泛型指標（Iterators）與 Traits 技術

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

Iterators（泛型指標）的中心觀點就是：如何讓資料結構和演算法兩者之間能夠彼此不互知地發展，而最後又能夠沒有間隙地膠合在一起。

註 1：本文對 iterator 來龍去脈的說明，大量得助於 [Austern99]。該書頗多筆誤，本文已修正，並附上一個完整範例程式。

註 2：本文所描述的 traits 技術，大量運用 struct。C++ 的 struct 幾乎等同於 class，唯一差別在於其內的存取層級預設爲 public，而 class 內的存取層級預設爲 private。

●一個簡易的例子

試看以下例子。假設有一個搜尋演算法 find()，以線性搜尋的方式尋找指定的某個元素。這個任務可輕易以 function tempalte 完成：

```
template <class I, class T>
I find(I first, I last, const T& value)
{
  while (first != last && *first != value)
    ++first;
  return first;
}
```

其中 template 參數 I 代表一個「類似指標」的東西，T 代表搜尋標的物的型別。由於我們要求 I 具備類似指標的性質，所以它應有提領（dereference）、累進（increment）的能力。這個 function template find() 是一個具備泛型概念的演算法。

現在，假設有一個原本就已發展好的資料結構 int_node：

```
struct int_node {
  int val;
  int_node* next;
};
```

如何將這個資料結構套用在上述的 find() 演算法當中呢？我們需要加上一層外包裝。事實上，這種

「加一層間接性」以解決問題的作法，層出不窮地出現在電算領域中。

我們希望寫出一個 wrapper class，權充 int_node 的指標。當我們對它做提領（dereference）動作時，傳回的是標的物 int_node；當我們對它做累進（increment）動作時，則造成它指向（代表）下一個 int_node object。爲了讓這個 wrapper 適用於任何型別的 node（而不只限於 int_node），可以將它設計爲一個 class template。以下就是我們的 node_wrap：

```
template <class Node>
struct node_wrap {
  Node* ptr;

  node_wrap(Node* p = 0) :  ptr(p) { }  // default ctor
  Node& operator*()  const { return *ptr; }
  Node* operator->() const { return ptr;  }

  // pre-increment operator
  node_wrap& operator++()    { ptr =  ptr->next; return *this; }
  // post-increment operator
  node_wrap  operator++(int) { node_wrap tmp = *this; ++*this; return tmp; }

  bool operator==(const node_wrap& i) const { return ptr == i.ptr; }
  bool operator!=(const node_wrap& i) const { return ptr != i.ptr; }
};

bool operator==(const int_node& node, int n) {  return node.val == n; }
bool operator!=(const int_node& node, int n) {  return node.val != n; }
```

現在，我們可以這樣子將它們搭配起來：

```
void main()
{
  int_node *list_head, *list_tail;
  int_node *in = new(int_node);
  in->val = 100;
  in->next = 0;
  list_head = list_tail = in;   // list 頭尾

  for (int i=0; i<10;++i)
  {
    int_node* in = new(int_node);
    in->val = i;
    in->next = 0;

    list_tail->next = in;      // list 串接
    list_tail = in;
  }

  // 此時的 list 內容爲 100,0,1,2,3,4,5,6,7,8,9

  node_wrap<int_node> r;   // VC6[o]  CB4[o]  G++[o]
```

```
  r = find(node_wrap<int_node>(list_head), node_wrap<int_node>(), 10);
  if (r != node_wrap<int_node>())
    std::cout << (*r).val << std::endl;   // output: none

  r = find(node_wrap<int_node>(list_head), node_wrap<int_node>(), 3);
  if (r != node_wrap<int_node>())
    std::cout << (*r).val << std::endl;   // output: 3
}
```

我們可否在上述 list_head 和 list_tail 指出的串列形成之後，直接以 find(list_head, null, 5) 來搜尋 '5' 這個元素呢？如果這能夠成功，也就不需要大費周張地寫一個 node_wrap 了。是的，答案當然是不行，因爲 find() 之中對於 iterator 的累進（++）動作，list_node 無法瞭解，無法提供對應的服務。

●Iterator 相關型別（associated type）

上述的 node_wrap，提供了一個 iterator 雛形。如果將思緒拉得更遠更宏大一些，我們會發現，演算法之中用到 Iterator 時，很可能會使用其相關型別（associated type）。什麼是相關型別？例如「iterator 所指標的物」的型別便是。當你有需要在演算法中宣告一個以「iterator 所指標的物之型別」爲型別的變數，如何是好？畢竟 C++ 並未支援 typeof(*iterator) 這樣的指令！

