侯捷觀點

芝麻開門
從 Iterator 談起

北京《程序員》2001.03
台北《Run!PC》2001.03

作者簡介：侯捷，臺灣電腦技術作家，著譯評兼擅。常著文章自娛，頗示己志。
侯捷網站：http://www.jjhou.com
http://jjhou.readme.com.tw
北京鏡站：http://expert.csdn.net/jjhou

- 讀者基礎：熟悉 C++ template.
- 本文適用：任何作業平台，任何 C++ 編譯器（注意，文中凡涉及 template partial specialization 者，無法通過 VC6，因 VC6 未支援這項語言性質）
- 本文源碼可至侯捷網站下載
- 侯捷網站有五篇 STL 相關文章可供閱讀（PDF 格式）。如果您對於 STL 和泛型思維尚無頭緒，不妨先閱讀該系列第一篇文章。本文以該系列第二篇文章爲基礎，重新整理並加大幅擴充。

連續兩個月我爲各位介紹了 C++/OOP/Genericity/STL 方面的經典好書。這個月我想深入帶你看看一些相關技術。我挑選 STL 的關鍵性技術 iterator 做爲本月主題，掌握它，就像掌握了阿里巴巴的口訣，得以進入 STL 的源碼寶庫，而透過 STL 源碼，我又得以解說數個重要的設計樣式（Design Patterns）。

不論是泛型思維（generic paradigm）或 STL（Standard Template Library）實際運用，iterators 都扮演重要角色。STL 的中心思想，在於將資料容器（containers）和演算法（algorithms）分開設計，彼此獨立，最後再以一帖膠著劑將它們撮合在一起。容器和演算法的泛型化，從技術角度來看並不困難，C++ 的 class templates 和 function templates 可分別達成目標。如何設計出兩者之間的良好膠著劑，才是大難題。

以下是容器、演算法、迭代器（iterator，扮演黏膠角色）的合作情形。只要給予不同的迭代器，演算法 find() 便可以對不同的容器工作：

```
// test1.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
   const int arraySize = 7;
   int ia[arraySize] = { 0,1,2,3,4,5,6 };

   vector<int> ivect(ia, ia + arraySize);
   list<int>  ilist(ia, ia + arraySize);

   vector<int>::iterator it1 = find(ivect.begin(), ivect.end(), 4);
   if (it1 == ivect.end())
      cout << "4 not found." << endl;
   else
      cout << "4 found. " << *it1 << endl;

   list<int>::iterator it2 = find(ilist.begin(), ilist.end(), 6);
   if (it2 == ilist.end())
      cout << "6 not found." << endl;
   else
      cout << "6 found. " << *it2 << endl;
}
```

從這個例子看來，迭代器似乎是依附在容器之下。是真的嗎？有沒有獨立的迭代器呢？我們又該如何自行設計特殊的迭代器呢？

漸次泛型化實例剖析

首先讓我們看看演算法如何泛型化。**[Meyers96]** 條款 35 有一個不錯的例子，我以該例爲起點，帶你展開這次旅程。

假設我們要寫一個 find() 函式，在陣列中尋找特定值。面對整數陣列，我們的直覺反應是這麼做：

```
// 節錄自 find1.cpp
int* find(int* arrayHead, int arraySize, int value)
```

```
{
  int i;
  for (i=0; i<arraySize; i++)
    if (arrayHead[i] == value)
       break;

  return &(arrayHead[i]);
}
```

這個函式在陣列範圍內搜尋 value，並傳回一個指標，指向它所找到的第一個目標；如果沒有找到，就傳回最後一個元素的下一位置（位址）。

「最後元素的下一位置」我們稱之爲 end。傳回 end 以表示「搜尋無結果」，似乎是個可笑的作法。爲什麼不傳回 null？因爲，一如稍後即將見到，end 指標可以對其他種類的容器帶來泛型效果，這是 null 做不到的。從小我們就被教導，使用陣列時千萬不要超越其範圍，但事實上一個指向陣列元素的指標，不但可以合法地指向陣列內的任何位置，也可以指向陣列尾端以外的任何位置。只不過當指標指向陣列尾端以外的位置時，它只能用來和其他陣列指標相比較，不能提領（*dereference*）其值。現在，你可以這樣使用 find() 函式：

```
// 節錄自 find1.cpp
const int arraySize = 7;
int ia[arraySize] = { 0,1,2,3,4,5,6 };
int* end = ia + arraySize;  // 最後元素的下一位置

int* ip = find(ia, sizeof(ia)/sizeof(int), 4);
if (ip == end)               // 兩個陣列指標相比較
   cout << "4 not found" << endl;
else
   cout << "4 found. " << *ip << endl;
```

find() 的這種作法曝露了容器的太多實作細節（例如 arraySize），也因此太依附特定容器。爲了讓 find() 適用於所有類型的容器，其動作應該更一般化些。讓 find() 接受兩個指標做爲參數，就是很好的作法：

```
// 節錄自 find2.cpp
int* find(int* begin, int* end, int value)
{
  while (begin != end && *begin != value)
    ++begin;

  return begin;
}
```

這個函式在「前閉後開」區間 [begin, end) 內（不含 end，end 指向陣列最後元素的下一位置）搜尋 value，並傳回一個指標，指向它所找到的第一個目標；如果沒有找到，就傳回 end。現在，你可以這樣使用 find() 函式：

```
// 節錄自 find2.cpp
const int arraySize = 7;
int ia[arraySize] = { 0,1,2,3,4,5,6 };
int* end = ia + arraySize;

int* ip = find(ia, end, 4);
if (ip == end)
   cout << "4 not found" << endl;
else
   cout << "4 found. " << *ip << endl;
```

find() 函式也可以很方便地用來搜尋陣列的子範圍：

```
int* ip = find(ia+2, ia+5, 3);
if (ip == end)
   cout << "3 not found" << endl;
else
   cout << "3 found. " << *ip << endl;
```

由於 find() 函式並無任何設計是針對整數陣列而發，所以我們可將它改成一個 template：

```
template<typename T>  // 關鍵字 typename 也可改爲關鍵字 class
T* find(T* begin, T* end, const T& value)
{
   while (begin != end && *begin != value)
     ++begin;

   return begin;
}
```

請注意數值的傳遞由 pass-by-value 改爲 pass-by-reference-to-const，因爲如今所傳遞的 value，其型別可爲任意；當物件過大，傳遞成本便會提昇，這是我們不願見到的。pass-by-reference 可完全避免這些成本（見 **[Meyers98]** 條款 22）。

