Dayananda Sagar University Bengaluru

Object Oriented Programming – 16CS305

Department of Computer Science and Engineering, Dayananda Sagar University, Bengaluru

Dayananda Sagar University Bengaluru

Module - 03

Inheritance – Types - Derived Class Constructors- Issues in Inheritance – Virtual base Class – Polymorphism – Virtual functions – Pure virtual functions

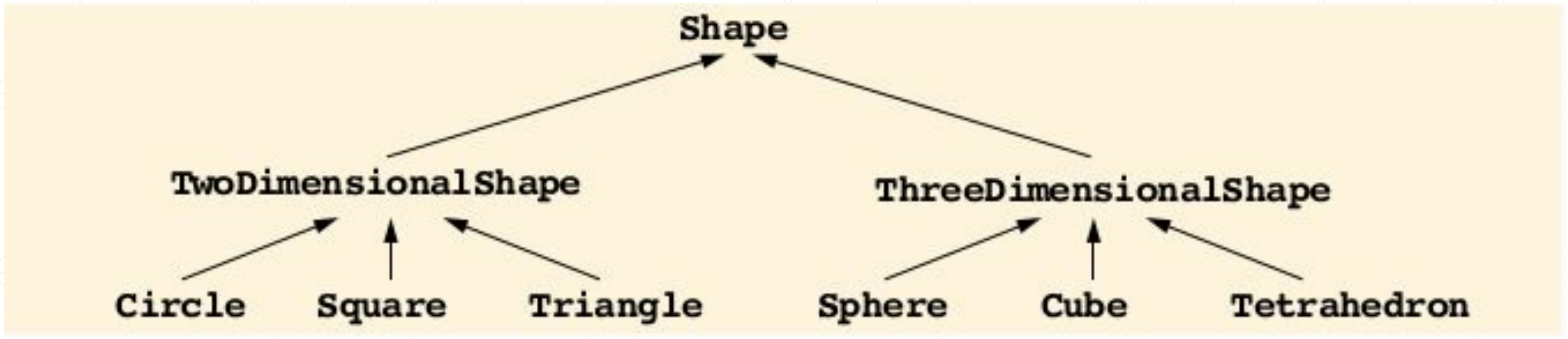
Topics

- Inheritance Basics
- Types of Inheritance

Inheritance

- One of the most powerful features of C++ is the use of inheritance to derive one class from another.
- Inheritance is the process by which a new class—known as a derived class—is created from another class, called the base class.
- A derived class automatically has all the member variables and functions that the base class has, and can have additional member functions and/or additional member variables.
- To inherit from a class, use the **:** symbol.
    - derived class (child) - the class that inherits from another class
    - base class (parent) - the class being inherited from
- Inheritance in C++ offers the feature of code reusability. We can use the same fragment of code multiple times. To sum it up, inheritance helps us save our development time, maintain data in a simplified manner and gives us the provision to make our code extensible.

Example of Inheritance



Base and Derived Classes

- A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form.
- Syntax: class derived-class: access-specifier base-class
- Where access-specifier is one of **public, protected**, or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is **private** by default.
- Consider a base class **Shape** and its derived class **Rectangle** as follows in example 1−

```
#include <iostream>

using namespace std;
// Base class
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};
// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};
```

```
int main(void) {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);
   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() <<
endl;
   return 0;
}
```

Output:

```
Total area: 35
```

Access Control and Inheritance

- A derived class can access all the non-private members of its base class.
- Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.
- It can be summarized as the different access types according to - who can access them in the following way −

Access	Public	protected	private
Same Class	yes	yes	yes
Derived Class	yes	yes	no
Outside Class	yes	no	no

Access Control and Inheritance

- Here’s a table of all of the access specifier and inheritance types combinations:

Access specifier in base class	Access specifier when inherited publicly	Access specifier when inherited privately	Access specifier when inherited protectedly
Public	Public	Private	Protected
Protected	Protected	Private	Protected
Private	Inaccessible	Inaccessible	Inaccessible

- A derived class inherits all base class methods with the following exceptions −
  - Constructors, destructors and copy constructors of the base class.
  - Overloaded operators of the base class.
  - The friend functions of the base class.

- WHAT IS INHERITANCE ACCESS CONTROL?
- When creating a derived class from a base class then, you can use different access specifiers to inherit the data members of the base class.
- The derived class can access all the non-private members of its base class. And the base class members that are not accessible to the member functions of derived classes should be declared private in the base class.
- The access specifiers that are used are public, private and protected.
- When deriving a class from a base class, the base class may be inherited through public, private and protected inheritance.

LET US HAVE A BRIEF DESCRIPTION ABOUT THESE SPECIFIERS:

PUBLIC INHERITANCE:

- When inheriting a class from a public parent class, public members of the parent class become public members of the child class and protected members of the parent class become protected members of the child class.
- The parent class private members cannot be accessible directly from a child class but can be accessible through public and protected members of the parent class.

PRIVATE INHERITANCE:

- When we derive from a private parent class, then public and protected members of the parent class become private members of the child class.

PROTECTED INHERITANCE:

- When deriving from a protected parent class, then public and protected members of the parent class become protected members of the child class.

```
class X
 {
 public:
     int a;

protected:

    int b;

private:

    int c;

};

class Y : public X

{

    // a is public

    // b is protected

    // c is not accessible from Y
```

Types of Inheritance in C++

- There are basically 5 types of inheritance in C++. The classification of inheritance is based on how the properties of the base class are inherited by the derived class(es).

1) Single Inheritance

- This type of inheritance in C++ happens when the parent class has only one child class. In other words, this is only one derived class formed from a base class.

2) Multiple Inheritance

- This type of inheritance happens when the child class inherits its properties from more than one base class. In other others, the derived class inherits properties from multiple base classes.

