S. RAJALAKSHMI
AP/CSE, SJIT

UNIT - 2

OBJECT ORIENTED PROGRAMMING CONCEPTS

String Handling - Copy Constructor - Polymorphism - Compile time and run time polymorphism - Function Overloading - Operator Overloading - Dynamic Memory Allocation - Nested classes - Inheritance - Virtual functions.

String Handling:

→ A string is represented as a sequence of characters.

Creating String Objects:

(i) String S1;

(ii) String S2("PDS");

(iii) A string can also be assigned with other string

```
S1 = S2;
```

(iv) String Concatenation

```
String S3;
S3 += S1;
S3 = "abc" + S1;
```

(v) String can also be read using keyboard

```
cin >> S1;
```

(vi) String can be displayed

```
cout << S1;
```

(vii) To get a line of text

```
getline(cin, S1);
```

Example

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    String S1;                          // Empty string object
    String S2("HI");                    // using string constant
    String S3("HELLO");
    // Assigning values to string objects
    S1 = S2;
    Cout << "S1 = " << S1 << endl;
    // using string constant
    S1 = "C++";
    Cout << "Now S1 = " << S1 << endl;
    // using another object
    String S4(S1);
    Cout << "S4 = " << S4 << endl;
    // Reading through keyboard
    Cout << "Enter a string" << endl;
    Cin >> S4;
    Cout << "Now S4 = " << endl;
    // Concatenating strings
    S1 = S2 + S3;
    Cout << "S1 finally contains : " << S1 << endl;
    Return 0;
}
```

Output:

S1 = HI

Now S1 = C++

S4 = C++

Enter a string

Computer

Now S4 = Computer

S1 finally contains : HI HELLO

Manipulating String objects

Some Member functions:

insert ( ) - to insert a string into other string

replace ( ) - to replace characters in string

erase ( ) - to erase characters in string

append ( ) - to append one string at the end of other

Example:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    String S1("12345");
    String S2("abcde");
    Cout << " Original Strings are :" <<endl;
    Cout << "S1 =" << S1 <<endl;
    Cout << "S2 =" << S2 <<endl;
// Inserting a string into other
    S1.insert(4, S2);    // To insert S2 at 4th position in S1
    Cout << " Now S1 =" << S1 <<endl;
```

```
// Removing characters
    cout << "Remove 5 characters from S1" << endl
    S1.erase (4, 5);
    Cout << " S1 has " << S1 << endl;
// Replacing characters
    S2.replace (1, 3, S1);
    Cout << " Now S2 has " << S2 << endl;
return 0;
}
```

Output:

```
Original Strings are
S1: 12345
S2: abcde
    Now S1 = 1234abcde5
    Remove 5 characters from S1
    S1 has 12345
Now S2 has a12345e
```

Relational Operations

Operator	Meaning
=	Assignment
+	Concatenation
+=	Concatenation Assignment
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal to
[]	Subscription

compare() function can be used to compare two strings.

Example:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1("ABC");
    string s2("XYZ");
    string s3 = s1 + s2;
    if (s1 != s2)
    {
        cout << "s1 is not equal to s2" << endl;
    }
    if (s1 > s2)
    {
        cout << "s1 is greater than s2" << endl;
    }
    if (s3 == s1+s2)
    {
        cout << "s3 is equal to s1+s2" << end;
    }
    int x = s1.compare(s2);
    if (x == 0)
    {
        cout << "s1 == s2" << endl;
    }
    else if (x > 0)
    {
        cout << "s1 > s2" << endl;
    }
    else
    {
        cout << "s1 < s2" << endl;
    }
    return 0;
}
```

Output

S1 is not equal to S2

S2 greater than S1

S3 is equal to S1+S2

S1 < S2

Accessing characters

The member functions to access characters in a string are

at() – Retrieving characters in specified position

Substr() – Retrieving a substring at specified position.

find() – Returns the position of first character in a substring, if that substring is found

find_first_of() – Returns the position of first occurence of specified character

find_last_of() – Returns the position of last occurence of specified character

Example

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    String S1("RED GREEN YELLOW");
    // Displaying String using 'at';
    for(int i=0; i< S1.length(); i++)
    {
        cout << S1.at(i);
    }
```

```
    cout << endl;
    //Display using subscription
    for (int i=0; i<s1.length(); i++)
    {
        cout << s1[i];
    }
    cout << endl;
    int x = s.find("GREEN");
    cout << "GREEN is found at " << x << endl;
    int x1 = s.find_first_of('E');
    cout << "E is found first at " << x1 << endl;
    int x2 = s.find_last_of('E');
    cout << "E is found last at " << x2 << endl;
    return 0;
}
```

OUTPUT:

```
RED GREEN YELLOW
RED GREEN YELLOW
GREEN is found at 4
E is found first at 1
E is found last at 11
```

COMPARING & SWAPPING STRINGS

compare() - functions is used to compare two strings

Swap() - function is used to swap two strings

Example:

```
#include <iostream>
using name space std;
int main()
{
```

```
String S1("ROAD");
String S2("READ");
String S3("RED");
int x = S1.Compare(S2);
if (x == 0)
{
    cout << " S1 is equal to S2" <<endl;
}
else if (x == 1)
{
    cout << "S1 is greater than S2" <<endl;
}
else
{
    cout << " S2 is greater than S1" <<endl;
}
cout << "Before Swapping" << endl;
cout << "S1 = " << S1 <<endl;
cout << " S2 = " << S2 <<endl;
cout << "After Swapping" <<endl;
S1.Swap(S2);
Cout << "S1 = " << S1 <<endl;
Cout << "S2 = " << S2 << endl;
}
```

OUTPUT:

S1 is greater than S2
Before Swapping
S1 = ROAD
S2 = READ
After Swapping
S1 = READ
S2 = ROAD

Copy constructor takes reference of the object as a argument.

