www.AllAbtEngg.com

UNIT II
INHERITANCE & POLYMORPHISM

Base Classes and Derived Classes – Protected Members – Casting Class pointers and Member Functions – Overriding – Public, Protected and Private Inheritance – Constructors and Destructors in derived Classes – Implicit Derived – Class Object To Base – Class Object Conversion – Composition Vs. Inheritance – Virtual functions – This Pointer – Abstract Base Classes and Concrete Classes – Virtual Destructors – Dynamic Binding.

2. Inheritance

2.1. Introduction

- Inheritance is the process by which new classes called *derived* classes are created from existing classes called *base* classes.
  - The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.
- For example, a programmer can create a *base* class named fruit and define *derived* classes as mango, orange, banana, etc.
  - Each of these derived classes, (mango, orange, banana, etc.) has all the features of the *base* class (fruit) with additional attributes or features specific to these newly created derived classes.
  - Mango would have its own defined features, orange would have its own defined features, banana would have its own defined features, etc. This concept of *Inheritance* leads to the concept of *polymorphism*.

Features or Advantages of Inheritance

1. ***Reusability:***

❖ Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked.

❖ Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

2. ***Saves Time and Effort:***

❖ The above concept of reusability achieved by inheritance saves the programmer time and effort, because the main code written can be reused in various situations as needed.

3. ***Increases Program Structure which results in greater reliability.***
4. ***Polymorphism***

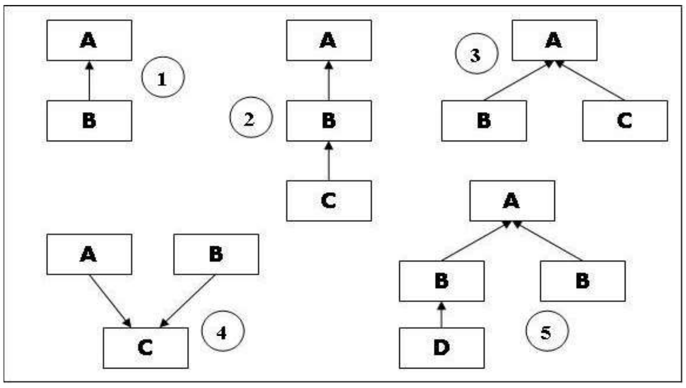
General Format for implementing the concept of Inheritance:

```
class derived-class-name:access-specifier base-class-name
{
      private   : variable declaration; //data member
                  Function declaration; //Member Fun.(Method)
      protected: variable declaration;
                  Function declaration;
      public    : variable declaration;
                  Function declaration;
};
```

Types of Inheritance

There are five different inheritances supported in C++:

1. **Simple / Single Inheritance** *:* There is only one Base Class and Only one Derived Class Means they have one to one Communication between them
2. **Multilevel Inheritance:** When a Derived Class again will be inherited by another Class then it creates a Multiple Levels.
3. **Hierarchical Inheritance :** When a Base Class is used or inherited by many Derived Classes.



4. **Multiple Inheritance :** When a Derived Class takes Features from two Base Classes.
5. **Hybrid Inheritance** *:* This is a Mixture of two or More Inheritance

1.1. Base Classes and Derived Classes

AllAbtEngg Android Application for Anna University, Polytechnic & School

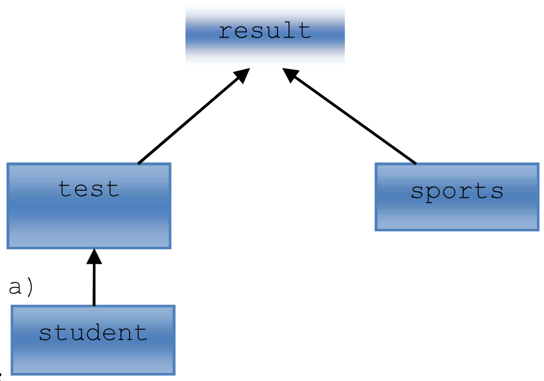
www.AllAbtEngg.com

- ✓ Derived-class object is an object of its base class, and one base class can have *many* derived classes, the set of objects represented by a base class typically is *l*arger than the set of objects represented by any of its derived classes.
- ✓ Inheritance relationships form class hierarchies. A base class exists in a hierarchical relationship with its derived classes.
- ✓ A base class's *public* members are accessible within its body and anywhere that the program has a handle (i.e., a name, reference or pointer) to an object of that class or one of its derived classes.
- ✓ A base class's *private* members are accessible only within its body and to the friends of that base class.