Types of Inheritance in C++

3) Hierarchical Inheritance

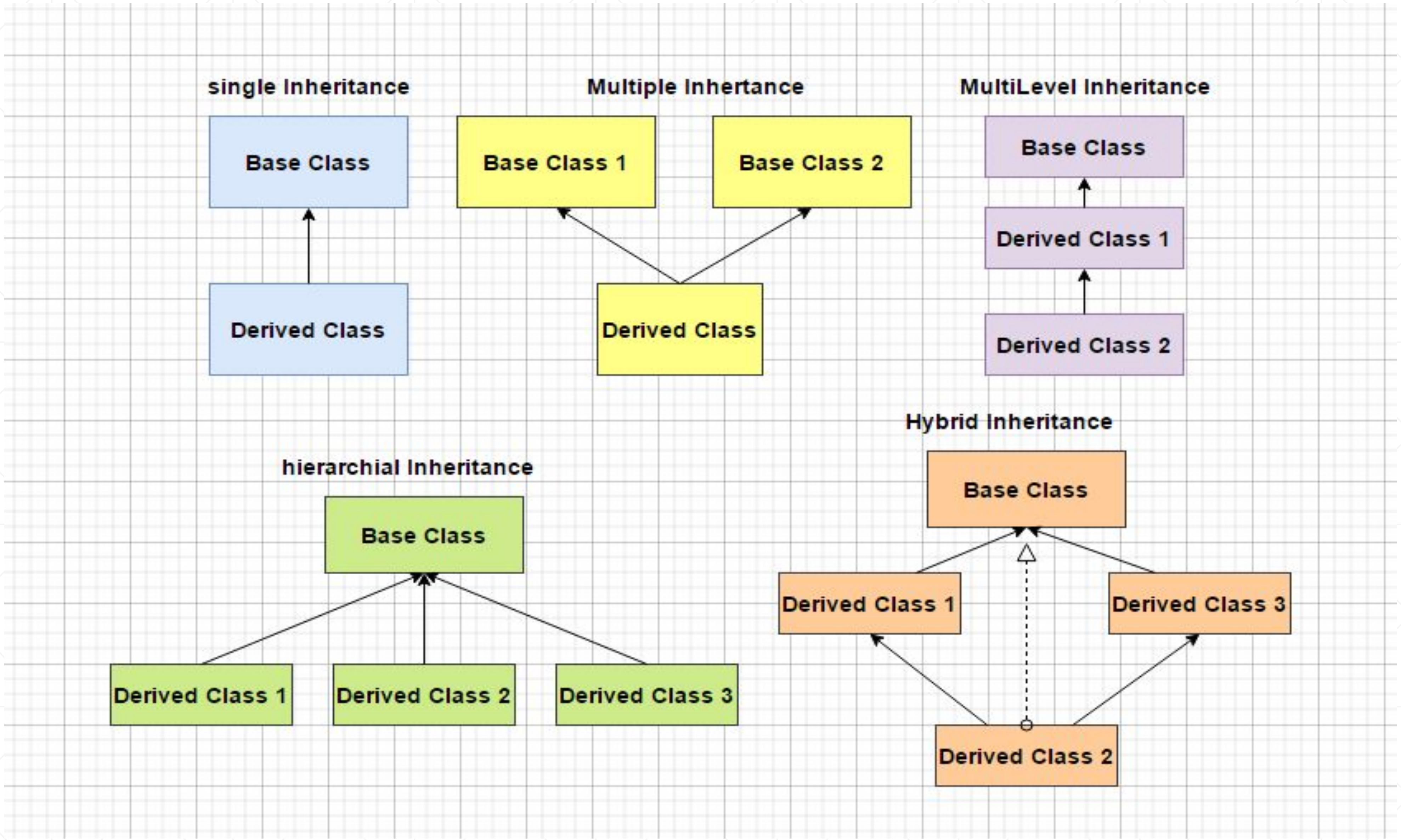
- When multiple child classes inherit their properties from just a single base class.

4) Multilevel Inheritance

- This type of inheritance is the best way to represent the transitive nature of inheritance. In multilevel inheritance, a derived class inherits all its properties from a class that itself inherits from another class.

5) Hybrid Inheritance

- This type of inheritance essentially combines more than two forms of inheritance discussed above. For instance, when a child class inherits from multiple base classes all of its parent classes and that child class itself serves as a base class for 3 of its derived classes.


single Inheritance
Base Class
Derived Class
Multiple Inhertance
Base Class 1
Base Class 2
Derived Class
MultiLevel Inheritance
Base Class
Derived Class 1
Derived Class 2
Hybrid Inheritance
Base Class
Derived Class 1
Derived Class 3
Derived Class 2
hierarchial Inheritance
Base Class
Derived Class 1
Derived Class 2
Derived Class 3

Single Inheritance Example

```
class Shape
{
protected:
    float width, height;
public:
    void set_data (float a, float b)
    {
        width = a;
        height = b;
    }};
class Rectangle: public Shape
{
public:
    float area ()
    {
        return (width * height);
    }};
class Triangle: public Shape
{
public:
    float area ()
    {
        return (width * height / 2);
    }};
```

```
int main ()
{
    Rectangle rect;
    Triangle tri;
    rect.set_data (5,3);
    tri.set_data (2,5);
    cout << rect.area() << endl;
    cout << tri.area() << endl;
    return 0;
}
```

output :
```
15
5
```

Multiple Inheritance Example

```
#include<iostream>
using namespace std;
class A
{
public:
int A_value;
void A_input()
{
cout<<"Enter the integer value of class A: ";
cin>>A_value;
}};
class B
{
public:
int B_value;
void B_input()
{
cout<<"Enter the integer value of class B: ";
cin>>B_value;
}};
class C : public A, public B //C is a derived class f
rom classes A and B
{
public:
void difference()
{
cout<<"The difference between the two values is: "
<< A_value - B_value<<endl;
}};
int main()
{
C c; // c is an Object of derived class C
c.A_input();
c.B_input();
c.difference();
return 0;
}
```

Output:

```
Enter the integer value of class A: 20
Enter the integer value of class B: 8
The difference between the two values is:
12
```

