泛型演算法（Generic Algorithms）與 Function Object

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

Adaptor（配接器）是什麼？凡是可以改變標的物介面的東西，即是。根據修飾的對象，STL adaptors 大致可分爲三類：

1. class adaptor
2. function (object) adaptor
3. iterator adaptor

以下兵分三路，各別討論這些物事。

●Class Adaptor

Class adaptor 本身也是一個 class，但其內部以某種 STL container 做爲底子，並修改其介面，俾能擁有特定的風貌。這樣的 class adaptors 包括：

1. stack：特性是先進後出（FILO，First In, Last Out）。底部容器預設是 deque。

```
// 摘自 STL SGI 版
template <class T, class Sequence = deque<T> >
class stack { ... }
```

2. queue：特性是先進先出（FIFO，First In, First Out）。底部容器預設是 deque。

```
// 摘自 STL SGI 版
template <class T, class Sequence = deque<T> >
class queue { ... }
```

3. priority_queue：特性是依優先權來決定誰是「下一個」元素。底部容器預設是 vector。

```
// 摘自 STL SGI 版
template <class T, class Sequence = vector<T>,
          class Compare = less<typename Sequence::value_type> >
class  priority_queue { ... }
```

下面這個例子實際操作了上述三種 container adaptor：

```
// vc6[x]  bcb4[o]  G++[x]
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int main()
{
  int ia[] = { 1,3,2,4 };
  deque<int> id(ia, ia+4);

  stack<int> istack(id);
  queue<int> iqueue(id);
  priority_queue<int> ipqueue(ia, ia+4);

  // 把各個 container 的所有內容列印出來
  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();
  }
}
```

究竟是什麼樣的結構，可以讓 class adaptor「以一個 container 爲底，並改變其介面」呢？說穿了它其實就是一個 class，內含一個由 container 構成的 data member，並提供一組必要的介面（例如 stack 的操作介面，或是 queue 的操作介面…），這些介面絕大多數又是以底部的這個 container 的 member functions 來完成。下面是摘錄自 STL SGI 版的 stack 原始碼，從這裡可以一目了然所謂 class adaptor 的技倆。

```
template <class T, class Sequence = deque<T> >
class stack {
  friend bool operator== (const stack&, const stack&);
  friend bool operator< (const stack&, const stack&);
public:
  typedef typename Sequence::value_type      value_type;
  typedef typename Sequence::size_type       size_type;
```

```
  typedef typename Sequence::reference      reference;
  typedef typename Sequence::const_reference const_reference;
protected:
  Sequence c;
public:
  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference top() { return c.back(); }
  const_reference top() const { return c.back(); }
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_back(); }
};

template <class T, class Sequence>
bool operator==(const stack<T, Sequence>& x, const stack<T, Sequence>& y) {
  return x.c == y.c;
}

template <class T, class Sequence>
bool operator<(const stack<T, Sequence>& x, const stack<T, Sequence>& y) {
  return x.c < y.c;
}
```

從這份 stack 原始碼可看出，它的每一個相關型別（associated types），都以其底部容器 Sequence 的相關型別爲依據；它的（幾乎）每一個 member functions，也都藉底部容器 Sequence 的相應 member functions 之力完成。

請注意，這份 stack 原始碼並未提供 stack 的任何 constructor，這也正是爲什麼前述那個範例無法通過 G++（使用 STL SGI 版）之故，因爲其中有這樣的動作：

```
int ia[] = { 1,3,2,4 };
deque<int> id(ia, ia+4);
stack<int> istack(id);  // 以 id 做爲 stack 的初值
```

而這段碼之所以能夠通過 C++Builder 4（使用 STL Rouge Wave 版），則是因爲 Rouge Wave 版的 stack 根據 C++ Standard 的要求，提供了 constructor 如下：

```
template <class T, class Container = deque<T> >
class stack
{
protected:
  Container c;

public:
  explicit stack(const Container& co = Container()) : c(co)
  { ; }
...
}
```