2.2. Protected Members

- A base class's protected members can be accessed within the body of that base class.
- Derived-class member functions can refer to public and protected members of the base class simply by using the member names.
- When a derived-class member function *redefines* a base-class member function, the base-class member can still be accessed from the derived class by preceding the base-class member name with the base-class name and the scope resolution operator (::).

Fig.2.2. Inheritance hierarchy for Student result



```
#include<iostream.h>
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";
                }
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
};
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
            obtined:"<<"\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";
            }
};
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
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;
}
```

Output

```
ROLL NO:1234
marks obtined:
part1=27.5
part2=33
sports wt:6
total score:66.5
```

2.3. Casting Class pointers and Member Functions

The class pointer can be cast by the base class or by the derived class.

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

2.3.1. Class Object To Base (Upcasting)

- **Upcasting** is converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as though it were its base type.
- It is always allowed for **public** inheritance, without an explicit type cast. This is a result of the **is-a** relationship between the base and derived classes.
- Here is the code dealing with shapes. We created **Shape** class, and derived **Circle**, **Square**, and **Triangle** classes from the **Shape** class. Then, we made a member function that talks to the base class:

```
void play(Shape& s)
{
        s.draw();
        s.move();
        s.shrink();
        ....
}
```

- The function speaks to any **Shape**, so it is independent of the specific type of object that it's drawing, moving, and shrinking. If in some other part of the program we use the **play()** function like below:

```
Circle c;
Triangle t;
Square sq;
play(c);
play(t);
play(sq);
```

- We will check. A **Triangle** is being passed into a function that is expecting **Shape**. Since a **Triangle** is a **Shape**, it can be treated as one by **play()**. That is, any message that **play()** can send to a **Shape** a **Triangle** can accept.
- **Upcasting** allows us to treat a derived type as though it were its base type.
- The most important aspect of inheritance is not that it provides member functions for the new class, however. It's the **relationship** expressed between the new class and the base class. This relationship can be summarized by saying, "The new class is a type of the existing class."

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
class Parent
{
   public:
   void sleep() {}
};
class Child: public Parent
{
   public:
   void gotoSchool(){}
};
int main( )
{
        Parent parent;
        Child child;// upcast - implicit type cast allowed
        Parent *pParent = &child;
        // downcast - explicit type case required
        Child *pChild=(Child *) &parent;
        pParent -> sleep();
        pChild -> gotoSchool();
   return 0;
}
```

A **Child** object is a **Parent** object in that it inherits all the data members and member functions of a **Parent** object.

Upcasting is **transitive**: if we derive a **Child** class from **Parent**, then **Parent** pointer (reference) can refer to a **Parent** or a **Child** object.

Upcasting can cause **object slicing** when a derived class object is passed by value as a base class object, as in **foo(Base derived_obj)**.

Because **implicit upcasting** makes it possible for a base-class pointer (reference) to refer to a base-class object or a derived-class object, there is the need for **dynamic binding**. That's why we have **virtual** member functions.

- **Pointer (Reference)** type: known at **compile** time.
- **Object** type: not known until **run** time.

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
#include<iostream.h>
class Mother
{
        public:
                void cooks()
                {
                        cout<<"Mother cooks food";
                }
};
class child:public mother
{
        public:
                void studies()
                {
                        cout<<"Child studies";
                }
};
int main()
{
        child chl;
        Mother *pMother=&chl;
        pMother->cooks();
        return 0;
}
```

2.3.2. Downcasting

- The opposite process, converting a base-class pointer (reference) to a derived-class pointer (reference) is called **downcasting**.
- Downcasting is not allowed without an explicit type cast. The reason for this restriction is that the **is-a** relationship is not, in most of the cases, symmetric.
- A derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

- As in the example, we derived **Child** class from a **Parent** class, adding a member function, **gotoSchool()**. It wouldn't make sense to apply the **gotoSchool()** method to a **Parent** object. However, if implicit downcasting were allowed, we could accidentally assign the address of a **Parent** object to a pointer-to-Child.

  `Child *pChild =  &parent; // actually this won't compile`

  `error: cannot convert from 'Parent *' to 'Child *'` and use the pointer to invoke the **gotoSchool()** method as in the following line.