Syntax:

```
Classname (Classname &o)
{
        // body of constructor
}
```

Example: To generate fibonacci series using copy constructor

```
#include <iostream>
using namespace std;
class Fibonacci
{
    int fo;
    int f1;
    int fib;
  public:
    Fibonacci()
    {
        fo = 0;
        f1 = 1;
        fib = fo + f1;
    }
    Fibonacci (Fibonacci &a)
    {
        fo = a.fo;
        f1 = a.f1;
        fib = a.fb;
        cout << fo << "\t";
        cout << f1 << "\t";
    }
```

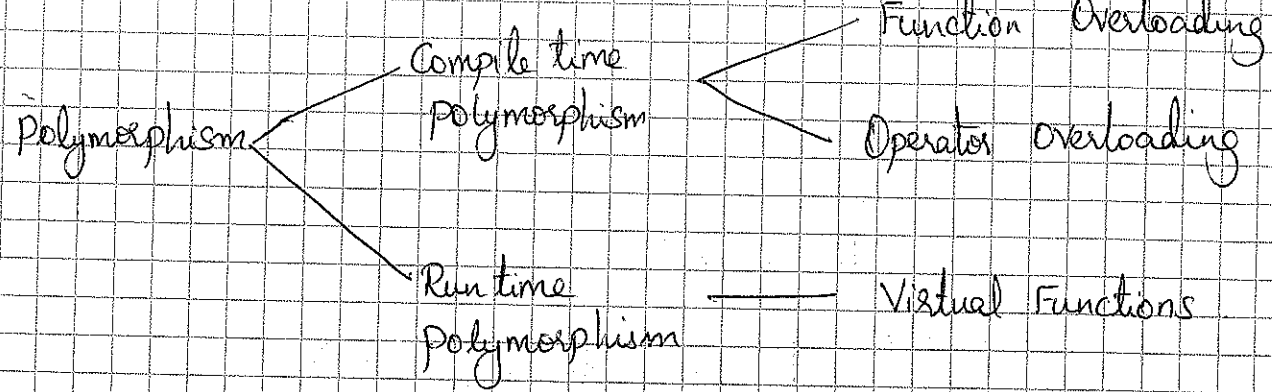
```
    void increment ( )
    {
        fo = f1;
        f1 = fib;
        fib = fo + f1;
    }
    void display ( )
    {
        cout << fib << "\t";
    }
};  // end of class
int main ( )
{
    int num;
    cout << "Enter number of terms" <<endl;
    cin >> num;
    Fibonacci number1;
    Fibonacci number(number1);
    for (int i=0; i <=num_series; i++)
    {
        number.display ( );
        number.increment ( );
    }
    return 0;
}
```

OUTPUT: Enter number of terms 5

0 1 1 2 3 5 8

POLYMORPHISM

Polymorphism means same code or operations or operators behave differently in different contexts



COMPILE TIME POLYMORPHISM

→ Same function name is used to accomplish different tasks

→ The overloaded member functions are selected for invoking by matching arguments, both type and number

→ This information is known to compiler at compile time and therefore the compiler is able to select appropriate function for a particular call at compile time itself.

→ It is also termed to be early binding or static binding or static linking.

FUNCTION OVERLOADING

→ Provide multiple definitions for the same function name in same scope

→ The definition of function must differ from each other based on the types of or number of arguments.

→ It cannot differ only based on return type of function.

Example:

```
#include <iostream>
using namespace std;
class Area
{
    private:
        int s;
        double l, b;
    public:
        void ComputeArea (int s)
        {
            cout << " Area of Square = " << s*s << endl;
        }
        void ComputeArea (double l, double b)
        {
            cout << " Area of Rectangle = " << l*b << endl;
        }
};
int main ()
{
    int a;
    double l1, b1;
    cout << "Enter Side of square " << endl;
    cin >> a;
    cout << "Enter length and breadth of rectangle " << endl;
    cin >> l1 >> b1;
    Area obj;
    obj. ComputeArea (a);
    obj. ComputeArea (l1, b1);
    return 0;
}
```

Output:

```
Enter the side of Square 4
Enter the length and breadth of rectangle 5 5
Area of Square : 16
Area of Rectangle : 25
```

OPERATOR OVERLOADING:

→ Ability to provide the operators with a special meaning for a data type.

→ The mechanism of giving such special meaning to an Operator is known as Operator overloading.

Operators that Cannot be overloaded are

→ Class member access operators (. , .*)

→ Scope resolution operator (::)

→ Size operator (sizeof)

→ Conditional Operator (?:)

Defining Operator overloading

→ The general form of an Operator function is

```
returntype  classname :: operator op (arglist)
{
        // FUNCTION BODY
}
```

→ return type is the type of value returned by the Specified operation.

→ op is the Operator being overloaded.

→ The process of overloading involves following steps:

1. Create a class that defines the data type that is to be used in overloading operation.

2. Declare the operator function op(), in the public part of class. It may be either a member function or friend function.

3. Define the operator function to implement required operation.

→ Overloaded operator functions can be invoked by expressions such as

```
op x
 (or)        } unary Operators
x op
```

This would be interpreted as

```
operator op (x)     // for Friend function
x.operator op()     // for Member function
```

For binary Operators,

```
x op y
```

The expression x op y would be interpreted as

For friend function,

```
operator op (x, y)
```

For Member functions

```
x.operator op (y)
```

RULES FOR OVERLOADING OPERATORS

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have atleast one operand that is of user defined type.
3. It is not possible to change the meaning of an operator. i.e, we cannot redefine plus (+) operator to subtract one value from other.
4. Overloaded operators follow syntax rules of original operators. They cannot be overridden.
5. Some operators that cannot be overloaded.
6. We cannot use friend function to overload certain operators. But member function can be used to overload them.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but those overloaded by means of a friend function take one reference argument (object)
8. Binary operators, overloaded through a member function take one explicit argument and those which are overloaded through a friend function can take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of relevant class.
10. Binary Arithmetic operators such as +, -, *, / must explicitly return a value. They must not attempt to change their own arguments.

