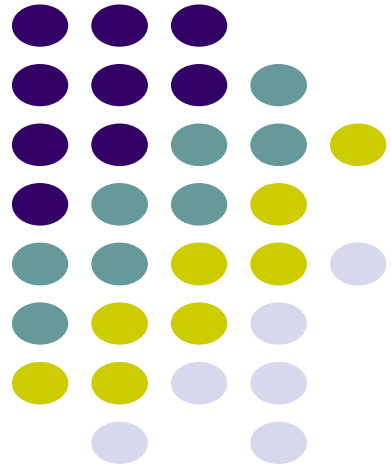
Object Oriented Programming in C++

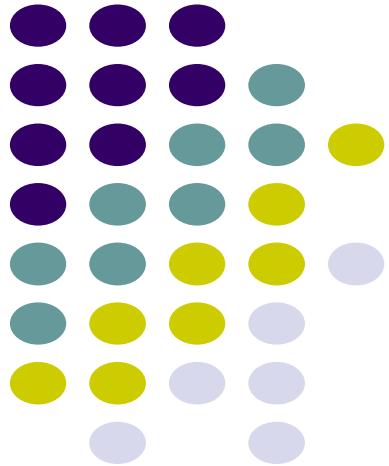
Presented by Bhimashankar T

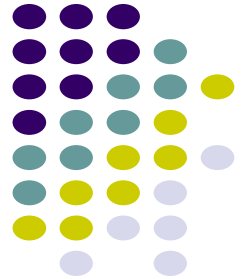
E-Mail: bhima.t@gmail.com

PhNo:+91 9980156833 / 6363863430



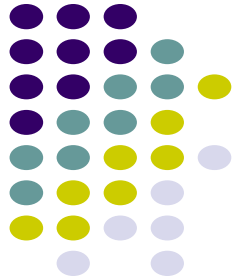
DAY 02





OOPS Concept

- Objects
- Classes
- Abstraction
- Encapsulation
- Inheritance
- Overloading
- Exception Handling



OOPS Concept Definitions

Objects

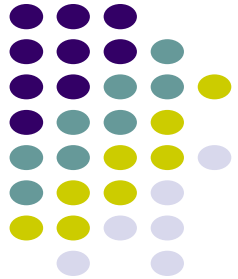
Objects are the basic unit of OOP. They are instances of class, which have data members and uses various member functions to perform tasks.

Class

It is similar to structures in C language. Class can also be defined as user defined data type but it also contains functions in it. So, class is basically a blueprint for object. It declare & defines what data variables the object will have and what operations can be performed on the class's object.

Abstraction

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes can provide methods to the outside world to access & use the data variables, keeping the variables hidden from direct access, or classes can even declare everything accessible to everyone, or maybe just to the classes inheriting it. This can be done using access specifiers.



OOPS Concept Definitions

Encapsulation

It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class.

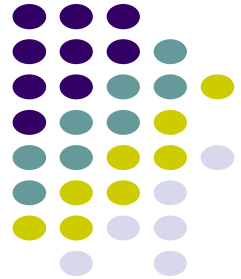
Inheritance

Inheritance is a way to reuse once written code again and again. The class which is inherited is called the **Base** class & the class which inherits is called the **Derived** class. They are also called parent and child class.

So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

Polymorphism

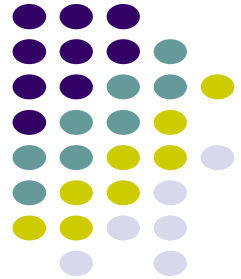
It is a feature, which lets us create functions with same name but different arguments, which will perform different actions. That means, functions with same name, but functioning in different ways. Or, it also allows us to redefine a function to provide it with a completely new definition.



OOPS Concept Definitions

Exception Handling

Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime.



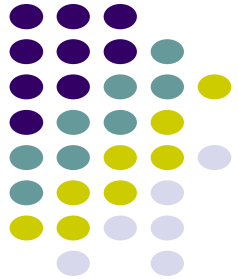
Introduction to Classes and Objects

Objects

Each object has different data variables. Objects are initialised using special class functions called **Constructors**. We will study about constructors later.

And whenever the object is out of its scope, another special class member function called **Destructor** is called, to release the memory reserved by the object. C++ doesn't have Automatic Garbage Collector like in JAVA, in C++ Destructor performs this task.

```
class Abc
{
 int x;
 void display(){} //empty function
};

in main()
{
 Abc obj;   // Object of class Abc created
}
```



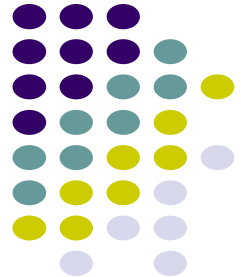
Access Control in Classes

Access specifiers in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

1. public
2. private
3. protected

```
class ProtectedAccess
{
 protected:   // protected access
 int x;            // Data Member
 void display();   // Member Function
}
```

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is subclass of class A.

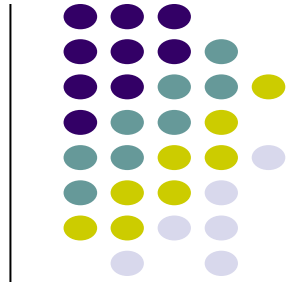


Defining Class and Declaring Objects

```
class ClassName
{
 Access specifier:
 Data members;
 Member Functions(){}
};
```

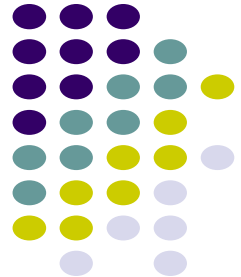
```
class Student
{
 public:
 int rollno;
 string name;
};
```

```
class Student
{
public:
int rollno;
string name;
}A,B;
```



Accessing Data Members of Class

```
class Student
{
 public:
 int rollno;
 string name;
};

int main()
{
 Student A;
 Student B;
 A.rollno=1;
 A.name="Adam";

 B.rollno=2;
 B.name="Bella";

 cout <<"Name and Roll no of A is :"<< A.name << A.rollno;
 cout <<"Name and Roll no of B is :"<< B.name << B.rollno;
}
```

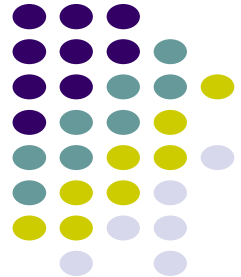


Accessing Data Members of Class

Accessing Private Data Members

```
class Student
{
 private:    // private data member
 int rollno;

 public:     // public accessor and mutator functions
 int getRollno()
 {
  return rollno;
 }

 void setRollno(int i)
 {
  rollno=i;
 }

};
```

```
int main()
{
 Student A;
 A.rollono=1;  //Compile time error
 cout<< A.rollno; //Compile time error

 A.setRollno(1);  //Rollno initialized to 1
 cout<< A.getRollno(); //Output will be 1
}
```



Constructors

Constructors are of three types :

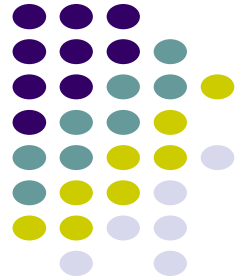
1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor

Default

```
class Cube
{
int side;
public:
Cube()
 {
  side=10;
 }
};

int main()
{
Cube c;
cout << c.side;
}                                   // Output : 10
```

Parameterized

```
class Cube
{
 public:
 int side;
 Cube(int x)
  {
   side=x;
  }
};

int main()
{
 Cube c1(10);
 Cube c2(20);
 Cube c3(30);
 cout << c1.side;
 cout << c2.side;
 cout << c3.side;
}                                   // OUTPUT : 10 20 30
```



Constructors

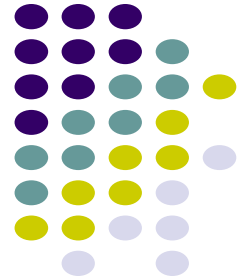
Copy Constructor

```
class Samplecopyconstructor
{
private:
int x, y; // data members
public:
Samplecopyconstructor(int x1, int y1)
{
x = x1;
y = y1;
}
// Copy constructor
Samplecopyconstructor (const Samplecopyconstructor &sam)
{
x = sam.x;
y = sam.y;
}
void display()
{
cout<<x<<" "<<y<<endl;
}
};
int main()
{
Samplecopyconstructor obj1(10, 15); // Normal constructor
Samplecopyconstructor obj2 = obj1; // Copy constructor
cout<<"Normal constructor : ";
obj1.display();
cout<<"Copy constructor : ";
obj2.display();
return 0;
}
```

Output:

Normal constructor : 10 15

Copy constructor : 10 15



Constructors

Constructor Overloading

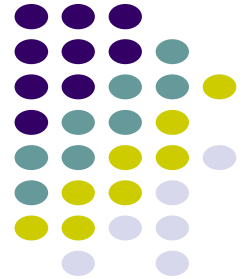
```
class Student
{
 int rollno;
 string name;
 public:
 Student(int x)
 {
  rollno=x;
  name="None";
 }
 Student(int x, string str)
 {
  rollno=x ;
  name=str ;
 }
};
```

```
int main()
{
 Student A(10);
 Student B(11,"Ram");
}
```

Note: Whenever we are using para.. constructor… without default constructor then we can declare like

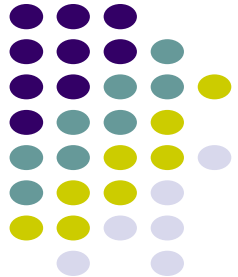
Student S;

it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.



