```
泛型容器（Containers）的應用與實作

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

     1. 大局觀：泛型程式設計與 STL                              2000.02
     2. 泛型指標（Iterators）與 Traits 技術                     2000.03
=>   3. 泛型容器（Containers）的應用與實作                      2000.04
     4. 泛型演算法（Generic Algorithms）與 Function Objects     2000.05
     5. 各色各樣的 Adaptors                                     2000.06
```

前兩篇文章我提到了 STL 之中所謂的 concepts。本期的主題 containers（容器），在 STL 的「概念結構（concepts structure）」中也是一種 concept，有所謂的 Forward Container, Reversible Container, Random Access containers,...

前一篇文章討論 iterators 時，我大量而深入地介紹了 traits 技術，同時也帶你看看 STL 中的相關原碼，並示範設計一個訂製型 iterator。會做這麼深度的探索，是因爲 iterator 十分抽象，而且扮演十分重要的膠合角色，沒有它就不可能將資料結構與演算法乾淨切割開來。

但是，夠了，我想。對於 STL 的實作技術以及原碼的探討，我想夠了。Container 是十分具象的東西，不論其所代表的資料結構特性，或是其操作行爲，都非常爲大家所熟悉。探討 containers 時，我將以一種新的角度進行。

●containers 元素的必要條件

STL 提供多種 containers。以特性分類的方式來帶它們出場，自然是個好方法。但是在此之前，我必須先做一些廣泛性的探討。

由於每個 containers 都提供一組操作行爲（例如 stack 提供 push 和 pop，vector 提供 insert，list 提供 push_back, pop_back, push_front, pop_front），所以其所內含的元素，必須滿足以下三個條件：

1. 元素必須是可拷貝的（copyable，經由 copy constructor 完成）。所有 containers 都會產生一份自己的元素副本，這樣才不會發生 alias 問題（意指兩個不同的 objects 代表相同的一塊記憶體）。所有 containers 操作行爲傳回的也都是其元素的一個副本。因此，copy constructor 的執行非常頻繁。
2. 元素必須是可指派的（assignable，經由 assignment operator 完成）。container 的操作行爲，以及各種 STL algorithms，都利用 assignment operator 來爲元素指派新值。
3. 元素必須是可摧毀的（destroyable，經由 destructor）。當使用者將元素從 container 中移除，意味 container 必須摧毀其元素。因此元素型別所具備的 destructor 絕不能設計爲 private。

這三個條件對所有 C++ classes 而言其實已經隱喻存在。

●"value" 語意與 "reference" 語意

所有的 containers 都保有它自己的一份資料。換句話說它內部的元素和你指派給它的元素，兩者的關係只是「內容相等（equal）」而非「視爲同一（identical）」。這種行爲我們稱爲「value semantics（語意）」。

然而實用上有時候需要所謂的「reference semantics」：讓 containers 內含 object reference 而不是 object 的實值，這可以明顯改善執行效率，而且可以造成多份資料（邏輯觀點而言）具備同步更新的效果（因爲所有資料其實源自一處）。然而就好像函式引數的傳遞一樣，by reference 總是比 by value 困難度高，容易出錯。事實上 STL containers 並不支援 reference semantics：

```
int ia[] = {1,2,3,4,5,6,7};

// value semantics.
vector<int>  iv(ia, ia+7);
for(int i=0; i<iv.size(); ++i)
   cout << iv[i] << ' ';      // 1 2 3 4 5 6 7

// reference semantics. STL containers not support.
vector<int&> irv;  // error!
```

爲求 reference semantics，我們得自己動手。最明顯的作法就是使用指標：

```
// reference semantics. using pointer.
vector<int*> ipv;
for(int i=0; i<sizeof(ia)/sizeof(ia[0]); ++i)
   ipv.push_back(&ia[i]);
for(int i=0; i<ipv.size(); ++i)
   cout << *(ipv[i]) << ' ';   // 1 2 3 4 5 6 7
```

