Inheritance in C++

By Avani M. Sakhapara
IT Dept, KJSCE

Content

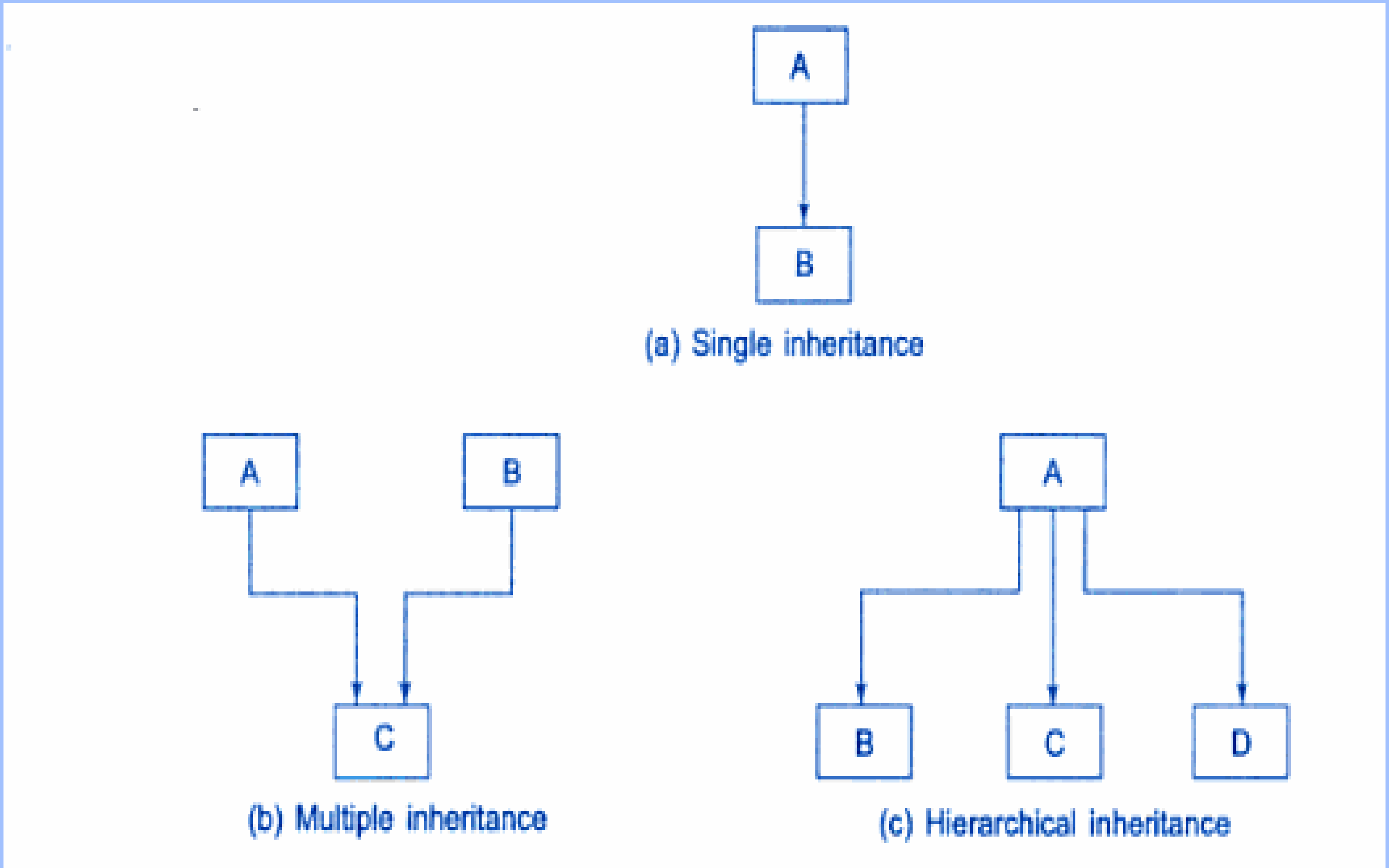
- Base and Derived Classes
  - Single Inheritance
  - Declaration of derived classes
  - Order of Constructor and Destructor Execution
  - Inherited member accessibility
- Multiple Inheritance
- Virtual Base Classes

Base and Derived Classes

- A *base class* is a previously defined class that is used to define new classes

- A *derived class* inherits all the data and function members of a base class (in addition to its explicitly declared members.)