Destructors

```
class A
{
A()
 {
  cout << "Constructor called";
 }

~A()
 {
  cout << "Destructor called";
 }
};

int main()
{
 A obj1;   // Constructor Called
 int x=1;
 if(x)
  {
    A obj2;  // Constructor Called
  }   // Destructor Called for obj2
} //  Destructor called for obj1.
```



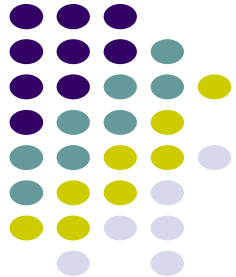
References in C++

```
int main()
{ int y=10;
  int &r = y;  // r is a reference to int y
  cout << r;
}
```

Output :10

Difference between Reference and Pointer

References	Pointers
Reference must be initialized when it is created.	Pointers can be initialized any time.
Once initialized, we cannot reinitialize a reference.	Pointers can be reinitialized any number of time.
You can never have a NULL reference.	Pointers can be NULL.
Reference is automatically dereferenced.	* is used to dereference a pointer.



C++ Static Data Members

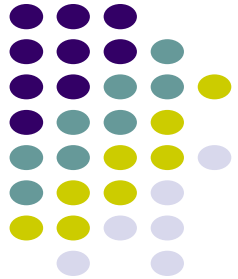
Static data members are class members that are declared using static keywords. A static member has certain special characteristics which are as follows:

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is initialized before any object of this class is created, even before the main starts.
- It is visible only within the class, but its lifetime is the entire program.

Syntax:

```
static data_type data_member_name;
```

```
class X {
        static int n;
};
// declaration (uses 'static')
int X::n = 1; // definition (does not use 'static')
```

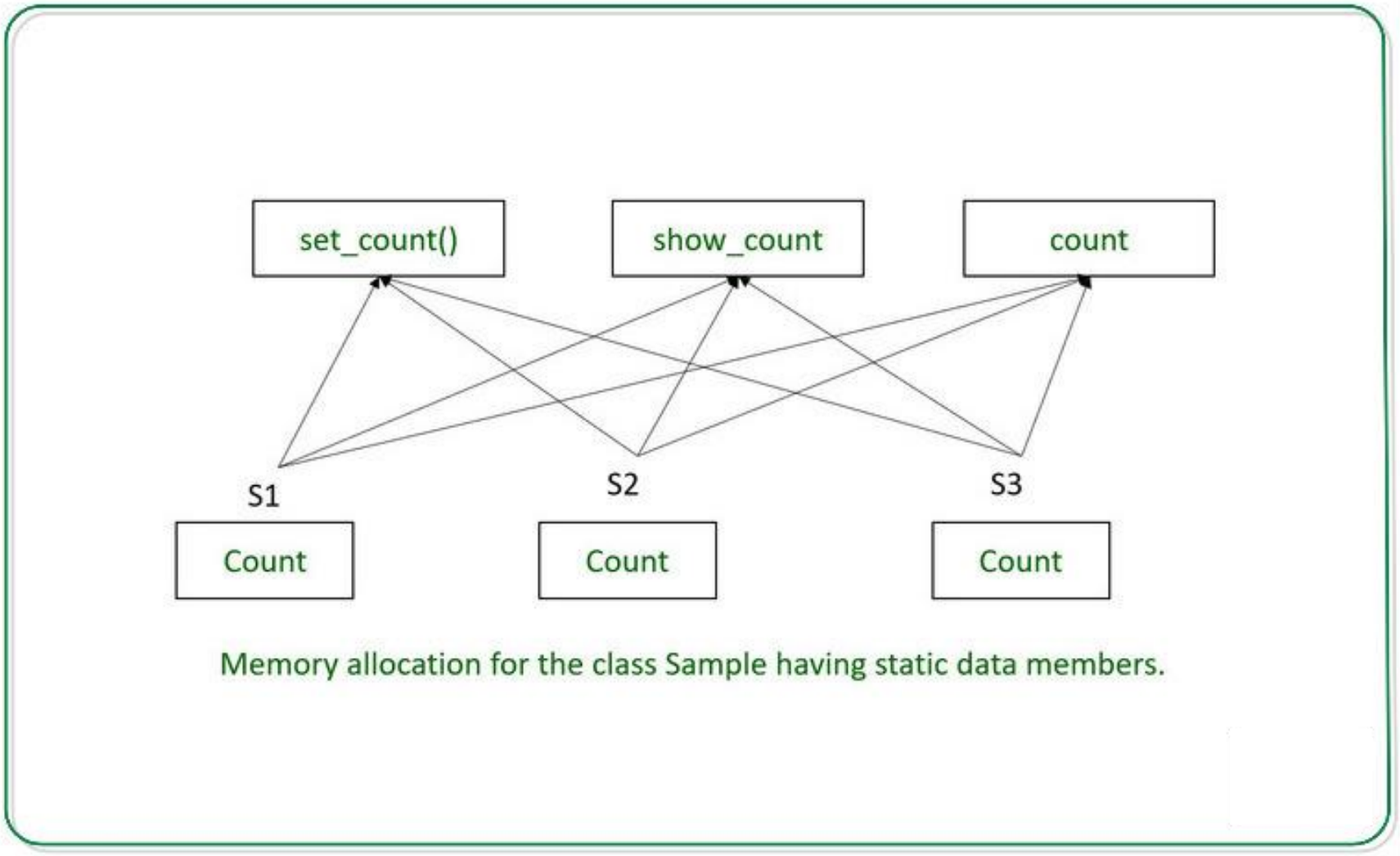


C++ Static Data Members

```
// C++ program to illustrate non-static data members
using namespace std;
#include <iostream>

// Class
class Test {
private:
        // Created a static variable
        static int count;

public:
        // Member function to increment
        // value of count
        void set_count()
        {
                count++;
        }

        // Member function to access the
        // private members of this class
        void show_count()
        {

                // print the count variable
                cout << count << '\n';
        }
};

int Test::count = 0;  //defining static members

// Driver Code
int main()
{
        // Objects of class Test
        Test S1, S2, S3;

        // Increment count variable
        // by 1 for each object
        S1.set_count();
        S2.set_count();
        S3.set_count();

        // Function to display count
        // for each object
        S1.show_count();
        S2.show_count();
        S3.show_count();

        return 0;
}
```

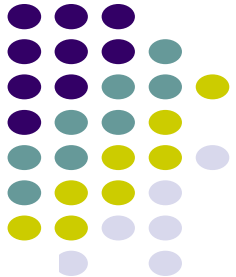
Output: `3 3 3`

Below is the illustration of memory allocation for the above program:



Memory allocation for the class Sample having static data members.

C++ Member Function in C++



Static Member Function in a class is the function that is declared as static because of which function attains certain properties as defined below:

- A static member function is independent of any object of the class.
- A static member function can be called even if no objects of the class exist.
- A static member function can also be accessed using the class name through the scope resolution operator.
- A static member function can access static data members and static member functions inside or outside of the class.
- Static member functions have a scope inside the class and cannot access the current object pointer.
- You can also use a static member function to determine how many objects of the class have been created.

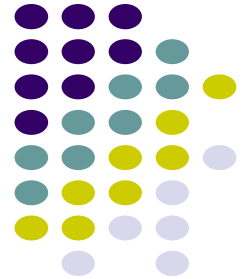
The reason we need Static member function:

- Static members are frequently used to store information that is shared by all objects in a class.
- For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter. This static data member can be increased each time an object is generated to keep track of the overall number of objects.