Hierarchical Inheritance Example

```
#include <iostream>
using namespace std;
class A
{
public:
int x, y;
void A_input()
{
cout<<"Enter two values of class A: ";
cin>>x>>y;
}
};
class B : public A // B is derived from A
{
public:
void product()
{
cout<<"The Product of the two values is: "<< x * y<<endl;
}
};
class C : public A //C is derived from A
{
public:
void division()
{
cout<<"The Division of the two values is: "<< x / y<<endl;
}
};
int main()
{
B b; // Object b of derived class B
C c; // Object c of derived class C
b.A_input();
b.product();
c.A_input();
c.division();
return 0;
}
```

Output:

```
Enter two values of class A: 12 2
The Product of the two values is: 24
Enter two values of class A: 12 2
The Division of the two values is: 6
```

Multilevel Inheritance Example

```
#include <iostream>
using namespace std;
class Base
{
public:
int base_value;
void Base_input()
{
cout<<"Enter the integer value of base class: ";
cin>>base_value;
}};
class Derived1 : public Base // Derived class of base class
{
public:
int derived1_value;
void Derived1_input()
{
cout<<"Enter the integer value of first derived class: ";
cin>>derived1_value;
}};
```

```
class Derived2 : public Derived1 // Derived
class of Derived1 class
{// private by deafult
int derived2_value;
public:
void Derived2_input()
{cout<<"Enter the integer value of
the second derived class: ";
cin>>derived2_value;
}
void sum()
{cout << "The sum of the three intger values is: " <<
base_value + derived1_value + derived2_value<<endl;
}};
```

Contd.. in next slide

```
int main()
{Derived2 d2; // Object d2 of second derived class
d2.Base_input();
d2.Derived1_input();
d2.Derived2_input();
d2.sum();
return 0;
}
```

```
Output:
Enter the integer value of base class: 5
Enter the integer value of first derived class: 6
Enter the integer value of the second derived class:
7
The sum of the three intger values is: 18
```

Hybrid Inheritance Example

```
#include <iostream>
using namespace std;
class A
{
        public:
        int A_value;
};
class B : public A
{
        public:
        B() // Use of a constructor to initialize A_value
{
A_value = 20;
}};
class C
{
        public:
        int C_value;
        C() //Use of a constructor to initialize C_value
        {
        C_value = 40;
        }
};
```

```
class D : public B, public C // D is derived from class B and
class C
{
public:
void product()
{
cout<<"The product of the two integer values is: " <<
A_value * C_value<<endl;
}
};
int main()
{
D d; // Object d of derived class D
d.product();
return 0;
}
```

Output:

```
The product of the two integer values is: 800
```

Derived class constructor

- If base class constructor does not have parameterised, then the derived class need not have constructor.

- If any one of the base class contains a constructor with one or more arguments then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructor because while applying inheritance, objects of the derived class are usually created.

- When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class are executed.

- There are two major questions that arise relative to constructors and destructors when inheritance is involved.
- First, when are base-class and derived-class constructors and destructors called?
- Second, how can parameters be passed to base-class constructors?

```
class base {
public:
base() {
cout<<“constructing base”<<endl;
}
base(int x,int y)
{
cout<<x<<y;
}
~base() {
cout<<“destructing base”<<endl;
   }
};

class derived: public base {
public:
```

Output:

```
constructing base
constructing derived
destructing derived
destructing base
```

- In the previous example, When an object of a derived class is created, the base class constructor will be called first, followed by the derived class constructor.
- When a derived object is destroyed, its destructor is called first, followed by the base class' destructor.
- Constructors are executed in their order of derivation. Destructors are executed in reverse order of derivation.

- In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.
- Ex: class A : public B, public C

```
          {

          }
```

Order of execution is B() C() and A()

- In multilevel inheritance, the constructors will be executed in the order of their inheritance.

```
class A
{
 //body of class A
}
class B:public A {
//body of class B
}
```

```
class C: public B {
//body of class c }
```

Output :

```
A()
B()
C()
```

Passing Parameters to Base class Constructors :

- The derived class is responsible for supplying initial values to its base class.
- The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class.
- The base class constructors are called and executed before executing the statements in the body of the derived constructors.
- **General form of defining a derived constructor :**

```
Derived constructor(arg-list):base1(arg-list),base2(arg-list)... baseN(arg-list))
{
}
```

- *base1 through baseN are the names of the base classes inherited by the derived* class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes.

- Ex: D(int a1,int a2,float b1,float b2,int d1): A(a1,a2),B(b1,b2)

```
{
 d=d1;
}
```

- In the main function, class D object is invoked as

  D obj(5,12,2.5,7.54,30)

  Here 5->a1,12->a2,2.5->b1,7.54-b2,30-d1

- Derived's constructor is declared as taking two parameters, x and y. However,
- Derived( ) uses only x.
- y is passed along to base( ). In general, the derived class' constructor
- It must declare both the parameter(s) that it requires as well as any required by the base class
- As the example illustrates, any parameters required by the base class are passed to it in the base class' argument list specified after the colon.

```
#include <iostream>
using namespace std;
class base {
protected:
int i;
public:
base(int x) {
 i=x; cout << "Constructing base\n"; }
~base() {
cout << "Destructing base\n"; }
};
class derived: public base {
int j;
int main()
{
derived ob(3, 4);
ob.show(); // displays 4 3
return 0;
}
```

Lets look at an other example that uses multiple base classes

```
class base1 {
protected:
int i;
public:
base1(int x) {
i=x;
cout << "Constructing base1\n"; }
~base1()
 {
cout << "Destructing base1\n"; }
};
class base2
{
protected:
```

```
class derived: public base1, public base2     {
int j;
public:
derived(int x, int y, int z): base1(y), base2(z)
{ j=x;
cout << "Constructing derived\n";
}
~derived() {
cout << "Destructing derived\n"; }
void show() {
cout << i << " " << j << " " << k << "\n"; }
};
```