Example : OVERLOADING UNARY OPERATOR AS MEMBER FUNCTION

```
#include <iostream>
using namespace std;
class increment
{
    int i;
    public:
        void get()
        {
            cin >> i;
        }
        void operator ++()
        {
            ++i;
        }
        void put()
        {
            cout << i;
        }
};
int main()
{
    increment s1;
    Cout << " Enter value " << endl;
    s1.get();
    ++s1;
    cout << " value after increment " << endl;
    s1.put();
    return 0;
}
```

OUTPUT:

Enter value 9

value after increament 10

Explanation:

The statement, ++s1; is interpreted as

s1. operator ++ ( )

EXAMPLE: OVERLOADING BINARY OPERATOR AS MEMBER FUNCTION

```
#include <iostream>
using namespace std;
class complex
{
    private:
        float rp;
        float ip;
    public:
        complex()
        {
            rp = 0.0;
            ip = 0.0;
        }
        void get()
        {
            cin >>rp >>ip;
        }
        void put()
        {
            cout << rp << "+i" <<ip;
        }
        Complex operator + (Complex c)
        {
```

```
        Complex t;
        t.rp = rp + c.rp;
        t.ip = ip + c.ip;
        return t;
    }
Complex operator - (Complex c)
{
    Complex t;
    t.rp = rp - c.rp;
    t.ip = ip - c.ip;
    return t;
}
Complex operator * (Complex c)
{
        Complex t;
        t.rp = rp * c.rp - ip * c.ip;
        t.ip = rp * c.ip + ip * c.rp;
        return t;
}
Complex operator / (Complex c)
{
        Complex t;
        float q;
        q = c.rp * c.rp + c.ip * c.ip;
    t.rp = (rp * c.rp + ip * c.ip) / q;
    t.ip = (ip * c.rp - rp * c.ip) / q;
        return t;
}
};
```

```
int main()
{
    Complex c1, c2, c;
    cout << " Enter rp and ip of first complex number" << endl;
    c1.get();
    cout << " Enter rp and ip of second complex number" << en
    c2.get();
    c = c1+c2;
    cout << " Result of Addition = " << endl;
    c.put();
    c = c1 - c2;
    cout << " Result of Subtraction = " << endl;
    c.put();
    c = c1 * c2;
    cout << " Result of Multiplication = " << endl;
    c.put();
    c = c1 / c2;
    cout << " Result of Division = " << endl;
    c.put();
    return 0;
}
```

Output:

```
Enter rp and ip of first Complex number = 3 5
Enter rp and ip of second Complex number = 2 1
Result of Addition = 5 + i6
```

Result of subtraction = 1 + i4

Result of Multiplication = 1 + i13

Result of Division = 2.2 + i).4

EXAMPLE: OVERLOADING UNARY OPERATOR AS FRIEND FUNCTION

```
#include <iostream>
using namespace std;
class sample
{
    private:
        int x;
        int y;
        int z;
    public:
        void get()
        {
            cin >> x >> y >> z;
        }
        void put()
        {
            cout << " value of x = " << x << endl;
            cout << " value of y = " << y << endl;
            cout << " value of z = " << z << endl;
        }
        friend void operator - (sample &);
};
int main()
{
```

```
    sample s;
    cout << " enter three integers " << endl;
    s.get();
    -s;
    s.put();
    return 0;
}
Void operator -(sample &r)
{
    r.x = -r.x;
    r.y = -r.y;
    r.z = -r.z;
}
```

OUTPUT:

```
Enter three integers  3
7
8
value of x = -3
value of y = -7
value of z = -8
```

Example: BINARY OPERATOR OVERLOADING AS FRIEND FUNCTION

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

```
class string
{
    char *p;
    int len;
    Public:
        string()
        {
            len = 0;
            p = 0;
        }
        string (char *s)
        {
            len = strlen(s);
            p = new char [len + 1];
            strcpy (p, s);
        }
        friend string operator + (string &, string &)
        void show()
        {
            cout << p;
        }
};
int main()
{
    string s1 = "PDS";
    String s2 = "IT";
    string s3;
    s3 = s1 + s2;
    cout << "Concatenated String = " << endl;
    s3. show();
    return 0;
}
```

```
string operator +(string &s1, string &s2)
{
    string t;
    t.len = s1.len + s2.len;
    t.p = new char[t.len + 1];
    strcpy(t.p, s1.p);
    strcat(t.p, s2.p);
    return t;
}
```

OUTPUT:

Concatenated string = PDS II

RUNTIME POLYMORPHISM

→ The function call is resolved at run time.

→ The compiler determines the type of object at run time and then binds the function call.

→ Such resolution of function call at run time is termed to be run time polymorphism.

→ It is otherwise termed to be late binding or Dynamic binding or Dynamic linking.

Output:

Enter the number of terms : 2

0 1 1 2 3

DYNAMIC MEMORY ALLOCATION

Dynamic Memory Allocation : done explicitly done by Programmer.

* Programmer explicitly requests the system to allocate memory and return starting address of memory allocated. This address can be used by the Programmer to access the allocated memory.

* When done using memory, it must be explicitly freed.

EXPLICITLY ALLOCATING MEMORY IN C++ : NEW Operator.

→ used to dynamically allocate memory.