有一個解決辦法：利用 function template 的引數推導。例如：

```
// VC6 [o], BCB4 [o], G++ [o]
template <class I, class T>
void func_impl(I iter, T t)
{
  T tmp;
  // ... 這裡做原本 func() 應該做的工作
};

template <class I>
inline
void func(I iter)
{
  func_impl(iter, *iter);
}

int main()
{
  int i;
  func(&i);
}
```

我們以 func() 爲對外介面，實際動作則全部設計在 func_impl() 中。由於 func_impl() 是一個 function

template，呼叫它時編譯器會自動進行 template 引數推導。於是導出上例的 T type。順利解決了問題。

template 引數推導機制（arguments deduction），在 STL 中佔非常重要的角色。Alexander Stepanov（STL 的創造者）在與 Dr. Dobb's Journal 進行的訪談中說道：『1992 我重回 generic-library 的開發工作。這時候 C++ 有了 template。我發現 Bjarne 完成了一個非常美妙的設計。之前我在 Bell Lab 曾參與數次 template 的相關設計討論，並且非常粗暴地要求 Bjarne 應該將 C++ template 設計得儘可能像 Ada generics 那樣。我想由於我的爭辯是如此地粗暴，他決定反對。我瞭解到在 C++ 中除了擁有 template classes 之外還擁有 template functions 的重要性。然而我認爲 template function 應該像 Ada generics 一樣，也就是說它們應該是 explicit instantiated。Bjarne 沒有聽進我的話，他設計了一個 template function 機制，其中的 template 是以一個多載化機制（overloading mechanism）來進行 implicitly instantiated。這項特殊的技術對我的工作具有關鍵性的影響，因爲我發現它使我得以完成 Ada 不可能完成的許多動作。我非常高興 Bjarne 當初沒有聽我的意見。』（請參考 DDJ 1995 年三月號）

●之一：value type

Iterator 所指標的物之型別，我們稱之爲該 iterator 的 value type。上述所謂的 "type inference trick"（型別推論技巧）雖然可用，卻非全面可用：萬一 value type 必須用於函式的傳回值，就沒輒了，畢竟函式的回返型別並不在 template 引數推導的參考範圍內。

我們需要其他方法。使用巢狀式的型別宣告似乎是個好主意。這次的作法**有點像**先前所提的 node_wrap：

```
// VC6 [o], BCB4 [o], G++ [o]
#include <iostream>

template <class T>
struct MyIter {         // like node_wrap.
  typedef T value_type;  // nested type（巢狀式的型別宣告）
  T* ptr;
  MyIter(T* p=0) { ptr = p; }
  ~MyIter() { delete ptr; }
  // 這裡還應該有身爲一個 iterator（pointer-like）所該有的其他介面
  // 例如 operator*, operator->, operator++, ...
};

template <class T>
typename MyIter<T>::value_type  // 這一整行是 func 的回返型別 (註 1)
func(MyIter<T> ite)
{ return *(ite.ptr);  };

void main()
{
  MyIter<int> ite(new int(8));
```

```
  std::cout << func(ite) << std::endl;  // 8
}
// 註 1：此處須使用關鍵字 typename，原因是 T 爲一個 template parameter，
// 在具現化之前，編譯器對 T 一無所悉，所以編譯器不知道 MyIter<T>::value_type
// 代表的是一個 type 或是一個 member function 或是一個 data member。
// 使用 typename 可以告訴編譯器說這是一個 type，俾得順利通過編譯。
```

當我們把型別爲 MyIter<T> 的 ite 傳入函式時，函式便可以使用 typename MyIter<T>::value_type 做爲型別；至於 ite 的指標性質則以 ite.ptr 來遂行。

這看起來不錯。但是有個陷阱存在。並不是所有的 iterator 都是 class 或 struct。原生指標就不是！但是 STL 必須接受原生指標做爲一種 iterator。所以上面這樣還是不夠。有沒有什麼辦法可以讓上述的泛型概念針對特定的某些情況做特殊的設計呢？

有，利用 partial specialization 就可以做到。

任何完整的 C++ 書籍對於 partial specialization 均有說明。大致的意思是：如果 class template 擁有一個以上的 template 參數，我們可以針對某一個（或某一組）template 參數（而非針對所有的 tempalte 參數）進行特殊化。也就是說，可以供應一個 template 版本，符合一般化條件，但其中某些 template 參數已經被實際型別或數值取代。這可以被用來定義一個比泛型版本更專屬或是更有效率的實作品。

