Lecture Notes

on

Object Oriented Programming

3CS4-06

Unit III

Department of Computer Science & Engineering
Jaipur Engineering College & Research Centre, Jaipur

Vision of the Institute

To become a renowned centre of outcome based learning and work toward academic, professional, cultural and social enrichment of the lives of individuals and communities.

Mission of the Institute

M1: Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.
M2: Identify, based on informed perception of Indian, regional and global needs, the areas of focus and provide platform to gain knowledge and solutions.
M3: Offer opportunities for interaction between academia and industry.
M4: Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

Vision of the Department

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

Mission of the Department

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.
M2: To provide opportunities for interaction between academia and industry.
M3: To provide platform for lifelong learning by accepting the change in technologies.
M4: To develop aptitude of fulfilling social responsibilities.

Program Outcomes (PO)

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and Computer Science & Engineering specialization to the solution of complex Computer Science & Engineering problems.
2. **Problem analysis**: Identify, formulate, research literature, and analyze complex Computer Science and Engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex Computer Science and Engineeringproblems and design system components or processes that meet the specified needs with appropriateconsideration for the public health and safety,and the cultural, societal, and environmentalconsiderations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of Computer Science and Engineering experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex Computer Science Engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional Computer Science and Engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional Computer Science and Engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the Computer Science and Engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings in Computer Science and Engineering.
10. **Communication**: Communicate effectively on complex Computer Science and Engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the Computer Science and Engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation andability to engage in independent and life-long learning in the broadest contextof technological changein Computer Science and Engineering.

Program Educational Objectives (PEO)

PEO1: To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science & Engineering by way of analyzing and exploiting engineering challenges.

PEO2: To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems inComputer Science and Engineering

PEO3: To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues for Computer Science & Engineering.

PEO4: To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self-motivated life-long learning needed for a successful professional career in Computer Science & Engineering.

PEO5: To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge in Computer Science & Engineering.

Course Outcomes

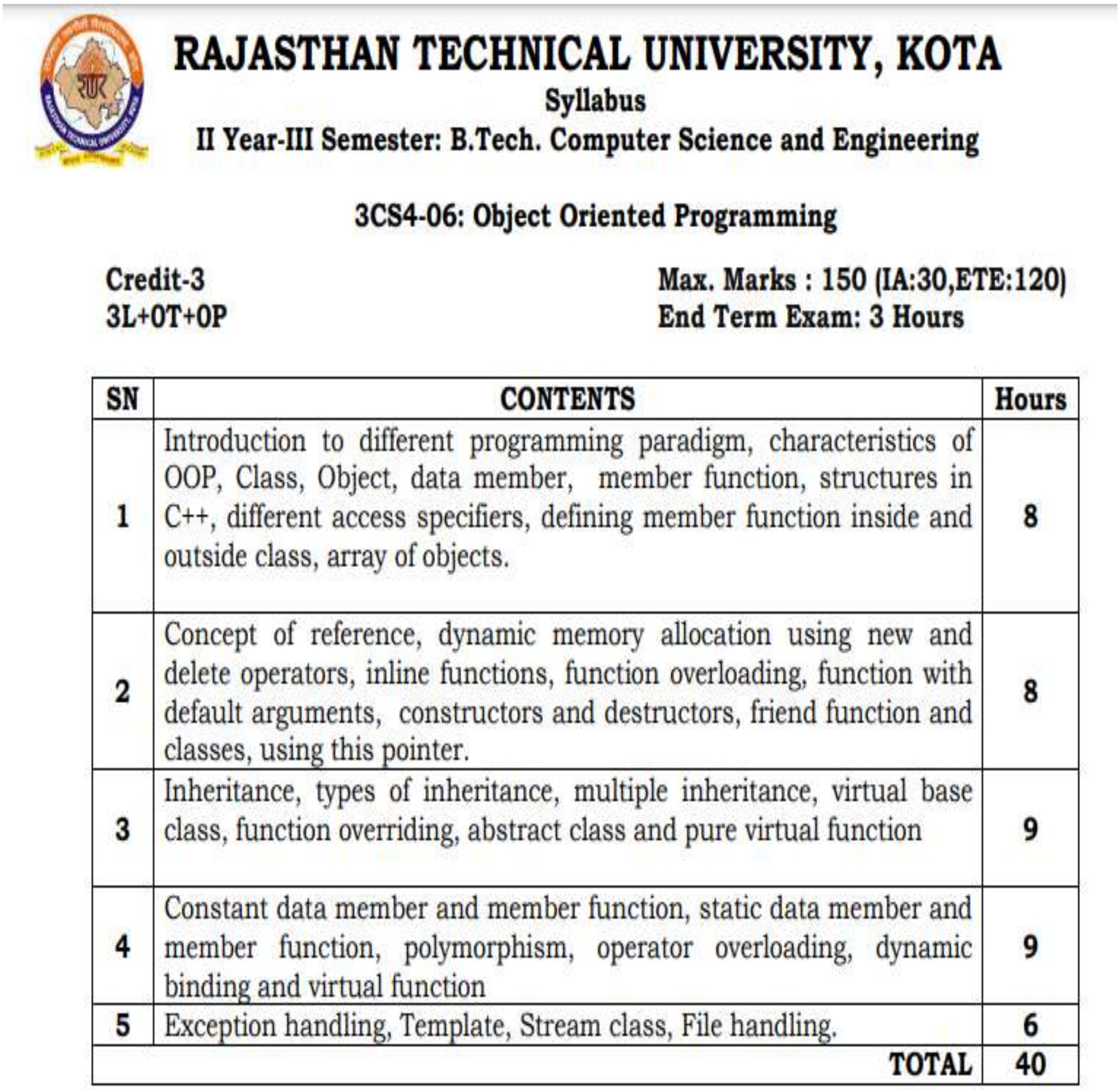
1. Understand the paradigms of object oriented programming in comparison of procedural oriented programming.
2. Apply the class structure as fundamental, building block for computational programming.
3. Apply the major object-oriented concepts to implement object oriented programs in C++.
4. Implement the concept of abstraction inheritance, polymorphism, dynamic binding and generic structure in building reusable code.

