第 17 章

標準樣板函式庫（一）：簡介

標準樣板函式庫 (Standard Template Library)，簡稱為 STL，為 C^{++} 提供給所有 C^{++} 程式設計員的一個大禮物，平常利用 C^{++} 撰寫程式的人如果沒有學著使用 STL 於其所設計的程式中，就如同 21 世紀的士兵仍拿著傳統的刀劍武器來打仗一般。簡單來說，C^{++} 所提供的 STL 是將許多常用的資料結構與樣板函式集結成一個方便使用的函式庫。由於使用方式相當的簡單，同時又具有相當的實用性，自其正式被加入 C^{++} 的標準函式庫後，就立即廣泛地為許多程式設計員使用。事實上，由於 STL 相當受到許多程式設計員的歡迎與採用，如今已有不少專書來詳細介紹 STL 的使用方法及其相關的應用。在本書中，筆者將以四章的篇幅來介紹 STL 常見的用法。本章中將透過一個簡單程式範例對 STL 的語法作一整體性的描述，在之後的三章，我們將以較詳細的方式說明。

17.1 標準樣板函式庫範例

C^{++} 所提供的標準樣板函式庫中，在使用上大體可以區分為以下幾個類型：容器 (container)，迭代器 (iterator)， 泛型演算函式 (generic algorithm)，函式物件類別 (functor class)，簡單的介紹如下：

- **容器**　是一種利用特殊資料存取機制設計而成的樣板資料型別。
- **迭代器**　為一種類似指標的資料型別，其物件可以用來間接地存取在容器內的資料。
- **泛型演算函式**　為樣板函式，此種函式透過迭代器存取容器內的資料來達成其功能。
- **函式物件類別**　為一種類別內有覆載「函式運算子」，即 `operator()`。

除了以上幾個主要類型之外，STL 尚有一些較次要的變化型式，這將留待以後介紹。以下我們利用一個簡單的程式來說明這四個類型的綜合使用方式。讀者可以暫時擱置了解程式碼詳細的運作機制，只須大略認識所使用的 STL 程式碼樣式，至於程式碼的說明，將會在以後的各小節中再加以介紹。

在這個範例中，我們首先產生一個可以儲存 20 個元素的向量陣列容器 (vector container)，與一個可以存取此向量陣列資料的迭代器，然後將數字 1 到 20 透過迭代器一一存入向量陣列內，接下來依據數字個位數的大小，由大到小將容器內的資料重新排序，最後利用一個自訂的泛型函式將此向量陣列的資料印出：

stl_example.cc

```
#include <iostream>
#include <vector>          // 使用向量陣列容器與迭代器
#include <algorithm>       // 使用排序演算函式

using namespace std ;

// (4) 定義一函式物件類別 : 用以比較兩筆資料的個位數字大小
template <class T>
struct  Remainder {
    bool  operator() ( const T& a , const T& b ) const {
        return  a%10 > b%10 ;
    }
} ;

// (5) 列印容器
template <class T>
void  print_container( const T& a , const T& b , char *sep ) {
    for ( T itr = a ; itr != b ; ++itr )  cout << *itr << sep ;
}

int  main() {

    // (1) 定義一個可以儲存 20 個整數的向量陣列容器
    vector<int>  foo(20) ;

    // (2) 定義一個針對整數向量陣列的迭代器物件
    vector<int>::iterator  itr ;

    // 利用迭代器物件將整數 1 到 20 一一的存入此向量陣列中
    int  no = 1 ;
    for ( itr = foo.begin() ; itr != foo.end() ; ++itr )
        *itr = no++ ;

    // (3) 利用排序演算函式將向量陣列元素依個位數的大小
    //      由大至小重作排序
    sort( foo.begin() , foo.end() , Remainder<int>() ) ;
```

```
    // (5) 利用函式將向量陣列印出，且元素之間以一空白分開
    print_container( foo.begin() , foo.end() , " " ) ;
    cout << endl ;

    return 0 ;

}
```

程式則會依個位數字的大小，由大到小印出

```
19 9 8 18 7 17 6 16 5 15 4 14 3 13 2 12 11 1 10 20
```

這裡的 (1) 代表容器，(2) 為迭代器，(3) 為排序泛型演算函式， (4) 為函式物件類別，(5) 為自定的樣板函式。我們將在以下的各小節中加以說明此範例所使用到的 STL 語法及其作用。

17.2 容器

在 STL 中，容器 (container) 是用一種利用特殊的資料結構所設計而成的樣板類別，不同用途的樣板類別各自提供了許多方便的函式可以用來存取類別內的資料，使得許多常用的而且複雜的資料結構在使用上變得相當簡便。

序列容器與關聯容器

在 STL 中總共提供了兩種類型的容器，一為序列容器 (sequence container)，另一種為關聯容器 (associative container)。在序列容器中，資料是以序列方式 (squential) 儲存，資料在記憶空間的儲存方式可以是緊鄰的或者是分散的，這類的容器包含向量陣列 (vector)，佇列陣列 (deque)，與串列 (list)。序列容器的元素資料值與其所儲存的位置並沒有任何關聯，也就是說，序列容器內的元素，不管儲存位置為何，其資料都可以隨時加以更動，沒有任何限制。相反的，在關聯容器內的元素，元素的儲存位置與其資料值是有所關聯的，資料被置放在樹狀結構[629] (tree structure) 的節點內，當資料須要存入樹狀結構或者是由樹狀結構取出時，都須要依次比較在節點 (node) 內資料的關聯值 (key)，才能找出資料在樹狀結構的存放地點。由於關聯容器的元素都是經過資料比較，找到適當的位置後才存入容器內，這樣的資料存取機制可以節省資料的搜尋時間，這對處理大型資料庫而言是相當重要的。關聯性容器有集合 (set) 與複集合 (multiset) 與映射 (map) 與複映射 (multimap) 等四種容器。本章只有略述序列容器的簡單使用方法，詳細的用法則須參考下一章的內容說明，而關聯容器的使用則將留待第十九章再加以說明。在程式內使用不同的容器須要加入個別的標頭檔，表格 17.1 內有各容器所須使用的標頭檔。

由於各容器內部資料結構設計的不同，每種容器都有其最佳使用條件，有如不同種類的汽車，會因設計的不同而有不一樣的最佳駕駛環境。舉例來說，向量陣列

名稱	容器	標頭檔
向量陣列	vector	#include <vector>
佇列陣列	deque	#include <deque>
串列	list	#include <list>
集合，複集合	set，multiset	#include <set>
映射，複映射	map，multimap	#include <map>

表格 17.1: 資料容器與其使用的標頭檔

(vector) 是 STL 提供的所有容器中最常被使用的容器。其主要特點是陣列長度可以在執行過程中隨著需求自動調整，完全不須要使用者的介入，使用者透過簡單的函式執行就可以輕易地將元素加到陣列末端，或者是將末端的元素移除，這種可調式的陣列沒有傳統陣列因固定長度所造成的種種不便缺點，但卻仍保有與傳統陣列相當的執行效率。佇列陣列 (deque) 的使用方式幾乎與向量陣列相同，但其元素可以由陣列的前後兩端加入或者是移除。 串列 (list) 為第三種序列容器，串列為一種利用雙鏈節資料結構[590] (doubly linked list) 所設計而成的容器，其特點是資料可以用大約等長的時間插入到串列之中，或者是由串列中移除。vector 與 deque 兩容器的資料儲存是緊鄰的[註1]，而 list 的資料卻是分散儲存的。此外，廣義來說，傳統的陣列與 C++ 字串 (string) 因為也是以緊鄰的方式儲存元素資料，因此也可以被視為一種序列容器。

容器的使用

STL 所定義的容器都是樣板類別， 程式設計員可以輸入所要的型別樣板參數來設定容器中所要處理的資料型別。例如，以下用不同的設定方式來定義向量陣列與佇列：

```
vector<int>   a(10) ;       // 可以儲存十個整數的向量陣列
vector<char>  b(20,'m') ;  // 可存二十個字元，每個字元的初值為字元 m
deque<double> c(5) ;        // 可存五個雙精確度浮點數的佇列陣列
```

不同的容器有著不同初值設定方式來產生物件，但也有一些相似的設定方式。此外每一種容器都有其特殊的使用方式，以向量陣列為例，其使用方式除了與傳統陣列的使用方式相當之外，容器也另外定義了一些特殊的成員函式以供使用，例如：

```
// 一個向量陣列  沒有任何元素
vector<int>  foo  ;
```

[註1]在 deque 陣列中，元素的真正儲存方式並不是完全緊鄰，但使用上卻可以將之看成是相鄰的

成員函式	功能	vector	deque	list
push_back	由末端加入元素	√	√	√
pop_back	由末端去除元素	√	√	√
push_front	由前端加入元素	×	√	√
pop_front	由前端去除元素	×	√	√
front	查看或更改最前端元素	√	√	√
back	查看或更改最末端元素	√	√	√
size	元素個數	√	√	√
operator[]	下標運算子	√	√	×

表格 17.2: 序列容器可以使用的成員函式，√ 代表可使用，× 代表不能使用

```
// 連續十次將整數 5 加入向量陣列的末尾
for ( int i = 0 ; i < 10 ; ++i )  foo.push_back(5) ;

// 列印此向量陣列
for ( int i = 0 ; i < foo.size() ; ++i ) cout << foo[i] << ' ' ;
```

這裡的 size() 成員函式回傳向量陣列元素的個數，而 push_back(5) 成員函式的作用是將整數 5 由末尾加入向量陣列，同時陣列的長度就會自動增加一個。對向量陣列而言，資料可以用 push_back 函式由末尾加入或是用 pop_back 函式將資料由末尾移除，這兩個函式也同時可以用在佇列 (deque) 與串列 (list) 容器上。除此之外，後兩者也可以用 push_front 函式將元素由容器的前端加入或者使用 pop_front 函式將前端元素移除，然而此兩個成員函式卻不能用於向量陣列容器。例如，以下將數字 1 到 5 由佇列陣列的前端依次加入，然後再一一印出，由於新的元素每次是由前端加入，因此其印出結果為 5 4 3 2 1：

```
deque<int>  foo ;
for ( int i = 1 ; i <= 5 ; ++i ) foo.push_front(i) ;
for ( int i = 0 ; i < foo.size() ; ++i ) cout << foo[i] << ' ' ;
cout << foo.front() << endl ;   // 印出第一個元素
cout << foo.back()  << endl ;   // 印出最後一個元素
```

這三個序列容器皆可以用 front() 成員函式取得第一個元素，back() 成員函式取得最後一個元素。

為方便比較起見，我們將一些常用的序列容器成員函式用一個簡表表示，如表格 17.2，欄位內的 √ 符號代表可以使用，而 × 代表不能使用。舉例來說：以下程式的串列物件先由末端存入 3 與 8，再由前端送入 5 與 7，之後分別由後端與前端各取出一個元素，最後再將最前端的元素改為 9，如此最後串列的最前端元素為 9，末端元素為 3，且串列只有兩個元素。

```
list<int> foo ;

foo.push_back(3)  ;    foo.push_back(8)  ;  // 末端依次加入 3 與 8
foo.push_front(5) ;    foo.push_front(7) ;  // 前端依次加入 5 與 7
foo.pop_front()   ;    foo.pop_back()    ;  // 去除前後端元素

foo.front() = 9 ;                           // 將前端元素改為 9

cout << foo.front() << ' ' << foo.back() << ' ' << foo.size() ;
```

以上的程式若是改為使用佇列容器仍然是正確的，但是若是改用向量陣列，則因為使用了 pop_front 與 push_front 的關係而無法執行。讀者請留意，pop_back 與 pop_front 兩成員函式只是單純地去除元素，並將容器的元素個數減一，並無法得知所去除的元素資料，若要觀察容器的前後端元素的數值，則要使用 front 與 back 兩成員函式。

