8. More on Classes and Data Encapsulation

在本章裡，我們將進一步探討類別之資料的抽象化和隱藏封裝。內容為進階的主題，如 const 函式成員與物件、this 指標、朋友(friend)函式與類別、靜態(static)資料成員與函式成員等。

8.1 const Function Member and const Object in C++

8.1.1 const Function Member

const 函式成員，其函式內容只能讀取資料成員的資料，而不能更改資料成員的資料。對於只用於讀取或列印的函式，可將其設為 const，以保護資料成員。

要將函式成員指定為 const，需同時於函式原型定義與宣告的參數列(…..)後加上關鍵字 const。

例；
```
void    print12HR( ) const{ ……}

void    print24HR( ) const;
void    Time::print24HR( ) const {…….}
```

程式範例: oop_ex64.cpp

8.1.2 const Object

有些物件需要能被修改其狀態，有些則不需要。我們可使用 const 關鍵字產生 const 物件，指定其資料成員是不可修改的(均為常數)。

例；
```
const    Time    noon(12, 0, 0);
```

對於 const 物件，compiler 只允許呼叫使用其 const 函式成員，non-const 函式成員可更改物件的資料成員，故 compiler 對資料成員應為常數之 const 物件不允許呼叫。

程式範例: oop_ex65.cpp

注意要點:

1. 建構函式與解構函式不可宣告為 const。
2. const 函式成員內不可呼叫使用 non-const 函式成員。
3. Java 可用 final 關鍵字宣告 final classes 與 methods，但其意義與 C++之 const 完不同，final class 代表此類別不可被繼承，final method 則代表此函式不可被子類別所重新定義(overridden)。

8.2 this Keyword

8.2.1 this Pointer in C++

說明:

對於 C++的每一個物件，其內部函式成員可使用一隱性的指標，取得物件本身的記憶體位址，此指標即為 this 指標，其固定指到此物件本身(儲存值為此物件本身之記憶體位址)。

this 指標並不是物件本身的一部分(隱性成員)，而是當物件呼叫任何一個 non-static 函式成員時，compiler 會將 this 指標當作隱性的第一個 argument 傳給函式，其值為 Current Object(函式成員被呼叫者)之位址。(static member 將於稍後介紹) 。

this 指標是 compiler 隱性地用來參照物件的資料成員與函式成員，但也可以被明確地在程式中使用，this 指標的型別(type)為此類別的定址指標(classname* const)，如對 const 函式成員，則亦為唯讀(const classname* const)

this 指標的最大功用，在於將物件本身(*this)作為傳回值傳回。最常用於運算子覆載，例如拷貝指定運算子(=)。

this 指標的另一個功用，是可達成接續式的呼叫函式成員(concatenated function calls)，亦即使用*this 與傳參考的方式傳回物件本身，使得同一 statement 中可接續地呼叫同一物件的函式成員或覆載之運算子(例：y=a*b+c;)。

程式範例: oop_ex66.cpp　oop_ex67.cpp

8.2.2 this Reference in Java

Java 的每一個物件亦均隱藏著一 this 參考，其基本性質與作用與 C++相同。除此之外，Java 之 this 參考另有以下功能。

1. Java 類別的建構函式中，可使用 this 來呼叫其他建構函式。
2. Java 類別的函式成員之參數名稱，可與資料成員同名，不過這時資料成員將被同名之參數所隱藏，必需要使用隱含的 this 參考來存取資料成員。

程式範例: oop_ex68.java

8.3 Member Initializers in C++

說明:

我們可於 C++的程式中，產生參考與常數(const 變數)，產生此兩種資料型態時有個重要條件，那就是在其宣告的同時，也必須為其設定初值。例如，以下為不合法的程式。

```
void main( )
{
    int i;
    int& r;
    const int j;
    int * p;
}
```

以下為合法的程式。

```
void main( )
{
    int i;
    int& r = i;
    const int j = 10;
    int * p;
}
```

C++的類別除了可用變數、指標與物件作為資料成員外，也可以使用參考或常數作為其資料成員。

程式範例: oop_ex69.cpp

1. 但根據之前的介紹，宣告類別內的資料成員時，並不能同時對其初始化(設定初值)，如此一來，類別內的參考或常數成員便會發生無法初始化的問題。

2. 原則上，C++編譯器不能接受函式內具有未初始化的參考或常數，但可用參考或常數作為資料成員，故以範例之類別之製作可通過編譯，但與一般類別不同，編譯器不提供 default constructor，故程式之錯誤在於沒有預設建構函式，而非類別之內容。

