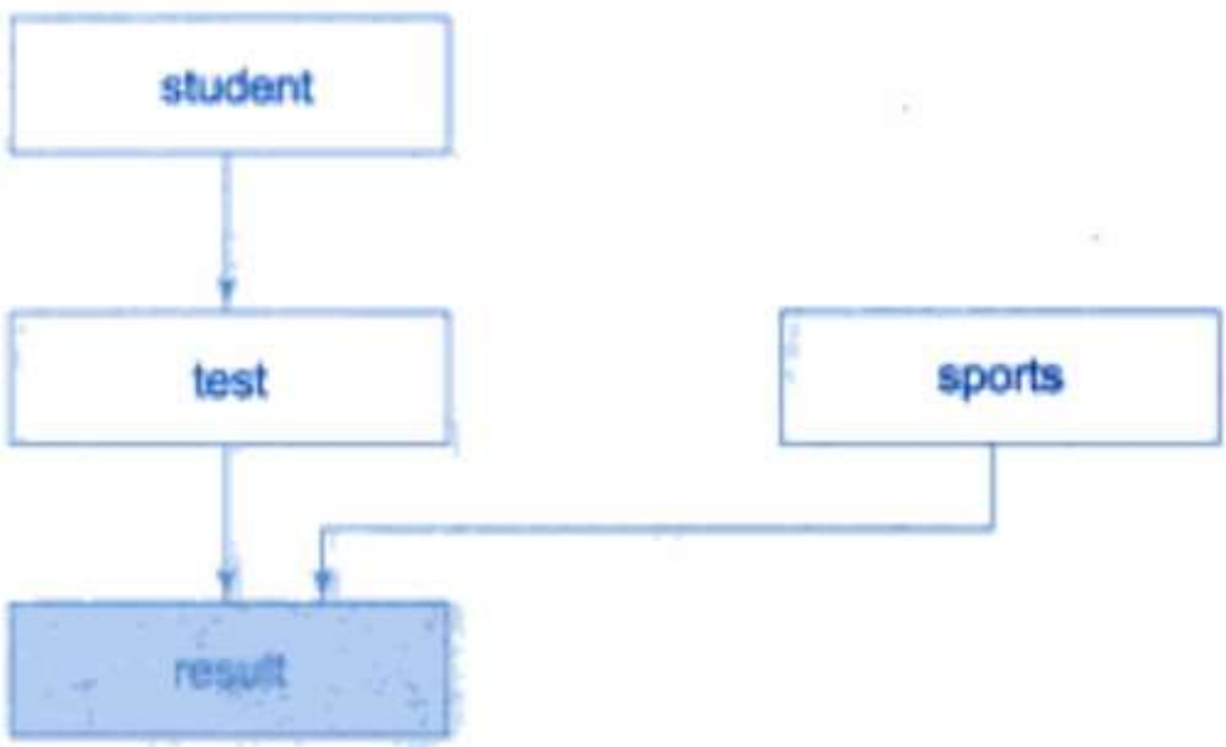
Hierarchical Inheritance

- Two or more classes can be derived from one base class.
- These derived classes are called subclasses.
- Base class includes all the features that are common to the subclasses.

```
class A // base class
{
    ...............
};
class B : access_specifier A // derived class from A
{
    ............
 } ;
 class C : access_specifier A // derived class from A
 {
    ............
 } ;
 class D : access_specifier A // derived class from A
 {
     ............
 } ;
```

Hybrid Inheritance

- Combination of two or more types of inheritance to design a program is called Hybrid Inheritance.



```
class student
{
  protected:
      int  roll_number;
  public:
      void get_number(int a)
      {
           roll_number = a;
      }
      void put_number(void)
      {
           cout << "Roll No: " << roll_number << "\n";
      }
};
```

```
class test : public student
{
  protected:
      float part1, part2;
  public:
      void get_marks(float x, float y)
      {
            part1 = x;  part2 = y;
      }
      void put_marks(void)
      {
            cout << "Marks obtained: " << "\n"
                 << "Part1 = " << part1 << "\n"
                 << "Part2 = " << part2 << "\n";
      }
};
```