Types of Inheritance

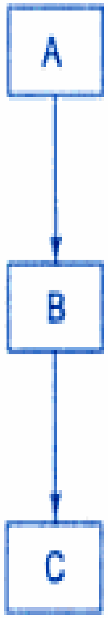


(a) Single inheritance

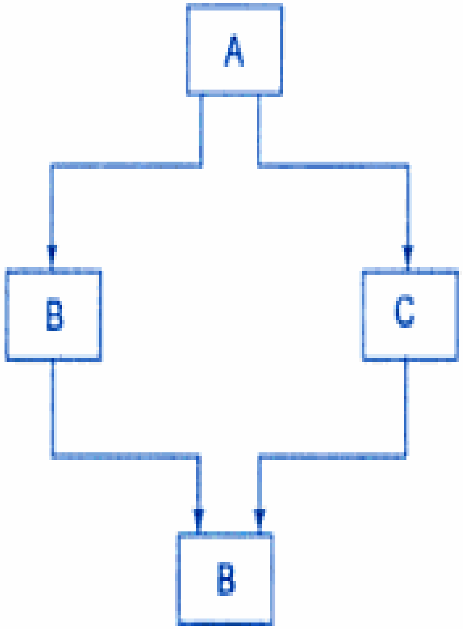
(b) Multiple inheritance

(c) Hierarchical inheritance

Types of Inheritance

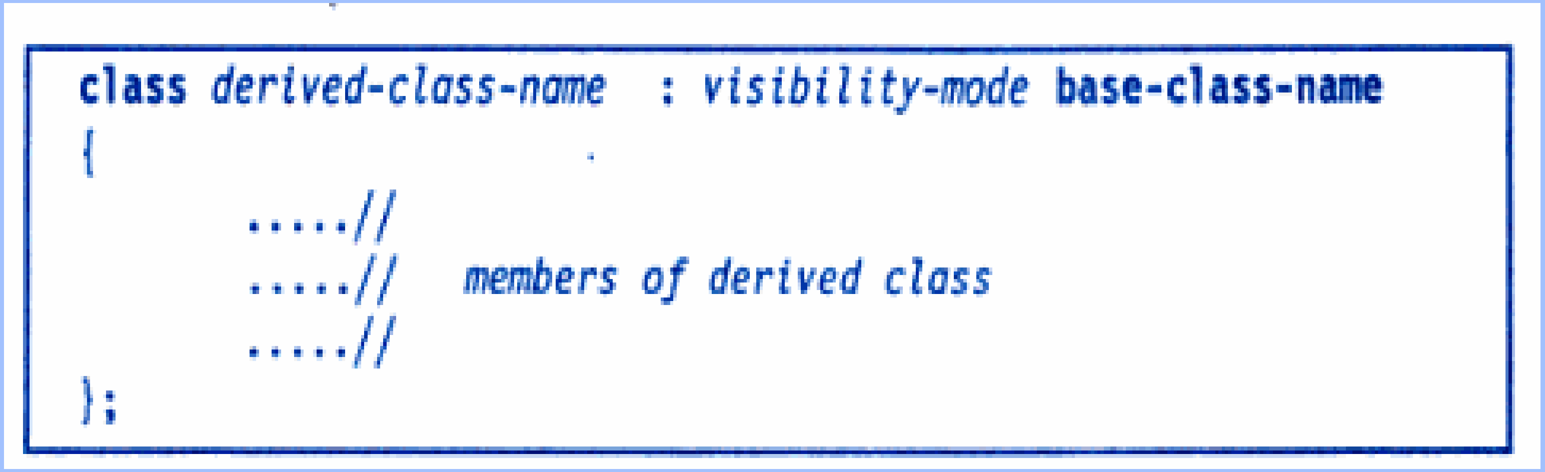


(d) Multilevel Inheritance



(e) Hybrid inheritance

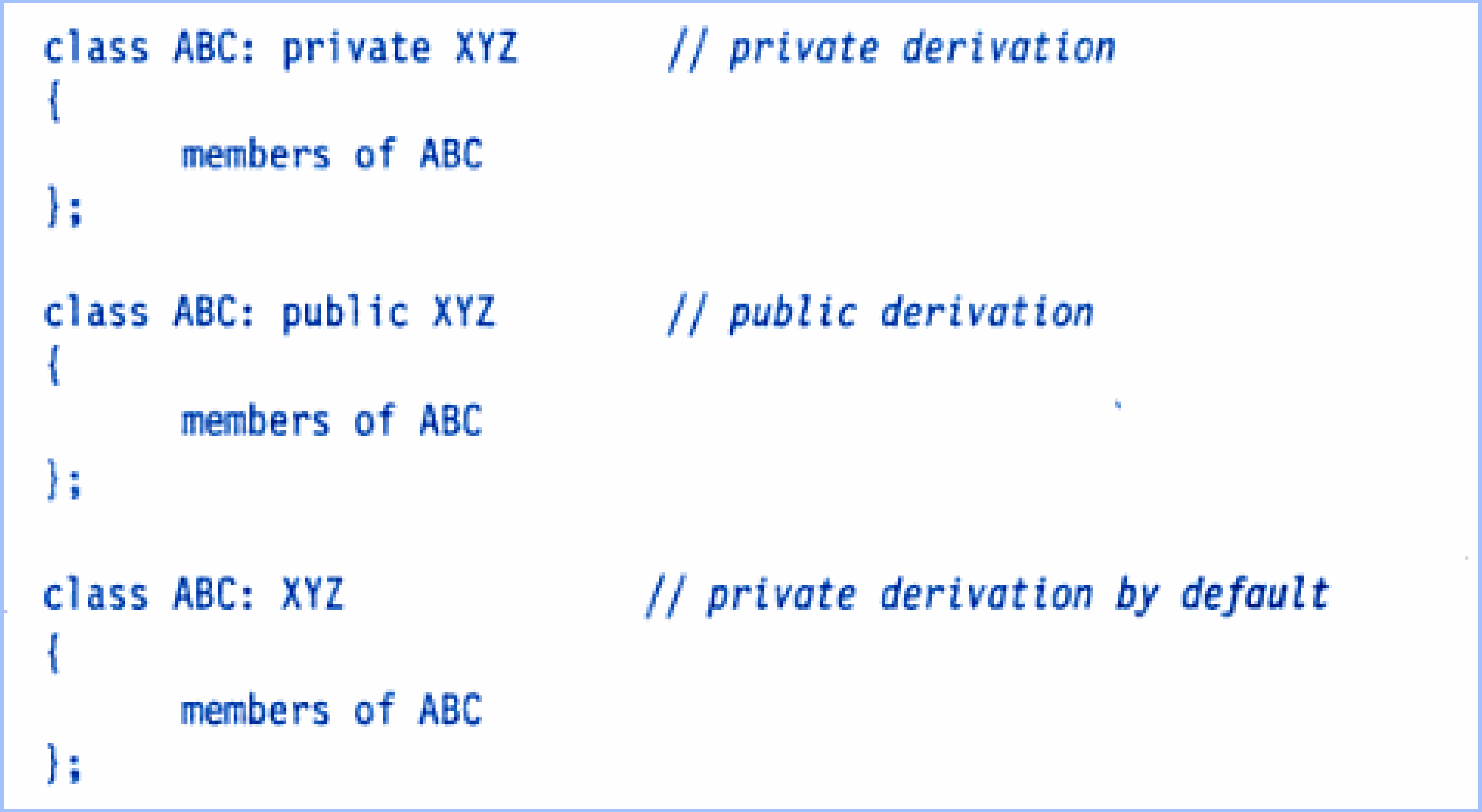
Declaring Derived Classes



```
class derived-class-name  : visibility-mode base-class-name
{
        .....//
        .....//   members of derived class
        .....//
};
```

*visibility-mode = public|protected|**private(default)***

Example



```
class ABC: private XYZ       // private derivation
{
        members of ABC
};

class ABC: public XYZ        // public derivation
{
        members of ABC
};

class ABC: XYZ               // private derivation by default
{
        members of ABC
};
```

Order of Constructor and Destructor Execution

- Base class constructors are ***always*** executed first.
- Destructors are executed in exactly ***the reverse order*** of constructors
- The following example, shows you the ordering of constructors.

Example

```
Class Employee{
Public:
      Employee();
      //…
};
Class SalariedEmployee:public Employee{
Public:
      SalariedEmployee();
      //…
};
Class ManagementEmployee:public SalariedEmployee{
Public:
      ManagementEmployee();
      //…
};
ManagementEmployee M;
```

Types of Class Members

- private
- protected
- public

Types of Inheritance

- public
- private
- protected

Public Inheritance

- Public and protected members of the base class become respectively public and protected members of the derived class.