這樣的 find() 很好，幾乎適用於任何容器 -- 只要該容器允許指標指入，而這些指標又都支援以下四種在 find() 中出現的操作：

1. inequality（判斷不相等）運算子

2. dereferencelm（提領，取值）運算子
3. prefix increment（前置式遞增）運算子
4. copy（複製）行爲（以便產生函式的傳回值）

C++ 的一個極大優點是，幾乎所有東西都可以改寫爲程序員自定的型式或行爲。是的，上述這些運算子或操作行爲都可以被多載化（*overloaded*），那又何必將 `find` 限制爲只能使用指標呢？何不讓支援以上四種行爲的物件都可以被 `find()` 使用呢？如此一來 `find()` 函式便可以從原生（*native*）指標的思想框框中跳脫出來。如果我們以一個原生指標指向某個串列（list），對此指標做 `++` 動作並不能使它指向下一個串列節點。但如果我們設計一個 class，擁有原生指標的行爲，並使其 `++` 動作指向下一個串列節點，那麼 `find()` 就可以施行於串列容器身上了。

這便是迭代器（iterator）的觀念。迭代器是一種**行爲類似指標**的物件，換句話說是一種 smart pointers（參考 **[Meyers96]** 條款 22）。現在我將 `find()` 函式內的指標以 迭代器取代，重新寫過：

```
// 節錄自 find3.h
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value)
{
  while (begin != end && *begin != value)
    ++begin;

  return begin;
}
```

這便是一個完全泛型化的 `find()` 函式。你可以在任何 C++ 標準程式庫的 `<algorithm>` 表頭檔中看到它，長像幾乎一模一樣。

迭代器（iterator）就是一種 smart pointer

`find()` 已經泛型化了，但上述可運用於串列（list）容器身上的迭代器，又該怎麼完成？

由於迭代器是一種行爲類似指標的物件，而指標行爲最常見也最重要的便是內容提領（*dereference*）和成員取用（member access），因此最重要的設計就是對 `operator*` 和 `operator->` 進行多載化工程（*overloading*）。C++ 標準程式庫有一個 `auto_ptr` 可

供我們參考。任何一本詳盡的 C++ 語法書籍都應該談到 auto_ptr（如果沒有，扔了它），這是一個用來包裝原生指標的物件，聲名狼藉的記憶體漏洞（memory leak）問題可藉此獲得解決。auto_ptr 用法如下，和原生指標一模一樣：

```
// 節錄自 autoptr.cpp
void func()
{
  auto_ptr<string> ps(new string("jjhou"));

  cout << *ps << endl;             // jjhou
  cout << ps->size() << endl;      // 5
  // ...
  // 離開前不需 delete, auto_ptr 會自動釋放記憶體
}
```

函式第一行的意思是，以 new 算式動態配置一個初值爲 "jjhou" 的 string 物件，並將所得結果（一個原生指標）做爲 auto_ptr<string> 物件的初值。注意，auto_ptr 角括號內放的是「原生指標所指物件」的型別，而不是原生指標的型別。

auto_ptr 的源碼在表頭檔 <memory> 中，根據它，我模擬了一份陽春版，可具體說明 auto_ptr 的行爲與能力：

```
// 節錄自 autoptr.cpp
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0): pointee(p) {}
  template<class U>
  auto_ptr(auto_ptr<U>& rhs): pointee(rhs.release()) {}
  ~auto_ptr() { delete pointee; }

  template<class U>
  auto_ptr<T>& operator=(auto_ptr<U>& rhs) {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }
  T& operator*() const { return *pointee; }
  T* operator->() const { return pointee; }
  T* get() const { return pointee; }
  // ...
private:
    T *pointee;
};
```

其中有些高階技術如關鍵字 explicit 和 member template，並不是今天的重點。我們

的重點放在其中的運算子多載化工程。

有了模倣對象，現在我來爲串列設計一個迭代器。假設串列及其節點的結構如下（可參考 **[lippman98]** 5.11 節）：

```
// 節錄自 mylist.h
template <typename T>
class List
{
  void insert_front(T value);
  void insert_end(T value);
  void display(std::ostream &os = std::cout) const;
  // ...
private:
  ListItem<T>* _end;
  ListItem<T>* _front;
  long _size;
};

template <typename T>
class ListItem
{
public:
  T value() const { return _value; }
  ListItem* next() const { return _next; }
  ...
private:
  T _value;
  ListItem* _next;  // 單向串列（single linked list）
};
```

如何將這個 `List` 套用到先前完成的 `find()` 呢？我們需要爲它設計一個**行爲類似指標**的外包裝，也就是一個迭代器。當我們提領此一迭代器，傳回的是個 `ListItem` 物件；當我們累加該迭代器，它會指向下一個 `ListItem` 物件。爲了讓迭代器適用於任何型別的節點，而不只限於 `ListItem`，我們可以將它設計爲一個 class template：

```
// 節錄自 myiter.cpp
template <class Item>     // Item 可以是單向串列節點或雙向串列節點。
struct ListIter           // 此處這個迭代器特定只爲串列服務，因爲其
{                         // 獨特的 operator++ 之故。
  Item* ptr;     // keep a reference to Container

  ListIter(Item* p = 0)  // default ctor
    : ptr(p) { }

  // 不必實作 copy ctor，因爲編譯器提供的預設行爲已足夠。
```

```
    // 不必實作 operator=，因爲編譯器提供的預設行爲已足夠。

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

    // pre-increment operator
    ListIter& operator++()
      { ptr =  ptr->next(); return *this; }

    // post-increment operator
    ListIter  operator++(int)
      { ListIter tmp = *this; ++*this; return tmp; }

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

其中兩個 operator++ 遵循標準作法，參見 **[Meyers96]** 條款 6。

現在，我們可以這樣子將 list 和 find() 藉由 ListIter 搭配黏著起來：

```
// 節錄自 myiter.cpp
void main()
{
  List<int> mylist;

  for(int i=0; i<5; ++i) {
     mylist.insert_front(i);
     mylist.insert_end(i+2);
  }
  mylist.display();              // 10 ( 4 3 2 1 0 2 3 4 5 6 )

