大局觀：泛型程式設計（Generic Programming）與 STL

侯捷 jjhou@ccca.nctu.edu.tw
http://www.jjhou.com

十年前趕上 OO（Object Oriented，物件導向）第一波工業浪潮的朋友們，想必今日有一番顧盼豪情，以及一份「好加在」的驚悚：「好加在」搭上了 OO 的早班列車，不然今天就玩不下去了！

面對 Generic Programming（泛型程式設計），我有相彷的感覺。

我將撰寫爲期五次的系列文章，爲各位介紹這個不算新卻又有點新的觀念與技術。

●C++ Template, Generic programming, STL

1968 Doug McIlroy 發表其著名論文 "Mass Produced Software Components"，揭櫫了以「可重用軟體元件（又稱軟體積木或軟體 IC）」構築標準程式庫的願景。如今三分之一個世紀已逝，components software -- 以組件（元件）爲本的軟體 -- 大行其道。然而在許多領域中，標準化仍未建立。甚至連任何軟體一定會用到的基本資料結構和基本演算法，對此亦付闕如。於是大量的程式員，被迫進行大量重複的工作，爲的竟只是重複完成前人早已完成而自己手上並未擁有的碼。這不但是人力資源的浪費，也是挫折與錯誤的來源。

撇開追求技術的熱誠不談（也許有人確實喜歡什麼都自己來），商用程式庫可以拯救這些程式員於水深火熱之中。只要你的老闆願意花一點點錢，就可以在人力資源與軟體開發效率上取得一個絕佳平衡。但是金錢並非萬能，商用程式庫彼此之間並無介面上的標準！換句話說你無法在發現了另一套更好、更豐富、效率更高的程式庫後，改弦更張地棄此就彼 -- 那可能會使你付出慘烈的代價。這或許是封閉型技術與觀念帶給軟體元件公司的一種小小的、短暫的利益保護。

除了「介面無標準」這一問題，另一個問題是，目前絕大部份軟體元件都使用物件導向技術，大量運用繼承與虛擬函式，導至執行成本增加。此外，一旦運用物件導向技術，我們此刻所說的基礎資料結構及各種基礎演算法，便皆需以 container（容器，放置資料的某種特殊結構）爲本。資料，放在 container classes 內（形成其 data members）；操作行爲，亦定義在 container classes 內（形成其 member functions）；這使得耦合（coupling，兩物過度相依而不獨立）的情況依然存在，妨礙了元件之所以爲元件的獨立性、彈性、交互操作性（相互合作性，interoperability）。

換句話說，要解決這些困境，第一需要標準介面的建立，第二需要 OO 以外的新技術，俾能夠比 OO 更高效率地完成工作。

◎新技術

C++ template 是新技術的曙光。

一言以蔽之，所謂 template 機制就是，將標的物的資料型別參數化。一旦程式以指定引數的方式，確定了這個（或這些）型別，編譯器便自動針對這個（或這些）型別產生出一份實體。這裡所說的實體，可以是一個 function body，也可以是一個 class body。而「由編譯器產生出一份實體」的動作，我們稱之爲具現化（instantiation）。

針對標的物之不同，C++ 支援 function templates 和 class templates 兩大類型。後者的 members 又可以是 templates（所謂 member templates），形成深度巢狀（nested），帶來極大的彈性與組合空間。關於這一點，本文稍後示範 STL 的使用時，會展現在你面前。

下面是一個簡單的 function template：

```
#include <iostream>
using namespace std;

template <class Type>
   Type mymin(Type a, Type b) {
      return a < b ? a : b;
}
```