→ Can be used to allocate a single variable / object or an array of variables / objects.

→ The new operator returns pointer to the type alloted.

→ The new operator is used to create a heap memory space for an object of a class.

Basically, an allocation expression must carry out following three things:

→ Find storage for the object to be created.

→ Initialize that object.

→ Return a suitable pointer type to the object.

```
char * my_char_ptr = new char;
int  * my_int_array = new int[20];
```

EXPLICITLY FREEING MEMORY IN C++ : DELETE OPERA

→ Used to free memory allocated with 'new' operator.

→ The delete operator should be called on a pointer to dynamically allocated memory where it is no longer needed.

→ Can delete a single variable/object or an array

```
delete ptr_name;
delete [] array_name;
```

→ After 'delete' is called on a memory region, that region should no longer be accessed by the progr

→ Any 'new' must have a corresponding 'delete', if not, the program has Memory leak.

DYNAMIC MEMORY ALLOCATION

C++ program to perform Arithmetic Operation

```
#include <iostream>
using namespace std;
int main ()
{
    int *ptr_a = new int ;
    int *ptr_b = new int ;
    int *ptr_sum = new int ;
    int *ptr_sub = new int ;
    int *ptr_mul = new int ;
    int *ptr_div = new int ;
    cout << " Enter two integers : " <<endl;
    cin >> *ptr_a >>*ptr_b ;
    *ptr_sum = *ptr_a + *ptr_b;
    *ptr_sub = *ptr_a - *ptr_b;
    *ptr_mul = *ptr_a * *ptr_b;
    *ptr_div = *ptr_a / *ptr_b;
    cout << " Addition = " << *ptr_sum << endl;
    cout << " Subtraction = " << *ptr_sub <<endl;
    cout << " Multiplication = " << *ptr_mul <<endl;
    cout << " Division = " << *ptr_div << endl;
    delete ptr_a;
    delete ptr_b;
    delete ptr_sum;
    delete ptr_sub;
    delete ptr_mul;
    delete ptr_div;
}   return 0;
```

OUTPUT :

```
Enter two integers : 10 5
Addition = 15
Subtraction = 5
Multiplication = 50
Division = 2
```

EXAMPLE : DYNAMIC MEMORY ALLOCATION FOR OBJECTS

```
#include <iostream>
using namespace std;
class sample
{
    private:
        int x, y;
    public:
        void getdata()
        {
            cout << "Enter the value for x" << endl;
            cin >> x;
            cout << "Enter the value for y" << endl;
            cin >> y;
        }
        void display()
        {
            cout << "value of x = " << x << endl;
            cout << "value of y = " << y << endl;
        }
};
int main()
{
    sample *ptr;
```

```
    ptr = new sample;
    ptr -> getdata ();
    ptr -> display ();
    delete ptr;
    return 0;
}
```

OUTPUT

```
Enter the value for x 3
Enter the value for y 4
Value of x = 3
Value of y = 4
```

INHERITANCE

→ The mechanism that allows to extend the class without making any changes is called inheritance.

→ Inheritance allows us to create new classes from existing class.

⇒ A new class is called derived class and existing class is called base class.

Types:

Single Inheritance - A derived class with only one base class.

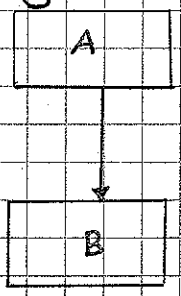
Multiple Inheritance - A derived class with several base class.

Multilevel Inheritance - Mechanism of deriving a class from another derived class.

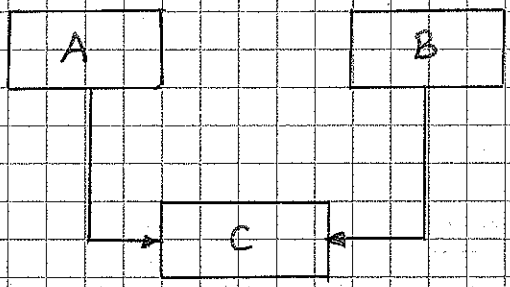
Hierarchical Inheritance - One base class, more derived class

Hybrid Inheritance - Combination of one or more inheritance.

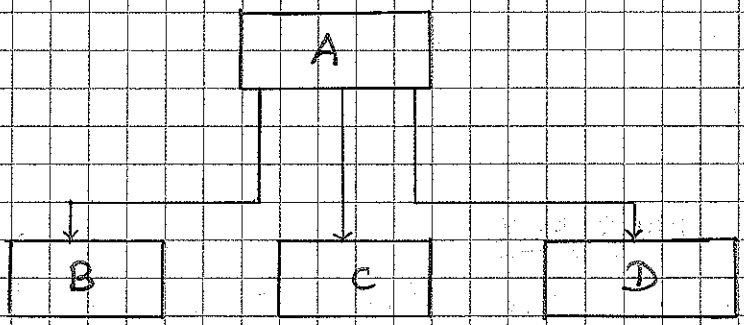
(i) Single inheritance



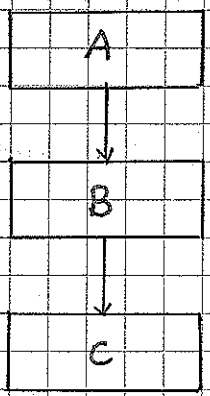
ii) Multiple inheritance



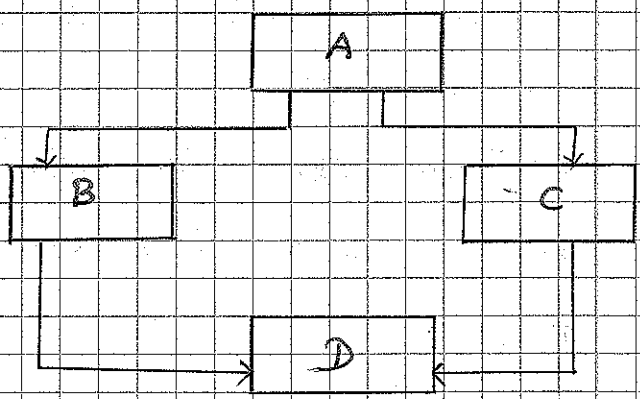
iii) Hierarchical inheritance



iv) Multilevel inheritance



v) Hybrid inheritance



Defining derived class

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

→ The colon indicates that the derived class name is derived from base-class-name.