  ListIter<ListItem<int> > begin(mylist.front());
  ListIter<ListItem<int> > end;  // default 0, null
  ListIter<ListItem<int> > iter;

  iter = find(begin, end, 3);
  if (iter == end)
     cout << "not found" << endl;
  else
     cout << "found. " << iter->value() << endl;

  iter = find(begin, end, 7);
  if (iter == end)
     cout << "not found" << endl;
  else
     cout << "found. " << iter->value() << endl;
```

```
}
```

注意，由於 find() 函式內是以 *iter != value 來檢查元素值是否吻合，本例之中 value 的型別是 int，而 iter 的型別是 ListItem<int>，兩者之間並無可供使用的 operator!=，所以我必須另外寫一個全域的 operator!= 多載函式，並以 int 和 ListItem<int> 做爲兩個參數型別：

```
// 節錄自 myiter.cpp
template <typename T>
bool operator!=(const ListItem<T>& item, T n)
{  return item.value() != n; }
```

從以上實作可以看出，爲了完成一個針對 List 而設計的迭代器，我們曝露了太多 List 實作細節：在 main() 之中爲了製作 begin 和 end 兩個迭代器，我們曝露了 ListItem；在 ListIter class 之中爲了達成 operator++ 的目的，我們曝露了 ListItem 的操作函式 next()。如果不是爲了迭代器， ListItem 原本應該完全隱藏起來不曝光的。換句話說，根據以上思考，要設計出 ListIter，首先必須對 List 的實作細節有非常豐富的瞭解。既然這無可避免，乾脆就把迭代器的開發工作交給 List 的設計者好了，如此一來所有實作細節反而得以封裝起來不被使用者看到。這正是爲什麼每一個 STL 容器都提供自己的迭代器的緣故。

迭代器相應型別（associated types）— traits 編程技法

上述的 ListIter 提供了一個迭代器雛形。如果將思考拉得更高更遠一些，我們便會發現，演算法之中運用迭代器時，很可能會用到其**相應型別**（associated type）。什麼是相應型別？迭代器所指之物的型別便是其一。如果演算法中有必要宣告一個變數，以「迭代器所指之物的型別」爲型別，如何是好？畢竟 C++ 只支援 sizeof()，並未支援 typeof()！即便動用 RTTI 性質中的 typeid()，獲得的也只是型別名稱，不能拿來做變數宣告之用。

解決辦法倒有一個：利用 function template 的引數推導（argument deducation）機制。例如：

```
template <class I, class T>
void func_impl(I iter, T t)
{
```

```
  T tmp;    // 這裡解決了問題。T 就是迭代器所指之物的型別，本例爲 int
  // ... 這裡做原本 func() 應該做的全部工作
};

template <class I>
inline
void func(I iter)
{
  func_impl(iter, *iter);      // func 的工作全部移往 func_impl
}

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

我們以 func() 爲對外介面，實際動作全部置於 func_impl() 中。由於 func_impl() 是一個 function template，一旦被呼叫，編譯器會自動進行 template 引數推導。於是導出上例的型別 T，順利解決了問題。

迭代器相應型別之一：*value type*

迭代器所指之物的型別，我們稱爲該迭代器的 *value type*。上述的型別推導技巧雖然可用，卻非全面可用：萬一 *value type* 必須用於函式的傳回值，就沒輒了，因爲函式的「template 引數推導機制」推而導之的只是引數，不含回返型別。

我們需要其他方法。巢狀型別宣告似乎是個好主意，像這樣：

```
// 節錄自 traits.cpp
template <class T>
struct MyIter
  typedef T value_type;  // nested type（巢狀型別宣告）
  T* ptr;
  MyIter(T* p=0) : ptr(p) { }
  T& operator*() const { return *ptr; }
  // ...
};

template <class Iterator>
typename Iterator::value_type  // 這一整行是 func 的回返型別
func(Iterator ite)
{ return *ite; }

// ...
```

```
MyIter<int> ite(new int(8));
cout << func(ite);    // 8
```

注意，func() 的回返型別必須加上關鍵字 typename，因爲 T 是一個 template 參數，在它被具現化之前，編譯器對 T 一無所悉，所以編譯器不知道 MyIter<T>::value_type 代表的是一個型別或是一個 member function 或是一個 data member。關鍵字 typename 告訴編譯器說這是一個型別，這麼一來才能順利通過編譯。

這看起來不錯。但是有個隱微的陷阱。並不是所有迭代器都是 class type。原生指標就不是！如果不是 class type，就無法定義巢狀型別。但 STL（以及整個泛型思維）絕對必須接受原生指標做爲一種迭代器，所以上面這樣還不夠。有沒有辦法可以讓上述的一般化概念針對特定情況（例如針對原生指標）做特殊化處理呢？

是的，template partial specialization 可以做到。

任何完整的 C++ 語法書籍都應該對 template partial specialization 有所說明（如果沒有，扔了它）。大致的意義是：如果 class template 擁有一個以上的 template 參數，我們可以針對其中某個（或數個，但非全部）template 參數進行特化工作。換句話說我們可以供應一個特別版本，符合泛化條件，但其中某些 template 參數已經由實際型別或數值取代。

假設有一個 class template 如下：

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

partial specialization 的意義容易誤導我們以爲，所謂「局部特化版本」一定是對 template 參數 U 或 V 或 T（或其任意組合）指定某個引數值。事實不然，**[Austern99]** 對於 partial specialization 的意義說得十分得體：『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，面對以下這麼一個 class template：

```
template<typename T>
class C { ... };      // 這個泛型版本允許（接受）T 爲任何型別
```

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

```
template<typename T>
class C<T*> { ... };  // 這個特殊版本僅適用於「T 爲原生指標」的情況
```

有了這項利器，我們可以解決前述「巢狀型別」未能解決的問題。先前的問題是，原生指標並非 class，因此無法爲它們定義巢狀型別。現在，我們可以針對「迭代器的 template 引數爲指標」者，設計特化版的迭代器。

醒醒，提高警覺，我們進入關鍵地帶了。現在讓我設計一個 class template 如下，專門用來「抽取」迭代器的特性，而 *value type* 正是迭代器的特性之一：

```
template <class I>
struct iterator_traits {   // traits 意爲「特性」
  typedef typename I::value_type   value_type;
};
```

這個所謂 **traits** 的意義是，如果 `I` 定義有自己的 *value type*，那麼透過這個 **traits** 的作用，流出來的 `value_type` 就是 `I::value_type`。換句話說先前的那個 `func()` 可以改寫成這樣：

```
template <class I>
typename iterator_traits<I>::value_type  // 這一整行是函式回返型別
func(I ite)
{ return *ite; }
```