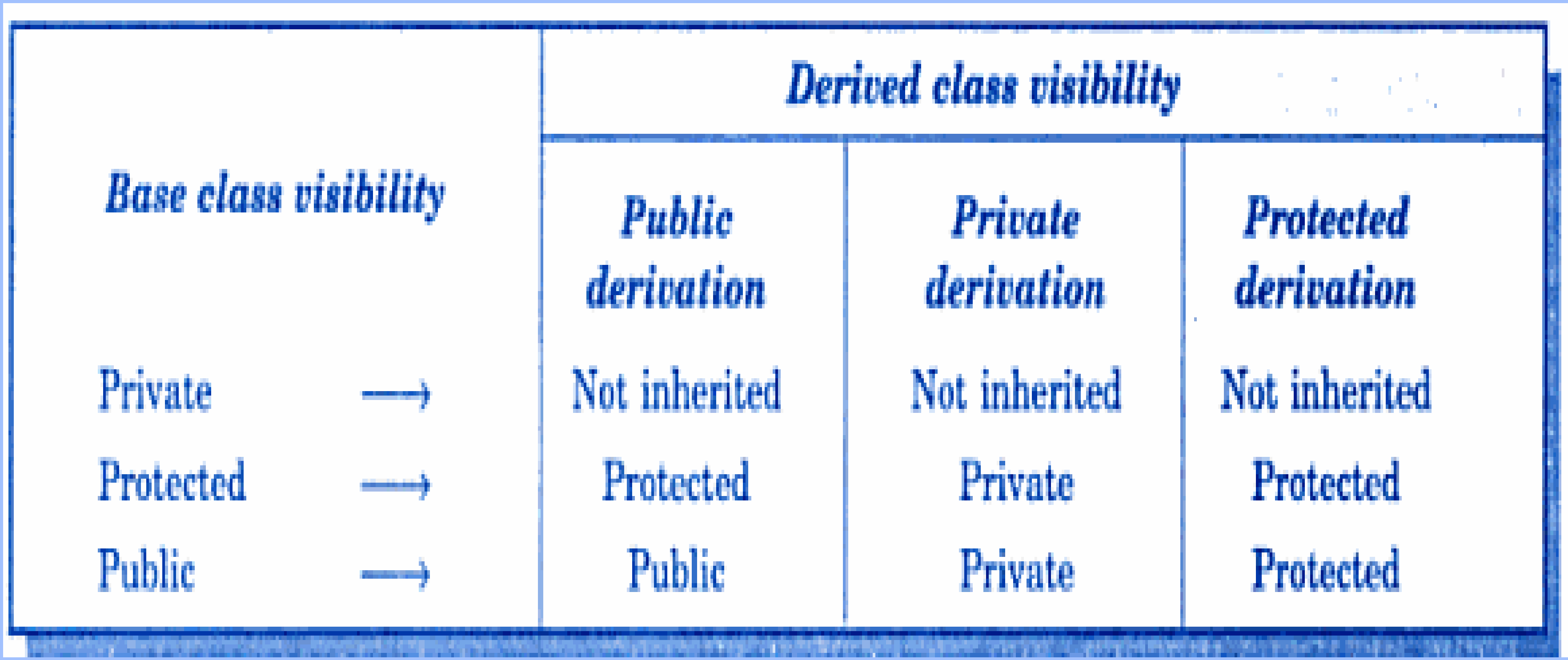
Private Inheritance

- Public and protected members of the base class become private members of the derived class.