與一般的物件一樣，所有的容器物件在離開了其存在的領域[43]後就會自動執行容器的解構函式而消失，並不須要使用者作額外的處理，有關於各種容器的使用差別，將留待下兩章再予以詳細介紹。

17.3 迭代器

為了讀取或是儲存元素到容器內，STL 特別設定了一種類似指標的迭代器 (iterator) 可以間接地存取容器內的資料。雖然有些容器，例如：向量陣列與佇列陣列，可以利用下標運算子，即 operator[]，來存取其內的元素。但是很不幸的，並不是所有的容器都可使用下標運算子。舉例來說，如果容器內元素間的儲存方式並不是緊鄰在一起，而是分散的，則利用下標運算子來存取元素就會出現問題。因此為了統一存取容器元素方式， STL 特別設計了一種資料型別，即迭代器 (iterator)，專門用來存取於不同容器內的元素資料。

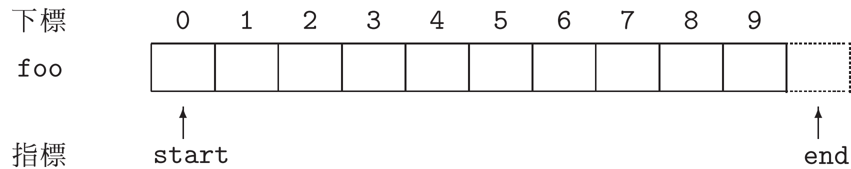
指標與迭代器

由於迭代器的運作方式與傳統指標類似，為了方便對照起見，這裡先來看看以傳統指標方式來產生 10 個介於 0 到 99 之間的亂數，並將其存入一陣列中的例子。

```
int  foo[10] ;          // 可儲存 10 個元素的整數陣列
int *start = foo ;      // start 指到第一個元素，即 foo[0]
int *end   = foo+10 ;   // end   指到第十個元素後的下一個整數位址
int *ptr ;              // ptr   為整數指標

for ( ptr = start ; ptr != end ; ++ptr )  *ptr = rand() % 100 ;
```

在程式碼中，我們先定義兩個指標，start 與 end，分別指到陣列 foo 的第一個位址與末尾元素後的下一個位址，這裡指標 end 並沒有指向任何陣列元素，指標與陣列

的關係可用下圖表示：

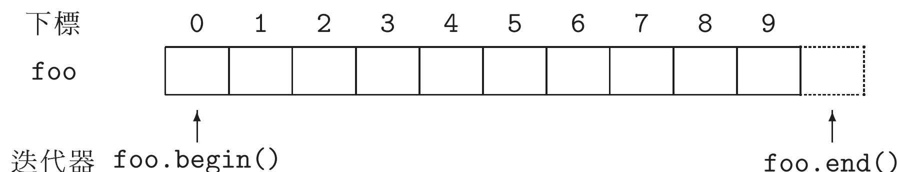


在迴圈中，我們利用 ptr 指標，由起始指標起，ptr = start，間接地將亂數一一的存入陣列 foo 之中，此迴圈將重複執行到 ptr 指標與 end 指標指到同樣的位址後跳離迴圈，在此範例中，for 迴圈內的判斷敘述，ptr != end 是之前較少使用的方式，指標 end 並非指向陣列的末尾元素，而是其後一個整數空間的位址，很明顯的，此位址已經不在陣列範圍內。當指標 ptr 與指標 end 相等時，即代表 ptr 已經指離陣列 foo 的範圍了，因此要跳離迴圈，這裡須留意一點，此時指標 ptr 所指的位址並不屬於陣列所佔有的記憶空間，很有可能是其它程式所使用的記憶空間，如果我們要透過 ptr 指標儲存資料到此位址上，則會造成執行上的錯誤。

若採用與上述的指標運作的方式，但改用迭代器來指向容器元素，則程式可改為：

```
vector<int>            foo(10) ;   // 可儲存 10 個整數的向量陣列
vector<int>::iterator  iter    ;   // 指向整數向量陣列元素的迭代器

for ( iter = foo.begin() ; iter != foo.end() ; ++iter )
    *iter = rand() % 100 ;
```

在程式碼中，vector<int>::iterator 為定義在向量容器 vector<int> 公共區內的迭代器資料型別，我們可以用它來定義物件 iter，此物件可以視為 vector<int> 的專屬指標，也就是專門用來指向 vector<int> 內的元素。在很多種的情況下，我們經常會利用迭代器指向容器內的首位元素，然後依次前進直到迭代器指向末尾元素後的一個位址終止。因此在每個 STL 的容器中，都有兩個函式可以分別回傳容器內首位元素的位址與末尾元素後一個位址。即 begin() 與 end()，其作用與之前的 start 與 end 傳統指標相當，因此兩個程式的使用方式相似。這裡的容器與迭代器可用下圖表示：



由圖所示，begin() 所指的位址是在容器內，而 end() 所指的位址卻不在容器內。

迭代器類型	用途與可以執行的運算子
input iterator 輸入迭代器	只可以用來取得資料，迭代器僅能以單向一步一步方式向前移動 x = *i , ++i , i++
output iterator 輸出迭代器	只可以用來存入資料，迭代器移動方式如上 *i = x , ++i , i++
forward iterator 向前迭代器	可以用來取得或儲存資料，迭代器移動方式如上 *i = x , x = *i , ++i , i++
bidirectional iterator 雙向迭代器	可用來取得或儲存資料，但迭代器可以往前後方一步步方式移動 *i = x , x = *i , ++i , i++ , --i , i--
random-access iterator 隨意存取迭代器	可用來取得或儲存資料，且迭代器可以自由跳躍到其它位址 *i = x , x = *i , ++i , i++ , --i , i-- , i + n , i - n , i += n , i -= n , i < j , i <= j , i > j , i >= j

表格 17.3: 迭代器類型與其用途

五種迭代器類型

STL 在每個容器內都有定義其專屬的迭代器，若依迭代器功能區分，迭代器類別總共有五種類型，如表格 17.3。在此表中 i，j 為同類型的迭代器，x 為資料數值， n 為移動的資料距離。所有的迭代器都可以執行指定動作，例如：i = j，除了輸出迭代器外，所有迭代器都可以執行 i == j，i != j 兩個邏輯判斷動作。

所有的容器都有其獨特的資料管理機制，因此相對的也有其適用的迭代器類型。例如：向量陣列與佇列陣列可以使用隨意存取類型的迭代器，但串列， (複)集合與(複)映射等容器只能使用雙向迭代器類型。不同的迭代器類型，其所能執行的功能也不盡相同。例如：雙向迭代器 (bidirectional iterator) 只能往前或往後一步一步移動，但隨意存取迭代器 (random-access iterator) 則可以利用加減法運算子，立即將迭代器的指向移動到其它位址。如此一來，如果在某個泛型函式內使用隨意存取迭代器來存取容器內的資料，則就表示此泛型函式並不能取用儲存在串列，集合等容器內的元素，這也表示，泛型函式不能適用於串列，集合等容器。

一般來說，迭代器的用法與傳統的指標相當，雖然在實作上卻不完全一樣，例如 operator++ 在迭代器的作用是讓迭代器物件指到容器下一個元素的位址，這個元素可能與前一個元素並不相鄰，但如果是傳統指標，則 operator++ 是指到下一個相鄰的位址，以下我們利用迭代器將 foo 容器的所有元素印出，

```
vector<int>::iterator  iter ;
for ( iter = foo.begin() ; iter != foo.end() ; ++iter )
```

```
        cout << '>' << *iter << ' ' ;
```

以上 foo 為一向量陣列，也可以使用下標運算子，operator[]，來取得各下標的元素資料，例如：

```
    for ( int i = 0 ; i < foo.size() ; ++i )
        cout << '>' << foo[i] << ' ' ;
```

但這樣的用法卻不適用於串列容器，主要原因在串列容器類並無定義下標成員函式，然而若是使用迭代器的方式，則三種序列容器都可適用的，例如：

```
    list<int>   foo ;   // 定義一串列物件
    ...
    list<int>::iterator  iter ;
    for ( iter = foo.begin() ; iter != foo.end() ; ++iter )
        cout << '>' << *iter << ' ' ;
```

但如果是使用下標運算子則僅能用在向量或者是佇列容器上，因此下標運算子在使用上是受到限制。

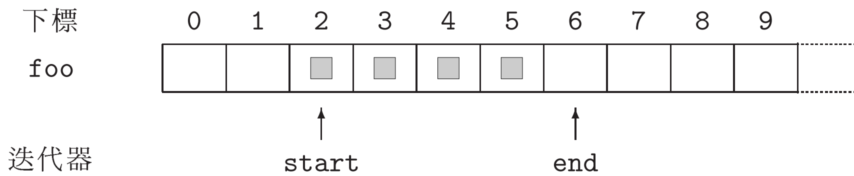
讀者須留意，若 iter 為指向某容器的迭代器，則 ++iter 與 iter = iter + 1 兩者的運作方式是有所差別的。前者讓迭代器移向容器的下一個元素位址，後者則讓迭代器跳到下一個緊鄰元素的位址。使用後者的前題是迭代器所指向的容器其元素間為緊鄰在一起，這個前題對串列或是集合容器而言是無法滿足的。因此若迴圈內的遞增方式改寫成以下型式：

```
    for ( iter = foo.begin() ; iter != foo.end() ; iter = iter + 1 )
        cout << '>' << *iter << ' ' ;
```

則這一小小的改變已無形中排除掉 foo 為串列容器的可能性。

設定容器範圍

在 STL 中，若要界定一段容器元素的範圍 (range)，須要透過兩個迭代器物件的協助。第一個迭代器指向起始位址，而第二個迭代器指著末尾元素的下一個位址，不是末尾元素的位址，這樣設計的好處是對向量或者是佇列陣列而言，在範圍內元素的個數剛好是兩者之差值。例如：以下兩個迭代器 start 與 end 所設定的範圍為有陰影的元素，對元素緊鄰的容器而言，其個數剛好為 end - start 的差值，如果用數學符號表示，相當於 [start,end)，也就是範圍包含 start 所指的元素但不包含 end 所指的元素，如果兩個迭代器相同的話，則代表範圍內的元素個數為零。請留意，若容器內的元素是以分散的方式儲存，例如：串列容器與所有的關聯容器，則 end 與 start 的差值就不能代表範圍內元素的個數。



在使用上，我們通常會用迴圈來迭代處理在範圍內的資料，

```
int  n ;
list<int>  foo ;                            // 產生一串列物件
for ( n = 1 ; n < 10 ; ++n )                // 儲存 1 到 9
    foo.push_back(n) ;

list<int>::iterator  i ;                    // 定義暫時迭代器
list<int>::iterator  start = foo.begin() ;  // 設定範圍 [start,end)
list<int>::iterator  end  = foo.end() ;
for ( i = start ; i != end ; ++i )          // 列印
    cout << *i << ' ' ;
```

當迭代器物件被當成參數傳入函式中時，也可以使用常數的方式傳入，這時就須要使用常數迭代器。常數迭代器物件須由常數型態的迭代器類別產生，這些常數迭代器類別的名稱與原來的名稱僅有 const 的差別，例如：

```
vector<int>::const_iterator  i1 ;
deque<char>::const_iterator  i2 ;
list<float>::const_iterator  i3 ;
```

這類的常數迭代器與常用的常數指標的作用相同，都不能改變指向的資料值，但是可以改變迭代器所指的位址，例如：

```
const vector<int>            foo(5,3) ;  // 儲存 5 個 3 的向量陣列
vector<int>::const_iterator  i ;         // 常數迭代器

for ( i = foo.begin() ; i != foo.end() ; ++i )  // 列印所有元素
    cout << *i << ' ' ;
```