C++ Member Function in C++

```
// C++ Program to show the working of static member functions
#include <iostream>
using namespace std;

class Box
{
private:
        static int length;
        static int breadth;
        static int height;

public:

        static void print()
        {
                cout << "The value of the length is: " << length << endl;
                cout << "The value of the breadth is: " << breadth << endl;
                cout << "The value of the height is: " << height << endl;
        }
};

// initialize the static data members

int Box :: length = 10;
int Box :: breadth = 20;
int Box :: height = 30;

// Driver Code

int main()
{

        Box b;

        cout << "Static member function is called through Object name: \n"
        << endl;
        b.print();

        cout << "\nStatic member function is called through Class name: \n"
        << endl;
        Box::print();

        return 0;
}
```

C++ Member Function in C++



Output:

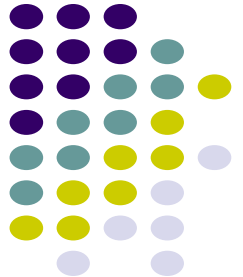
Static member function is called through Object name:

The value of the length is: 10
The value of the breadth is: 20
The value of the height is: 30

Static member function is called through Class name:

The value of the length is: 10
The value of the breadth is: 20
The value of the height is: 30

‘this’ pointer in C++

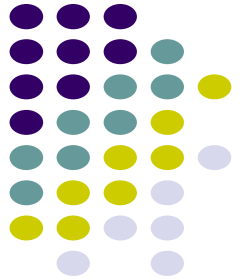


To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

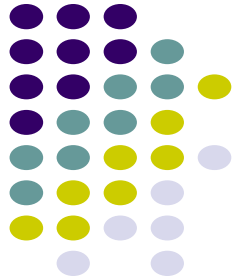
Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
The compiler supplies an implicit pointer along with the names of the functions as ‘this’.

‘this’ pointer in C++



The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).

‘this’ pointer in C++



Following are the situations where ‘this’ pointer is used:

1) When local variable’s name is same as member’s name

```
/* local variable is same as a member's name */
class Test
{
    private:
        int x;
    public:
        void setX (int x)
        {
            // The 'this' pointer is used to retrieve the object's x
            // hidden by the local variable 'x'
            this->x = x;
        }
        void print() { cout << "x = " << x << endl; }
};
```

```
int main()
{
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}
```

Output: 20

Note: For constructors, initializer list can also be used when parameter name is same as member’s name.

‘this’ pointer in C++

2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
// Some processing
return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

```
class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test &setX(int a) { x = a; return *this; }
    Test &setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1(5, 5);

    // Chained function calls. All calls modify the same object
    // as the same object is returned by reference

    obj1.setX(10).setY(20);

    obj1.print();
    return 0;
}
```

Output: x = 10 y = 20

Scope Resolution Operator vs this pointer in C++

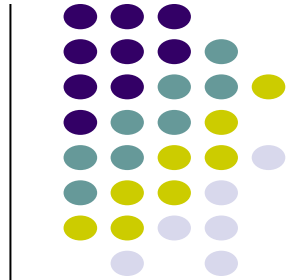
Scope resolution operator is for accessing static or class members and **this pointer** is for accessing object members when there is a local variable with the same name.

```
// C++ program to show that local parameters hide
// class members
#include <iostream>
using namespace std;

class Test {
        int a;

public:
        Test() { a = 1; }

        // Local parameter 'a' hides class member 'a'
        void func(int a) { cout << a; }
};
```

```
// Driver Code
int main()
{
        Test obj;
        int k = 3;
        obj.func(k);
        return 0;
}
```

O/P

3



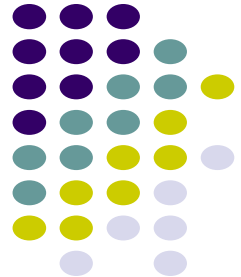
```
class Test {
        int a;

public:
        Test() { a = 1; }

        // Local parameter 'a' hides object's member
        // 'a', but we can access it using this.
        void func(int a) { cout << this->a; }
};
```

```
// Driver code
int main()
{
        Test obj;
        int k = 3;
        obj.func(k);
        return 0;
}
```

Output: 1

```
// C++ program to show that scope resolution operator can
be
// used to access static members when there is a local
// variable with same name

class Test {
        static int a;

public:
        // Local parameter 'a' hides class member
        // 'a', but we can access it using ::
        void func(int a) { cout << Test::a; }
};

// In C++, static members must be explicitly defined
// like this
int Test::a = 1;

// Driver code
int main()
{
        Test obj;
        int k = 3;
        obj.func(k);
        return 0;
}
```

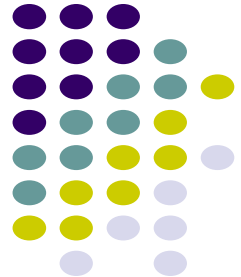
Output: 1



References in C++

References in Funtions

```
int* first (int* x)
{ (*x++);
  return x;   // SAFE, x is outside this scope
}

int& second (int& x)
{ x++;
  return x;   // SAFE, x is outside this scope
}

int& third ()
{ int q;
  return q;   // ERROR, scope of q ends here
}

int& fourth ()
{ static int x;
  return x;   // SAFE, x is static, hence lives till the end.
}

int main()
{
 int a=0;
 first(&a);   // UGLY and explicit
 second(a);   // CLEAN and hidden
}
```



Pointers to class members

Defining a pointer of class type

```
class Simple
{
 public:
 int a;
};

int main()
{
 Simple obj;
 Simple* ptr;   // Pointer of class type
 ptr = &obj;

 cout << obj.a;
 cout << ptr->a;  // Accessing member with
pointer
}
```

Pointer to Data Members of class

Syntax for Declaration :

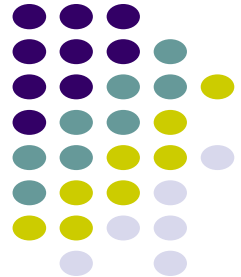
datatype class_name :: *pointer_name ;

Syntax for Assignment :

pointer_name = &class_name :: datamember_name ;

Both declaration and assignment can be done in a single statement too.

datatype class_name::*pointer_name = &class_name::datamember_name;



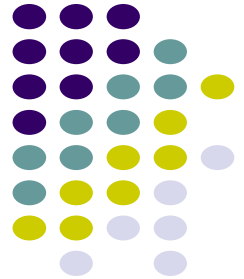
Pointers to class members

Pointer to Data Members of class

```
class Data
{
 public:
 int a;
 void print() { cout << "a is "<< a; }
};

int main()
{
 Data d, *dp;
 dp = &d;     // pointer to object

 int Data::*ptr=&Data::a;   // pointer to data member 'a'
 d.*ptr=10;
 d.print();

 dp->*ptr=20;
 dp->print();
}
```

Output :

a is 10 a is 20



Types of Member Functions

1.Simple functions
2.Static functions
3.Const functions
4.Inline functions
5.Friend functions

Static Function

```
class X
{
 public:
 static void f(){};
};

int main()
{
 X::f();   // calling member function directly with
class name
}
```

These functions cannot access ordinary data members and member functions, but only static data members and static member functions.
It doesn't have any "this" keyword which is the reason it cannot access ordinary members

Const Member functions

When used with member function, such member functions can never modify the object or its related data members.

//Basic Syntax of const Member Function

```
void fun() const {}
```

Inline functions
All the member functions defined inside the class definition are by default declared as Inline

Const member functions in C++

Constant member functions are those functions that are denied permission to change the values of the data members of their class. To make a member function constant, the keyword const is appended to the function prototype and also to the function definition header.

Important Points

•When a function is declared as const, it can be called on any type of object, const object as well as non-const objects.

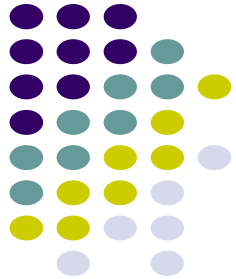
•Whenever an object is declared as const, it needs to be initialized at the time of declaration. however, the object initialization while declaring is possible only with the help of constructors.

•A function becomes const when the const keyword is used in the function's declaration. The idea of const functions is not to allow them to modify the object on which they are called.