3. 參考或常數只能被初始化，而不能指定(assign)其值。物件之資料成員是在執行建構函式之前便已先宣告產生，於建構函式區塊內的「=」均是指定運算，而非成員之初始化，故於建構函式區塊內以來不及做成員之初始化，因為成員早已產生。(非 int& r = i; 而是 int& r; r = i; 故錯誤)

對於參考或常數初始化之問題，C++提供了特別的初始化方式，即使用成員初值設定運算子(Member Initializers)，其格式如下：

格式:

```
類別名稱( )：參考或 const 變數(初值)   // <= 初值設定運算子
{
    其他函式成員值之設定(使用=);
}
```

例:

```
class X
{
    int& r;
    const int n;
    int i;
public:
```

```
        X(int& s, int t, int u ):r(s), n(t)
        {
              i = u;
        }
        …….
}
```

程式範例: oop_ex70.cpp

1. 使用初值設定運算子，其效果如同於物件產生之同時宣告它的資料成員，並對其指定初值(範例之呼叫有如 int& r = s; const int n = t;)。

2. 對於為參考的資料成員，其通常是要參考到呼叫程式中的變數，而非建構函式中的區域變數，故於此情況，使用的參數應為參考型態(呼叫變數的參考)，因為參數為函式內的區域變數(範例之呼叫有如 int& s = y; int& r = s;)。

8.4 Friend Functions and Friend Classes in C++

8.4.1 Friend Functions

說明:

在資料封裝的要求下，一類別的私有(private)成員(還有繼承時會提到的 protected 成員)，只能被同類別內之函式成員存取使用。

對此 C++提供了一例外的情況，假如希望某一類別外的函式可以直接來存取一類別的私有資料成員，此類別可將這函式宣告為其的朋友(friend)函式，賦予此外部函式可直接存取其私有成員的權限。

使用朋友函式可增進程式的效率，但也破壞了資料封裝的原則，故只有特殊情況才考慮使用，例如運算子覆載與容器類別。

一朋友函式的宣告格式如下，其做法是於類別的定義區塊中，將該函式的原型宣告前加上關鍵字 friend，由於朋友函式不屬於類別的成員，故其定義需於類別的定義區塊外，只能以其原型宣告代表之。有關 private、protected、public 等

成員之存取設定與朋友函式無關(因其非成員)，故朋友函式的宣告可置於定義區塊中的任何位置，不過習慣上是集中放在定義的最前端或最後端。朋友函式之定義需位於賦予權限之類別的定義之後。

格式:

```
class 類別名稱
{
    friend  朋友函式原型宣告；
}

朋友函式定義;
```

例:

```
class Person
{
    ……
    friend void setName(Person& p, string c);
    …….
}

void setName(Person& p, string c)
{
    p.name = c;
}
```

程式範例: oop_ex71.cpp

注意要點:

1. 朋友函式的使用權限，是由某類別所賦予(grant)給某函式，而非自函式向某類別取得。
2. 對於朋友函式，賦予其存取權限之類別的物件通常為函式之參數列之一。於朋友函式內，對於此物件之私有資料成員，即可以用「.」或「->」直接存取。
3. 朋友函式並非類別內的成員，故沒有 public 或 private 之分。

程式範例: oop_ex72.cpp

注意要點:

朋友函式也具有函式覆載的功能，類別對每一個欲作為其朋友函式之覆載函式，均需個別宣告其為 friend (宣告朋友函式是基於函式原型宣告，覆載函式其原型宣告各不相同)。

程式範例: oop_ex73.cpp

注意要點:

1. 一函式也同時為兩個(或兩個以上)類別的朋友函式，於朋友函式內，對此兩個類別內之私有資料成員，均可以用「.」或「->」來存取。
2. 於兩個類別內，均需要將此函式宣告為 friend，個別賦予各自私有成員的存取權限。
3. 對於此情況，朋友函式的原型宣告必會出現再某一類別的定義之前，為了使編譯器能在編譯朋友函式時能認得此一尚未定義的類別，再此之前需先宣告此一尚未定一的類別名稱，之後再補其定義。
4. 朋友函式之定義需位於賦予權限之兩類別的定義之後。

程式範例: oop_ex74.cpp

注意要點:

