12.9 虛擬解構函式 — 補充

當物件離開了其存在領域之後，C++ 隨即執行物件所屬類別內的解構函式，清除物件所佔有的記憶空間，歸還系統重新使用。然而在類別架構下 (class hierarchy)，當衍生類別物件離開其存在領域後，C++ 就須考慮執行誰的解構函式。在正常的情況下，當衍生類別物件消失時，程式會以與執行建構函式相反次序方式來執行解構函式。例如，若有一類別架構與物件如下:

```
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 foo
Derived  bar ;  // 產生一衍生類別物件 bar
```

當基礎類別物件 `foo` 消失時，C++ 會因執行基礎類別的解構函式而列印 `Base`。同理，當衍生類別物件 `bar` 消失時，螢幕會依次顯示 `Derived`，`Base` 兩行文字，似乎一切都是以預期的方式來進行記憶空間的回收。

但當衍生物件是以動態的方式產生，且為某一基礎類別的指標所指向，則以上的記憶空間資源回收方式就會產生漏洞，例如:

```
Base * ptr = new Derived ;
...
delete  ptr ;   // 僅執行基礎類別的解構函式 !!!
```

當基礎類別的 `ptr` 指標被清除 (`delete`) 時， C++ 僅是執行定義在基礎類別的解構函式，而 `ptr` 指標所真正指向的衍生類別解構函式卻沒被執行，如此一來會造成衍生類別物件所佔用的記憶空間無法在物件消失之前被加以回收處理。解決這種資源釋放問題的一個簡單方法就是將基礎類別的解構函式虛擬化，使得基礎類別指標在進行清除 (`delete`) 時，先由指標的動態型別[445] (dynamic type) 開始清除，之後再接續執行於基礎類別內的解構函式，後者同時也是虛

擬解構函式 (virtual destructor) 的特點，因為一般的虛擬函式僅會執行定義於動態型別內的函式。因此上述類別架構內的解構函式須改為：

```
class  Base {
   ...
   public :
      virtual  ~Base() { cout << "Base" << endl ; }
} ;

class  Derived : public  Base {
   ...
   public :
      ~Derived() { cout << "Derived" << endl ; }
} ;
```

讀者須留意，並不是所有類別架構的基礎類別都須要定義成虛擬解構函式。一般的原則是如果基礎類別須至少設計一個以上的虛擬函式，則此時也要同時將基礎類別的解構函式更改成虛擬解構函式，使得動態產生的衍生類別物件得以透過基礎類別指標的運作，在操作完畢後可以正確地釋放出其所佔用的記憶空間資源。

此外即使基礎類別內並無任何虛擬函式，而繼承的公共衍生類別內有使用到動態資料成員，若程式中有使用基礎類別指標來清除動態產生的衍生類別物件，則記憶空間流失[114]的問題就等著被發生。例如：

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

class  Derived : public  Base {
   private :
      int  *data ;   // 指向動態產生的空間
   public :
      ...
      ~Derived() { if ( data != NULL ) delete data ; }
} ;
...
Base  *ptr = new Derived ;
...
delete  ptr ;    // 僅執行 Base 內的解構函式
```

以上最後一行的 `delete ptr` 敘述僅執行基礎類別的解構函式，衍生類別的解構函式並未被執行，因此衍生類別物件所動態配置的記憶空間就無法被去除，自然造成所謂的記憶空間流失問題。

有時虛擬解構函式可以簡單到不須有任何敘述，此時並不能將其加以省略不寫，但只要將其定義為一個空殼的虛擬解構函式即可。例如:

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

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

程式設計員也可以將解構函式定義為純虛擬解構函式 (pure virtual destructor)，藉以將基礎類別變成為抽象基礎類別[442] (abstract base class)，此時仍須在基礎類別之後定義解構函式的實作碼，即使其為空的實作也不能加以省略，例如:

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

// 虛擬解構函式
Base::~Base() { ... }
```