由於 mymin() 內對於函式引數 a 和 b 動用到 less-than operator，所以任何型別如果希望能夠滿足這個函式的 Type，必須支援 operator<。下面是個例子：

```
class rect
{
friend ostream& operator<<(ostream& os, const rect& rhs);
public:
  rect(int w, int h) : _w(w), _h(h)
      { _area = _w * _h; }
  bool operator<(const rect& rhs) const
      { return _area < rhs._area; }
private:
  int _w, _h, _area;
};

ostream& operator<<(ostream& os, const rect& rhs)
{
   os << '(' << rhs._w << ',' << rhs._h << ')' << endl;
   return os;
}
```

```
int main()
{
  // 此行具現出函式實體 int mymin(int, int);
  cout << mymin(10, 20) << endl;   // 10

  // 此行具現出函式實體 double mymin(double, double);
  cout << mymin(30.0, 20.0) << endl;   // 20

  rect r1(3,5), r2(5,7);
  // 此行具現出函式實體 rect mymin(rect, rect);
  cout << mymin(r1, r2) << endl;   // (3,5)
}
```

下面是一個簡單的 class template：

```
template <typename Type>
class Queue {
public:
   Queue()  { /* ... */  };
   ~Queue() { /* ... */  };

   Type& remove();
   void add( const Type & );
   bool is_empty();
   bool is_full();
private:
   // ....
};
```

一旦程式開始使用 class template Queue，例如：

```
Queue<int> qi;
Queue< complex<double> > qc;
Queue<string> qs;
```

編譯器就會分別具現出針對 int, complex<double> 和 string 的 Queue class 實體出來。

（關於 template 的語法與技術，請參考任何一本「年輕的」C++ 書籍。我推薦你看 C++ Primer 3/e, by Lippman & Lajoie，第 10 章和第 16 章）

由於具現化行爲是在編譯時期完成，所以愈是複雜的 templates，愈會耗損編譯時間。然而這卻無損程式的執行時間。因此，對於元件的執行效率，帶來一種保障。換言之，使用 template 並不必然會降低元件的執行效率（但對元件的開發效率則有顯著的提昇）。事實上，Alexander Stepanov（STL 的創造者）非常重視 STL 的執行效率，並視之爲一種「基本國策」。他說過，『以抽象資料型別 stack 爲例，並不是擁有 push 和 pop 兩個操作行爲，就是一個好的 stack。最重要的是，pushing 動作應

該耗用固定時間，不因 stack 的大小而改變。如果我的 stack 的 pushing 動作愈來愈慢，沒有人會用我這個 stack。』

軟體的最高理想，就是能夠設計出一些可重複使用（reusable）的碼。所謂可重複使用，不僅意味可以像樂高（Lego）積木那樣廣泛地加以配接組合（沒有如此的彈性又怎稱得上重複使用？），而且在配接組合的過程中不會耗損其效率表現。

效率議題，大概是你在明瞭 STL 的強大威力與彈性之後，讓你躊躇不前的唯一可能因素。然而我可以告訴你，使用 STL，並不會造成效率下降。STL 的效率表現，是經過嚴格規範與實現後的一個保證，而不是一種臆測。

C++ template 使泛型技術有了理想的實現環境。「型別參數化」，噢，那不正是「泛型」嗎 :)

◎新標準

泛型（Generic，Genericity）是一種觀念，就好像物件導向（Object Oriented）是一種觀念。STL（Standard Template Library）則是一個以泛型技術完成的實作品，就好像 MFC（Microsoft Foundation Classes）是一個以物件導向技術完成的實作品。

前面曾經說過，在許多應用領域之中，標準化尚未建立。甚至連需求最殷的基本資料結構和基本演算法，亦無所謂的標準介面。

STL 成功地扮演了這方面（基本資料結構和基本演算法）的標準介面角色。

乍見之下，STL 的價值在於其所帶來的一套極具實用價值的組件（components，稍後會詳加展示）。這種價值就像 MFC 或 VCL 之於 Windows 軟體開發過程所帶來的價值一樣，直接而明瞭。然而，STL 更有不同。STL 最重要的價值，其實是在泛型思維模式（Generic Paradigm）下建立起一個系統化的、條理分明的「軟體組件分類學」。這個分類學告訴我們什麼是 requirements，什麼是 concepts，什麼是 models，什麼是 refinements。目前沒有任何電腦語言對於這些概念有實質的對應（Alexander Stepanov，STL 的創造者，曾經說他的下一步就是要發展一個這樣的語言）。