→ The visibility mode is optional, if present may be either private or public.

→ The default visibility mode is private.

Ex:

```
class ABC : public XYZ    // public derivation
{

};
```

→ When a base class is privately inherited by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of derived class. They are inaccessible to objects of derived class.

→ When the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class, and therefore they are accessible to objects of derived class.

* In both cases, the private members are not inherited and therefore the private members of a base class will never become the members of its derived class

SINGLE INHERITANCE

```
#include <iostream
using namespace <std>;
class B
{
    int a;              // private ; not inheritable
    public:
        int b;          // public ; inheritable
        void get_ab();
        int get_a();
        void show_a();
};
class D : public B
{
    int c;
    public:
        void mul();
        void display();
};
void B :: get_ab()
{
    a = 5;
    b = 10;
}
int B :: get_a()
{
    return a;
}
void B :: show_a()
{
    cout << "a = " << a << endl;
}
```

```
void D :: mul ()
{
    c = b * get_a ();
}
void D :: display ()
{
    cout << "a = " << get_a () << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
}
int main ()
{
    D d;                    // creating object for derived class
    d.get_ab ();            // Accessing base class member function
                               with derived class object.
    d.mul ();
    d.show_a ();
    d.display ();
    d.b = 20;
    d.mul ();
    d.display ();
    return 0;
}
```

OUTPUT:

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

a = 5
b = 10
c = 100
```

Explanation:

Base class 'B'

Derived class 'D'

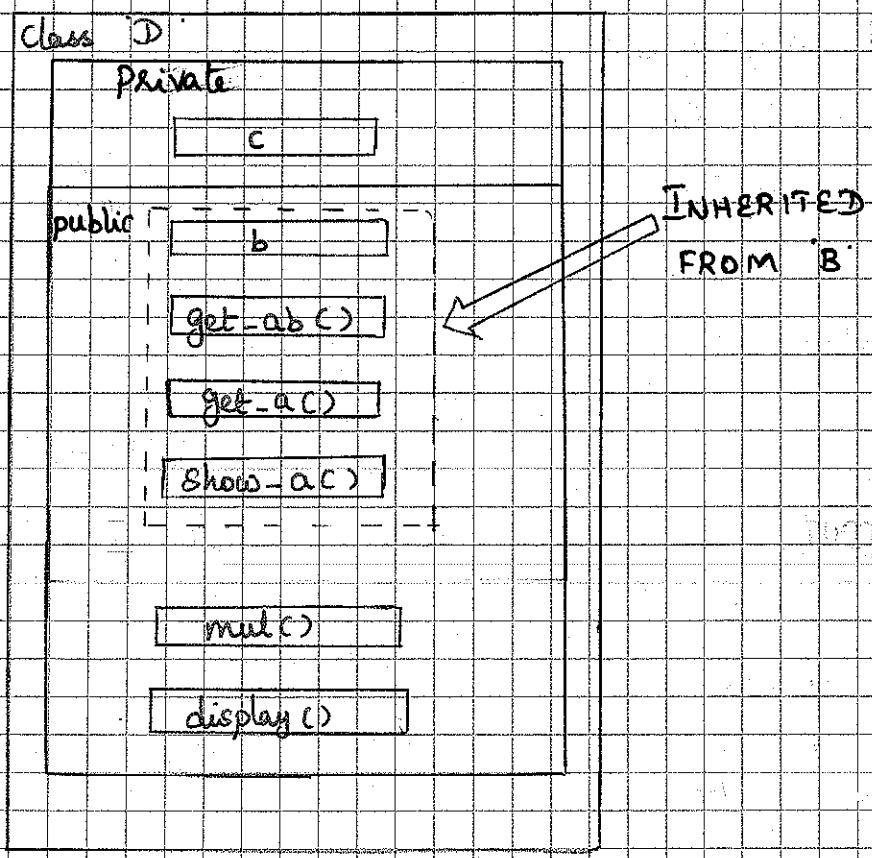
Class 'B' contains one private data member and one public data member and three public member functions.

Class 'D' contains one private data member and two public member functions.

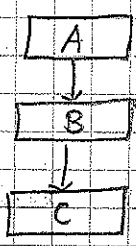
The class 'D' is a public derivation of class 'B'. Therefore 'D' inherits all public members of 'B' and retains their visibility.

Thus a public member of base class 'B' is also a public member of derived class 'D'.

The private members of 'B' cannot be inherited, by 'D'.



MULTILEVEL INHERITANCE



Base class

Intermediate Base class

Derived class

→ The class 'A' serves as a base class for Derived class B, which in turn serves as a base class for the derived class 'C'.

→ The class 'B' is known as intermediate base class, since it provides a link for the inheritance between 'A' and 'C'.

→ The chain 'ABC' is known as inheritance path.

EXAMPLE:

```
#include <iostream>
using namespace std;
class student
{
   protected:
            int roll_number;
   public:
            void get_number(int a);
            void put_number();
};
void student ::get_number(int a)
{
      roll_number = a;
}
```

