泛型演算法（Generic Algorithms）與 Function Object

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

泛型演算法應該算是 STL 六大組件中最容易的一部份了。可能你的直覺反應是：『喔是嗎，演算法聽起來是高深的學問！』是的，這種印象並沒有錯，經過嚴謹的數學推導，並通過時間的考驗、眾多使用者千錘百鍊後碩果僅存的演算法，是電算領域中最寶貴的資產。但如果不是要你推導演算法，而只是要你根據演算法寫出程式碼，我想對任何程式員而言，都不會有什麼困難。

爲什麼我說「泛型演算法是 STL 六大組件中最容易的一部份」，因爲就實作技術面上，它只是一般的 function templates，不需要特殊的 C++ 語言技巧或是像 traits 那樣的抽象技術。

●泛型演算法通則

所有泛型演算法的前兩個引數都是一對 iterators，通常我們稱之爲 first 和 last，用以標示將被操作之 container（或陣列）的元素範圍。元素範圍表示法是一個左涵蓋區間，通常寫成這樣：

```
[first, last)
```

這表示範圍從 first 開始，直到 last 結束，但不含 last。由兩個 iterators 標示出來的所謂 range，有嚴謹的定義，我曾在本系列的第一篇文章中談過。

每個演算法的宣告式，都表現出它所需要的最低層次的 iterators 需求。（我曾在第二篇文章中介紹過五種層次的 iterators）。例如：

```
template < class InputIterator, class Type >
Type accumulate(InputIterator first, InputIterator last, Type init );
```

這個宣告式表示，accumulate() 需要的泛型指標類型爲 InputIterator。在此，InputIterator 扮演的只是一個 template 參數，所以上式若這樣表現當然也可以：

```
template < class I, class T >
T accumulate(I first, I last, T init );
```

但是這樣一來表現不出在 accumulate() 函式內部對於 I 的需求。事實上整個 STL 對於 concepts 的嚴謹分類與組織，正是它在泛型思維下的一個重要貢獻。

再舉一個例子：

```
template < class InputIterator, class OutputIterator >
OutputIterator adjacent_difference(
  InputIterator first, InputIterator last,
  OutputIterator result );
```

這表示 adjacent_difference() 需要的前兩個函式引數是 InputIterator，而第三個函式引數以及傳回值的型別是 OutputIterator。

另外三個不同的例子是（注意它們使用不同層級的 iterators）：

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end (ForwardIterator1 first1,
                           ForwardIterator1 last1,
                           ForwardIterator2 first2,
                           ForwardIterator2 last2);

template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward (BidirectionalIterator1 first,
                                      BidirectionalIterator1 last,
                                      BidirectionalIterator2 result);

template <class RandomAccessIterator>
void random_shuffle (RandomAccessIterator first,
                     RandomAccessIterator last);
```

注意，編譯器無法爲你偵測「傳遞一個無效型式的 iterator 給某個演算法」這種錯誤，因爲「iterator 型式」並不是真實的型別，它只是傳遞給 function template 的型別參數（type parameters）而已。

STL 演算法多支援有多重版本，其中一個版本使用內建的運算子，另一個版本使用 function object 或函式指標，來供應對元素的另一種操作行爲。例如 sort()，預設情況下它使用「小於」運算子來比較 container 內的兩個相鄰元素。然而如果 container 的元素型別並未供應「小於」運算子，或如果我們希望定義屬於我們自己的「小於」運算子，我們可以傳一個 function object 或函式指標給 sort()，以滿足「小於」的語意。另有一些演算法則是將這樣的兩個版本乾脆分開爲兩個不同名稱的演算法，允許訂製「特殊操作行爲」的那個，總是以 _if 做爲尾詞，例如 find_if(), replace_if()。

凡演算法會修改其操作標的物（一個 container）之內容者，通常提供有兩個版本：一個是 in-place（就地）版，就地改變其標的物內容；另一個是 copy 版，將原標的物內容製作一份副本，修改該副本，然後傳回該副本。copy 版總是以 _copy 做爲名稱尾詞，例如 replace() 和 replace_copy()。但並不是所有「會改變其標的物內容」的演算法都有一個 copy 版，例如 sort() 就沒有 copy 版。

欲使用任何一個泛型演算法，必須先含入相關的表頭檔：

```
#include <algorithm>
```

但對於四個數值演算法：adjacent_difference(), accumulate(), inner_product(), 和 partial_sum()，則應該含入：

