Chapter 12

類別間的關係(一)
嵌入與繼承

類別間的關係(一)：使用

- 使用：

  某甲類別使用某乙類別物件來完成某項事情

  - 學生「使用」手機傳遞訊息
  - 公司「使用」金庫儲存重要文件
  - 人類「使用」交通工具旅行

uses a

類別間的關係(二)：有一個

■ 有一個：

某乙類別物件被置入某甲類別內當成必要的一部分使用

- 汽車「有」輪子
- 三角形「有」三個頂點
- 昆蟲「有」六隻腳
- 計算機內「有」中央處理單元

❖ 「有一個」 涵義上相當於 「包含」

has a

類別間的關係(三)：是一個

- 是一個：

  某甲類別物件也可以被當成某乙類別物件使用

  - 每個正方形「是一個」矩形
    矩形的內角皆是直角 → 正方形的內角也是直角
  - 每輛吉普車都「是一輛」汽車
    若汽車可以用來載貨、代步 → 吉普車同樣也可以
  - 每匹白馬都「是」匹馬
    馬可以用來載人、代步、馱物 → 白馬當然也可以

is a

使用，有一個，是一個（一）

- 隨著問題的不同，類別間的關係也不是僅有一種，且保持不變

- 若有以下九種類別：

  書籍、歷史書、兒童書、小說、書名、作者、價格、書局、封套

  - 每一本書籍都有書名、作者、價格　　has 有
  - 歷史書、兒童書、小說都是一種書　　is 是
  - 每間書局都有許多書　　has 有
  - 每一本書籍可以使用封套來保護書籍　　uses 使用

使用，有一個，是一個（二）

- 若有以下幾種類別：

  主機、零件、中央處理單元、硬碟、機殼、光碟機、記憶體、鍵盤、滑鼠、風扇、鍵盤、列表機、喇叭、螢幕、鍵盤價格、存貨數量

  - 每種零組件都有價格與存貨數量　　has 有
  - 硬碟、機殼、記憶體等都是電腦零件　　is 是
  - 每個主機都有硬碟、CPU、RAM　　has 有
  - 主機用喇叭發出聲音，螢幕顯示畫面　　uses 使用

❖ C++ 透過類別嵌入與類別繼承兩種方式來處理這三種類別間的關係

類別嵌入（一）

- 類別嵌入：

  將某甲類別物件「埋入」某乙類別內，相當於甲類別物件被當成乙類別內的資料成員

- 擁有：

  乙類別內「擁有」著甲類別物件資料成員

  → 每個乙類別物件就帶有著甲類別的特徵

class embedding

類別嵌入（二）

- 每個學生都有姓名、性別、出生年月日、學號等資料

```
// 列舉資料型別：性別
enum Gender { female , male } ;

// 結構資料型別：日期
struct Date {
   int year , month , day ;
};
class Student {
   private:
      string  name     ;      // 姓名
      Gender  gender   ;      // 性別
      Date    birthday ;      // 出生時間
      int     id       ;      // 學號
};
```

類別嵌入：建構函式設定（一）

- 當甲類別內有著乙類別物件的資料成員時，則每個甲類別物件在產生之際，乙類別物件資料成員須先行建構完成
  - 每輛車子內都有引擎、方向盤等零組件，當汽車要出廠販賣之前，引擎、方向盤等零組件一定要先建構完成且安裝到車子內
  - 電腦內有硬碟，顯示卡，**CPU**等零組件，當電腦要販售之前，所有須要的零組件一定要先被生產出來，才能安裝在機殼內

類別嵌入：建構函式設定（二）

- 汽車類別：

```
class Car {
   private:
      Engine engine ;   // 引擎資料成員
      Wheel  wheel  ;   // 方向盤資料成員
      Tire   tire   ;   // 輪子資料成員
      . . .
};
. . .
Car BMW ;
```

以上 **BMW** 相當於執行了以下的備用建構函式

```
Car::Car() : engine() , wheel() , tire(){}
```

❖ 類別內資料成員建構次序與資料成員在類別內宣告的次序一致，與起始設定列的次序無關

類別嵌入：解構函式

- 嵌入物件消失時，會在依次去除其內資料成員所佔有的記憶空間後，才會真正消失
- C++是透過執行各成員類別的解構函式來去除資料成員各自佔用的空間
- 各內部資料成員解構函式執行的順序剛好與建構函式執行的順序相反，也就是先產生的資料成員，後被移除

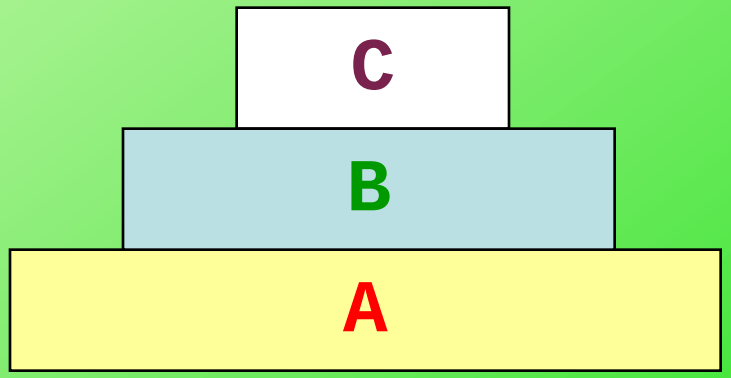
建構與解構的執行步驟

```
class A {
   private:
      B  a ;
};
```

```
class B {
   private:
      C  b ;
};
```

```
class C {
   . . .
};
```

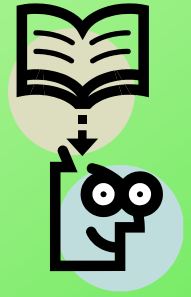
```
A  foo ;
```

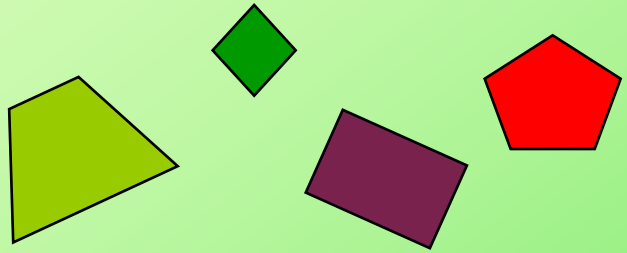


類別嵌入的使用時機（一）

- 擁有，包含：

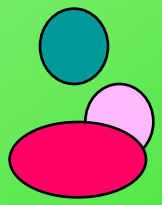
```
// 多邊形包含許多頂點
class Polygon {
   private:
      Point pts[20] ;
};
```

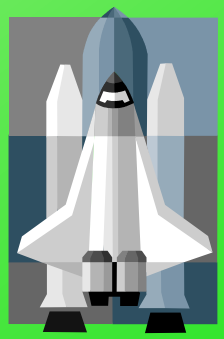




```
// 大學包含許多學院
class University {
   private:
      College  col[20] ;
};
```

```
// 顏色由紅，綠，藍三種顏色組成
class Color {
   private:
      int  red , green , blue ;
};
```





類別嵌入的使用時機（二）

- 使用，利用：

  某甲類別利用所嵌入的乙類別物件的某些功能來完成某些事情

類別嵌入的使用時機（三）

- 向量陣列：可調式陣列

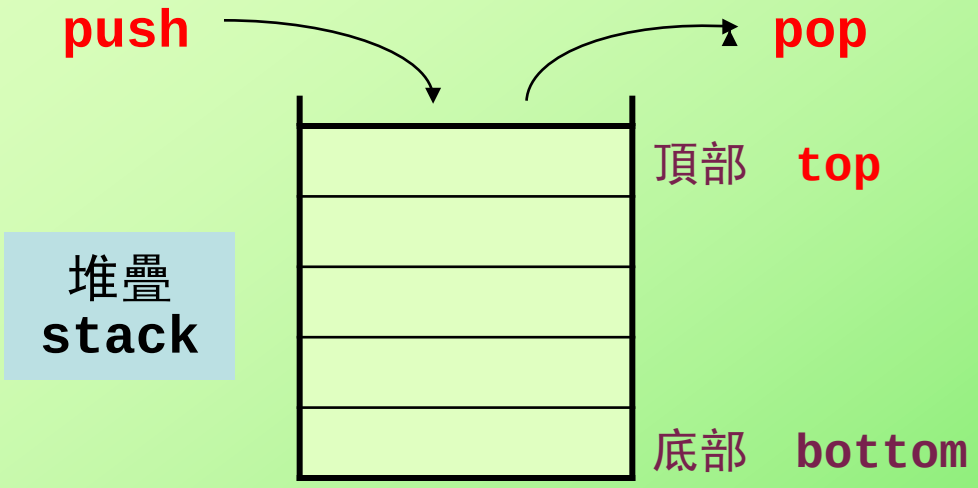
```
// 定義一個零長度的向量陣列
vector<int> a ;

// 存入 6 個整數 5
for( int i=0 ; i<6 ; ++i ) a.push_back(5);

// 印出所有向量陣列的元素值：5 5 5 5 5 5
for( int i=0 ; i< a.size() ; ++i ) cout << a[i] << ' ';

// 移除末尾兩個元素後，印出陣列長度：4
a.pop_back();  a.pop_back();
cout << a.size() << endl ;
```

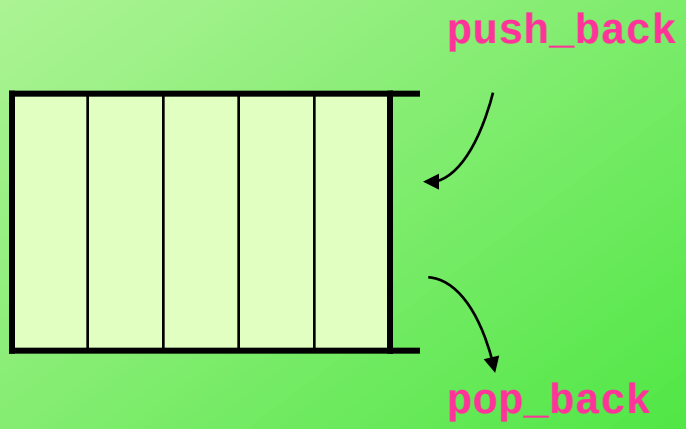
- 陣列複製

```
vector<int> foo , bar ;
...
bar = foo ;   // 將 foo 向量陣列複製給 bar 向量陣列
```