- Output:

```
Constructing base1
Constructing base2
Constructing derived
4 3 5
Destructing derived
Destructing base2
Destructing base1
```

- In this program, the derived class' constructor takes no arguments, but **base1( ) and base2( ) do:**

```
class base1 {
protected:
int i;
public:
base1(int x) {
 i=x;
cout << "Constructing base1\n"; }
};
class base2 {
protected:
int k;
public:
base2(int x) {
k=x;
cout << "Constructing base2\n"; }
```

```
int main()
{
derived ob(3, 4);
ob.show(); // displays 3 4
return 0;
}
```

- A derived class' constructor is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class.
- Put differently, passing an argument along to a base class does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
class derived: public base {

int j;

public:

derived(int x, int y): base(x, y)

{

j = x*y;

cout << "Constructing derived\n";

}
```

- Why is Base class Constructor called inside Derived class?
- Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

ISSUES in INHERITENCE:

Ambiguity in Multiple Inheritance

The most obvious problem with multiple inheritance occurs during function overriding.

Suppose, two base classes have a same function which is not overridden in derived class.

If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. For example,

```
class base1
{
  public:
     void someFunction( )
     { .... ... .... }
};
class base2
{
    void someFunction( )
     { .... ... .... }
};
class derived : public base1, public base2
{
```

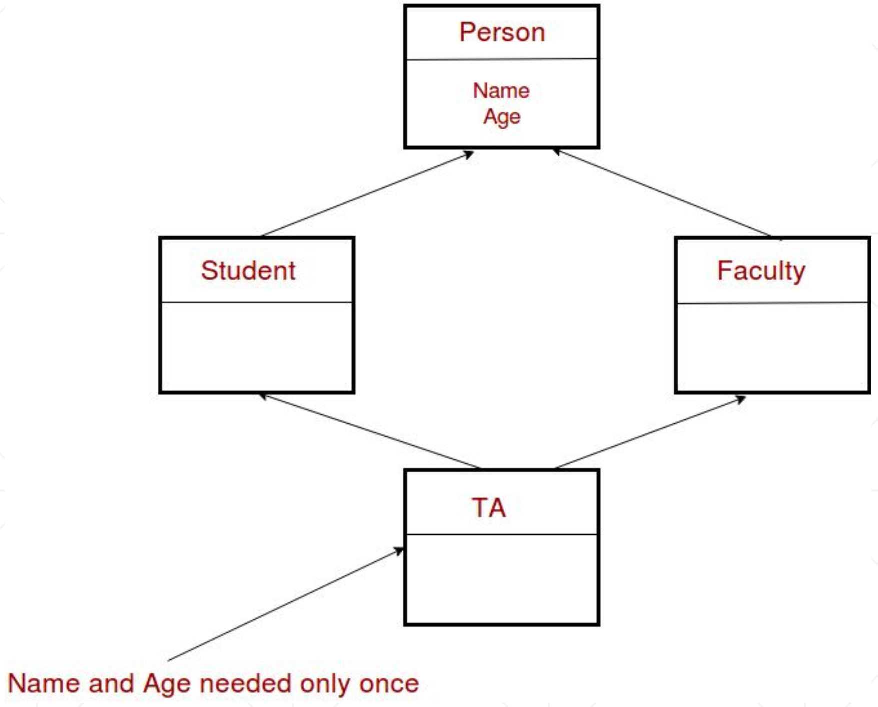
This problem can be solved using scope resolution function to specify which function to class either base1or base2

```
int main()

{

    obj.base1::someFunction( );  // Function of base1 class is called

    obj.base2::someFunction();   // Function of base2 class is called.

}
```

ISSUES in INHERITENCE:

The diamond problem

The diamond problem occurs when two super classes of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class,



For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
   // Data members of person
public:
    Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
};

class Faculty : public Person {
   // data members of Faculty
public:
    Faculty(int x):Person(x)   {
       cout<<"Faculty::Faculty(int ) called"<< endl;
    }
```

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In the above program, constructor of ‘Person’ is called two times.

Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed.

So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities.

The solution to this problem is ‘**virtual**’ keyword. We make the classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class.

For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
    Person()     { cout << "Person::Person() called" << endl;   }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x)   {
       cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
```

Output:

```
Person::Person() called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In the above program, **constructor of ‘Person’ is called once.**

One important thing to note in the above output is, **the default constructor of ‘Person’ is called**.