但這麼做除了多一層間接性，又帶來什麼好處呢？好處是 **traits** 可以擁有特化版本。現在我爲 `iterator_traites` 設計一個 partial specializations：

```
template <class T>
struct iterator_traits<T*> {  // 特化版：當 iterator 是一個原生指標
  typedef T value_type;
};
```

於是 `int*` 亦可透過 **traits** 取其 *value type*。但是注意，下面這個式子得到什麼結果：

```
iterator_traits<const int*>::value_type
```

獲得的是 `const int` 而非 `int`。這是我們所希望的嗎？我們希望利用這種機制來宣告一個暫時變數，使其型別與迭代器的 *value type* 相同，而現在，宣告一個無法賦值（因 `const` 之故）的暫時變數，沒什麼用。因此如果迭代器是個 pointer-to-`const`，我們應該設法令其 *value type* 爲一個 non-`const` 型別。沒問題，只要另外設計一個特化版本，就能解決這個問題：

```
template <class T>
```

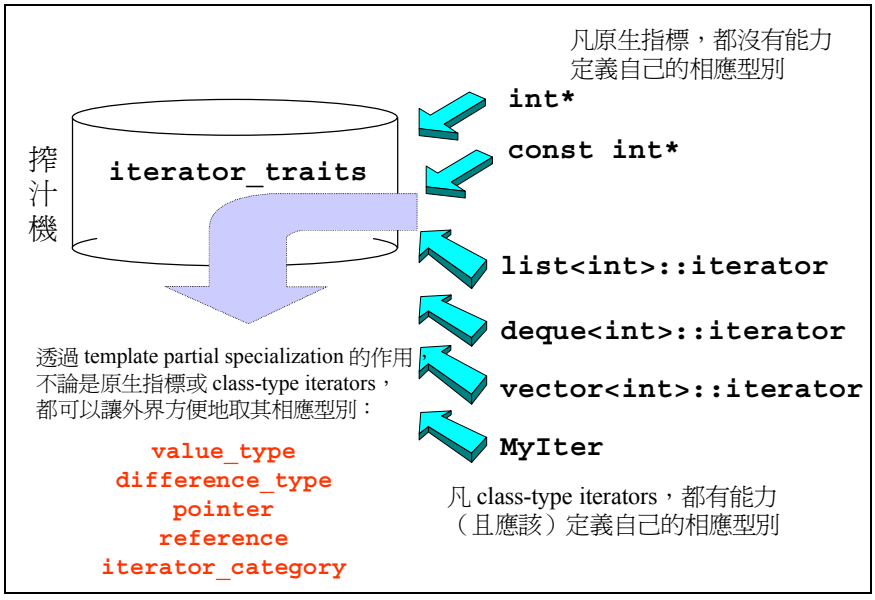
```
struct iterator_traits<const T*> { //  特化版。當迭代器是一個 pointer-to-const
  typedef T value_type;              // 萃取出來的型別爲 T 而非 const T
};
```

現在，不論面對的是 `MyIter` 或 `int*` 或 `const int*`，都可以透過 **traits** 取出正確的 *value type*：

```
MyIter<int> ite(new int(8));
cout << func(ite);

int* pi = new int(5);
cout << func(pi);

const int* pci = new int(9);
cout << func(pci);
```

圖一說明 **traits** 所扮演的榨汁機角色，榨取各個迭代器的特性（相應型別）。

圖一/ **traits** 就像一台榨汁機，榨取各個迭代器的特性（相應型別）。



迭代器相應型別之二：*difference type*

difference type 用來表示兩個迭代器之間的距離，也因此，它可以用來表示一個容器的最大容量，因爲對於連續空間的容器而言，頭尾之間的距離就是其最大容量。如果一

個泛型演算法有著計數功能，例如 STL 的 `count()`，其傳回值就必須使用迭代器的 *difference type*：

```
template <class I, class T>
typename iterator_traits<I>::difference_type // 這一整行是函式回返型別
count(I first, I last, const T& value) {
  typename iterator_traits<I>::difference_type n = 0;
  for ( ; first != last; ++first)
    if (*first == value)
      ++n;
  return n;
}
```

我們仍然可使用前述的 **traits** 技術解決這個問題。至於原生指標，我們可以設計對應的 partial specializations，並指定 C++ 內建的 `ptrdiff_t` 做爲其 *difference type*：

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

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

// partial specialization for native pointer-to-const
template <class T>
struct iterator_traits<const T*> {
  typedef T          value_type;
  typedef ptrdiff_t  difference_type;
};
```

C++ 內建的 `ptrdiff_t` 定義於 `<cstddef>` 表頭檔中：

```
typedef int ptrdiff_t;
```

現在，任何時候當我們需要任何迭代器的 *difference type* 時，便可以這麼寫：

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

迭代器相應型別之三：*reference type*

從「迭代器所指之物的內容是否允許改變」的角度觀之，迭代器分爲兩種：不允許改變「所指物之內容」者，稱爲 constant iterators，例如 `const int* pic`；允許改變「所指物之內容」者，稱爲 mutable iterators，例如 `int* pi`。當我們對一個 mutable iterators 做提領動作時，獲得的不應該是個右值，應該是個左值，因爲右值不允許賦值動作（assignment），左值才允許：

```
int* pi = new int(5);
const int* pci = new int(9);
*pi = 7;   // 對 mutable iterator 做提領動作時，獲得的應該是個左值，允許賦值。
*pci = 1;  // 這個動作不允許，因爲 pci 是個 constant iterator，
           // 其提領值本身是個右值，不允許被賦值。
```

在 C++ 中，函式如果要傳回左值，都是以 by reference 的方式進行，所以當 `p` 是個 mutable iterators 時，如果其 *value type* 是 `T`，那麼 `*p` 的型別不應該是 `T`，應該是 `T&`。將此道理擴充，如果 `p` 是一個 constant iterators，其 *value type* 是 `T`，那麼 `*p` 的型別不應該是 `const T`，而應該是 `const T&`。這裡所討論的 `*p` 的型別，即所謂的 *reference type*。

迭代器相應型別之四：*pointer type*

pointers 和 references 在 C++ 中有非常密切的關連。如果「傳回一個左值，代表 `p` 所指之物」是可能的，那麼「傳回一個左值，代表 `p` 所指之物的位址」也一定可以。也就是說我們能夠傳回一個 pointer，指向迭代器所指之物。

這些相應型別已在先前的 `ListIter` class 中出現過：