●function (object) adaptor

STL 提供有一組 function adaptors，用來特殊化或擴充單一運算元或雙運算元的 function objects。區分爲以下兩大類型：

1. Binders：所謂 binder 可將一個雙運算元的 function object 轉換爲單運算元的 function object，方法是將兩個引數之一繫結(binding)爲某特定數值。STL 提供有兩個預先定義的 binder adaptors：bind1st 和 bind2nd。bind1st 將數值繫結至二元 function object 的第一個引數上，bind2nd 將數值繫結至第二個引數。例如，爲計算一個 container 之中所有小於或等於 5 的元素，我們可以這麼做：

```
count_if( vec.begin(), vec.end(),
          bind2nd( less_equal<int>(), 5 ));
```

於是 less_equal<int>() 的第二個運算元被繫結爲 5，形成「小於或等於 5」的語意。

2. Negators：所謂 negator 會將一個 function object 的真假值反相顛倒。STL 也提供有兩個預先定義的 negator adaptors：not1 和 not2。not1 顛倒的是一元 function object 的真假值，而 not2 顛倒的對象是二元 function object 的真假值。例如爲了否定 less_equal，我們應該這麼寫：

```
count_if( vec.begin(), vec.end(),
          not1( bind2nd( less_equal<int>(), 5 )));
```

於是形成了「小於或等於 5」的否定語意，也就是「大於 5」。

圖 1 是這四個常用的 function adaptors 的意義整理。

圖 1\ 四個常用的 function adaptors 的意義整理

算式型式	意義
bind1st(op, value)	op(value, param)
bind2nd(op, value)	op(param, value)
not1(op)	!op(param)
not2(op)	!op(param1, param2)

●member function adaptor

如果你希望針對某個 container 的每一個元素，都呼叫元素型別的某個 member function，該怎麼做？這下子可不能再以 function object 放到演算法之中了。

幸運的是 STL 提供了兩個所謂的 member function adaptors：mem_func_ref 和 mem_fun。當我們的 container 元素是 class object，就用前者；當我們的 container 元素是 pointer to class object，就用後者。下面是個例子：

```
// vc6[x]  cb4[x]  g++[o]
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <functional>  // for mem_fun_ref, mem_fun
using namespace std;

class Student {
private:
  string _name;
public:
  Student(const string& name) : _name(name) { }
  void print() const { cout << _name << ' ';  }
};

int main()
{
  Student s1("張三"), s2("李四"), s3("王五");
  s1.print();
  s2.print();
  s3.print();

  vector<Student> sv;
  sv.push_back(s1);
  sv.push_back(s2);
  sv.push_back(s3);
  for_each(sv.begin(), sv.end(), mem_fun_ref(&Student::print));
  // 張三 李四 王五

  vector<Student*> spv;
  spv.push_back(&s1);
  spv.push_back(&s2);
  spv.push_back(&s3);
  for_each(spv.begin(), spv.end(), mem_fun(&Student::print));
  // 張三 李四 王五
}
```

●ordinary function adaptor

當我們需要一個 function object 時，有時候函式指標可以取而代之。但是函式指標畢竟不是真正的 function object，它不能和任何 function adaptor 配合使用，威力大減。STL 提供了一個 ptr_fun，可以將一般函式修飾爲真正的 function object。用法如下（舉例）：

```
find_if(c.begin(), c.end(), not1(ptr_fun(ordinaryfunc)));

find_if(c.begin(), c.end(), bind2nd(ptr_fun(ordinaryfunc), param));
```