  `pChild -> gotoSchool();`
- Because a **Parent** isn't a **Child** (a **Parent** need not have a **gotoSchool()** method), the downcasting in the above line can lead to an **unsafe** operation.
- Downcasting is the opposite of the basic object-oriented rule, which states objects of a derived class, can always be assigned to variables of a base class.

```
#include<iostream.h>
class Mother
{
    public:
            void cooks()
            {
                    cout<<"Mother cooks food";
            }
};
class child1:public mother
{
    public:
            void studies()
            {
                    cout<<"Child1 studies";
            }
};
class child2:public mother
{
    public:
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
                void studies()
                {
                        cout<<"Child2 studies";
                }
};
int main()
{
        Mother *pMother=new child1;
        child1 *pcld1= (child1*)pMother;
        child2 *pcld2= (child2*)pMother;
        cout<<"Using object of Mother class";
        pMother->cooks;
        cout<<"Using object of Child1 class";
        pcld1->studies();
        cout<<"Using object of Child1 class";
        pcld2->studies();
        return 0;
 }
```

2.4. Overriding

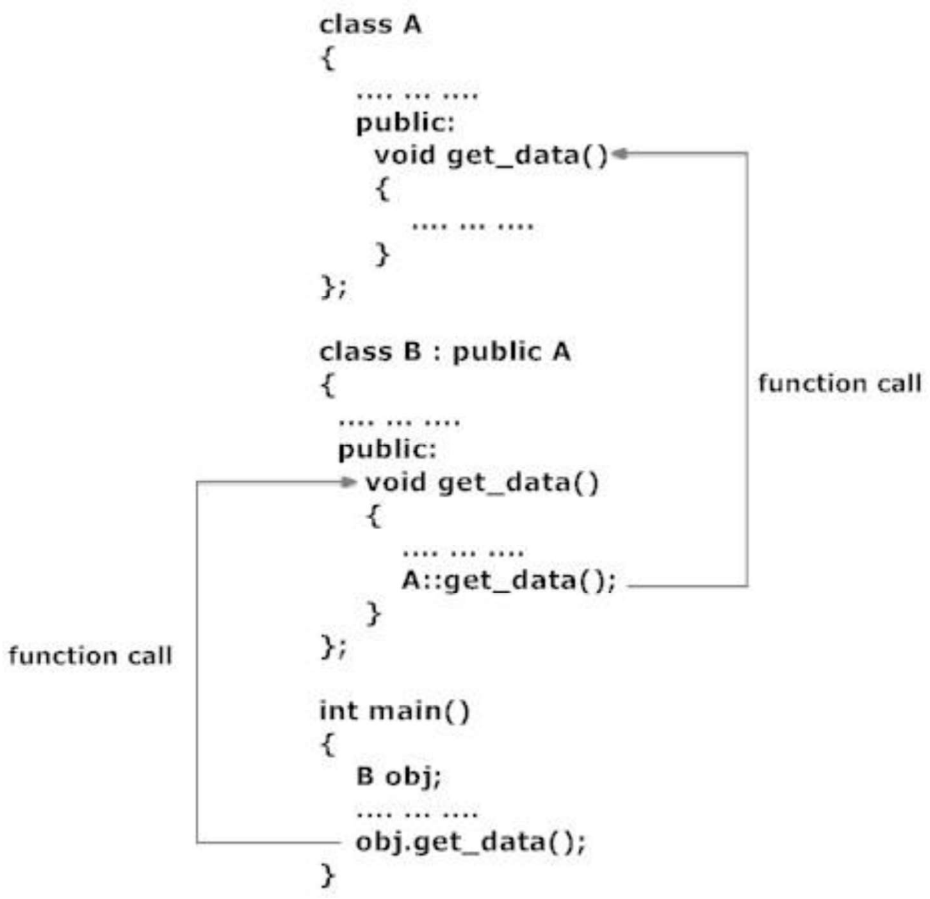
If base class and derived class have member functions with same name and arguments. If you create an object of derived class and write code to access that member function then, the member function in derived class is only invoked, i.e., the member function of derived class overrides the member function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

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
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com



