Object Oriented Programming with C++

Introduction and Course details

Lecturer : Astha Sharma

CHAPTER 4

Object Inheritance and Reusability

9 hrs

Introduction to Inheritance

Inheritance

•Inheritance is a process of organizing information in a hierarchical form.

•Through inheritance, we can create new class, called <u>derived class</u>, from existing class called <u>base class.</u>

•The derived class inherits all the features of the base class and can add new extra features.

•It allows new classes to build from existing and less specialized class instead of writing from the scratch.

- Inheritance is an essential part of OOP.

- It provides code reusability.

- Reusability is one of the striking features of C++.

- We don't have to write code from the scratch each and every time while writing similar type of construct.

- The reused code is already tested , more reliable and less error prone.

- Also, it saves time, effort and ultimately money.

Base and derived class
or
Super and Sub class

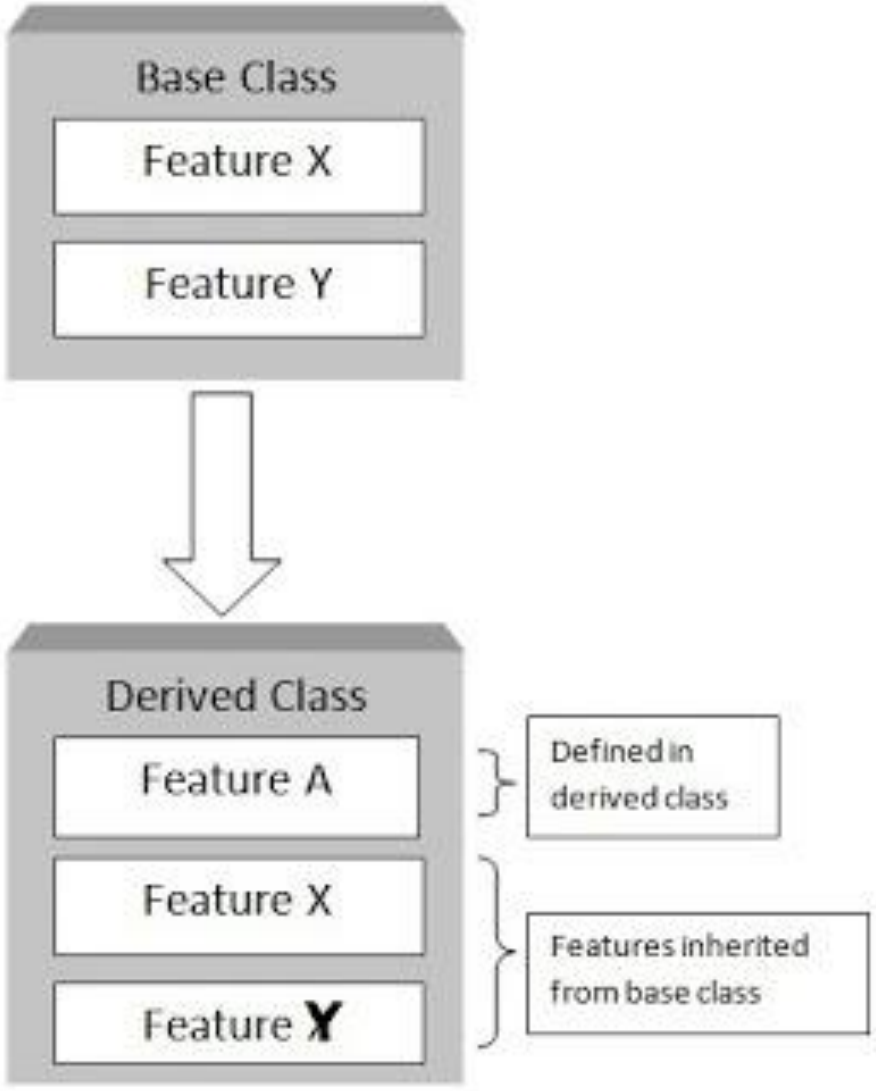
Base and Derived Classes

•In inheritance, the existing classes that are used to derive new classes are called base classes and the new classes derived from existing classes are called derived classes.

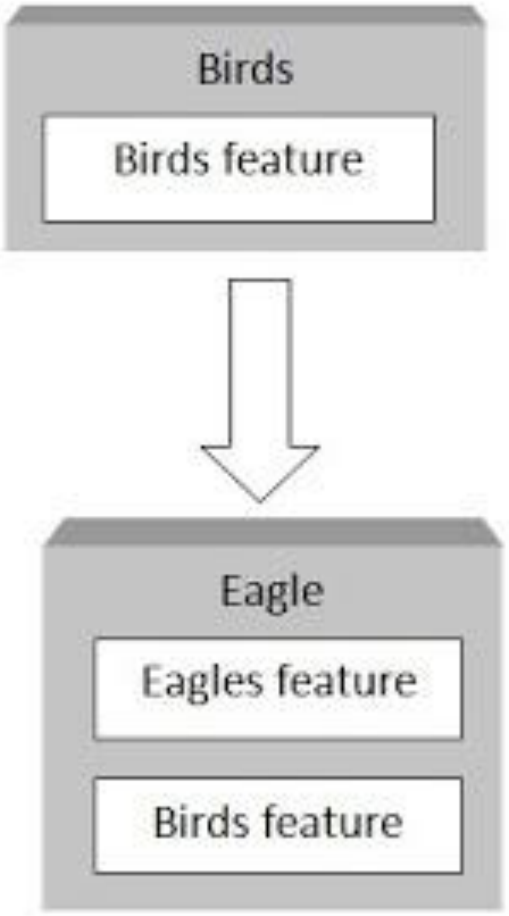
•When a class is derived from a base class, the derived class inherits all the characteristics of base class and can add new features as refinements and improvements.

•Base class : ancestor class : parent class : super class

•Derived class : descendent class : child class : sub class



Example of inheritance



Derived class definition

•A class derivation list names one or more base classes and has the form –

class derived-class: access-specifier base-class

where

•access-specifier is one of <u>public, protected, or private</u>,

•base-class is the name of a previously defined class.

•If the access-specifier is not used, then it is private by default.

Example of Derived class and Base class

```
#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;
 }
```

Forms of polymorphism and their implementation in C++,

Polymorphism in real life

As we all know, polymorphism is taking many forms. We can relate it with the example of a man, who is a husband, an employee at an office, son, father hence, man can take different forms.

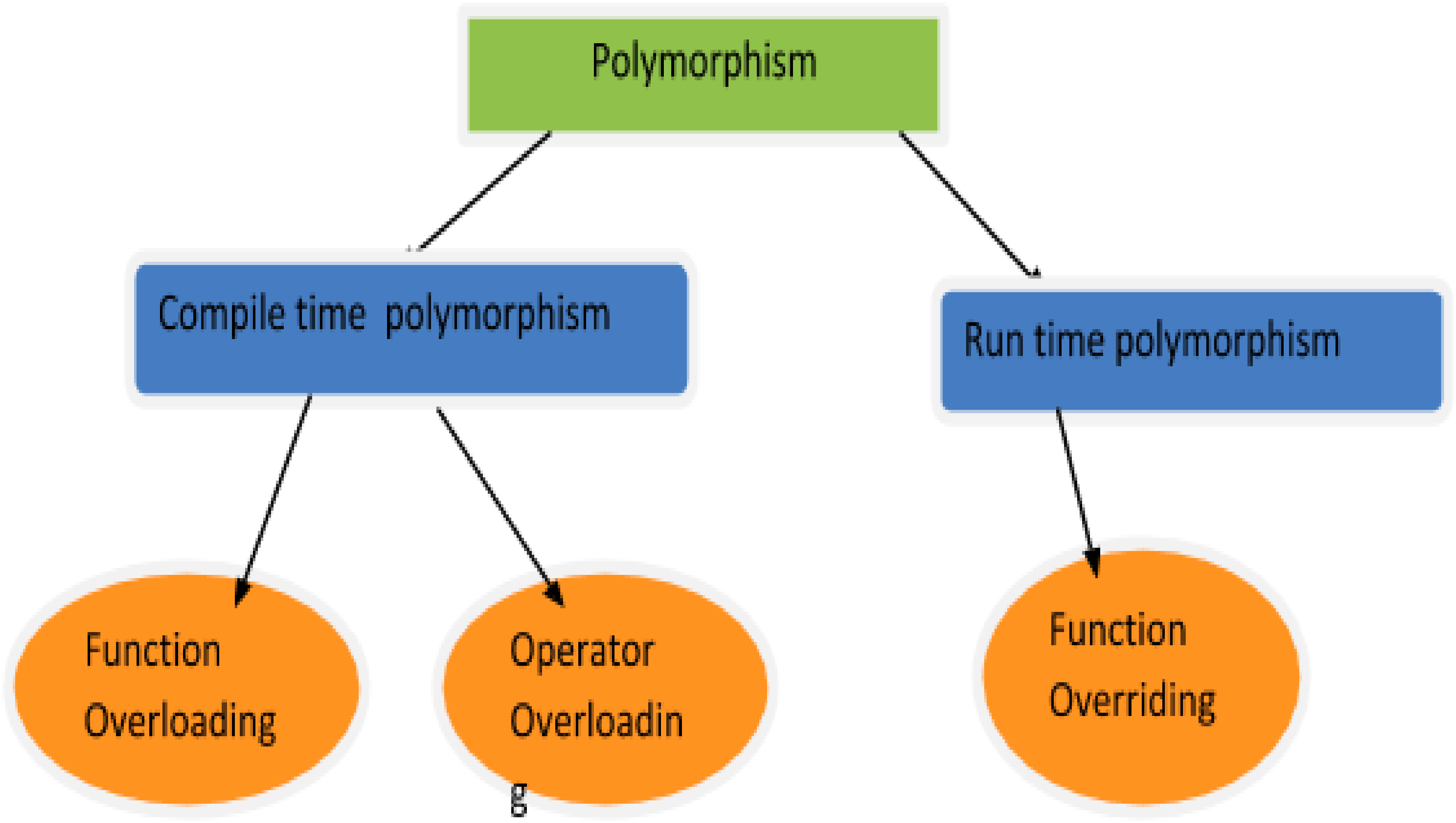
Suppose a teacher at a school at teaching different subjects in different classes like he can be an English teacher for class 9 and the same teacher can be a science class for class 10 hence taking various forms at different times.