如果有一個 class template 如下：

```
template<typename U, typename V, typename T>
class C { ... };
```

上述對 partial specialization 的定義，容易誤導我們以爲所謂「局部性特製版本」一定是對 template 參數 U 或 V 或 T（或其任意組合）指定特定引數值。事實不然，[Austern99] 對於 partial specialization 的定義使我們得以跳脫這樣的框框。他說：『所謂 partial specialization 的意思是提供另一份 template 定義式，而其本身仍爲 templatized』。

由此，面對以下的 class template：

```
template<typename T>
class C { ... };      // 這個泛型版本適用於任何型別的 template 引數
```

我們便容易接受它有一個型式如下的 partial specialization：

```
template<typename T>
class C<T*> { ... };  // 這個特殊版本適用於 template 引數爲指標者
```

有了這項利器，我們可以解決前述「巢狀式型別宣告」未能解決的問題。先前的問題是，原生指標不是 class，因此無法爲它們定義巢狀型別。現在，我們可以爲指標型別的 template 引數設計 partial specialization。

現在讓我們設計一個 class template 如下，專門用來「抽取」iterator 的 value_type：

```
template <class Iterator>
struct iterator_traits {   // traits 是「特性」的意義
  typedef typename Iterator::value_type value_type;
};
```

它有兩個 partial specializations：

```
template <class T>
struct iterator_traits<T*> {  // 針對「template 引數爲指標型別」的特殊版本
  typedef T value_type;
};

template <class T>
struct iterator_traits<const T*> { // 針對「template 引數爲 const 指標型別」的特殊版本
  typedef T value_type;  // 注意，型別爲 T 而非 const T
};
```

於是當我們在 template function 中需要 iterator I 的 value type 時，便可以這麼寫：

```
typename iterator_traits<I>::value_type;
```

不論 I 是任何型式的 iterator，甚至是原生指標，上一行都成立。

這樣的型別可以運用做爲 algorithms 的函式回返型別，於是解決了先前的問題。

●之二：difference type

另一個與 iterator 相關的型別是其 difference type，用來表示兩個 iterator 間的距離。也可以用來表示一個容器的最大容量。

這個問題仍然可利用前述的 traits 技術解決。至於原生指標，我們可以設計對應的 partial specializations，在其中使用 C++ 語言內建的 ptrdiff_t（定義於 <cstddef> 表頭檔中）：

```
template <class Iterator>
struct iterator_traits {
  typedef typename Iterator::value_type      value_type;
  typedef typename Iterator::difference_type  difference_type;
};

// partial specialization for regular pointers
template <class T>
struct iterator_traits<T*> {
  typedef T         value_type;
  typedef ptrdiff_t  difference_type;
};

// partial specialization for regular const pointers
template <class T>
```

```
struct iterator_traits<const T*> {
  typedef T         value_type;
  typedef ptrdiff_t  difference_type;
};
```

於是當我們在 template function 中需要 I 的 difference type 時，便可以這麼寫：

```
typename iterator_traits<I>::difference_type;
```

●之三：reference type

從「iterator 所指標的物之內容是否允許改變」的角度觀之，iterators 分爲兩種：const（常數的）iterators 和 mutable（可變的）iterators。當我們面對 mutable（可變的）iterators 做提領（dereference) 動作時，獲得的不應該是個 rvalue，必須是個 lvalue。C++ 函式如要傳回 lvalue，都是以 by reference 的方式傳回，所以當 p 是一個 mutable iterators 時，如果其 value type 是 T，那麼 *p 的型別不應該是 T，而應該是 T&。以此道理擴充到 const iterators p 身上，如果其 value type 是 T，那麼 *p 的型別不應該是 const T，而應該是 const T&。

換句話說 *p 的型別不應該是 p 的 value type，而應該是所謂的 reference type。

●之四：pointer type

pointers 和 references 在 C++ 中有非常密切的關連。如果「傳回一個 lvalue，代表 p 所指之標的物」是可能的，那麼「傳回一個 lvalue，代表 p 所指之標的物的位址」也一定是可能的。也就是說，我們一定（必須）能夠傳回一個 pointer，指向該 object。

事實上這些 types 已經出現在本文一開始的 node_wrap class 中。該 class 的 operator* 傳回一個 Node&，operator-> 傳回一個 Node*。前者便是其 reference type，後者便是其 pointer type。