搭配物件導向設計時，爲求實現多型（polymorphism）的效應，必須使用 reference semantics（因爲多型只有在使用 pointer 或 reference 時才得實現）。下面是個例子：

```
class shape
{  public: virtual void display()=0; };
class rect : public shape
{  public: virtual void display() { cout << "rect" << ' '; }  };
class circle : public shape
{  public: virtual void display() { cout << "circle" << ' '; }  };
class square : public rect
{  public: virtual void display() { cout << "square" << ' '; }  };

int main()
{
```

```
    rect r;
    circle c;
    square s;

    shape* a[] = {&r, &c, &s};   // polymorphically
    vector<shape*> sv(a, a+3);

    // polymorphism
    for(int i=0; i<sv.size(); ++i)
        (sv[i])->display();       // rect circle square
}
```

不過，直接使用原生指標會有一些常見的問題。例如它們所指的標的物可能不再存在，而元素間的比較結果也可能不如預期，因爲比較的是指標而非標的物本身。理想的作法是設計出 smart pointer，這是一種行爲像指標但有額外的檢驗能力或資料處理能力的 class（技術面而言，就是一種對 operator-> 和 operator* 做多載化動作的 class）。問題是該多麼 smart？例如 C++ Standard Library 提供的 auto_ptr 就不足以堪此大任。你得自己寫堪用的 smart pointer 才成。稍後我會示範這種技法。

●STL containers 分類

基本上 STL containers 分爲以下數類。如果要對每一種 container 的操作都寫出範例，恐怕篇幅過大且沒有必要。每一種 container 所支援的操作行爲，在 [Josuttis99] 中有非常完整詳盡的說明。

sequence containers（有序性容器）：

容器中的元素是有序的（ordered），但並未排序（sorted）。包括：

1. vector：動態陣列，可取代 C++ 語言本身提供的傳統陣列。提供隨機存取的能力。操作尾端元素的速度最快。由於所有元素佔用連續空間，所以一旦進行安插或移除動作，有可能使原本的某些 iterators 失效。
2. deque：雙向佇列。行爲與特性都很類似 vector。但因爲是雙向開口，所以操作兩端元素的速度都很快，不像 vector 只在操作尾端元素時才有高效率。由於所有元素佔用連續空間，所以一旦進行安插或移除動作，有可能使原本的某些 iterators 失效。
3. list：雙向串列。不提供隨機存取能力。在任何位置做安插或移除動作都很快，不像 vector 只侷限於尾端而 deque 只侷限於兩端才有高效率。由於各元素並不佔用連續空間，所以一旦進行安插或移除動作，原本的 iterators 仍然有效。注意，許多旨在「元素搬移」的 STL algorithms，用於 list 身上會有不佳的效率。所幸這些 STL algorithms 都有對應的 list member functions 可以瓜代。後者之所以有比較好的效率，原因是它們只操作指標，而非真正去拷貝或移動元素。

associative containers（關聯性容器）：

容器中的元素都經過排序（sorted）。包括：

4. set：經過排序的結構體，以某個可指定的排序方式來排序。每個元素獨一無二。由於已經排序，所以搜尋速度極快。但也因此不允許我們直接修改某個元素的內容，因爲這可能會影響排序。修改元素內容的正確的作法是，先將該元素移除，再加入（此時使用新值）。set 通常是以 balanced binary tree 來實作（但並不強制規定），甚至是以 red-black trees 完成。
5. multiset：允許元素重複。
6. map：由一對一對的「鍵值（key）/實值（value）」所組成的排序結構體。鍵值獨一無二。map 通常是以 balanced binary tree 來實作（但並不強制規定）。事實上 map 的內部結構通常與 set 是一樣的，因此我們可以將 set 視爲一種特殊的 map：鍵值和實值相同的 map。所以 map 和 set 幾乎擁有完全相同的能力。
7. multimap：允許鍵值重複。
8. hash table：這並不是 C++ Standard 規範內的一個 container（因標準委員會作業時間的關係），但是它對於大量資料的搜尋而言，很實用也很重要。有許多 STL 實作品（例如 SGI）都涵蓋了它。通常 STL 產品廠商會提供四種 hash tables：hash_set, hash_multiset, hash_map, hash_multimap。