以上若不使用常數型的迭代器，則會造成編譯錯誤。 STL 所定義的迭代器也是物件的一種，因此當離開了其所在的領域後，迭代器物件也會自動執行其解構函式而消失。

列印容器資料

STL 在各個容器內都有定義其專屬的迭代器，以下範例利用這個性質設計一樣板函式來列印容器內的所有元素資料：

```
template <class  T>
void  print_container( const T& foo ) {
```

```
        typename  T::const_iterator  iter ;
        for ( iter = foo.begin() ; iter != foo.end() ; ++iter )
            cout << *iter << ' ' ;
    }
    ...
    vector<int>  foo ;   // 整數向量陣列容器
    list<char>   bar ;   // 字元串列容器
    ...
    print_container( foo ) ; // 列印 foo 向量，樣板參數 T 為 vector<int>
    print_container( bar ) ; // 列印 bar 串列，樣板參數 T 為 list<char>
```

如此，不管是哪一種容器都可以用同樣的方式列印。讀者須留意，在定義 iter 時，我們須要在敘述之前加上一保留字 typename，用以表示其後的 T::const_iterator 是一種資料型別，而 iter 為此資料型別的一個物件。如果在這裡的泛型參數 T 之前不加入 typename，例如，可能會遇到以下的情況發生：

```
template <class T>
void  foo( const T& bar ) {
    T::bar * ptr ;           // 語法不清
    ...
}
```

以上的敘述，編譯器可能將其視為 T::bar 物件與 ptr 物件的乘積或者是 ptr 為定義在 T 類別 (或是名稱空間) 的 bar 資料型別內的指標，使得編譯器無法取捨而產生錯誤。如果在名稱之前加上 typename 則明確告知編譯器 typename 之後的名稱為資料型別名稱 (type name)。

```
typename  T::bar * ptr ;   // ptr 為指向 T::bar 資料型別的指標
```

17.4 泛型演算函式

STL 內定義的泛型演算函式 (generic algorithm) 是指著函式的執行與使用的資料型別無關，由第十四章所述可知這當然與樣板函式有所相關。事實上，在 STL 內定義的所有泛型演算函式都是建築在樣板函式的基礎之上，因此我們將由樣板函式來說明其使用方式。

樣板函式

為了讓程式碼可以對不同容器都能提供同樣的功能，C++ 特別利用樣板函式來達到這個目的。舉例來說，我們可設計一個樣板函式來列印一容器在某個範圍內的資料：

```
template <class T>
void  print_items( T start , T end ) {
```

```
        for ( T iter = start ; iter != end ; ++iter )
            cout << *iter << ’ ’ ;
    }
```

而使用的程式碼為：

```
vector<int>            a(10,3) ;      // 儲存 10 個 3 的向量陣列
vector<int>::iterator  itr1 , itr2 ;  // 定義 vector 的迭代器物件
...                                   // 設定 a, itr1, itr2 的值
print_items( itr1 , itr2 ) ;
```

在此函式中，型別樣板參數 T 為 vector<int>::iterator，程式碼所列印的範圍是由 iter1 所指的位址開始一直到 iter2 的前一個位址，函式並沒有設定使用的型別，因此也可以列印儲存於其它容器某範圍內的元素，例如：

```
deque<int>            b(10) ;        // 須要加入 deque 標頭檔
deque<int>::iterator  itr3 , itr4 ;  // 定義 deque 的迭代器物件
...                                  // 設定 b, itr3, itr4 的值
print_items( itr3 , itr4 ) ;
```

或者是串列類別：

```
list<int>            c ;             // 須要加入 list 標頭檔
list<int>::iterator  itr5 , itr6 ;   // 定義 list 的迭代器物件
...                                  // 設定 c, itr5, itr6 的值
print_items( itr5 , itr6 ) ;
```

由此可知，透過迭代器與泛型演算函式的結合，同樣的函式可以對不同的容器執行相同的動作，使得一個普通的函式具有普遍的 (generic) 適用性。

泛型演算函式

在 C^{++} 的 STL 內定義了許多泛型演算函式用來處理各種不同的演算問題，大部份的泛型函式是定義在 algorithm 的標頭檔內，使用時須要將此標頭檔加入。在範例中，我們使用了排序泛型演算函式 sort 用來對容器物件的某一段範圍作排序的動作，例如：以下只對由 foo[3] 開始的五個元素作由小到大的排序動作。

```
vector<int>   foo(10) ;                     // 10 個向量陣列
for ( int i = 0 ; i < 10 ; ++i )            // 存入 10 個亂數
    foo[i] = rand() ;
sort( foo.begin()+3 , foo.begin()+8 ) ;     // 重排 foo[3]..foo[7]
```

同樣的，我們可以利用取代函式 replace 對容器物件內的某個範圍作元素取代的動作，例如：以下的程式中首先定義一個含有 5 個 ’a’ 字元的佇列陣列，接下來我們將 [foo.begin()+3,foo.end()) 範圍內的元素由字元 ’a’ 取代為 ’b’。

```
deque<char>   foo(5,’a’) ;                       // foo = a a a a a
replace( foo.begin()+3 , foo.end() ,             // foo = a a a b b
```

```
        'a' , 'b' ) ;
```

傳統陣列與 C^{++} 字串也可以用在 STL 所定義的泛型演算函式之中，使用方式與之前類似，例如：

```
int  a[5] = { 3 , 8 , 6 , 2 , 1 } ;            // a = 3 8 6 2 1
sort( a , a+5 ) ;                              // a = 1 2 3 6 8

int  b[5] = { 2 , 2 , 2 , 2 , 2 } ;            // b = 2 2 2 2 2
replace( b+1 , b+4 , 2 , 8 ) ;                 // b = 2 8 8 8 2

char c[11] = "2000/01/01" ;                    // c = 2001/01/01
replace( c , c+11 , '/' , '-' ) ;              // c = 2001-01-01

string  d  = "aaaaa" ;                         // d = aaaaa
replace( d.begin()+3 , d.end() , 'a' , 'b' ) ; // d = aaabb
```

以上範例說明了所謂的泛型演算函式 (generic algorithm) 名稱的意思，即是函式內的程式碼與所處理的資料型別無關，而達成這個目標的主要工具就是迭代器。

雖然 STL 的泛型演算函式在設計上已盡量考慮到接納所有不同容器，但仍有一些泛型演算函式由於函式所使用演算法則 (algorithm) 的限制並無法適用到所有的容器上，例如：這裡所使用的 `sort` 函式並無法對串列容器 (`list`) 排序。

如果要找尋某個元素是否在某範圍內時，可以用搜尋演算函式 `find`，使用者可以設定在容器的某個範圍內找尋元素，若找到，則回傳一指向元素的迭代器，若無，則回傳給定範圍內的末尾迭代器，例如：

```
vector<int>   foo  ;
vector<int>::iterator   iter ;

foo.push_back(7) ;  foo.push_back(8) ;  foo.push_back(2) ;
foo.push_back(9) ;  foo.push_back(1) ;  foo.push_back(3) ;

if ( find( foo.begin() , foo.end() , 3 ) == foo.end() )
    cout << "3 is not found" << endl ;
else
    cout << "3 is found" << endl ;
```

`find` 搜尋演算函式的程式碼基本型式如下：

```
template <class InputIter，class T>
InputIter find( InputIter start , InputIter end , const T& val ) {
    while ( start != end && *start != val ) ++start ;
    return  start ;
}
```

基本上，搜尋演算函式僅是在範圍內一個接著一個元素做比較，且只須使用到最簡單的輸入迭代器，因此 `find` 搜尋函式可以用於搜尋 STL 提供的所有容器資料。

17.5 函式物件類別

函式物件類別 (functor class) 為一類別 (class) 或者是結構 (struct)，在其類別或結構內有定義函式成員函式 (function call operator)，也就是 operator()。此類別或結構內可以不須要有任何其它資料成員或者是成員函式，雖然函式物件類別的定義如此簡單，但出乎意外的是其應用卻是相當的廣泛。

定義函式物件類別

要在類別或者是結構資料型別內覆載函式運算子，operator()，可以使用以下方式：

```
template <class T>
struct  Small {
    bool  operator() ( const T& a , const T& b ) const {
        return  a < b ;
    }
} ;
```

這裡 Small 結構是用來比較輸入函式運算子的第一個參數值是否比第二個參數值小，如果是，則回傳真，否則回傳假值。函式運算子不見得須要兩個參數，也可以使用一個參數即可，例如：

```
// 平方函式物件類別 : 計算 T 型別物件的平方
template <class T>
class  Square {
    T  operator() ( const T& a ) const { return  a * a ; }
} ;

// 立方函式物件類別 : 計算 T 型別物件的立方
template <class T>
struct  Cubic {
    T  operator() ( const T& a ) const { return a * a * a ; }
} ;
```

這些函式物件類別內只是單純的覆載 operator()，並沒有定義任何的資料成員。由於此種類別大多不須要其它的成員函式，若要產生一個函式物件類別物件可以使用：

```
Square<int>   foo ;              // 使用備用建構函式來產生一個物件
cout << foo(2)  << endl ;        // foo 物件執行呼叫函式
```

以上程式碼中的 foo(2) 事實上為 foo.operator()(2) 的簡寫。我們將函式物件類別 (functor class) 所產生的物件稱為函式物件 (function object 或者 functor)。

若物件名稱不是很重要，則也可以用下列方式來達到同樣的目的，如此可以省去一個物件名稱，例如：

```
cout << Square<int>()(3) << endl ;  // 使用備用建構函式後執行呼叫成員函式
```

這裡的 Square<int>() 為使用 Square<int> 備用建構函式所產生的物件，而 Square<int>()(3) 則為 Square<int>().operator()(3) 的簡寫。

函式物件

函式物件類別最有利的使用情況是與泛型演算函式結合在一起使用，例如，若使用以上的 Square 與 Cubic 兩個函式物件於以下的樣板函式內：

```
template <class S，class T>
void  print_items( S start , S end , T fn ) {
    for ( S  iter = start ; iter != end ; ++iter )
        cout << fn(*iter) << ' ' ;
}
```

在此樣板函式中，我們輸入了一個函式物件 fn，當程式執行 fn(*iter) 時，程式相當於執行了 fn.operator()(*iter) 敘述，因此以下的程式碼可以依輸入的函式物件的分別計算出不同的結果：

```
vector<int>  foo(3,2) ;  // 三個 2

print_items( foo.begin() , foo.end() ,          // 印出 : 4 4 4
             Square<int>() ) ;
print_items( foo.begin() , foo.end() ,          // 印出 : 8 8 8
             Cubic<int>() )  ;
```

這裡所定義的泛型演算函式 print_items 也可以處理傳統陣列，

```
int  foo[4] = { -1 , 2 , 3 , 1 } ;
print_items( foo   , foo+4 , Square<int>() ) ;   // 印出 : 1 4 9 1
print_items( foo+1 , foo+3 , Cubic<int>()  ) ;   // 印出 : 8 27
```

同樣的，如果要列印僅滿足某個條件的資料，則可以設定：

```
template <class S，class T>
void  print_items( S iter1 , S iter2 , T fn ) {
    for ( S iter = iter1 ; iter != iter2 ; ++iter )
        if ( fn(*iter) ) cout << *iter << ' ' ;
}
```

由於 fn(*iter) 是在 if 條件式中，此時函式物件類別的函式運算子須回傳一布林值 (bool)，例如，以下的 Odd<T> 函式物件類別用來判斷參數資料是否為奇數，Greater<T> 函式物件類別則用來判斷參數資料是否大於某設定數，

```
template <class T>
struct  Odd {
    bool  operator() ( const T& foo ) const {
```

```
        return ( foo % 2 ? true : false ) ;
    }
} ;

template <class T>
class  Greater {
  private :
    T  num ;
  public :
    Greater( const T& n = 0 ) : num(n) {} ;
    bool  operator() ( const T& foo ) const {
        return  foo > num  ;
    }
} ;
```