現在我們把兩個新的 iterator associated types 加入 traits 內：

```
template <class Iterator>
struct iterator_traits {
  typedef typename Iterator::value_type        value_type;
  typedef typename Iterator::difference_type   difference_type;
  typedef typename Iterator::pointer           pointer;
  typedef typename Iterator::reference         reference;
};

// partial specialization for regular pointers
template <class T>
struct iterator_traits<T*> {
  typedef T                       value_type;
  typedef ptrdiff_t                difference_type;
  typedef T*                      pointer;
  typedef T&                      reference;
```

```
};

// partial specialization for regular const pointers
template <class T>
struct iterator_traits<const T*> {
  typedef T                         value_type;
  typedef ptrdiff_t                  difference_type;
  typedef const T*                   pointer;
  typedef const T&                   reference;
};
```

●concept Iterator 的分類

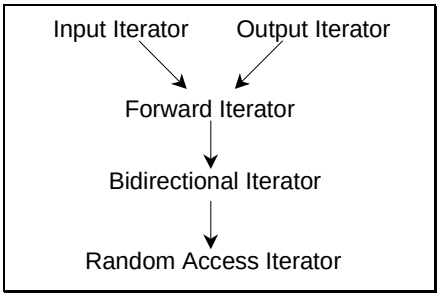
最後一個 iterator associated type 會引發比較大的寫碼工程。在那之前，我必須先討論 Iterator 的分類。

Iterators 是一個 concept，而指標是其中一個 model。Iterators 並非只是單一的 concept，它是一大家族，共有五個不同的 concepts：

1. Input Iterator - 不允許外界改變 iterator 所指物件。唯讀（read only）性質。
2. Output Iterator -- 唯寫（write only）性質。
3. Forward Iterator -- 允許「寫入型」algorithms 使用同一個範圍做讀/寫動作。這樣的 algorithms 例如 replace()。
4. Bidirectional Iterator -- 支援雙向移動。某些 algorithms 需要逆向巡訪某個範圍，例如逆向拷貝某範圍內的元素，就可以使用這種 Bidirectional Iterators。
5. Random Access Iterator -- 先前四種 iterator concepts 都只供應一小組指標算術運算能力（(1),(2),(3) 允許 operator++，(4) 允許 operator--），Random Access Iterator 則涵蓋其餘所有指標算術運算能力，包括 p+n, p-n, p[n], p1-p2, p1<p2。

此一 concept Iterators 的分類與從屬關係，可以圖一表示。直線與箭頭代表的並非 C++ 的繼承關係，而是 concept 與 refinement 的關係。

圖一/ concept iterators 的分類與從屬關係



●之五：iterator tags

通常，設計 algorithms 時，我們會令一個 algorithm 對某種 iterator concept 提供一個明確的定義，而針對該 concept 的 refinement 提供另一種定義。例如我們有個 algorithm 適用 Forward Iterator，那麼當然你以 Random Access Iterators 餵給他，他也會接受，因爲 Random Access Iterators 必然是 Forward Iterator（見圖一）。

但是可用（usable）並不代表最佳（optimal）。

以 advance() 爲例。這是其他 algorithms 內部常用的一個函式。此函式有兩個參數，iterator p 和數值 n；函式內部將 p 累進 n 次（前進 n 距離）。下面有三種不同的定義，一是針對 Input Iterator，一是針對 Bidirectional Iterator，一是針對 Random Access Iterator。並沒有特別針對 ForwardIterator 而設計的版本，因爲其動作和 InputIterator 版全無二致。

```
template <class InputIterator, class Distance>
void advance_II(InputIterator& i, Distance n)
{
  for ( ; n > 0; --n, ++i );  // 單向，逐一前進
}

template <class BidirectionalIterator, class Distance>
void advance_BI(BidirectionalIterator& i, Distance n)
{
  if (n >= 0)               // 雙向，逐一前進
     for ( ; n > 0; --n, ++i ) { }
  else
     for ( ; n < 0; ++n, --i ) { }
}

template <class RandomAccessIterator, class Distance>
void advance_RAI(RandomAccessIterator& i, Distance n)
{
  i += n;                   // 雙向，跳躍前進
}
```

現在，當程式呼叫 advance()，應該使用上述哪一個定義呢？如果選擇 advance_II，對 Random Access Iterator 而言就極度缺乏效率，原本 O(1) 的操作竟成爲 O(N)。如果選擇 advance_RAI，則它無法接受 Input Iterator。我們需要將三者合一。你可以想像成這樣：

```
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n)
{
  if (is_random_access_iterator(i))
    advance_RAI(i, n);
  else if (is_bidirectional_iterator(i))
    advance_BI(i, n);
  else
    advance_II(i, n);
}
```