•It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided.


constFunc1.cpp


consFunc2.cpp



Types of Member Functions

Friend functions

Friend functions are actually not class member function. Friend functions are made to give **private** access to non-class functions.

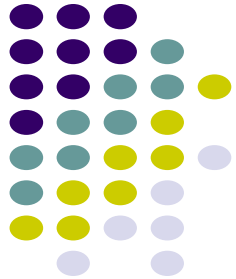
You can declare a global function as friend, or a member function of other class as friend.

```
class WithFriend
{
 int i;
 public:
 friend void fun(); // Global function as friend
};
void fun()
{
 WithFriend wf;
 wf.i=10;  // Access to private data member
 cout << wf.i;
}
int main()
{
fun(); //Can be called directly
}
```

We can also make an entire class as friend class

```
class Other
{
 void fun();
};

class WithFriend
{
 private:
 int i;
 public:
 void getdata();  // Member function of class WithFriend
 friend void Other::fun();   // making function of class Other as friend here
 friend class Other;  // making the complete class as friend
};
```

When we make a class as friend, all its member functions automatically become friend functions.
Friend Functions is a reason, why C++ is not called as a pure Object Oriented language. Because it violates the concept of Encapsulation.



Types of Member Functions

Friend functions

Friend functions are actually not class member function. Friend functions are made to give **private** access to non-class functions.

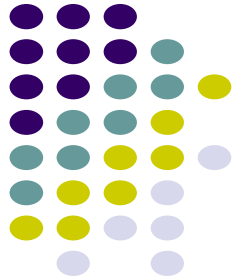
You can declare a global function as friend, or a member function of other class as friend.

```
class WithFriend
{
 int i;
 public:
 friend void fun(); // Global function as friend
};
void fun()
{
 WithFriend wf;
 wf.i=10;  // Access to private data member
 cout << wf.i;
}
int main()
{
fun(); //Can be called directly
}
```

We can also make an entire class as friend class

```
class Other
{
 void fun();
};

class WithFriend
{
 private:
 int i;
 public:
 void getdata();  // Member function of class WithFriend
 friend void Other::fun();   // making function of class Other as friend here
 friend class Other;  // making the complete class as friend
};
```

When we make a class as friend, all its member functions automatically become friend functions.
Friend Functions is a reason, why C++ is not called as a pure Object Oriented language. Because it violates the concept of Encapsulation.



Function Overloading

If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

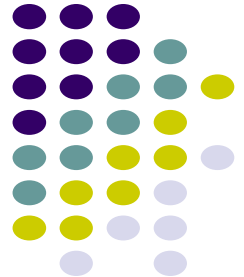
Ways to overload a function

1. By changing number of Arguments.
2. By having different types of argument.

```
int sum (int x, int y)
{
 cout << x+y;
}

int sum(int x, int y, int z)
{
 cout << x+y+z;
}

int main()
{
sum (10,20);  // sum() with 2 parameter will be called

sum(10,20,30);  //sum() with 3 parameter will be called
}
```



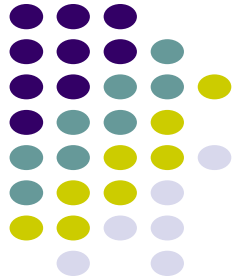
```
int sum(int x,int y)
{
 cout<< x+y;
}

double sum(double x,double y)
{
 cout << x+y;
}

int main()
{
 sum (10,20);
 sum(10.5,20.5);
}
```



Function Overloading

Default Arguments

```
sum(int x,int y=0)
{
 cout << x+y;
}

int main()
{
 sum(10);
 sum(10,0);
 sum(10,10);
}
```



Inheritance

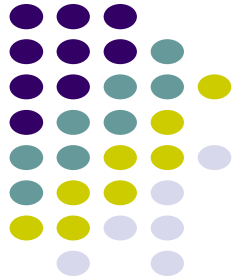
Inheritance is the capability of one class to acquire properties and characteristics from another class.

The class whose properties are inherited by other class is called the Parent or Base or Super class.

And, the class which inherits properties of other class is called Child or Derived or Sub class.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

NOTE : All members of a class except Private, are inherited



Purpose of Inheritance

1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

Basic Syntax of Inheritance

class Subclass_name : access_mode Superclass_name

While defining a subclass like this, the super class must be already defined or at least declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, privtate or protected.

Inheritance

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

class Subclass : public Superclass

Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass // By default its private inheritance

Inheritance

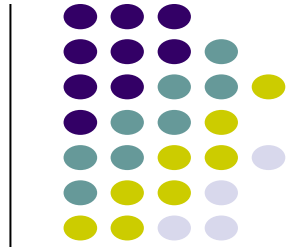
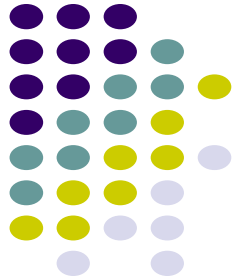


Table showing all the Visibility Modes

	Derived Class	Derived Class	Derived Class
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Inheritance

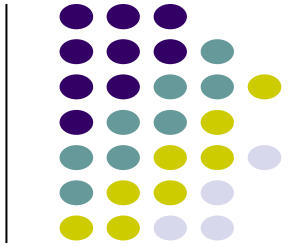


Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

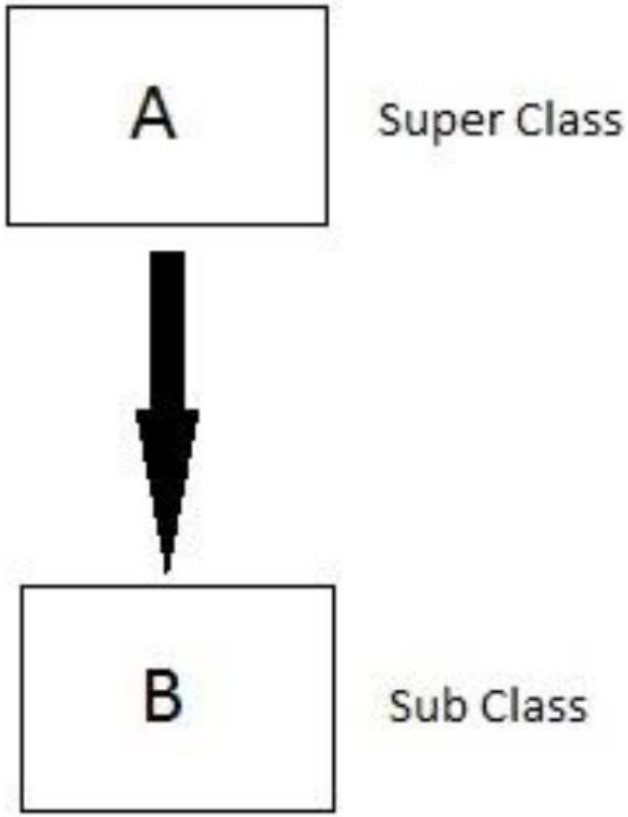
1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

Inheritance



Single Inheritance

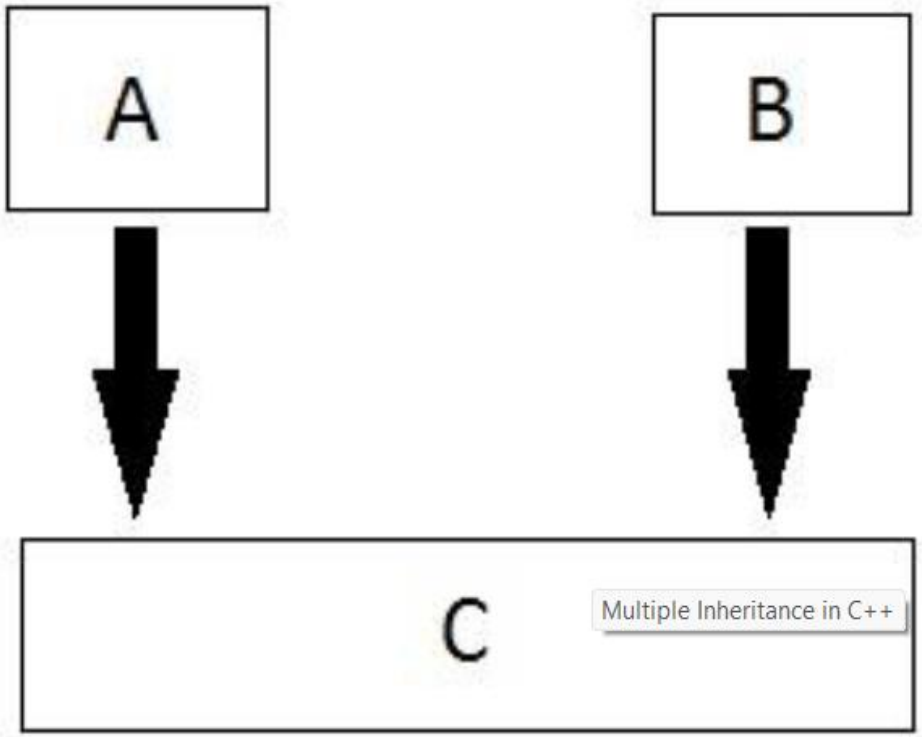
In this type of inheritance one derived class inherits from only one base class. It is the simplest form of Inheritance.