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

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

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

```
#include <iostream>
using namespace std;
class TRectangle
{
      public:
          TRectangle(double l, double w) : length(l), width(w)
      {}
          virtual void print() const;

      private:
          double length;
          double width;
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
};
void TRectangle::print() const
{
   // print() method of base class.
cout<< "Length = "<< this->length<< "; Width = " << this->width;
}
class TBox : public TRectangle
{
      public:
TBox(double l, double w, double h) : TRectangle(l, w), height(h)
{}    // virtual is optional here, but it is a good practice to
remind it to the developer.
          virtual void print() const;
      private:
          double height;
};
// print() method of derived class.
void TBox::print() const
{
         // Invoke parent print() method.
         TRectangle::print();
         std::cout << "; Height= " << this->height;
}
int main(int argc, char** argv)
{
         TRectangle rectangle(5.0, 3.0);
          // Outputs: Length = 5.0; Width = 3.0
         rectangle.print();
         TBox box(6.0, 5.0, 4.0);
         // The pointer to the most overridden method in the
      vtable in
           on TBox::print,
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
        // but this call does not illustrate overriding.
        box.print();
        // This call illustrate overriding.
        // outputs: Length = 6.0; Width = 5.0; Height= 4.0
        static_cast<TRectangle&>(box).print();
}
```

2.5. Public, Protected and Private Inheritance

- When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance.
- Figure 2.5 summarizes for each type of inheritance the accessibility of base-class members in a derived class.
  - The first column contains the base-class access specifiers. When deriving a class from a public base class, public members of the base class become public members of the derived class, and protected members of the base class become protected members of the derived class.
  - A base class's private members are *never* accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.
  - When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.
  - When deriving from a private base class, public and protected members of the base class become private members (e.g., the functions become utility functions) of the derived class. Private and protected inheritance are not *is-a* relationships.

Base-class Member access specifier	**Type of inheritance**		
	Public inheritance	**protected inheritance**	**private inheritance**
Public	*public* in derived class. Can be accessed directly by member functions, *friend* functions and nonmember functions.	*protected* in derived class. Can be accessed directly by member functions and *friend* functions.	*private* in derived class. Can be accessed directly by member functions and *friend* functions

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

protected	*protected* in derived class. Can be accessed directly by member functions and *friend* functions.	*protected* in derived class. Can be accessed directly by member functions and *friend* functions.	*private* in derived class. Can be accessed directly by member functions and *friend* functions.
private	Hidden in derived class. Can be accessed by member functions and *friend* functions through *public* or *protected* member functions of the base class.	Hidden in *derived* class. Can be accessed by member functions and *friend* functions through *public* or *protected* member functions of the base class.	Hidden in derived class. Can be accessed by member functions and *friend* functions through *public* or *protected* member functions of the base class.

Fig 2.5. Base-class member accessibility in a derived class.

2.6. Constructors and Destructors in derived Classes

- If the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in this chain is the one of the class at the base of the hierarchy, whose body actually finishes executing *first*. The original derived-class constructor's body finishes executing *last*.
- Each base-class constructor initializes the base-class data members that the derived-class object inherits. In the Commission-
- When a derived-class object is destroyed, the program calls that object's destructor. This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes' members execute in *reverse* of the order in which the constructors executed.
- When a derived-class object's destructor is called, the destructor performs its task, then invokes the destructor of the next base class up the hierarchy. This process repeats until the destructor of the final base class at the top of the hierarchy is called. Then the object is removed from memory.

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

- Base-class constructors, destructors and overloaded assignment operators are *not* inherited by derived classes. Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class versions.
- If any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to pass arguments to the base class constructors.
- The general format of a derived class is:

```
derived-constructor(arglist1,arglist2,…arglistN,arglist(D):
base1(arglist1), base2(arglist2),…bas2(arglistN)
{ //body of 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
{
};
	A(); derived constructor
	B(); base constructo1
	C(); base constructor2

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

```
Example
#include <iostream.h>
#include <string>
using namespace std;
class User
{                                // BASE
     string name;
     int age;
     public:
         User( string nm, int a )
         {
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
                    name = nm;
                    age = a;
                }
                void print()
                {
                    cout<<"Name:"<<name<<"Age:"<<age;
                }
};
class StudentUser : public User
{          // DERIVED
        string schoolEnrolled;
        public:
            StudentUser(string  nam,int  y,string  school):User(
nam, y )
            {                        //(A)
                schoolEnrolled = school;
            }
             void print()
             {
                User::print();
              cout<<"School Enrolled:"
                   <<schoolEnrolled;
                }
};
int main()
{
        StudentUser student( "Maura", 20, "ece" );
        student.print();
        return 0;
}
```

2.1.1. Initializing the class objects

General format is,

