Appendix G

STL 成員函數和函數

標準樣版函式庫（STL）的目的，是提供常見的演算法之有效率的實作方式。它以一般的函數和成員函數表示這些演算法，一般的函數（general function）可以和收納器來滿足特定演算法的需求，而成員函數（method）則可以和特定收納器類別的實體化來使用。這附錄是假設您已經熟悉 STL 的相關知識。例如，第十六章的迭代器（iterator）和建構函數。

STL 與 C++11

本書包含 C++11 對 C++語言廣泛的改變，同樣地，此附錄也涵蓋 STL 廣泛的變化.。然而此處只是對新加入的資料做一扼要的說明。

C++11 注入新的元素於 STL。首先，它加入一些新的收納器(container)，第二，加入新的特性於舊的放納器，第三，加入一些新的樣版函數於演算法家族。所有的改變在此附錄皆可以查到，不過您可以下列介紹前兩項的總論得到一些知識 。

新的收納器

C++11 加入以下收納器，包括: `array`、 `forward_list` 及 `unordered_set`。還有無序性關聯式收納器包括 `unordered_multiset` 、 `unordered_map` 、及 `unordered_multimap`。

陣列收納器 (array container) 宣告後，其大小就被固定了，而且它使用靜態或堆疊記憶體，而不是動態的記憶體配置。它傾向用來取代內建的陣列。和 vector 比起來，它限制較多，但比較有效率。

串列收納器 (list container) 是一雙向的鏈結串列。除了尾端外，每一項目都有指向前一個和後一個項目的鏈結。forward_list 是一單向鏈結串列，除了尾端，每一項目有一指向下一個項目的鏈結。相對而言，它比一般的串列較簡潔但較多的限制。

如同串列和關聯式收納器，無序性關聯式收納器利用鍵值快速搜尋資料。關聯式收納器則是利用樹狀結構，而無序性的關聯式收納器則使用雜湊表格。

C++98 收納器的改變

C++11 注入三個主要改變於收納器的類別成員函數。

第一，加入 rvalue references 使得它可提供移動語意於收納器。(請參閱第 18 章“檢視新的 C++標準 “。) 於是現在 STL 有提供移動建構函數和移動指定運算子給收納器。這些成員函數 rvalue reference 為其參數。

第二，附加的 initializer_list 樣版類別 (請參閱第 18 章) 使得建構函數和指定運算子接再 initializer_list 的參數。其程式碼有如下列所示:

```
vector<int> vi{100, 99, 97, 98};
vi = {96, 99, 94, 95, 102};
```

第三，附加的可變參數樣版與函數參數包裝 (請參閱第 18 章) 使得 emplacement 的成員函數變成可行性。這是什麼意思呢? 有如移動語意，emplacement 為增加效率的意思，請考慮下列片段的程式碼。

```
class Items
{
    double x;
    double y;
    int m;
public:
    Items(); // #1
    Items (double xx, double yy, int mm); // #2
    ...
};
...
vector<Items> vt(10);
...
vt.push_back(Items(8.2, 2.8, 3)); //
```

insert()的呼叫引起記憶體配置函數在 vt 尾端建立一預設 Items 物件。接下來 Items()建構函數建立一暫時的 Items 物件。此物件被複製到 vt 的前端位置，然後將暫時的物件刪除。在 C++11，您可以如此做

```
vi.emplace_back(8.2, 2.8, 3);
```

emplace_back() 成員函數是一以函數包裝為其參數的可變參數樣版：

```
template <class... Args> void emplace_back(Args&&... args);
```

三個參數 8.2、 2.8 及 3 被包裝於 args 參數。這些參數被傳遞到配置函數中，然後可以解開這些參數，並利用具有三個參數的 Items 建構函數(#2)取代預設的建構函數(#1)。在此範例中，它可以使用 Items(args...) 擴展到 Items(8.2, 2.8, 3)。因此需要的物件是在 vector 中建立，而不是在暫時位置建立後再複製到 vector。

STL 利用 emplacement 成員函數使用此技術。

所有收納器共有的成員

所有收納器定義的型態列於表 G.1。在此表中，X 是收納器型態，如 vector<int>，而 T 是儲存在收納器中的型態，如 int。表格後面的範例可以釐清這些意義。

表 G.1 所有收納器定義的型態

型態	值
X::value_type	T，元素型態
X::reference	T &
X::const_reference	const T &
X::iterator	迭代器型態，指向 T，行為就像是 T *
X::const_iterator	迭代器型態，指向 const T，行為就像是 const T *
X::difference_type	有號的整數型態，用來表示兩個迭代器之間的距離（例如，兩個指標之間的差）
X::size_type	無號的整數型態 size_type，可以表示資料物件的大小，元素的個數，以及足標。