●composing function adaptors

STL 提供的組件搭配能力，應該能夠讓我們以簡單的 function objects 搭配組合出非常複雜的結果。可惜的是 STL 並未提供足夠的 adaptor 給我們。倒是 C++ 社群中流傳不少這類有用的 adaptors。STL SGI 版也提供了兩個：compose1 和 compose2。

compose1 的用途是：以一個一元運算式的結果做爲另一個一元運算式的引數。下面是個運用實例：

```
#include <vector>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
  vector<int> iv;
  for(int i=1; i<=9; ++i)
     iv.push_back(i);
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
  // 1 2 3 4 5 6 7 8 9

  transform(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "),
            compose1(bind2nd(multiplies<int>(), 2),
                     bind2nd(plus<int>(), 5)));
  // 12 14 16 18 20 22 24 26 28
}
```

最後一個式子的意思是，將 iv 的所有元素施行以某運算，然後放到 cout 去。指定的運算行爲是「先將元素加 5，再乘以 2」。

compose2 的用途是：允許我們以邏輯運算方式，串連兩個條件，使成爲單一條件。下面是個例子：

```
#include <vector>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
  vector<int> iv;
  for(int i=1; i<=9; ++i)
     iv.push_back(i);
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
  // 1 2 3 4 5 6 7 8 9
```

```
  vector<int>::iterator ite;
  ite = remove_if(iv.begin(), iv.end(),
                  compose2(logical_and<bool>(),
                           bind2nd(greater<int>(), 3),
                           bind2nd(less<int>(), 8)));

  iv.erase(ite, iv.end());
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
  // 1 2 3 8 9
}
```

其中 remove_if 式子的意義是：以某指定條件判定 iv 的每一個元素是否需要移除。此處的指定條件是：「大於 3 且小於 8」。

●iterator adaptor

我曾經在本系列第二篇文章中，深度剖析了 iterator 的技術成份，以及五種類型的 iterators：Input, Output, Forward, Bidirectional, RandomAccess。除了這五種 iterators，爲了運應各種應用場合，STL 還導入了許多其他型式的 iterators。

◎Insert Iterator

通常，STL algorithms 運用 iterator 時，如果做爲寫入之用，都是將資料以指派（assignment）而非安插（insertion）的方式放入 iterator 所指位置。如果 iterator 所指的並不是一個可放資料的空間（例如一個空的 container），就會出錯。例如：

```
int ia[] = { 1,1,1,3,3 };
vector<int> iv1(ia, ia+5), iv2;
unique_copy(iv1.begin(), iv1.end(), iv2.begin());  // 錯誤。iv2 是空的。
```

爲了能夠很方便地以 insertion 動作取代 assignment 動作，STL 提供了以下三個 insert iterator（adaptors）：

1. back_inserter()。它需要一個 container 做爲引數，並會造成該 container 的 push_back() 安插動作被喚起，以取代 assignment 動作。
2. front_inserter()。它需要一個 container 做爲引數，並會造成該 container 的 push_front() 安插動作被喚起，以取代 assignment 動作。由於 vector 並不支援 push_front()，所以不能夠在 vector 身上使用 front_inserter()。
3. inserter()。它需要兩個引數：一是 container 本身，另一是「指入 container 內」的一個 iterator，用以標示安插動作的起始位置。它會造成 container 的 insert() 安插動作被喚起，以此取代 assignment 動作。

實例如下：

```
#include <list>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
  int ia[] = { 1, 2, 3, 4, 4, 5, 6, 6, 9 };
  list<int> ilist;

  // 將 [ia, ia+9) 中獨一無二的元素，循序安插於 ilist 的尾端
  unique_copy(ia, ia+9, back_inserter(ilist));
  copy(ilist.begin(), ilist.end(), ostream_iterator<int>(cout, " "));
  // 1 2 3 4 5 6 9

  // 將 [ia, ia+9) 中獨一無二的元素，循序安插於 ilist 的頭端
  unique_copy(ia, ia+9, front_inserter(ilist));
  copy(ilist.begin(), ilist.end(), ostream_iterator<int>(cout, " "));
  // 9 6 5 4 3 2 1 1 2 3 4 5 6 9

  // 找出 ilist 中 3 的第一次出現位置
  list<int>::iterator ite = find(ilist.begin(), ilist.end(), 3);
  // 將 [ia, ia+9) 中獨一無二的元素，循序安插於 ite 處
  unique_copy(ia, ia+9, inserter(ilist, ite));
  copy(ilist.begin(), ilist.end(), ostream_iterator<int>(cout, " "));
  // 9 6 5 4 1 2 3 4 5 6 9 3 2 1 1 2 3 4 5 6 9
}
```