Mapping of Course Outcomes with Program Outcomes

H=3, M=2, L=1

Semester	**Subject**	**Code**	**L/T/P**	**CO**	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
III	**Object Oriented Programming**	**3CS4-06**	**L**	Understand the paradigms of object oriented programming in comparison of procedural oriented programming.	3	3	3	2	2	2	1	1	0	1	1	3
			L	Apply the class structure as fundamental, building block for computational programming.	3	3	3	3	2	2	1	1	1	2	1	3
			L	Apply the major object-oriented concepts to implement object oriented programs in C++.	3	3	3	3	2	1	2	2	1	2	1	3
			L	Implement the concept of abstraction inheritance, polymorphism, dynamic binding and generic structure in building reusable code.	3	3	3	2	1	1	1	1	1	1	1	3

Syllabus



RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

II Year-III Semester: B.Tech. Computer Science and Engineering

3CS4-06: Object Oriented Programming

Credit-3
3L+0T+0P

Max. Marks : 150 (IA:30,ETE:120)
End Term Exam: 3 Hours

SN	CONTENTS	Hours
1	Introduction to different programming paradigm, characteristics of OOP, Class, Object, data member, member function, structures in C++, different access specifiers, defining member function inside and outside class, array of objects.	8
2	Concept of reference, dynamic memory allocation using new and delete operators, inline functions, function overloading, function with default arguments, constructors and destructors, friend function and classes, using this pointer.	8
3	Inheritance, types of inheritance, multiple inheritance, virtual base class, function overriding, abstract class and pure virtual function	9
4	Constant data member and member function, static data member and member function, polymorphism, operator overloading, dynamic binding and virtual function	9
5	Exception handling, Template, Stream class, File handling.	6
	TOTAL	**40**

UNIT III

Inheritance:

Reaccessability is yet another feature of OOP's. C++ strongly supports the concept of reusability. The C++ classes can be used again in several ways. Once a class has been written and tested, it can be adopted by another programmers. This is basically created by defining the new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called 'INHERTTENCE'. This is often referred to as IS-A' relationship because very object of the class being defined "is" also an object of inherited class. The old class is called 'BASE' class and thenew one iscalled'DERIEVED'class.

Defining DerivedClasses

A derived class is specified by defining its relationship with the base class in addition to its own details. The general syntax of defining a derived class is as follows:

```
class d_classname : Access specifier baseclass name
{
__
___// members of derivedclass
};
```

The colon indicates that the a-class name is derived from the base class name. The access specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private. Visibility mode describes the status of derived features e.g.

```
class xyz         //baseclass
{
        members of xyz
};
class ABC :publicxyz      //publicderivation
{
        members of ABC
};
class ABC: XYZ     //private derivation (bydefault)
{
        members of ABC
};
```

In the inheritance, some of the base class data elements and member functions are inherited into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Single Inheritance

When a class inherits from a single base class, it is known as single inheritance. Following program shows the single inheritance using public derivation.

```
#include<iostream.h>
#include<conio.h>
class worker
{
```

```
            int age;
            char name [10];
            public:
            void get ( );
    };
    void worker : : get ( )
    {
            cout <<”yout name please”
            cin >> name;
            cout <<”your age please” ;
            cin >> age;
    }
    void worker :: show ( )
    {
            cout <<”In My name is :”<<name<<”In My age is :”<<age;
    }
    class manager ::publicworker      //derived class(publicly)
    {
            int now;
            public:
            void get ( ) ;
            void show ( ) ;
    };
    void manager : : get ( )
    {
            worker : : get ();      //the calling of base class input fn.
            cout << “number of workers underyou”;
            cin >> now;
            cin>>name>>age;
    }                        ( if they were public )
    void manager :: show ()
    {
            worker :: show();      //calling of base class o/pfn.
            cout <<“in No. of workers under me are: “ << now;
    }
            main ( )
    {
            clrscr ( ) ;
            worker W1;
            manager M1;
            M1 .get ( );
            M1.show ( ) ;
    }
```

If you input the following to this program:

```
            Your name please

            Ravinder

            Your age please

            27

            number of workers under you

            30
```

Then the output will be as follows:

My name is : Ravinder

My age is : 27

No. of workers under me are : 30

The following program shows the single inheritance by private derivation.

```
#include<iostream.h>

#include<conio.h>

classworker     //Base classdeclaration

{
        int age;
        char name [10] ;
        public:
        void get ( ) ;
        void show ( ) ;
};
void worker : : get ( )
{
        cout << “your name please” ;
        cin >> name;
        cout << “your age please”;
        cin >>age;
}
void worker : show ( )
{
        cout << “in my name is: “ <<name<< “in” << “my age is : “ <<age;
}
class manager : worker //Derived class (privately by default)
{
        int now;
        public:
        void get ( ) ;
        void show ( ) ;
};
void manager : : get ( )
{
        worker : : get ( ); //calling the get function of base
        cout << “number of worker under you”; class which is
        cin >> now;
        }
void manager : : show ( )
{
        worker : : show ( ) ;
        cout << “in no. of worker under me are : “ <<now;
}
main ( )
{
        clrscr ( ) ;
        worker wl ;
        manager ml;
        ml.get ( ) ;
        ml.show ( );
}
```