此類別定義是使用 typedef 定義這些成員。您可以用這些型態宣告合適的變數。例如，下面的工作是採用間接的方式，對 vector 中的 string 物件，將第一次出現 "bonus" 之處，取代為 "bogus"，藉以說明您可以用成員型態宣告變數。

```
vector<string> input;
string temp;
while (cin >> temp && temp != "quit")
    input.push_back(temp);
vector<string>::iterator want=
```

```
    find(input.begin(), input.end(), string("bonus"));
if (want != input.end())
{
    vector<string>::reference r = *want;
    r = "bogus";
}
```

這程式碼將 r 宣告為 want 所指之 input 元素的 reference。類似地，接著前面的範例，您可以撰寫如下的程式碼：

```
vector<string>::value_type s1 = input[0];  // s1 is type string
vector<string>::reference s2 = input[1];   // s2 is type string &
```

這使得 s1 成為一個 input[0] 之複本的新 string 物件，並且 s2 為 input[1] 的 reference。在這個範例中，假設您已經知道樣版是基於 string 型態，那麼可以使用較為簡單的方式撰寫程式碼，並且達到相同的效果：

```
string s1 = input[0]; // s1 is type string
string & s2 = input[1]; // s2 is type string &
```

然而，其它在表 G.1 中的型態，也可以用在更一般化的程式碼中，其中收納器和元素都是通用型態。例如，假設您希望 min() 函數的引數是收納器的 reference，並回傳收納器中最小的項目。這前提是 < 運算子已經定義處理該值型態，且您不希望使用 STL min_element() 演算法(這使用迭代器介面)。因為這引數是 vector<int> 或是 list<string> 或是 deque<double>，使用具有樣版參數的樣版，如 Bag 來表示此收納器（也就是說，Bag 是樣版型態並且可以是如 vector<int>, list<string>，或者其它的收納器型態）。所以此函數的引數型態會是 const Bag & b。回傳型態為何？應是此收納器的值型態，也就是 Bag::value_type。但是在此時 Bag 只是樣版參數，而且編譯程式無法知道 value_type 成員實際上是一個型態。但是您可用關鍵字 typename 說明類別成員是一 typedef：

```
vector<string>::value_type st;    // vector<string> a defined class
typename Bag::value_type m;       // Bag an as yet undefined type
```

對於第一個定義，編譯程式要存取 vector 的樣版定義，這會說明 value_type 是 typedef。對於第二個定義，關鍵字 typename 承諾，無論 Bag 結果是什麼，Bag::value_type 是一型態名稱。這些考量導致下面的定義：

```
template<typename Bag>
typename Bag::value_type min(const Bag & b)
{
    typename Bag::const_iterator it;
    typename Bag::value_type m = *b.begin();
```

```
    for (it = b.begin(); it != b.end(); ++it)
        if (*it < m)
            m = *it;
    return m;
}
```

然後，您可以下面的方式使用此樣版函數：

```
vector<int> temperatures;
// input temperature values into the vector
int coldest = min(temperatures);
```

參數 temperatures 會使 Bag 的求出值為 vector<int>，而且 typename Bag::value_type 的求出值為 vector<int>::value_type，也就是 int。

所有的收納器也包含成員函數或是操作，請參閱表 G.2。同樣的，X 是收納器型態，如 vector<int>，而 T 是儲存在收納器中的型態，如 int。同時，a 和 b 是型態 X 之值，u 是識別字，r 和 rv 都是型態 x 的 non-const 值。有關移動的操作是 C++11 新增的。

表 G.2 所有收納器成員函數的操作

成員函數/操作	說明
X U;	建構一空的物件 U
X()	建構一空的物件
X(a)	建構一物件 a 的複本
X u(a)	u 是 a 的複本（複製建構函數）
X u = a;	u 是 a 的複本（複製建構函數）
r = a	r 等於 a 值
X u(rv)	u 等於 rv 建構前的值（移動建構函數）
X u = rv;	u 等於 rv 建構前的值（移動建構函數）
a = rv	a 等於 rv 建構前的值（移動建構函數）
(&a) -> ~X()	運用於 a 的每一元素之解構函數
begin()	回傳迭代器，指向第一個元素
end()	回傳迭代器，指向 past-the-end
rbegin()	回傳反向迭代器，指向 past-the-end
rend()	回傳反向迭代器，指向第一個元素
size()	回傳元素個數
maxsize()	回傳最大可能之收納器的大小。
empty()	若收納器是空的，則回傳 true。

成員函數/操作	說明
swap()	交換兩個收納器的內容。
==	若兩個收納器的大小相等，且有相同的元素，順序也相同，則回傳 true。
!=	a != b 等於 !(a == b)

這些收納器是可逆式的，使用雙向或隨機迭代器（如 vetctor，list，deque，queue，array，set 及 map）。有關其成員函數請參閱表 G.3。