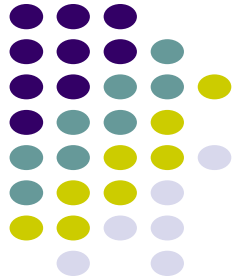
Inheritance

Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes

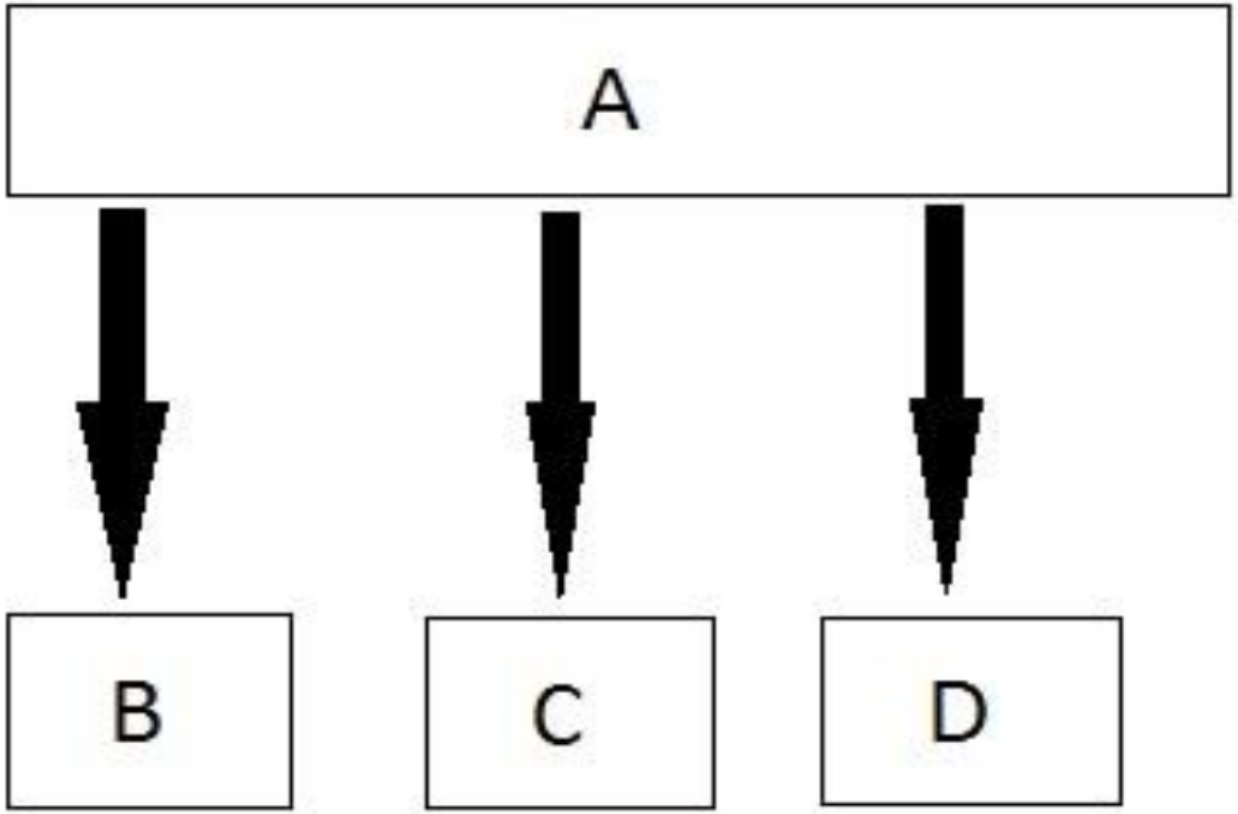


Inheritance

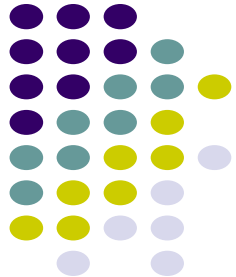


Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.

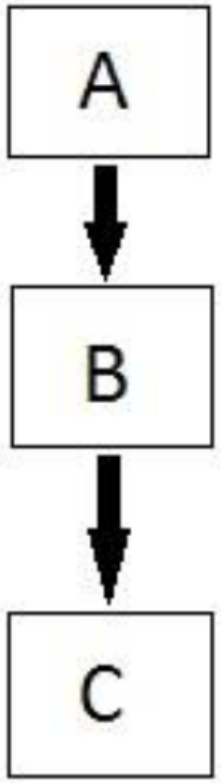


Inheritance

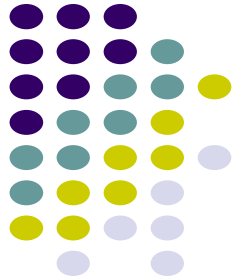


Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.

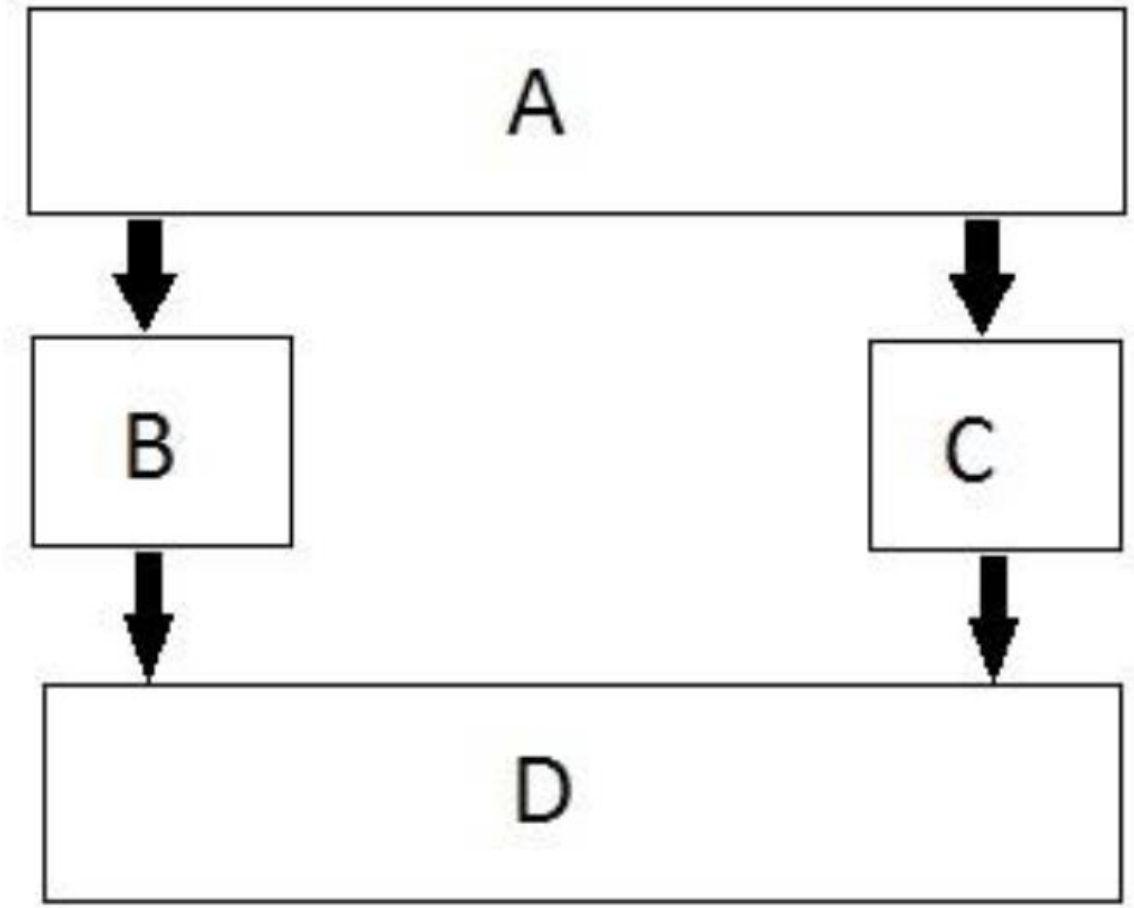


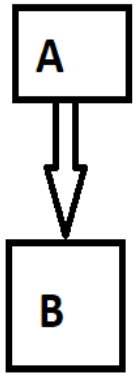
Inheritance



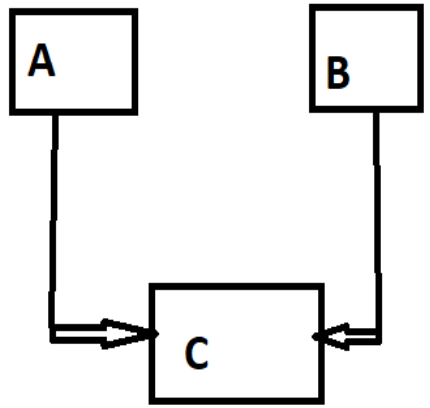
Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.