```
#include <numeric>
```

[Austern99] 有一份很棒而且很詳細的泛型演算法規格與討論。

●泛型演算法列表

STL 共提供了 70 個泛型演算法。下面是它們的列表（按字母排序）並附簡單說明。你可以在 [Lippman98] 的附錄中找到每一個 STL 演算法的文字說明和使用實例。

泛型演算法名稱	說明
accumulate()	元素累加
adjacent_difference()	相鄰元素的差額
adjacent_find()	搜尋相鄰的重複元素
binary_search()	二元搜尋
copy()	複製
copy_backward()	逆向複製
count()	計數
count_if()	在特定條件下計數
equal()	判斷相等與否
equal_range()	判斷相等與否（傳回一個上下限區間範圍）
fill()	改填元素值
fill_n()	改填元素值，n 次
find()	搜尋
find_if()	在特定條件下搜尋
find_end()	搜尋某個子序列的最後一次出現地點
find_first_of()	搜尋某些元素的首次出現地點
for_each()	對範圍內的每一個元素施行某動作
generate()	以指定動作的運算結果充填特定範圍內的元素
generate_n()	以指定動作的運算結果充填 n 個元素內容
includes()	涵蓋於
inner_product()	內積
inplace_merge()	合併並取代（覆寫）
iter_swap()	元素互換
lexicographical_compare()	以字典排列方式做比較
lower_bound()	下限
max()	最大值
max_element()	最大值所在位置
min()	最小值
min_element()	最小值所在位置
merge()	合併兩個序列
mismatch()	找出不吻合點
next_permutation()	獲得下一個排列組合

```
nth_element()                重新安排序列中第 n 個元素的左右兩端
partial_sort()               局部排序
partial_sort_copy()          局部排序並複製到它處
partial_sum()                局部總和
partition()                  切割
prev_permutation()           獲得前一個排列組合
random_shuffle()             隨機重排
remove()                     移除某種元素（但不刪除）
remove_copy()                移除某種元素並將結果複製到另一個 container
remove_if()                  有條件地移除某種元素
remove_copy_if()             有條件地移除某種元素並將結果複製到另一個 container
replace()                    取代某種元素
replace_copy()               取代某種元素，並將結果複製到另一個 container
replace_if()                 有條件地取代
replace_copy_if()            有條件地取代，並將結果複製到另一個 container
reverse()                    顛倒元素次序
reverse_copy()               顛倒元素次序並將結果複製到另一個 container
rotate()                     旋轉
rotate_copy()                旋轉，並將結果複製到另一個 container
search()                     搜尋某個子序列
search_n()                   搜尋「連續發生 n 次」的子序列
set_difference()             差集
set_intersection()           交集
set_symmetric_difference()   對稱差集
set_union()                  聯集
sort()                       排序
stable_partition()           切割並保持元素相對次序
stable_sort()                排序並保持等值元素的相對次序
swap()                       置換（對調）
swap_range()                 置換（指定範圍）
transform()                  以兩個序列爲基礎，交互作用產生第三個序列
unique()                     將重複的元素摺疊縮編，使成唯一
unique_copy()                將重複的元素摺疊縮編，使成唯一，並複製到他處
upper_bound()                上限

-- 以下是 heap 相關演算法 --
make_heap()                  製造一個 heap
pop_heap()                   從 heap 內取出一個元素
push_heap()                  將一個元素推進 heap 內
sort_heap()                  對 heap 排序
-------------------------------------------------------------------------
```

●應用實例

以下示範數個演算法的使用方式。程式中的註解足以說明其動作目的。

```
#include <numeric>
#include <vector>
#include <functional>
```

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

int main()
{
  int ia[] = { 1, 2, 3, 4, 5, 7, 9, 11 };
  vector<int> iv(ia, ia+8);

  // 累加
  cout << accumulate(iv.begin(), iv.end(), 10) << endl;  // 52

  // 相鄰差值
  int result;
  adjacent_difference(iv.begin(), iv.end(), iv.begin());

  // 拷貝到 ostream_iterator 去，每列印一個元素，即加上一個空格
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
  // 1 1 1 1 1 2 2 2

  // 計算元素值爲 2 的個數
  cout << count(iv.begin(), iv.end(), 2) << endl;  // 3

  // 計算奇數元素的個數
  cout << count_if(iv.begin(), iv.end(),
                   bind2nd(modulus<int>(),2)) << endl ; // 5

  // 從頭開始填入新值 7，填 3 次。
  fill_n(iv.begin(), 3, 7);
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
  // 7 7 7 1 1 2 2 2

  // 內積。7*7 + 7*7 + 7*7 + 1*1 + 1*1 + 2*2 + 2*2 + 2*2
  cout << inner_product(iv.begin(), iv.end(), iv.begin(), 0) << endl; // 161

  // 排序
  sort(iv.begin(), iv.end());
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
  // 1 1 2 2 2 7 7 7

  // 顛倒元素次序
  reverse(iv.begin(), iv.end());
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
  // 7 7 7 2 2 2 1 1

  // 旋轉。將 [first, middle) 和 [middle, last) 交換。
  rotate(iv.begin(), iv.begin()+3, iv.begin()+6);
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
```

```
  // 2 2 2 7 7 7 1 1
}
```