◎Reverse Iterators

大部份 containers 都擁有一對函式：rbegin() 和 rend()，它們分別傳回一個 reverse iterator，指向 container 的第一元素和 container 的「最後元素的下一元素」。所謂 reverse iterator，意思是利用它來遊歷 container 時，次序乃由最後一個元素前進到第一個元素。

對於一個 forward iterator 而言，++ 係用來存取下一個元素；但對於一個 reverse iterator 而言，++ 存取的是前一個元素。雖然 ++ 運算子和 -- 運算子的意義顛倒，似乎會讓人迷惑，但是它讓程式員可以傳遞一對 reverse iterators 給演算法，完成反方向動作。例如，爲了以遞減次序對 vector 排序，我可以簡單地將一對 reverse iterators 交給 sort()：

```
// 以漸增次序做排序動作
sort( vec0.begin(), vec0.end() );

// 以漸減次序做排序動作
sort( vec0.rbegin(), vec0.rend() );
```

◎iostream Iterators

STL 提供有對 iostream iterators 的支援。istream_iterator 支援 istream 及其 derived classes（例如 ifstream）身上的 iterator 操作動作。ostream_iterator 則支援在 ostream 及其 derived classes（例如 ofstream）身上的 iterator 操作動作。欲使用上述任何一種 iterator，必須先含入表頭檔：

```
#include <iterator>
```

1. istream_iterator

istream_iterator 的宣告方式如下：

```
istream_iterator<Type> identifier( istream& );
```

其中 Type 代表任何型別，唯其必須定義有 input 運算子（operator<<）。你可以指定一個 istream class object（例如 cin），或是其任何一個 publicly derived class（例如 ifstream），做爲建構式的引數。一旦如此，此一 istream_iterator 便與該 input stream 繫結在一起；於是程式中對此 iterator 的任何動作，都會被導引至該 input stream 身上。如果沒有爲建構式指定引數，便是產生一個表現 end-of-stream 狀態的 iterator。例如：

```
istream_iterator<int> iite(cin), eos;
```

那麼，iite 和 eos 兩個 iterators 便構成一個繫結自 cin 的首尾範圍。任何演算法，例如 copy，對這個範圍做動作，意思便是從 cin 讀取資料。例如：

```
vector<int> iv;
copy(iite, eos, inserter(iv, iv.begin()) );
```

2. ostream_iterator

ostream_iterator 的宣告型式如下：

```
ostream_iterator<Type> identifier( ostream& )
ostream_iterator<Type> identifier( ostream&, char* delimiter )
```

其中 Type 代表任何，唯其必須定義有 output 運算子（operator>>）。delimiter 表示一個 C-style 字串，會緊跟在每一個元素之後被輸出到 output stream 去。例如：

```
ostream_iterator<string> oite(cout, " ");
```

還記得嗎，在整系列文章中，每當我想要將一個 container 的所有元素輸出到螢幕上時，常常這麼做：

```
copy(c.begin(), c.end(), ostream_iterator<T>(cout, " "));
// T 表示 container c 的元素型別
```

●結語

利用五篇文章，我將泛型程式設計的觀念、STL 的概念、以及 STL 的用法，爲讀者做了一個大局觀。

掌握這些概念，便掌握了 STL 的重心。當然，實際運用 STL 時，你還需要一些好書在手邊。[Josuttis99] 和 [Austern99]都是我推崇的好書，它們的特性簡介，可在 RunPC 二月號的無責任書評專欄找到。

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. [Lippman98] "C++ Primer 3/e", by Lippman & Lajoie, AW, 1998

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。