The following program shows the single inheritance using protected derivation

```
#include<conio.h>
#include<iostream.h>
classworker             //Base class declaration
{ protected:
        int age; char name [20];
        public:
        void get ( );
        void show ( );
};
void worker :: get ()
{
        cout >> "your name please";
        cin >>name;
        cout << "your age please";
        cin >> age;
}
void worker :: show ( )
{
        cout << "in my name is: " << name << "in my age is " <<age;
}
class manager:: protected worker // protected inheritance
{
        int now;
        public:
        void get ();
        void show ( ) ;
};
void manager : : get ( )
{
        cout << "please enter the name In";
        cin >> name;
        cout<< "please enter the age In"; //Directly inputting thedata
        cin>>age;       members of baseclass
        cout << " please enter the no. of workers under you:";
        cin >> now;
}
void manager : : show ( )

{
        cout « "your name is : "«name«" and age is : "«age;
        cout «"In no. of workers under your are : "«now;
main ( )
{
        clrscr ( ) ;
        manager ml;
        ml.get ( ) ;
```

```
                cout « "\n \n";
                ml.show ( );
        }
```

Making a Private Member Inheritable

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

a. **Private:** 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 be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derivedclass.

b. **Public:** On the other hand, 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 the objects of the derivedclass.

c. **Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these twoclasses.

The below mentioned table summarizes how the visibility of members undergo modifications when they are inherited

Base Class Visibility	Derived Class Visibility		
	Public	Private	Protected
Private	X	X	X
Public	Public	Private	Protected
Protected	Protected	Private	Protected

The private and protected members of a class can be accessed by:

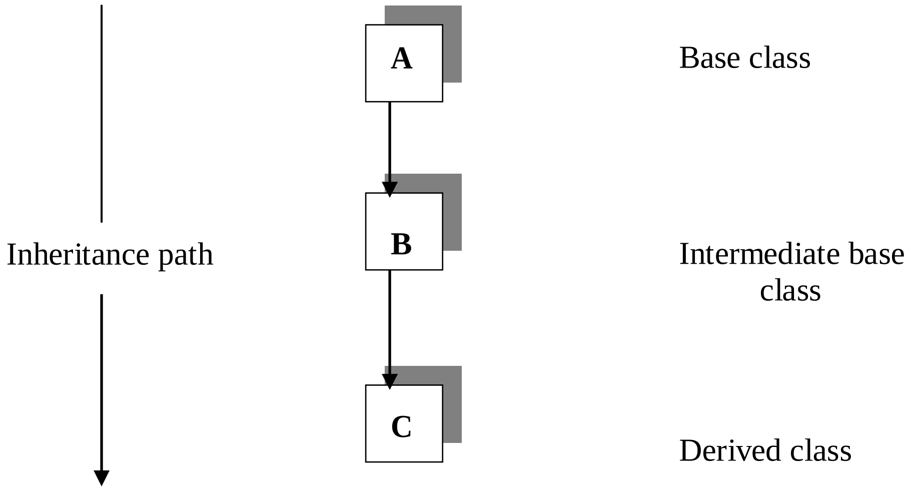
a. A function i.e. friend of aclass.

b. A member function of a class that is the friend of theclass.

c. A member function of a derived class.

Multilevel Inheritance

When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called ‘MULTILEVEL INHERITENCE’. The class B is known as the ‘INTERMEDIATE BASE CLASS’ since it provides a link for the inheritance between A and C. The chain ABC is called ‘ITNHERITENCE*PATH’ for e.g.



The declaration for the same would be:

```
        Class A
        {
        //body
        }
        Class B : public A
        {
        //body
        }
        Class C : public B
        {
        //body
        }
```

This declaration will form the different levels of inheritance.

Following program exhibits the multilevel inheritance.

```
        #include<iostream.h>
        #include<conio.h>
        classworker        // Base classdeclaration
        {
                int age;
                char name [20] ;
                public;
                void get( ) ;
                void show( ) ;
        }



        void worker: get ( )
        {
                cout << “your name please” ;
                cin >> name;
                cout << “your age please” ;
        }
        void worker : : show ( )
        {
                cout << “In my name is : “ <<name<< “ In my age is : “ <<age;
        }
        class manager : public worker //Intermediate base class derived
        {                    //publicly from the base class
                intnow;
                public:
                void get ( ) ;
                void show( ) ;
        };
        void manager :: get ( )
        {
                worker : :get ();         //calling get ( ) fn. of base class
                cout << “no. of workers under you:”;
                cin >> now;
        }
        void manager : : show ( )
        {
                worker : : show ();        //calling show ( ) fn. of base class
                cout << “In no. of workers under me are: “<<now;
        }
        class ceo:publicmanager         //declaration of derivedclass
        {                    //publicly inherited fromthe
                intnom;            //intermediate baseclass
                public:
                void get ( ) ;
                void show ( ) ;
        };
        void ceo : : get ( )
        {
                manager : : get ( ) ;
                cout << “no. of managers under you are:”; cin >> nom;
        }
        void manager : : show ( )

        {

                cout << “In the no. of managers under me are: In”;
                cout << “nom;
        }
```

```
main ( )
{
          clrscr ( ) ;
                    ceo cl ;
                    cl.get ( ) ; cout << “\n\n”;
                    cl.show ( ) ;
          }
```

Worker

Private:
int age;
char name[20];

Protected:

Private:
int age;
char name[20];