表 G.3 可逆式收納器的型態與操作

成員函數/操作	說明
X::reverse_iterator	指向型態為 T 的可逆式迭代器。
X::const_reverse_iterator	指向型態為 T 的 const 可逆式迭代器。
a.rbegin()	回傳一可逆式迭代器，其指向 a 尾端的下一個元素。
a.rend()	回傳一可逆式迭代器，其指向 a 的前端。
a.crbegin()	回傳一 const 可逆式迭代器，其指向 a 尾端的下一個元素。
a.crend()	回傳一 const 可逆式迭代器，其指向 a 的前端。

無序性的資料集和無序性的收納器，並不需要提供如表 G.4 選擇性收納器的操作，其餘的收納器有支援這些。

表 G.4 選擇性收納器的成員函數

成員函數/操作	說明
<	若在詞彙編纂的順序上 a 在 b 之前，則 a < b 回傳 true
>	a > b 回傳 b < a
<=	a <= b 回傳 !(a > b)
>=	a >= b 回傳 !(a < b)

對於收納器> 運算子，會假設此運算子已經定義處理此值的型態。詞彙的比較是字母排序的一般化。它比較兩個收納器的元素，直到遇見一個收納器中的元素不等於另一個收納器的對應元素。此時，收納器的順序與這兩個不相同元素的順序相同。例如，若兩個收納器的前 10 個元素都相同，但是第一個收納器的第 11 個元素，

小於第二個收納器的第 11 個元素，則第一個收納器在第二個之前。若兩個收納器比較都相等，直到一個收納器沒有元素，則較短的收納器在較長者之前。

序列收納器的其他成員

vector，forward_list，list，deque 及 array 樣版類別，都是**序列**（sequence）收納器。除了 forward_list 不是可逆性，而且沒有提供如表 G.3 所列的成員函數外，都有前面所列的成員函數。

同樣的 X 是收納器型態，如 vector<int>，而 T 是儲存在收納器中的型態，如 int。a 是型態 X 之值，t 是型態 X::value_type 之值，i 和 j 是輸入迭代器，q2 和 p 是迭代器，q 和 q1 是可提領的迭代器（可以將 * 用在它們身上），而 n 是整數的 X::size_type。

表 G.5 序列收納器所定義的的成員函數

成員函數	說明
X(n, t)	建立含有 n 個 t 複本的序列收納器。
X a(n, t)	建立含有 n 個 t 複本的 a 序列收納器。
X(i, j)	建立區間為[i, j)的序列收納器。
X a(i, j)	建立區間為[i, j)的 a 序列收納器。
X(i1)	建立初值為 i1 的序列收納器。
a = (i1)	複製 i1 值給 a。
a.emplace(p, args)	使用帶有 args 參數的 T 建構函數，在 p 前面插入型態為 T 的物件。
a.insert(p, t)	將 t 的副本安插在 p 之前，回傳指向插入副本 t 的迭代器。t 的預設值是 T()，也就是說，此值是用於型態 T 未被明確初始化時。
a.insert(p, rv)	將 rv 的副本安插在 p 之前，回傳指向插入副本 t 的迭代器。可以使用移動語意。

成員函數	說明
a.insert(p, n, t)	將 t 的 n 份副本插入在 p 之前，無回傳值。
a.insert(p, i, j)	將區間 [i, j] 的元素副本置於 p 之前，無回傳值。
a.insert(p, i1)	如同 a.insert(p, i1.begin(), i1.end())。
a.resize(n)	若 n > a.size()，將 n – a.size() 個元素置於 a.end() 之前；在型態未被明確初始化時，此值用於每一新元素，其型態為 T。若 n < a.size()，則會清除第 n 個元素之後的元素。
a.resize(n, t)	若 n > a.size()，將 t 的 n – a.size() 份副本置於 a.end() 之前，t 的預設值是 T()，也就是說，此值是用於型態 T 未被明確初始化時。若 n < a.size()，則會清除第 n 個元素之後的元素。
a.assign(i, j)	將 a 目前的內容用區間 [i, j) 的元素副本取代。
a.assign(n, t)	將 a 目前的內容用 n 份 t 的副本取代。t 的預設值是 T()，也就是說，此值是用於型態 T 未被明確初始化時。
a.assign(i1)	如同 a.assign(i1.begin(), i1.end())。
a.erase(q)	清除 q 所指的元素，回傳迭代器，指向 q 之後的元素。
a.erase(q1, q2)	清除區間 [q1, q2] 的元素，回傳迭代器，指向 q2 原來所指的元素。
a.clear()	同 erase(a.begin(), a.end())。
a.front()	回傳 *a.begin()（第一個元素）。

表 G.6 列出一些序列類別中相同的成員函數（vector，forword_list，list 與 deque）。

表 G.6 處理某些序列的成員函數

成員函數	說明	收納器
a.back()	回傳 *--a.end()（最後一個元素）。	
a.push_back(t)	在 a.end() 之前插入 t。	vector, list,deque
a.push_back(rv)	在 a.end() 之前插入 t。可使用移動語意。	vector, list,deque
a.pop_back()	清除最後一個元素。	vector, list,deque