類別嵌入的使用時機（四）

■ 堆疊類別：






```
class Stack {
      private:
         vector<int>  data ;      // 儲存整數
      public:
         void push( int item ){ data.push_back(item) ; }
         void pop(){ data.pop_back() ; }
         int  top() const { return data[ data.size()-1 ] ; }
};
```

簡易選課程式

- 某補習班開設一門課程

```
const int MAX_CLASS = 3 ;
const int MAX_STUDENT = 10 ;
```

- 主要類別

  課程 (Course)， 班別 (Class)， 學生 (Student)

- 類別關係

  每一個課程包含若干個班別

  每一個班別包含若干個學生

程式

輸出

唐詩三百首資料庫

- 設計程式儲存不限定數量的五言詩與七言詩
- 主要類別

  詩 (Poem)， 唐詩資料庫 (Tang_Poem)
- 類別關係

  唐詩資料庫內儲存兩種類型的唐詩

```
vector<Poem> five  ;
vector<Poem> seven ;
```

程式

輸出

類別間的繼承關係

■ 三種繼承方式與語法

- 公共繼承

```
class Derived : public Base {
    ...
} ;
```

- 保護繼承

```
class Derived : protected Base {
    ...
} ;
```

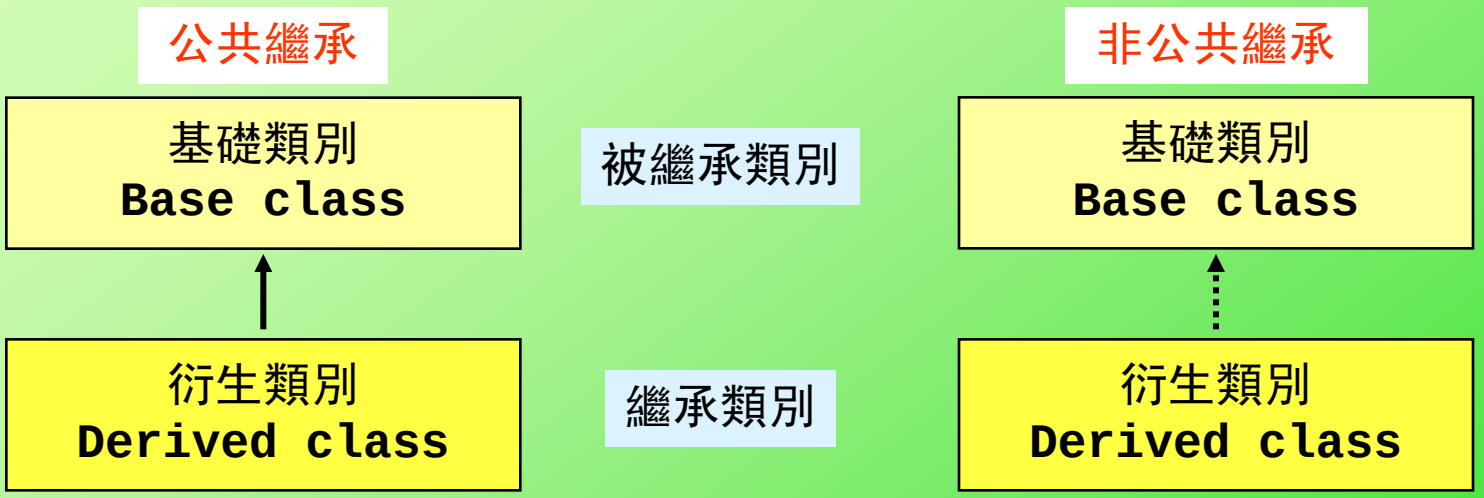
- 私有繼承

```
class Derived : private Base {
    ...
} ;
```

public , protected and private inheritance

基礎類別與衍生類別（一）

- 基礎類別：被繼承的類別
- 衍生類別：繼承的類別



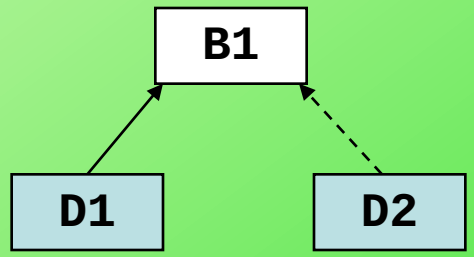
❖ 基礎類別也可稱為父類別（parent class）
衍生類別也可稱為子類別（child class）

base class, derived class

基礎類別與衍生類別（二）

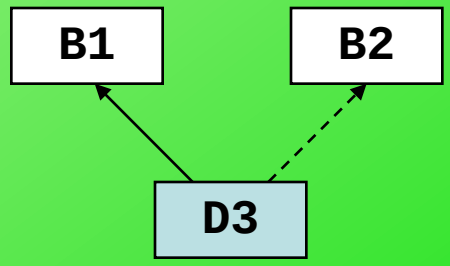
- 單一繼承：一個基礎類別有若干個衍生類別

```
class B1 {...} ;
class D1 : public B1 {...} ;
class D2 : private B1 {...} ;
```



- 多重繼承：一個衍生類別有若干個基礎類別

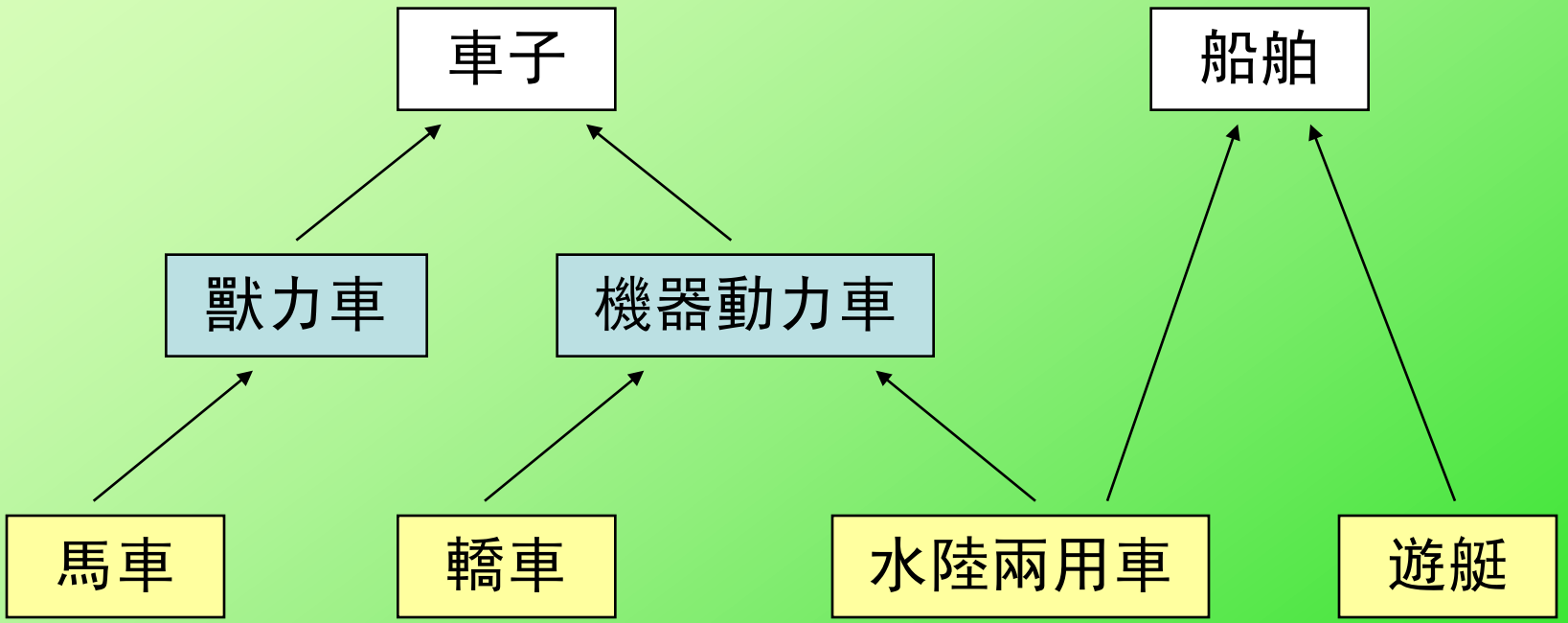
```
class B2 {...} ;
class D3 : public B1 , private B2 {
   ...
} ;
```



single inheritance , multiple inheritance

類別架構

- 若干個類別繼承關係可聯結在一起成為類別架構



class hierarchy

最常見的繼承關係

- 「公共繼承」是三種繼承方式中最常用的繼承模式

```
// 基礎類別
class Base {...} ;

// 衍生類別：公共繼承方式
class Derived : public Base {
    ...
} ;
```

- 其他兩種繼承類型在實際應用上較少出現

三種類別資料存取區域（一）

- 類別內的資料與函式都須置放在以下三種存取區域的其中之一：
  - 私有存取區　**private section**
  - 保護存取區　**protected section**
  - 公共存取區　**public section**
- 三種存取區域的差別

資料成員與成員函式	私有存取區	保護存取區	公共存有區
可為類別內的所有成員函式使用	o	o	o
可為衍生類別所有成員函式使用	x	o	o
可為類別的物件所使用	x	x	o

三種類別資料存取區域（二）

- 使用方式

```
class Base {

   private:
      int a() ;

   protected:
      int b() ;

   public:
      int c1() ;
      int c2() ;
      int c3() ;

};
...
Base foo ;

foo.a() ;   // 錯誤，a() 為私有區域內的成員函式，不得直接使用
foo.b() ;   // 錯誤，b() 為保護區域內的成員函式，不得直接使用
foo.c1() ;  // 正確
```

三種類別資料存取區域（三）

■ 各種存取方式：

```
class Derived : public Base {
  private:
    int d1(){ return a(); }          // 錯誤，a()為 Base 私有成員
    int d2(){ return b(); }          // 正確
    int d3(){ return c3(); }         // 正確
    int c1(){ return Base::c2(); } // 使用 Base::c2()
    int c2(){ ... return c2(); }   // 遞迴使用 Derived::c2()
  public:
    int d4(){ return b(); }          // 正確
    int d5(){ return c3(); }         // 正確
};
```