換句話說，STL 所實現的，是依據泛型思維模式而建構的一個概念結構。這個以 concepts 爲主體而非以 classes 爲主體的概念結構，嚴謹地形成了一個介面標準。在此介面之下，任何組件有最大的獨立性；組件之間以 iterators 膠合起來，或是以 adaptors 互相配接，或是以 function objects 傳遞一整組運算需求。

當然啦，不是所有的組件都可以互相配接。它們必須是可配接的（adaptable），否則會損及效率，甚至根本配接不起來。

上述這些看似極度抽象的話語，也許使初初接觸 STL 的你如墜五里霧。當你逐漸從 STL 的實用面，到 STL 的實作面，到 STL 的擴展面，全面浸淫於 STL 之後，請你再回頭來細想我的這些描述。

●STL 的歷史（註 1）

STL 係由 Alexander Stepanov 創造於 1979 年前後，這也正是 Bjarne Stroustrup 創造 C++ 的年代。雖然 David R. Musser 於 1971 開始即在計算機幾何領域中發展並倡導某些泛型程式設計觀念，但早期並沒有任何程式語言支援泛型程式設計。第一個支援泛型概念的語言是 Ada，Alex 和 Musser 曾於 1987 開發出一套相關的 Ada library。然而 Ada 在美國國防工業以外並未被廣泛接受，C++ 卻如星火燎原般地在程式設計領域中攻城略地。當時的 C++ 尚未導入 template 性質，但是 Alex 已經瞭解到，C/C++ 允許程式員經由指標，以非常彈性的方式處理記憶體，而這正是又要一般化（泛型）又要不失效率的一個重要關鍵。

更重要的是，必須研究並實驗出一個「立基於泛型程式設計之上的 component library 的完整架構」。Alex 在 AT&T 實驗室及 Hewlett-Packard Palo Alto 實驗室，分別實驗許多種架構和演算法公式，先以 C 而後以 C++ 完成。1992 年 Meng Lee 加入 Alex 的專案，成爲另一位主要貢獻者。

Bell 實驗室的 Andrew Koenig 於 1993 年知道這個研究計劃後，邀請 Alex 於是年 11 月的 ANSI/ISO C++ 標準委員會會議上展示其觀念。獲得熱烈的迴應。Alex 於是再接再勵於次年夏天在 Waterloo 舉行的會議前完成其正式的提案，並以壓倒性多數，一舉讓這個巨大的計劃成爲 C++ Standard 的一部份。

註 1：如欲更清楚知道 STL 的歷史，請參考 Dr Dobb's Journal 於 1995 年三月刊出的 "Alexander Stepanov and STL" 一文。

●實作品、編譯器、網路資源、書籍

由於 STL 已被納入 C++ Standard，成爲 C++ Standard Library 的重要一員，因此目前各 C++ 編譯器都支援有一份 STL。

STL 在哪裡？就在相應的各個 C++ 含入檔中，如 <vector>, <list>,<functional>, <algorithm>, <iterator>…。這些未帶 .h 副檔名的 C++ 含入檔，是 C++ Standard 所規定的標準命名法。某些編譯器並沒有完全遵循這樣的命名規則，仍沿用舊有的 .h 命名法，例如 Inprise C++Builder。但是你仍然可以在程式中使用上述無 .h 副檔名的含入檔，原因是這類編譯器的前置處理器對此做了一些手腳。

大部份 C++ 編譯器都提供 (1) 有 .h 副檔名，(2) 沒有任何副檔名的兩套含入檔。例如 Visual C++，以無 .h 副檔名者含入有 .h 副檔名者，來涵蓋早先的寫碼習慣。又例如 GNU C++，則是以幾乎相同的兩份含入檔（一個無 .h 副檔名，一個有 .h 副檔名），再於其中含入名爲 stl_xxx.h 的實際主角。