Protected Inheritance

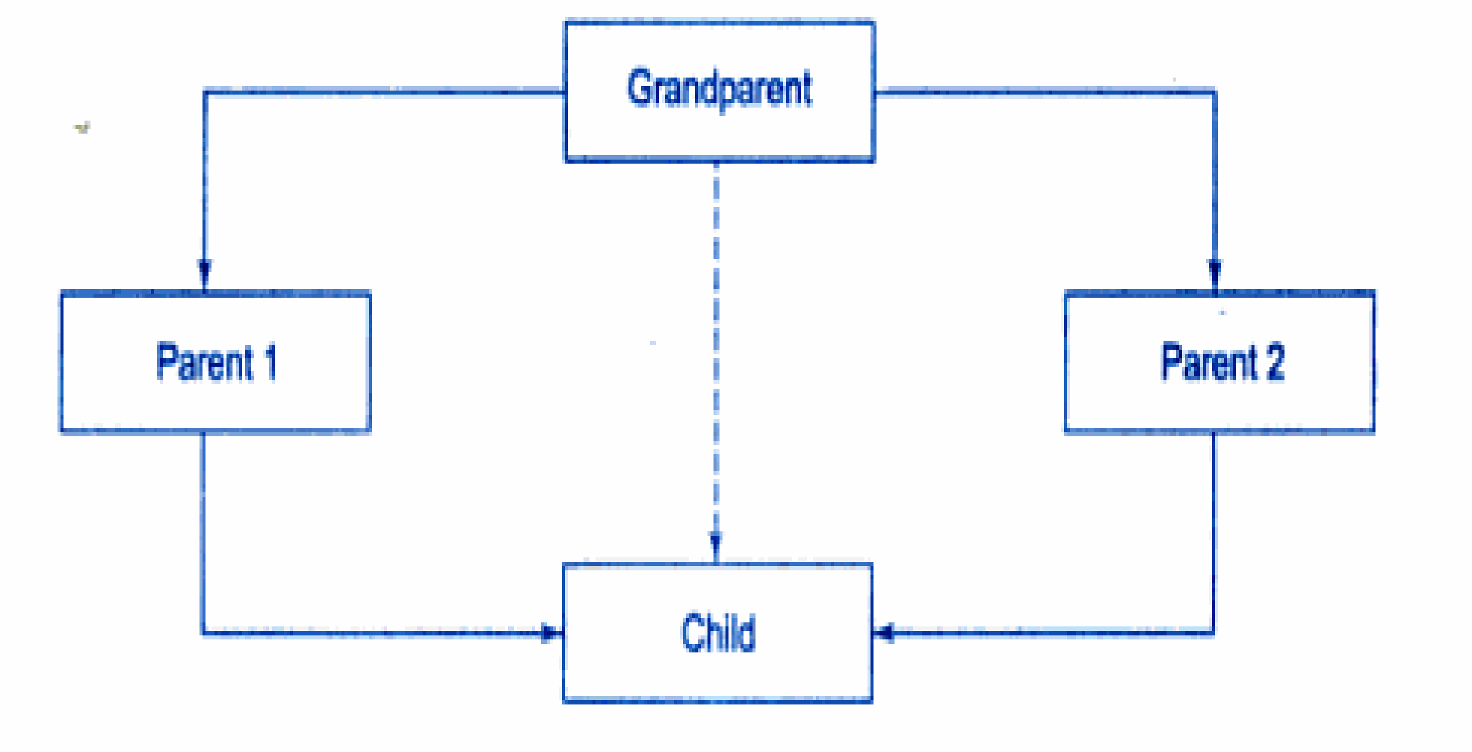
- Public and protected members of the base class become protected members of the derived class.