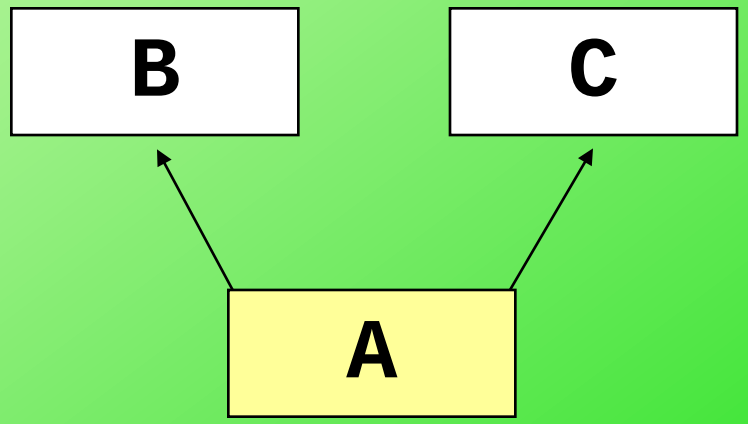
❖ 衍生類別的成員函式所能使用到的基礎類別成員與繼承方式無關

多重繼承（一）

- 多重繼承：

  衍生類別繼承了兩個以上的基礎類別

```
class B {...} ;

class C {...} ;

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

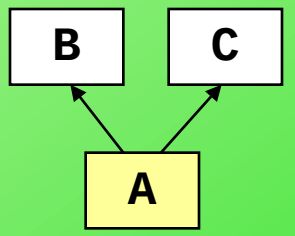


multiple inheritance

多重繼承（二）

- 使用多重繼承時，若無法明確分辨所繼承來的成員函式的來源，就須將被繼承類別名稱一併寫上

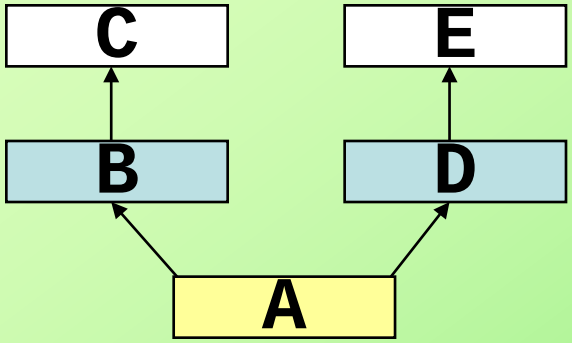
```
class B {
   public:
      int abs( int i ){ return i > 0 ? i : -i ; }
};
class C {
   public:
      double abs( double d ){ return d > 0 ? d : -d ; }
};
class A : public B , public C {...} ;
```




```
A   foo;
cout << foo.abs(2) << endl ;      // 錯誤，無法分辨 abs(2) 為
                                  // B::abs(2) 或者是 C::abs(2)

cout << foo.A::abs(2) << endl ;   // 錯誤，須使用 foo.B::abs(2)
                                  // 或 foo.C::abs(2)
```

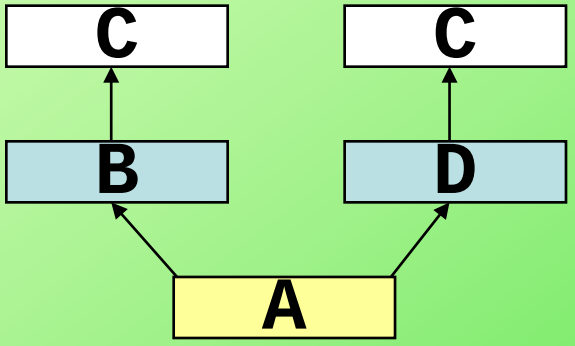
虛擬基礎類別（一）

- 若有一多層次的多重繼承為




```
class C {...} ;
class E {...} ;
class B : public C {...} ;
class D : public E {...} ;
class A : public B , public D {...} ;
```

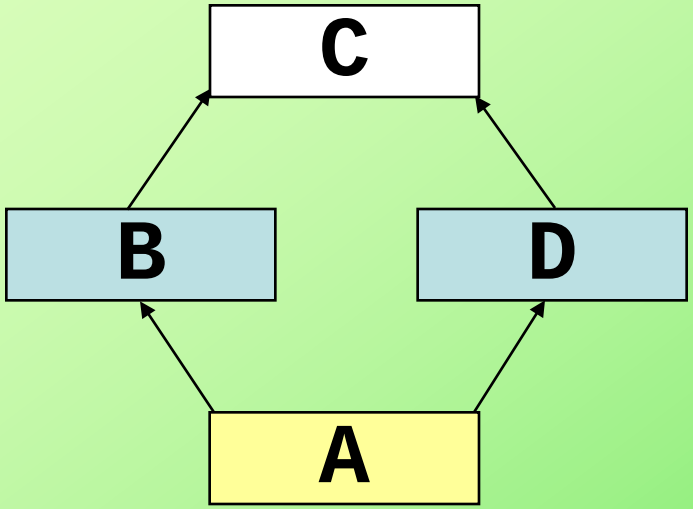
以上若類別 C 與類別 E 相同時則以上的定義方式會被解讀為




```
class C {...} ;
class B : public C {...} ;
class D : public C {...} ;
class A : public B , public D {...} ;
```

虛擬基礎類別（二）

- 之前的定義造成類別 C 重複繼承，造成資料重複儲存，可以用以下方式加以調整




```
class C {...} ;
class B : virtual public C {...} ;
class D : virtual public C {...} ;
class A : public B , public D {...} ;
```

- 此菱形架構的基礎類別 C 被稱為虛擬基礎類別

virtual base class

繼承與遺傳

■ 不同的繼承方式代表著不同型式的「遺傳」

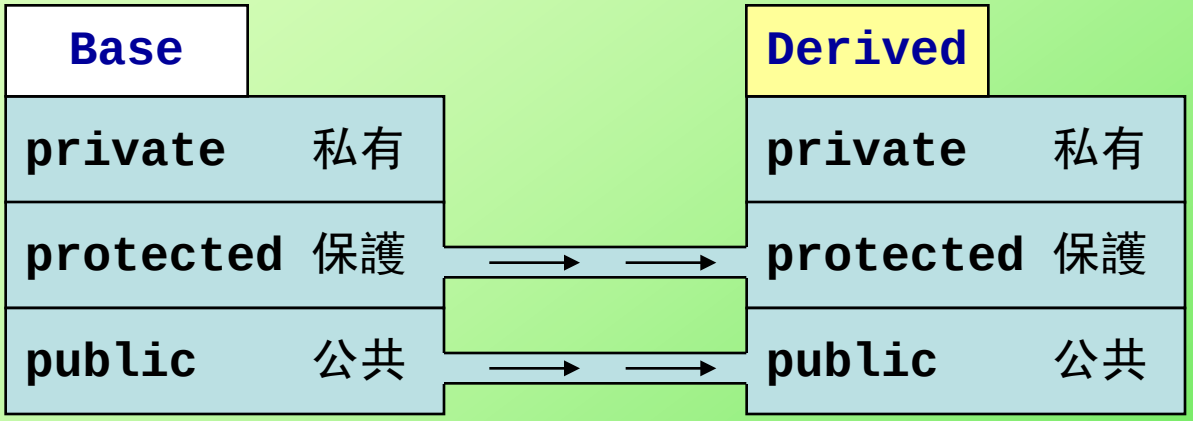
衍生類別的繼承方式	基礎類別的		
	私有區	保護區	公共區
私有繼承	不繼承	私有區	私有區
保護繼承	不繼承	保護區	保護區
公共繼承	不繼承	保護區	公共區

- 在保護繼承下，基礎類別的公共區成員被當成衍生類別保護區成員
- 在公共繼承下，基礎類別的公共區成員被當成衍生類別公共區成員

❖ 不同的繼承方式代表著不同的使用義涵

公共繼承（一）

- 資料繼承模式




```
class Derived : public Base { ... } ;
```

- 義涵：是一個 (is a)
  - 對基礎類別而言，基礎類別物件與衍生類別物件有著同樣的使用權限
  - 基礎類別物件所能執行的成員函式，每個衍生類別物件也能執行
  - 每個衍生類別物件可視為一個基礎類別物件

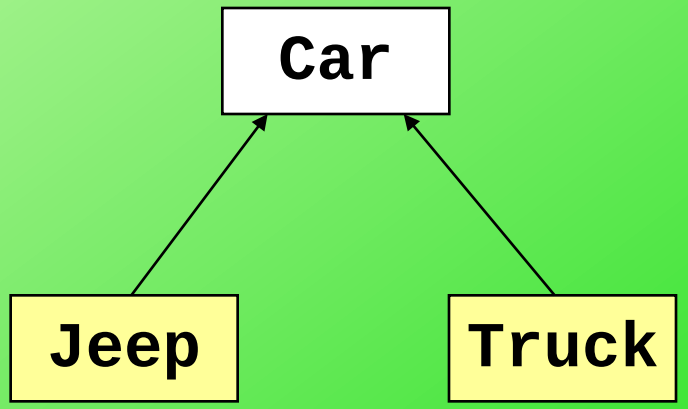
public inheritance

公共繼承（二）

- 是一個 ⟺ 公共繼承
  - 吉普車、車子、卡車

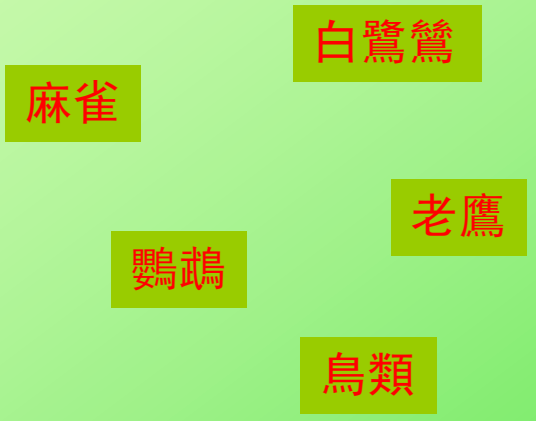
```
class Car {...} ;
class Jeep  : public Car {...} ;
class Truck : public Car {...} ;
```

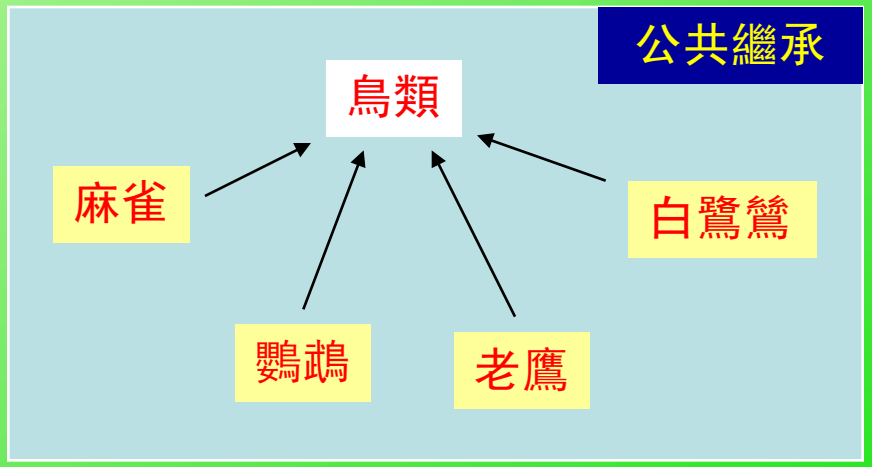
車輛類別架構



公共繼承（三）

- 在公共繼承下的類別架構
  - 基礎類別：為所有衍生類別的共同交集
  - 衍生類別：為與其它衍生類別間的個別差異





公共繼承書商範例（一）

- 相關類別

  `Book , History_Book , Kid_Book , Novel , Book_name , Author , Price , Bookstore`

- 類別關係

  每一本書籍都有書名、作者、價格　　has　有
  書局都有許多書　　has　有
  歷史書、兒童書、小說都是一種書　　is　是

公共繼承書商範例（二）

- 每本書都有書名，作者，價格