以下是我手上三套 C++ 編譯器內含的 STL 實作品的供應者：

```
Microsoft VC6：P.J. Plauger
Inprise C++Builder4：Rogue Wave Software, Inc.
GNU C++ egcs-2.91.57：Silicon Graphics Computer Systems, Inc
```

所有 STL 實作品的老祖宗，都是源於 Hewlett-Packard Company 並由 Alexander Stepanov 和 Meng Lee 完成的原本。其中任何一個檔案上頭，都有一份聲明，允許你任意使用、拷貝、修改、傳播、販賣這些碼，無需付費，但必須將該份聲明置於你的檔案內。

你還可以從網路下載其他的 STL 實作品。下面是幾個資源豐富的網站，可再從中往外連結：

```
http://www.sgi.com/Technology/STL/
http://www.stlport.org/
```

至於 STL 相關書籍，請見本期的【無責任書評】專欄。我一共介紹了四本書。

●學習 STL 的三個境界

王國維說大事業大學問者的人生有三個境界。我認爲學習泛型程式設計以及 STL 也有三個境界：

第一個境界是使用 STL。

第二個境界是瞭解泛型程式設計的內涵與 STL 的實作原理。

第三個境界是擴充 STL。

應該還要有個第 0 境界，或說學習泛型程式設計與 STL 的門檻，那就是 C++ template 機制。

◎境界一：使用 STL

對程式員而言，諸多抽象描述，不如實象的 code 可以直指人心。稍後我直接列幾段程式碼，你就大略知道 STL 的威力了。

STL 有六大組件（components）：

1. containers：泛型容器，各種基本資料結構如 vector, list,deque, set, map…
2. algorithms：泛型演算法，各種基本演算法如 sort, search,copy, erase…。STL 提供了 70 個。
3. iterator：應用於 containers 與 algorithms 身上的所謂「泛型指標」，扮演兩者間的膠著劑。共有五種型態，還有各種衍生變化。iterator 是 STL 中最重要最抽象的一個組件，它使 containers 和 algorithms 可以各自獨立發展，互不干連。
4. function object：行爲上類似「一整個函式」的一種組件。實作技術上則是一個改寫了 "call operator" 的 classes。C/C++ 的函式指標，可以被用來做爲一個 function object。STL 提供有 15 個現成的 function objects。

5. adaptor：可改變（修飾）containers 或 function object 介面的一種組件。例如 STL 提供的 queue 和 stack，雖然看似泛型容器，但它其實只是一種 container adaptor。用來改變 function object 介面者，稱爲 function adaptor（或稱 functor）。用來改變 container 介面者，稱爲 container adaptor。用來改變 iterator 介面者，稱爲 iterator adaptor。
6. allocator：記憶體配置系統。STL 提供有現成的 allocator。

◎境界二：瞭解 generic programming 的內涵與 STL 的實作原理

知道怎麼運用 STL 之後，有了點成就感，對實際工作也貢獻了一些，可以開始比較不那麼現實（但其實仍能回饋到現實面）的深鑽功夫了。

最好能對一兩個（不必太多）STL 組件做一番深刻追蹤。STL 原始碼都在手上（就是相應的那些含入檔嘛），好好認識並體會一下 STL 所謂的 Concepts 和 Modeling 和 Refinement，好好瞭解一下爲什麼需要所謂的 traits 技術（這一技術非常重要，大量運用於 STL 之中，我將在第二篇文章詳細解釋其內涵）。好好做幾個個案研究，便能夠對泛型程式設計以及 STL 有理論上的通盤掌握。

◎境界三：擴充 STL

最高境界，當然是在 STL 不能滿足你的時候，自己動手寫一個可融入 STL 體系中的軟體組件了。這當然得先徹底瞭解 STL，也就是得通過上述第二境界的痛苦折磨。

●STL 運用實例

對世人而言，1815 年的滑鐵盧（waterloo in Belgium）標示了失敗的印記。但對 C++ 社群而言，1994 年的滑鐵盧（waterloo in Ontario, USA）標示的則是極大的成功。這一年於該處舉行之 C++ 標準委員會，正式將 STL 納入。