成員函數	說明	收納器
a.emplace_back(args)	使用帶有 args 參數的 T 建構函數，附加型態為 T 的物件。	vector, list,deque
a.push_front(t)	在第一個元素之前插入一份 t 的副本。	Forward_list, list,deque
a.push_front(rv)	在第一個元素之前插入一份 rv 的副本。可使用移動語意。	Forward_list, list,deque
a.emplace_front(t)	使用帶有 args 參數的 T 建構函數，加入一型態為 T 的物件於前端。	Forward_list, list,deque
a[n]	回傳 *(a.begin() + n)	vector,deque, array
a.at(n)	回傳 *(a.begin() + n)，若 n > a.size() 則丟出 out_of_range 異常。	vector,deque, array

vector 樣版還有其他成員函數，列於表 G.7。此處 a 是 vector 收納器，n 是整數的 X::size_type。

表 G.7 vector 的其他成員函數

成員函數	說明
a.capacity()	在不要求重新配置的情況下，回傳 vector 可以儲存的總元素個數。
a.reverse(n)	通知物件 a 至少需要 n 個元素的記憶體。在呼叫此成員函數之後，這容量至少是 n 個元素。若 n 大於目前的容量，則會產生重新配置。若 n 大於 a.max_size()，則此成員函數丟出 length_error 異常。

list 樣版還有其它的成員函數，列於表 G.8。此處 a 和 b 是 list 收納器，T 是儲存在 list 中的型態，如 int，t 是型態 T 之值，i 和 j 是輸入迭代器，q2 和 p 是迭代器，q 和 q1 是可提領的迭代器，而 n 是整數的 X::size_type。這表格使用標準的 STL 表示法 [i, j)，代表從 i 至 j，但不包括 j 的區間。

表 G.8 list 的其他成員函數

成員函數	說明
a.splice(p, b)	將 list b 的內容移至 list a，安置在 p 之前。
a.splice(p, b, i)	將 list b 中 i 所指的元素，移至 list a 的位置 p 之前。
a.splice(p, b, i, j)	將 list b 之區間 [i, j) 的元素移至 list a 的位置 p 之前。
a.remove(const T& t)	清除 list a 中值為 t 的所有元素。
a.remove_if(Predicate pred)	已知 i 是指向 list a 的迭代器，清除 list a 中 pred(*i) 為 true 的所有值（Predicate 是 Boolean 函數或是函數物件，如第 15 章的討論）。
a.unique()	對於每一群連續相同的元素，除第一個元素外，全部清除。
a.unique(BinaryPredicate bin_pred)	對於使 bin_pred(*i, *(i - 1)) 為 true 的每一群連續元素，除第一個元素以外全部清除（BinaryPredicate 是 Boolean 函數或是函數物件，如第 15 章的討論）。
a.merge(b)	利用定義於值型態的 < 運算子，合併 list b 和 list a 的內容。若 a 中的元素等於 b 中的元素，則 a 的元素置於前面。在合併之後，list b 變成空的。
a.merge(b, Compare comp)	使用 comp 函數或函數物件，合併 list b 和 list a 的內容。若 a 中的元素等於 b 中的元素，則 a 的元素置於前面。在合併之後，list b 變成空的。
a.sort()	使用 < 運算子，對 list a 排序。
a.sort(Compare comp)	使用 comp 函數或函數物件，對 list a 排序。
a.reverse()	反轉 list a 中的元素順序。

forward_list 的操作大致上相似。由於 forward_list 樣版類別迭代器無法往前，所以有些成員函數需要調整。因此，利用 insert_after()，erase_after()與 splice_after()取代 insert()，erase()與 splice()這些成員函數。這些作用點的位置是在迭代器的後面滿，而不是前面。

Set 和 Map 的其他成員

關聯式收納器(associative container)，如 set 和 map，有 Key 樣版參數和 Compare 樣版參數，分別表示用於排序內容的關鍵值型態，以及用於比較關鍵值的函數物件，稱為**比較物件**（comparison object）。對於 set 和 multiset 收納器，儲存的關鍵值就是儲存的值，所以關鍵值型態就是值的型態。對於 map 和 multimap 收納器，儲存值是結合一種型態（樣版參數 T）與關鍵值型態（樣版參數 Key），所以值的型態是 pair<const Key, T>。關聯式收納器有其他的成員來描述這些特性，如表 G.9 所列。

表 G.9 為關聯式收納器定義的型態

型態	值
X::key_type	Key，關鍵值型態
X::key_compare	Compare，這有預設值 less<key_type>
X::value_compare	這是二元的判斷型態，對於 set 和 multiset，這與 key_compare 相同，而它會提供 map 或 multimap 中 pair<const Key, T> 值的排序。
X::mapped_type	T，相關資料的型態（只用於 map 和 multimap）