containers adaptor（容器配接器）：

這是以某種 STL container 做爲底子，並修改其介面，使其擁有不同的風貌。包括：

9. stack：特性是先進後出（FILO，First In, Last Out）。底部定義是 deque。
10. queue：特性是先進先出（FIFO，First In, First Out）。底部預設是 deque。
11. priority_queue：特性是依優先權來決定誰是「下一個」元素。底部預設是 vector。

幾近 STL containers 的資料結構：

12. string：字串，內放字元。注意，string 其實不是一個 class，而是一個 typedef。它的真正型別是：

```
template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
class basic_string;

typedef basic_string<char> string;
```

13. bitset：內放 bits 的一種結構體。每個 bit 可表示一個 flag。它的長度固定，因爲長度便是其 template 引數。例如：

```
// vc6[o]  bcb4[x]  g++[x]
#include <bitset>
```

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

int main()
{
bitset<10> bs1(7);
bitset<10> bs2(string("1000101011"));

cout << bs1 << endl;   // 0000000111
cout << bs2 << endl;   // 1000101011
}
```

如果你需要動態調整大小，可以 vector<bool> 取代。

14. valarray：valarray 是數學中的線性數列概念的一種呈現。它帶來的好處是，我們可以像對單一數值一樣的方式，對整個 valarray（中的每個元素）做運算動作。

```
// vc6[o]  bcb4[o]  g++[x]
#include <valarray>
#include <iostream>
using namespace std;

template <class T>
void printValarray (const valarray<T>& va)
{
  for(int i=0; i<va.size(); ++i)
     cout << va[i] << ' ';
  cout << endl;
}

int main()
{
  valarray<int> va1(4);
  printValarray(va1);      // 未指定初值（內容未定）

  valarray<int> va2(3, 4);
  printValarray(va2);      // 3,3,3,3

  int ia[] = {1,2,3,4,5,6};
  valarray<int> va3(ia, sizeof(ia)/sizeof(ia[0]));
  printValarray(va3);         // 1,2,3,4,5,6

  valarray<int> va4(ia+1, 4);
  printValarray(va4);         // 2,3,4,5

  va1 = (va2 + va4) * va4;
  printValarray(va1);         // 10,18,28,40
}
```

●以 smart pointer 來實作 STL containers 的 reference semantics

讓我重新回到 reference semantics 這個主題。前面曾說，STL containers 只支援 value semantics，但某些時候是有必要使用 reference semantics，尤其在搭配 polymorphism 特性時。這時候我們得自己動手，以指標來實現 reference semantics。

但是，使用原生指標有一些無法避免的缺點。最好是以 smart pointer 取而代之，如此一來便可以掌控所有對指標的行爲，達到我們可能需要的檢驗或加工處理的目的。雖然，它的效率會比直接使用原生指標差一些些。

使用 smart pointer，我們便可完全控制：

(1) 建構（construction）與解構（destruction）。

(2) 拷貝（copying）和指派（assignment）動作。

(3) 提領（dereferencing）動作。

就實作面而言，smart pointer 至少必須對 operator* 和 operator-> 進行多載化。此外，原生指標所需的算術運算如 ++ 和 --，邏輯運算如 == 和 !=，也都應該視需要而多載化。如果你希望兩個 smart pointer 的比較動作比的是其所指標的物之大小，那麼還應該對 operator< 和 operator> 進行多載化。

我希望示範一個 smart pointer，可以將本文早先出現過的 shape 階層體系，以 reference semantics 的方式分別包裝到一個 deque、一個 list 和一個 set 之中。我不但要實證它們仍然保有 polymorphism 的特性（一如使用原生指標該樣），還要實證它們有大小先後次序（才可以納入 set 之中），以及實證這三個 containers 的內容可以同步更新。這樣的技術可以應用於繪圖軟體。是的，或許你需要以不同性質的 containers 存放螢幕上的同一堆繪圖物件（各式各樣形狀），其中以 set 存放者，係以各繪圖物件的面積來排序。而如果某一個繪圖物件有了改變（例如擴充大小），不同 containers 中相應的那個元素應該同步更新（因爲資料畢竟只有一份）。

下面是 shape classes 階層體系：