Polymorphism in C++

Polymorphism is a feature of OOP that allows the object to behave differently in different conditions. In C++ we have two types of polymorphism:

1) <u>Compile time Polymorphism</u> – This is also known as static (or early) binding.

2) <u>Runtime Polymorphism</u> – This is also known as dynamic (or late) binding.


Polymorphism
Compile time polymorphism
Run time polymorphism
Function Overloading
Operator Overloadin g
Function Overriding

1) Compile time Polymorphism

Function overloading and Operator overloading are perfect example of Compile time polymorphism.

Function overloading example:

In this example, we have two functions with same name but different number of arguments. Based on how many parameters we pass during function call determines which function is to be called, this is why it is considered as an example of polymorphism because in different conditions the output is different. Since, the call is determined during compile time thats why it is called compile time polymorphism.

```
#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)<<endl;
return 0;
}
```

Output:

```
Output: 30
Output: 66
```

Operator overloading example:

•In C++, we can make operators work for user-defined classes. This means C++ can provide the operators with a special meaning for a data type; this ability is known as operator overloading.

•We can overload most of the inbuilt operators in c++ such as +,-, and many more defined operator

•In this example we take two integer values feet and inches and negate it respectively using – operator.

•Syntax:
Class_name operator symbol (information)
Where;
Operator is keyword,
Symbol is +,-,etc,
Information is info for the operator, it can or cannot exist.

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

 class Distance
{
private:      int feet;             // 0 to infinite
             int inches;            // 0 to 12
public:
 // required constructors
            Distance() {        feet = 0;
                                 inches = 0;}
             Distance(int f, int i)
            {         feet = f;
                       inches = i;       }
// method to display distance
            void displayDistance()
            {cout << "F: " << feet << " I:" << inches<<endl;    }

// overloaded minus (-) operator
            Distance operator - ()
            {         feet = -feet;         inches = -inches;
            return Distance(feet, inches);      }
};
```

```
int main()
{
Distance D1(11, 10), D2(-5, 11);
-D1;                       // apply negation
D1.displayDistance();     // display D1
-D2;                       // apply negation
D2.displayDistance();     // display D2
return 0;
}
```

OUTPUT:

```
F: -11  I:-10
F: 5  I:-11
```

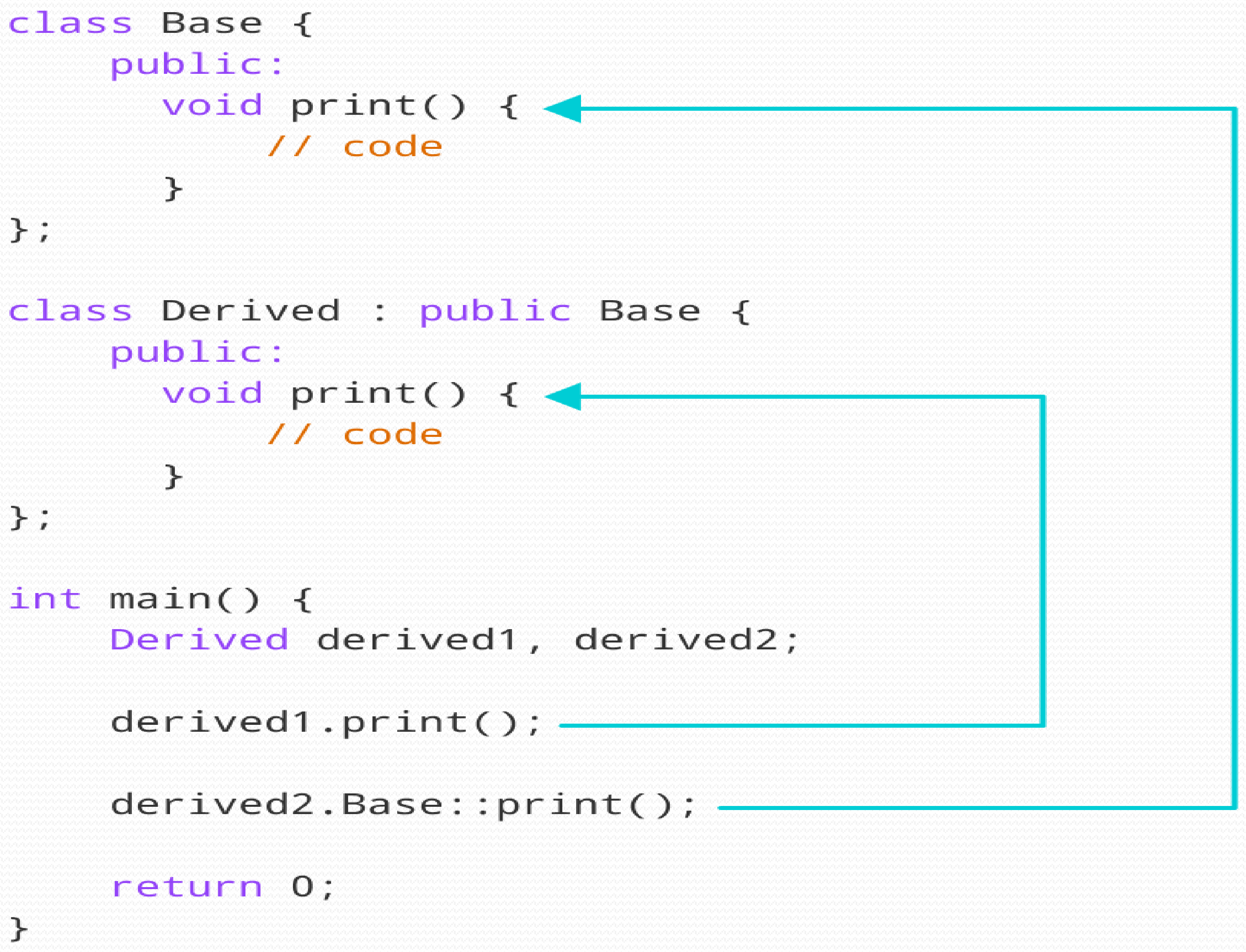
2) Runtime Polymorphism

Function overriding is an example of Runtime polymorphism.

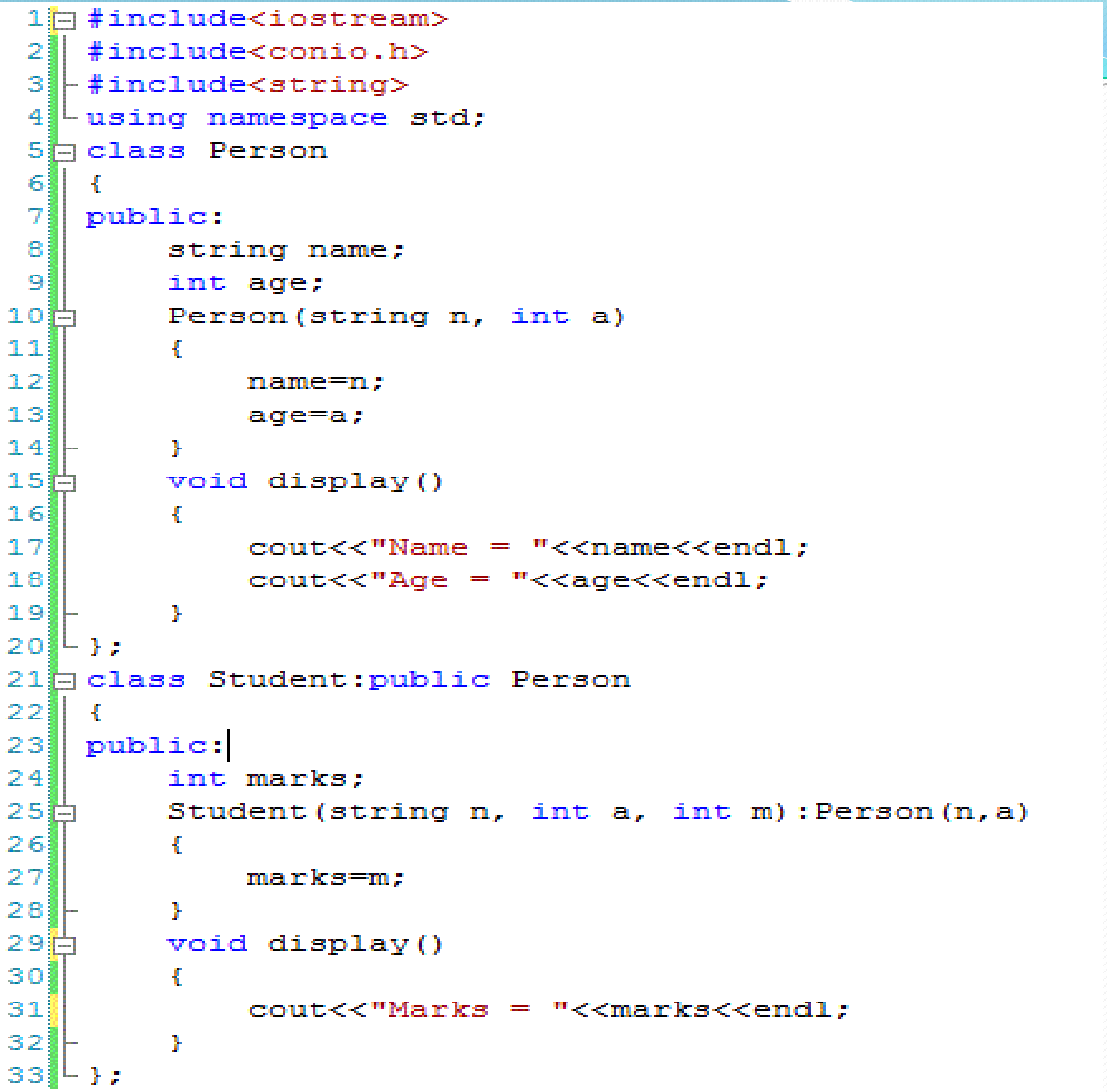
Function Overriding:
When child class declares a method, which is already present in the parent class then this is called function overriding, here child class overrides the parent class.
In case of function overriding we have two definitions of the same function, one is in parent class and one in child class. The call to the function is determined at **runtime** to decide which definition of the function is to be called, that's the reason it is called 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"<<end;}
 };
 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
```