關聯式收納器提供表 G.10 所列的成員函數。一般而言，比較物件不要求具有相同關鍵值之值要相同，名詞**相等關鍵值**（equivalent key）表示兩個值不一定相同，但有相同的關鍵值。在表格中，X 是收納器類別，a 是型態 X 的物件。若 X 使用唯一的關鍵值（也就是 set 或 map），a_uniq 是型態 X 的物件。若 X 使用多個關鍵值（也就是 multiset 或 multimap），則 a_eq 是型態 X 的物件。與之前一樣，i 和 j 是輸入迭代器，參考 value_type 的元素，[i, j) 是有效的區間，p 和 q2 是指向 a 的迭代器，q 和 q1 是指向 a 可提領的迭代器，[q1, q2) 是有效的區間，t 是 X::value_type 之值（這可能是一對值）。還有 k 是 X::key_type 之值。同時 i1 是 initializer_list<value_type>的物件。

表 G.10 set，multiset，map，和 multimap 定義的成員函數

成員函數	說明
X (i, j, c)	建立一空的收納器，從[i, j)插入元素，並且使用 c 當作比較物件。
X a(i, j, c)	建立一空的收納器 a，從[i, j)插入元素，並且使用 c 當作比較物件。
X(i, j)	建立一空的收納器，從[i, j)插入元素，並且使用 Copmare() 當作比較物件。
X a(i, j)	建立一空的收納器 a，從[i, j)插入元素，並且使用 Compare()當作比較物件。
X(i1)	如同 X(i1.begin(), i1.end())。
a = i1	指定 X(i1.begin(), i1.end())空間給 a。
a.key_comp()	回傳用在建構 a 的比較物件
a.value_comp()	回傳 value_compare 型態的物件
a_uniq.insert(t)	只有在 a 尚未包含具有相同關鍵值之元素時，才將值 t 插入收納器 a。這成員函數回傳型態 pair<iterator, bool> 之值。若發生插入，則 bool 部分為 true，否則為 false。迭代器部分指向關鍵值等於 t 之關鍵值的元素。
a_eq.insert(t)	插入 t 並回傳指向此位置的迭代器。
a.insert(p, t)	插入 t，p 是 insert() 開始搜尋之處。若 a 是具有唯一關鍵值的收納器，則當 a 不具有相同關鍵值的元素時，才會發生插入；其他則都會發生插入。不管是否發生插入，這成員函數都會回傳迭代器，指向具有相同關鍵值的位置。
a.insert(i, j)	將區間 [i, j) 的元素插入 a。
a.insert(i1)	從 initializer_list i1 插入元素到 a。
a_uniq.emplace(args)	如同 a_uniq.insert(t)，但是它是使用 T 建構函數，其參數列與 args 參數內容相匹配。
A_eq.emplace(args)	如同 a_eq.insert(t)，但是它是使用 T 建構函數，其參數列與 args 參數內容相匹配。
A.emplace_hint(args)	如同 a.insert(p, t)，但是它是使用 T 建構函數，其參數列與 args 參數內容相匹配。
a.erase(k)	清除 a 中關鍵值等於 k 的元素，並回傳清除的元素個數。

成員函數	說明
a.erase(q)	清除 q 所指的元素。
a.erase(q1, q2)	清除區間 [q1, q2) 中的元素。
a.clear()	等於 erase(a.begin(), a.end())。
a.find(k)	回傳迭代器，指向關鍵值等於 k 之元素；若找不到這種元素，則回傳 a.end()。
a.count(k)	回傳關鍵值等於 k 的元素個數。
a.lower_bound(k)	回傳指向關鍵值不小於 k 之第一個元素的迭代器。
a.upper_bound(k)	回傳指向關鍵值大於 k 之第一個元素的迭代器。
a.equal_range(k)	回傳數對，其第一個成員是 a.lower_bound(k)，第二個成員是 a.upper_bound(k)。
a.operator[](k)	回傳與關鍵值 k 關聯之值的 reference（只有 map 收納器）。

無序性關聯式容器

前面曾提及，**無序性關聯式容器**（unordered_set，unordered_multiset，unordered_map，unordered_multimap）是使用**鍵值**（key）與**雜湊表格**（hash table）來快速的擷取資料。讓我們來看這些概念。首先，利用**雜湊函數**（hash function）轉換鍵值為一索引值。例如，若鍵值是一 string ，雜湊函數將 string 中所有字元所對應的 ASCII 數值碼加總，然後除以 13，所得到的餘數（0 到 12）即為索引值。此時的無序性關式容器是以 13 個**桶子**（bucket）來儲存 string 的。若一 string 計算出來的索引值是 4，則它就是儲存在第 4 號的桶子上。假使您想要搜尋某鍵值的容器，則必需應用雜湊函數將鍵值轉為索引值所對應的桶子。理想上，我們應有足夠的桶子，使每個桶子可以儲存一些 string。

在 C++11 預設的函式庫中，提供 hash<Key>樣版來處理有關無序性關聯式的容器。對各種的整數與浮點數，指標及樣版類別（如 string），皆已定義了一些規格。