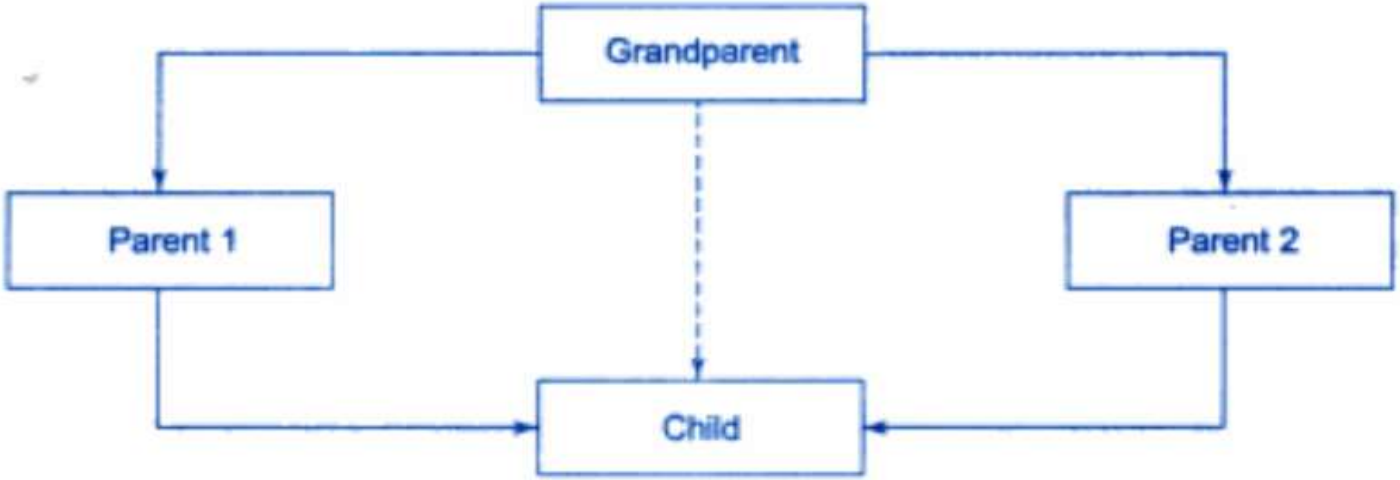
```
class sports
{
   protected:
       float score;
   public:
       void get_score(float s)
       {
            score = s;
       }
       void put_score(void)
       {
            cout << "Sports wt: " << score << "\n\n";
       }
};

class result : public test, public sports
{
       float total;
  public:
       void display(void);
};
```

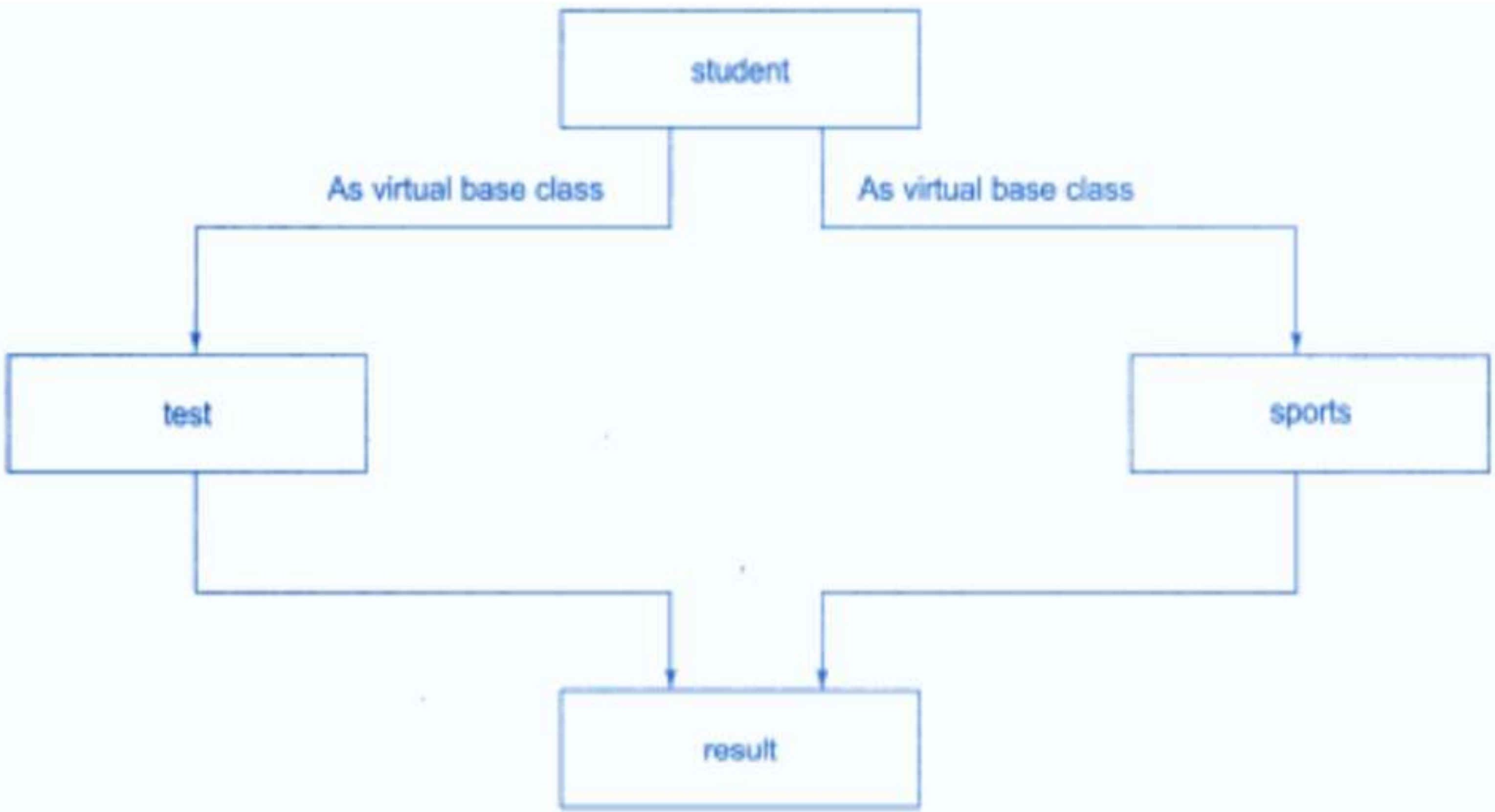
```
void result :: display(void)
{
      total = part1 + part2 + score;

      put_number();
      put_marks();
      put_score();

      cout << "Total Score: " << total << "\n";
}

int main()
{
      result student_1;
      student_1.get_number(1234);
      student_1.get_marks(27.5, 33.0);
      student_1.get_score(6.0);
      student_1.display();

      return 0;
}
```

```
Roll No: 1234
Marks obtained:
Part1 = 27.5
Part2 = 33
Sports wt: 6

Total Score: 66.5
```