STL 是泛型程式設計的一個研究成果。它的內涵或許有相當的難度，但是它的運用卻是極其簡單而又令人愉悅的。下面就是幾個簡單的實例。我在程式中先製作出一個陣列，再以此爲初值，放進各種 STL containers 之中（各個 STL containers 的特性，將在本系列第三篇文章中說明）。然後我示範如何使用 STL algorithms for_each()。注意，STL containers 可以放置各種型別，包括使用者自定的 class 型別；爲求簡易，我一概以 int 型別爲例。

圖一是使用各種 STL 組件時，程式所需要的含入檔。使用 STL 時必須注意，由於 C++ Standard Library 的所有組件均封閉於一個名爲 std 的 namespace，所以使用前必須先使其曝光。最簡單的作法就是撰寫 using directive 如下：

```
using namespace std;
```

圖一\使用各種 STL 組件時，程式所需要的含入檔

STL 組件	程式所需含入檔
algorithms	<algorithm>
四個數值相關演算法	<numeric>（註 2）
vector	<vector>
list	<list>
deque	<deque>
stack	<stack>
queue	<queue>
priority queue	<queue>
map	<map>
set	<set>
multimap	<map>
multiset	<set>
function objects	<functional>
iterator adaptor	<iterator>

註 2：四個數值相關演算法是：accumulate(), adjacent_difference(), partial_sum(), inner_product().

以下動作完成一個陣列：

```
int ia[] = { 1, 3, 2, 4 };
```

以下各動作將此陣列分別當做各個 containers 的初值。每一個 containers 通常都擁有數個版本的建構式，提供很大的彈性讓我們設定初值。一般而言不外乎指定另一個 container 做爲初值，或是指定某個 container 的某個範圍做爲初值。運用「範圍」這個觀念時，就要用到 iterator（泛型指標）；此處由於所處理的是簡單的 int 型別，所以 C++ 原生指標可以拿來當做 iterator 使用。

請注意，各家編譯器對 C++ Standard 的支援程度不一。以下程式可在 Inprise C++Builder 4.0 順利編譯完成。

```
list<int> ilist(ia, ia+4);
vector<int> ivector(ia, ia+4);
deque<int> ideque(ia, ia+4);
stack<int> istack(ideque);
queue<int> iqueue(ideque);
priority_queue<int> ipqueue(ia, ia+4);
set<int> iset(ia, ia+4);
```

注意，map 容器放置的是一對一對的（鍵值/實值），所以我以這樣的方式安排其初值：

```
map<string, int> simap;  // 以 string 爲鍵值（索引），以 int 爲實值
simap[string("1")] = ia[0]; // 第一對內容是 ("1", 1)
simap[string("2")] = ia[1]; // 第二對內容是 ("2", 3)
simap[string("3")] = ia[2]; // 第三對內容是 ("3", 2)
```

```
simap[string("4")] = ia[3]; // 第四對內容是 ("4", 4)
```

以下動作分別將各個 containers 的內容印出。其中用到 for_each()，是一個 STL algorithms，要求使用者指定一個範圍，以及一個「動作」。範圍可以（應以）iterators 表示。每一種 containers 幾乎都提供有 begin() 和 end() 兩個 member functions，分別傳回指向頭部和指向尾部的兩個 iterators，標示出整個 container。未提供 begin() 和 end() 者，我改以 while 迴圈來進行巡訪，而不使用 for_each()。