```
void student :: put_number()
{
    cout << "Roll Number : " << roll_number << endl;
}
class test : public student // First level derivation
{
    protected:
        float sub1;
        float sub2;
    public:
        void get_marks (float s1, float s2);
        void put_marks();
};
void test :: get_marks (float x, float y)
{
    sub1 = x;
    sub2 = y;
}
void test :: put_marks()
{
    cout << "Marks in SUB1 = " << sub1 << endl;
    cout << "Marks in SUB2 = " << sub2 << endl;
}
class result : public test // Second level derivation
{
    float total;
    public:
        void display();
};
```

```
void result :: display ()
{
    total = sub1 + sub2;
    put_number();
    put_marks();
    cout << "Total = " << total << endl;
}
int main()
{
    result stud1;
    stud1.get_number(111);
    stud1.get_marks(75.0, 80.0);
    stud1.display();
    return 0;
}
```

OUTPUT :

```
Roll Number : 111
Marks in SUB1 = 75
Marks in SUB2 = 80
Total = 155
```

Class result

private
- total

public
- get_number(int)
- put_number()
- get_marks(float x, float y)
- put_marks()
- display()

Protected
- roll_number
- sub1
- sub2

MULTIPLE INHERITANCE

A class is derived from more than one base class.

Example:

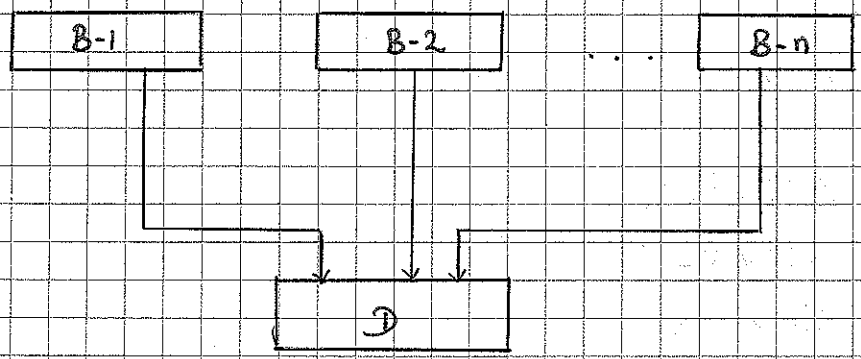
```
#include<iostream>
using namespace std;
class A
{
   protected:
            int a;
   public:
            A(int x)
            {
                a = x;
            }
            void get()
            {
                cin>>a;
            }
            void put()
            {
                cout<<"a = "<<a<<endl;
            }
};
class B
{
  protected:
            int b;
  public:
            B(int y)
            {
                b=y;
            }
            void get()
            {
                cin>>b;
            }
```

```
        void put()
        {
            cout << "b = " << b << endl;
        }
};
class C : public A, public B
{
    private:
        int c;
    public:
        C(int z) : A(z), B(z)
        {
            c = z;
        }
        void getc()
        {
            A::getc();
            B::getc();
            cin >> c;
        }
        void putc()
        {
            A::putc();
            B::put();
            cout << "c = " << c << endl;
            cout << "Sum = " << a+b+c << endl;
        }
};
int main()
{
    C obj(0);
    cout << "Enter three integer values for a, b, c " << endl;
    obj.getc();
    cout << "values are : ";
```

```
    obj.put();
    return 0;
}
```

OUTPUT:

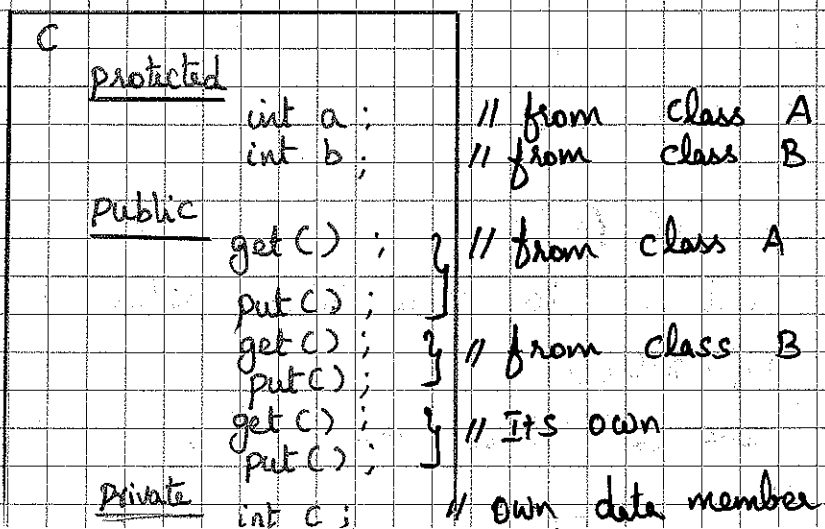
```
Enter three integer values for a, b and c
4  5  6
values are
a = 4
b = 5
c = 6
Sum = 15
```



SYNTAX

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

};
```



HIERARCHICAL INHERITANCE

→ Multiple class inherit from more than one base class.