表 G.11 列出這些容器所使用的型態。

無序性關聯式容器的界面與關聯式容器相似。所以表 G.10 也可以應用於無序性關聯式的容器。其中 lower_bound()與 upper_bound()成員函數是不需要的，而 X(I, j, c)建構函數也不必要。事實上，一般性的關聯性容器是有序性，所以可使用比較來知道運算式小於的概念。而無序性的關聯式容器只能比較是否為相等的概念。

表 G.11 無序性關聯式收納器定義的型態

型態	值
X::key_type	Key，關鍵值型態
X::key_equal	Pred，它是二元判斷條件，用以測試兩個參數的 Key 是否相等。
X::hasher	Hash，這是單元函數物件型態，若 hf 是 Hash 型態，而且 k 是 key 型態。則 hf(k)是 std::size_t 型態。
X::local_iterator	有如 X::local_iterator 型態的迭代器。但只用於單一桶子內。
X::local_const_iteator	有如 X::local_const_iterator 型態的迭代器。但只用於單一桶子內。
X::mapped_type	T，相關資料的型態（只用於 map 和 multimap）

除了表 G.10 所列的成員函數外，無序性的關聯式容器包括更多的成員函數，如表 G.12 所列。表中的 X 是一無序性的關聯式容器的類別，a 是型態為 X 的物件，b 可能是型態為 X 的常數物件，a_uniq 是一型態為 unordered_set 或 unordered_map 的物件，a_eq 是一型態為 unordered_multiset 或 unordered_multimap 的物件，hf 是型態為 hasher 的值，eq 是型態為 key_equal 的值，n 是型態為 size_type 的值，而 z 是型態 float 的值。與之前一樣，i 和 j 是輸入迭代器，參考 value_type 的元素，[i, j] 是有效的區間，p 和 q2 是指向 a 的迭代器，q 和 q1 是指向 a 可提領的迭代器，[q1, q2) 是有效的區間，t 是 X::value_type 之值（這可能是一對值）。還有 k 是 X::key_type 之值。同時 i1 是 initializer_list<value_type>的物件。

表 G.12 無序性 set，multiset，map，和 multimap 額外定義的成員函數

成員函數	說明
X (n, hf, eq)	建立一至少有 n 個桶子的空，使用 hf 雜湊函數及 eq 當作鍵值相等的判斷式。若省略 eq，則使用 key_equal 當作鍵值相等的判斷式。若省略 hf，則使用 hasher()當作雜湊函數。
X a(n, hf, eq)	建立一至少有 n 個桶子的空，使用 hf 雜湊函數及 eq 當作鍵值相等的判斷式。若省略 eq，則使用 key_equal 當作鍵值相等的判斷式。若省略 hf，則使用 hasher()當作雜湊函數。

成員函數	說明
X (i, j, n, hf, eq)	建立一至少有 n 個桶子的空，使用 hf 雜湊函數、eq 當作鍵值相等的判斷式以及從從[i, j)插入元素。若省略 eq，則使用 key_equal 當作鍵值相等的判斷式。若省略 hf，則使用 hasher()當作雜湊函數。若 n 省略，則使用未指定數目的桶子。
X a(i, j, n, hf, eq)	建立一至少有 n 個桶子的空，使用 hf 雜湊函數、eq 當作鍵值相等的判斷式以及從從[i, j)插入元素。若省略 eq，則使用 key_equal 當作鍵值相等的判斷式。若省略 hf，則使用 hasher()當作雜湊函數。若 n 省略，則使用未指定數目的桶子。
b.hash_function()	回傳 b 所使用的雜湊函數。
b.key_eq()	回傳使用於 b 的鍵值相等判斷式。
b.bucket_count()	回傳 a 的桶子數目
b.max_bucket_count()	回傳 b 可能包含上限的桶子數目
b.bucket(k)	回傳包含鍵值為 k 之桶子的索引值。
b.bucket_size(n)	回傳索引值為 n 之桶子所包含的元素個數。
b.begin(n)	回傳一迭代器，它指向索引值為 n 之桶子的第一個元素。
b.end(n)	回傳一迭代器，它指向索引值為 n 之桶子的最後一個元素的下一個。
b.cbegin(n)	回傳一常數的迭代器，它指向索引值為 n 之桶子的第一個元素。
b.cend(n)	回傳一常數的迭代器，它指向索引值為 n 之桶子的最後一個元素的下一個。
b.load_factor()	回傳每一桶子的平均元素個數。
b.max_load_factor()	回傳負載因子的最大限制值。若負載因子超過此值，則容器將增加桶子的數目。
b.max_load_factor(z)	可能改變最大的負載因子，則使用 z 當作提示。

成員函數	說明
a.rehash(n)	重設桶子的個數為 >=n，它滿足 a.bucket_count() > a.size()/a.max_load.factor()。
a.reserve(n)	與 a.rehash(ceil(n/a.max_load_factor()) ，此處的 ceil(x)是大於等於 x 的最小正整數。