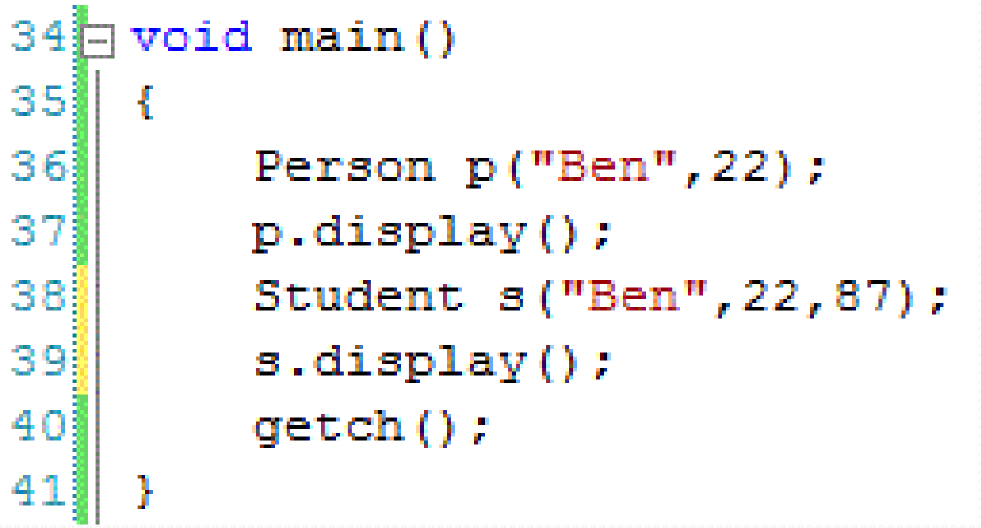


```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1, derived2;

    derived1.print();

    derived2.Base::print();

    return 0;
}
```



```
#include<iostream>
#include<conio.h>
#include<string>
using namespace std;
class Person
{
public:
	string name;
	int age;
	Person(string n, int a)
	{
		name=n;
		age=a;
	}
	void display()
	{
		cout<<"Name = "<<name<<endl;
		cout<<"Age = "<<age<<endl;
	}
};
class Student:public Person
{
public:
	int marks;
	Student(string n, int a, int m):Person(n,a)
	{
		marks=m;
	}
	void display()
	{
		cout<<"Marks = "<<marks<<endl;
	}
};
```

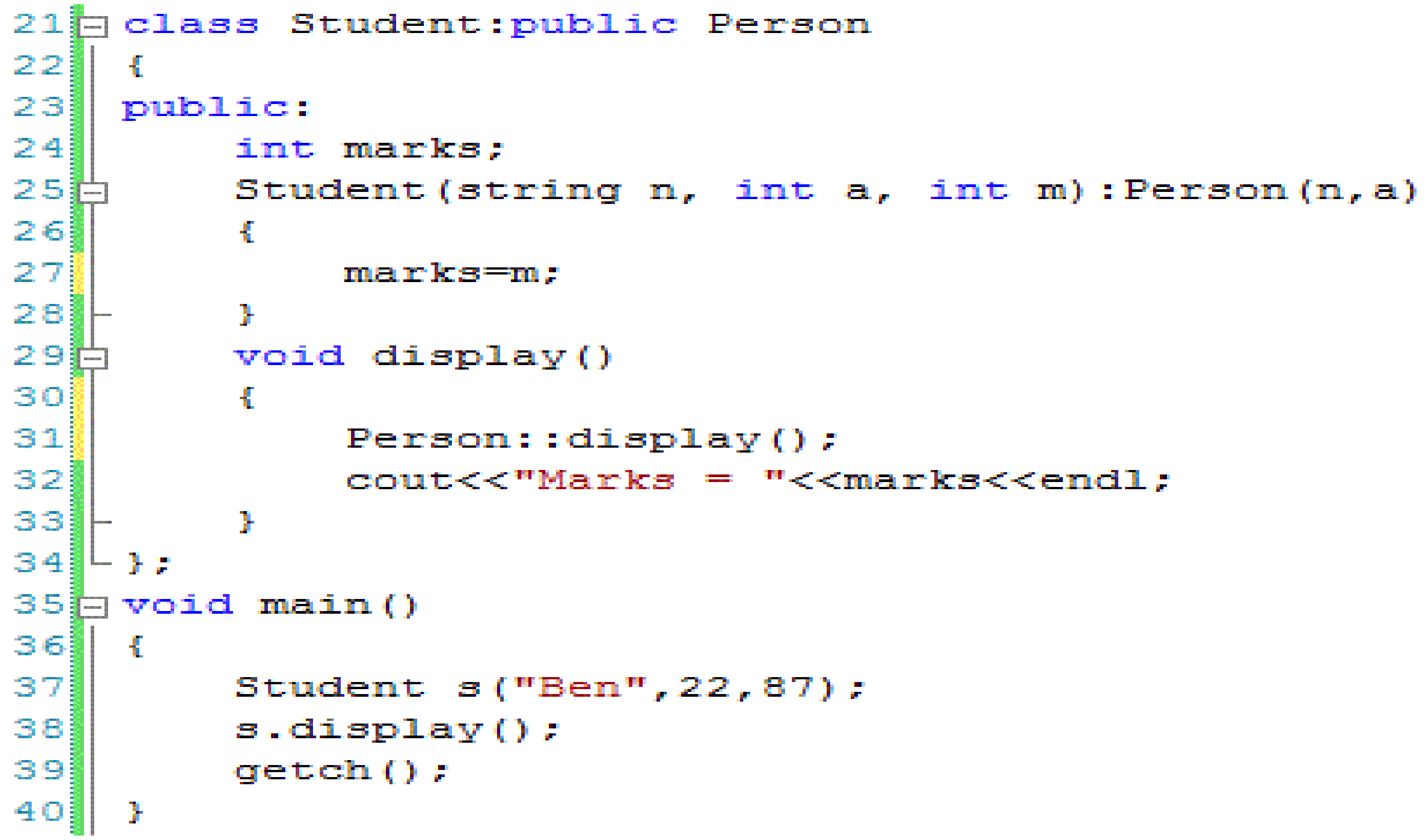
After overriding, the member function can be called in a number of ways.
For instance, we can create objects of class Person and of class Student.
Each object will call the corresponding member function display as shown in the following code.



```
34 void main()
35 {
36         Person p("Ben",22);
37         p.display();
38         Student s("Ben",22,87);
39         s.display();
40         getch();
41 }
```

