14. Templates in C++

使用 C++之樣版(Templates)功能，我們可以製作出適用於多種型別之程式，例如我們可製作出某一函式，其可通用於處理整數、浮點數、字串等等型態之陣列資料的排序。

C++之樣版可用於製作出適用於多種型別之函式，稱為函式樣版(function templates)，或可適用於多種型別之類別，稱為類別樣版(class templates)。

對於 Java 而言，其並不需要如 C++之樣版功能，才能使程式適用於多種型別，由於 Java 所有類別均繼承自 Object 類別，如程式是針對 Object 型態所製作，則其可適用於所有 Java 類別(sub-type of Object)。

樣版的使用方式，其是於函式樣版或類別樣版之前，以 template 關鍵字宣告樣版之使用，接著於其後之< >中，以 class 或 typename 宣告一列代表未定型別之形式化型別參數(type parameters)，之後便可以形式化型別參數代表未知之型別，製作函式樣版或類別樣版，如下所示，其中型別參數之名稱可自訂，不一定為 T。

```
template<class T>
void functionX(… )
{
    //…..
    T data;
    //…..
}
```

```
template<typename T>
class Y
{
    //…..
    T data;
    //…..
};
```

```
template<class T1, class T2>
class Z
{
    //……
    T1 data1;
    T2 data2;
    //…….
};
```

14.1 Function Templates

函式覆載(overloading)的功能一般是用於對不同型別之資料，做類似的操作。但如果對每一種型別資料的操作方式都相同(函式內容相同)，則使用函式樣版，將可製作出更嚴謹、更方便、更簡潔的程式。

程式範例: oop_ex126.cpp

注意要點:

1. 對< ＞內形式化型別參數之宣告，例如 T 之宣告，可使用 class 或 typename。
2. 無論其為內建或自訂，只要資料之型別可支援函式樣版內的操作方式，即可配合使用函式樣版，例如 int, double char，反之則不可，例如自訂之 test 型態資料，因其不支援<<運算，故無法與 printarray 函式樣版配合，必須提供<<運算子支覆載才行。
3. 函式樣版並不會與一般覆載函式因模擬兩可的情況而產生衝突，例如本例中 iarr 陣列之列印。
4. 當函式呼叫時，編譯器會執行配對，判斷應呼叫哪一個函式，首先，編譯器會去尋找參數型態與呼叫引數型態完全相同者，如沒有，編譯器才會去尋找是否有合適的函式樣版存在。
5. 樣版運作的方式，是當函式呼叫時，編譯器會依 T 之型別(呼叫引數之型別)，來產生樣版函式，例如呼叫 printArray(darr, dcount)時，編譯器及產生參數列型態為(const double*, const int)，內容{ }與函式樣版定義之內容完全相同的樣版函式。

14.2 Class Templates

有些時候，我們希望能定義出所謂的泛型類別(genetic class)，其可與多種不同類型之資料配合使用，C++之類別樣版(class templates)即具有此功能。

例如 list、stack 等資料結構(Data Structures)或稱為容器類別(Container Classes)，我們希望其最好能存放整理各種類別的資料，而不是只針對某一類別資料的定型類別，這時使用樣版來製作，即可達成。

程式範例: oop_ex127.cpp

注意要點:

1. Node 定義於 List 之 body 內，為只供 List 內部使用之資料型態。
2. 此 Linked list 只能存放 int 型態之資料。
3. 欲使其能廣泛存放各種型別之資料，需將其定義為類別樣版。

程式範例: oop_ex128.cpp

注意要點:

1. 類別樣版之表示法與一般類別不同，除了類別樣版的名稱之外，還需於其後加上< >，< >中是 template<class …>所指定的各形式化型別參數。例如：List<T>，而非 List。
2. 以類別樣版之定義產生物件時，需指定其形式化型別參數所代表之型別，例如：List<int>* numlist = new List<int>(i); 則編譯器會以 int 取代 T，定義新型別並依其產生物件。
3. 類別樣版之函式成員如參數為 List<T>，則表示其偕接受 List<int>、List<double>、List<float>、List<String>，甚至 List<List>等物件為輸入參數，不過先決條件是< >中形式化型別參數之型別需能支援函式成員之定義所使用的運算。
4. 當類別樣版之函式成員之定義位於類別樣版之 body{ }外時，則如同函式樣版，需於其定義前加上 template<class …..>。
5. 對於類別樣版之定義，C++編譯器允許只使用類別樣版之名稱，而不需加上其後之<…>，但對於指定函式成員之歸屬，範疇解析運算子(::)前之類別則不可使用簡寫。

14.3 Template Specialization

經常，我們無法製作出能完全適用於所有型別的類別樣版，對於此情況，我們需能對於此類別樣版提供替代之定義，使編譯器對於無法適用類別樣版定義之型別，可選擇所提供之替代定義。

例如前例之 List<T>類別樣版將無法適用於沒有提供'=='與'!='運算子覆載的型別，例如 C style 之字串(char*)，其所支援之比較方式是使用 string.h 中之 strcmp 函式，而非'=='與'!='運算子，產生 List<char*>之物件將不可能。故對於此類之情況，我們須對於 char*型態之資料特別提供替代定義。

程式範例: oop_ex129.cpp

注意要點:

1. 對於特殊的替代定義，template<class T>之前置宣告將不需要，如仍欲加上，則使用 template< >。
2. 雖無 template<class T>之前置宣告，編譯器仍能知道其為先前定義之 List<T>類別樣版針對 char*型別之替代定義，故當我們要產生 List<char*>之物件

時，其將自動被選擇引用。

14.4 Class Templates and Non-Type Parameter

樣版類別除了可定義使用型別參數(type parameters)，亦可定義使用非型別參數(non-type parameters)，非型別參數可被視為樣版類別內之一 const 變數，其值於物件建立時所指定。

例：

```
template<class T, int i>
class Buffer
{
    T v[i];          //i is const int, so OK
    int size;
public:
    Buffer( ) : size(i){ }
    ……..
};

Buffer<char, 127> cbuf;        //assign T is char, i is 127
```

程式範例: oop_ex130.cpp

注意要點:

1. 類別樣版之表示法，除了類別樣版的名稱之外，還需於其後加上< >，< >中是 template<class …>所指定的各型別參數與非型別參數。例如：List<T, id>。
2. 物件建立時需指定各型別參數所代表之型別，以及非型別參數之值。
3. 非型別參數被視為 const。

14.5 Default Values for Template Parameters

我們可為類別樣版之各型別參數與非型別參數設定預設值，當產生物件時如沒有指定樣版參數之值時，則使用其預設值。此有如函式參數之設定預設值。

例：

```
template<class T = string, int i = 127>
class Buffer
{
    T v[i];
    int size;
public:
    Buffer( ) : size(i){ }
    ……..
};

Buffer<char> cbuf;        //assign T is char, i is 127 by default
Buffer< > sbuf;           //T is string by default, i is 127 by default
```

程式範例: oop_ex131.cpp

注意要點:

1. 雖無指定參數值，< >仍需要。