Visibility of inherited members

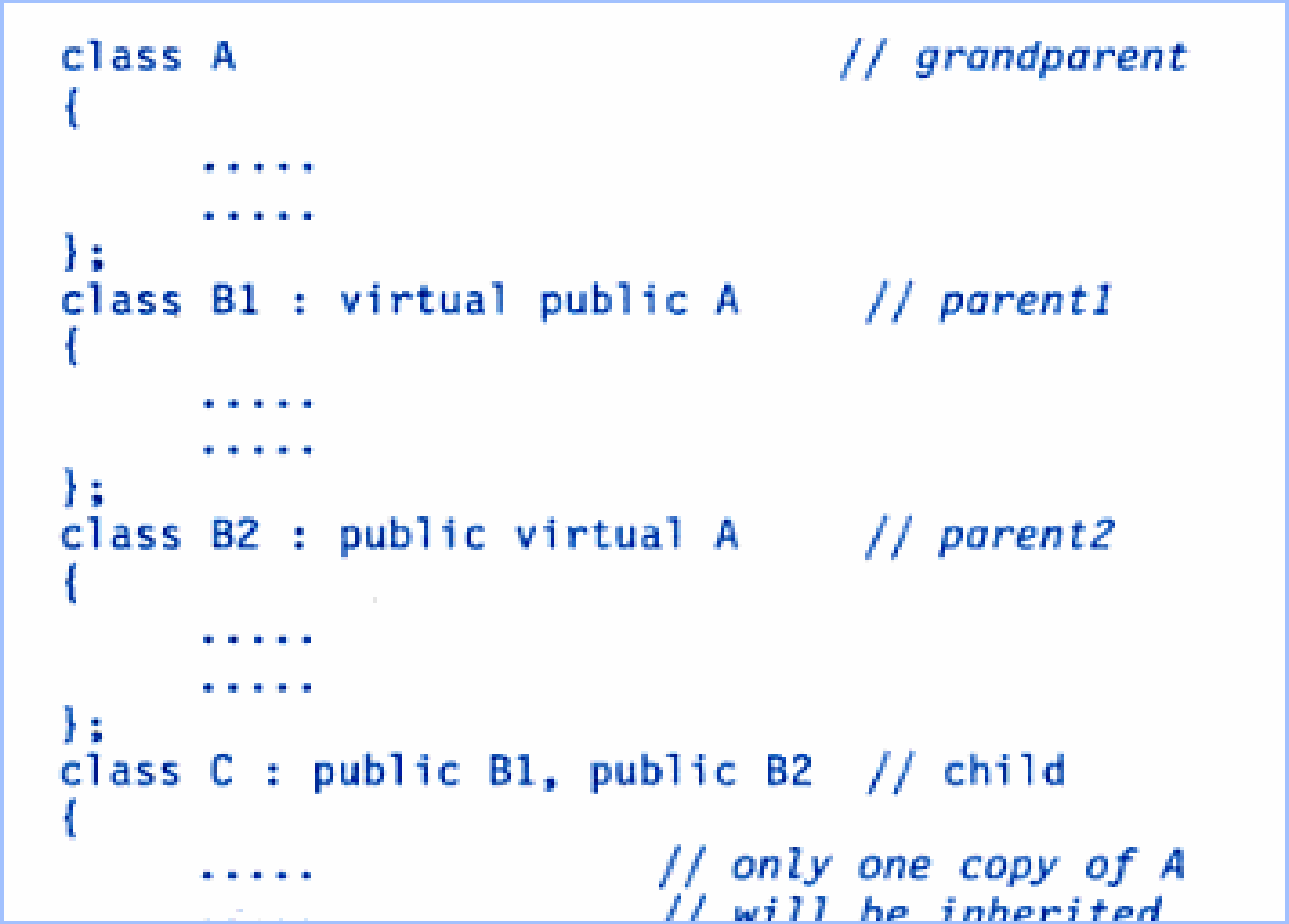


Base class visibility		Derived class visibility		
		Public derivation	Private derivation	Protected derivation
Private	⟶	Not inherited	Not inherited	Not inherited
Protected	⟶	Protected	Private	Protected
Public	⟶	Public	Private	Protected