```
Constructor(arglist):initialization-section
{                //assignment section
}
```

www.AllAbtEngg.com

Example

```
#include<iostream.h>
class namespace
{
	int x;
	public:
		alpha(int i)
		{
			x=i;
			cout<<"alpha constructed";
		}
		void show_alpha()
		{
			cout<<"x="<<x;
		}
};
class beta
{
	float p,q;
	public:
		beta(float a,float b):p(a),q(b+p)
		{
			cout<<"\nBeta constructed";
		}
		void show_beta()
		{
			cout<<"p="<<p<<"q="<<q;
		}
};
class gamma:public beta,public alpha
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
{
      int u,v;
      public:
            gamma(int a,int b,float c):alpha(a*2),beta(c,c),
      u(a)
            {
                  v=b;
                  cout<<"\nBeta constructed";
            }
            void show_gamma()
            {
                  cout<<"u="<<u<<"v="<<v;
            }
};
int main()
{
      gamma g(2,4,2.5);
      cout<<"\nDisplay member values\n\n"
      g.show_alpha();
      g.show_bea();
      g.show_gamma();

}
```

```
Output
Beta constructedalpha constructed
Beta constructed
Display member values
x=4 p=2.5 q=5 u=2 v=4
```

2.1.2. Member Classes:Nesting of Classes

An object can be a collection of many other objects. Example,

```
class alpha{….}
class beta{…..}
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
class gamm
{
      alpha a;
      beta b;………
};
```

2.2. Composition Vs. Inheritance

- The *is-a* relationship represents inheritance. In an *is-a* relationship, an object of a derived class also can
  be treated as an object of its base class—for example, a Car *is a* Vehicle, so any attributes and behaviors of a Vehicle are also attributes and behaviors of a Car.
- The *has-a* relationship represents *composition*. In a *has-a* relationship, an object *contains* one or more objects of other classes as members. For example, a Car has many components—it *has a* steering wheel, *has a* brake pedal, *has a* transmission, etc.

2.3. Virtual functions

A **virtual function** is a special type of function that resolves to the most-derived version of the function with the same signature. To make a function virtual, simply place the "virtual" keyword before the function declaration.

```
#include<iostream.h>
#include<conio.h>
class  base
{
         private:
                 int x;
                 float y;
         public:
                 virtual void getdata( );
                 virtual void display( );
};
class dev:public base
{
         private:
                 int roll;
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
                char name[20];
        public:
                void getdata( );
                void  display( );
};
void base::getdata( ){}
void base::display( ){}
void dev::getdata( )
{
        cout<<" Enter Roll of  the Student ";
        cin>> roll;
        cout<<"Enter name of  the student";
        cin>>name;
}
void dev::display()
{
        cout<<"Name is :"<<name<<endl;
        cout<<" Roll no is :"<<roll <<endl;
}
void main()
{
        base * ptr;
        dev obj;
        clrscr();
        ptr = &obj;
        ptr -> getdata( );
        ptr -> display( );
        getch( );
}
Output
Enter Roll of the Student 12
Enter name of the student XXX
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
        Name is : XXX
        Roll no is : 12
```

2.3.1. Pure virtual function

A virtual function body is known as Pure Virtual Function. In above example we can see that the function is base class never gets invoked. In such type of situations we can use pure virtual functions

```
class AB
{
        public:virtual void f()= 0;
};
```

```
Example
#include<iostream.h>
class base
{
        public:
        virtual void show()=0; //pure virtual function
};
class derived1 : public base
{
        public:
                void show()
                {
                        cout<<"\n Derived 1";
                }
};
class derived2 : public base
{
        public:
                void show()
                {
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
                    cout<<"\n Derived 2";

            }
};
void main()
{
      base *b; derived1 d1; derived2 d2;
      b = &d1;
      b->show();
      b = &d2;
      b->show();
}
```

Output

```
Derived 1
Derived 2
```

Rules for Virtual Functions

1. The virtual function must be member of class
2. They cannot be static members
3. They are accessed by using object pointers
4. Prototype of base class function & derived class must be same
5. Virtual function in base class must be defined even though it is not used
6. A virtual function can be friend function of another class
7. We could not have virtual constructor
8. If a virtual function is derived in base class, it need not be necessarily redefined in the derived class
9. Pointer object of base class can point to any object of derived class but reverse is not true
10. When a base pointer points to derived class, incrementing & decrementing it will not make it point to the next object of derived class