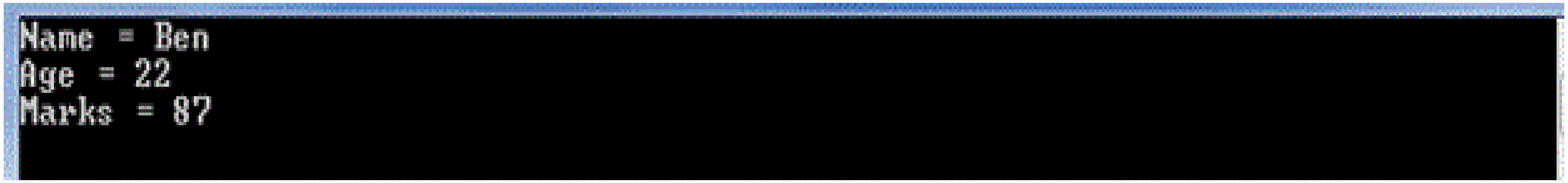
We can see that display function is called at line 37 and 39. At line 37, the member function defined in class Person is called and at line 39, the member function defined in class Student is called.

It may be required to call the overridden member function of parent class from the child class. For example when we call the display member function of child class then it should call the display member function of parent class first and then proceed with its own code. In our case if an object of Student class call the display member function of Student class then first the display member function of Person class should be called to display the data member like name and age. Once name and age are displayed then the marks of the student should be displayed. This can be achieved as below.



```
class Student:public Person
{
public:
        int marks;
        Student(string n, int a, int m):Person(n,a)
        {
                marks=m;
        }
        void display()
        {
                Person::display();
                cout<<"Marks = "<<marks<<endl;
        }
};
void main()
{
        Student s("Ben",22,87);
        s.display();
        getch();
}
```

We can see that at line 31, the display member function of Student class calls the display member function of its parent class Person first and the will proceed with its own code. As a result if we create an object of Student class (line 37) and call the display member function of Student class then both the member functions written in the Person class and the Student class will be called. If we run the above code then the following output will be produced.



```
Name = Ben
Age = 22
Marks = 87
```

It is important to note that the first two lines of the output come as a result of calling the display member function of Person class and the last line of the output comes from the display function written in the Student class.

Inheritance : merits and demerits

Advantages:

1. Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inherited class.
2. Reusability enhanced reliability. The base class code will be already tested and debugged.
3. As the existing code is reused, it leads to less development and maintenance costs.
4. Inheritance makes the sub classes follow a standard interface.
5. Inheritance helps to reduce code redundancy and supports code extensibility.
6. Inheritance facilitates creation of class libraries.

Disadvantages:-

1. Inherited functions work slower than normal function as there is indirection.
2. Improper use of inheritance may lead to wrong solutions.
3. Often, data members in the base class are left unused which may lead to memory wastage.
4. Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.

Composition and its implementation in C++

•A common technique used to create programs with various objects consists of declaring one object inside of another.

•Composition refers to the building of new classes by the assembly of existing classes.

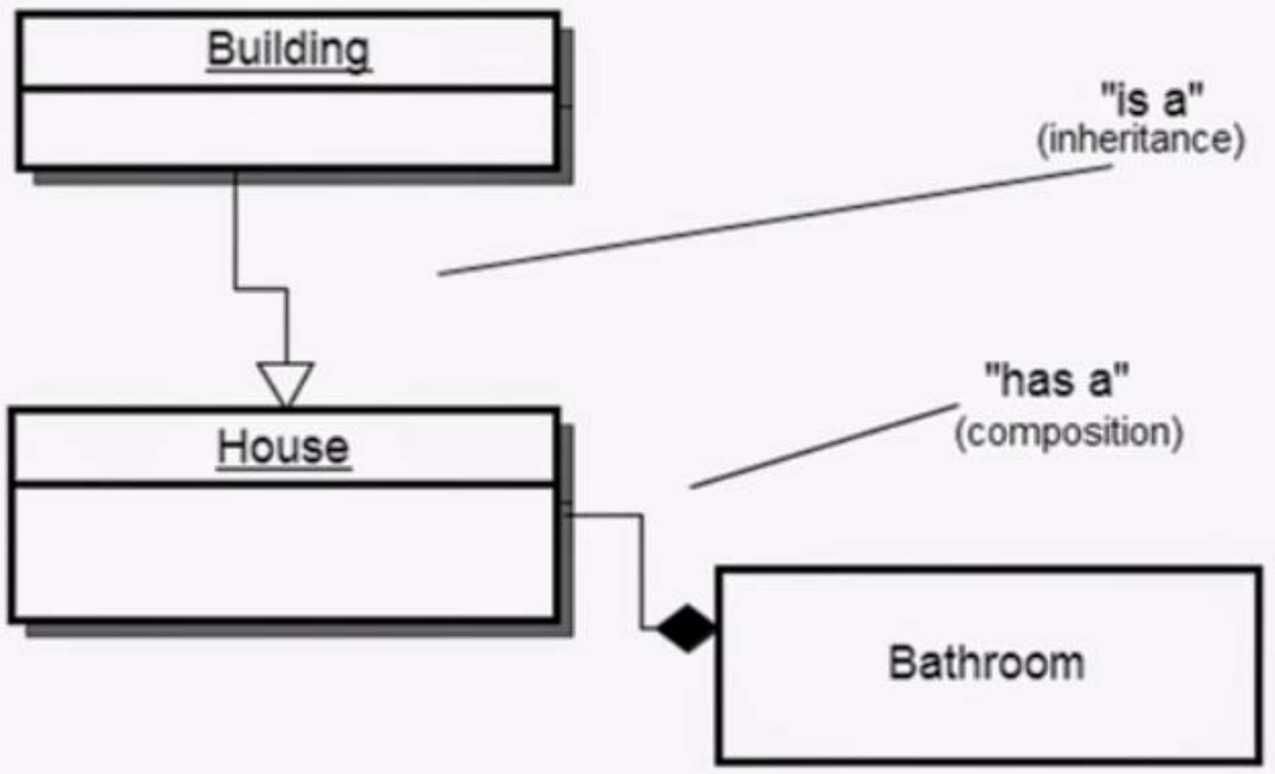
•Once you have or know the object to use, you can declare it like a regular member of the object.

•.Child object does not have any lifecycle and it dies whenever parent object dies.

•Composed object becomes a part of composer.

•Composed object cannot exist independently.

Composition vs. Inheritance



Composition

- Relatioship between objects, where one object owns, or *has* the other object
- Car *has or owns* Motor
- When Car is build, it's motor is built also
- When Car is destroyed it's motor is destroyed

```
#include <iostream>
#include <conio.h>
#include <string>
using namespace std;
class Engine
{
public:
    int power = 50;
};
class Car
{
private:
    float price;
    string color;
    string model;
public:
    Engine e;
    void setData(float p, string c, string m)
    {
        price = p;
        color = c;
        model = m;
    }
 void displayData()
    {
        cout << "Price :" << price << endl;
        cout << "Color :" << color << endl;
        cout << "Model :" << model << endl;
        cout << "Engine Power :" << e.power <<
"KW" << endl;
    }
};

int main()
{
    Car c;
    c.setData(2000000.00 , "Black" , "Hyundai
i10");
    c.displayData();
    return 0;
}
```

```
Output :
Price : 2000000.00
Color : Black
Model : Hyundai i10
Engine Power : 50 KW
```

The is-a rule and has-a rule

IS – A RULE

•In object-oriented programming, we can use inheritance when we know there is an "is a" relationship between a child and its parent class.

•Some examples would be:
A person *is a* human.
A cat *is an* animal.
A car *is a* vehicle.

•In each case, the child or subclass is a *specialized* version of the parent or super class. Inheriting from the super class is an example of code reuse.

HAS – A RULE

•In object-oriented programming, we can use composition in cases where one object "has" (or is part of) another object.

•Some examples would be:
A car *has a* battery (a battery *is part of* a car).
A person *has a* heart (a heart *is part of* a person).
A house *has a* living room (a living room *is part of* a house).