```
class Book {
   private:
      Book_Name bookname ;          // 書名
      Author    author   ;          // 作者
      Price     price    ;          // 價格
};
```

- 書店裡都有許多本書

```
const int MAX = 1000 ;
class Bookstore {
   private:
      Book  book[MAX];              // 1000 本書
};
```

公共繼承書商範例（三）

- 每一本歷史書、兒童書、小說都是一本書

```
class History_Book : public Book {...};   // 歷史書是書
class Kid_book     : public Book {...};   // 兒童書是書
class Novel        : public Book {...};   // 小說是書
```

- 書有衍生類別

```
class Book {
   protected:
      Book_Name bookname ;    // 書名
      Author    author   ;    // 作者
      Price     price    ;    // 價格
};
```

公共繼承：是一個

- 在公共繼承下，衍生類別物件都可視為基礎類別物件看待
  - 詩，五言詩

```
class Poem {...};
class Five_Short_Poem : public Poem {...};
```

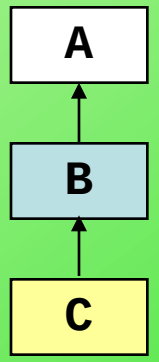
  - 列印詩的內容

```
void print_poem( const Poem& P ){...};

Poem foo ;
print_poem( foo ) ;

Five_Short_Poem bar ;
print_poem( bar ) ;      // bar 也可以傳入 print_poem
```

公共繼承：多層繼承（一）

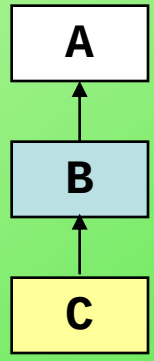
- A , B , C 三類別為

```
class A {
   protected:
      int abs( int x ){ return x > 0 ? x : -x ; }
   public:
      int square( int x ){ return x * x ; }
};

class B : public A {
   public:
      int cubic( int x ){ return x * square(x) ; }
};

class C : public B {
   public:
      int abs_sum( int x , int y ){ return abs(x) + abs(y) ; }
      int square_sum( int x , int y ){
          return square(x) + square(y) ;
      }
      int cubic_sum( int x , int y ){
          return cubic(x) + cubic(y) ;
      }
};
```



公共繼承：多層繼承（二）

- 使用方式

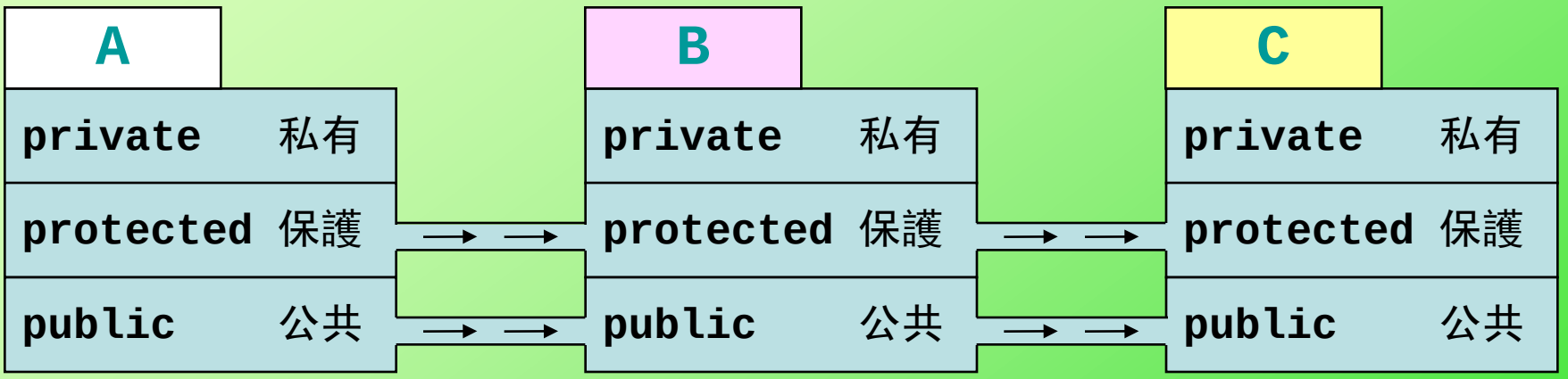



```
C  foo ;


// 錯誤，A::abs 函式並不能為類別物件所使用
cout << foo.abs(-3)           << endl ;

cout << foo.square(3)         << endl ; // 印出：9
cout << foo.cubic(3)          << endl ; // 印出：27
cout << foo.abs_sum(-3,4)     << endl ; // 印出：7
cout << foo.square_sum(3,4)   << endl ; // 印出：25
cout << foo.cubic_sum(10,1)   << endl ; // 印出：1001
```

- 任何類別物件都僅能使用定義於類別公共區的成員

公共繼承：多層繼承（三）

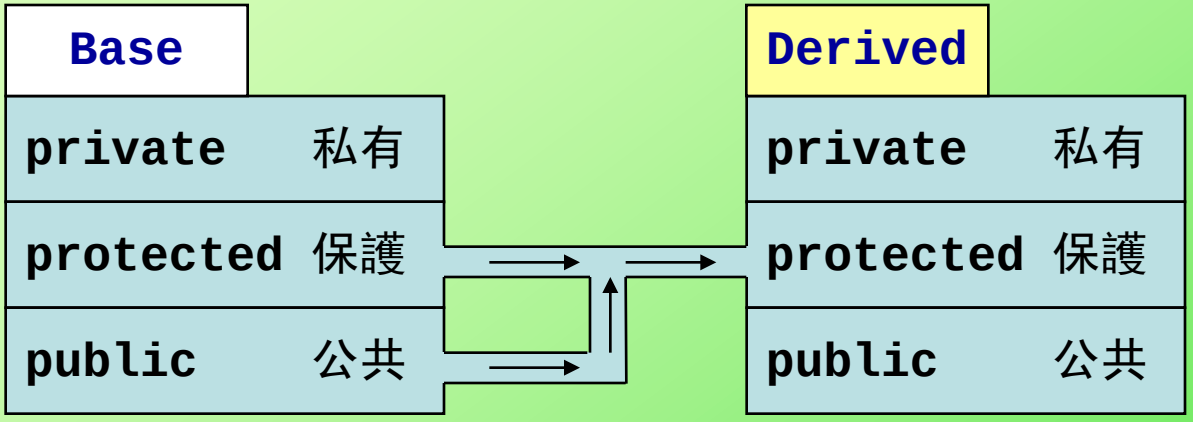
- 公共繼承類別架構下的資料繼承模式




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

保護繼承（一）

- 資料繼承模式




```
class Derived : protected Base { ... };
```

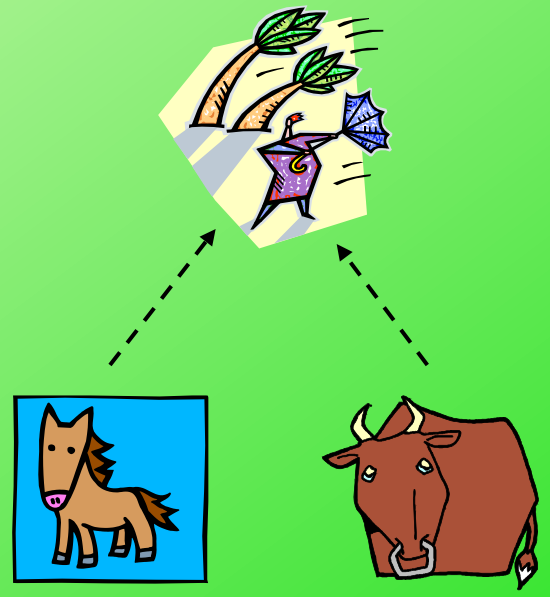
- 義涵：使用（use）
  - 衍生類別的成員函式可以自由地「使用」定義在基礎類別保護區與公共區內的成員，可避免程式碼的重複撰寫
  - 衍生類別物件無法察覺基礎類別的存在

protected inheritance

保護繼承（二）

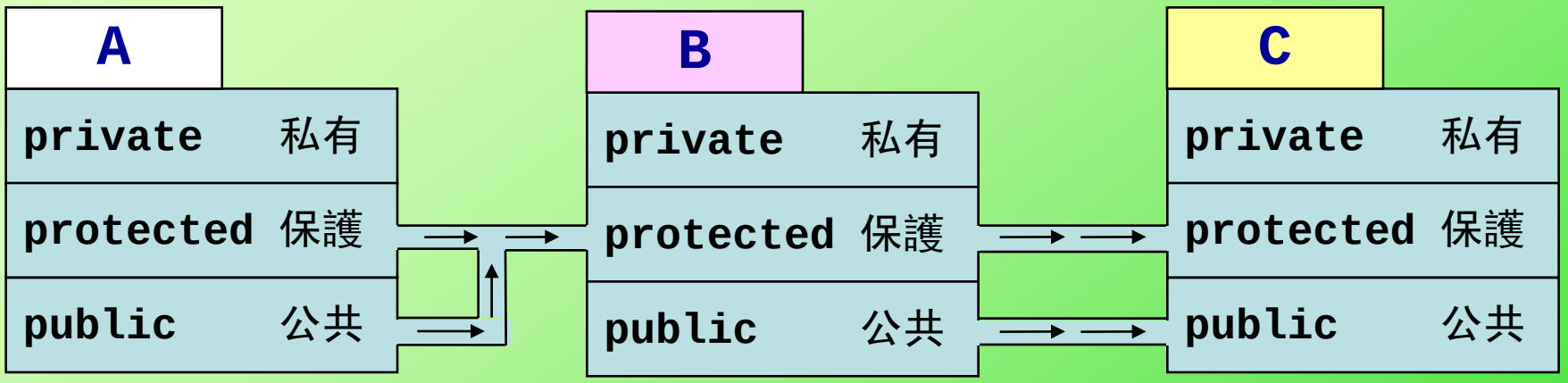
- 在保護繼承架構下，基礎類別與衍生類別並無因果、邏輯間的關係