但是在執行時期才決定哪一個版本，會影響程式效率。最好是在編譯期就選擇正確的版本。多載化函式機制可以達成這個目標。

目前這三個 advance() 版本的兩個函式參數，其型別都是 template 參數，我們必須加上第三個 non-template type（型別已確定的）函式參數，使函式多載化機制能夠有效運作。

設計考量是這樣的：如果能夠在 traits 內增加一個 iterator associated type，使其得以獨一無二地辨識出不同的 Iterator concept，我們便可以利用這個類似 ID 的東西做爲 advanced() 的第三個函式參數。這個類似 ID 的東西絕不能只是個常數 ID，因爲編譯器要仰賴其型別來決定執行哪一個多載化函式。那麼，最佳（也是唯一）的選擇就是把它們設計爲 classes（至於爲什麼用到繼承，稍後再解釋）：

```
// 五個 tag types
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag : public input_iterator_tag { };
struct bidirectional_iterator_tag : public forward_iterator_tag { };
struct random_access_iterator_tag : public bidirectional_iterator_tag { };
```

然後將 advance() 重新設計如下：

```
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n, input_iterator_tag)
{
  for ( ; n > 0; --n, ++i );  // 單向，逐一前進
}

// 一個單純的轉呼叫函式（trivial forwarding function）
template <class ForwardIterator, class Distance>
void advance(ForwardIterator& i, Distance n, forward_iterator_tag)
{
  advance(i, n, input_iterator_tag());  // 單純地轉呼叫（forwarding）
}
```

```
template <class BidiectionalIterator, class Distance>
void advance(BidiectionalIterator& i, Distance n, bidirectional_iterator_tag)
{
  if (n >= 0)              // 雙向，逐一前進
     for ( ; n > 0; --n, ++i ) { }
  else
     for ( ; n < 0; ++n, --i ) { }
}

template <class RandomAccessIterator, class Distance>
void advance(RandomAccessIterator& i, Distance n, random_access_iterator_tag)
{
  i += n;                  // 雙向，跳躍前進
}
```

注意上述語法，每個 advance() 的最後一個函式參數都只宣告其型別，沒有指定參數名稱 -- 因爲函式中根本不會用到該參數。

最後我們還需寫一個上層函式，呼叫上述的多載化 advance()。此一上層函式只需兩個參數，函式內自行加上第三個引數（五個 tag types 之一）後，呼叫上述的 advance()。因此，上層函式必須有能力從它所獲得的 iterator 中推導（取出）其 tag type：

```
template <class Iterator, class Distance>
inline void advance(Iterator& i, Distance n)
{
  advance(i, n, iterator_traits<Iterator>::iterator_category());
}
```

注意上述語法。iterator_traits<Iterator>::iterator_category() 產生一個暫時性物件，那當然會是前述五個 tag types 之一。根據這個暫時物件的確實型別，編譯器決定呼叫哪一個多載化的 advance() 函式。爲了滿足上述行爲，我們的 traits 必須再增加一個 associated type：

```
template <class Iterator>
struct iterator_traits {
  typedef typename Iterator::iterator_category    iterator_category;
  typedef typename Iterator::value_type           value_type;
  typedef typename Iterator::difference_type      difference_type;
  typedef typename Iterator::pointer              pointer;
  typedef typename Iterator::reference            reference;
};

// partial specialization for regular pointers
template <class T>
struct iterator_traits<T*> {
  typedef random_access_iterator_tag    iterator_category;
```

```
  typedef T                           value_type;
  typedef ptrdiff_t                   difference_type;
  typedef T*                          pointer;
  typedef T&                          reference;
};

// partial specialization for regular const pointers
template <class T>
struct iterator_traits<const T*>
  typedef random_access_iterator_tag    iterator_category;
  typedef T                             value_type;
  typedef ptrdiff_t                     difference_type;
  typedef const T*                      pointer;
  typedef const T&                      reference;
};
```

所謂 iterator category，是指與「該 iterator 隸屬之 concepts 中，最明確者」所對應之 tag type。例如 int* 既是 Random Access Iterator 又是 Bidirectional Iterator, 同時也是 Forward Iterator, 也是 Input Iterator，但其 category 是 random_access_iterator_tag。這便是爲什麼上述 traits 針對指標所做的 partial specialization 中要使用 random_access_iterator_tag 之故。

以 class 來定義 tag types，不唯可以促成多載化機制的運作，它帶來的另一個好處是，透過繼承，我們可以去除 ¡單純只做轉呼叫¡¨的函式（trivial forwarding function，見前述的 advance() ForwardIterator 版）。