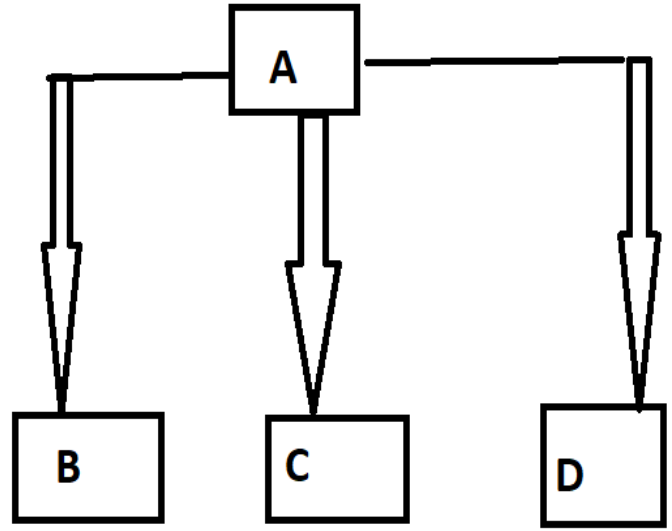




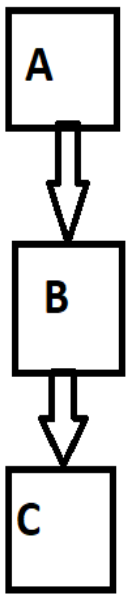
Single Inheritance



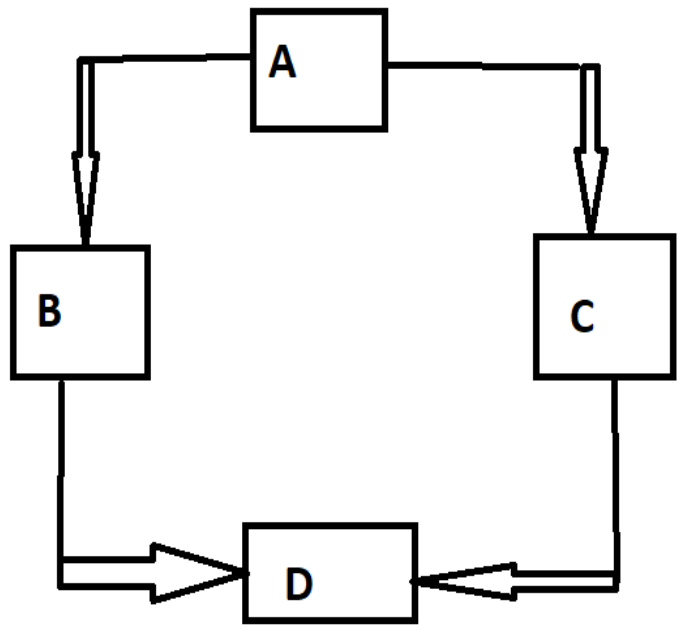
Multiple Inheritance



Hierarchical Inheritance

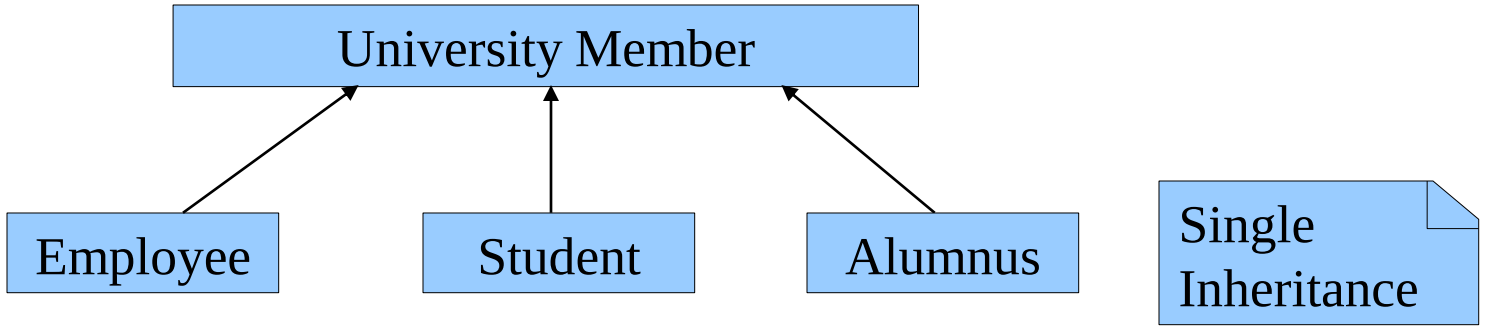


MultiLevel Inheritance

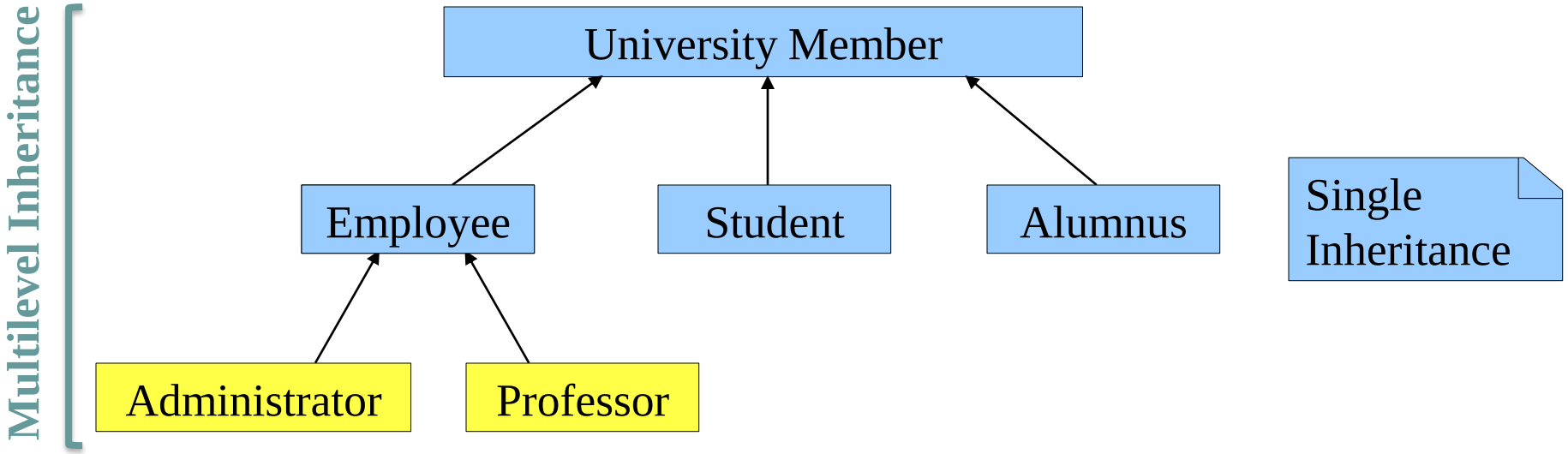


Hybrid/Multipath Inheritance

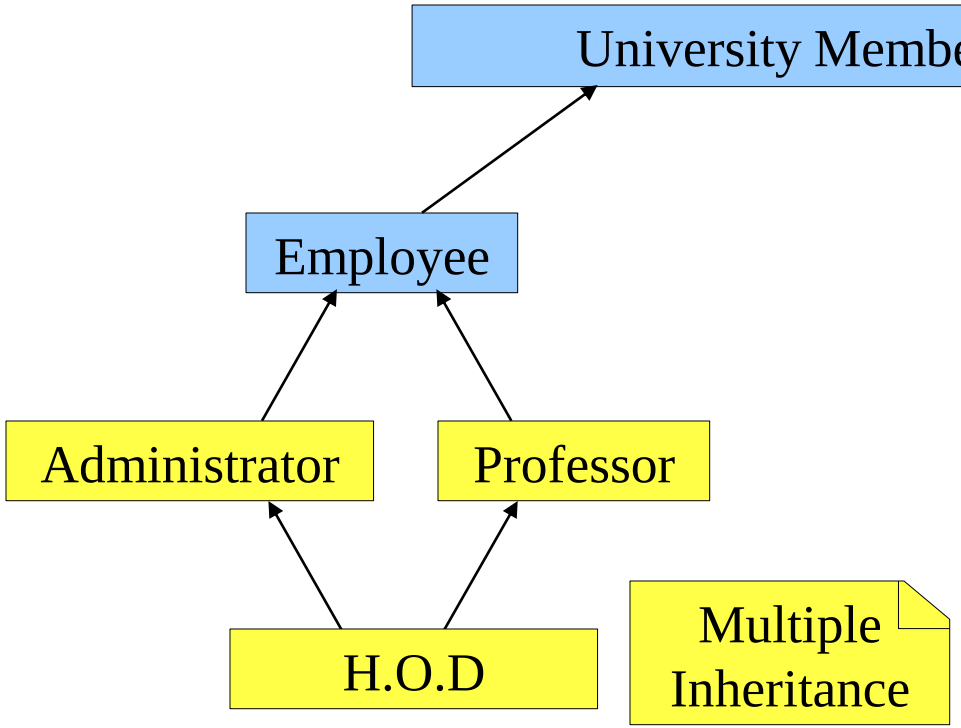
Class Hierarchy (Example)



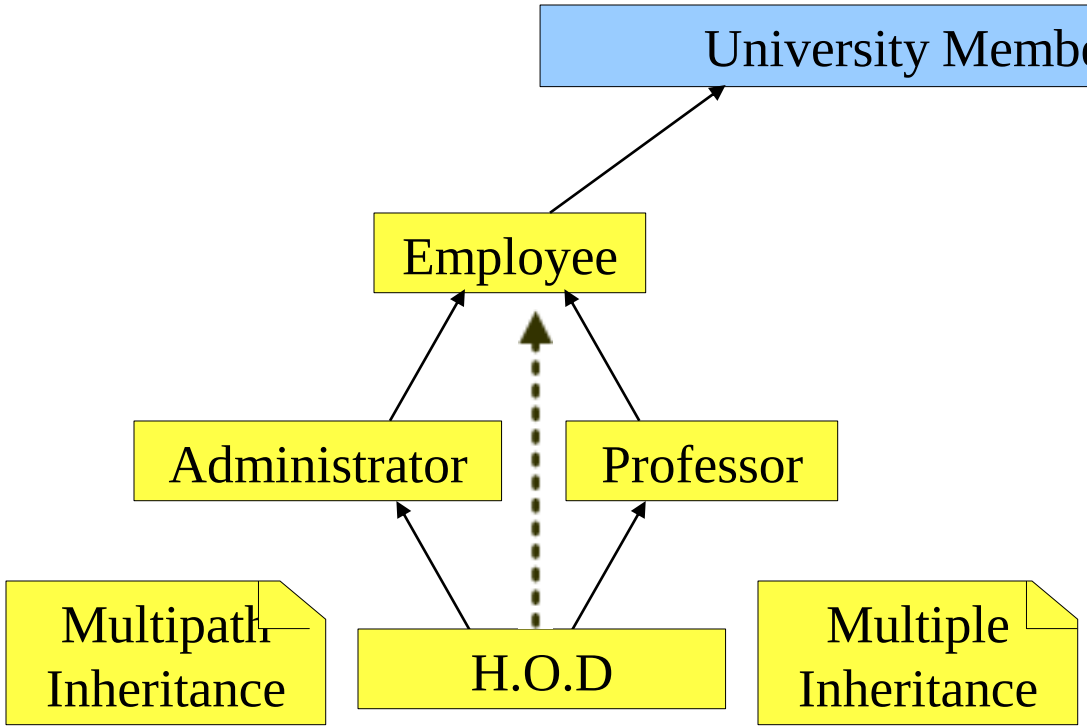