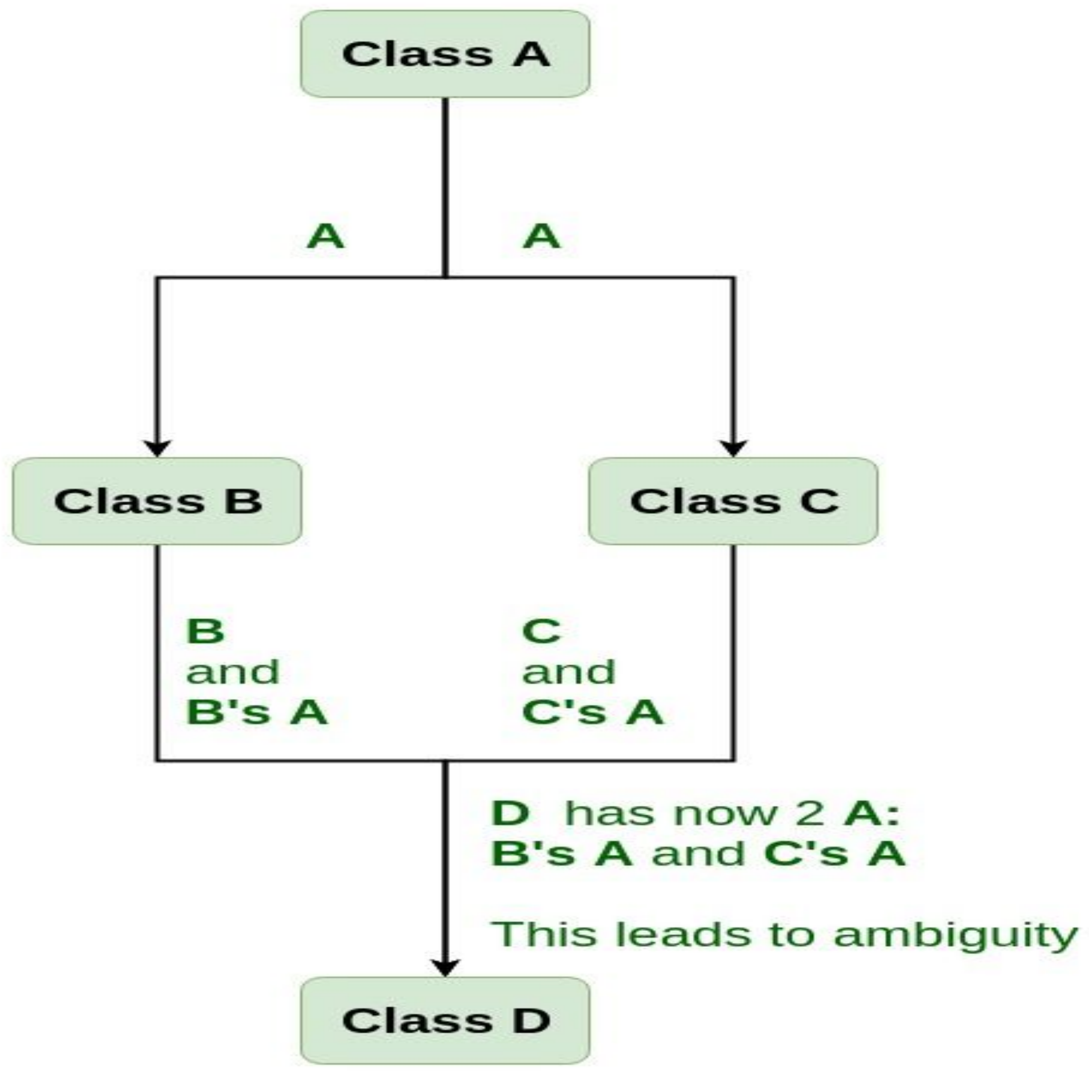
When we use ‘virtual’ keyword, the default constructor of grandparent class is called by **default** even if the parent classes explicitly call parameterized constructor.

Virtual base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes:

Consider the situation where we have one class A .This class is A is inherited by two other classes B and C. Both these class are inherited into another in a new class D as shown in figure below.



- As we can see from the figure that data members/function of class A are inherited twice to class D.
- One through class B and second through class C.
- When any data / function member of class A is accessed by an object of class D, ambiguity arises as to which data/function member would be called?
- One inherited through B or the other inherited through C. This confuses compiler and it displays error.

Topics Covered

- Issues in Inheritance
- Virtual Base Class

Issues in Inheritance: The diamond problem

The diamond problem occurs when two super classes of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.

```
#include<iostream>
using namespace std;
class Person {
  // Data members of person
public:
    Person(int x)  { cout << "Person::Person(int ) called" <<
    endl;   }
};
 class Faculty : public Person {
  // data members of Faculty
public:
    Faculty(int x):Person(x)   {
       cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
class Student : public Person {
   public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};
class TA : public Faculty, public Student  {
public:
    TA(int x):Student(x), Faculty(x)   {
        cout<<"TA::TA(int ) called"<< endl;
    }
};
int main()  {
    TA ta1(30);
}
```

Output:

```
    Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called TA::TA(int
) called
```

- In the above program, constructor of ‘Person’ is called two times. Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed. So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities.

Virtual Base Class

- *The solution to the problem in previous example is ‘virtual’ keyword*. We make the classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class.

```
#include<iostream>
using namespace std;
class Person {
public:
   Person(int x)
    { cout << "Person::Person(int ) called" << endl;   }
   Person()
       { cout << "Person::Person() called" << endl;   }
};
class Faculty : virtual public Person {
public:
   Faculty(int x):Person(x)
     {
     cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
class Student : virtual public Person {
public:
   Student(int x):Person(x) {
       cout<<"Student::Student(int ) called"<< endl;
   }
};
class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x)   {
       cout<<"TA::TA(int ) called"<< endl;
   }
};
int main()  {
   TA ta1(30);
}
```

Output

```
   Person::Person() called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
    Person()     { cout << "Person::Person() called" << endl;   }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x)   {
       cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
```

>>>>>Contd…

How to call the parameterized constructor of the ‘Person’ class?

The constructor has to be called in ‘TA’ class.

```
class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};
class TA : public Faculty, public Student  {
public:
    TA(int x):Student(x), Faculty(x), Person(x)   {
        cout<<"TA::TA(int ) called"<< endl;
    }
};
int main()  {
    TA ta1(30);
}
```

Output:

```
Person::Person(int ) called
    Faculty::Faculty(int ) called
    Student::Student(int ) called
    TA::TA(int ) called
```

Note:
In general, it is not allowed to call the grandparent's constructor directly, it has to be called through parent class. It is allowed only when 'virtual' keyword is used.

Exercise (Predict the Output)

```
#include <iostream>
using namespace std;
class A {
public:
    void show()
    {
        cout << "Hello form A \n";
    }
};

class B : public A {
};
```

Output:
error: request for member 'show' is ambiguous object.show();

Solution:To resolve this ambiguity when class A is inherited in both class B and class C, it is declared as virtual base class by placing a keyword virtual as

```
#include <iostream>

using namespace std;

class A {

public:

      int a;

      A() // constructor

      {

            a = 10;

      }

};

class B : public virtual A
      {

};

class C : public virtual A {

};
```

Output:
```
a = 10
```

Topics Covered

- Polymorphism

Polymorphism

- Polymorphism is crucial feature of Object Oriented Programming.
- The process of representing one form in multiple forms is known as Polymorphism. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.
- Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and morphs means forms.
- Polymorphism is the ability to create a variable, a function or an object that has more than one form.