Composition and Inheritance contrasted

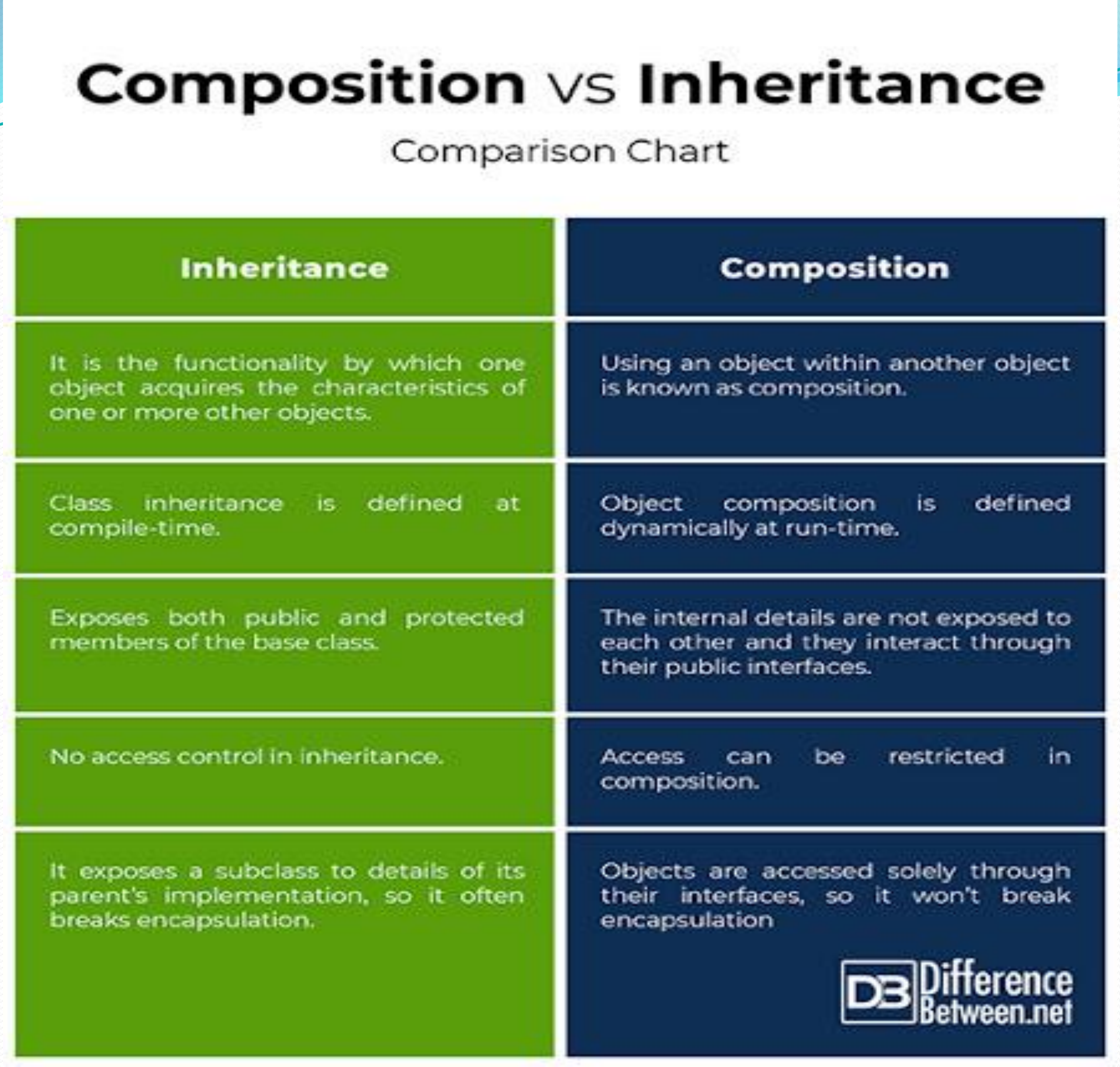
• Both **Is-A** and **Has-A** are relationships that are used in the C++ **code reusability**.

• **Is-A** relationship depends on inheritance and means that the child class is a type of parent class. in other words, **Is-A is a way of saying that this object is a type of that object.**

• For instance, Mammal **Is-A** Animal, Reptile **Is-A** Animal, Dog **Is-A** Mammal, and so on. Please note that "Dog **Is-A** Animal" as well.

• **Has-A** is known as a **composition** where an instance of a class has a reference to an instance of another class.

• For instance, a **cat has a tail,** a car has a door, and so on.



Composition vs Inheritance

Comparison Chart

Inheritance	Composition
It is the functionality by which one object acquires the characteristics of one or more other objects.	Using an object within another object is known as composition.
Class inheritance is defined at compile-time.	Object composition is defined dynamically at run-time.
Exposes both public and protected members of the base class.	The internal details are not exposed to each other and they interact through their public interfaces.
No access control in inheritance.	Access can be restricted in composition.
It exposes a subclass to details of its parent's implementation, so it often breaks encapsulation.	Objects are accessed solely through their interfaces, so it won't break encapsulation

DB DifferenceBetween.net

Software reusability

•The systematic reuse of the components as building blocks to create new system is called reusability.

•Software reuse is the process of implementing or updating software systems using existing software assets.

•Every programmer must abide the rule that the resources (memory, cpu, sw, etc.)are expensive and must be reused for maximum utilization of them.

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.

- All companies put their assets in the first priority and make sure that the available resources are used efficiently and effectively .

- Software engineering has been more focused on original development but it is now recognized that to achieve better software, more quickly and at lower cost, we need a design process that is based on systematic software reuse.

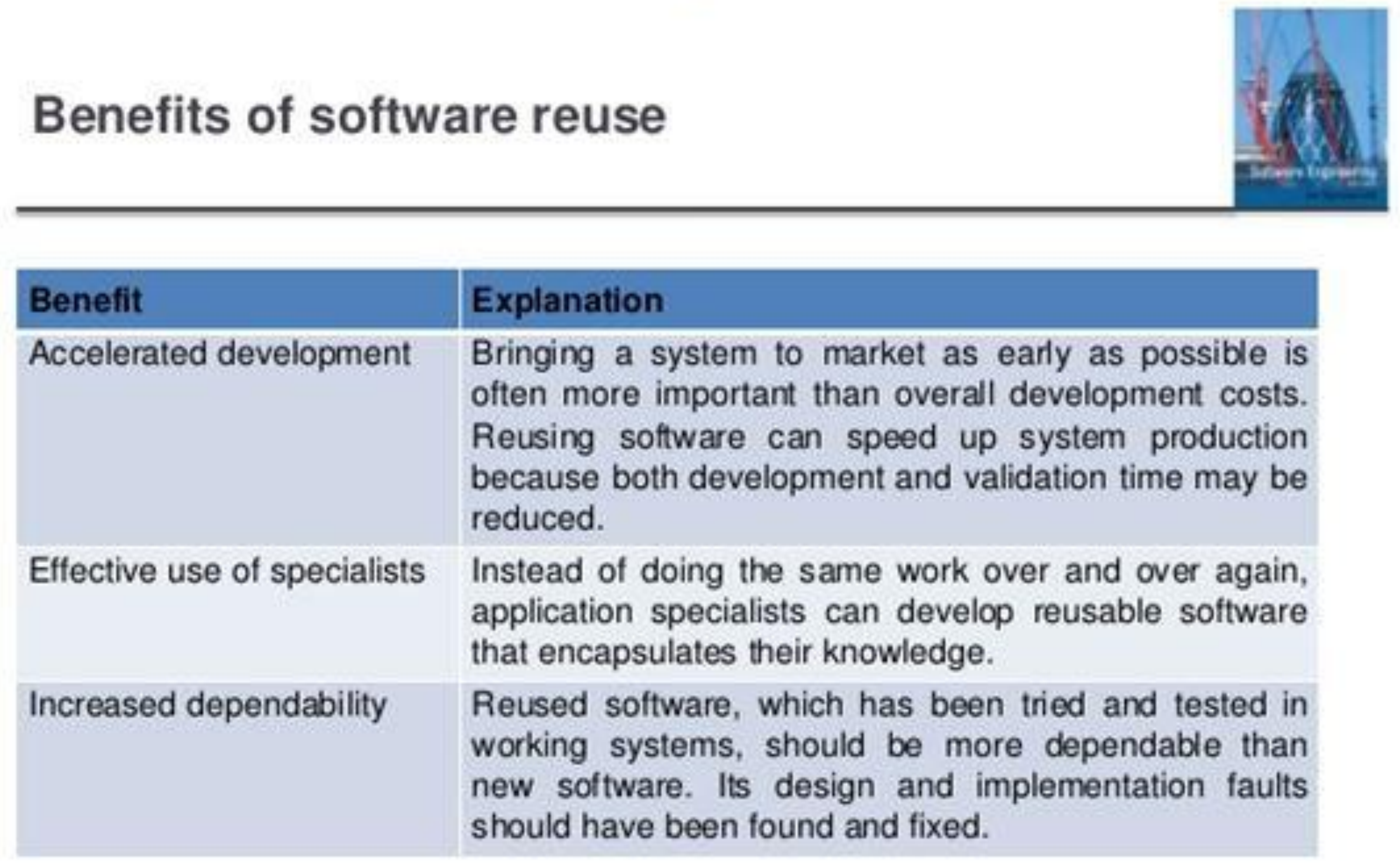
Reuse types:

1. **System reuse** : Complete systems, which may include several application programs may be reused.