Class Hierarchy (Example)



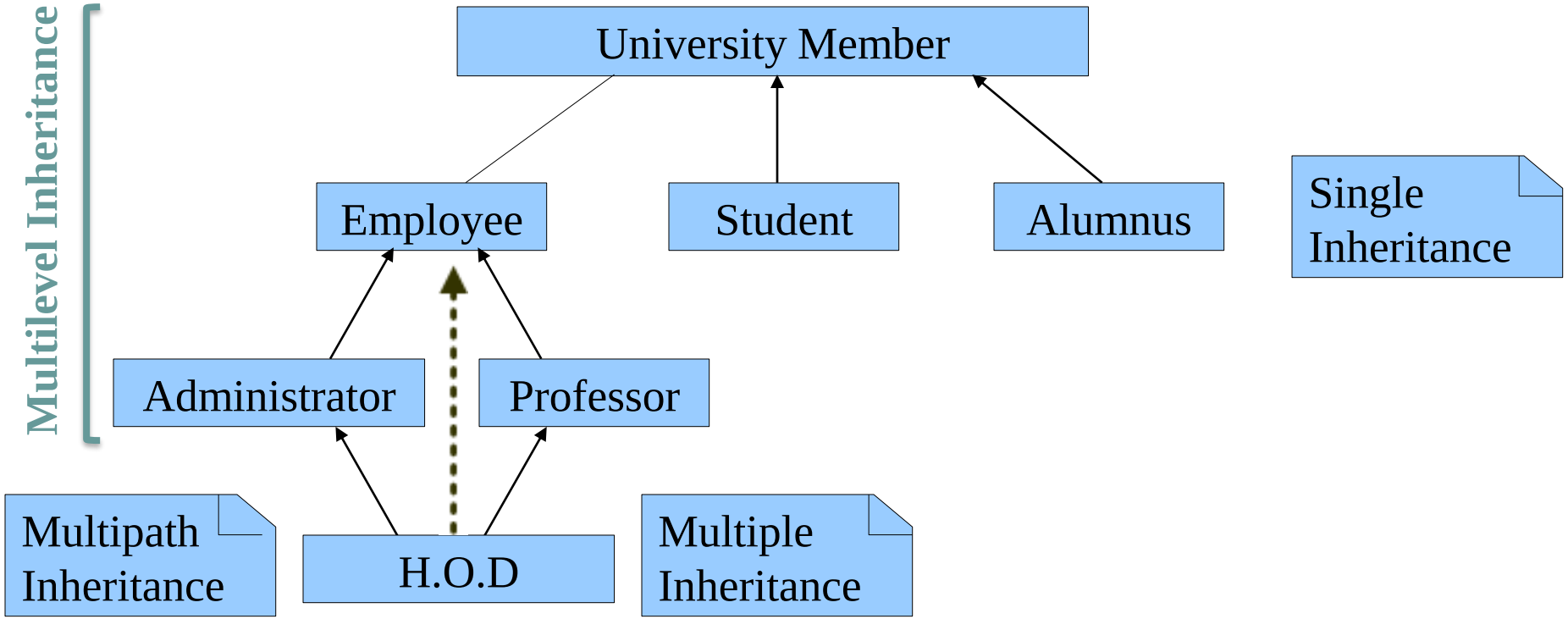
Class Hierarchy (Example)



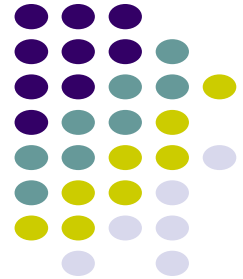
Class Hierarchy (Example)



Class Hierarchy (Example)



Example - 1



```
class B //base class
{
  int x;
protected:
  int y;
public:
  int z;
};
```

```
Class D : public B
{
public:
  D()
  {
    x = 5;//Error
    y = 6;
    z = 7;
  }
  void myfunc ()
  {
    cout<<y<<z<<endl;
  }
};
void main
{
  D objD;

  objD.myfunc ();
}
```