Manager:Worker

Private: int now;
Protected:
Public: void get() void show() worker::get() worker::get()

Ceo: Manager

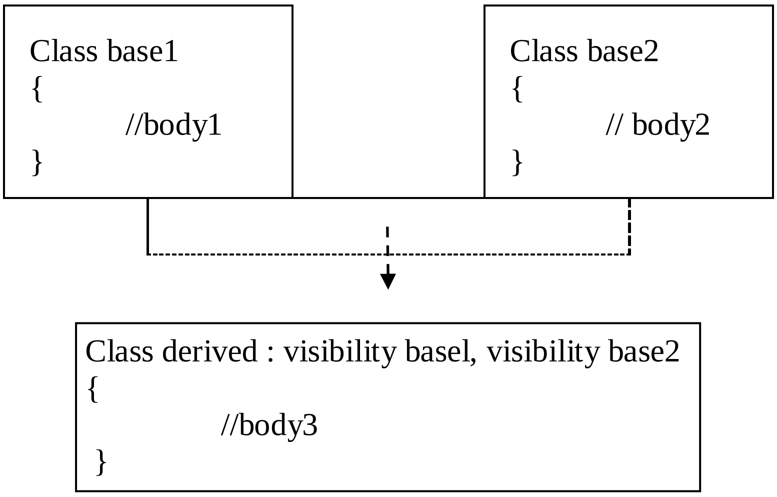
Public:
Protected:
Public:

All the inherited members

Multiple Inheritances

A class can inherit the attributes of two or more classes. This mechanism is known as 'MULTIPLE INHERITENCE'. Multiple inheritance allows us to combine the features

of several existing classes as a starring point for defining new classes. It is like the child inheriting the physical feature of one parent and the intelligence of another. The syntax of the derived class is asfollows:



```
Class base1
{
        //body1
}
```

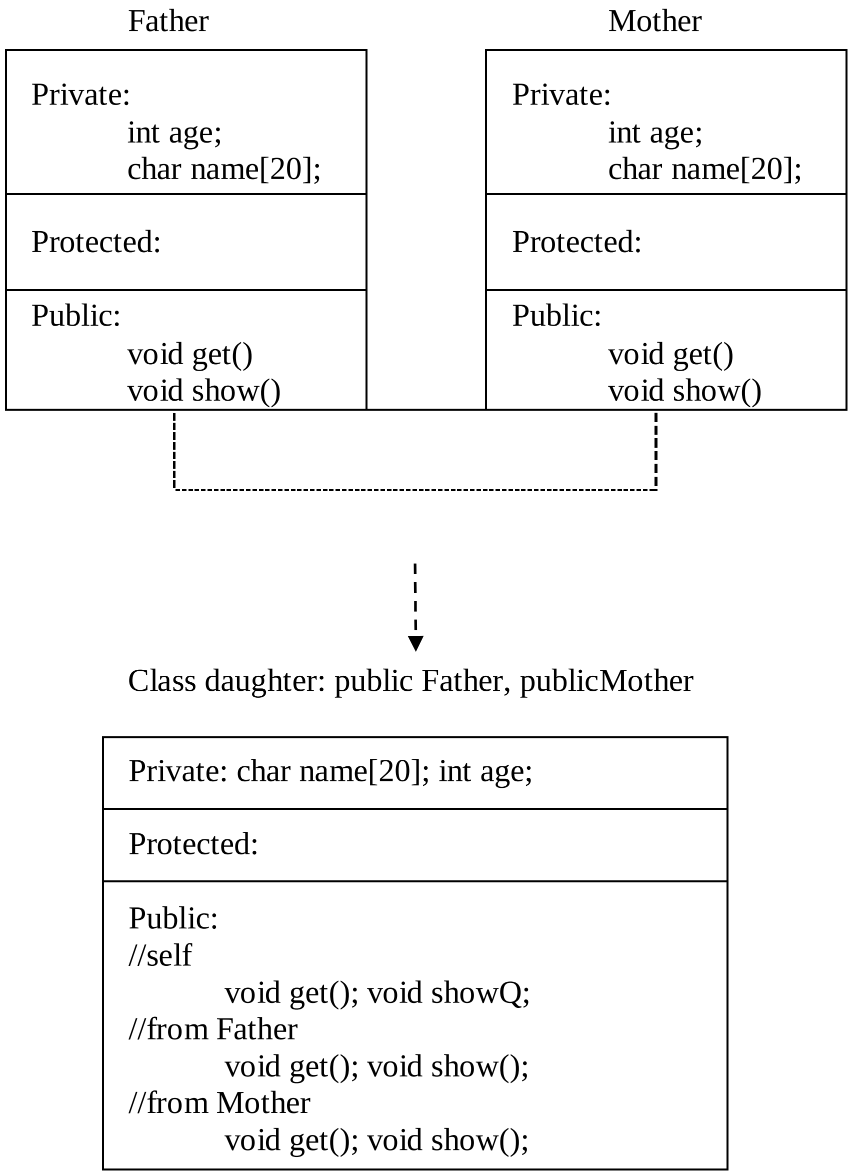
```
Class base2
{
        // body2
}
```

```
Class derived : visibility basel, visibility base2
{
            //body3
 }
```