2. **Application reuse** : An application may be reused either by incorporating it without change into other or by developing application families.

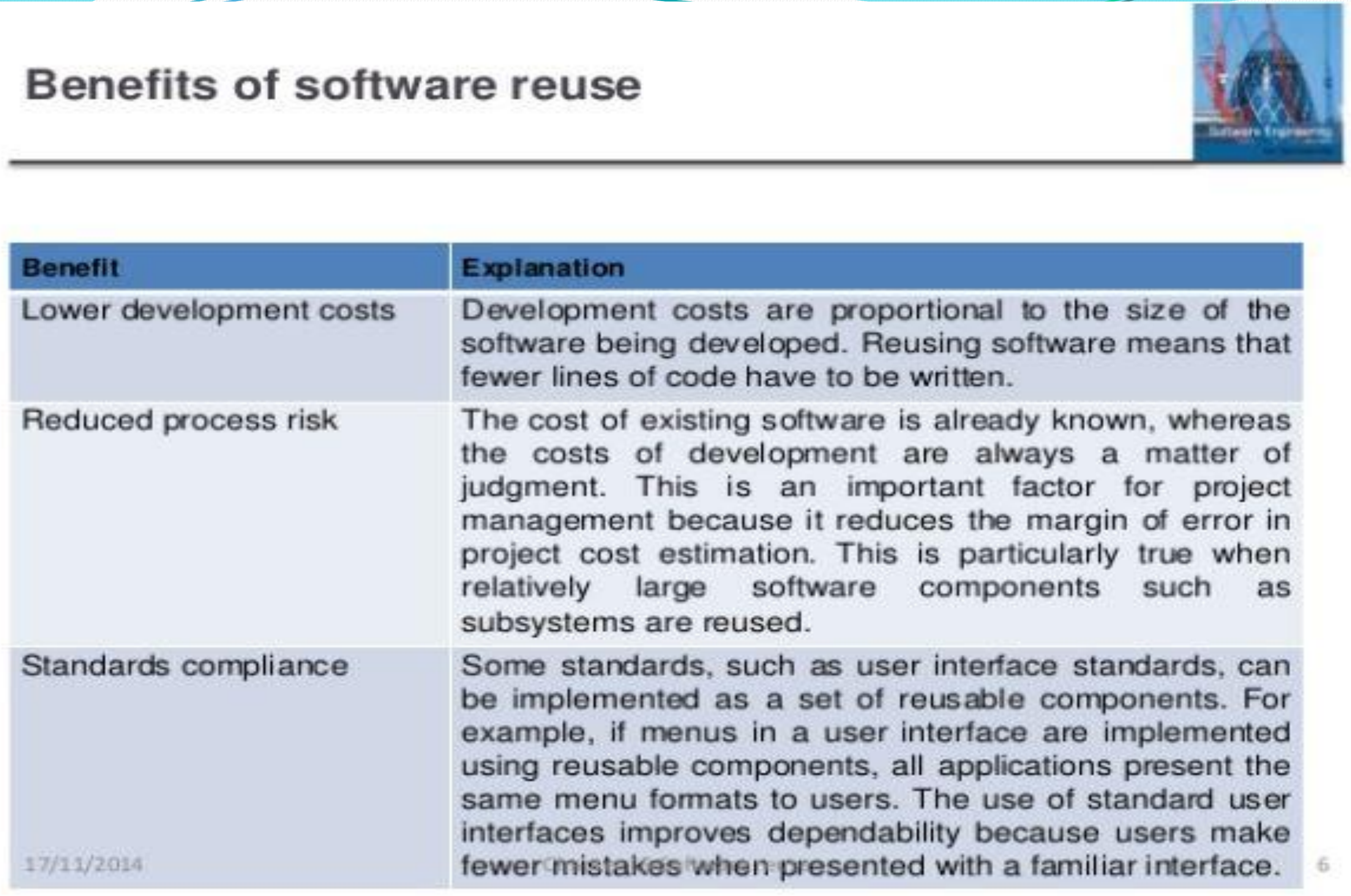
3. **Component reuse :** Components of an application from sub-systems to single objects may be reused.

4. **Object and function reuse** : Small-scale software components that implement a single well- defined object or function may be reused.



Benefits of software reuse

Benefit	Explanation
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
Increased dependability	Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.



Benefits of software reuse

Benefit	Explanation
Lower development costs	Development costs are proportional to the size of the software being developed. Reusing software means that fewer lines of code have to be written.
Reduced process risk	The cost of existing software is already known, whereas the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as subsystems are reused.
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.

Problems with reuse

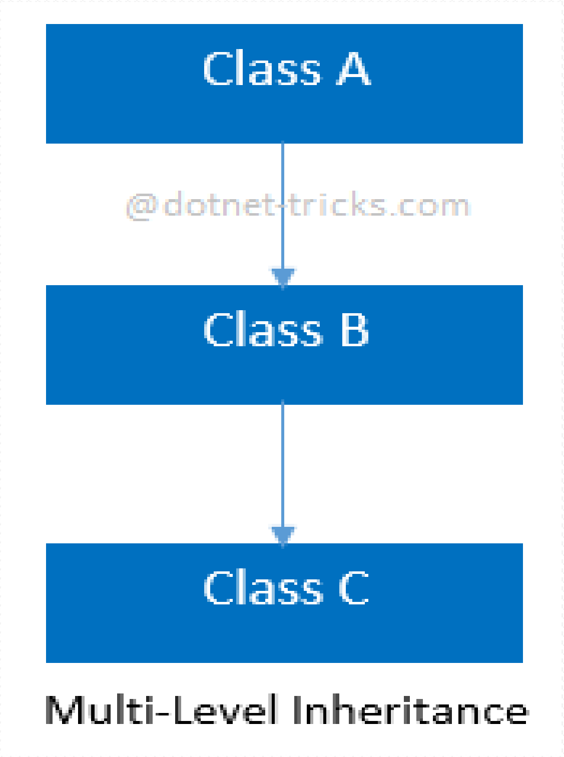
Problem	Explanation
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.
Increased maintenance costs	If the source code of a reused software system or component is not available then maintenance costs may be higher because the reused elements of the system may become increasingly incompatible with system changes.

Types of Inheritance

1. **Multilevel Inheritance**

•In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class.

•This form of inheritance is known as multilevel inheritance.



Multi-Level Inheritance

Syntax:

```
class A
{ … .. … };
class B: public A
{ … .. … };
class C: public B
{ … … … };
```

Here, class B is derived from the base class A and the class C is derived from the derived class B.

Example:

```
#include <iostream>
using namespace std;
class A
{ public: void display()
{ cout<<"Base class content."; }
};
class B : public A
{ };
class C : public B
{ };
int main()
{ C obj;
 obj.display();
return 0; }
```