至於巡訪獲得一個元素之後所要執行的「動作」，可以函式指標或 STL function object 表示。我希望對每一個元素施行的「動作」是將元素內容丟到標準輸出裝置 cout，這樣的「動作」並沒有任何 STL function objects 可以提供，所以我自己設計一個函式來完成。

```
for_each(ilist.begin(), ilist.end(), pfi);      // 1 3 2 4
for_each(ivector.begin(), ivector.end(), pfi);  // 1 3 2 4
for_each(ideque.begin(), ideque.end(), pfi);    // 1 3 2 4
for_each(iset.begin(), iset.end(), pfi);        // 1 2 3 4（排序）

while(!istack.empty()) {
  cout << istack.top() << " ";     // 4 2 3 1（先進後出）
  istack.pop();
}
while(!iqueue.empty()) {
  cout << iqueue.front() << " ";   // 1 3 2 4（先進先出）
  iqueue.pop();
}
while(!ipqueue.empty()) {
  cout << ipqueue.top() << " ";    // 4 3 2 1（按優先權次序取出）
  ipqueue.pop();
}
```

請注意，stack（先進後出），queue（先進先出），priority queue（依優先權次序決定「下一個」是誰），以及 set（內部有排序行爲）的輸出結果，在在都顯示出其特性。

map 的處理稍稍不同。由於 map 容器所放的資料都是一對一對的「鍵值/實值（key/value）」，所以當我們巡訪到一對資料時，必須以 first 和 second 來取出其第一份內容（鍵值）和第二份內容（實值）：

```
map<string, int>::iterator iter;  // iterator 的型別必須先指定清楚
for (iter=simap.begin(); iter!=simap.end(); ++iter)
  cout << iter->first << " " << iter->second << "  "; // 1 1  2 3  3 2  4 4
}
```

每一種 containers，都定義有適合它自己所用的 iterators。下面是 vector<int> 的 iterator 定義方式：

```
vector<int>::iterator iter;
```

```
vector<int>::const_iterator citer;
```

下面是 list<string> 的 iterator 定義方式：

```
list<string>::iterator iter;
list<string>::const_iterator citer;
```

以下使用 STL algorithm accumulate()，對 ilist 做累積動作，並指定所謂「累積動作」是指乘法。我採用 STL function objectmultiplies<int> 完成乘法動作。

```
// 以下行爲，造成 1 * 3 * 2 * 4.
cout << accumulate(ilist.begin(), ilist.end(),
                   1, multiplies<int>()) << endl;   // 24
```

以下使用 STL algorithm inner_product()，對 ilist 和 iset 進行內積行爲。

```
// 以下行爲，造成 0 + 1*1 + 3*2 + 2*3 + 4*4.
cout << inner_product(ilist.begin(), ilist.end(), // 第一範圍的頭尾
                      iset.begin(),               // 第二範圍的起始點
                      0 )                         // 運算初值
     << endl;                                     // 29
```

●STL 彈性展示

以下試舉數例。可略窺 STL 的彈性。

宣告一個 list，每個元素又是個 list，後者的每個元素是個字串：

```
list < list <string> > mylist;
```

宣告一個 list，每個元素是個 vector，後者的每個元素是個 pair。pair 有兩個元素，第一元素是個 string，第二元素是個 int：

```
list < vector < pair < string, int > > > mylist;
```

下面是 generic algorithms for_each()、function object modulus<int>、function adaptor bind2nd()、container list<int> 的運用實例：

```
// BCB4 : bcc32 test.cpp
// GCC : g++ -o test.exe test.cpp
#include <functional>
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;

template <typename T>            // function template
void print_elements(T elem)
```

```
{ cout << elem << " "; }

void (*pfi)(int) = print_elements;  // 函式指標

void main()
{
  int ia[7] = {0,1,2,3,4,5,6};
  list<int> ilist(ia, ia+7);     // 以陣列做爲 list 的初值
  for_each(ilist.begin(), ilist.end(), pfi);  // 0 1 2 3 4 5 6

  ilist.push_back(7);
  ilist.push_back(0);
  ilist.push_back(7);
  ilist.push_back(9);
  for_each(ilist.begin(), ilist.end(), pfi);  // 0 1 2 3 4 5 6 7 0 7 9

  ilist.remove_if(bind2nd(modulus<int>(), 2)); // 去除所有奇數
  for_each(ilist.begin(), ilist.end(), pfi);   // 0 2 4 6 0
}
```