STL 函數

STL 演算法函式庫，定義在標頭檔 algorithm 和 numeric，提供許多非成員，以迭代器為基礎的樣版函數。如第 16 章的討論，從樣版參數的名稱就可看出參數要模擬的特殊概念。例如，ForwardIterator 是用來表示參數最少應該符合往前迭代器的需求，而 Predicate 是用來表示參數具有一個引數和 bool 回傳值的函數物件。C++ 標準將演算法分成 4 類：**非修改的序列操**作（non-modifying sequence operation），**變動序列操作**（mutating sequence operation），**排序**（sorting）和**關係運算子**（related operator），以及數值操作。（C++11 將數值操作從 STL 移到數值函數庫，但這些並不影響其用法）。名詞**序列操作**（sequence operation）表示函數以一對迭代器為引數，定義要操作的區間，或是序列。**變動**（mutating）的意思是函數可以改變收納器。

非修改的序列操作

表 G.13 概括**非修改**（nonmodifying）的序列操作。沒有顯示引數，多載的函數只列出一次。在表格之後是較完整的描述和原型。因此，您可以瀏覽表格，大致瞭解函數的功能，再閱讀感興趣的函數細節。

現在我們來仔細看一下有關非修改的序列操作。對於每一個函數的討論，我們都會顯示其原型，並且後面接著一段簡短的解釋。成對的迭代器表示區間，使用精選的樣版參數名稱表示迭代器的型態。一般的區間形式為 [first, last)，表示從 first 至 last 但不包括 last。有些函數使用兩個區間，不一定要在相同的收納器內。例如，您可用 equal() 比較 list 和 vector。將函數作為傳遞的引數是為函數物件，這可以是指標（函數名稱就是一個例子）或是已定義 () 操作的物件。如第 16 章所述，判斷函數（predicate）是具有一個引數的 Boolean 函數，而二元判斷函數是具有兩個引數的 Boolean 函數（這函數不一定是 bool 型態，只要回傳 0 值表 false，回傳非 0 值表 true 即可）。

表 G.13 非修改的序列操作

函數	說明
all_of()	對所有的元素，假使預測試驗為真，則回傳 true。（C++11）
any_of()	對任何的元素，假使預測試驗為真，則回傳 true。（C++11）
none_of()	對所有的元素，假使預測試驗為假，則回傳 true。（C++11）
for_each()	將非修改的函數物件，應用在區間的每個元素上。
find()	在區間中找尋某值的第一次出現之處。
find_if()	在區間中找尋第一個滿足預測試驗需求的值。
Find_if_not()	在區間中找尋第一個無法滿足預測試驗需求的值。（C++11）
find_end()	找出一序列值匹配第二個序列值的最後發生之處。匹配也許是相等或是應用一個二元判斷函數。
find_first_of()	在第二個序列中找出任何元素匹配第一個序列值的第一次發生之處。匹配也許是相等或是應用一個二元判斷函數。
adjacent_find()	找出在匹配元素之後的第一個元素。匹配也許是相等或是應用一個二元判斷函數。
count()	回傳特定值在區間中出現的次數。
count_if()	回傳特定值在一區間中匹配值的次數，匹配是根據二元判斷函數。

函數	說明
mismatch()	找出兩個區間中第一個不匹配的元素，並同時回傳這兩個迭代器。匹配也許是相等或是應用一個二元判斷函數。
equal()	若兩個區間的對應每個元素都相互匹配，則回傳 true。匹配也許是相等或是應用一個二元判斷函數。
is_permutation()	若兩個區間的對應每個元素都在一些排列上相互匹配，則回傳 true。匹配也許是相等或是應用一個二元判斷函數。
search()	找出一序列其值匹配第二序列值的第一次出現之處。匹配也許是相等或是應用一個二元判斷函數。
search_n()	找出 n 個元素中每個都匹配特定值的第一個序列。匹配也許是相等或是應用一個二元判斷函數。

all_of() (C++11)

```
template<class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last,
            Predicate pred);
```

對每一迭代器在區間 [first, last) 上，若 pred(*i)為真，或此區間是空的，則 all_of()函數將回傳 true。否則，回傳 false。

any_of() (C++11)

```
template<class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last,
            Predicate pred);
```

對每一迭代器在區間 [first, last) 上，若 pred(*i)為假，或此區間是空的，則 all_of()函數將回傳 false。否則，回傳 true。

none_of() (C++11)

```
template<class InputIterator, class Predicate>
bool none_of(InputIterator first, InputIterator last,
             Predicate pred);
```

對每一迭代器在區間 [first, last) 上，若 pred(*i)為假，或此區間是空的，則 none_of()函數將回傳 true。否則，回傳 false。

for_each()