1. 一類別的朋友函式，可以是另一類別的函式成員。
2. 需注意函式定義的位置。

8.4.1 Friend Classes

說明:

除了函式之外，我們亦可以將某類別宣告唯一類別的朋友類別，例如，B 類別被 A 類別宣告為朋友類別，如此一來，在 B 朋友類別內的所有函式成員，均可存取 A 類別的私有資料成員。

朋友類別的宣告方式，大致與朋友函式的宣告相同，例如，B 類別被 A 類

別宣告為朋友類別，是於 A 類別的定義內，做如下之宣告，此宣告同樣可置於定義區塊中的任何位置。

```
class A
{
    …….
    friend class B;
    …….
};
```

程式範例: oop_ex75.cpp

注意要點:

1. 朋友權限並不具對稱性，例如 A 類別宣告 B 類別為其朋友類別，對 B 類別而言，A 類別並非朋友類別。
2. 朋友權限亦不具傳遞性，例如 B 類別為 A 類別的朋友類別，C 類別為 B 類別的朋友類別，並不代表 C 類別為 A 類別的朋友。

8.5 Static Members in C++

8.5.1 Static Data Members

說明:

C++可讓我們宣告同類別之物件共同使用的一筆記憶體資料，此資料成員稱為靜態資料成員。就一般的資料成員，各同類物件配置有各自的記憶體，其內資料可不相同，但就靜態資料成員，各同類別物件共用同一記憶體與其內的資料，對每一個物件而言，靜態資料成員的值均相同。

靜態資料成員需用 static 關鍵字來宣告。由於靜態資料成員是屬於類別，而非個別物件所擁有，故對其存取需要用類別名稱。

靜態成員可以是 public、private、或 protected，亦可為 const，所有(public、private、或 protected)靜態資料成員需在檔案範圍內為其設定初值一次(且只有一次)，其格式如下，須注意設定時同時要註明資料型態，以及所屬類別。

格式:

```
class 類別名稱
{
    static  靜態資料成員；
}

類別名稱::靜態資料成員  = 3;     //靜態成員初始化
```

例:

```
class Person
{
public:
    static int count;
    ……
    string name;
    int age;
    ……
}

int Person::count = 0;          //初值設定
Person::count++;                //存取 public 靜態資料成員
```

程式範例: oop_ex76.cpp

注意要點:

1. 於類別內，靜態資料成員如一般資料成員一樣，可自由存取。
2. 如靜態資料成員為公開，於類別外部可使用類別名稱加上::運算子來對其存取與修改。(也可以透過某一物件來更改，但此改變是對所有同類的物件。)
3. 靜態成員可以是 public、private、或 protected，如靜態資料成員為 private，則其存取需透過函式(靜態函式)。
4. 即使類別沒有物件存在，類別的 static 成員仍能單獨存在(靜態成員於類別之定義完成後即產生)。

8.5.2 Static Function Members

說明:

如果函式成員內，其所使用資料都是靜態資料成員，則此函式成員可以定義為靜態函式成員。

靜態函式成員，於系統中只會產生一份屬於此函式的記憶體空間，其由屬於此類別的所有物件所共享使用。(一般的函式，每一個物件都會各自為其產生一份記憶體空間。)

靜態函式成員的宣告同樣是使用 static 關鍵字，其格式如下

格式:

```
class 類別名稱
{
    static  靜態函式成員(參數列)；          //靜態函式宣告
}

類別名稱::靜態函式成員(參數列);          //靜態函式呼叫
```

例:

```
class Person
{
    string name;
    int age;
    ……
    static int count;
public:
    static int getCount( )
    {
        return count;
    }
}

cout<<Person::getCount()<<endl;          //靜態函式呼叫
```

程式範例: oop_ex77.cpp

注意要點:

1. 靜態函式的呼叫，可以透過類別名稱加上「::」運算子，或透過任一個屬於此類別的物件來呼叫。
2. 靜態函式內所使用的資料成員，都必須是靜態資料成員，如必須使用到非靜態資料成員，則此函式只能是一般函式成員，而不能宣告為靜態函式。
3. 於靜態函式成員內使用 this 指標為錯誤的語法，因為靜態成員是獨立於任何物件之外，故沒有 this 指標。
4. 靜態函式成員不能宣告成 const。