●STL Algorithms 的原始碼

我說過了，從程式技術觀之，STL algorithms 是六大組件中最簡單的一種。只要你會寫 function template，並知道你要寫的演算法的實際意義，也知道所謂的 iterator 的意義，就行了。以下我便挑幾個簡單的演算法原碼，給大家看看（摘錄自 STL SGI 版）。

◎inner_product

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init) {
  for ( ; first1 != last1; ++first1, ++first2)
    init = init + (*first1 * *first2);
  return init;
}
```

◎for_each

```
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f) {
  for ( ; first != last; ++first)
    f(*first);
  return f;
}
```

明顯可見，這個演算法就是把 [first, last) 範圍巡訪一遍，每遇到一個元素，在以 f 施行於上。f 是一個使用者指定的函式，可以函式指標的型式或是 function object（稍後介紹）的型式傳入。

◎find

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

這個演算法把 [first, last) 範圍巡訪一遍，每遇到一個元素，就判斷是否等於 value。如果相等就回返並傳回該處位置。

◎count

```
template <class InputIterator, class T, class Size>
void count(InputIterator first, InputIterator last, const T& value,
           Size& n) {
  for ( ; first != last; ++first)
    if (*first == value)
      ++n;
}
```

這個演算法把 [first, last) 範圍巡訪一遍，每遇到一個元素，就判斷是否等於 value。如果是就將計數器加 1。最後傳回計數器。

◎transform

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryOperation op) {
  for ( ; first != last; ++first, ++result)
       *result = op(*first);
  return result;
}
```

這個演算法把 [first, last) 範圍巡訪一遍，每遇到一個元素，就以一元運算式 op 施行於上，並將結果置於 result 所指位置。

●function object

所有的 STL 演算法都是在一個被指定的範圍內，針對每一個元素進行某種運算動作。但是前面也說過，STL 演算法通常提供第二版本，允許使用者指定某種特殊運算，取代預設的運算動作。例如 inner_product，數學中對於向量內積的定義，以 (a, b, c) 爲例，是 a*a + b*b + c*c。這也正是 inner_product() 第一版本的行爲。所以前例數列（視爲向量）的內積結果是：

```
// 原爲 { 7 7 7 1 1 2 2 2 }
// 內積。7*7 + 7*7 + 7*7 + 1*1 + 1*1 + 2*2 + 2*2 + 2*2 + 0（初值）
cout << inner_product(iv.begin(), iv.end(), iv.begin(), 0) << endl; // 161
```

但是 STL 允許你自由改變內積的定義。inner_product 有一個版本如下：

```
template <class InputIterator1, class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init, BinaryOperation1 binary_op1,
                BinaryOperation2 binary_op2) {
```

因此 STL 允許你這麼做：

```
// int ia[] = { 1, 2, 3, 4, 5, 7, 9, 11 };
// vector<int> iv 原爲 { 7, 7, 7, 1, 1, 2, 2, 2 }
// -(7+1)-(7+2)-(7+3)-(1+4)-(1+5)-(2+7)-(2+9)-(2+11)+ 100（初值）
```

```
inner_product(iv.begin(), iv.end(), ia, 100,
              minus<int>(), plus<int>() );   // 29
```

從規格可見，最後兩個參數都是 BinaryOperation，在 STL concepts 架構中，它代表一個雙運算元的運算式。我們可以自己寫這樣的函式，並以其函式指標做爲引數傳給 inner_product()。