在第二個函式物件類別中，我們特別利用了一個資料成員用來儲存所要比較的數字，這個數字須在建構函式內設定，因此 `Greater<int>(30)` 代表著使用 `Greater<int>` 的參數建構函式且以參數值為 30 來產生一個函式物件，而 `Greater<int>(30)(9)` 代表物件要執行 `Greater<int>(30).operator()(9)`，以下的程式碼片段，我們可以在列印函式 `print_items` 中作更進一步的列印處理，例如：

```
// 產生 10 個在 0 到 99 之間的亂數
vector<int>  foo(10) ;
for ( int i = 0 ; i < 10 ; ++i ) foo[i] = rand() % 100 ;

// 列印奇數
print_items( foo.begin() , foo.end() , Odd<int>() ) ;

// 列印大於 50 的數字
print_items( foo.begin() , foo.end() , Greater<int>(50) ) ;
```

同樣的方式也可以應用於傳統陣列

```
int  foo[5] = { 3 , 5 , 8 , 7 , 4 } ;

// 列印大於 5 的整數
print_items( foo , foo+5 , Greater<int>(5) ) ;
```

樣板函式與函式物件

在 C++ 的設計中，(樣板) 函式與函式物件都可以當成另一個樣板函式的參數，這樣的處理可以增加函式運用的自由度。例如，如果將平方樣板函式，立方函式與絕對值樣板函式結構定義為：

```
// 平方樣板函式
template <class T>
T square( const T& i ) {  return ( i * i  ) ; }
```

```
// 立方函式
int cubic ( int i ) { return i * i * i ; }

// 絕對值樣板函式結構
template <class T>
struct Abs {
    T operator() ( const T& i ) { return ( i >= 0 ? i : -i ) ; }
} ;
```

然後再定義一使用樣板函式為

```
template <class Function , class T>
T  compute ( Function fn , T item ) {  return fn(item) ; }
```

則我們可以使用以下的程式碼

```
cout << compute( square<int> ,  2 )  ;  // 印出 4
cout << compute( cubic ,        2 )  ;  // 印出 8
cout << compute( Abs<int>()  , -2 )  ;  // 印出 2
```

在此的 `compute` 函式參數列的 `fn` 可以是一個函式 `square<int>` , `cubic`，或者是函式物件 `Abs<int>()`，前者是透過函式指標[200] (function pointer) 的方式將函式當成參數送入另一個函式處理，樣板函式指標可參考第 14.3 節的說明。

在 STL 所提供的 `sort` 函式中也可以選擇性的輸入一函式 (function) 或者函式物件類別 (function object) 當成第三個參數，用以設定排序的標準。例如，如果要以絕對值的大小由大到小排序，則可以先設計：

```
int  abs( int a ) { return  a >= 0 ? a : -a ; }

struct  large_abs {
    bool  operator() ( int a , int b ) const {
        return  abs(a) > abs(b) ;
    }
} ;
```

然後再使用

```
int  foo[5] = { 3 , -5 , 8 , -7 , 4 } ;

// 依絕對值大小由大排到小排序
sort( foo , foo+5 , large_abs() ) ;  // 排序後 foo 為 8 -7 -5 4 3
```

同樣地，如果一個簡單分數結構與其對應的輸出運算子定義為

```
struct  Fraction {
    unsigned  int num , den ;  // num : 分子 , den : 分母
    Fraction( unsigned n , unsigned d ) : num(n) , den(d) {} ;
} ;
```

```
ostream&  operator<< ( ostream& out , const Fraction& foo ) {
    return  out << foo.num << '/' << foo.den ;
}
```

且將分數比較大小的函式物件類別定義為，

```
struct  Bigger {
    // 比較分數 x 是否比 分數 y 大
    bool  operator() ( const Fraction& x , const Fraction& y ) {
        return  x.num*y.den > x.den*y.num  ;
    }
} ;
```

則以下可以將分數陣列由大到小排序後印出 7/3 7/5 2/3 3/5

```
Fraction   foo[4] = { Fraction(2,3) , Fraction(3,5) ,
                      Fraction(7,5) , Fraction(7,3) } ;

sort( foo , foo+4 , Bigger() ) ;
for ( int i = 0 ; i < 4 ; ++i ) cout << foo[i] << ' ' ;
```

真假判斷式

如果一函式或者是在函式物件類別內的函式運算子回傳一真假值，則此函式被稱為(真假)判斷式 (predicate)，如果判斷式只須要一個參數，則稱為單元判斷式 (unary predicate)，如果須要兩個參數，則稱為雙元判斷式 (binary predicate)。例如：

```
// 單元判斷式 : 判別整數 i 是否大於零
bool  positive( int i ) { return i > 0 ; }

// 雙元判斷式 : 判別整數 a 是否較整數 b 為大
struct  Larger {
    bool  operator() ( int a , int b ) { return a > b ; }
} ;
```

雖然以上判斷式很簡單，但真假判斷式卻在 STL 中伴演相當重要的角色。

STL 預設的函式物件類別

STL 在標頭檔 `functional` 中定義了許多常用的函式物件類別，其中大部份是雙元函式，但也有一些是單元函式，如表格 17.4。 表中的 `T` 為使用的型別，而 `a` 與 `b` 為此型別的物件資料。這些函式物件類別的程式都很簡單，例如，以下為 `less` 函式物件類別的基本型式：

函式物件類別	用途	函式物件類別	用途
equal_to<T>	a == b	logical_not<T>	!a
not_equal_to<T>	a != b	negate<T>	-a
less<T>	a < b	plus<T>	a + b
less_equal<T>	a <= b	minus<T>	a - b
greater<T>	a > b	multiplies<T>	a * b
greater_equal<T>	a >= b	divides<T>	a / b
logical_and<T>	a && b	modulus<T>	a % b
logical_or<T>	a || b		

表格 17.4: C++ 預設的函式物件類別與其用途

```
// less 函式物件類別 : 比較是否 a 比 b 小
template <class T>
struct  less {
    bool operator() ( const T& a , const T& b ) const {
        return a < b ;
    }
} ;
```

現在以一個簡單的例子示範如何使用這些預設的函式物件類別，假設我們設計了一個 compute 函式專門用來處理陣列元素之間的運算：

```
#include <iostream>
#include <functional>

using namespace std ;

// 計算兩迭代器之間的元素運算計算值
template <class Function , class Iterator , class T>
T  compute( Iterator start , Iterator end , T value ,
            Function fn ) {
    for ( ; start != end ; ++start ) value = fn(value,*start) ;
    return value ;
}

int main() {
    int  a[5] = { 24 , 2 , 1 , 3 , 1 } ;

    // 計算陣列元素相加 : 0 + 24 + 2 + 1 + 3 + 1        => 印出: 31
    cout << compute( a , a+5 , 0          , plus<int>()       )
         << endl ;

    // 計算陣列元素相減 : (2*24) - 24 - 2 - 1 - 3 - 1  => 印出: 17
```

```
        cout << compute( a , a+5 , 2*a[0]    , minus<int>()      )
             << endl ;

        // 計算陣列元素相乘 : 1 * 24 * 2 * 1 * 3 * 1       => 印出: 144
        cout << compute( a , a+5 , 1         , multiplies<int>() )
             << endl ;

        // 計算陣列元素相除 : (24*24) / 24 / 2 / 1 / 3 / 1 => 印出: 4
        cout << compute( a , a+5 , a[0]*a[0] , divides<int>()    )
             << endl ;

        return 0 ;
    }
```

這裡的 compute 泛型函式內只有一個簡單的迴圈，在迭代器所設定的範圍內重複地執行 fn 函式， compute 函式的第三個參數 value 除了告知 compute 函式所要運算的型別外，也附帶當為計算過程的初始值。舉例來說，當我們要計算 a[0] - a[1] - ... - a[4] 時，在但在程式內則是執行 value - a[0] - a[1] - ... - a[4]，因此兩者若要相等，則 value 的起始值要設為陣列第一個元素的兩倍數值，即 2*a[0]。同理，可以知道，為何相加時要設為 0，相乘時要設為 1，而相除時則要定為第一個元素的平方值。

同樣的，如果要設計一泛型演算函式來計算兩個等長容器範圍的元素內積和，即

$$a \cdot b = a_0 * b_0 + a_1 * b_1 + \cdots + a_{n-1} * b_{n-1}$$

這裡的 a 與 b 分別代表兩個容器內的某段範圍，則內積函式可以設計為

```
    template<class Iter1 , class Iter2 , class T>
    T  inner_product( Iter1 i1 , Iter1 i2 , Iter2 j1 , Iter2 j2 ,
                      T val ) {
        for ( ; i1 != i2 ; ++i1 , ++j1 )
            val = plus<T>()(val,multiplies<T>()(*i1,*j1)) ;
        return val ;
    }
```

以上的 val 為計算時輸入的初始值，由程式中可以看出，在此輸入的 j2 迭代器由於並沒有在程式中使用，所以在程式設計中常常將之省略。此外在 plus<T>() 與 multiplies<T>() 的 () 並不能省略，此兩者分別代表函式物件。事實上，迴圈內的敘述是以下式子的簡寫：

```
    val = plus<T>().operator()( val ,
                                multiplies<T>().operator()(*i1,*j1) ) ;
```

由於內積函式使用了兩段迭代器所設定的範圍，但並沒有規定須屬於相同的容器，此兩段範圍可以分別屬於不同容器上的片段元素，但也可以是相同的，例如：

```
    int       foo[5] = { 2 , 3 , 1 , 2 , 5 } ;    // foo = 2 3 1 2 5
```

```
list<int>  bar( foo+2 , foo+5 ) ;              // bar = 1 2 5

// 計算 (3,1,2)*(1,2,5) 的內積和，結果為 15 ( = 3*1 + 1*2 + 2*5 )
cout << inner_product( foo+1 , foo+4 , bar.begin() , bar.end() , 0 )
     << endl ;

// 計算 2*2 + 3*3 + 1*1 + 2*2 + 5*5 的數值 ，結果為 43
cout << inner_product( foo , foo+5 , foo , foo+5 , 0 ) << endl ;
```

這裡的內積函式計算的起始值被設為零，由最後一個參數 val 傳入。

束縛函式與否定函式

若有一個須要兩個參數的數學函式設定為 $f(x,y) = x^2 - 2xy + y^2$，今若固定了其中一個參數，例如：$y = c$，c 為某一固定數值，則函式 $f(x,y)$ 就變成了一個參數的函式 $g(x) = f(x,c) = x^2 - 2cx + c^2$。採用同樣的方式，我們也可以對函式物件類別內的雙元函式運算子固定任何一個參數，使得其變成單元函式運算子。為達到這種目的，STL 設計了兩個束縛函式 (binder)，即 bind1st 與 bind2nd， 可以分別對雙元函式運算子的第一個參數或第二個參數的值做限制，經過「參數束縛」後的雙元函式就相當於單元函式一般，也就是說，束縛函式的回傳值是一個單元函式。舉例來說：函式物件類別 less<T>()(a,b) 是用來比較資料 a 是否比資料 b 為小。如果要將第二個參數限制為 100，則可以用 bind2nd(less<T>() , 100)，例如，在以下的泛型函式 count_no 是用來計算在某段容器範圍內滿足某個輸入條件的個數：

```
template <class Iter，class Function>
int  count_no( Iter iter1 , Iter iter2 , Function  fn) {
    int  no = 0 ;
    for ( ; iter1 != iter2 ; ++iter1 ) if ( fn(*iter1) ) ++no ;
    return  no ;
}
...
int  a[10] = { 3 , 4 , 10 , 7 , 8 , 9 , 21 , 9 , 5 , 12 } ;

// 計算陣列元素 x 不大於數字 8 的個數
// 設定 greater<int> 的第一個參數為 8，即滿足 8 > x 的 x 個數
// 印出 : 4
cout << count_no( a , a+10 , bind1st( greater<int>() , 8 ) )
     << endl ;

// 計算陣列元素 x 不被 5 整除的個數
// 設定 modulus<int> 第二個參數為 5，即滿足 x % 5 != 0 的 x 個數
// 印出 : 8
cout << count_no( a , a+10 , bind2nd( modulus<int>() , 5 ) )
     << endl ;
```