Where the visibility refers to the access specifiers i.e. public, private or protected. Following program shows the multipleinheritance.

```
#include<iostream.h>
#include<conio . h>
classfather                 //Declaration of baseclassl
{
        int age ;
        char flame [20] ;
        public:
        void get ( ) ;
        void show ( ) ;
};
void father : : get ( )
{
        cout << "your father name please";
        cin >> name;
        cout << "Enter the age";
        cin >> age;
}
void father : : show ( )
{
cout<< "In my father's name is: ' <<name<< "In my father's age is:<<age;
}
classmother                     //Declaration of base class 2
{
        char name [20] ;
        int age ;
        public:
        void get ( )
        {
                cout << “mother’s name please” << “In”;
                cin >> name;
                cout << “mother’s age please” << “in”;
                cin >> age;
        }
void show ( )
{
        cout << “In my mother’s name is: “ <<name;
        cout << “In my mother’s age is: “ <<age;
        }
        class daughter : public father, public mother //derived class inheriting
        {                           //publicly
        char name[20];      //the features of both the baseclass
        intstd;
        public:
                void get ( ) ;
                void show ( ) ;
        };
        void daughter :: get ( )
        {
                father :: get ( ) ;
                mother :: get ( ) ;
                cout << “child's name: “;
                cin >> name;
                cout << “child's standard”;
                cin >> std;
        }
        void daughter :: show ()
        {
                father :: show ();
                nfather :: show ( ) ;
                cout << “In child’s name is : “ <<name;
                cout << “In child's standard: “ << std;
        }
        main ( )
        {
        clrscr ( ) ;
                daughter d1;
                d1.get ( ) ;
                d1.show ( ) ;
        }
```

Diagrammatic Representation of Multiple Inheritance is asfollows:



Father

Private:
int age;
char name[20];

Protected:

Public:
void get()
void show()

Mother

Private:
int age;
char name[20];

Protected:

Public:
void get()
void show()

Class daughter: public Father, publicMother

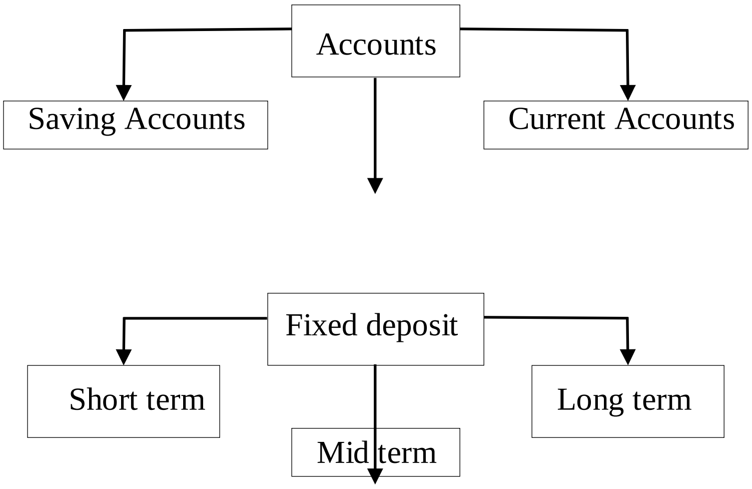
Private: char name[20]; int age;

Protected:

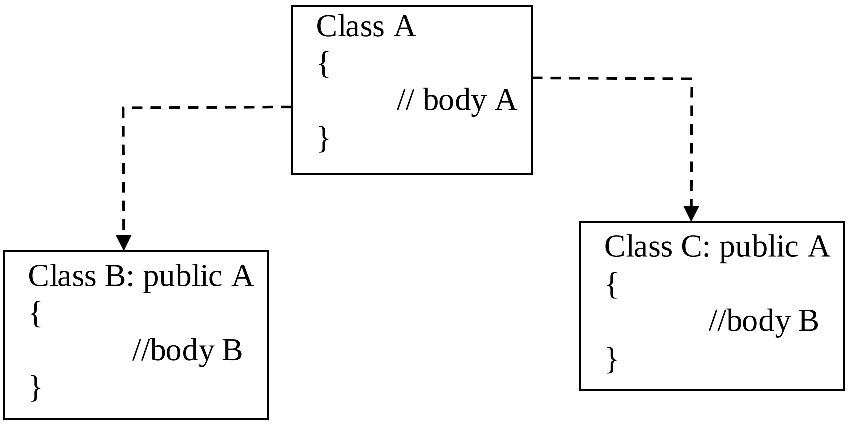
Public:
//self
void get(); void showQ;
//from Father
void get(); void show();
//from Mother
void get(); void show();

Hierarchical Inheritance

Another interesting application of inheritance is to use is as a support to a hierarchical design of a class program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level for e.g.



In general the syntax is given as



In C++, such problems can be easily converted into hierarchies. The base class will include all the features that are common to the subclasses. A sub-class can be constructed by inheriting the features of base class and so on.

```
// Program to show the hierarchical inheritance
#include<iostream.h>
# include<conio. h>
classfather                    //Base classdeclaration
{
        int age;
        char name [15];
        public:
        void get ( )
        {
                cout<< “father name please”; cin >> name;
                cout<< “father’s age please”; cin >> age;
        }
        void show ( )
        {
                cout << “In father’s name is ‘: “<<name;
                cout << “In father’s age is: “<< age;
        }
};
class son :publicfather            //derived class1
{
        char name [20] ;
        int age ;
        public;
        void get ( ) ;
        void show ( ) ;
} ;
void son : : get ( )




{
        father :: get ( ) ;
        cout << “your (son) name please” << “in”; cin >>name;
        cout << “your age please” << “ln”; cin>>age;
}
void son :: show ( )
{
        father : : show ( ) ;
        cout << “In my name is : “ <<name;
        cout << “In my age is : “ <<age;
}
class daughter :publicfather               //derived class2.
{
        char name [15] ;
        int age;
        public:
        void get ( )
        {
                father : : get ( ) ;
                cout << “your (daughter’s) name please In” cin>>name;
                cout << “your age please In”; cin >>age;
        }
        void show ( )
        {
                father : : show ( ) ;
                cout << “in my father name is: “ << name << “
                In and his age is : “<<age;
        }
};
main ( )
{
        clrscr ( ) ;
```

```
                son S1;
                daughter D1 ;
                S1. get ( );
                D1. get ( ) ;
                S1 .show( ) ;
                D1. show ( ) ;
        }
```

Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. Basically Hybrid Inheritance is the combination of one or more types of the inheritance. Here is one implementation of hybrid inheritance.

```
//Program to show the simple hybridinheritance
        #include<i sos t ream. h>
        #include<conio .h>
        classstudent                 //base classdeclaration
        {
                protected:
                int r_no;
                        public:
                                void get _n (int a)
                                {
                                        r_no =a;
                                        }
                                void put_n (void)
                                {
                                cout << "Roll No. : "<< r_no;
                                cout << "In";
                                }
                        };
                        class test : public student
                        {                //Intermediate baseclass
                        protected : int parti, par2;

                        public :
                                void get_m (int x, int y) {
                                        parti = x; part 2 = y; }
                                void put_m (void) {
                                        cout << "marks obtained: " << "In"
                                                << "Part 1 = " << part1 << "in"
                                                << "Part 2 = " << part2 << "In";
                                }
                        };
                        classsports                // base forresult
                        {
                        protected : int score;
                        public:
                                void get_s (int s){
                                        score = s }
                                void put_s (void){
                                        cout << " sports wt. : " << score << "\n\n";
                        }
                };
        class result : public test, public sports //Derived from test
                        &sports
        {
                int total;
                public:
                void display (void);
                };


        void result : : display (void)
        {
                total = part1 + part2 + score;
                put_n ( ) ;.
                put_m   (  );
                put_S    ( );
                cout << “Total score: “ <<total<< “\n”
}
main ( )
{
        clrscr ( ) ;
        result S1;
        S1.get_n (347) ;
        S1.get_m (30, 35);
        S1.get_s (7) ;
        S1.dciplay ( ) ;
}
```

Virtual Base Classes

We have just discussed a situation which would require the use of both multiple and multi level inheritance. Consider a situation, where all the three kinds of inheritance, namely multi-level, multiple and hierarchical are involved.

Let us say the 'child' has two direct base classes ‘parent1’ and ‘parent2’ which themselves has a common base class ‘grandparent’. The child inherits the traits of ‘grandparent’ via two separate paths. It can also be inherit directly as shown by the broken line. The grandparent is sometimes referred to as ‘INDIRECT BASE CLASS’. Now, the inheritance by the child might cause some problems. All the public and protected members of ‘grandparent’ are inherited into ‘child’ twice, first via ‘parent1’ and again via ‘parent2’. So, there occurs a duplicacy which should beavoided.

The duplication of the inherited members can be avoided by making common base class as the virtual base class: fore.g.

```
    classg_parent
    {
        //Body
    };
    class parent1: virtual public g_parent
    {
        // Body
    };


    class parent2: public virtual g_parent
    {
        // Body
    };
    class child : public parent1, public parent2
    {
        // body
    };
```

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between virtual base class and derived class. Note that keywords ‘virtual’ and ‘public’ can be used in eitherorder.

```
//Program to show the virtual base class
    #include<iostream.h>
    #include<conio . h>
    classstudent            // Base classdeclaration
    {
        protected:
        int r_no;
        public:
        void get_n (inta)
        { r_no = a; }
        void put_n(void)
        { cout << “Roll No. “ << r_no<< “ln”;}
        };
    class test : virtual public student // Virtually declaredcommon
    {                             //base class 1
        protected:
        int part1;
        int part2;
        public:
        void get_m (int x, int y)
        { part1= x; part2=y;}
            void putm (void)
            {
            cout << “marks obtained: “ << “\n”;
            cout << “part1 = “ << part1 << “\n”;
            cout << “part2 = “<< part2 << “\n”;
        }
    };
class sports : public virtual student // virtually declared common
{                             //base class 2
    protected:
    int score;
        public:
        void get_s (int a) {
        score = a ;
        }
        void put_s (void)
        { cout << “sports wt.: “ <<score<< “\n”;}
        };
        class result: public test,publicsports        //derivedclass
        {
            private : int total ;
            public:
                void show (void) ;
        };
        void result : : show (void)
        { total = part1 + part2 + score ;
            put_n ( );
            put_m ( );
            put_s ( ) ; cout << “\n total score= “ <<total<< “\n” ;
        }
        main ( )
        {
            clrscr ( ) ;
            result S1 ;
            S1.get_n (345)
            S1.get_m (30, 35) ;
            S1.get-S (7) ;
            S1. show ( );
            }

    //Program to show hybrid inheritance using virtual base classes
        #include<iostream.h>
        #include<conio.h>
        Class A
        {
```

```
        protected:
            int x;
        public:
            void get (int) ;
            void show (void) ;
        };
            void A : : get (int a)
                { x = a ; }
            void A : : show(void)
                { cout << X ;}
            Class A1 : Virtual PublicA
            {

protected:
    int y ;
public:
    void get (int) ;
    void show (void);
};
void A1 :: get (int a)
    { y = a;}
void A1 :: show (void)
{
cout <<y ;
{
class A2 : Virtual public A
{
protected:
    int z ;
public:
    void get (int a)
    { z =a;}
    void show (void)
    { cout << z;}
};
class A12 : public A1, public A2
{
int r, t ;
public:
    void get (int a)
    { r = a;}
    void show (void)
    { t = x + y + z + r ;
        cout << "result =" << t ;
    }
};
main ( )
{
clrscr ( ) ;
```

```
            A12 r ;
            r.A : : get (3) ;
            r.A1 : : get (4) ;
            r.A2 : : get (5) ;
            r.get (6) ;
            r . show ( ) ;
    }
```

Pure Virtual Functions

Generally a function is declared virtual inside a base class and we redefine it the derived classes. The function declared in the base class seldom performs any task.