- For example: 1

  The + (plus) operator in C++:

  4+5 <-- Integer addition

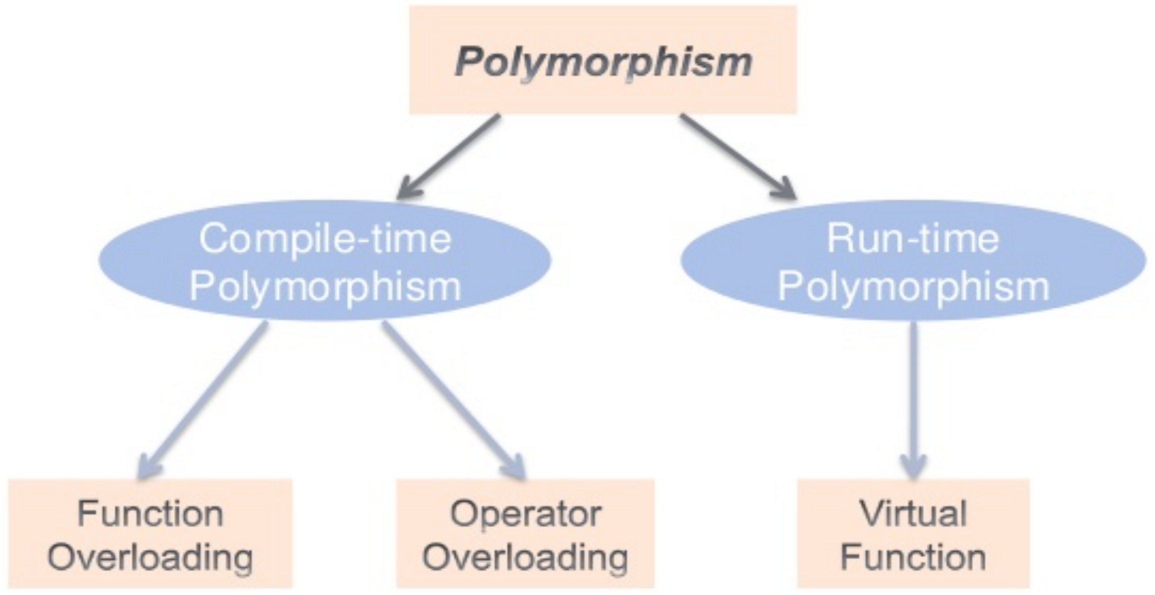
  3.14 + 2.0 <-- Floating point addition

  s1 + "bar" <-- String concatenation!

2. Real time example of Polymorphism

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.

Types of Polymorphism



Compile time polymorphism

- In compile time polymorphism, compiler is able to select the appropriate function a particular call at the compile time.

Example:

```
#include<iostream> using namespace std;                                        Output:
                                                                               Hello, World!
void sayHi();

int main()

{                                                  sayHi(); // the compiler binds any invocation of sayHi()
                                 // to sayHi()'s entry point.
}                                                       void sayHi()
{                                                              cout << ''Hello, World!\n'';
                                               }
```

In C, only compile-time binding is provided.

2. Compile time Polymorphism Example for function overloading

```
#include <iostream>
using namespace std;
class Add {
public:

  int sum(int num1, int num2){
     return num1+num2;
  }

  int sum(int num1, int num2, int num3){
     return num1+num2+num3;
  }
};
int main() {
  Add obj;
  //This will call the first function
  cout<<"Output: "<<obj.sum(10, 20)<<endl;
  //This will call the second function
  cout<<"Output: "<<obj.sum(11, 22, 33);
  return 0;
}
```

Output:-

```
Output: 30

Output: 66
```

by virtual functions and run-time binding mechanism in C++. A class is called polymorphic if it contains virtual functions.

Run time polymorphism

- Function overriding is an example of Runtime polymorphism.

- **Function Overriding**: When child class declares a method, which is already present in the parent class

C++ Runtime Polymorphism Example: 1

```
#include <iostream>
using namespace std;
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {   cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

Output:

```
Eating bread...
```

2. **Example of Runtime Polymorphism**

```
#include <iostream>
using namespace std;
class A {
public:
  void disp(){
     cout<<"Super Class Function"<<endl;
  }
};
class B: public A{
public:
  void disp(){
     cout<<"Sub Class Function";
  }
};
int main() {
  //Parent class object
  A obj;
  obj.disp();
  //Child class object
  B obj2;
  obj2.disp();
  return 0;
}
```

```
Output:

Super Class Function
Sub Class Function
```

Differences b/w compile time and run time polymorphism

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

BENEFITS OF POLYMORPHISM

- **Simplicity:** This makes your code easier for you to write and easier for others to understand.
- **Extensibility:** Polymorphism design and implements system that are more extensible.

Topics to be Covered.

- Virtual Functions
- Pure Virtual Functions

Virtual Functions

Virtual function: – Polymorphism in biology means ability of an organism to assume a variety of forms.

- In C++, it indicates the form of a member function that can be changed at run time. Such member function are called virtual member function and the corresponding class is called polymorphic class.
- A virtual function is a member function that is declared within a base class and redefined by a derived class.
- To create a virtual function, precede the function's declaration in the base class with the keyword **virtual.**
- When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs.

Virtual Functions

- Virtual functions implement the **"one interface, multiple methods"** philosophy that underlies polymorphism.

- It is used to tell the compiler to perform dynamic linkage or late binding on the function

- The object of the polymorphic class, addressed by pointer, change at run time and respond differently for the same message. Such a mechanism requires postponement of binding of a function call to the member function until run time.

- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

- Rules for Virtual Function in C++:

- They are always defined in a base class and overridden in derived class but it is not mandatory to override in the derived class.

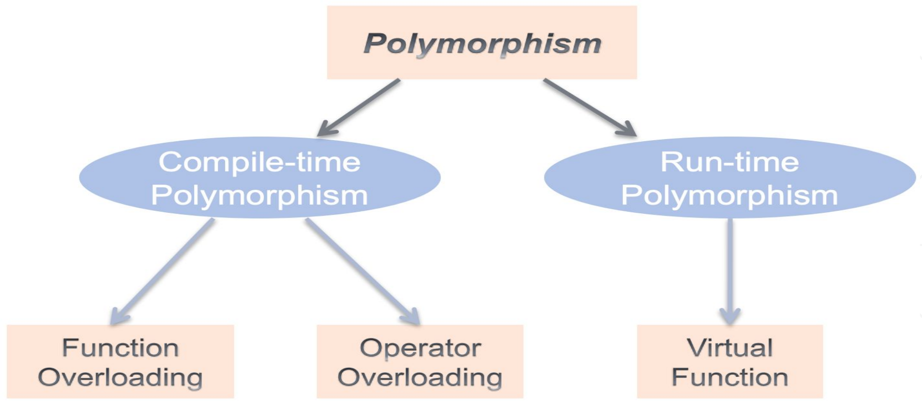
- The virtual functions must be declared in the public section of the class.

- They cannot be static or friend function also cannot be the virtual function of another class.

- The virtual functions should be accessed using a pointer to achieve run time polymorphism.

- The virtual functions should be accessed using a pointer to achieve run time polymorphism.

When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this determination is made *at run time*.