在最後一個敘述中，modulus<int> 是用來計算輸入的兩整數間相除後的餘數，當回傳的值不為 0 時，則代表前面的整數無法整除後面的整數，且其所代表的真假值則被視為真，因此 bind2nd(modulus<int>(),5) 就表示不被 5 整除的數字。

以上方式是用來找出滿足某一條件的元素個數，如果要找出同時滿足兩個條件的元素個數則可以定義以下的泛型函式：

```
template <class Iter，class Fn1，class Fn2，class Fn3>
int  count_no( Iter i1 , Iter i2 , Fn3 logical_fn ,
               Fn2 fn2 , Fn3 fn3 ) {
    int  c = 0 ;
    for ( ; i1 != i2 ; ++i1 )
        if ( logical_fn( fn2(*i1) , fn3(*i1) ) ) ++c ;
    return c ;
}
```

使用上可以用

```
// 計算介於 (5,15) 之間元素的個數，即滿足 ( 5 < x ) && ( x < 15 )
// 印出 : 6
cout << count_no( a , a+10 , logical_and<bool>() ,
                  bind1st(less<int>(),5) ,
                  bind2nd(less<int>(),15) ) << endl ;
```

此外 STL 還定義了一種否定函式 (negator)，not1 與 not2 可以分別用來對單元與雙元的真假判斷式物件 (predicate object) 作否定的動作，也就是將輸入真假判斷式的真假值對調。舉例來說， a 大於或者等於 b 的關係式與 a 不小於 b 的說法是一樣，若使用 STL 的否定函式，則可以寫成 not2(less<T>())，其作用與 greater_equal<T>() 相當，這裡使用的 not2 是針對雙元判斷式物件，若是輸入的為單元判斷式則須使用 not1。使用以上的陣列，以下為例：

```
// 計算可以被 5 整除的元素個數，即滿足 ! ( x % 5 != 0 ) 的元素個數
// 印出 : 2
cout << count_no( a , a+10 , not1(bind2nd(modulus<int>(),5)) )
     << endl ;

// 計算可以被 3 整除但不被 2 整除的元素個數，
// 即滿足 ( x % 3 == 0 ) && ( x % 2 != 0 )
// 印出 : 4
cout << count_no( a , a+10 , logical_and<bool>() ,
                  not1(bind2nd(modulus<int>(),3)) ,
                  bind2nd(modulus<int>(),2) ) << endl ;
```

這裡之所以使用 not1 不是 not2 的主要原因在於束縛函式所回傳出來的函式一定為單元判斷式。 not2 可以使用在須要雙元判斷式的泛型演算函式中，例如：

```
int  a[10] = { 3 , 4 , 10 , 7 , 8 , 9 , 21 , 9 , 5 , 12 } ;

// 排序 : 由大到小 ,  a 陣列變成 : 21 12 10 9 9 8 7 5 4 3
```

```
sort( a , a+10 , not2(less<int>()) ) ;
```

束縛函式與否定函式都定義在 `functional` 標頭檔。由於這兩種類型的函式都可以調整輸入的函式物件的功能，我們將此兩種函式稱為函式物件轉換器 (functor adaptor)。

17.6 應用範例

以下我們利用幾個簡單範例來示範 STL 程式使用方式。

堆疊類別

在第九章的作業習題[279]與第十四章的範例[474]中，我們介紹了堆疊資料處理機制，之前的處理方式都是以傳統固定陣列來儲存元素，當元素個數已經存滿時，堆疊就不能再加以儲存元素，這在使用上會造成相當的不方便，尤其在許多應用問題，欲處理的元素多寡常常是不可預知的，固定長度的堆疊時常會使得程式受到一些限制。為去除堆疊長度的限制，我們可以利用向量陣列來解決這個問題，每當加入或者是取出一個元素時，向量陣列長度會隨之自然調整，向量陣列可儲存的最大長度是受到計算機記憶空間的容量影響，而不是程式碼的問題。在程式中，每加入一個元素到堆疊中時，我們使用 `push_back` 成員函式將元素加入到向量陣列的最末端，當要由堆疊中去除元素時，則使用 `pop_back` 成員函式，如果要查看堆疊中的最頂端元素則使用 `back()` 成員函式。

堆疊類別：向量陣列　　`stack_by_vector.cc`

```
01   #include <iostream>
02   #include <vector>
03   #include <cassert>
04
05   using  namespace  std ;
06
07   // 堆疊類別
08   template  <class  T>
09   class  Stack {
10
11     private :
12
13       vector<T>   data   ;  // 元素陣列
14
15     public :
16
17       void pop() ;                                // 從堆疊中取出元素
18       void push( const T& item ) ;                // 將元素放入堆疊中
19       T    top() const { return  data.back() ; } // 告知最上一個元素
20       int  size() const { return data.size() ; } // 告知堆疊元素大小
21
22       template <class S>
```

```
23      friend  ostream&  operator << ( ostream&  , const Stack<S>& ) ;
24
25  } ;
26
27  // 從堆疊中取出元素
28  template <class T>
29  void  Stack<T>::pop() {
30      assert( data.size() > 0 ) ;
31      data.pop_back() ;
32  }
33
34  // 將元素放入堆疊中
35  template <class T>
36  void  Stack<T>::push( const T& item ) { data.push_back(item) ; }
37
38  // 列印堆疊物件
39  template <class T>
40  ostream&  operator << ( ostream& out , const Stack<T>& foo ) {
41
42      out << "> 堆疊共有 " << foo.size() << " 個元素" << endl ;
43      out << "  頂端 [ "  ;
44      for ( int i = foo.size()-1 ; i >=0 ; --i )
45          out << foo.data[i] << ' ' ;
46
47      return  out << "] 末端" << endl ;
48
49  }
50
51  int main() {
52
53      Stack<int>  foo ;
54
55      // 加入 1 , 2 , 3 , 8 至 堆疊
56      foo.push(1) ;  foo.push(2) ;  foo.push(3) ;  foo.push(8) ;
57      cout << "> 最頂端元素為 : " << foo.top() << endl ;
58      cout << "> 元素個數為   : " << foo.size() << endl << endl ;
59
60      // 取出兩個元素後，加入 9 , 7 , 5
61      foo.pop() ;  foo.pop() ;  foo.push(9) ;  foo.push(7) ;
62      foo.push(5) ;
63      cout << foo  ;
64
65      return 0 ;
66
67  }
68
```

執行結果

```
01  > 最頂端元素為 : 8
02  > 元素個數為   : 4
03
04  > 堆疊共有 5 個元素
05    頂端 [ 5 7 9 2 1 ] 末端
06
```

向量運算

向量陣列通常可以用來模擬數學的向量 (vector)，在第十章的範例[307]中，我們利用動態記憶空間來分配所須要的元素空間，而在這裡我們利用了向量陣列來管理空間的問題，在程式中，向量物件可以直接使用向量陣列來產生，也可以由輸入的元素讀入，到達到目的，我們特別定義了輸入運算子來設定向量物件的向量陣列，使用的方式為將一整行的數字讀入存成 istringstream 物件，然後再一一用向量陣列的 push_back 成員函式存到陣列末尾。

向量類別：使用向量陣列	math_vector.cc

```
01   #include <iostream>
02   #include <sstream>
03   #include <vector>
04   #include <cassert>
05
06   using namespace std ;
07
08   template <class T>
09   class Vec {
10
11     private :
12
13       vector<T>  dat ;     // 用向量陣列儲存向量的元素
14
15     public :
16
17       Vec() {}
18
19       Vec( const vector<T>& foo ) : dat(foo) {}
20
21       // 向量元素個數
22       int size() const { return dat.size() ; }
23
24       // 下標運算子
25       T&         operator[] ( int i )       {  return dat[i] ; }
26       const  T&  operator[] ( int i ) const {  return dat[i] ; }
27
28       // 定義 += 運算子
29       Vec&  operator-= ( const Vec& a ) {
30           assert( dat.size() == a.size() ) ;
31           for ( int i = 0 ; i < dat.size() ; ++i ) dat[i] -= a.dat[i] ;
32           return  *this ;
33       }
34
35       // 定義 Vec 專用的輸入運算子
36       template <class S>
37       friend  istream&  operator>> ( istream& in , Vec<S>& foo ) {
38           string  line ;
39           getline( in , line ) ;
```

```
40          istringstream  istr(line) ;
41          S  tmp ;
42          while ( istr >> tmp ) foo.dat.push_back(tmp) ;
43          return  in ;
44      }
45
46  };
47
48  // 定義 Vec 專用的輸出運算子
49  template <class S>
50  ostream&  operator<< ( ostream& out , const Vec<S>& foo ) {
51      out << "( " ;
52      for ( int i = 0 ; i < foo.size() ; ++i ) out << foo[i] << ' ' ;
53      return  out << ")" ;
54  }
55
56  // 計算兩向量的加法
57  template <class T>
58  Vec<T>  operator+ ( const Vec<T>& a , const Vec<T>& b ) {
59      assert( a.size() == b.size() ) ;
60
61      // 定義一向量陣列  後將兩向量物件的元素合成到向量陣列中
62      vector<T>  foo(a.size()) ;
63      for ( int i = 0 ; i < foo.size() ; ++i ) foo[i] = a[i] + b[i] ;
64      return  Vec<T>(foo) ;
65  }
66
67  // 計算兩向量的內積和
68  template <class T>
69  T  operator* ( const Vec<T>& a , const Vec<T>& b ) {
70      assert( a.size() == b.size() ) ;
71      T  sum = static_cast<T>(0.) ;
72      for ( int i = 0 ; i < a.size() ; ++i ) sum += a[i] * b[i] ;
73      return  sum ;
74  }
75
76  int  main() {
77
78      Vec<int>  a , b ;
79      cout << "> 分別輸入兩個向量 (元素個數相同) : \n" ;
80      cout << "a = " ;    cin  >> a ;
81      cout << "b = " ;    cin  >> b ;
82
83      cout << "a + b  => " << a + b  << endl ;
84      cout << "a * b  => " << a * b  << endl ;
85      cout << "a -= b => " << (a -= b) << endl ;
86
87      return 0 ;
88
89  }
90
```

執行結果

```
01  > 分別輸入兩個向量 (元素個數相同) :
```

```
02   a = 2 3 4 1
03   b = 1 2 0 3
04   a + b  => ( 3 5 4 4 )
05   a * b  => 11
06   a -= b => ( 1 1 4 -2 )
07
```

在本程式中，為節省篇幅僅示範一些成員函式的撰寫方式，讀者如果有興趣可以將之擴充成為專門處理數學所使用向量類別。

進階排序

資料的排序是經常遇到的應用問題之一。在許多資料庫中，通常是以記錄 (record) 當成資料的最小單位，所謂的一筆記錄通常是由若干個細目 (field) 合成。例如，圖書館的一筆書籍記錄中可能包含：書名，作者，出版書局，出版年，書價等等個別資料，一筆國小學校的學生記錄可能包含：姓名，性別，出生日期，家長姓名，地址，聯絡電話等等個別資料。在程式設計中，我們通常會設計一特殊類別用來代表資料庫中的記錄，然後使用一個物件來儲存一筆記錄，例如，我們可以將圖書館的書設計成以下的類別型式：

```
class  Library_Book {
    private :
        string  book_name ;   // 書名
        string  author ;      // 作者
        ...
    public :
        ...
} ;
```