```
Item& operator*() const { return *ptr; }
Item* operator->() const { return ptr; }
```

`Item&` 便是 `ListIter` 的 *reference type* 而 `Item*` 便是其 *pointer type*。

現在我們把兩個新的相應型別加入 **traits** 內：

```
template <class I>
struct iterator_traits {
  typedef typename I::value_type        value_type;
  typedef typename I::difference_type   difference_type;
  typedef typename I::pointer           pointer;
```

```
  typedef typename I::reference         reference;
};

// partial specialization for native pointer
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 native pointer-to-const
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;
};
```

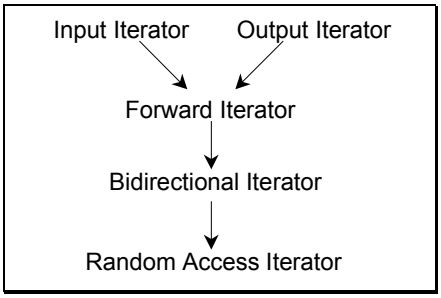
迭代器的分類

最後一個（第五個）迭代器相應型別會引發規模比較龐大的寫碼工程。在那之前，我必須先討論迭代器的分類。根據迭代器的移動特性與施行動作，它被分爲五類：

- *Input Iterator*：不允許外界改變這種迭代器所指物件。唯讀（read only）。
- *Output Iterator*：唯寫（write only）。
- *Forward Iterator*：允許「寫入型」演算法（例如 `replace()`）使用這種迭代器所形成的同一個範圍做讀寫動作。
- *Bidirectional Iterator*：可雙向移動。某些演算法需要逆向走訪某個迭代器範圍，例如逆向拷貝某範圍內的元素，就可以使用 *Bidirectional Iterator*s。
- *Random Access Iterator*：前四種迭代器都只供應一部份指標算術能力（前三種支援 `operator++`，第四種另加上 `operator--`），第五種則涵蓋其餘所有指標算術能力，包括 `p+n, p-n, p[n], p1-p2, p1<p2`。

這些迭代器的分類與從屬關係，可以圖二表示。直線與箭頭代表的並非 C++ 的繼承關係，而是所謂 concept（概念）與 refinement（強化）的關係（見 **[Austern98]**）。

圖二/ 迭代器的分類與從屬關係



迭代器相應型別之 II ：*iterator tags*

設計演算法時，如果可能，我們儘量針對圖一某種迭代器提供一個明確定義，而針對更強化的某種迭代器提供另一種定義，這樣才能在不同情況下提供最大效率。假設我們有個演算法可接受 *Forward Iterator*，你以 *Random Access Iterator* 餵給它，它當然也會接受，因爲一個 *Random Access Iterator* 必然是一個 *Forward Iterator*（見圖一）。但是，可用並不代表最佳。

以 `advance()` 爲例（這是許多演算法內部常用的一個函式）。此函式有兩個參數，迭代器 `p` 和數值 `n`；函式內部將 `p` 累進 `n` 次（前進 `n` 距離）。下面有三份定義，一是針對 *Input Iterator*，一是針對 *Bidirectional Iterator*，一是針對 *Random Access Iterator*。`advance()` 並無針對 *ForwardIterator* 設計的版本，因爲這和針對 *InputIterator* 者完全一致。

```
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()，應該使用哪一份定義呢？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，下面是一種作法：

```
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);
}
```

但是像這樣在執行時期才決定使用哪一個版本，會影響程式效率。最好能夠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。

前述三個 advance_() 版本都有兩個函式參數，型別都未定（因爲都是 template 參數）。我們必須加上一個型別已確定的函式參數，使函式多載化機制能夠有效運作起來。

設計考量是這樣的：如果能夠在 **traits** 內再增加一個新的迭代器相應型別，使其得以萃取出迭代器的種類，我們便可以利用這個相應型別做爲 advanced() 的第三參數。這個相應型別一定必須是個 class type，而不能只是個號碼之類的東西，因爲編譯器需得仰賴它（一個真正的型別）來進行多載化決議程序（overloaded resolution）。下面定義出代表五種迭代器的五個 classes：

```
// 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 { };
```

由於這些 classes 只做爲標示用，所以不需要任何成員。至於爲什麼運用繼承機制，稍後再解釋。接下來重新設計 advance()，令它們全部同名，並加上第三參數，形成多載化：

```
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());  // 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() 的最後參數都只宣告其型別，並沒有指定參數名稱，因爲函式之中根本不使用該參數。當然硬要加上參數名稱也是可以的，畫蛇添足而已。

最後我們還需寫一個上層控制函式，呼叫上述各個多載化的 advance()。此一上層函式只需兩個參數，函式內自行加上第三引數(迭代器的相應型別，五個 tag types 之一)，然後呼叫上述的 advance()。因此，上層函式必須有能力從它所獲得的迭代器中推導其 tag type，那自然是交給 **traits** 機制啦：

```
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()` 產生一個暫時物件，其型別隸屬前述五個 tag types 之一。這種語法就像 `int()` 一樣，產生一個 `int` 暫時物件。根據這個暫時物件的型別（五個 tag types 之一），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，**traits** 必須再增加一個相應型別：

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

// partial specialization for native pointer
template <class T>
struct iterator_traits<T*> {
  // 注意，native pointer 是一種 Random Access Iterator
  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 native-pointer-to-const
template <class T>
struct iterator_traits<const T*>
  // 注意，native pointer-to-const 是一種 Random Access Iterator
  typedef random_access_iterator_tag    iterator_category;
  typedef T                             value_type;
  typedef ptrdiff_t                     difference_type;
  typedef const T*                      pointer;
  typedef const T&                      reference;
};
```

迭代器之分類歸屬，應該落在「該迭代器隸屬之種類中最強化的那個」。例如 `int*` 既

是 *Random Access Iterator* 又是 *Bidirectional Iterator*，同時也是 *Forward Iterator*，而且也是 *Input Iterator*，但其分類歸屬爲 *random_access_iterator_tag*。

以 class 來定義迭代器的各種分類標籤，不唯可以促成多載化機制的成功運作（編譯器得以正確執行多載化決議程序，overloaded resolution），另一個好處是，透過繼承，我們可以不必再寫「單純只做轉呼叫」的函式（例如前述的 `advance()` *ForwardIterator* 版）。爲什麼能夠如此？考慮下面這個小例子：

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

struct B { };                   // B  可比擬爲 InputIterator
struct D1 : public B { };       // D1 可比擬爲 ForwardIterator
struct D2 : public D1 { };      // D2 可比擬爲 BidirectionalIterator

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());   // 參數與引數完全吻合。輸出: "B version"
  func(p, D1());  // 參數與引數未能完全吻合。因繼承而自動轉呼叫。輸出:"B version"
  func(p, D2());  // 參數與引數完全吻合。輸出: "D2 version"
}
```