```
      #include <iostream>
      using namespace std;
      class base
      {
      public:

     virtual void vfunc()

      {

     cout << "This is base's
vfunc().\n";

      } };
      class derived1 : public base {
      public:

     void vfunc() {
     cout << "This is derived1's
vfunc().\n";

      } };

   class derived2 : public base {
      public:

   void vfunc() {
std::cout << "This is derived2's
vfunc().\n";

   } };
```

```
     int main() {
        base *p, b;
        derived1 d1;
        derived2 d2;

   // point to base
p = &b;
p->vfunc(); // access base's

                //vfunc()

   // point to derived1
p = &d1;

   // access derived1's
vfunc()
p->vfunc();

   // point to derived2
p = &d2;
p->vfunc(); // access
derived2's

                //vfunc()

   return 0; }
```

Calling Virtual func through a base class pointer

OR

```
// Use a base class reference parameter.

void f(base &r)

 {

r.vfunc();

}

int main() {
     base b;
     derived1 d1;
     derived2 d2;
     f(b); // pass a base object to f()
     f(d1); // pass a derived1 object to
f()
     f(d2); // pass a derived2 object to
f()

return 0; }
```

Calling Virtual function through Base class reference

Output:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

Note :virtual function must match exactly the prototype specified in the base class.

(This differs from overloading a normal function, in which return types and the number and type of parameters may differ.)

Although you can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when a**ccess is through a base-class pointer (or reference) that run-time polymorphism is achieved**.

For example, assuming the preceding example, this is syntactically valid:

d2.vfunc(); // calls derived2's vfunc()

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc( )**.

Note: Term *overriding* is used to describe virtual function redefinition by a derived class.

The Virtual Attribute Is Inherited (Example)

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


class base {
public:
virtual void vfunc() {
    std::cout << "This is base's vfunc().\n";
}};
class derived1 : public base {
public:
void vfunc() {
    std::cout << "This is derived1's vfunc().\n";
} };
```

```
/* derived2 inherits virtual function
vfunc() from derived1. */

class derived2 : public derived1 {

public:

  // vfunc() is still ////virtual

  void vfunc() {

std::cout << "This is derived2's
vfunc().\n";

                }};
```

```
  int main() {

  base *p, b;

  derived1 d1;

  derived2 d2;

// point to base

p = &b;

p->vfunc(); // access base's vfunc()

// point to derived1

p = &d1;p->vfunc(); // access derived1's vfunc()

// point to derived2

p = &d2;

p->vfunc();

return 0;

}
```

```
Output:This is base's vfunc().
       This is derived1's vfunc().
       This is derived2's vfunc().
```

Virtual Functions Are Hierarchical -1

```
#include <iostream>
   using namespace std;
   class base {
   public:

virtual void vfunc() {
std::cout << "This is base's
vfunc().\n";

} };
   class derived1 : public
base {
   public:

void vfunc() {
std::cout << "This is
derived1's vfunc().\n";

} };
```

```
class derived2 :
public base
{
 public:
  /* vfunc() not
overridden by
derived2, base's
is used*/
   };
```

```
int main() {
     base *p, b;
     derived1 d1;
     derived2 d2;

// point to base
p = &b;
p->vfunc(); // access

           //base’s vfunc()

// point to derived1
p = &d1;
p->vfunc(); // access
derived1’s

            //vfunc()

// point to derived2
p = &d2;
p->vfunc(); // use base's

             //vfunc()

return 0; }
```

Output

```
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
```

Virtual Functions Are Hierarchical -2

```
#include <iostream>
   using namespace std;
   class base {
   public:

virtual void vfunc() {
std::cout << "This is base's
vfunc().\n";

} };
   class derived1 : public
base {
   public:

void vfunc() {
std::cout << "This is
derived1's vfunc().\n";

} };
```

```
class derived2 :
public derived1
{
 public:
  /* vfunc() not
overridden by
derived2, base's
is used*/
   };
```

```
int main() {
     base *p, b;
     derived1 d1;
     derived2 d2;

// point to base
p = &b;
p->vfunc(); // access

           //base’s vfunc()

// point to derived1
p = &d1;
p->vfunc(); // access
derived1’s

            //vfunc()

// point to derived2
p = &d2;
p->vfunc(); // use derived1

             //vfunc()

return 0; }
```

Note: This means that when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used.

```
Output:This is base's vfunc().
       This is derived1's vfunc().
       This is derived1’s vfunc().
```

Pure Virtual Function

- a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created.

- In some situations you will want to ensure that all derived classes override a virtual function.