```
class Wind {...};

class Horse : protected Wind {
    ...
};

class Cow   : protected Wind {
    ...
};
```



保護繼承：多層繼承（一）

- 使用保護繼承類別架構的資料繼承模式




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

❖ 類別 C 的成員函式仍可自由地使用在 A 類別保護區與公共區內的成員

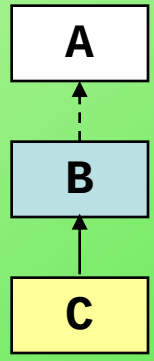
保護繼承：多層繼承（二）

- 三類別 **A , B , C** 分別為

```
class A {
   public:
      int square( int x ){ return x * x ; }
};

class B : protected A {
   public:
      int cubic( int x ){ return x * square(x) ; }
};

class C : public B {
   public:
      int square_sum( int x , int y ){
          return square(x) + square(y) ;
      }
      int cubic_sum( int x , int y ){
          return cubic(x) + cubic(y) ;
      }
};
```

保護繼承：多層繼承（三）

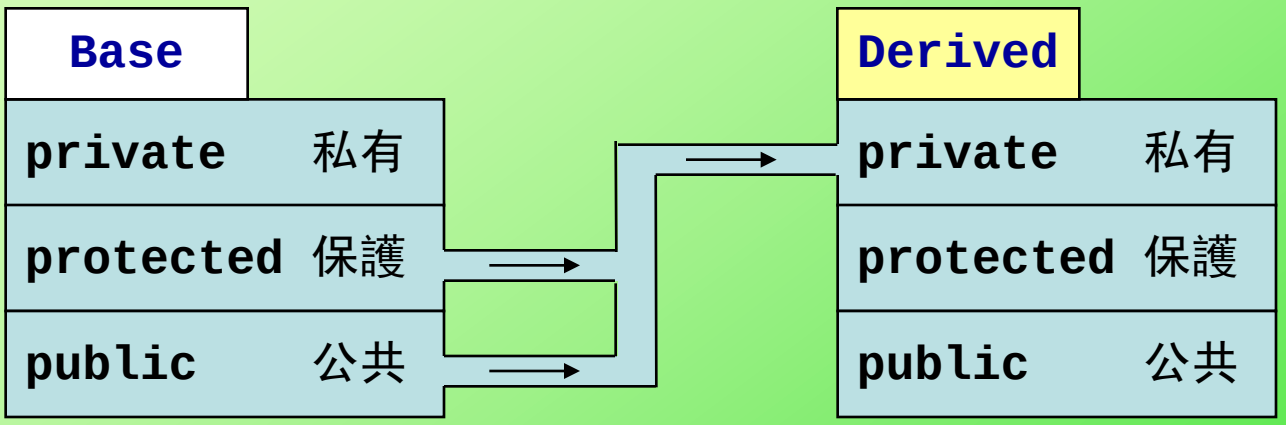
- 使用方式




```
C  foo ;


// 錯誤，C 類別的公共區內無 square 函式
cout << foo.square(3) << endl;

cout << foo.cubic(3)        << endl;   // 印出：27
cout << foo.square_sum(3,4) << endl;   // 印出：25
cout << foo.cubic_sum(10,1) << endl;   // 印出：1001
```

私有繼承（一）

- 資料繼承模式

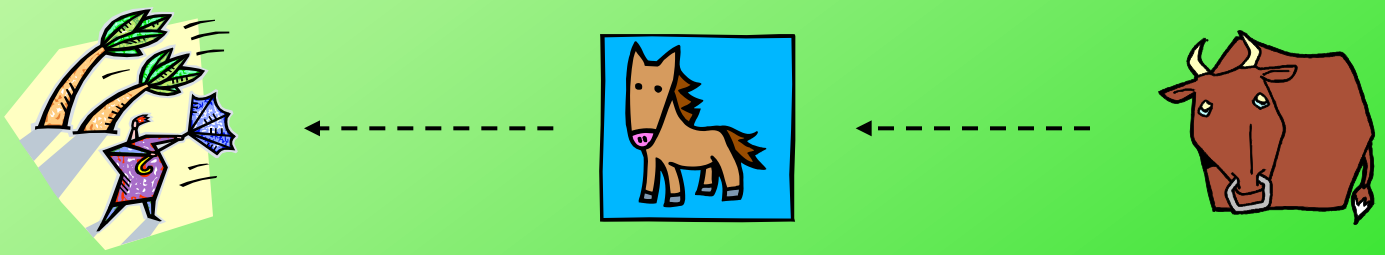



```
class Derived : private Base { ... };
```

private inheritance

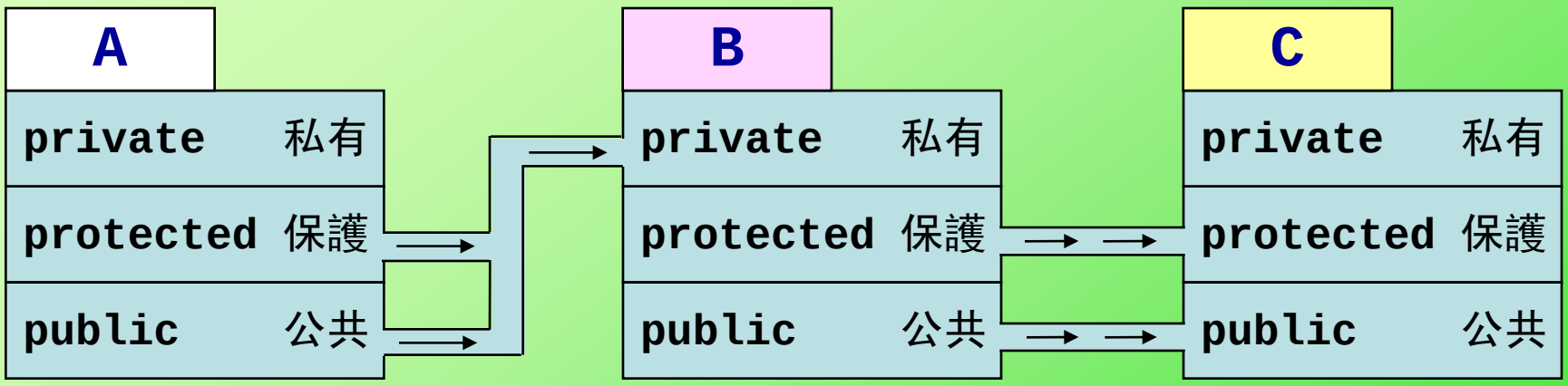
私有繼承（二）

- 義涵：利用
  - 衍生類別所繼承來的基礎類別成員僅能為其內的函式使用
  - 以衍生類別物件的觀點來看，基礎類別並不存在
  - 基礎類別與衍生類別之間不須有邏輯關係存在



私有繼承：多層繼承（一）

- 使用私有繼承類別架構的資料繼承模式




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

❖ 對 **C** 類別而言，類別 **A** 並不存在，所有類別 **A** 的可繼承成員僅止於類別 **B**

私有繼承：多層繼承（二）

- 三類別 A , B , C 分別為

```
class A {
   public:
      int square( int x ){ return x * x ; }
};

class B : private A {
   public:
      int cubic( int x ){ return x * square(x) ; }
};

class C : public B {
   public:
      int square_sum( int x , int y ){
          return square(x) + square(y) ;
      }
      int cubic_sum( int x , int y ){
          return cubic(x) + cubic(y) ;
      }
};
```

錯誤，C 類別無法取用 A 類別的 square 函式

私有繼承：多層繼承（三）

- 使用方式

```
C foo;

// 錯誤，C 類別的公共區內並無 square 函式
cout << foo.square(-3) << endl ;