但其實正式形式並非如此！

STL 將 BinaryOperation 這類東西稱爲 function object，或稱爲 functor。以 C++ 程式技術而言，這是一種「將 call operator 多載化」的 class。例如（以下是 STL 原始碼，摘自 STL SGI 版）：

```
template <class Arg1, class Arg2, class Result>
struct binary_function {
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};

template <class T>
struct plus : public binary_function<T, T, T> {
   T operator()(const T& x, const T& y) const { return x + y; }
};

template <class T>
struct minus : public binary_function<T, T, T> {
   T operator()(const T& x, const T& y) const { return x - y; }
};
```

對於這種「將 call operator 多載化」的 template class，有兩種用法，一是以獨立角色來使用：

```
minus<int> m;
cout << m(4, 1) << endl;  // 3

plus<int> p;
cout << p(4, 1) << endl;  // 5
```

當程式執行 m(4, 1)，喚起 minus::operator()(const T& x, const T& y) const。正是由於其型式類似「函式呼叫」，所以它才被稱爲一個 function object。

另一種用法是搭配 STL algorithms 使用，這也是它的最大用途。例如先前所舉的例子：

```
inner_product(iv.begin(), iv.end(), ia, 100, minus<int>(), plus<int>());
```

注意其語法，參數 minus<int>() 的意義是：根據 class minus<int>，產生一個無具名的 object。這裡的小括號並不意味喚起其 call operator，而是用來表示「產生一個無具名的 object」。這個「無具名的 function object」被傳進 inner_product 後，才會被喚起其 call operator（當然得指定引數）：

```
template <class InputIterator1, class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
```

```
                InputIterator2 first2, T init, BinaryOperation1 binary_op1,
                BinaryOperation2 binary_op2) {
  for ( ; first1 != last1; ++first1, ++first2)
    init = binary_op1(init, binary_op2(*first1, *first2));
  return init;
}
```

下面是數個會用到 function object 的 STL algorithms 的原始碼。

◎find_if

```
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}
```

這個演算法把 [first, last) 範圍巡訪一遍，每遇到一個元素，就判斷 pred 施行於上是否爲真。如爲真，就回返並傳回該處位置。注意最後一個參數的型別 Predicate，它在 STL concepts 結構中意指一個「傳回 bool 值的一元函式」。如果是 「傳回 bool 值的二元函式」，則以 BinaryPredicate 表示。

◎count_if

```
template <class InputIterator, class Predicate, class Size>
void count_if(InputIterator first, InputIterator last, Predicate pred,
              Size& n) {
  for ( ; first != last; ++first)
    if (pred(*first))
      ++n;
}
```

這個演算法把 [first, last) 的範圍巡訪一遍，每遇到一個元素，就判斷 pred 施行於上是否爲真。如爲真，就將計數器加 1。最後傳回計數器。

◎generate

```
template <class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen) {
  for ( ; first != last; ++first)
    *first = gen();
}
```

這個演算法把 [first, last) 範圍巡訪一遍，每遇到一個元素，就將其值改爲 gen 所生結果。

●STL 內建的 function object

STL 內建的 function objects 分爲算術、相對關係、邏輯三大類。每個 object 都是一個 class template。全部都定義於含入檔 <functional> 內。

◎算術類（Arithmetic）Function Objects

STL 內建的「算術類 function objects」支援加法、減法、乘法、除法、模數（modulus）和否定（negation）運算。

* 加法：plus<Type>
* 減法：minus<Type>
* 乘法：multiplies<Type>
* 除法：divides<Type>
* 模數（Modulus）：modulus<Type>
* 否定（Negation）：negate<Type>

◎相對關係類（Relational）Function Objects

STL 內建的「相對關係類 function objects」支援了等於、不等於、大於、大於等於、小於、小於等於六種運算。

* 等於（Equality）：equal_to<Type>
* 不等於（Inequality）：not_equal_to<Type>
* 大於（Greater than）：greater<Type>
* 大於或等於（Greater than or equal）：greater_equal<Type>
* 小於（Less than）：less<Type>
* 小於或等於（Less than or equal）：less_equal<Type>

◎邏輯類（Logical）Function Objects

STL 內建的「邏輯類 function objects」支援了邏輯運算中的 And、Or、Not 三種運算。

* Logical And：logical_and<Type>
* Logical Or：logical_or<Type>
* Logical Not：logical_not<Type>

●自製 function object

讓我們嘗試自己做一個 function object。如果我打算爲一個 vector 產生初值，其元素值呈累進型式，每次增加 5。我決定使用 generate()，並傳給它一個自定的 function object 做爲數值產生器。由於 generate() 內部每一次迴圈迭代，便呼叫一次我們所給予的 function object，所以我有兩種作法：一是設計一個函式，內有一個 static 變數，每次被呼叫，該變數就累加 5。第二個作法是設計一個 function object class，有一個 data member，每次其 call operator 被喚起，就將 data member 累加 5。