Virtual Base Class

- It is a way of preventing multiple instances of a given class appearing in an inheritance hierarchy when using multiple/hybrid inheritance.
- If two or more classes are derived from a common base class, and then a child is derived from these inherited classes using multiple inheritance, ambiguity arises due to multiple paths.
- To avoid this, we can declare base class as virtual when it is inherited.
- Such a class is known as virtual base class.


Grandparent
Parent 1
Parent 2
Child


student
As virtual base class
As virtual base class
test
sports
result

Syntax: Virtual Base Class

```
class A                                // grandparent
{
      .....
      .....
};
class B1 : virtual public A      // parent1
{
      .....
      .....
};
class B2 : public virtual A      // parent2
{
      .....
      .....
};
class C : public B1, public B2  // child
{
      .....                   // only one copy of A
      .....                   // will be inherited
};
```

- When a class is made virtual,
- There may be multiple paths between virtual base class and a derived class.
- only one copy of that class in Inherited.

VIRTUAL BASE CLASS

```
#include <iostream>

using namespace std;

class student
{
  protected:
    int roll_number;
  public:
    void get_number(int a)
    {
          roll_number = a;
    }
    void put_number(void)
    {
          cout << "Roll No: " << roll_number << "\n";
    }
};
```

```
class test : virtual public student
{
  protected:
    float part1, part2;
  public:
    void get_marks(float x, float y)
    {
         part1 = x;  part2 = y;
    }
    void put_marks(void)
    {
         cout << "Marks obtained: " << "\n"
              << "Part1 = " << part1 << "\n"
              << "Part2 = " << part2 << "\n";
    }
};

class sports : public virtual student
{
  protected:
    float score;
  public:
    void get_score(float s)
  {
         score = s;
  }
  void put_score(void)
  {
         cout << "Sports wt: " << score << "\n\n";
  }
};
```

```
class result : public test, public sports
{
    float total;
  public:
        void display(void);
};
void result :: display(void)
{
        total = part1 + part2 + score;

        put_number();
        put_marks();
        put_score();

        cout << "Total Score: " << total << "\n";
}

int main()
{
        result student_1;
        student_1.get_number(678);
        student_1.get_marks(30.5, 25.5);
        student_1.get_score(7.0);
        student_1.display();

        return 0;
}
```

The output of Program

```
Roll No: 678
Marks obtained:
Part1 = 30.5
Part2 = 25.5
Sport wt: 7

Total Score: 63
```

Abstract Classes

- Abstract class is designed only to act as a base class.
  - Not used to create objects.
  - Used for creating derived classes only.
- A class can only be considered as an abstract class if it has at least one pure virtual function.

```
class vehicle //abstract base class
{
    private:
        data-type d1;
        data-type d2;

    public:
        virtual void spec()=0; //pure virtual function
};

class LMV : public vehicle
{

    public:
        void spec()
         {
             //LMV definition of spec function
         }
};
```

```
class HMV : public vehicle
{
    public:
        void spec()
        {
            //HMV definition of spec function
        }
};

class TW : public vehicle
{
    public:
        void spec()
        {
            //TW definition of spec function
        }
};
```

Constructors in derived classes

- If any base class is having a constructor with one or more arguments
  - It is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.
  - The base class constructor is executed first and then the constructor in the derived class is executed.
- In multiple inheritance,
  - base classes are constructed in the order in which they appear in the declaration of derived class.