```
#include <iostream>
using namespace std;
// Base class
class Shape
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
 {
     public:
        // pure virtual function providing interface framework.
        virtual int getArea() = 0;
        void setWidth(int w)
        {
           width = w;
        }
        void setHeight(int h)
        {
           height = h;
        }
     protected:
        int width;
        int height;
};
// Derived classes
class Rectangle: public Shape
{
     public:
        int getArea()
        {
           return (width * height);
        }
};
class Triangle: public Shape
{
     public:
        int getArea()
        {
           return (width * height)/2;
        }
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
 };
 int main(void)
 {
         Rectangle Rect;
         Triangle  Tri;
         Rect.setWidth(5);
         Rect.setHeight(7);
         // Print the area of the object.
         cout<<"Total Rectangle area:"<<Rect.getArea()<<endl;
         Tri.setWidth(5);
         Tri.setHeight(7);
         // Print the area of the object.
         cout<<"Total Triangle area:"<<Tri.getArea()<<endl;
         return 0;
}
```

Output `Total Rectangle area: 35    Total Triangle area: 17`

2.4. This Pointer

To represent an object that invokes a member function. *this* pointer that points to the object for which this function was called.

Example:

```
      A=123; or this->a=123
      #include<iostream.h>
      #include<string.h>
      class per
      {
         char name[20];
         float saralry;
         public :
            per (char *s,float a)
            {
                  strcpy(name,s);
                  salary ='a'
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
		}
		per GR(per &x)
		{
			if (x.salary> =salary)
				return &x;
			else
				return *this;
		}
		void display()
		{
			cout<<"name : "<<name<<'\n';
			cout<<"salary :"<<salary<<'\n';
		}
	};
	void main()
	{
		per
		p1("REEMA:,10000),p2("KRISHANAN",20000),p3("GEORGE",
		50000);
		per p=p1.GR(p2);
		p.display();
		p=p1.GR(p3);
		p.display();
	}
	output
	name : REEMA
	salary : 10000
	name : KRISHANAN
	salary : 20000
```

2.5. Abstract Base Classes and Concrete Classes

- In C++ an *abstract class* is one which defines an interface, but does not necessarily provide implementations for all its member functions. An abstract class is meant to be used

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

as the base class from which other classes are derived. The derived class is expected to provide implementations for the member functions that are not implemented in the base class.

- A derived class that implements all the missing functionality is called a *concrete class* .
- A virtual member function for which no implementation is given is called a *pure virtual function* . If a C++ class contains a pure virtual function, it is an *abstract class*. In C++ it is not possible to instantiate an abstract class.

Example

```
#include <iostream.h>
using namespace std;
class CPolygon
{
        protected:
                int width, height;
        public:
                void setup (int first, int second)
                {
                        width= first;
                        height= second;
                }
                virtual int area() = 0;
};
class CRectangle: public CPolygon
{
        public:
                int area(void)
                {
                        return (width * height);
                }
};
class CTriangle: public CPolygon
{
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
        public:
                int area(void)
                {
                        return (width * height/2);
                }
};
int main ()
{
        CRectangle rectangle;
        CTriangle triangle;
        CPolygon * ptr_polygon1 = &rectangle;
        CPolygon * ptr_polygon2 = &triangle;
        ptr_polygon1->setup(2,2);
        ptr_polygon2->setup(2,2);
        cout << ptr_polygon1->area () << endl;
        cout << ptr_polygon2->area () << endl;
        return 0;
}
```

Output

```
4
2
```

Difference between Abstract class and Concrete class

Abstract class	Concrete class
Abstract class is a class that is declared with the keyword abstract	Only the keyword class is used for declaration of concrete class
The Abstract class cannot be initiated. But we can inherit the Abstract class	The Concrete class can be initiated as well as inherited.
At least one method must be abstract in abstract class	No method should be abstract in concrete class

2.6. Virtual Destructors

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

- If the destructor in the base class is not made virtual, then an object that might have been declared of type base class and instance of child class would simply call the base class destructor without calling the derived class destructor.
- Hence, by making the destructor in the base class virtual, we ensure that the derived class destructor gets called before the base class destructor.