下面是兩種作法的實現：

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

int elem_gen1() {
  static int _seed = 0;
  return _seed += 5;
}

class elem_gen2 {
public:
  elem_gen2(int seed=0) : _seed(seed) { }
  int operator()() { return _seed += 5; }
private:
  int _seed;
};

int main()
{
  vector<int> iv(10);

  generate(iv.begin(), iv.end(), elem_gen1);
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
  // 5 10 15 20 25 30 35 40 45 50

  generate(iv.begin(), iv.end(), elem_gen2());
  copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
  // 5 10 15 20 25 30 35 40 45 50
}
```

上述的 elem_gen2 當然稱得上是個 function object，但是不夠健全，它沒辦法和 function adaptors（下期主題）搭配使用；而這種互相配接的能力，在 STL 中至爲重要。這樣的陽春型 function objects 之所以不能搭接 function adaptors，原因是 function adaptors 可能會需要用到其引數和傳回值的型別。STL 的含入檔 <functional> 中提供有兩個 struct，可以幫助我們建立起健全的 function objects：

```
template <class Arg, class Result>
struct unary_function {
   typedef Arg argument_type;
   typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
   typedef Arg1 first_argument_type;
```

```
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

如果你想自定一個健全的 function object，必須衍生自上述兩個 structs 之一。例如：

```
template <typename T1, typename T2>
struct myfunctor : public std::binary_function<T1, T2, T1>
{  ... }
```

●Predicate

前面我曾說過，所謂 Predicate，意指「傳回 bool 值的一元函式」。凡附帶條件需求的演算法，例如 count_if, find_if, remove_if, replace_if，都需要一個做爲條件式的判斷準則。

但，是否只要傳回 bool 值，便可視爲一個有效的 predicate 呢？不然。[Josuttis99] 給了這樣一個例子：

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

class Nth {
private:
  int nth;
  int count;
public:
  Nth(int n) : nth(n), count(0) { }
  bool operator()(int) {  return ++count == nth;  }
};

int main()
{
  list<int> col1;
  for (int i=1; i<=12; ++i)
    col1.push_back(i);

  copy(col1.begin(), col1.end(), ostream_iterator<int>(cout, " "));
  // 1 2 3 4 5 6 7 8 9 10 11 12

  list<int>::iterator pos;
  pos = remove_if(col1.begin(), col1.end(),
                  Nth(3)),
                  col1.erase(pos, col1.end());

  copy(col1.begin(), col1.end(), ostream_iterator<int>(cout, " "));
  // 1 2 4 5 7 8 9 10 11 12
```

```
}
```

其中 Nth class 將 call operator 多載化並令之傳回一個 bool，所以它似乎符合 predicate 的條件。但是當我們以 Nth(3) 做爲 remove_if() 的條件，企圖移除第三個元素時，但導至第三個元素和第六個元素都被移除。這是爲什麼？

原因出在 remove_if 之中又呼叫了 remove_copy_if（SGI 版、PJ Plauger 版、Rouge Wave 版皆是如此）：

```
template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first,
                          ForwardIterator last,
                          Predicate pred)
{
  first = find_if(first, last, pred);
  ForwardIterator next = first;
  return first == last ? first : remove_copy_if(++next, last, first, pred);
}
```

於是 remove_if 刪除掉第三個元素後，remove_copy_if 再次以 Nth(3) 爲刪除條件，將第六個元素給移除掉了。第九個元素會不會也被刪除掉呢？誰也猜不準，這得看 remove_copy_if 是怎麼做的：

```
template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred) {
  for ( ; first != last; ++first)
    if (!pred(*first)) {
      *result = *first;
      ++result;
    }
  return result;
}
```

好加在，沒有暗中又謀殺了誰。

因爲行爲的不符預期，致使用者必須翻箱倒櫃地找出並詳細觀察原始碼，是一件荒謬事兒。但是這不能算是一個 bug，因爲 C++ Standard 並沒有規定 predicate 不能再被複製，也沒有規定不能被複製多少次。因此，爲了讓你的 predicate 能有符合預期的表現，最保險的作法，是將 operator() 宣告爲 const member function。也就是說，一個良好的 predicate，不論被呼叫多少次，其狀態（state，內部資料）應該不變。

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。