// 分別印出：27  1001
cout << foo.cubic(3)          << endl ;
cout << foo.cubic_sum(10,1)   << endl ;
```

❖ 不管是使用保護繼承或私有繼承，衍生類別物件都不能使用在基礎類別公共區的成員

衍生類別物件的建構（一）

- 衍生類別物件的建構

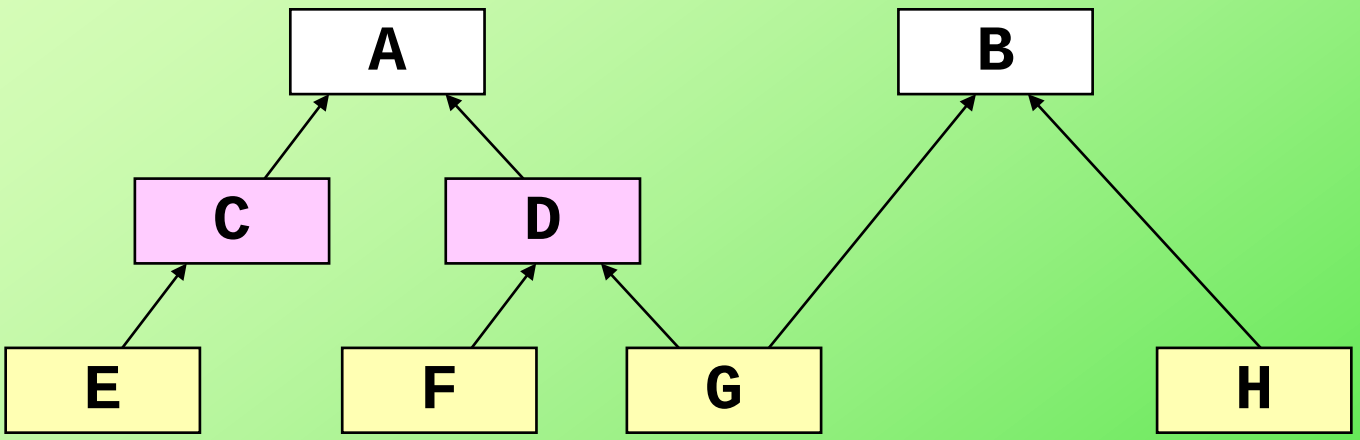
  衍生類別是建立在基礎類別的基礎之下，因此基礎類別的資料成員須在衍生類別資料成員建立之前完成

- 物件建構策略

  利用衍生類別建構函式中的初值設定列來設定直屬基礎類別的資料

衍生類別物件的建構（二）

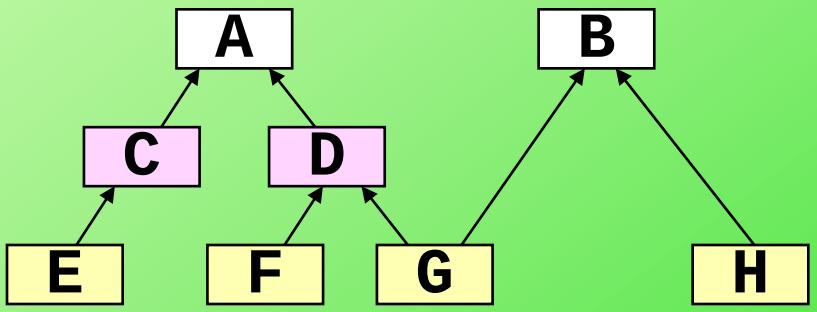
- 若一類別架構為



- 個別建構函式執行與完成的順序
  - 建構 **E** 物件　　**E -> C -> A -> C -> E**
  - 建構 **F** 物件　　**F -> D -> A -> D -> F**

衍生類別物件的建構（三）

- 類別若有兩個以上的基礎類別則基礎類別資料建構的順序與繼承次序一致的

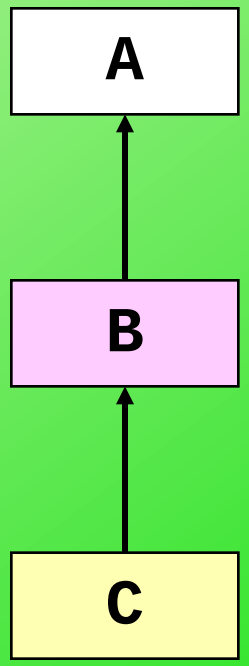


- 若是建構 G 物件
  - `class G : public D , public B {...};`
    `G -> D -> A -> D -> B -> G`
  - `class G : public B , public D {...};`
    `G -> B -> D -> A -> D -> G`

衍生類別物件的建構（四）

- 若有 A , B , C 三類別

```
class A {
   public :
      A() { ... }                          // 1
      A( int i ) { ... }                   // 2
} ;

class B : public A {
   public :
      B( int i = 0 ) { ... }               // 3
      B( int i , int j ) : A(i) { ... }    // 4
} ;

class C : public B {
   public :
      C() { ... }                          // 5
      C( int i ) : B(i) { ... }            // 6
      C( int i , int j ) : B(i,j) {...}    // 7
} ;
```



衍生類別物件的建構（五）

- 物件建構

```
C  a ;          // 遞迴執行建構函式 5 -> 3 -> 1 -> 3 -> 5
C  b(2) ;       // 遞迴執行建構函式 6 -> 3 -> 1 -> 3 -> 6
C  c(1,2) ;     // 遞迴執行建構函式 7 -> 4 -> 2 -> 4 -> 7
```

- 在衍生類別建構函式的初始設定列中，僅須執行直屬的基礎類別建構函式即可，不須設定在其上的所有被繼承類別

```
class C : public B {
    public :
        C( int i ) : B(i) , A(i) {...}      // 錯誤
};
```

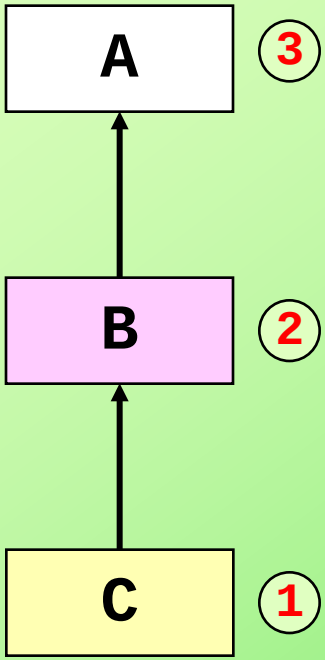
衍生類別物件的消失（一）

- 衍生類別物件的解構順序與建構的完成順序剛好相反

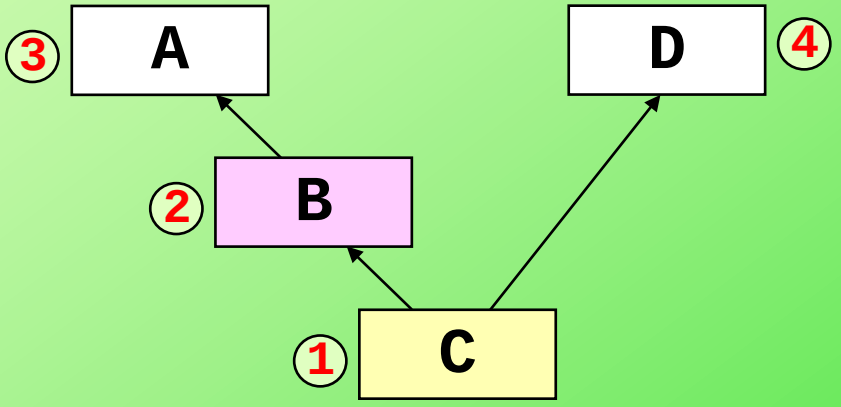
- 解構步驟：

  由衍生類別解構函式起往上依次執行基礎類別的解構函式直到最頂層為止

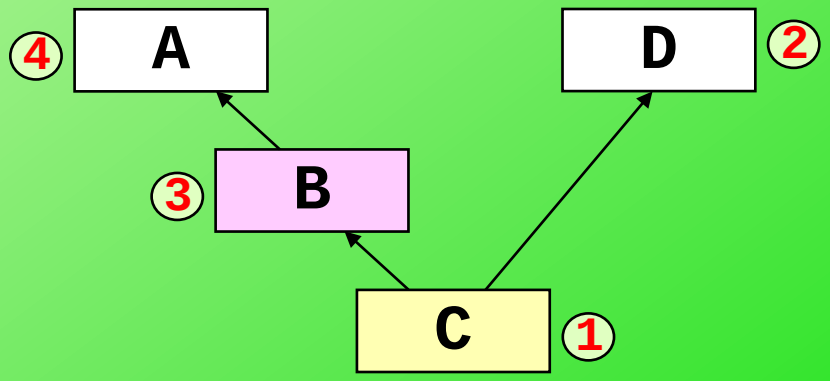
衍生類別物件的消失（二）




```
class C : public B , public D {...} ;
```




```
class C : public D , public B {...} ;
```



類別物件的複製與指定（一）

- 類別架構下的同類別物件複製：

  利用各類別的複製建構函式完成複製，複製的執行步驟與先前的物件產生時所執行的方式相同

- 類別架構下的類別若未使用到動態空間成員，則可以省略不寫，C++會自行產生

- 若某類別內有使用到動態空間成員，則仍要針對此類別撰寫複製建構函式，以防止兩物件共享記憶空間的問題發生

❖ 資料的指定動作與複製程序相同

類別物件的複製與指定（二）

- 衍生類別物件複製到基礎類別物件

  採用切割方式複製，即僅切割衍生類別所繼承來的基礎類別資料成員來複製

```
class Base {...} ;
class Derived : public Base {...} ;


Derived foo ;
Base    bar(foo) ;
```

- 基礎類別物件複製成衍生類別物件

  基礎類別的資料不足，無法複製成衍生類別物件

slicing

類別物件的複製與指定（三）

- 衍生類別物件指定給基礎類別物件

```
class Base {...} ;
class Derived : public Base {...} ;
Derived foo ;
Base    bar ;
bar = foo   ;
```

也是採用切割方式指定

❖ bar = foo 與 bar = static_cast<Base>(foo) 相同

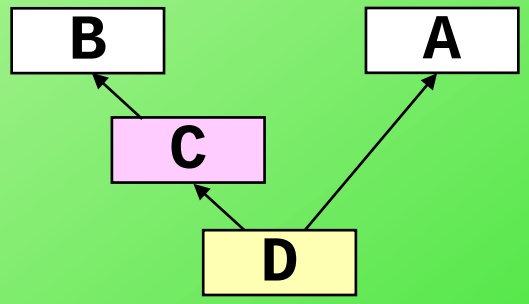
- 基礎類別物件指定給衍生類別物件

```
Derived foo ;
Base    bar ;
foo = bar   ;    // 錯誤，資料不足
```

衍生類別物件的執行效率

- 在類別架構下，建構一個越底層的衍生類別物件執行時間越久