由於記錄個數通常相當龐大，在使用上經常用陣列來儲存這些記錄，例如：

```
Library_Book           book1[10]  ;   // 傳統固定陣列
vector<Library_book>   book2      ;   // 向量陣列
```

如果要列印資料庫中的所有記錄，通常我們會對存放在記錄中的資料依某種方式加以比較排序後才加以印出。在本範例程式中，我們以學生類別為例，假設每一位學生 (Student) 記錄中僅包含了姓名 (name)，年齡 (age) 與性別 (gender) 三筆個別資料，然後定義比較學生姓名次序的函式物件類別判斷式如下：

```
template <class Compare>
struct  by_name {
    bool operator() ( const Student& a , const Student& b ) const {
        return  Compare()( a.get_name() , b.get_name() ) ;
    }
} ;
```

此樣板函式的型別樣板參數為 Compare，是用來比較是否學生 a 比學生 b 較大或者較小，如果要比較是否較大，則可將 Compare 設定為 greater<NAME>，若較小，則可以設定為 less<NAME>，或者也可以是其它使用者自訂的函式物件類別。在 by_name 函式程式碼中，當 less 取代 Compare 後， by_name 函式物件類別就執行 less()(a.get_name() , b.get_name())，其作用相當於比較 a 學生名字中的字母順序是否小於 b 學生姓名的字母順序。如此，以下的敘述：

```
sort( foo , foo+5 , by_name< less<NAME> >() ) ;
sort( foo , foo+5 , by_name< greater<NAME> >() ) ;
```

前者是以學生的姓名字母順序由小到大排序，而後者則由大到小的方式來排序。我們也可以在函式物件類別中設計較複雜的比較方式，依次比較各種不同的排序條件，如此一來，排序的方式就可以相當多變。

進階資料排序：物件排序 | student_sorting.cc

```
001   #include <iostream>
002   #include <iomanip>
003   #include <string>
004   #include <functional>
005   #include <algorithm>
006
007   using  namespace  std ;
008
009   // 定義 姓名 年齡  性別 資料型別
010   typedef  string    NAME ;
011   typedef  unsigned  AGE  ;
012   enum     GENDER    { female , male } ;
013
014   // 學生類別
015   class  Student {
016
017     private :
018
019       NAME    name   ;  // 姓名
020       AGE     age    ;  // 年齡
021       GENDER  gender ;  // 性別
022
023     public :
024
025       Student( NAME n , AGE a ,GENDER g )
026           : name(n) , age(a) , gender(g) {}
027
028       NAME    get_name()   const {  return name ;   }
029       AGE     get_age()    const {  return age ;    }
030       GENDER  get_gender() const {  return gender ; }
031
032       friend  ostream& operator << ( ostream& out ,
033                                      const Student& foo ) ;
034
035   };
036
037   // 輸出學生資料
038   ostream& operator << ( ostream& out , const Student& foo ) {
```

```
039       return  out << "姓名 : " << setw(6) << foo.name.c_str()
040                   << " [" << ( foo.gender == female ? 'F' : 'M' )
041                   << "]"  << setw(5) << " 年齡 : " << foo.age  ;
042   }
043
044   // 函式類別 : 比較名稱  (比較的方式設定成樣板型別參數)
045   template <class Compare>
046   struct  by_name {
047       bool operator() ( const Student& a , const Student& b ) const {
048           return  Compare()( a.get_name() , b.get_name() ) ;
049       }
050   };
051
052   // 函式類別 : 比較年齡 (比較的樣板型別參數內定為 less<AGE>)
053   template <class  Compare = less<AGE> >
054   struct  by_age {
055       bool operator() ( const Student& a , const Student& b ) const {
056           return  Compare()( a.get_age() , b.get_age() ) ;
057       }
058   };
059
060   // 樣板函式 : 比較性別  (比較的方式設定成樣板型別參數)
061   template <class Compare>
062   bool  by_gender ( const Student& a , const Student& b ) {
063       return  Compare()( a.get_gender() , b.get_gender() ) ;
064   }
065
066   // 函式類別 : 先比較性別, 再比較年齡, 最後比較姓名
067   template <class Compare_g , class Compare_a , class Compare_n>
068   struct  my_rule {
069       bool operator() ( const Student& a , const Student& b ) const {
070           if ( a.get_gender() == b.get_gender() ) {
071
072               if ( a.get_age() ==  b.get_age() )
073                   return  Compare_n()( a.get_name() , b.get_name() ) ;
074               else
075                   return  Compare_a()( a.get_age() , b.get_age() ) ;
076
077           } else
078               return Compare_g()( a.get_gender() , b.get_gender() ) ;
079       }
080   };
081
082   int  main() {
083
084       int i ;
085       const  int  Student_No = 5 ;
086
087       // 學生陣列
088       Student  foo[Student_No] = {
089           Student("John" ,13,male) , Student("Mary" ,10,female) ,
090           Student("Tony" ,17,male) , Student("Amy"  ,10,female) ,
091           Student("Grace",18,female) } ;
092
093       cout << "> 依姓名排序 [小->大] " << endl ;
094       sort( foo , foo+Student_No , by_name< less<NAME> >() ) ;
095       for ( i = 0 ; i < Student_No ; ++i )  cout << foo[i] << '\n' ;
096       cout << endl ;
```

```
097
098     cout << "> 依年齡排序 (使用內定排序方式[小->大]) " << endl ;
099     sort( foo , foo+Student_No , by_age<>() ) ;
100     for ( i = 0 ; i < Student_No ; ++i )  cout << foo[i] << '\n' ;
101     cout << endl ;
102
103     cout << "> 依性別排序 [大->小] " << endl ;
104     sort( foo , foo+Student_No , by_gender< greater<GENDER> > ) ;
105     for ( i = 0 ; i < Student_No ; ++i )  cout << foo[i] << '\n' ;
106     cout << endl ;
107
108     cout << "> 依下列方式順序排序\n"
109          << "  (1) 性別 [小->大] (2) 年齡 [大->小] "
110          << "(3) 姓名 [小->大]\n";
111     sort( foo , foo+Student_No ,
112           my_rule< less<GENDER>, greater<AGE> , less<NAME> >() ) ;
113     for ( i = 0 ; i < Student_No ; ++i )  cout << foo[i] << '\n' ;
114     cout << endl ;
115
116     return 0 ;
117
118 }
119
```

執行結果

```
01  > 依姓名排序 [小->大]
02  姓名 :   Amy [F]    年齡 : 10
03  姓名 : Grace [F]    年齡 : 18
04  姓名 :  John [M]    年齡 : 13
05  姓名 :  Mary [F]    年齡 : 10
06  姓名 :  Tony [M]    年齡 : 17
07
08  > 依年齡排序 (使用內定排序方式[小->大])
09  姓名 :   Amy [F]    年齡 : 10
10  姓名 :  Mary [F]    年齡 : 10
11  姓名 :  John [M]    年齡 : 13
12  姓名 :  Tony [M]    年齡 : 17
13  姓名 : Grace [F]    年齡 : 18
14
15  > 依性別排序 [大->小]
16  姓名 :  John [M]    年齡 : 13
17  姓名 :  Tony [M]    年齡 : 17
18  姓名 :   Amy [F]    年齡 : 10
19  姓名 :  Mary [F]    年齡 : 10
20  姓名 : Grace [F]    年齡 : 18
21
22  > 依下列方式順序排序
23    (1) 性別 [小->大] (2) 年齡 [大->小] (3) 姓名 [小->大]
24  姓名 : Grace [F]    年齡 : 18
25  姓名 :   Amy [F]    年齡 : 10
26  姓名 :  Mary [F]    年齡 : 10
27  姓名 :  Tony [M]    年齡 : 17
28  姓名 :  John [M]    年齡 : 13
```

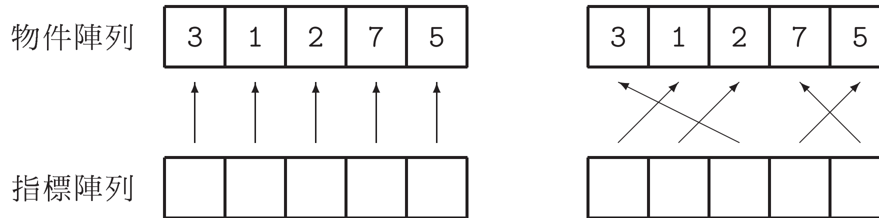


圖 17.1: 指標陣列輔助排序

29

在程式碼中，by_name，by_age 與 my_rule 都是函式物件類別，而 by_gender 則刻意地被設計為樣板函式，但都一樣可以被 sort 函式當作排序比較大小的標準。若程式碼中的學生陣列為一向量陣列，則相關的排序敘述須改為

```
sort( foo.begin() , foo.end() , by_name< less<NAME> >() ) ;
sort( foo.begin() , foo.end() , by_age<>() ) ;
...
```

一般而言，當對物件陣列排序時，若兩物件彼此大小順序不對，則物件須要在陣列中對調位置。當物件對調位置時，通常須要使用到暫時物件，若陣列元素相當多且類別的資料成員很複雜時，則處理物件的產生與消失所要耗費的計算時間就相當龐大。在這種情況下，可以利用指標的方式來間接處理。例如：如果有五個數字要由小到大排序，這時可以先產生一個同樣大小的指標陣列一一指到各個數字元素，然後再對指標所指的元素大小排序，當指標所指的數字的順序不對時，就對調指標，如此當排序完成時，指標陣列所指到的元素順序就是排序後的順序，如圖 17.1。

以此範例為例，若要用指標方式對年齡由小到大排序，則可程式可以寫成：

進階資料排序：指標排序 | sorting_by_ptr.cc

```
01  #include <iostream>
02  #include <iomanip>
03  #include <string>
04  #include <functional>          // 提供 : less  greater
05  #include <algorithm>           // 提供 : sort
06
07  using  namespace  std ;
08
09  // 定義 姓名 年齡  性別 資料型別
10  typedef  string    NAME ;
11  typedef  unsigned  AGE  ;
12  enum     GENDER    { female , male } ;
13
```

```
14  // 學生類別
15  class  Student {
16
17    private :
18
19      NAME    name   ;  // 姓名
20      AGE     age    ;  // 年齡
21      GENDER  gender ;  // 性別
22
23    public :
24
25      Student( NAME n , AGE a ,GENDER g )
26          : name(n) , age(a) , gender(g) {}
27
28      AGE     get_age()    const {  return age ;    }
29
30      friend  ostream& operator << ( ostream& out ,
31                                     const Student& foo ) ;
32
33  };
34
35  // 輸出學生資料
36  ostream& operator << ( ostream& out , const Student& foo ) {
37      return  out << "姓名 : " << setw(6) << foo.name.c_str()
38                  << " [" << ( foo.gender == female ? 'F' : 'M' )
39                  << "]"  << setw(5) << " 年齡 : " << foo.age  ;
40  }
41
42  // 函式類別 : 比較年齡 (比較的樣板型別參數內定為 less<AGE>)
43  template <class  Compare = less<AGE> >
44  struct  by_age {
45      bool operator() ( const Student* a , const Student* b ) const {
46          return  Compare()( a->get_age() , b->get_age() ) ;
47      }
48  };
49
50  int  main() {
51
52      int i ;
53      const  int  Student_No = 5 ;
54
55      // 學生陣列
56      Student  foo[Student_No] = {
57          Student("John" ,13,male) , Student("Mary" ,10,female) ,
58          Student("Tony" ,17,male) , Student("Amy"  ,10,female) ,
59          Student("Grace",18,female) } ;
60
61      // 指標陣列
62      Student  *ptrs[Student_No] ;
63      for ( i = 0 ; i < Student_No ; ++i ) ptrs[i] = &foo[i] ;
64
65      // 使用指標排序
66      cout << "> 依年齡排序 [大->小] " << endl ;
67      sort( ptrs , ptrs+Student_No , by_age< greater<AGE> >() ) ;
68      for ( i = 0 ; i < Student_No ; ++i ) cout << *ptrs[i] << '\n' ;
69      cout << endl ;
70
71      cout << "> 原始陣列順序 " << endl ;
```

```
72      for ( i = 0 ; i < Student_No ; ++i ) cout << foo[i] << '\n' ;
73      cout << endl ;
74
75      return 0 ;
76
77  }
78
```