是的，請考慮下面這樣的物件導向設計。至於它和 tag types 的關係，就留給你好好思索。

```
// 模擬測試 tag types 繼承關係所帶來的影響。
#include <iostream>
using namespace std;

struct B { };
struct D1 : public B { };
struct D2 : public D1 { };

template <class I>
func(I& p, B)
{   cout << "B version" << endl;  }

template <class I>
func(I& p, D2)
{   cout << "D2 version" << endl;  }

int main()
{
  int* p;
```

```
  func(p, B());  // exact match. output: "B version"
  func(p, D1()); //              output: "B version"
  func(p, D2()); // exact match. output: "D2 version"
}
```

●一個完整的範例

以下程式把上述的 iterator_traits 整合起來，並加入幾個足以示範如何運用各種 iterator associated types 的 algorithms。注意，這個技術正是 STL 所使用的技術，所以程式內千萬不能打開 namespace std，否則會和 STL 內的 traits 相衝突。你可以在 C++Builder4 和 GNU C++ 所附的 STL 原始碼中，看到大同小異的寫法。VC6 目前尚未支援 partial specialization，所以其所附之 STL 對此的作法稍有不同。

```
#001 // file : 41.cpp
#002 // VC6 [x], BCB4 [o], G++ [o]  (VC6 does not support partial specification)
#003
#004 #include <iostream>
#005 #include <cstddef>  // for ptrdiff_t
#006 // 注意，不能開放 namespace std; 否則會和 std 的 iterator_trait 衝突
#007
#008 struct input_iterator_tag { };
#009 struct output_iterator_tag { };
#010 struct forward_iterator_tag : public input_iterator_tag { };
#011 struct bidirectional_iterator_tag : public forward_iterator_tag { };
#012 struct random_access_iterator_tag : public bidirectional_iterator_tag { };
#013
#014 template <class Iterator>
#015 struct iterator_traits {
#016   typedef typename Iterator::iterator_category  iterator_category;
#017   typedef typename Iterator::value_type        value_type;
#018   typedef typename Iterator::difference_type    difference_type;
#019   typedef typename Iterator::pointer           pointer;
#020   typedef typename Iterator::reference         reference;
#021 };
#022
#023 // partial specialization for regular pointers
#024 template <class T>
#025 struct iterator_traits<T*> {
#026   typedef random_access_iterator_tag  iterator_category;
#027   typedef T                        value_type;
#028   typedef ptrdiff_t                 difference_type;
#029   typedef T*                       pointer;
#030   typedef T&                       reference;
#031 };
#032
#033 // partial specialization for regular const pointers
#034 template <class T>
#035 struct iterator_traits<const T*> {
```

```
#036  typedef random_access_iterator_tag  iterator_category;
#037  typedef T                        value_type;
#038  typedef ptrdiff_t                 difference_type;
#039  typedef const T*                  pointer;
#040  typedef const T&                  reference;
#041 };
#042
#043
#044 template <class InputIterator>
#045 typename iterator_traits<InputIterator>::value_type
#046 sum_nonempty(InputIterator first, InputIterator last)
#047 {
#048  typename iterator_traits<InputIterator>::value_type  result = *first++;
#049  for (; first != last; ++first)
#050    result += *first;
#051
#052  return result;
#053 }
#054
#055 template< class InputIterator, class T >
#056 typename iterator_traits<InputIterator>::difference_type
#057 count( InputIterator first, InputIterator last, const T& x )
#058 {
#059   typename iterator_traits<InputIterator>::difference_type n = 0;
#060   for ( ;  first != last; ++first)
#061     if (*first == x)
#062       ++n;
#063   return n;
#064 }
#065
#066
#067 template <class InputIterator, class Distance>
#068 void advance(InputIterator& i, Distance n, input_iterator_tag)
#069 {
#070  for ( ; n > 0; --n, ++i );
#071 }
#072  // 以下的 trivial forwarding function 其實可以忽略不寫
#073 template <class ForwardIterator, class Distance>
#074 void advance(ForwardIterator& i, Distance n, forward_iterator_tag)
#075 {
#076  advance(i, n, input_iterator_tag());
#077 }
#078
#079 template <class RandomAccessIterator, class Distance>
#080 void advance(RandomAccessIterator& i, Distance n, random_access_iterator_tag)
#081 {
#082  i += n;
#083 }
#084
```

```
#085 template <class BidiectionalIterator, class Distance>
#086 void advance(BidiectionalIterator& i, Distance n, bidirectional_iterator_tag)
#087 {
#088  if (n >= 0)
#089      for ( ; n > 0; --n, ++i ) { }
#090  else
#091      for ( ; n < 0; ++n, --i ) { }
#092 }
#093
#094 // top level
#095 template <class Iterator, class Distance>
#096 inline void advance(Iterator& i, Distance n)
#097 {
#098  advance(i, n, iterator_traits<Iterator>::iterator_category());
#099 }
#100
#101
#102 void main()
#103 {
#104  int ia[5] = {0, 1, 2, 3, 4};
#105  int total = sum_nonempty(ia, ia+5);
#106  int c = count(ia, ia+5, 2);
#107  std::cout << total << std::endl;  // 10
#108  std::cout << c << std::endl;     // 1
#109
#110  std::cout << *ia << std::endl;   // 0
#111  // advance(ia, 3); // error!
#112  int* pi = &(ia[0]);
#113  advance(pi, 3);
#114  std::cout << *pi << std::endl;   // 3
#115 }
```