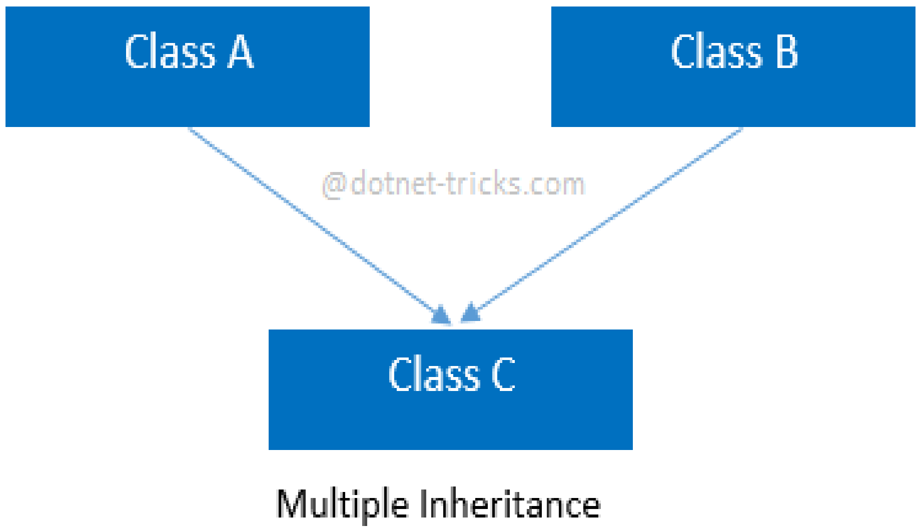
Output

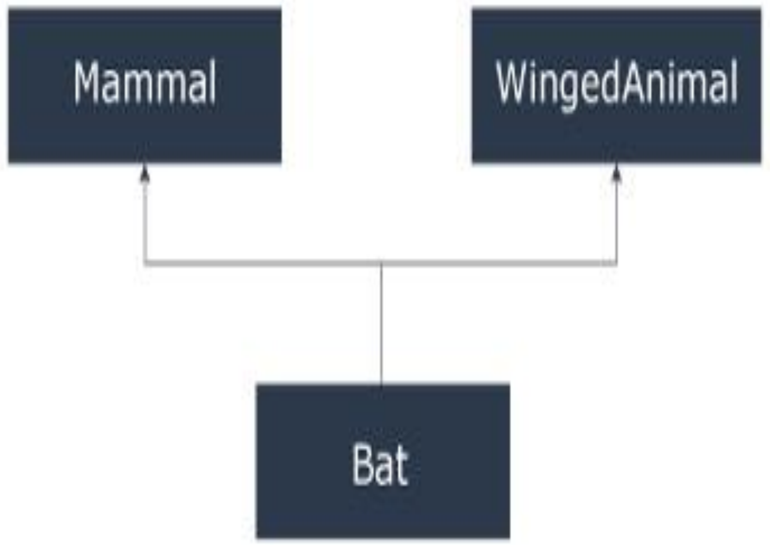
```
Base class content.
```

2. Multiple Inheritance

•In C++ programming, a class can be derived from more than one parents.

•For example: A class Bat is derived from base classes Mammal and WingedAnimal. It makes sense because bat is a mammal as well as a winged animal.





```
#include <iostream>
using namespace std;
class Mammal
{ public:
Mammal()
{ cout << "Mammals can give direct birth." << endl; }
};
 class WingedAnimal
{ public:
WingedAnimal()
{ cout << "Winged animal can flap." << endl; }
 };
 class Bat: public Mammal, public WingedAnimal
{public:
Bat(){cout << “Bat are generally black in color”} };
 int main()
{ Bat b1;
return 0; }
}
```

Output:

```
Mammals can give direct birth.
Winged animal can flap.
Bat are generally black in color
```

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
{public:
void someFunction( )
{ .... ... .... }
};
 class derived : public base1, public base2
{};
int main()
{
derived obj;
obj.someFunction() ;
// Error! }
```

```
class base1
{ public:
void printFunction( )
{ cout << “base1 is called”<<endl;}
};
 class base2
{ void printFunction( )
{ cout << “base2 is called”<<endl;}
};
 class derived : public base1, public base2
{ };
int main()
{
derived obj;
obj.printFunction() ;
// Error! }
```

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

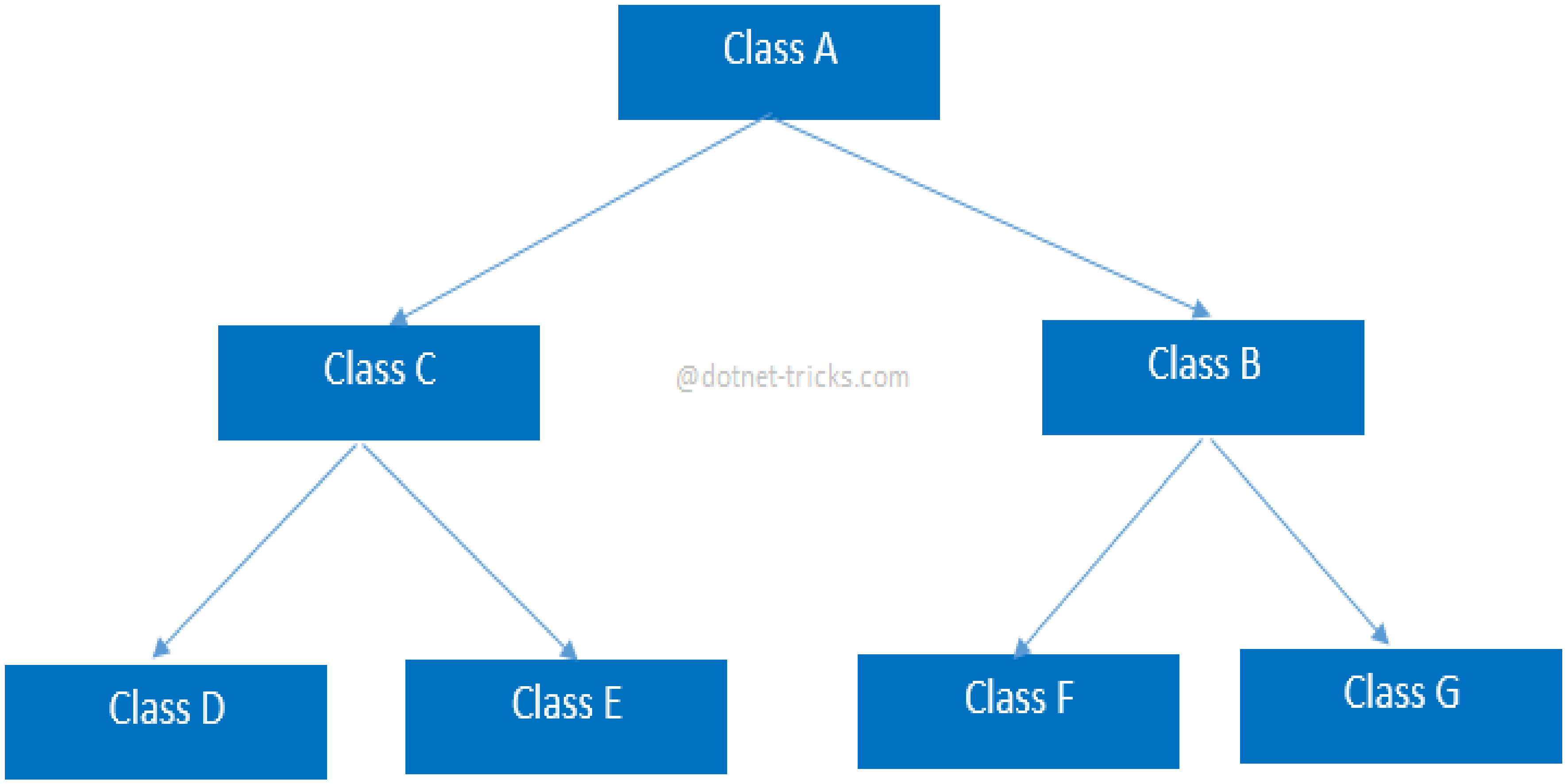
```
int main()
{
derived obj;
obj.base1::someFunction( ); // Function of base1 class is called
obj.base2::someFunction(); // Function of base2 class is called.
}

int main()
{
obj.base1::printFunction( ); // Function of base1 class is called
obj.base2::printFunction(); // Function of base2 class is called.
}
```

3. Hierarchical Inheritance

•If more than one class is inherited from the base class, it's known as hierarchical inheritance.

• In hierarchical inheritance, all features that are common in child classes are included in the base class.

For example: Physics, Chemistry, Biology are derived from Science class.



Hierarchical Inheritance

Syntax :

```
class base_class
{ … .. … }
class first_derived_class: public base_class
{ … .. … }
class second_derived_class: public base_class
{ … .. … }
class third_derived_class: public base_class
{ … .. … }
```