The following program demonstrates how a pure virtual function is defined, declared and invoked from the object of a derived class through the pointer of the base class. In the example there are two classes employee and grade. The class employee is base class and the grade is derived class. The functions getdata ( ) and display ( ) are declared for both the classes. For the class employee the functions are defined with empty body or no code inside the function. The code is written for the grade class. The methods of the derived class are invoked by the pointer to the base class.

```
#include<iostream.h>

#include<conio.h>

class employee {

int code

char name [20] ;

public:

virtual void getdata ( ) ;

virtual void display ( ) ;

};
class grade: public employee
{
        char grd [90] ;
        float salary ;
public :
        void getdata ( ) ;
        void display ( );
};
void employee :: getdata ( )
{
}
void employee:: display ( )
{
}
void grade : : getdata ( )
{
        cout<< " enter employee's grade ";
        cin>> grd ;
        cout<< "\n enter the salary " ;
        cin>> salary;
}
void grade : : display ( )
{
        cout«" Grade salary \n";
        cout« grd« " "« salary« endl;
}
void main ( )
{
        employee *ptr ;
        grade obj ;
        ptr = &obj ;
        ptr->getdata ( ) ;
        ptr->display ( ) ;
        getche ( ) ;
}
```

```
Output
enter employee's grade A
enter the salary 250000
Grade salary
A       250000
```

Object Slicing:

In C++, a derived class object can be assigned to a base class object, but the other way is not possible.

```
classBase { intx, y; };

classDerived : publicBase { intz, w; }; intmain()
{
     Derived d;
     Base b = d; //ObjectSlicing,            z and w of d are slicedoff
}
```

Object Slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

```
#include
<iostream>usingnamespac
estd;

classBase
{
protected:
     inti;
public:
     Base(inta)          { i = a; }
     virtualvoiddisplay()
     { cout << "I am Base class object, i = " << i << endl;}
};

classDerived : publicBase
{
     intj;
public:
     Derived(inta, intb) : Base(a) { j = b; }
     virtualvoiddisplay()
     { cout << "I am Derived class object, i = "
              << i << ", j = " << j<<endl;            }
};
```

```
// Global method, Base class object is passed by value void
somefunc (Baseobj)
{
	obj.display();
}

intmain()
{
	Base b(33); Derived
	d(45, 54);
	somefunc(b);
	somefunc(d);	// Object Slicing, the member j of d is sliced off return0;
}
```

Output:

```
I am Base class object, i = 33 I am
Base class object, i = 45
```

We can avoid above unexpected behavior with the use of pointers or references. Object slicing doesn’t occur when pointers or references to objects are passed as function arguments since apointer or reference of any type takes same amount of memory. For example, if we change the global method myfunc() in the above program to following, object slicing doesn’thappen.

```
// rest of code is similar to above void
somefunc (Base&obj)
{
	obj.display();
}
// rest of code is similar to above
```

Output:

```
I am Base class object, i = 33
I am Derived class object, i = 45, j = 54
```

We get the same output if we use pointers and change the program to following.

```
// rest of code is similar to above void
somefunc (Base*objp)
{
	objp->display();
}
intmain()
{
	Base *bp = new Base(33);
	Derived *dp = new Derived(45,54); somefunc(bp);
	somefunc(dp);	// No Object Slicing
	return0;
}
```

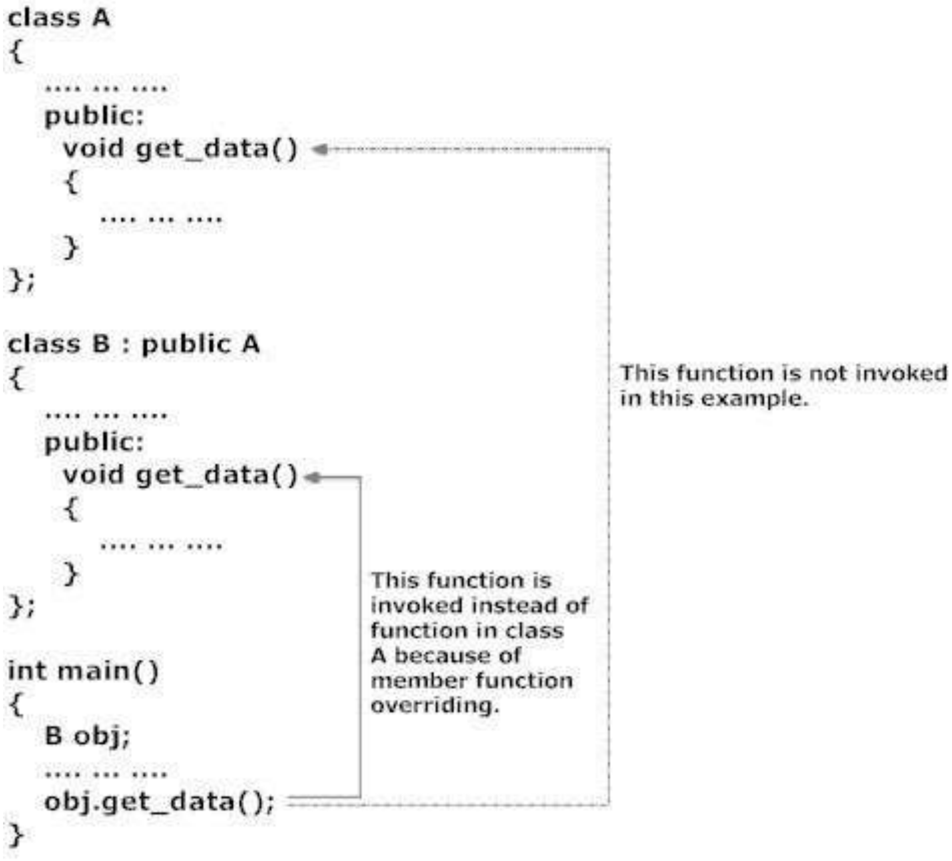
Output:

```
I am Base class object, i = 33
I am Derived class object, i = 45, j = 54
```