●STL container 所提供的 iterators

STL iterators 分爲五大類。每一個 STL containers 都定義有一個 iterator 型別和一個 const_iterator 型別。以下是其宣告型式示範：

```
vector<int>::iterator             viter;
vector<int>::const_iterator       cviter;
list<int>::iterator               liter;
list<int>::const_iterator         cliter;
deque<int>::iterator              diter;
deque<int>::const_iterator        cditer;
map<string, int>::iterator        miter;
map<string, int>::const_iterator  cmiter;
set<int>::iterator                siter;
set<int>::const_iterator          csiter;
```

這些 iterators 隸屬哪一種 iterator concepts 呢？[Austern99] 和 [Josutiss99] 都有明確的說明。當我們

要使用某個 algorithms 時(所有 STL algorithms 的最前面兩個參數都是 iterators，標記出一個範圍)，只需切記，你的 container iterators 的層級必須「優於」algorithms 所需層級。例如 remove() 需要 ForwardIterator，而 vector 和 deque 提供的是 RandomAccessIterator，list 提供的是 BidirectionalIterator，set 和 map 提供的 iterators 是 ForwardIterator，所以它們都可以和 remove() 搭配。

●分析 STL istream_iterator 並訂製一個 line_iterator

STL 提供有所謂的 stream iterator：istream_iterator 用於輸入，ostream_iterator 用於輸出。讓我們好好分析 istream_iterator 的實作技巧，以便爲自己訂製 iterator 鋪路。

以下是 SGI STL 的 istream_iterator 實作內容：

```
#0001 // VC6[x]  CB4[x]  G++[o]
#0002 // 執行：c:\> type stl-2-3.dat | stl-2-3.exe
#0003
#0004 #include <iostream>
#0005 #include <cstddef>   // for ptrdiff_t
#0006 #include <algorithm> // for for_each()
#0007 #include <iterator>  // for inserter
#0008 #include <vector>
#0009 using namespace std;
#0010
#0011 // 以下完全模仿 g++ SGI stl_iterator.h 中的 istream_iterator，
#0012 // 只是將名稱改加 my，並將其中的一些 #ifdef 拿掉
#0013
#0014 template <class T, class Distance = ptrdiff_t>
#0015 class myistream_iterator {
#0016   friend bool
#0017   operator== <> (const myistream_iterator<T, Distance>& x,
#0018                  const myistream_iterator<T, Distance>& y); //ref #0051
#0019 protected:
#0020   istream* stream;
#0021   T value;
#0022   bool end_marker;
#0023   void read() {
#0024     end_marker = (*stream) ? true : false;
#0025     if (end_marker) *stream >> value;
#0026     end_marker = (*stream) ? true : false;
#0027   }
#0028 public:
#0029   typedef input_iterator_tag iterator_category;
#0030   typedef T                  value_type;
#0031   typedef Distance           difference_type;
#0032   typedef const T*           pointer;
#0033   typedef const T&           reference;
```

```
#0034
#0035  myistream_iterator() : stream(&cin), end_marker(false) {}
#0036  myistream_iterator(istream& s) : stream(&s) { read(); }  //如果拿掉 read()，執行結果會出錯
#0037  reference operator*() const { return value; }
#0038  pointer operator->() const { return &(operator*()); }
#0039  myistream_iterator<T, Distance>& operator++() {
#0040    read();
#0041    return *this;
#0042  }
#0043  myistream_iterator<T, Distance> operator++(int)  {
#0044    myistream_iterator<T, Distance> tmp = *this;
#0045    read();
#0046    return tmp;
#0047  }
#0048 };
#0049
#0050 template<class T, class Distance>
#0051 inline bool operator==(const myistream_iterator<class T, class Distance> & x,
#0052                        const myistream_iterator<class T, class Distance> & y) {
#0053  return x.stream == y.stream && x.end_marker == y.end_marker ||
#0054         x.end_marker == false && y.end_marker == false;
#0055 }
#0056
#0057
#0058 template <class T>
#0059 void print_elements(T elem) { cout << elem << " "; }
#0060
#0061 int main()
#0062 {
#0063  vector<int> iv;
#0064
#0065  myistream_iterator< int > inputi(cin);
#0066  myistream_iterator< int > eos;
#0067  copy(inputi, eos, inserter(iv, iv.begin()));
#0068
#0069  void (*pfi)(int) = print_elements;
#0070  for_each(iv.begin(), iv.end(), pfi);  // 列印出來
#0071 }
```