完整範例

截至目前，本文從演算法的泛型化，到迭代器的基本形式，到迭代器與容器的密切相依關係，到迭代器必要的五種相應型別，以及爲兼顧原生指標而設計出來的 **traits** 編程技巧，總結心得是：設計適當的相應型別，是迭代器的責任，設計適當的迭代器，則是容器的責任。唯容器本身，才知道該設計出怎樣的迭代器來走訪自己，並配合容器的特性完成某些獨特行爲（例如 `operator++`）。至於演算法，可以完全獨立於容器和迭代器之外自行發展。

我將截至目前所討論的 **traits** 編程技術，寫成一個完整範例 `traits2.cpp`，放在侯捷網站供下載。其中 **traits** 相關部份與 SGI STL 源碼（GNU C++採用）幾無二致，與 RougeWave STL 的源碼（Borland C++Builder 採用）也相差無幾，這使我們吃了一顆定心丸。

以上所討論的 **traits** 機制，**[Austern99]** 第三章有更豐富的探討。

std::iterator 的保證

爲了符合規範，任何迭代器都應該提供五個巢狀相應型別，以利 **traits** 萃取，否則便是自外於整個 STL 架構，可能與其他 STL 組件無法順利搭配。但是寫碼難免掛一漏萬，誰也不能保證不會有粗心大意的時候。如果能夠將事情簡化，就好多了。如果我們設計一個 `iterators` class 如下，然後讓每一個新設計的迭代器都繼承自它，就可以保證符合 STL 的規範了：

```
template <class Category,
          class Value,
          class Distance = ptrdiff_t,
          class Pointer = Value*,
          class Reference = Value&>
struct iterator {
  typedef Category   iterator_category;
  typedef Value      value_type;
  typedef Distance   difference_type;
  typedef Pointer    pointer;
  typedef Reference  reference;
};
```

`iterator` class 不含任何成員，純粹只是型別定義，所以繼承它不會招致任何額外負擔。這樣的設計出現在 STL `<iterator>` 中。由於後三個參數皆有預設值，新的迭代器只需提供前兩個參數即可

利用 std::iterator 完成自定的迭代器

稍早的 `ListIter` 是土法煉鋼，現在我改用上述正式規格：

```
// 節錄自 myiter2.cpp
template <class Item>
struct ListIter : public iterator<std::forward_iterator_tag, Item>
{ ... }
```

[Josuttis99] 7.5.2 節提供了一個有趣的例子。我們知道，`insert_iterator` 要求使用它時必須指定第二個函式引數，做爲安插動作的起始位置。但是對關聯式容器（如 `set` 或 `map`）而言，每個元素的內容便已決定其在容器的位置，不受所謂「安插起始位置」的影響。因此，**[Josuttis99]** 7.5.2 節便設計了一個所謂的 `asso_insert_iterator`，不接受第二引數：

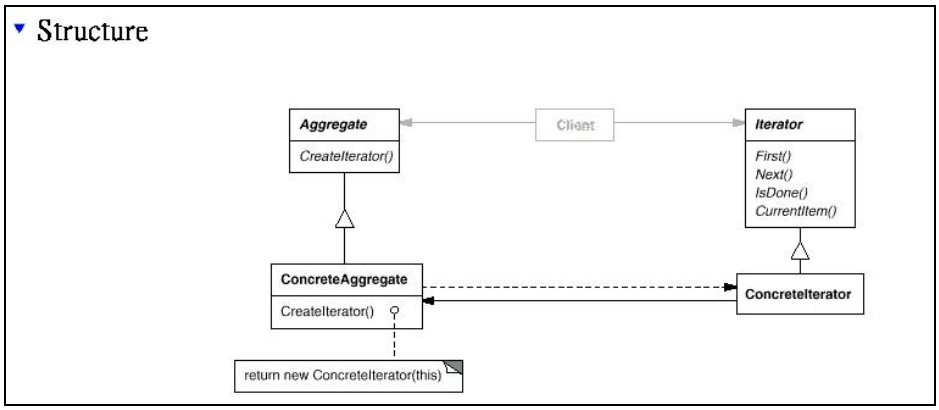
```
template <class Container>
class asso_insert_iterator
  : public iterator<std::output_iterator_tag, void, void, void, void>
{
protected:
   Container& container;

Public:
   // ctor. 注意，不再有第二參數。
   explicit asso_insert_iterator(Container& c) : container(c) { }
...
}
```

這個例子可能令你困惑的是，爲什麼指定給 base `itertator` 的五個相應型別，最後四個都是 `void` 呢？這是因爲 *output iterator* 純粹只用來寫入某些東西，因此其 *value type* 和 *difference type* 都應該是 `void`。

設計樣式 #16：Iterator

[Gamma95] 提供有 23 個設計樣式的完整描述。其中樣式 #16 即爲 *iterator*，其定義爲：提供一種方法，使得依序巡訪某個聚合物件（容器）所含的各個元素，而又無需暴露該聚合物件的內部表述方式。這個樣式的結構如圖三。

圖三/ *iterator* 樣式結構



首先我以先前使用過的 `List`，根據 **[Gamma95]** 所提供的實作指南和示例，完成一個 `ListIterator`。**[Gamma95]** 示例如下，請注意 `List` 的 `Get()` 操作函式竟以索引爲依據（例如取出第 n 個元素），這在串列之中實屬罕見（我的意思是這樣並不理想）。

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};

template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};

template <typename Item>
class ListIterator : public Iterator<Item>
{
public:
```