Virtual Base Class

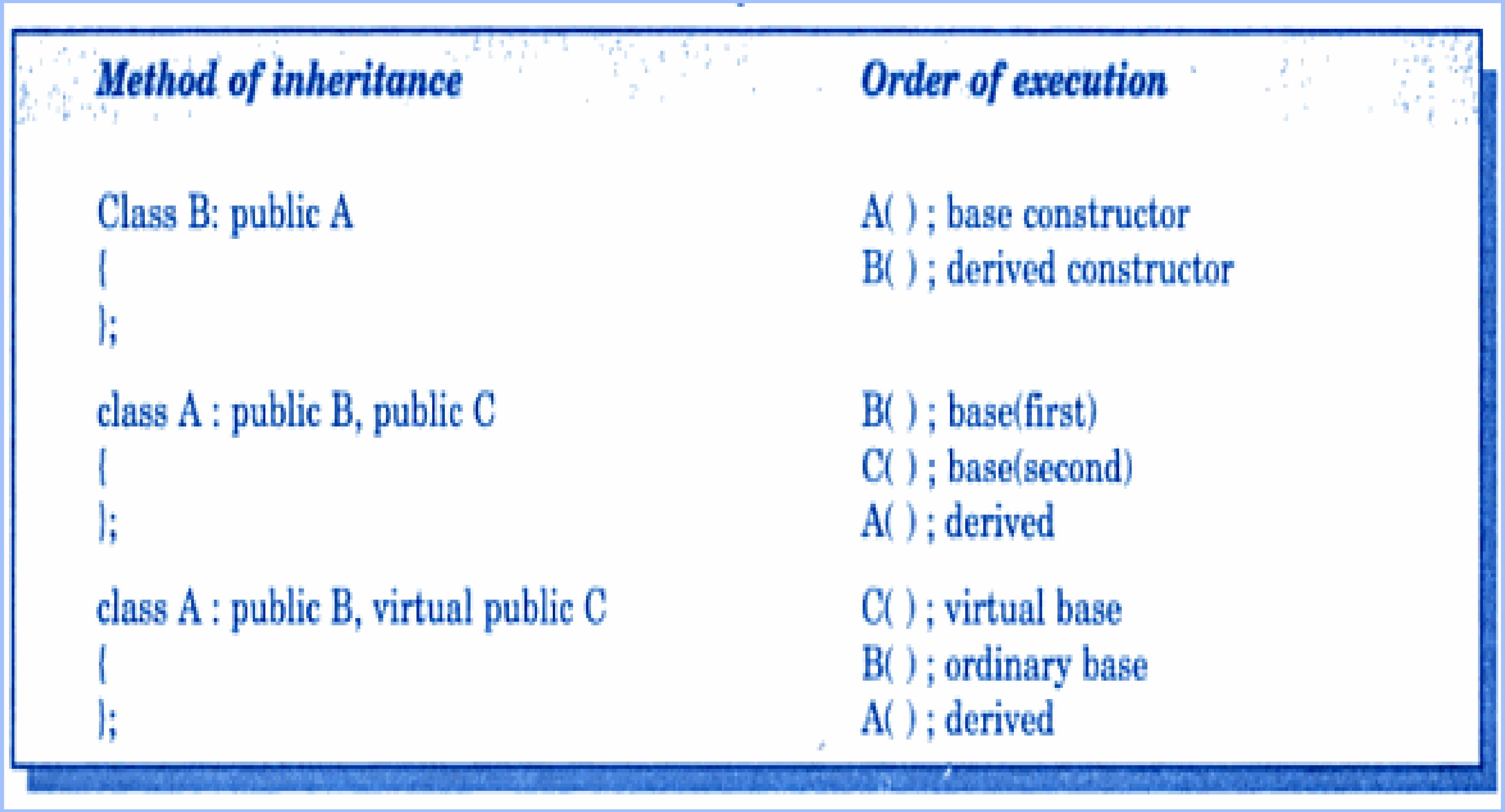


Virtual Base Class



```
class A                                     // grandparent
{
        .....
        .....
};
class B1 : virtual public A              // parent1
{
        .....
        .....
};
class B2 : public virtual A              // parent2
{
        .....
        .....
};
class C : public B1, public B2   // child
{
        .....                        // only one copy of A
        .....                        // will be inherited
```

Execution of Constructors



Method of inheritance	*Order of execution*

Class B: public A
{
};
	A( ) ; base constructor
	B( ) ; derived constructor

class A : public B, public C
{
};
	B( ) ; base(first)
	C( ) ; base(second)
	A( ) ; derived