```
#include <iostream>
using namespace std;
class A //single base class
{
public:
 int x, y;
void getdata()
{ cout << "\nEnter value of x and y:\n";
  cin >> x >> y; }
};
 class B : public A //B is derived from
//class A
{ public: void product()
{ cout << "\nProduct= " << x * y; }
};
class C : public A //C is also derived from
//class base  A
{
 public:
void sum()
{ cout << "\nSum= " << x + y; }
};
 int main()
{ B obj1; //object of derived class B
C obj2; //object of derived class C
obj1.getdata();
obj1.product();
obj2.getdata();
 obj2.sum();
return 0; } //end of program
```

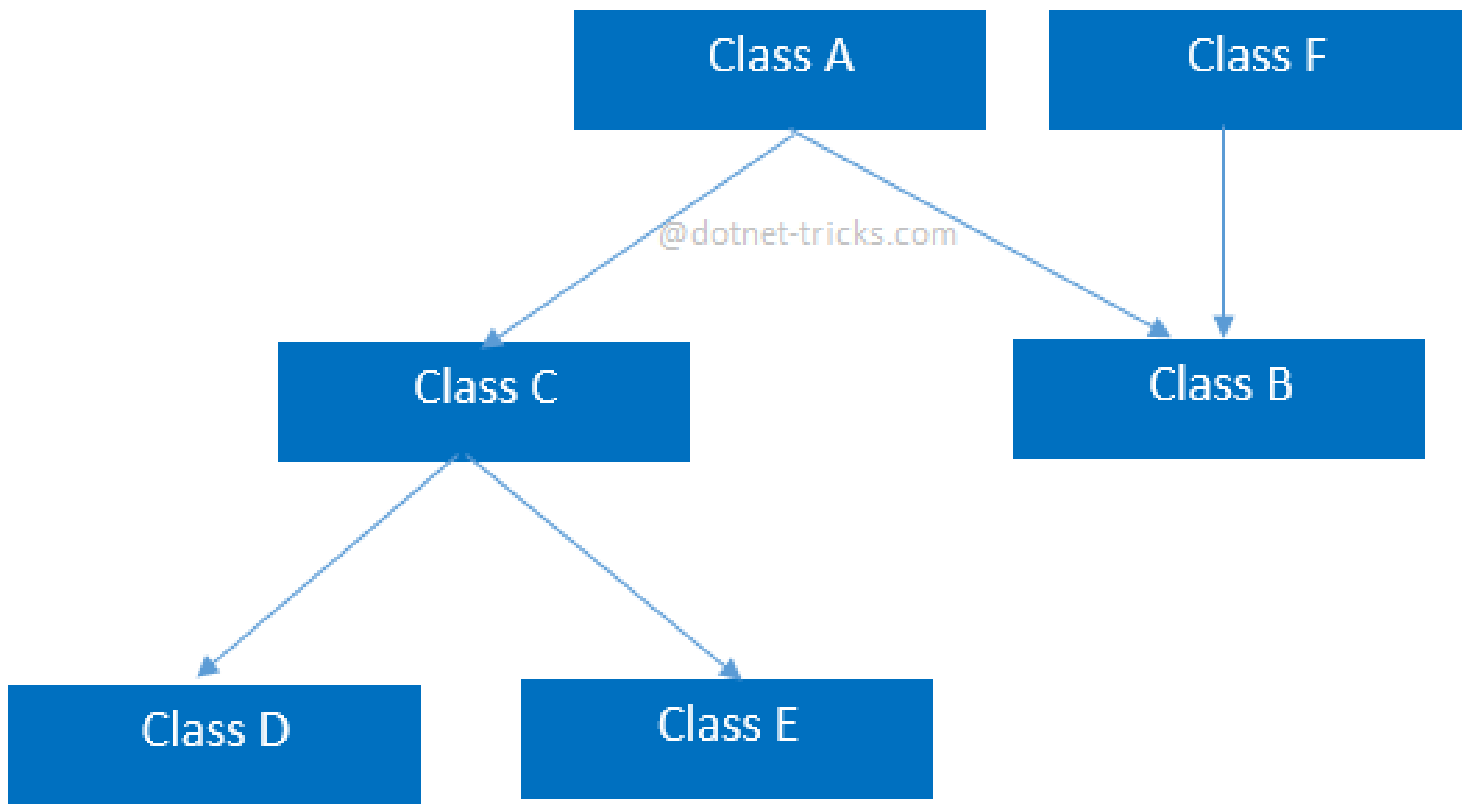
Output

```
Enter value of x and y:
2 3
Product= 6
Enter value of x and y: 2 3
Sum= 5
```

4. Hybrid Inheritance

•This is combination of more than one inheritance.

•Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance or Hierarchical and Multipath inheritance or Hierarchical, Multilevel and Multiple inheritance.



Hybrid Inheritance – (a combination of Hierarchical and multiple)

Syntax of hybrid inheritance

```
//Base Class
class A
{ public:
 void funcA()
{ //TO DO: }
};
//Base Class
class F
{ public:
void funcF()
{ //TO DO: }
};
//Derived Class
class B : A, F
{ public:
 void funcB()
{ //TO DO: }
} ;
```

```
//Derived Class
class C : A
{ public:
void funcC()
{ //TO DO: }
};
 //Derived Class
class D : C
{ public:
void funcD()
{ //TO DO: }
}
//Derived Class
class E : C
{ public:
 void funcE()
{ //TO DO: }
 }
```

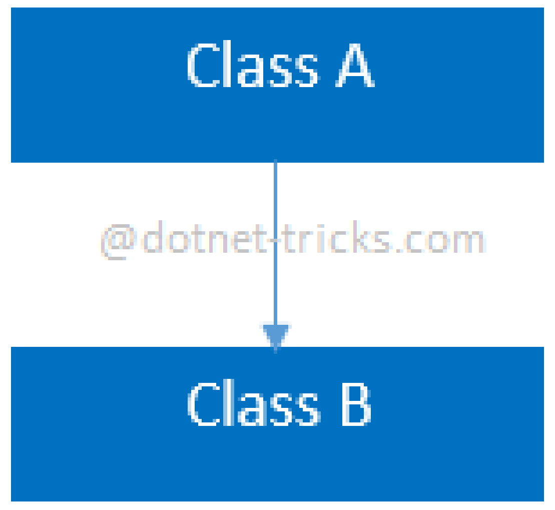
Example of hybrid inheritance

```
//Base Class
class A
{ public:
int x, y;
void getData()
{ cout << "\nEnter value of x and
y:\n";
cin >> x >> y;} };
//Base Class
class F
{ public:
void printData()
{ cout<< “x :” <<x;
cout<< “y :” <<y;} };
//Derived Class
class B : A, F
{} ;
//Derived Class
class C : A
{};
 //Derived Class
class D : C
{};
//Derived Class
class E : C
{};
int main()
{
B objb;
objb.getData();
objb.printData();
D objd;
objd.getData();
E obje;
obje.getData();
return 0;
}
```

```
OUTPUT:
Enter value of x and y:
2 3
x : 2 y : 3
Enter value of x and y:
6 7
Enter value of x and y:
10 20
```

5. Single Inheritance

In this inheritance, a derived class is created from a single base class.
In the given example, Class A is the parent class and Class B is the child class since Class B inherits the features and behavior of the parent class A.



Single Inheritance

Syntax of Single Inheritance

```
//Base Class
class A
{ public:
 void funcA()
{ //TO DO: }
};
 //Derived Class
class B : public A
{ public:
 void funcB()
{ //TO DO: }
};
```

```
#include <iostream>
using namespace std;
class A //single base class
{
public:
 int x, y;
void getdata()
{ cout << "\nEnter value of x and y:\n";
  cin >> x >> y; }
};
 class B : public A
//B is derived from class base
{ public: void product()
{ cout << "\nProduct= " << x * y; }
};
```

```
int main()
{
B obj1; //object of derived class B
obj1.getdata();
obj1.product();
return 0;
}
//end of program
```

Output

```
Enter value of x
and y:
2 3
Product= 6
```

Subtype in C++

•Any derived class (or subclass) is the subtype of the base class.

•Example:

1. Consider a base class : <u>Science</u> and derived classes: <u>Physics Chemistry and Biology</u>

•<u>Physics chemistry biology</u> are known as subtype of class <u>science.</u>

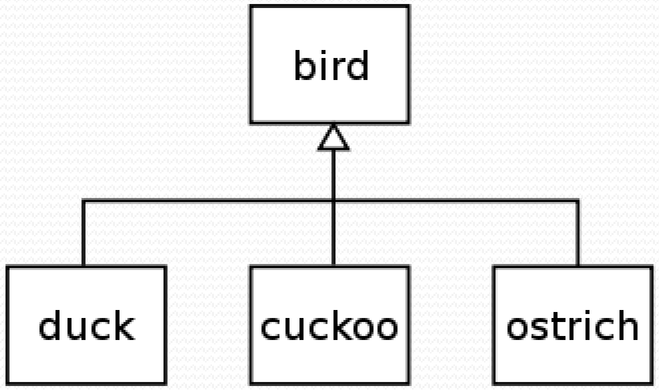
2. Consider a base class : <u>Country</u> and derived classes: <u>Nepal, America and Norway.</u>

• <u>Nepal, America and Norway.</u> are known as subtype of class <u>Country.</u>

•A simple practical example of subtype is shown in the diagram, below.

•The type "bird" has three subtypes "duck", "cuckoo" and "ostrich".

•Conceptually, each of these is a variety of the basic type "bird" that inherits many "bird" characteristics but has some specific differences.



Principle of substitutability

•Any object of derived class(subclass) can be assigned to base class object. That is called substitutability.

•Example:

1. Consider a base class : <u>Science</u> and derived classes: <u>Physics Chemistry and Biology</u>

•<u>Physics chemistry biology</u> are known as subtype of class <u>science.</u>
•Let object of class Science be Sci
•Let object of class Physics , Chemistry and Biology be Phy, Che and Bio respectively
•Then Phy or Che or Bio can be assigned to Sci.

•**Substitutability** is a principle in OOP stating that, in a computer program, if S is a subtype of T, then objects of type T may be *replaced* with objects of type S (i.e., an object of type T may be *substituted* with any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.).

•More formally, the **Liskov substitution principle** (**LSP**) is a particular definition of a subtyping relation, called (**strong**) behavioral subtyping, that was initially introduced by Barbara Liskov in a 1987 conference keynote address titled *Data abstraction and hierarchy*.