Example:

```
#include <iostream>
using namespace std;
class A
{   protected:
            int a;
};
class B : public A
{   int b;
    public:
        void get()
        {
            cin>>a>>b;
        }
        int add()
        {
            return a+b;
        }
};
class C : public A
{
    int c;
    public:
        void get()
        {
            cin >>a >> c;
        }
        int product()
        {
            return a*c;
        }
};
```

```
class D : public A
{
    int d;
  public:
    void get()
    {
        cin >> a >> d ;
    }
    int divide()
    {
        return a % d ;
    }
};

int main()
{
    B  o1 ;
    cout << " Enter two integers to add " << endl;
    o1. get() ;
    cout << " Sum = " << o1.add() ;
    C  o2 ;
    cout << " Enter two integers to multiply " << endl;
    o2. get() ;
    cout << " Product = " << o2. product() ;
    D  o3 ;
    cout << " Enter two integers to divide " << endl;
    o3. get() ;
    cout << " Remainder = " << o3. divide() ;
    return 0 ;
}
```

OUTPUT

```
Enter two integers to add 4 5
Sum = 9
Enter two integers to multiply 5 5
product = 25
Enter two integers to divide 37 6
Remainder = 1
```

B

protected
int a ; //class A

Public
get()
add()

private
b

C

protected
int a //class A

public
get()
product()

private
c

D

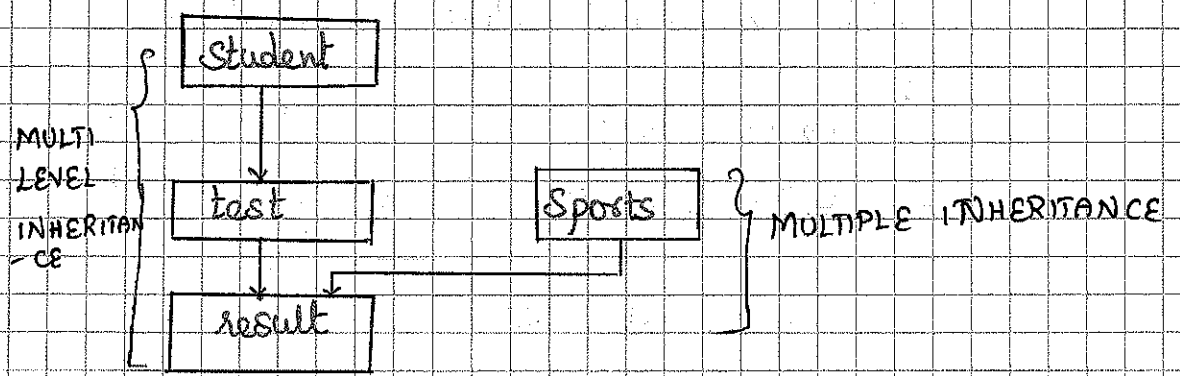
protected
int a // from cla A

public
get()
divide()

private
d

HYBRID INHERITANCE

-> Combination of one or more inheritances.



Example:

```
#include <iostream>
using namespace std;
class student
{
    protected:
        int roll_number;
    public:
        void get_number(int a)
        {
            roll_number = a;
        }
        void put_number()
        {
            cout << "Roll Number = " << roll_number << endl;
        }
};
class test : public student
{
    protected:
        float part1, part2;
    public:
        void get_marks(float x, float y)
        {
            part1 = x;
            part2 = y;
        }
        void put_marks()
        {
            cout << "Marks obtained : " << endl;
            cout << "Part 1 = " << part1 << endl;
            cout << "Part 2 = " << part2 << endl;
        }
};
```

```
class Sports
{
    protected:
            float score;
    public:
            void get_score (float s)
            {
                score = s;
            }
            void put_score ( )
            {
                cout << "Sports wt : " << score << endl;
            }
};
class result: public test, public sports
{
    float total;
    public:
            void display ( );
};
void result :: display ( )
{
    total = part1 + part2 + score;
    put_number ( );
    put_marks ( );
    put_score ( );
    cout << "Total score : " << total << endl;
}
int main ( )
{
    result s1;
    s1. get_number (1234);
    s1. get_marks (45, 50);
    s1. get_score (5.0);
```

```
    S1.display();
    return 0;
}
```

OUTPUT

```
Roll No: 1234
Marks obtained:
    Part 1 = 45
    Part 2 = 50
    Sports Wt = 5
    Total Score = 100
```

VIRTUAL FUNCTION

→ If the base class and derived class has member functions with same name, virtual functions provide the ability to call the member function of different class by same function call depending on different context.

→ The function in base class is declared as "virtual" using the keyword "virtual".

→ When a function is made virtual, C++ determines, which function to use at runtime based on the type of the object pointed to by base pointer, rather than type of pointer.

Example:

```
#include <iostream>
using namespace std;
class Base
{
  public:
      void display()
      {
         cout <<"Display base" <<endl;
      }
      virtual void show()
      {
         cout <<" show Base" <<endl;
      }
};
class Derived : public Base
{
  public:
      void display()
      {
         cout <<"Display derived" <<endl;
      }
      void show()
      {
         cout <<"show Derived" <<endl;
      }
};
int main()
{
   Base B;
   Derived D;
```

```
Base *bptr;
cout << "bptr points to Base" << endl;
bptr = &B;
bptr -> display();    // calls Base version
bptr -> show();       // calls Base version.
cout << "bptr points to Derived" << endl;
bptr = &D;
bptr -> display();    // calls Base version
bptr -> show();       // calls Derived version
return 0;
}
```

OUTPUT

```
bptr points to Base
Display base
Show base
bptr points to Derived
Display base
Show derived.
```

When 'bptr' is made to point to the object D, the statement

```
bptr -> display();
```

calls only the function associated with Base, whereas the statement

```
bptr -> show();
```

calls derived version of show().

RULES FOR VIRTUAL FUNCTION

1. The virtual functions must be members of class
2. They cannot be static members
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototype of base class version of a virtual function and all the derived class versions must be identical.
7. We cannot have virtual constructor, but can have virtual destructors
8. While a base pointer can point to any type of derived object, the reverse is not true.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to next object of derived class.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in derived class.

PURE VIRTUAL FUNCTION

→ A pure virtual function is a function declared in a base class that has no definition relative to the base class.

→ The compiler requires each derived class to either define the function or redeclare it as pure virtual function.