下面程式從檔案中讀取所有單字，再全部列出於螢幕上。

```
// CB4 : bcc32 test.cpp
// VC6 : cl -GX test.cpp
// GCC : g++ -o test.exe test.cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <fstream>
#include <iterator>
using namespace std;

int main()
{
   string file_name;
   cout << "please enter a file to open: ";
   cin >> file_name;

   if ( file_name.empty() || !cin ) {  // 測試檔名
        cerr << "unable to read file name\n";
        return -1;
   }

   ifstream infile( file_name.c_str());
   if ( ! infile ) {  // 測試開檔成功否
        cerr << "unable to open " << file_name << endl;
        return -2;
   }
```

```
    // 宣告三個 iterators，
    // 一個針對檔案，輸入用，輸入單位是 string；
    // 一個針對檔案，代表 end-of-stream；
    // 一個針對 cout，輸出用，輸出單位是 string，後面緊跟一個 " "。
    istream_iterator< string > ins( infile ), eos;
    ostream_iterator< string > outs( cout, " " );

    copy( ins, eos, outs );  // 把檔案內容 copy 到螢幕上
}
```

以上這些例子，有些東西及其對應動作並不能讓你有直接的想像，尤其是 stream iterator。但是只要花一點時間，瞭解它們的規格與用途，使用起來非常簡單方便。

●軟體組件分類學

前面說過，STL 其實是在泛型思維模式之下建立起一個系統化的、條理分明的「軟體組件分類學」。這個分類學嚴謹定義了什麼是 concept, 什麼是 model 什麼是 refinement, 什麼是 range，也定義了什麼是 predicate, 什麼是 iterator, 什麼是 adaptor…。

我將在此描述其中一二，企圖讓你在最短篇幅中對 STL 本質建構出一個基本的認識。你可以參考 [Austern99]，獲得更豐富更詳細的說明。

◎concept, model

所謂 concept，描述某個抽象型別的一組條件（或說需求，requirements）。concept 並不是一個 class，也不是一個變數或是一個 template 參數；事實上 C++ 程式中沒有任何東西可以直接代表一個 concept。然而，在每一個用到泛型程式設計方法的 C++ 程式中，concept 非常重要。由 concepts 所構成的階層體系，正是 STL 的主體概念結構。

當某個型別滿足所有條件，我們便說此型別是該 conecpt 的一個 model。concept 可被視爲是一組型別條件。如果 type T 是 concept C 的一個 model，那麼 T 就一定滿足 C 的所有條件（需求）。因此，concept 亦可被視爲是一組 types。例如 concept Input Iterator 可以由 char*, int*, float* 及其他符合條件的 wrap classes…共同組成。如果 type T 是 concept C 的一個 model，那麼我們可以說 T 隸屬於「C 所表現的一組型別」。

concepts 的發現與描述，並不是藉由寫下某些需求條件而達成，而是由於我們定義了明確的 algorithms 並學習如何在它們身上使用 formal template arguments，於是逐步完成。換句話說這是一個反覆修潤的過程，不是紙上單向作業。

STL 所規範的基本 concepts 包括：

1. Assignable：type X 如果是 concept Assignable 的一個 model，那麼我們可以將 type X 的 object 內容拷貝並指派給 type X 的另一個 object。如果 x, y 是 Assignable，那麼保證以下動作中的 x, y 有著相同的值：

```
X x(y)
x = y
tmp = y, x = tmp
```

換言之，如果 type X 是 Assignable 的一個 model，它將會有一個 copy constructor。

2. Default Constructible：如果 type T 是 Default Constructible 的一個 model，它將有一個 default constructor。也就是說我們可以這樣寫，產生出一個 type T object：

```
T()
```