Object slicing can be prevented by making the base class function pure virtual there by disallowing object creation. It is not possible to create the object of a class which contains a pure virtual method.

C++ Function Overriding

If base class and derived class have member functions with same name and arguments. If you create an object of derived class and write code to access that member function then, the member function in derived class is only invoked, i.e., the member function of derived class overrides the member function of base class. This feature in C++ programming is known as function overriding.



```
class A
{
   .... ... ....
   public:
     void get_data()
     {
        .... ... ....
     }
};

class B : public A
{
   .... ... ....
   public:
     void get_data()
     {
        .... ... ....
     }
};

int main()
{
   B obj;
   .... ... ....
   obj.get_data();
}
```

This function is not invoked in this example.

This function is invoked instead of function in class A because of member function overriding.

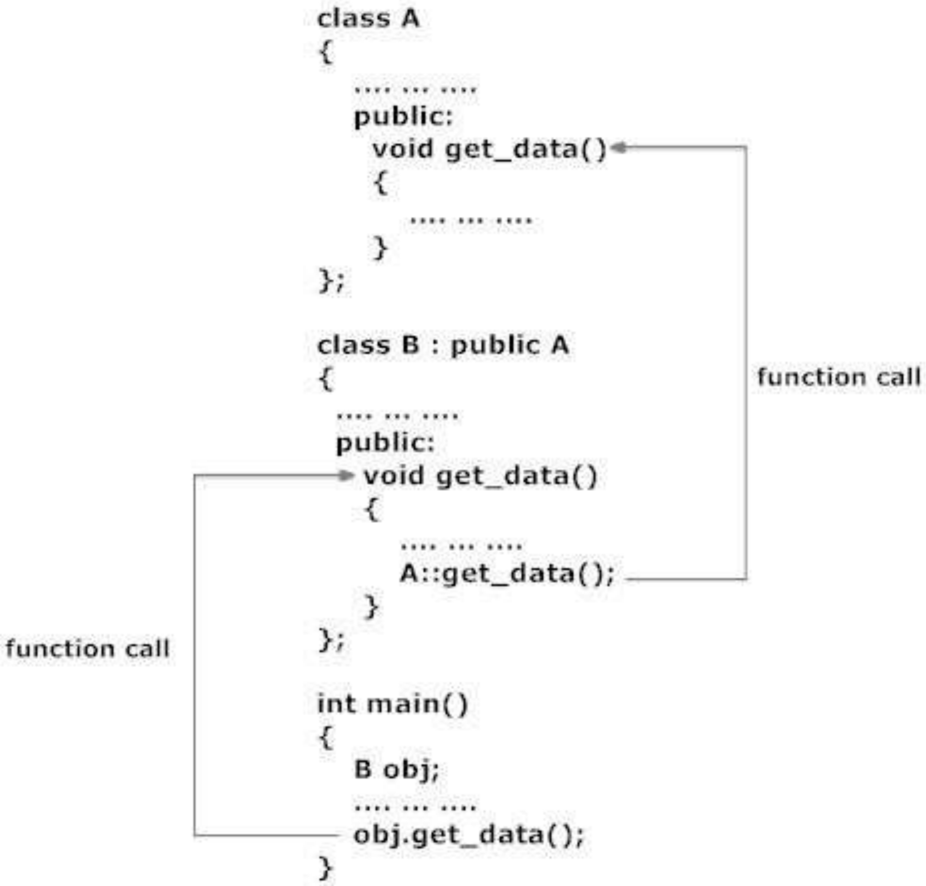
Figure: Member Function Overriding in C++

Accessing the Overridden Function in Base Class From Derived Class

To access the overridden function of base class from derived class, scope resolution operator **::**. For example: If you want to access get_data() function of base class from derived class in above example then, the following statement is used in derived class.

A::get_data; // Calling get_data() of class A.

It is because, if the name of class is not specified, the compiler thinks get_data() function is calling itself.



```
class A
{
   .... ... ....
   public:
    void get_data()
    {
       .... ... ....
    }
};

class B : public A
{
  .... ... ....
  public:
   void get_data()
   {
      .... ... ....
      A::get_data();
   }
};

int main()
{
   B obj;
   .... ... ....
   obj.get_data();
}
```

Abstract Class

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtualfunction.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can useits interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstracttoo.

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. Here is the syntax for a pure virtual function,

virtual void f() = 0;

Example of Abstract Class

classBase   //Abstract baseclass

```
{
 public:
 virtual void show()=0;              //Pure VirtualFunction
};

class Derived:public Base
{
 public:
 void show()
 { cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
 Baseobj;        //Compile TimeError
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Output : Implementation of Virtual Function in Derived class

In the above example Base class is abstract, with pure virtual **show()** function, hence we cannot create object of base class.

Why can't we create Object of Abstract Class ?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE(studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.

As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an errror message whenever you try to do so.