→ otherwise called 'do-nothing' function

Syntax:

```
virtual returntype fname ( ) = 0 ;
```

Example:

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

class Shape
{
    public:
        virtual void draw ( ) = 0 ;
};
class circle : public Shape
{
    public:
        void draw ( )
        {
            cout << " Drawing circle " << endl ;
        }
};
class Rectangle : public shape
{
    public:
        void draw ( )
        {
            cout << " Drawing Rectangle " << endl ;
        }
};
class Square : public Shape
{
    public:
        void draw ( )
        {
```

```
    cout << "Drawing Square" << endl;
    }
};  int main()
{
    circle c;
    Rectangle r;
    Square s;
    Shape *sh[] = {&c, &r, &s};
    for (int i=0; i<3; i++)
    {
        sh[i] -> draw();
    }
    return 0;
}
```

OUTPUT:

```
Drawing circle
Drawing Rectangle
Drawing Square
```

Runtime Polymorphism

```
#include <iostream>
#include <string>

using namespace std;

class media
{
   protected:
         char title[50];
         float price;
    public:
        media (char *s, float a)
        {
            strcpy (title, s);
            price = a;
        }
        virtual void display ( ) // EMPTY VIRTUAL FUNCTION
        {

        }
};
class book : public media
{
      int pages;
      public:
          book (char *s, float a, int p) : media (s,a)
          {
          }    pages = p;
```

```
        void display ( ) ;
};
class tape : public media
{
        float time ;
        public :
            tape ( char *s, float a, float t) : media (s,a)
            {
                time = t ;
            }
            void display ( ) ;
};
void book :: display ( )
{
    cout << "\n Title : " << title ;
    cout << "\n Pages : " << pages ;
    cout << "\n Price : " << price ;
}
void tape :: display ( )
{
    cout << "\n Title : " << title ;
    cout << "\n play time : " << time << "mins" ;
    cout << "\n price : " << price ;
}
```

```
int main ()
{
    char *title = new char[30];
    float price, time;
    int pages;
    //BOOK DETAILS
    cout << "\n Enter Book Details \n";
    cout << " Title: ";
    cin >> title;
    cout << " Price : ";
    cin >> price;
    cout << " Pages : ";
    cin >> pages;
  book book1 (title, price, pages);
    // Tape details
    cout << "\n Enter Tape Details \n";
    cout << " Title : ";
    cin >> title;
    cout << " Price : ";
    cin >> price;
  cout << " Playtime (mins) : ";
  cin >> time;
```

```
tape tape1 (title, price, time);
media *list[2];
list[0] = &book1;
list[1] = &tape1;
cout << "\n MEDIA DETAILS ";
cout << "\n ... BOOK .... ";
list[0] -> display();
cout << "\n ... TAPE .... ";
list[1] -> display();
result 0;
}
```

FRIEND FUNCTION

→ To make an outside function friendly to a class.

→ Syntax:

```
class ABC
{
    public:
        friend void XYZ ( );
};
```

→ The function declaration should be preceded by the keyword 'friend'.

→ The function is defined elsewhere in the program like normal function.

→ The function definition doesnot use either the keyword 'friend' or scope resolution operator '::'.

→ The functions that are declared with the keyword friend are known as 'friend functions'.

→ A friend function is not a member function. but has full rights over private members of class.

→ characteristics:

* It is not in the scope of the class, to which it is declared as friend.
* It cannot be called using object of class.
* It can be invoked like normal function, without help

of any object.

* Member names can be accessed with the help of object and dot membership operator.
* Can be declared either in public or private part of class.
* Usually it has object as arguments.

Example:

```
#include <iostream>
using namespace std;
class Sample
{
    int a;
    int b;
    public:
        void setvalue()
        {
            a = 25;
            b = 40;
        }
        friend float mean (Sample s);
};
float mean (Sample S)
{
    float temp;
    temp = (S.a + S.b) /2.0;
    return temp;
}
```

```
int main()
{
    Sample X;
    X.setvalue();
    Cout << "Mean value = " << mean(X) << endl;
    return 0;
}
```

O/P

Mean value = 32.5

Member function of one class can be friend of other

```
class X
{
    int fun1();        // Member function of X
};
class Y
{
    friend int X::fun1();
};
```

A function friend to two classes

```
#include <iostream>
using namespace std;
class ABC;
class XYZ
{
    int x;
  Public:
    void setvalue (int i)
    {
        x = i;
    }
    friend void max (XYZ x1, ABC a1);
};
class ABC
{
    int a;
  Public:
    void setvalue (int i)
    {
        a = i;
    }
    friend void max (XYZ x1, ABC a1);
};
Void Max (XYZ m, ABC n)
{
    if (m.x >= n.a)
        Cout << " Max = " << m.x << endl;
    else
        Cout << " Max = " << n.a << endl;
}
```

```
int main ()
{
    ABC  abc
    abc. setvalue (10);
    XYZ  xyz;
    xyz. setvalue (20);
    max (xyz, abc);

    return 0

}
```

FRIEND CLASS

→ Member function of one class can be made as friend functions of another class.

→ Then the class is declared as 'friend class'

→ Syntax:

```
class Z
{

    friend class X ;   // All Member functions of
                          X are friends to Z

};
```

Example:

```
#include <iostream>
using namespace std;
class A
{
   int x, y;
```

```
public:
    void get()
    {
        cin >> x >> y ;
    }
    friend class B ;
};
class B
{
    int z;
    public:
        void get()
        {
            cin >> z ;
        }
        int Sum(A obj)
        {
            return obj.x + obj.y + z ;
        }
};
int main()
{
    A a;
    B b;
    cout << " Enter three integers = " ;
    a.get();
    b.get();
    cout << " Sum = " << b.Sum(a);
    return 0
}
```

OUTPUT:

```
Enter three integers = 2 3 4
Sum = 9
```