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

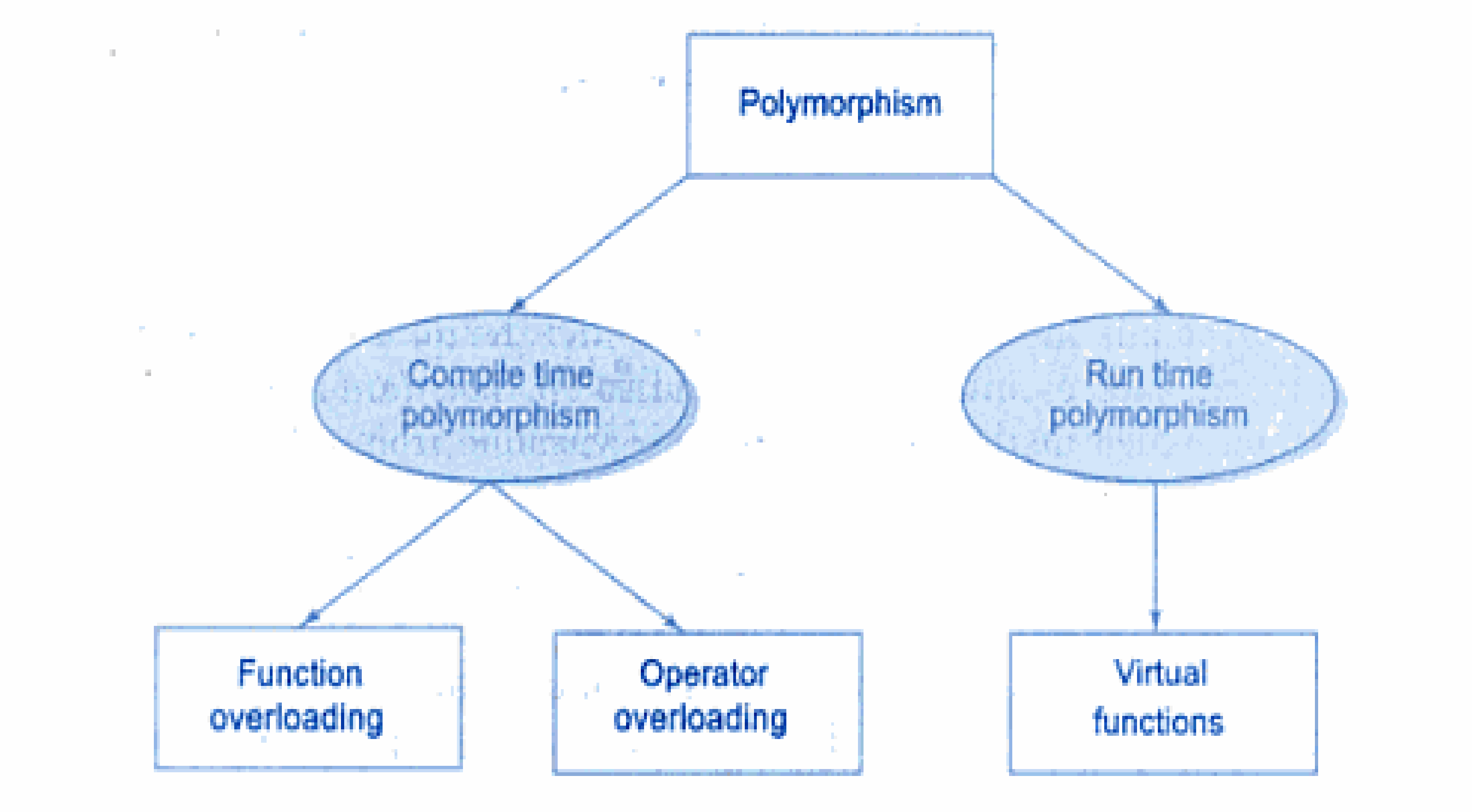
Destructor Function

- Destructors are called implicitly starting with the last derived class and moving in the direction of the base class.

Compatibility Between Base and Derived Classes

- An object of a derived class can be treated as an object of its base class.
- The reverse is not true.

Polymorphism



Overriding

- A function in the derived class with the same function name will override the function’s variables in the base class.
- You can still retrieve the overridden functions variables by using the scope resolution operator ”::”.

Overriding

```
#include <iostream.h>
#include <stdlib.h>
class A
{   int x;
public:
    A(){x = 5;}
    int get(){return x;}
};
class B: public A
{   int y;
public:
    B(){y = 10;}
    int get(){return y;}
};
```

```
void main()
{         B b;
          cout << b.get()<<endl;
          cout << b.A::get()<<endl;
}
```

Need of Virtual Function

```
#include <iostream.h>
#include <stdlib.h>
class A
{   int x;
public:
    A(){x = 5;}
    int get(){return x;}
};
class B: public A
{   int y;
public:
    B(){y = 10;}
    int get(){return y;}
};
```

```
void main()
{
A *ptr;
A m;
B q;
ptr=&m;//point to base class
cout <<ptr->get()<<endl; //5
ptr=&q;//point to derived class
cout << ptr->get()<<endl; //5
}
```

Need of Virtual Function

- In the previous example, the statement ptr->get() calls the get() function of base class A every time.
- This is because C++ determines which function to use based on the type of the pointer instead of based on the type of the object pointed to by the base pointer.

Virtual Function

- Run time polymorphism is achieved using virtual function
- When we have overridden functions in the base and derived classes, the function in base class is declared as *virtual* using the keyword **virtual** before its declaration.

Virtual Function

- When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of pointer.

Virtual Function Example

```
#include <iostream.h>
#include <stdlib.h>
class A
{   int x;
public:
    A(){x = 5;};
    virtual int get(){
    return x;};
};
class B: public A
{   int y;
public:
    B(){y = 10;};
    int get(){return y;};
};
```

```
void main()
{
A *ptr;
A m;
B q;
ptr=&m;//point to base class
cout <<ptr->get()<<endl; //5
ptr=&q;//point to derived class
cout << ptr->get()<<endl; //10
}
```

Pure Virtual Function

- A pure virtual function is a function that has no definition relative to base. class.
- It is defined as

```
virtual int get()=0;
```

- In such case each derived class should define the function or redeclare it as a pure virtual function.

Abstract class

- A class having pure virtual function is called abstract class
- An object of an abstract class cannot be created
- A base class which is abstract is called abstract base class.
- Abstract base class is used to provide some traits to the derived classes and to create a base pointer required for run time polymorphism.

Pure Virtual Function Example

```
#include <iostream.h>
class A //abstract class
{   int x;
public:
    A(){x = 5;}
    virtual int get()=0;
//pure virtual function
};
class B: public A
{   int y;
public:
    B(){y = 10;};
    int get(){return y;}
//function defined in
    derived class
};
```

```
void main()
{
A *ptr;
A m;//compiler error
B q;
ptr=&q;//point to derived class
cout << ptr->get()<<endl; //10
}
```

Thank You