執行結果

```
01  > 依年齡排序 [大->小]
02  姓名 :  Grace [F]     年齡 : 18
03  姓名 :   Tony [M]     年齡 : 17
04  姓名 :   John [M]     年齡 : 13
05  姓名 :   Mary [F]     年齡 : 10
06  姓名 :    Amy [F]     年齡 : 10
07
08  > 原始陣列順序
09  姓名 :   John [M]     年齡 : 13
10  姓名 :   Mary [F]     年齡 : 10
11  姓名 :   Tony [M]     年齡 : 17
12  姓名 :    Amy [F]     年齡 : 10
13  姓名 :  Grace [F]     年齡 : 18
14
```

在此修正程式中，我們首先產生一個指標陣列依次指向各個物件陣列元素，然後在 sort 函式中，將指標當成排序的輸入參數。為了對指標所指的物件作排序的動作，因此在負責比較年齡大小的 by_age 函式中，我們所比較的是指標所指向物件內的年齡資料。如此當指標所指向的物件年齡順序不對時，指標將會自動對調兩者的位址。在輸出的結果中，我們也順便將原有的物件陣列順序印出，藉以說明原始的物件陣列並沒有在排序過程中被改變。由於排序函式無法直接用在串列容器，透過指標排序法，我們也一樣可以對串列資料作須要的排序動作。以上如果是使用向量陣列來為儲存學生物件及學生物件指標，則程式相異部份主要為：

```
vector<Student>   foo(Student_No)  ; // 向量陣列內含學生物件
vector<Student*>  ptrs(Student_No) ; // 向量陣列內含指標指向學生物件
...
for ( i = 0 ; i < Student_No ; ++i ) ptrs[i] = &foo[i] ;
sort( ptrs.begin() , ptrs.end() , by_age< greater<AGE> >() ) ;
...
```

這裡的 vector<Student*> ptrs 為一向量陣列物件，但其所儲存的元素為學生指標。

當容器內的元素為某種複雜類別的物件時，為了避免花費太多的執行時間於元素的建構或者是移除上，使用指標方式來間接的存取元素是相當常見的作法，例如：對

此問題而言，如果我們要將學生 foo 陣列的學生次序打亂，可以仿照在第 124 頁上的作法，利用指標的對調來代替學生次序的更動，如此一來可以避免處理複雜物件的產生或去除所須要的時間，例如：

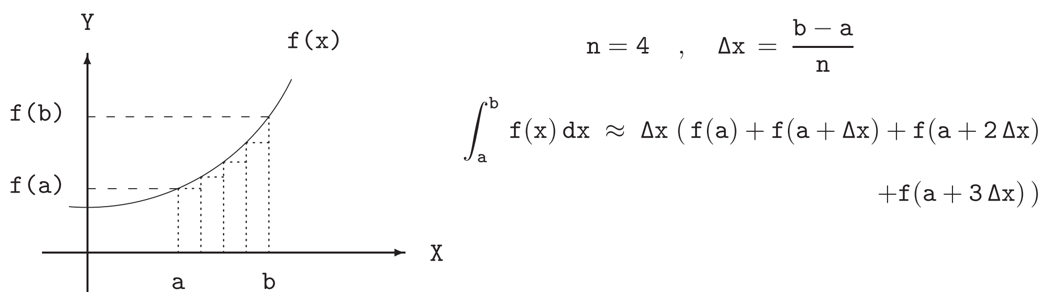
```
int      i , j ;
Student  foo[Student_No] , *ptrs[Student_No] , *tmp ;
...
for ( i = 0 ; i < Student_No ; ++i ) ptrs[i] = &foo[i] ;

// 對調兩個指標所指到的位置
for ( i = 0 ; i < (Student_No-1) ; ++i ) {
    j       = rand() % (Student_No-i) + i ;
    if ( i == j ) continue ;
    tmp     = ptrs[i] ;
    ptrs[i] = ptrs[j] ;
    ptrs[j] = tmp     ;
}
```

定積分的數值計算

在微積分 (Calculus) 中，一個函式 f(x) 的定積分 $\int_a^b f(x)\,dx$ 為函數曲線與 X-軸在 $[a,b]$ 區間所圍成的區域面積，此面積可以利用無限多個細長條矩形的面積合成計算出來，因此定積分在數學上被定義成為以下的式子：

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(a+(i-1)\,\Delta x)\,\Delta x \qquad , \quad \Delta x = \frac{b-a}{n}$$

當然在實際的計算層面上，我們無法計算無限多個矩形的總面積，而是在 [a,b] 區間將其分割成 n 個相同等份，然後計算每個等份內函數與 X 軸之間所圍成的長條矩形面積之總和，例如：下圖是將 [a,b] 區間切割為四等份，即 n = 4



$$n = 4 \quad , \quad \Delta x = \frac{b-a}{n}$$

$$\int_a^b f(x)\,dx \approx \Delta x\,(\,f(a) + f(a+\Delta x) + f(a+2\,\Delta x) + f(a+3\,\Delta x)\,)$$

在此範例中，我們特別設計一個泛型函式來計算函數在 [a,b] 區間的積分值，為控制計算的精確度，使用者可以自行設定切割等份的數值 n，使用以下簡單的方式將函式的定積分數值求出：

```
// 定義平方函式物件類別
struct  Square {
    double  operator() ( double x ) const { return  x * x ; }
} ;
...
// 計算平方函數在 2 到 10 之間的定積分 (n 為 1000000 等份)
cout << integral( Square() , 2 , 10 ) ;

// 計算平方函數在 2 到 10 之間的定積分 (n 為 500 等份)
cout << integral( Square() , 2 , 10 , 500 ) ;

// 計算平方函數在 [0,1] 且函數值須小於 0.25 的定積分
cout << integral( Square() , 0 ,  1 , bind2nd( less<double>() ,
                                           0.25 ) ) ;
```

定積分的數值計算 | integral.cc

```
01   #include <iostream>
02   #include <math.h>
03   #include <functional>
04
05   using namespace std ;
06
07   // 定義 X 平方，立方與 Sin 函數
08   struct  Square {
09       double  operator()( double x ) const { return x * x ; }
10   };
11
12   template <typename T>
13   T  Cubic ( T x ) { return x * x * x ; }
14
15   template <typename T>
16   double Sin( T x ) { return sin(x) ; }
17
18   // 計算函數 fn 在 [a,b] 區間的定積分，
19   // 積分以切割為 n 個平均等份的長條形面積計算
20   //
21   template <typename Function>
22   double  integral( Function fn , double a , double b ,
23                     int n = 1000000 ) {
24
25       double  area = 0.  ;
26       double  dx = ( b - a ) / n ;
27
28       for ( int i = 0 ; i < n ; ++i ) area += fn(a+dx*i) ;
29
30       return  dx * area ;
31   }
32
33   // 計算函數 fn 在 [a,b] 區間且函數值滿足某個設定條件的定積分，
34   // 積分以切割為 n 個平均等份的長條形面積計算
35   //
36   template < typename Fn1 , typename Fn2 >
37   double  integral( Fn1 fn , double a , double b ,
```

```
38                      const Fn2& fn2 , int n = 1000000 ) {
39
40        double  area = 0. , tmp ;
41        double  dx = ( b - a ) / n ;
42
43        for ( int i = 0 ; i < n ; ++i ) {
44            tmp = fn(a+dx*i) ;
45            if ( fn2(tmp) ) area += tmp ;
46        }
47
48        return  dx * area ;
49    }
50
51    int main() {
52
53        const double PI = 3.141592654 ;
54
55        // 計算平方函數在 [0,1] 區間的定積分
56        cout << "> 平方函數在 [0,1] 區間的定積分 = "
57             << integral( Square() , 0 , 1 ) << endl ;
58
59        // 計算平方函數在 [0,1] 且函數值須小於 0.25 的定積分
60        cout << "> 平方函數在 [0,1] 且函數值須小於 0.25 的定積分 = "
61             << integral( Square() , 0 ,  1,
62                          bind2nd( less<double>() , 0.25 ) ) << endl ;
63
64        // 計算立方函數在 [0,1] 區間的定積分
65        cout << "> 立方函數在 [0,1] 區間的定積分 = "
66             << integral( Cubic<double> , 0 , 1 ) << endl ;
67
68        // 計算 sin 函數在 [0,PI] 區間的定積分
69        cout << "> sin(x) 在 [0,PI] 區間的定積分 = "
70             << integral( Sin<double> , 0 , PI ) << endl ;
71
72        // 計算 sin 函數在 [0,PI] 且 sin(x) 須大於零的定積分
73        cout << "> sin(x) 在 [0,PI] 且 sin(x) 須大於零的定積分 = "
74             << integral( Sin<double> , 0 , 2*PI ,
75                          bind2nd( greater<double>() , 0. ) ) << endl ;
76
77        return 0 ;
78
79    }
80
```

執行結果

```
01    > 平方函數在 [0,1] 區間的定積分 = 0.333333
02    > 平方函數在 [0,1] 且函數值須小於 0.25 的定積分 = 0.0416665
03    > 立方函數在 [0,1] 區間的定積分 = 0.25
04    > sin(x) 在 [0,PI] 區間的定積分 = 2
05    > sin(x) 在 [0,PI] 且 sin(x) 須大於零的定積分 = 2
06
```

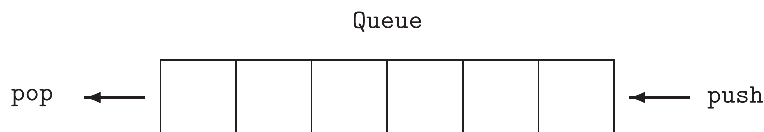
在數值分析上，計算函數定積分的數值方法有相當多種方式，這裡所使用的方式，即利用許多長條矩形面積相加的方式，是一種相當不經濟的計算方法，須要耗費相當的計算量，同時方法的精確度也相當低。既然定積分是計算函數在區間內與 X 軸的面積，讀者可以推知利用細長的梯形面積相加會比長方形面積相加較為精確，有關這方面請參考本章習題。

17.7 結語

本章內容只是簡單的介紹 STL 語法的使用，對各種容器， 迭代器與許多功能強大的泛型演算函式都沒有介紹，雖然有許多內容將留待以後的三章才加以講解，然而整個 STL 因包含相當豐富的內容與應用，筆者無法在本書中一一介紹，讀者若想更進一步使用 STL，請參閱一些 STL 專書。

17.8 練習題

1. 在第 279 頁中，我們介紹了佇列 (queue) 資料結構，在佇列的儲存機制中，先放入佇列的元素會先被取出來，也就是先進先出，這種資料結構常常被用來模擬一些與排隊有關的實際問題，

   佇列資料結構 (queue data structure) 剛好可以用佇列容器 (`deque` container) 來模擬，當元素要加入 (push) 佇列結構時，可以用 `push_back` 送到佇列資料容器的最後，當元素要由佇列資料結構取出 (pop) 時，可以使用 `pop_front` 將元素由佇列容器中去除，但在去除之前，可以用 `front` 將存在資料容器最前端的元素複製出來，請用佇列容器寫一個程式來模擬佇列資料結構的機制。