當我們要訂製一個 iterator 時，istream_iterator 的作法可以提供一個有效的參考。假設現在我要設計一個 myistream_line_iterator，它讀取資料的方式不像 istream_iterator 那般以 token（語彙單元）爲單位，而是以一行文字爲單位。也因此，它的 value type 固定爲 string，而不再是個 template 參數。下面是 myistream_line_iterator 的實作碼與測試範例：

```
#0001 // VC6[x]  CB4[x]  G++[o]
#0002 // 執行：c:\> type stl-2-4.dat | stl-2-4.exe
#0003
#0004 #include <iostream>
```

```
#0005 #include <cstddef>  // for ptrdiff_t
#0006 #include <algorithm> // for for_each()
#0007 #include <iterator>  // for inserter
#0008 #include <vector>
#0009 using namespace std;
#0010
#0011 // 以下完全模仿 g++ SGI stl_iterator.h 中的 istream_iterator，
#0012 // 只是將名稱改掉，將其中的一些 #ifdef 拿掉，並修改 read()
#0013 // 及 value type, pointer type, reference type，使之
#0014 // 不再成爲一個 class template。
#0015
#0016 #include <string>
#0017
#0018 class myistream_line_iterator {
#0019   friend bool
#0020   operator== (const myistream_line_iterator& x,
#0021               const myistream_line_iterator& y);
#0022 protected:
#0023   istream* stream;
#0024   string value;
#0025   bool end_marker;
#0026   void read() {
#0027     end_marker = (*stream) ? true : false;
#0028     if (end_marker) getline(*stream, value);
#0029     end_marker = (*stream) ? true : false;
#0030   }
#0031 public:
#0032   typedef input_iterator_tag iterator_category;
#0033   typedef string             value_type;
#0034   typedef ptrdiff_t          difference_type;
#0035   typedef const string*      pointer;
#0036   typedef const string&      reference;
#0037
#0038   myistream_line_iterator() : stream(&cin), end_marker(false) {}
#0039   myistream_line_iterator(istream& s) : stream(&s) { read(); }
#0040   reference operator*() const { return value; }
#0041   pointer operator->() const { return &(operator*()); }
#0042   myistream_line_iterator& operator++() {
#0043     read();
#0044     return *this;
#0045   }
#0046   myistream_line_iterator operator++(int)  {
#0047     myistream_line_iterator tmp = *this;
#0048     read();
#0049     return tmp;
#0050   }
#0051 };
#0052
#0053 inline bool operator==(const myistream_line_iterator& x,
```

```
#0054                          const myistream_line_iterator& y) {
#0055   return x.stream == y.stream && x.end_marker == y.end_marker ||
#0056          x.end_marker == false && y.end_marker == false;
#0057 }
#0058
#0059
#0060 template <class T>
#0061 void print_elements(T elem) { cout << elem << " "; }
#0062
#0063 int main()
#0064 {
#0065   vector<string> sv;
#0066   cout << sv.size() << endl;  // 0
#0067
#0068   myistream_line_iterator inputli(cin);
#0069   myistream_line_iterator eosli;
#0070   copy(inputli, eosli, inserter(sv, sv.begin()));
#0071
#0072   cout << sv.size() << endl;  // [some value!]
#0073
#0074   for(int i=0; i< sv.size(); ++i)
#0075     cout << sv[i] << endl;
#0076 }
```

這一期我們歷經了份量很重的技術洗禮。徹底瞭解了所謂的 traits 技術。這項技術無所不在地存在於 STL 的各個角落。下一期我們可以喘口氣，看看 containers 的應用 -- 那將是很令人愉快的程式經驗。

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
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。