```
class a
{
          public:
          a(){printf("\nBase Constructor\n");}
          ~a(){printf("\nBase Destructor\n");}
};
class b : public a
{
        public:
        b(){printf("\nDerived Constructor\n");}
        ~b(){printf("\nDerived Destructor\n");}
};
int main()
{
        a* obj=new b;
        delete obj;
        return 0;
}
Output:
Base Constructor
Derived Constructor
Base Destructor
```

By Changing

```
~a()
{
   printf("\nBase Destructor\n");
}
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
 to
virtual ~a()
{
   printf("\nBase Destructor\n");
}
```

Output

```
Base Constructor
Derived Constructor
Derived Destructor
Base Destructor
```

2.9. Dynamic Binding.

Dynamic binding means the JVM will decide at runtime which method implementation to invoke based on the class of the object.

The compiler should match function calls with the correct definition at the run time.

Dynamic binding is achieved using virtual function

Static Binding Vs Dynamic Binding

Static Binding	Dynamic Binding
Static binding happens at compile time	Dynamic binding happens at run time
Static binding is also called as early binding	Dynamic binding is also called as late binding
There is no use of virtual in this binding	The virtual is used in Dynamic binding
It is more efficient that the late binding as extra level of indirection is involved in late binding	It is flexible

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

Syntax

1. Main Function

```
int main()
{
    cout<<"Welcome to C++";
    return 0;
}
```

2. Declaring variable names

```
data-type variable-name;
```

3. Reference Variables

```
data-type &reference-name=variable-name;
```

4. if ….else statement

```
if(condition)
    statement;
else
    statement;
```

5. The switch statement

```
switch(expression)
{
    case constant1: group of statements 1; break;
    case constant2: group of statements 2; break;...
    default: default group of statements
}
```

6. do{}while Loop

```
do
{
}while(condition is true);
```

7. while loop

```
while(expression){}
```

8. for loop

```
for(initialization;condition;increase or decrease)
```

9. function definition

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

```
type name_of_the_function(argument list)
{
    //body of the function
}
```

10. inline function

```
inline function_header
{
        //body of the function
}
```

11. friend function

```
friend return-type function-name(class-name object-name);
```

12. class definition

```
class name_of_class
{
  private  : variable declaration; //data member
                  Function declaration; //Member Fun.(Method)
    protected: variable declaration;
             Function declaration;
    public   : variable declaration;
                  Function declaration;
};
```

13. Outside class definition using scope resolution operator(::)

```
Name_of_the_class::function_name
```

14. Member function definition outside the class definition

```
return_type name_of_the_class::function_name(argument list)
{
        //body of function
}
```

15. Declaration of Objects as Instances of a Class

```
class-name object-name;
```

16. Static Data Member

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

static int count;

17. Constructor

```
class name()
{
        argumentlist
}
```

18. Copy constructors

class-Name e3(e2); or class-Name e3=e2;

19. **Destructors**

~classname();

20. **operator overloading**

```
return_type classname::operator operator-symbol(argument)
{
        …………..
        statements;
}
```

21. **conversion function**

```
operator typename()
{
        ....... //statements
}
```

22. **Inheritance**

```
class derived-class-name:access-specifier base-class-name
{
   private   : variable declaration; //data member
               Function declaration; //Member Fun.(Method)
   protected: variable declaration;
               Function declaration;
   public    : variable declaration;
                 Function declaration;
};
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

23. **virtual base class**

```
class A
{
        ……….   //grand parent
};
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)
};
```

24. **Initializing the class objects**

```
constructor(arglist):initialization-section
{
        //assignment section
}
```

25. ***Initializing Pointers***

```
data-type *pointer-variable = value;
```

26. **Pointers to Functions and objects**

```
data_type(*function_name)();
```

27. **Pure virtual function**

```
class AB
{
        public:
                virtual void f()= 0;
};
```

AllAbtEngg Android Application for Anna University, Polytechnic & School

www.AllAbtEngg.com

28. **catches exceptions**

```
catch(type arg)
{
        …………… // Block of statements that handles the exceptions
}
…………
```

29. ***Multiple Catch Statements***

```
try
{
        //try block
}
catch(type1 arg)
{
        //catch block1
}
catch(type2 arg)
{
        //catch block2
}
…………….
catch(typeN arg)
{
        //catch blockN
}
```

30. **Catch All Exceptions**

```
catch(. . .)
{
        //statement for processing all exceptions
}
throw;
```

AllAbtEngg Android Application for Anna University, Polytechnic & School