欲產生出一個 type T 的變數，可以這樣寫：

```
T t;
```

所有的 C++ 內建型別如 int 和 void，都隸屬於 Default Constructible。

3. Equality Comparable：如果 type T 是 Equality Comparable 的一個 model，那麼我們可以這樣比較兩個 type T objects 是否相等：

```
x==y
```

或

```
x!=y
```

4. LessThan Comparable：如果 type T 是 LessThan Comparable 的一個 model，我們可以這樣測試某個 T object 是否小於另一個 T object：

```
x < y
```

或

```
x > y
```

所謂 regular type，是指同時身爲 Assignable, Default Constructible,Equality Comparable 等 concepts 的 model。針對一個 regular type，你可以這樣想：如果你將 x 指派（assign）給 y，那麼 x==y 一定爲真。

大部份 basic C++ types 都是 regular types（int 是本節談及的所有 concepts 的一個 model），而幾乎所有定義於 STL 中的 types 也都是 regular types。

◎refinements

如果 concept C2 供應 concept C1 的所有機能，並且（可能）加上其他機能，我們便說 C2 是 C1 的一個 refinement（精鍊品）。

Modeling 和 refinement 必須滿足以下三個重要特性。只要把 concepts 想像爲一組 types，以下三者就很容易驗證：

1. Reflexivity 反身性。每一個 concept C 是其本身的一個 refinement。
2. Containment 涵蓋性。如果 type X 是 concept C2 的一個 model，而 C2 是 concept C1 的一個 refinement，那麼 X 必然是 C1 的一個 model。
3. Transitivity 遞移性。如果 C3 是 C2 的一個 refinement，而 C2 是 C1 的一個 refinement，那麼 C3 是 C1 的一個 refinement。

一個 type 可能是多個 concepts 的 model，而一個 concept 可能是多個 concept 的 refinement。

◎range（範圍）

對於 range [first, last)，我們說，只有當 [first, last) 之中的所有指標都是可提取的（dereferenceable），而且我們可以從 first 到達 last（也就是說對 first 累加有限次數之後，最終會到達 last），我們才說 [first, last) 是有效的。所以，[A, A+N) 是一個有效的 range，empty range [A, A) 也是有效的。[A+N, A) 就不是一個有效的 range。

一般而言，ranges 滿足以下性質：

1. 對任何指標 p 而言，[p, p) 是一個有效的 range，代表一個空範圍。
2. 如果 [first, last) 是一個有效而且非空的 range，那麼 [first+1, last) 也是一個有效的 range。
3. 如果 [first, last) 是一個有效的 range，而且 mid 是一個可由 first 前進到達的指標，而且 last 又可以由 mid 前進到達，那麼 [first, mid) 和 [mid, last) 都是有效的 ranges.
4. 反向來講，如果 [first, mid) 和 [mid, last) 都是有效的 ranges，那麼 [first, last) 便是有效的 ranges。

下一期，我要帶各位看看 iterator 的實作以及所謂的 traits 技術。Traits 技術幾乎於 STL 的每一個地方出沒，至爲重要。瞭解它的發展原由與最後形象，是掌握 STL 原始碼的重要關鍵。至於 iterators，是使資料結構（STL containers）與演算法（STL algorithms）相互獨立發展又可交互搭接的關鍵，其重要性與關鍵性居 STL 六大組件之首。

參考資料：

1. "Alexander Stepanov and STL", Dr. Dobb's Journal, Mar. 1995.
2. [Austern99] "Generic Programming and the STL" by Matthew H. Austern, AW, 1999
3. [Lippman98] "C++ Primer" by Lippman & Lajoie, AW, 1998
4. [Musser96] "STL Tutorial and Reference Guide" by David R. Musser, AW, 1996

作者簡介：侯捷，資訊技術自由作家，專長 Windows 作業系統、SDK/MFC 程式設計、C/C++ 語言、物件導向程式設計、泛型程式設計。目前在元智大學開授泛型程式設計課程，並進行新書《泛型程式設計》之寫作。