```
class A {...} ;
class B {...} ;
class C : public B {...} ;
class D : public A , public C {...} ;
...
for ( int i = 0 ; i < 100 ; ++i ) {
   D  foo ;        // 缺乏效率
   ...
}
```

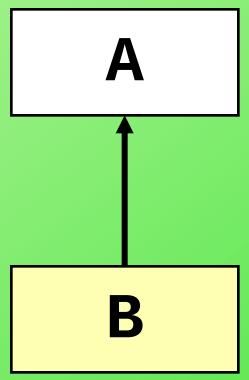


- 增進效率方式：
  - 經常使用的衍生類別在設計上儘量放置於類別架構上層
  - 在底層的衍生類別物件儘量避免經常性的產生與消失

衍生類別與基礎類別的成員函式

- 避免在衍生類別內定義與基礎類別相同特徵資料的成員函式

- 在公共繼承下的類別關係，每個衍生類別物件都可視為基礎類別物件，衍生類別物件執行基礎類別的公共成員函式應產生相同的結果，因此沒有必要在衍生類別內重複定義相同特徵資料的成員函式



點、多邊形、三角形（一）

■ 類別關係

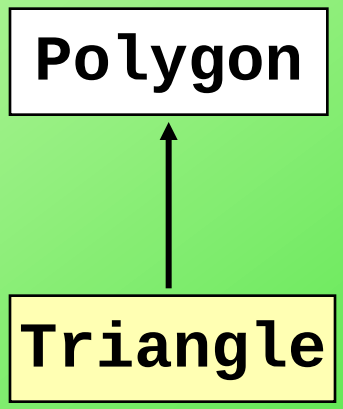
- 每一多邊形、三角形都「包含」若干個點 → 嵌入
- 每一個三角形都「是一個」多邊形 → 公共繼承

```
struct Point {
   double x , y ;
   Point( double a , double b ) : x(a) , y(b){}
} ;

class Polygon {
   protected :
      vector<Point> vertex ;
   public :
      int vertex_no() const { return vertex.size() ; }
      Point geometrical_center() const { ... }
} ;

class Triangle : public Polygon {
   public :
      Triangle() {}
      Triangle( Point pt1 , Point pt2 , Point pt3 ){
         vertex.push_back( pt1 ) ; vertex.push_back( pt2 ) ;
         vertex.push_back( pt3 ) ;
      }
      Point circumcenter() const {...}
} ;
```

點、多邊形、三角形（二）

- 避免在三角形類別定義以下函式

```
int  Triangle::vertex_no() const {
   return 3 ;
}
```



❖ 在公共繼承的類別架構下，基礎類別是所有衍生類別的交集，而個別的衍生類別內定義的成員函式則是衍生類別間的個別差異

靜態型別與動態型別（一）

- 相同的成員函式介面，不同的實作程式

```
string Polygon::name()  const { return ”多邊形” ; }
string Triangle::name() const { return ”三角形” ; }
```

若定義函式

```
void print_vertex_no( const Polygon& p ) {
    cout << p.name() << ” : ” << p.vertex_no() ;
}
```

靜態型別與動態型別（二）

- 多邊形物件

```
Polygon foo ;                    // 產生一多邊形物件
...
print_vertex_no( foo ) ;         // 列印多邊形與頂點數
```

- 三角形物件

```
Triangle bar ;                   // 產生一三角形物件
...
print_vertex_no( bar ) ;         // 列印三角形與 3
```

靜態型別與動態型別（三）

- 之前的函式參數設定中，C++執行了

```
// 這裡 bar 是 Triangle 物件，p 是 bar 的參考
const Polygon& p = bar ;
```

  - ❖ 以上等號的左右兩邊型別不同

- 靜態型別：

  等號左邊的型別，在宣告時原有的固定型別，即 **Polygon**

- 動態型別：

  等號右邊的型別，執行時所真正參考的型別，即 **Triangle**

❖ 靜態型別在編譯時就可確定，動態型別在執行時才能確定

static type dynamic type

靜態型別與動態型別（四）

- 雙型別型式的物件也可能出現在指標型物件

```
// p 的靜態型別為 Polygon ，動態型別與 bar 型別相同
Polygon *p = &bar ;
```

❖ 沒有使用參考或指標類型的物件則沒有所謂的動態型別

虛擬函式（一）

- 類別架構內的函式區分為兩大類型
  - 虛擬函式：成員函式之前有 **virtual** 保留字
  - 非虛擬函式：成員函式之前無 **virtual** 保留字

```
class Polygon {
    ...
    public :
        virtual string name() const { return "多邊形" ; }
} ;
class Triangle {
    ...
    public :
        virtual string name() const { return "三角形" ; }
} ;
```

虛擬函式 (二)

- 虛擬函式為動態設定函式
  - 物件執行虛擬函式時，其所真正執行的成員函式是定義在物件的動態型別內
- 非虛擬函式為靜態設定函式
  - 物件執行非虛擬函式時，其所真正執行的成員函式是在物件的靜態型別內

dynamically bound, statically bound

虛擬函式（三）

- 多邊形類別

```
class Polygon {
    ...
   public :
      virtual string name() const { return ”多邊形” ; }
} ;
```

則在

```
void print_vertex_no( const Polygon& p ) {
    cout << p.name() << ” : ” << p.vertex_no() ;
}
```

中的

p.name()	執行 p 動態型別內的	name()
p.vertex_no()	執行 p 靜態型別內的	vertex_no()

❖ 若動態型別內並無定義對應的虛擬函式，則 C++ 會執行在靜態型別內的函式

虛擬函式的涵義（一）

- 虛擬函式：動態型別的對應函式

  由於動態型別須留待執行時才能確定，因此真正執行的程式碼也隨之不同

- 虛擬函式的涵義
  - 程式執行同樣的函式介面，但卻可因真正執行實體不同而造成不一樣的結果
  - 虛擬函式 → 多型函式

polymorphic function

虛擬函式的涵義（二）

- 多型函式：

  在程式正式執行後才能決定函式介面所對應的程式實體

```
int kind ;
cout << "輸入物件：1 為 Polygon, 2 為 Triangle " ;
cin >> kind ;

// 若是 1 則動態產生 Polygon 物件 , 2 為 Triangle 物件
// 再由 Polygon 指標 ptr 指向此新產生的物件
Polygon *ptr = ( kind == 1 ? new Polygon :
                             new Triangle ) ;
```

虛擬函式的涵義（三）

- 虛擬函式

  相同的使用介面，但執行不同型別的程式實體

- 非虛擬函式

  相同的使用介面，且執行固定的程式實體

各種成員函式的使用時機 (一)

- 虛擬函式：

  若某函式為類別架構內所有類別共同擁有的功能或特徵，且會因類別的不同導致運算方式有所差異

- 非虛擬函式：

  若某函式為類別架構內的所有類別共同擁有的功能或特徵，但不會因類別的不同而運算方式有所差異

- 一般成員函式：

  函式僅是某類別所特有的功能

各種成員函式的使用時機 (二)

- 幾何範例

  - 類別：幾何圖形 (Geometric_Entity)
    三角形 (Triangle)
    圓 (Circle)

  - 函式：計算周長 (perimeter)
    面積 (area)
    顯示顏色 (display_color)
    圓心 (circle_center)
    內接圓圓心 (incenter)

各種成員函式的使用時機 (三)

- 類別關係：
  - 每一個三角形與圓形「都是一個」幾何圖形

```
struct Point { double x , y ; } ;
enum Color { red , green , blue , cyan , yellow } ;

class Geometric_Entity {
    protected :
        Color entity_color ;      // 幾何圖形的顏色
} ;
class Triangle : public Geometric_Entity {
    private :
        Point pt1 , pt2 , pt3 ;   // 三頂點座標
} ;
class Circle : public Geometric_Entity {
    private :
        Point  center ;    // 圓心
        double radius ;    // 半徑
} ;
```

各種成員函式的使用時機 (四)

- 函式分類
  - 周長、面積計算為所有圖形的共同功能但圖形不同，計算方式也不同
    - ➔ 虛擬函式
  - 顯示顏色為所有圖形的共同功能，但顯示方式不因類別不同而有所差異
    - ➔ 非虛擬函式
  - 圓心、內接圓圓心為分別為圓形與三角形類別所特有功能
    - ➔ 各類別內的一般成員函式

各種成員函式的使用時機 (五)

- 成員函式定義

```
class Geometric_Entity {
    public :
        virtual double perimeter() const { ... }
        virtual double area()      const { ... }
        Color   display_color()    const { ... }
} ;
```

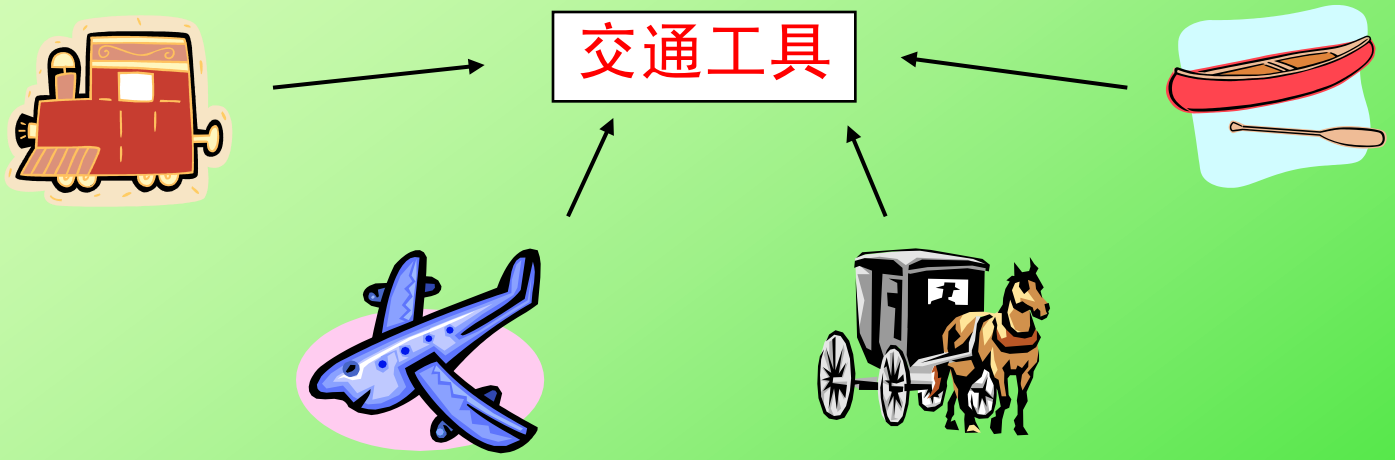
```
class Triangle : public Geometric_Entity {
    public :
        virtual double perimeter() const { ... }
        virtual double area()      const { ... }
        Point   incenter()         const { ... }
} ;
```