```
  ListIterator(const List<Item>* aList)
    : _list(aList), _current(0) {  }

  virtual void First() { _current = 0; }
  virtual void Next()  { _current++;   }
  virtual bool IsDone() const { return _current >= _list->size(); }
  virtual Item CurrentItem() const {
    if (IsDone())
       throw IteratorOutOfBounds();
    return _list->Get(_current);
  }

private:
  const List<Item>* _list;
  long _current;
};

template <typename Item>
class ReverseListIterator : public Iterator<Item>
{
public:
  ReverseListIterator(const List<Item>* aList)
    : _list(aList), _current(_list->size()-1) {  }

  virtual void First() { _current = _list->size()-1; }
  virtual void Next()  { _current--;   }
  virtual bool IsDone() const { return _current < 0; }
  virtual Item CurrentItem() const {
    if (IsDone())
       throw IteratorOutOfBounds();
    return _list->Get(_current);
  }

private:
  const List<Item>* _list;
  long _current;
};
```

我把測試程式分別寫爲 ite-ptn.h 和 ite-ptn.cpp，連同 list.h 和 list.cpp 一起運作。這個程式令 List 的每一個元素型別爲 Employee*，每個 Employee 有自己的名稱，還有一個 Print() 函式：

```
class Employee {
public:
  Employee(string name)
    : _name(name) {  }
```

```
  void Print() { cout << _name << endl; }
private:
  string _name;
};
```

現在我把六個職員加入串列之中，然後爲此串列製作出兩個迭代器，分別用於順向走訪和逆向走訪：

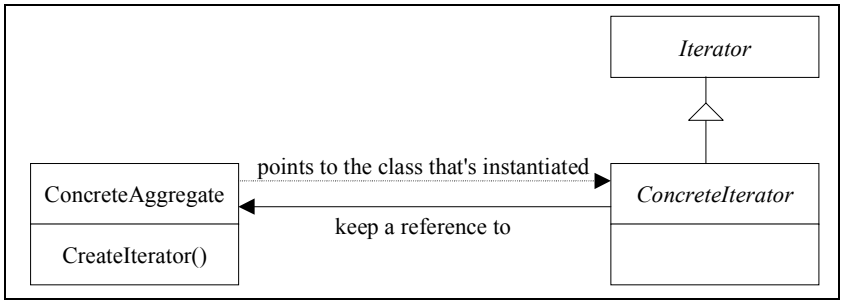
```
List<Employee*>* myEmployees = new List<Employee*>;

myEmployees->insert_end(new Employee(string("jjhou")));
...// 六個相同的 insert_end 動作

ListIterator<Employee*> forward(myEmployees);
PrintEmployees(forward);

ReverseListIterator<Employee*> backward(myEmployees);
PrintEmployees(backward);
```

檢閱 SGI STL 的迭代器實作碼

既然手上有 STL 原始碼，我不相信你沒有一絲一毫好奇心，企圖窺探他們如何實現 *iterator* 樣式。如果沒有這樣的好奇心，可能你不適合走資訊研究這條路☺。經歷了前面艱深技術的洗禮後，你有足夠的能力繼續完成這趟探索。

圖三的樣式結構顯示四個 classes `Aggregate`, `ConcreteAggregate`, `Iterator`, `ConcreteIterator` 交互作用。由於 STL 並沒有爲容器設計一個抽象基礎類別，因此 STL `list` 對這張結構圖的實現如圖四。當然啦，其中還是有微少差異，例如 `std::iterator` 其實並不是一個抽象類別，所有迭代器也不是非得繼承它不可。

圖四 / STL `list` 的 *iterator* 樣式結構圖

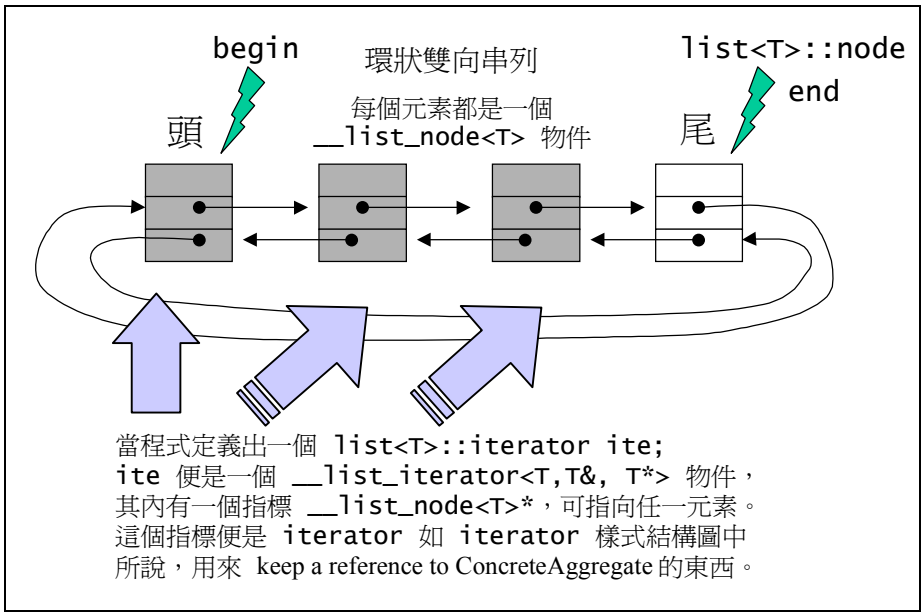



```
template <class T>
struct __list_node {
  typedef void* void_pointer;
  void_pointer next;       // 這是一個雙向串列（double linked list）
  void_pointer prev;       // 注意其型別爲 void*。其實可設爲 __list_node<T>*
  T data;
};

template<class T, class Ref, class Ptr>
struct __list_iterator {       // 未繼承 std::iterator
  // 未繼承 std::iterator，所以必須自行撰寫五個必要的相應型別
  typedef bidirectional_iterator_tag    iterator_category;
  typedef T                             value_type;
  typedef Ptr                           pointer;
  typedef Ref                           reference;
  typedef ptrdiff_t                     difference_type;

  // 另外數個常用的相應型別（內部使用）
  typedef __list_node<T>*               link_type;
  typedef __list_iterator<T, T&, T*>    iterator;
  typedef __list_iterator<T, Ref, Ptr>   self;

  link_type node;  // keep a reference to ConcreteAggregate

  // constructors
  __list_iterator(link_type x) : node(x) {}
  __list_iterator() {}
  __list_iterator(const iterator& x) : node(x.node) {}

  // 必要的操作行爲
  reference operator*() const { return (*node).data; }
  pointer operator->() const { return &(operator*()); }
  self& operator++() { ... }
```