  To handle the above two cases, C++ supports **the pure virtual function.**

- A *pure virtual function* is a virtual function that has no definition within the base class.
- When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.
- To declare a pure virtual function, use this general form:

```
virtual type func-name(parameter-list) = 0;
```

Pure Virtual Functions Example :

```
#include <iostream>
using namespace std;
class number {
protected:
  int val;
public:
  void setval(int i)
   { val = i; }
// show() is a pure
//virtual function
  virtual void show() = 0;
};

class hextype : public number {
public:
  void show() {
    std::cout << hex << val << "\n";
    }};
```

```
class dectype : public number {
  public:
    void show() {
      cout << val << "\n";
  } };

class octtype : public number {
public:
  void show() {
    std::cout << oct<< val << "\n";
} };
```

```
int main() {
  dectype d;
  hextype h;
  octtype o;
d.setval(20);
d.show(); // displays 20 -
decimal
h.setval(20);
h.show(); // displays 14 -
hexadecimal
o.setval(20);
o.show(); // displays 24 -
octal
return 0;

}
```

```
Output:
20
14
24
```

Abstract Classes

- A class that contains at least one pure virtual function is said to be *abstract*.
- Because an abstract class contains one or more functions for which there is no definition (that is,
  a pure virtual function), no objects of an abstract class may be created.
- A subclass of an abstract class is also abstract if it does not provide implementations for all the pure virtual functions in the superclass
- A class that has all its member functions pure virtual is called an interface

Abstract Classes

- Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class.
- For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn’t have implementation of move() (assuming that all animals move), but all animals must know how to move.
- **We cannot create objects of abstract classes.**
- A pure virtual function (or abstract function) in C++ is a virtual function for which we don’t have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.
- // An abstract class

class Test

Abstract Classes

- A complete example:
- A pure virtual function is implemented by classes which are derived from a Abstract class. Following is a simple example to demonstrate the same.
- #include<iostream>
- using namespace std;
-
- class Base
- {
-    int x;
- public:
-     virtual void fun() = 0;
-     int getX() { return x; }
- };
-
- // This class inherits from Base and implements fun()
- class Derived: public Base
- {

Output:

fun() called

Some Interesting Facts:

***1) A class is abstract if it has at least one pure virtual function*.**

- In the following example, Test is an abstract class because it has a pure virtual function show().
- // pure virtual functions make a class abstract
- #include<iostream>
- using namespace std;
-
- class Test
- {
-     int x;

Output:

Compiler Error: cannot declare variable 't' to be of abstract type 'Test' because the following virtual functions are pure within 'Test': note: virtual void Test::show()

Abstract Classes

Some Interesting Facts:

2) We can have pointers and references of abstract class type.

For example the following program works fine.

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

class Base
{
public:
    virtual void show() = 0;
};
```

Output:

```
In Derived
```

Some Interesting Facts:

3) If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

The following example demonstrates the same.

```
#include<iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};
```

Compiler Error: cannot declare variable 'd' to be of abstract type 'Derived' because the following virtual functions are pure within 'Derived': virtual void Base::show()

Abstract Classes

Some Interesting Facts:

4) An abstract class can have constructors.

For example, the following program compiles and runs fine..

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

// An abstract class with constructor
class Base
{
protected:
   int x;
public:
  virtual void fun() = 0;
  Base(int i) { x = i; }
};

class Derived: public Base
{
```

Output:

```
x = 4, y = 5
```

Difference between Virtual function and Pure virtual function in C++

1. **Virtual Function in C++**

A virtual function a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

2. **Pure Virtual Functions in C++**

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have an implementation, we only declare it. A pure virtual function is declared by assigning 0 in the declaration.

3. Similarities between virtual function and pure virtual function

Difference between Virtual function and Pure virtual function in C++

VIRTUAL FUNCTION	PURE VIRTUAL FUNCTION
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
Classes having virtual functions are not abstract.	Base class containing pure virtual function becomes abstract.

Syntax:
 	
`virtual<func_type><func_name>()`
`{`
`    // code`
`}`
	Syntax:
	
	`virtual<func_type><func_name>()`
	`    = 0;`

Definition is given in base class.	No definition is given in base class.
Base class having virtual function can be instantiated i.e. its object can be made.	Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.
If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

Early Binding Versus Late Binding

early binding

- *Early binding* refers to events that occur at compile time.
- In essence, early binding occurs when all information needed to call a function is known at compile time.
- an object and a function call are bound during compilation.)
- The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.
- Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators

***late binding*.**

- it relates to C++, late binding refers to function calls that are not resolved until run time.
- Virtual functions are used to achieve late binding.
- As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer.
- The main advantage of late binding is flexibility.
- Late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code."

Practicle example of"One Interface and Multiple Method"

```
// Virtual function practical example.
#include <iostream>
using namespace std;
   class convert {
     protected:
     double val1;  // initial value
     double val2;  // converted value
   public:
     convert(double i) {
       val1 = i;
     }
     double getconv() { return val2; }
     double getinit() { return val1; }
     virtual void compute() = 0;
   };
```

```
// approach : use virtual functions,
//abstract classes, and run-time polymorphism.
```

```
// Liters to gallons.
    class l_to_g : public convert
    {
public:
l_to_g(double i) : convert(i) { }

        void compute() {
    val2 = val1 / 3.7854; }
    };


class f_to_c : public convert {

    public:
     f_to_c(double i) : convert(i) { }
     void compute() {
     val2 = (val1-32) / 1.8; }
     };
```

```
int main() {
    convert *p;  // pointer to base class
    l_to_g lgob(4);
    f_to_c fcob(70);
     p=&lgob;
   std::cout << p->getinit() << " liters is\n ";

   p->compute();
  std::cout << p->getconv() << " gallons\n";
                                    //l_to_g happened
   p = &fcob;
  std::cout << p->getinit() << " in Fahrenheit is ";
   p->compute();
   std::cout << p->getconv() << " Celsius\n";
                                    //    f_to_c happened
return 0; }
```

Output:

```
4 liters is
 1.05669 gallons
70 in Fahrenheit is 21.1111 Celsius
```