```
class Circle : public Geometric_Entity {
    public :
        virtual double perimeter() const { ... }
        virtual double area()      const { ... }
        Point   circle_center()    const { ... }
} ;
```

抽象基礎類別與純虛擬函式 (一)

- 抽象基礎類別

  不能產生實體物件的基礎類別



❖ 交通工具的實體都是屬於衍生類別的物件，交通工具為所有衍生類別的公約數，無實體可言

Abstract Base Class

抽象基礎類別與純虛擬函式 (二)

- 純虛擬函式：無實作程式碼的虛擬函式

```
class  Vehicle {
    public :
        virtual bool use_engine() const = 0 ;
};
```

❖ 在語法上，純虛擬函式為虛擬函式末尾加上 = 0

抽象基礎類別與純虛擬函式 (三)

- 純虛擬函式無實作碼，因此所有繼承此類別的衍生類別都須要定義此虛擬函式的實作程式碼

```
class Car : public Vehicle {
    ...
    public :
        bool use_engine() const { return true ; }
} ;
class Horse_wagon : public Vehicle {
    ...
    public :
        bool use_engine() const { return false ; }
} ;
```

抽象基礎類別與純虛擬函式 (四)

- 抽象基礎類別語法

  至少含有一個純虛擬函式的基礎類別

```
class  Vehicle {
    public :
        virtual bool use_engine() const = 0 ;
};
```

❖ 以上 **Vehicle** 為抽象基礎類別

抽象基礎類別與純虛擬函式 (五)

- 抽象類別不能產生物件，卻可用來當成衍生類別物件的參考或指標

```
Vehicle a ;           // 錯誤，Vehicle 是一個抽象類別
Car     b ;
Vehicle &c = b ;      // c 為汽車類別物件 b 的參考物件


// d 為一抽象類別指標指到馬車類別
Vehicle *d = new Horse_wagon ;

// e 為一抽象類別指標指到馬車類別
Vehicle *e = &d ;
```

抽象基礎類別與純虛擬函式 (六)

- 抽象類別可以使用建構函式來設定其內的資料成員

```
class Car {
    protected :
        int passanger_no ;
    public :
        Car( int no ) : passenger_no(no) {}
        virtual int tire_no() const = 0 ;
} ;
class Sedan : public Car {
    public :
        Sedan( int no = 5 ) : Car(no) {};
} ;
```

- 為避免抽象類別的建構函式被誤用，可以將其放到保護區內隱藏起來

```
class Car {
    protected :
        int passenger_no ;
        Car( int no ) : passenger_no(no) {}
} ;
```

各種繼承函式的比較

- 函式型式與繼承內容

函式型式	衍生類別所繼承的內容
非虛擬函式	繼承函式介面與其程式的實作部分
一般虛擬函式	繼承函式介面與備用的程式實作部分
純虛擬函式	僅繼承函式介面

虛擬解構函式 (一)

- 物件消失時所執行的解構函式順序與建構函式相反

```
class Base {
    ...
    public :
        ~Base() { cout << "Base" << endl ;}
} ;
class Derived : public Base {
    ...
    public :
        ~Derived() { cout << "Derived" << endl ;}
} ;
...
Base     foo ;    // 產生一基礎類別物件
Derived  bar ;    // 產生一衍生類別物件
```

- ❖ 當 **foo** 物件消失時，螢幕印出 **Base**
- ❖ 當 **bar** 物件消失時，螢幕印出 **Derived Base**

虛擬解構函式 (二)

- 衍生物件指標去除動態衍生物件

```
Derived * p = new Derived ;
...
delete  p ;          // 印出 Derived  Base
```

❖ 當指標 **p** 所指向的物件被去除時，程式依次執行指標 **p** 靜態型別與其繼承來的基礎類別的解構函式

- 基礎物件指標去除動態衍生物件

```
Base *  q = new Derived ;
...
delete  q ;          // 印出 Base
```

❖ 當指標 **q** 所指向的物件被去除時，程式僅執行 **q** 靜態型別的解構函式

➔ 動態衍生物件的衍生類別資料部份並沒有被去除 !!!

虛擬解構函式 (三)

- 虛擬解構函式

```
class Base {
    public :
        virtual ~Base() { cout << ”Base” << endl ; }
} ;
class Derived : public Base {
    public :
        ~Derived() { cout << ”Derived” << endl ; }
} ;
```

```
Derived * p = new Derived ;
...
delete  p ;        // 印出 Derived  Base
```

```
Base * q = new Derived ;
...
delete  q ;        // 印出 Derived  Base
```

virtual destructor

虛擬解構函式 (四)

- 虛擬解構函式的使用時機
  - 使用到虛擬函式的類別架構
  - 衍生類別使用到動態資料成員

```
class Base {
    ...
    public :
        virtual ~Base() {}
} ;

class Derived : public Base {
    private :
        int *data ;
    public :
        ...
        ~Derived() { if ( data != NULL ) delete data ; }
} ;
```

虛擬解構函式 (五)

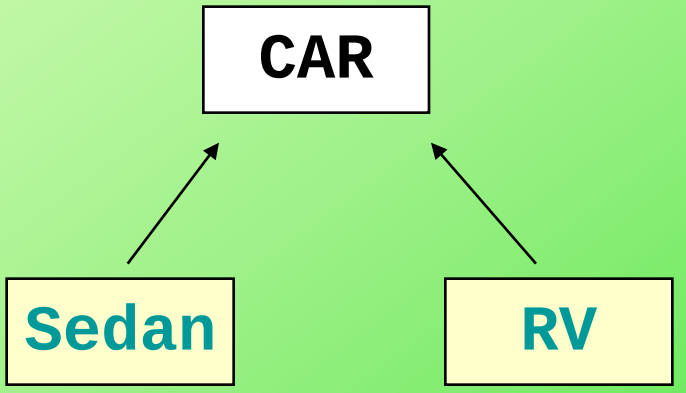
- 虛擬解構函式可以簡單到空函式，但不能省略

- 可以將解構函式定義成純虛擬函式藉以設定基礎類別為抽象基礎類別，但仍須撰寫空實作的虛擬函式，不能省略

```
class Base {
    ...
    public :
        // 定義純虛擬解構函式
        virtual  ~Base() {} = 0 ;
} ;

// 空實作的解構函式
Base::~Base() {}
```

類別架構下的型別轉換 (一)

- 若有一類別架構如右




```
Car    *car = new Sedan ;
Sedan *sedan ;

sedan = car  ;  // 錯誤，無法將 Car 指標轉成 Sedan 指標
```

- 上式的左邊 **sedan** 為指向 **Sedan** 的指標，右邊 **car** 為指向 **Car** 的指標，指定運算子無法在靜態型別間轉型

類別架構下的型別轉換 (二)

- **dynamic_cast**：動態型別的轉換

```
int kind ;
Car *car ;

cin >> kind ;
car = ( kind == 1 ? new Sedan : new RV ) ;

Sedan *s1 = dynamic_cast<Sedan*>(car) ;     // 使用指標

Sedan &s2 = dynamic_cast<Sedan&>(*car) ;    // 使用參考

Sedan &s3 = dynamic_cast<Sedan>(*car) ;     // 錯誤
```

類別架構下的型別轉換 (三)

- 動態轉型不見得會成功，轉型後都要測試

```
Car  *car ;    // 定義 Car 指標
...
Sedan *sedan = dynamic_cast<Sedan*>(car) ;

// 確定是否轉型成功
if ( sedan )
    cout << sedan->tire_no() << endl ;
else
    cout << "動態轉型失敗" << endl ;
```

❖ 當動態轉型失敗時，C++ 會回傳 0 給指標

大學、學院、學系類別 (一)

- 相關類別
  - University 大學
  - College 學院
  - Dept 學系
  - Science 理學院
  - Engineering 工學院
- 類別關係
  - 每個大學包含著若干學院 → has a → 嵌入
  - 每個學系包含著若干學系 → has a → 嵌入
  - 理學院、工學院是學院 → is a → 公共繼承

大學、學院、學系類別 (二)

- 類別程式

```
class University {
   private :
      string           name ;        // 學校的名稱
      vector<College> colleges ;     // 學校內的學院
      ...
} ;
class College {
   protected :
      string           name ;        // 學院的名稱
      vector<string> depts ;         // 學院內的學系
} ;

class Science     : public College { .. } ;
class Engineering : public College { .. } ;
```

大學、學院、學系類別 (三)

- 學院為一抽象概念，無任何實體，須利用純虛擬函式設定其為抽象基礎類別，同時將 **University** 類別的資料成員

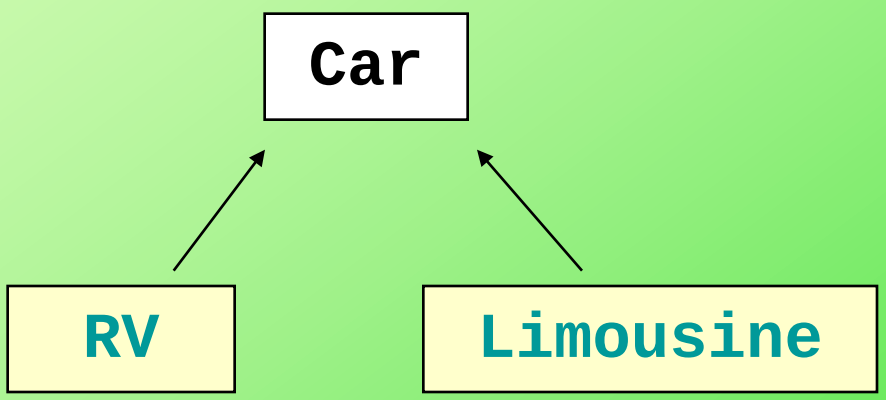
```
    vector<College>
➔   vector<College*>
```

以避免學院類別物件的建立

程式　輸出

租車商車子管理 (一)

- 若車子類別架構為



- Car 為抽象基礎類別

租車商車子管理（二）

- 物件使用方式

```
vector<Car*>  cars ;
cars.push_back( new RV(...) ) ;
cars.push_back( new Limousine(...) ) ;
...
RV  *rv ;

// 僅列印 cars 陣列中所有的 RV 車資料
for ( int i = 0 ; i < cars.size() ; ++i ) {
   if ( rv = dynamic_cast<RV*>(cars[i]) )
        cout << *rv << endl ;
}
```

程式　輸出