8.5 Static Members in Java

Java 類別同樣支援靜態成員，同樣使用 static 關鍵字，其性質與 C++之靜態成員相同，程式格式上稍有不同，其列述如下。

與 Java 之資料成員相同，靜態成員可於類別之定義內直接初始化，與 C++中之靜態成員初始化格式不同。

透過類別名稱存取或呼叫公開之靜態成員，是使用「.」，如下，格式與 C++不同。

```
類別名稱 . 靜態資料成員 = …..;
類別名稱 . 靜態函式成員(…..);
```

程式範例: oop_ex78.java

8.6 Static Initialization Blocks in Java

Java 類別靜態資料成員之初始化(如例 oop_ex78.cpp 之方式)，有以下的限制。

1. 初始化只能是單純使用等號的指定命令(assignment statement)，不可使用 if-else 條件判斷式或迴圈。

2. 無法表示為 try-catch 區塊，故不能呼叫宣告能擲出非執行時例外之函式(method that is declared to throws a non-runtime (checked) exception)，且如呼叫宣告能擲出執行時(runtime)例外之函式，錯誤將無法被 catch 處理。

針對以上的限制，Java 提供了 Static Initialization Block，以供從事更有彈性的靜態資料成員初始化工作。其為一名為 static 之程式區塊，格式如下，{ }中可使用條件判斷式、迴圈或例外處理，來從事靜態資料成員之初始化。

```
class X
{
    ……..
    static
    {
        靜態資料成員初始化程式;
    }
    ……..
}
```

程式範例: oop_ex79.java

1. Static Initialization Block 之性質為 static，是針對類別而非物件，故只會當某類別被載入(loaded)時執行一次，且只有一次。
2. Static Initialization Block 中只能初始化 static data members。

程式範例: oop_ex80.java

1. 我們也可以為 Java 類別提供 Non-Static Initialization Block，其即是沒有加 static 關鍵字的 Initialization Block。
2. Non-Static Initialization Block 可初始化一般資料成員與設定靜態資料成員，唯靜態資料成員不可為 final，因為對靜態成員是執行設定，一般資料成員則可，因為對一般資料成員是執行初始化。
3. Non-Static Initialization Block 每產生一物件即執行一次，其次序在建構函式之前。

8.7 Finalizers in Java

我們之前介紹過 Java 之 Garbage Collection 機制，當一物件不再被參照到時，此物件會被標記為廢棄，必要時系統會將其釋放並取回其所佔記憶體再利用，基於此機制，通常我們無法控制與察覺一物件何時被釋放。

Java 允許我們為物件提供一 Finalizer，其會在物件被 Garbage Collection 機制釋放前呼叫執行，此可允需我們做一些清理的工作，例如我們可於其中終止非 Java 之資源、更新靜態成員、或提示使用者物件之釋放。

Finalizer 為一名稱為 finalize，且無任何參數與傳回值之函式(method)，格式如下，一類別只能有一個 finalizer(note: finalize 已定義於 Object class)。

```
class X
{
    ……..
    protected void finalize( )
    {
        //actions to do for cleanup;
    }
    ……..
}
```

如為衍生類別，必要時其 Finalizer 內需以 super.finalize( )確保其父類別之 Finalizer 能被執行。

程式範例: oop_ex81.java

1. 即使物件以不再被參照到，Garbage Collection 機制不一定會取回其配置記憶體，亦即 Finalizer 不保證一定會被執行，故最好不要依賴其來做資源釋放之工作。
2. 可使用 System.gc( )要求系統即刻做 Garbage Collection 之工作，但不保證所有標示為廢棄之物件均會被回收。

8.8 將 C++物件之宣告定義與函式成員程式分開為.h 與.cpp 檔

說明:

發展一個大程式，為方便程式的管理與合作研發，我們可將一程式分成為幾個不同的檔案。

常用的分解方式為，將相近或相關的類別與獨立函式定義在一同名稱之.h 與.cpp 檔內，其中.h 檔為各類別與獨立函式原型之宣告，而.cpp 檔中為各函式成員與獨立函式之定義程式碼。

當我們需要使用一物件時，只需將包含此物件之.h 檔 include 進來。

程式範例: oop_ex82.cpp

注意要點:

1. 使用 MS VisualC++，各.h 與.cpp 檔均需加入 project 中。
2. 通常各檔案同置於相同的目錄。
3. #include “filename.h” (使用雙引號)。
4. 常需使用 #ifndef ，#define ，#endif 來避免類別重複宣告。