2. 承上題，請改用串列容器 (`list` container) 來模擬佇列資料結構 (queue data structure)。

3. 若定義某日期結構為：

   ```
   struct  Date {
       int  year , month , day ;
   } ;
   ```

   請使用 STL 的 `sort` 函式，針對此資料結構，定義一函式物件類別 (functor class) 用以比較日期的前後。

4. 某時間結構定義為：

```
struct  Time {
    int  hr , min ;
} ;
```

   請使用 STL 的 sort 函式，針對此定義，撰寫相關的函式物件類別用以比較時間的前後。假設輸入一行不等數量的時刻為：

```
15:07   21:35   08:23   10:02   12:59
```

   (a) 請依時間的早晚順序印出
   (b) 請依分鐘的大小由大到小排出

5. 輸入一數列，請根據以下的要求設計相關的泛型函式：

   (a) 列印整個數列
   (b) 僅印出大於某數值的數字
   (c) 印出介於在某範圍內的數字
   (d) 計算大於零的數字和

6. 元順帝時，汝陽王舉辦了一場射箭比賽，經過初始階段的淘汰後，最後共有八個人進入決賽，假設此八人分別為趙一傷，錢二敗，孫三毀，李四摧，周五輸，吳六破，鄭七滅，王八衰。假設最後決賽的射箭成績如下表，請寫一個程式讀入資料後，依總分高低依次將比賽成績由高到低排出，並將總分數印出，若總分相同，則比較第一次成績，如果第一次成績相同，則比較第二次成績，其餘依此類推。

趙一傷	10	8	10	10	10
錢二敗	8	8	10	8	10
孫三毀	9	9	9	10	10
李四摧	7	8	9	10	8
周五輸	5	6	8	9	8
吳六破	10	10	9	10	9
鄭七滅	9	8	7	8	10
王八衰	6	9	10	8	9

7. 承上題，請設計相關的泛型函式：

   (a) 將每回合的射箭分數都在 7 分以上的人印出
   (b) 射箭總分介於 a 分數與 b 分數的人印出，包含 a 與 b 兩分數

(c) 射箭總分在 a 分數以上的人數，包含分數 a

8. 某一唐詩目錄如下：

   王維　　送別 相思 辛夷塢 渭川田家
   李白　　夜思 秋浦歌 勞勞亭 怨情 獨坐敬亭山 月下獨酌
   杜甫　　八陣圖
   王之渙　送別 登鸛雀樓
   白居易　問劉十九 夜雪 池上
   王昌齡　答武陵太守 送郭司倉 塞上曲
   柳宗元　江雪
   孟浩然　送朱大入秦 宿建德江 春曉 訪袁拾遺不遇

   (a) 請寫一個程式讀入此目錄資料檔後，依每個詩人的作品數量由小至大印出，若是作品數量相同，則依所有詩的名稱總長度由小至大來排序，同時輸出的詩也以詩的名稱長度由短至長排序，以此為例，輸出應為：

   柳宗元　江雪
   杜甫　　八陣圖
   王之渙　送別 登鸛雀樓
   白居易　夜雪 池上 問劉十九
   王昌齡　塞上曲 送郭司倉 答武陵太守
   王維　　送別 相思 辛夷塢 渭川田家
   孟浩然　春曉 宿建德江 送朱大入秦 訪袁拾遺不遇
   李白　　夜思 怨情 秋浦歌 勞勞亭 月下獨酌 獨坐敬亭山

   相關類別可定義為：

```
class  Poem {
    private :
        string          author ;
        vector<string>  poems  ;
    public :
        friend istream& operator>> ( istream& , Poem& ) ;
        friend ostream& operator<< ( ostream& ,
                                     const Poem& ) ;
} ;
```

   (b) 請設計泛型函式，回傳出作品在 n 首以上的詩人人數。

9. 承上題，若是輸入檔的詩人數目不固定，請將之存入向量陣列中，且用指標排序方式來排序，若要使用以下的向量陣列，則程式將如何改寫。

```
vector<Poem>   poet ;    // 向量陣列儲存詩人物件
vector<Poem*>  ptrs ;    // 向量陣列儲存詩人物件指標
```

10. 承上題，如果程式要使用串列來存詩人物件，用佇列陣列儲存指標，則程式又該如何撰寫？

```
list<Poem>    poet ;    // 向量陣列儲存詩人物件
deque<Poem*>  ptrs ;    // 向量陣列儲存詩人物件指標
```

11. 輸入兩個等長的容器範圍，請用系統內定的函式物件類別，設計一泛型函式計算兩個容器元素的絕對值乘積和，即

$$\text{絕對值和} = |\,a_0\,| * |\,b_0\,| + |\,a_1\,| * |\,b_1\,| + \cdots + |\,a_{n-1}\,| * |\,b_{n-1}\,|$$

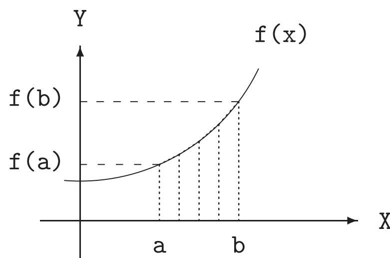
使得以下的程式碼可以執行：

```
int           foo[5] = { 2 , -3 , 1 , -2 , 5 } ;

// bar = 1 -2 5
vector<int>  bar(foo+2,foo+5) ;

// 印出  15 ( = 3*1 + 1*2 + 2*5 )
cout << abs_sum( foo+1 , foo+4 , bar.begin() , 0 ) << endl ;
```

12. 一個函式 f(x) 的定積分 $\int_a^b f(x)\,dx$ 就是代表著函數的曲線與 X-軸在 $[a,b]$ 區間所圍成的區域面積，此面積除了可以利用無限多個細長條矩形的面積合成計算出來之外，也可以使用無限多個細長的梯形面積逼近求得，因此定積分也可以定義為以下的式子：

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=1}^{n} \frac{\Delta x}{2}\,(\,f(a+(i-1)\,\Delta x) + f(a+i\,\Delta x)\,)\ ,\ \Delta x = \frac{b-a}{n}$$

如同範例[566]一樣，在實際的計算層面上，我們無法計算無限多個梯形的總面積，而是須要在 [a,b] 區間分割成 n 個相同的等份，然後將每個等份在函數與 X 軸之間所圍成的長條梯形面積一一算出後相加，例如：下圖是將 [a,b] 之間切割為四等份，即 n = 4



$$n = 4 \quad , \quad \Delta x = \frac{b-a}{4}$$

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{2} \sum_{i=1}^{n} (\,f(a+(i-1)\,\Delta x) + f(a+i\,\Delta x)\,)$$

請依以上的說明，設計一個泛型函式來計算一個函數在 a 與 b 之間的積分值，同時也可以讓使用者自行設定切割 n 等份的數值，假設預設的切割等份 n 的內定值為 100。

```
// 計算平方函數在 2 到 10 之間的定積分 (n 為 100 等份)
double  integral( Square() , 2 , 10 ) ;

// 在 2 到 10 之間切割成 10000 個等份，計算立方函數的定積分
double  integral( Cubic() , 2 , 10 , 10000 ) ;
```

13. 在數學上要計算兩個平面點 $p(x_1, y_1)$ 與 $q(x_2, y_2)$ 之間的距離通常是利用以下的公式來求得，

$$d(p, q) \ = \ \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

平面上任兩點的距離一定大於等於零，且當兩點距離為零的唯一情況就是兩點重疊。將這個概念稍加延伸，我們也可以採用類似的方式來計算兩個函數在 [a,b] 區間的「距離」。一種較常用的作法是在 [a,b] 區間內計算兩個函數差值平方的定積分值的平方根，也就是說，若兩函數分別為 f(x)，g(x)，則兩函數在 [a,b] 區間的「距離」， d(f,g)，可以寫成：

$$d(f, g) \ = \ \sqrt{\int_a^b \ (f(x) - g(x))^2 \ dx}$$

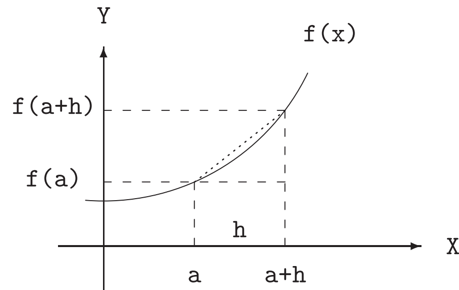
讀者可以很容易地看出，當兩個函式重疊時，距離剛好為零[註2]。請利用此定義，及上一題定積分的泛型函式，撰寫一個泛型函式，在輸入兩個函式與積分區域，求得兩函數的「距離」。

14. 若要求得函式 f(x) 在某點 a 的微分，在數學上是計算函數在 a 點的斜率，也就是：

$$f'(a) = \lim_{h \to 0} \ \frac{f(a + h) - f(a)}{h}$$

當 h 趨近於零時上式的比值就是函數 f 在點 a 的微分值。當然，若 h 趨近於零時，計算的比值若不逼近於相同的數字，則函數在 a 點的微分就不存在。在實際計算層面上，我們通常是將 h 設定為一很小的數值，例如 0.00001，然後計算 $\frac{f(a+0.00001)-f(a)}{0.00001}$ 的比值，如下圖所示：

[註2]在數學上，兩函數也有可能距離為零但不完全重疊，但此細節並不是這裡的重點，所以略而不談



現在請設計一泛型函式 derivative，接受一函式物件計算此函式在某點 a 的微分值，例如，以下分別為函數 $f(x) = x^2$ 在 $x = 1.5$ 與 $g(x) = x^3$ 在 $x = 2.5$ 的微分近似值：

```
struct  Square {
    double  operator()(double x) const { return  x * x ; }
} ;

struct  Cubic {
    double  operator()(double x) const { return  x * x * x ; }
} ;

// 計算平方函數在 x = 1.5 且以內定的 h = 0.00001 的方式算得微分
double  derivative( Square() , 1.5 ) ;

// 計算立方函數在 x = 2.5 且以 h = 0.001 的方式算得微分
double  derivative( Cubic() , 2.5 , 0.001 ) ;
```

15. 請利用上題求函數微分的函式，撰寫一泛型函式，使用牛頓迭代法[75]，在輸入函式與起始值後，計算函式的數值根，例如：

```
struct  F {
    double  operator()(double x) const {
        return  x * x * x - 2 * x * x + 1 ;
    }
} ;

struct  G {
    double  operator()(double x) const { return cos(x) - x ; }
} ;

// 利用牛頓迭代法且起始值 3 求 F 函式的根
cout << Newton_root( F() , 3 ) << endl ;

// 利用牛頓迭代法且起始猜想值為預設值來計算 G 函式的根
cout << Newton_root( G() ) << endl ;
```

16. 某學系欲建立畢業系友資料庫，假設每一筆系友資料包含：姓名，入學年份，

畢業年份等三項個別資料，請寫一個程式讀入系友資料檔，由資料的第一年起，將每一年的學生人數印出。

17. 修改第 503 頁的點矩陣函式圖形範例程式碼，使得 Plot_Function 類別也可以畫出以函式物件類別所定義函式的點矩陣圖形。相關使用的程式碼如下：

```
// 定義 SIN2 函式物件類別為 sin(x) 函數的平方
struct  SIN2 {
    double operator()( double x ) const {
        return  sin(x)*sin(x) ;
    }
} ;

// 直接畫出 SIN2 函數物件圖形於 [-4,4] 區間內
cout << Plot_Function<SIN2,17,81>( SIN2() , "sin(x)**2" , -4 , 4 )
     << endl ;


// 設定 f(x) 函數為 sin(x)+cos(x)
double  f( double x ) {  return  sin(x)+cos(x) ; }

// 產生 foo 物件可畫出 f(x) 函數於 [-4,4] 區間內
Plot_Function<double(*)(double),17,81>  foo( f , "sin(x)+cos(x)" ,
                                            -4 , 4 ) ;

// 輸出 foo 物件, 即畫出 f(x) 圖形
cout << foo << endl ;
```

請留意：以上在定義 foo 物件時，double(*)(double) 為 f(x) 函式輸入 Plot_Function 內的樣板型別。