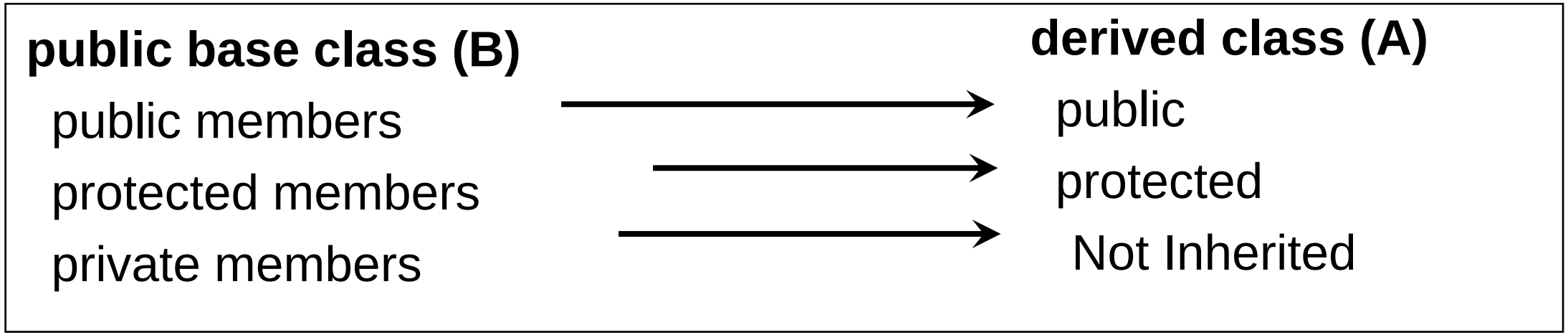
Q: Point out the errors, if any?

A: Compiler Error: Cannot access private member declared in class B

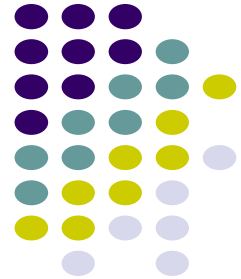
Inheritance & Member accessibility

Public Inheritance

```
class A : public B
{       // Class A now inherits the members of Class B
        // with no change in the “access specifier” for
       // the inherited members
}
```



Example - 2



```
class B //base class
{
protected:
  int y;
public:
  int z;
};
```

```
Class D : protected B
{
public:
  D()
  {
    y = 6;
    z = 7;
  }
  void myfunc ()
  {
    cout<<y<<z<<endl;
  }
};
void main()
{
  D objD;

  objD.myfunc ();
}
```

Q: What is the output?

A: 67

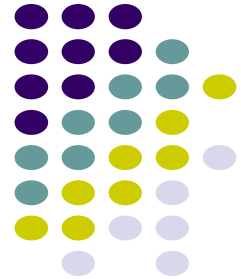
Inheritance & Member accessibility

Protected Inheritance

```
class A : protected B
{       // Class A now inherits the members of Class B
        // with public members “converted” to protected
}       // but no other changes to the inherited members
```

public base class (B)		**derived class (A)**
public members	→	protected
protected members	→	protected
private members	→	Not Inherited

Example - 3



```
class B //base class
{
  int x;
protected:
  int y;
  void myfunc ()
  {
    cout<<x;
  }
public:
  int z;
  B()
  {
    x = 5;
  }
};
```

```
Class D : private B
{
public:
  D()
  {
    y = 6;
    z = 7;
  }
  void myfunc ()
  {
    B::myfunc();
    cout<<y<<z<<endl;
  }
};
void main()
{
  D objD;

  objD.myfunc ();
}
```

Q: What is the output?

A: 567

Inheritance & Member accessibility

Private Inheritance

```
class A : private B
{       // Class A now inherits the members of Class B
        // with public & protected  members “converted” to
}       // private
```

public base class (B)		derived class (A)
public members	→	private
protected members	→	private
private members	→	Not Inherited

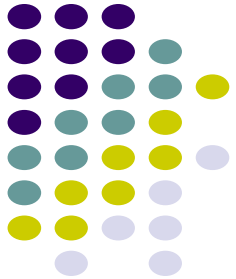
Q - 01

```
class A //base class
{
public:
  int i;
};
```

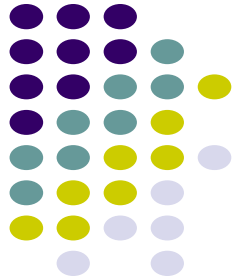
Q: Point out the Errors, if any?

A: 'i' not accessible because 'B' uses 'private' to inherit from 'A'

```
class B : private A
{
public:
  B()
  {
    i = 277246;
  }
};

class C : public B
{
}c;

void main ()
{
  cout<<c.i; //Error
}
```



Q - 02



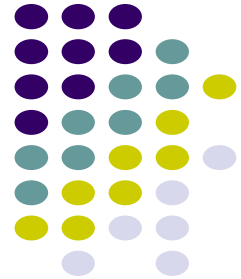
```
class B  //base class
{
    int x;
protected:
    int y;
public:
    int z;
    B():x(2),y(3),z(4){}
    void myfunc()
    {
        cout<<x<<y<<z;
    }
};
```

Q: What is the output?

A: 234567

```
class D: private B
{
public:
    int x;
private:
    int y;
protected:
    int z;
Public:
    D():x(5),y(6),z(7){}
    void myfunc()
    {
      B::myfunc();
      cout<<x<<y<<z;
    }
};
int main ()
{
    D objD;
    objD.myfunc ();
}
```

Problem Statement - 1

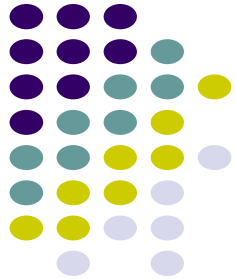


```
class B //base class
{
  int privateB;
  protected:
    int protectedB;
  public:
    int publicB;
    int getBPrivate()
    {
      return privateB;
    }
};
```

```
Class D : B
{
public:
     int publicD;
     void myfunc()
     {
       int a;
       a = privateB;
       a = getBprivate();
       a = protectedB;
       a = publicB;
     }
};
void main()
{
  D objd;
  int a;
  a = objd.publicB;
  a = objd.protectedB;
}
```

Q: Point out the errors, if any?

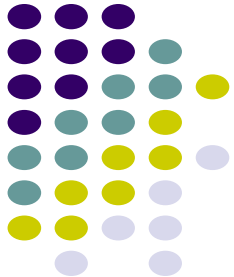
Problem Statement - 1



```
class B //base class
{
  int privateB;
  protected:
    int protectedB;
  public:
    int publicB;
    int getBPrivate()
    {
      return privateB;
    }
};
```

```
Class D : B
{
public:
     int publicD;
     void myfunc()
     {
       int a;
       a = privateB; //Error
       a = getBprivate();
       a = protectedB;
       a = publicB;
     }
};
void main
{
  D objd;
  int a;
  a = objd.publicB; //Error
  a = objd.protectedB; //Error
}
```

Conclusion

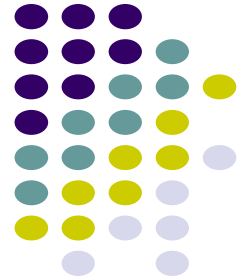


- The base class must have a default constructor

- Defining a default constructor is always a good practice

- If the base class does not have a default constructor and has an parameterised constructor, they must be explicitly invoked, otherwise compiler generates an error

Example

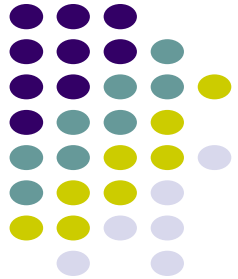


```
class A
{
public:
  void func()
  {
    cout<<"A::func()";
  }
};
class B : public A
{
};
class C : public A
{
};
class D : public B, public C
{
};
```

```
void main ()
{
    D objd;

    objd.func();
}
```

Q: Is there any problem with this program?
A: D::func1() is ambiguous

Example – (Contd...)



```
class A
{
public:
  void func()
  {
    cout<<"A::func()";
  }
};
class B : public virtual A
{
    //body of class B
};
class C : public virtual A
{
    //body of class C
};
class D : public B, public C
{
    //body of class D
};
```

```
void main ()
{
    D objd;

    objd.func();
}
```