```
class shape
{
public:
  virtual void display()=0;
  virtual int area()=0;
  virtual void expand(int d)=0;
};

class square : public shape
{
public:
```

```
  square(int x, int y, int w) :
    _x(x), _y(y), _w(w) { _area = _w * _w; }
  virtual void display()
    { cout << "square:" << _x << _y << _w << _area << endl; }
  virtual int area() { return _area; };
  virtual void expand(int d) { _w += d;  _area = _w * _w; }

protected:
  int _x, _y, _w, _area;
};

class rect : public square
{
public:
  rect(int x, int y, int w, int h) :
    square(x,y,w), _h(h) { _area = _w * _h; }
  virtual void display()
    { cout << "rect:" << _x << _y << _w << _h << _area << endl; }
  virtual int area() { return _area; };
  virtual void expand(int d) { _w += d;  _h += d; _area = _w * _h; }
protected:
  int _h, _area;
};

class circle : public shape
{
public:
  circle(int x, int y, int r) :
    _x(x), _y(y), _r(r) { _area = 3.14 * _r * _r; }
  virtual void display()
    { cout << "circle:" << _x << _y << _r << _area << endl; }
  virtual int area() { return _area; };
  virtual void expand(int d) { _r += d;  _area = 3.14 * _r * _r; }
protected:
  int _x, _y, _r;
  double _area;
};
```

你可以在網際網路上找到許多現成可用且免費的 smart pointers 程式碼；多半都以 template 完成，以保持最大彈性。下面這個參考自 [Josuttis99]：

```
template <class T>
class CountedPtr {
protected:  // [Josuttis99] 中原爲 private。爲讓它可被繼承，遂改。
  T* ptr;
  long* count;
public:
  explicit CountedPtr (T* p=0) : ptr(p), count(new long(1)) { }
  CountedPtr(const CountedPtr<T>& p) throw()
```

```
    : ptr(p.ptr), count(p.count) { ++*count; }
  ~CountedPtr() throw() { dispose(); }

  CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {
    if (this != &p) {
        dispose();
        ptr = p.ptr;
        count = p.count;
        ++*count;
    }
    return *this;
  }

  T& operator*() const throw()  {  return *ptr;  }
  T* operator->() const throw() {  return ptr;   }

private:
  void dispose() {
    if (--*count == 0) {
      delete count;
      delete ptr;
    }
  }
};
```

現在我可以這樣把資料以 reference semantics 的方式放進 deque 和 list 內：

```
rect* pr = new rect(1,2,3,4);
circle* pc = new circle(1,2,5);
square* ps = new square(1,2,3);
shape* a[] = {pr, pc, ps};   // polymorphically

typedef CountedPtr<shape> shapePtr;
deque<shapePtr> col1;
list<shapePtr> col2;

for (int i=0; i< sizeof(a)/sizeof(a[0]); ++i) {
  shapePtr ptr(a[i]);
  col1.push_back(ptr);
  col2.push_front(ptr);
}
```

然後將它們印出來檢驗。以下用到的 for_each 是個 STL 泛型演算法，它接受兩個 iterators，標示出一個範圍，然後巡訪其間的每一個元素，並將第三個引數所表現的一個 function object（或一個函式）施行於該元素身上。