```
  self operator++(int) { ... }
  self& operator--() { ... }
  self operator--(int) { ... }
  ...
};

template <class T, class Alloc = alloc>
class list {
protected:
  typedef __list_node<T>                list_node;
public:
  typedef list_node*                    link_type;

  // 當 client 定義一個 list<T>::iterator 物件，便相當於結構圖中
  // 的 CreateIterator()，隨後喚起 __list_iterator 的 ctor，
  // 設定 a-reference-to-ConcreteAggregate
  typedef __list_iterator<T, T&, T*> iterator;

protected:
  // 從實作細節看來，本 list 只維護一個節點指標，指向尾節點。由於這是一個
  //  環狀雙向串列，所以欲對外供應頭節點或尾節點，都十分容易。見 front(), back()
  link_type node;

public:
  iterator begin() { return (link_type)((*node).next); }
  iterator end() { return node; }

  reference front() { return *begin(); }
  reference back() { return *(--end()); }
...
};
```

圖五係根據 STL 源碼而繪的示意圖，顯示 list 和其迭代器之間的關係。

圖五/ STL 中的 list 和其迭代器之間的關係（本圖根據 STL 源碼而繪）



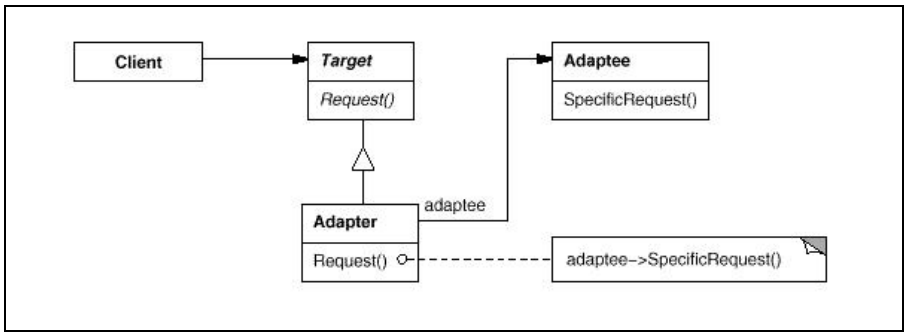
設計樣式 #6：Adaptor

[Gamma95] 對於 *adaptor* 樣式的定義是：將一個 class 的介面轉換爲另一個 class 的介面，使原本因介面不相容而不能合作的 classes，可以一起運作。在 STL 中，改變函式物件（function object）介面者稱爲 function adaptor，改變容器介面者稱爲 container adaptor。改變迭代器介面者稱爲 iterator adaptor。STL 提供的兩個容器 queue 和 stack，其實都只不過是一種 adaptor，它們修飾 deque 的介面而成就出另一種容器風貌。STL 亦提供有三個用來「將賦值動作改爲安插動作」的 iterator adaptors：back_insert_iterator, front_insert_iterator, insert_iterator。

根據 **[Gamma95]** 對 *adaptor* 樣式的定義，其實先前所見的 list::iterator 就是一個 adaptor。我們再仔細看看其實作碼，它有一個原生指標，指向節點結構，但是它開放給外界的操作行爲如 operator*, operator->, operator++, operator-- 卻都不是原生指標原有的行爲。它轉換了「指向節點結構」之原生指標的介面，並使得原本無法合作的 list 容器和 find 演算法現在可以合作了。

[Gamma95] 將 *adaptor* 樣式分爲 *class adaptor* 和 *object adaptor*，圖六是 *object adaptor* 的結構圖。

圖六/ *object adaptor* 的結構



檢閱 SGI STL 的 iterator adaptors 和 container adaptors

讓我們從 STL 源碼中尋找靈感。以下的源碼摘錄和附加說明，可讓你清楚看到 adaptor 的實作手法。下面是 SGI STL `insert_adaptor` 源碼節錄：

```
template <class Container>
class insert_iterator {
protected:
  Container* container;        // keep-a-reference-to-Adaptee
  typename Container::iterator iter;     // 此物只對循序式容器有用
public:
  typedef output_iterator_tag  iterator_category;
  typedef void                 value_type;
  typedef void                 difference_type;
  typedef void                 pointer;
  typedef void                 reference;

  insert_iterator(Container& x, typename Container::iterator i)
    : container(&x), iter(i) {}

    insert_iterator<Container>&
  operator=(const typename Container::value_type& value)
    iter = container->insert(iter, value);    // 轉呼叫
    ++iter;
    return *this;
  }
};
```

下面是 SGI STL `stack` 源碼節錄：

```
template <class T, class Sequence = deque<T> >
class stack {
protected:
  Sequence c;    // containment
public:
  // 新介面，其實只是轉呼叫（forwarding）
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_back(); }
};
```

下面是 SGI STL `queue` 源碼節錄：

```
template <class T, class Sequence = deque<T> >
class queue {
protected:
  Sequence c;    // containment
public:
  // 新介面，其實只是轉呼叫（forwarding）
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_front(); }
};
```

相關書 籍

[Austern98]: *Generic Programming and the STL - Using and Extending the C++ Standard Template Library*, by Matthew H. Austern, Addison Wesley 1998. 548 pages
繁體中文版：《泛型程式設計與 STL》，侯捷/黃俊堯合譯，碁峰 2000。548 頁

[Gamma95]: *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995. 395 pages
簡體中文版：《設計模式》，李英軍等譯，機械工業出版社，2000. 254 頁

[Josuttis99]: *The C++ Standard Library - A Tutorial and Reference*, by Nicolai M. Josuttis, Addison Wesley 1999. 799 pages
繁體中文版：侯捷計劃中

[Lippman98]: *C++ Primer*, 3rd Editoin, by Stanley Lippman and Josée Lajoie, Addison Wesley Longman, 1998. 1237 pages.
繁體中文版：《C++ Primer 中文版》，侯捷譯，碁峰 1999. 1237 頁

[Meyers96]: *More Effective C++*, by Scott Meyers, Addison-Wesley, 1996. 318 pages
繁體中文版：《More Effective C++ 中文版》，侯捷譯，培生 2000. 318 頁

[Meyers98]: *Effective C++*, Second Edition, by Scott Meyers. Addison Wesley Longman, 1998. 256 pages
繁體中文版：《Effective C++ 2/e 中文版》侯捷譯，培生 2000. 256 頁