General form of defining a derived constructor

```
Derived-constructor     (Arglist1, Arglist2, ... ArglistN,   Arglist(D)

        base1(arglist1),
        base2(arglist2),
        .....
        .....
        .....                 arguments for base(N)
        baseN(arglistN),
        {
              Body of derived constructor
        }
```

Constructors for virtual base class

- Constructors for virtual base classes are invoked before non virtual base classes.
- In case of multiple virtual base classes, they are invoked in the order of declaration.
- The execution sequence of construtors is as follows:
  - Constructors for virtual base class
  - Constructors for non virtual base class
  - Constructor for derived class

Execution of base class constructors

Method of inheritance	***Order of execution***

Class B: public A
{
};
	A( ) ; base constructor
	B( ) ; derived constructor

class A : public B, public C
{
};
	B( ) ; base(first)
	C( ) ; base(second)
	A( ) ; derived

class A : public B, virtual public C
{
};
	C( ) ; virtual base
	B( ) ; ordinary base
	A( ) ; derived

CONSTRUCTORS IN DERIVED CLASS

```
#include <iostream>

using namespace std;

class alpha
{
      int x;
   public:
      alpha(int i)
      {
            x = i;
            cout << "alpha initialized \n";
      }
      void show_x(void)
      { cout << "x = " << x << "\n"; }
};

class beta
{
      float y;
   public:
      beta(float j)
      {
            y = j;
            cout << "beta initialized \n";
      }
      void show_y(void)
      { cout << "y = " << y << "\n"; }
};
```

```
class gamma: public beta, public alpha
{
    int m, n;
  public:
    gamma(int a, float b, int c, int d):
         alpha(a), beta(b)
    {
         m = c;
         n = d;
         cout << "gamma initialized \n";
    }
    void show_mn(void)
    {
         cout << "m = " << m << "\n"
              << "n = " << n << "\n";
    }
};
```

```
int main()
{
    gamma g(5, 10.75, 20, 30);
    cout << "\n";
    g.show_x();
    g.show_y();
    g.show_mn();

    return 0;
}
```

The output of Program

```
beta initialized
alpha initialized
gamma initialized

x = 5
y = 10.75
m = 20
n = 30
```

Initialization list in constructor function

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

- Assignment section
  - Body of constructor function.
  - Used to assign initial values to data members.
- Initialization section
  - Provides initial values to base constructors
  - Also initializes its own class members
  - Basically contains a list of initializations separated by commas.
  - Also called as initialization list.

Example

```
class XYZ
{
      int a;
      int b;
  public:
      XYZ(int i, int j) : a(i), b(2 * j) { }
};

main()
{
      XYZ x(2, 3);
}
```

The following statements are also valid:

```
XYZ(int i, int j) : a(i) {b = j;}
XYZ(int i, int j) { a = i; b = j;}
```

INITIALIZATION LIST IN CONSTRUCTORS

```
#include <iostream>

using namespace std;

class alpha
{
     int x;
   public:
     alpha(int i)
     {
          x = i;
          cout << "\n alpha constructed";
     }

     void show_alpha(void)
     {
          cout << " x = " << x << "\n";
     }
};
```

```
class beta
{
    float p, q;
  public:
    beta(float a, float b): p(a), q(b+p)
    {
         cout << "\n beta constructed";
    }
    void show_beta(void)
    {
         cout << " p = " << p << "\n";
         cout << " q = " << q << "\n";
    }
};
```

```
class gamma : public beta, public alpha
{
    int u,v;
  public:
    gamma(int a, int b, float c):
    alpha(a*2), beta(c,c), u(a)
    { v = b; cout << "\n gamma constructed"; }

    void show_gamma(void)
    {
    cout << " u = " << u << "\n";
    cout << " v = " << v << "\n";
    }
};
```

```
int main()
{
      gamma g(2, 4, 2.5);

      cout << "\n\n Display member values " << "\n\n";

      g.show_alpha();
      g.show_beta();
      g.show_gamma();

      return 0;
};
```

The output of Program

```
beta constructed
alpha constructed
gamma constructed

Display member values

x = 4
p = 2.5
q = 5
u = 2
v = 4
```

END OF UNIT-II