```
for_each(col1.begin(), col1.end(), printCountedPtr);
/* 輸出結果
```

```
rect:1 2 3 4 12
circle:1 2 5 78.5
square:1 2 3 9
*/

for_each(col2.begin(), col2.end(), printCountedPtr);
/* 輸出結果
square:1 2 3 9
circle:1 2 5 78.5
rect:1 2 3 4 12
*/
```

其中的 printCountedPtr() 是針對 CountedPtr<shape> 的輸出函式：

```
void printCountedPtr(CountedPtr<shape> elem)
{  elem->display(); }
```

看起來很不錯。現在我打算把原先的那一套繪圖物件放進 set 中：

```
set<shapePtr> col3;
col3.insert(ptr);  // error!
```

第二行會失敗，原因是編譯器找不到 set 排序所需的 operator<。如果要把同一組資料放進 set 內，我必須爲 smart pointer 多載化 operator<。但是 CountedPtr<T> 不宜更動，因爲它是一個泛型設計，不宜針對特定數值來比大小。於是我再設計一個 class，繼承自 CountedPtr<shape>，於其中加上 operator< 的設計。我以 shape classes 的面積做爲比大小的標準。

```
class sPtr : public CountedPtr<shape>
{
public:
  explicit sPtr(shape* p) : CountedPtr<shape>(p) {   }
  bool operator<(const sPtr& rhs) const   // 以面積來比大小
    { return ( ((*this).ptr)->area() < (rhs.ptr)->area() ); }
};
```

下面是針對 sPtr 的輸出函式：

```
void displayshape(sPtr elem)
{  elem->display(); }
```

於是，我可以這樣使用 set：

```
set<sPtr> col3;

for (i=0; i< sizeof(a)/sizeof(a[0]); ++i) {
  sPtr ptr(a[i]);
  col3.insert(ptr);   // 由於 sPtr 提供有 operator<，所以這動作沒問題
}
```

```
for_each(col3.begin(), col3.end(), displayshape);  //
/* 輸出結果。注意，set 內部已排序（以面積排序）
square:1 2 3 9
rect:1 2 3 4 12
circle:1 2 5 78.5
*/
```

當我將 col1 的第一個元素所指標的物擴充大小後：

```
(*col1.begin())->expand(7);
```

三個 containers（col1, col2 和 col3，分別是一個 deque 和一個 list 和一個 set）同受影響：

```
for_each(col1.begin(), col1.end(), printCountedPtr);
/* 輸出結果
rect:1 2 10 11 110
circle:1 2 5 78.5
square:1 2 3 9
*/

for_each(col2.begin(), col2.end(), printCountedPtr);
/* 輸出結果
square:1 2 3 9
circle:1 2 5 78.5
rect:1 2 10 11 110
*/

for_each(col3.begin(), col3.end(), printCountedPtr);
/* 輸出結果。set 排序受到破壞
square:1 2 3 9
rect:1 2 10 11 110
circle:1 2 5 78.5
*/
```

注意，直接修改元素可能造成 set 內部排序被破壞。所以，實在不應如此。修改 set 元素的正確作法是，先將該元素移除，再加入（此時使用新值）。不過這又帶來一個問題，由於三個 containers 指向同一份實際資料，造成所謂的 alias（別名）現象，我們進行資料的刪除時將會相當棘手，很容易顧此失彼，形成 dangling pointer（空懸指標）。因此在正面和負面之間，需要你以實用價值做爲判斷。

參考資料：

1. [Austern99] "Generic Programming and the STL" by Matthew H. Austern, AW, 1999
2. [Josuttis99] "The C++ Standard Library" by Nicolai M. Josuttis, AW, 1999
3. [Hughes99] "Mastering the Standard C++ Classes", Cameron Hughes and Tracey Hughes, Wiley, 1999

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。