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
                  Function f);
```

for_each() 函數將函數物件 f 應用在區間 [first, last)的每個元素上。它也回傳 f。

find()

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last,
                   const T& value);
```

find() 函數回傳一個迭代器，指向區間 [first, last) 中值為 value 的第一個元素。若找不到此項資料，回傳 last。

find_if()

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

find_if() 函數回傳一個迭代器 it，指向區間 [first, last) 中函數物件的呼叫 pred(*i) 之值為 true 的第一個元素。

find_if_not()

```
template<class InputIterator, class Predicate>
InputIterator find_if_not(InputIterator first, InputIterator last,
                          Predicate pred);
```

find_if_not() 函數回傳一個迭代器 it，指向區間 [first, last) 中函數物件的呼叫 pred(*i) 之值為 false 的第一個元素。若找不到項目，則回傳 last。

find_end()

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

find_end() 函數回傳迭代器 it，指向區間 [first1, last1) 中匹配區間 [first2, last2) 之起始序列中的最後元素。第一個版本用 == 運算子作為元素的比較方式。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說若 pred(*it1, *it2) 回傳 true，表示 it1 和 it2 所指的元素相互匹配。若找不到此項資料，兩者都回傳 last1。

find_first_of()

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

template<class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
ForwardIterator1 find_first_of(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2,
         BinaryPredicate pred);
```

find_first_of() 函數回傳迭代器 it，指向區間 [first1, last1) 中與區間 [first2, last2) 的任一元素相匹配的第一個元素。第一個版本用 == 運算子作為元素的比較方式。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說若 pred(*it1, *it2) 回傳 true，表示 it1 和 it2 所指的元素相互匹配。若找不到此項資料，兩者都回傳 last1。

adjacent_find()

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first,
                              ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first,
         ForwardIterator last, BinaryPredicate pred);
```

adjacent_find() 函數回傳迭代器 it，指向區間 [first1, last1) 中與其後面元素相匹配的第一個元素。若找不到這種數對則回傳 last1。第一個版本用 == 運算子作為元素的比較方式。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說若 pred(*it1, *it2) 回傳 true，表示 it1 和 it2 所指的元素相互匹配。

count()

```
template<class InputIterator, class T>
 typename iterator_traits<InputIterator>::difference_type
  count(InputIterator first, InputIterator last, const T& value);
```

count() 函數回傳區間 [first, last) 中與值 value 相匹配的元素個數。對於值型態用 == 運算子比較值。回傳型態是足以表示收納器可以儲存之最多元素的整數型態。

count_if()

```
template<class InputIterator, class Predicate>
 typename iterator_traits<InputIterator>::difference_type
  count_if(InputIterator first, InputIterator last, Predicate pred);
```

count_if() 函數回傳在區間 [first, last) 中使函數物件 pred 回傳 true 值的元素個數。

mismatch()

```
template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1,
             InputIterator1 last1, InputIterator2 first2);
template<class InputIterator1, class InputIterator2,
                               class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1,
             InputIterator1 last1, InputIterator2 first2,
             BinaryPredicate pred);
```

每個 mismatch() 函數是在區間 [first1, last1) 和以 first2 開始的區間中找尋第一個不匹配的對應元素，並回傳儲存兩個不匹配之元素的迭代器對。若找不到不匹配的元素，則回傳值是 pair<last1, first2 + (last1 - first1)>。第一個版本用 == 運算子來檢查是否匹配。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說若 pred(*it1, *it2) 回傳 false，表示 it1 和 it2 所指的元素不匹配。

equal()

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);
```

```
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);
```

若區間 [first1, last1) 和以 first2 開始的序列，其對應的每個元素都相等，則 equal() 函數回傳 true；不然回傳 false。第一個版本用 == 運算子來比較元素。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說若 pred(*it1, *it2) 回傳 true，表示 it1 和 it2 所指的元素相互匹配。

is_permutation() (C++11)

```
template<class InputIterator1, class InputIterator2>
bool is_permutation(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
bool is_permutation(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);
```

若區間 [first1, last1] 和以 first2 開始的序列，其對應的元素具有相同的排列，則 equal() 函數回傳 true；不然回傳 false。第一個版本用 == 運算子來比較元素。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說，若 pred(*it1, *it2) 回傳 true，表示 it1 和 it2 所指的元素相互匹配。

search()

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

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2,
                        BinaryPredicate pred);
```

search() 函數在區間 [first1, last1) 中尋找匹配區間 [first2, last2) 的第一個序列。若無這種序列，則回傳 last1。第一個版本用 == 運算子來比較元素。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說若 pred(*it1, *it2) 回傳 true，表示 it1 和 it2 所指的元素相互匹配。

search_n()

```
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value);

template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value, BinaryPredicate pred);
```

search_n() 函數在區間 [first, last) 中找尋匹配由 count 個連續 value 值組成之序列的第一次發生之處。若無這種序列，則回傳 last。第一個版本用 == 運算子來比較元素。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說若 pred(*it1, *it2) 回傳 true，表示 it1 和 it2 所指的元素相互匹配。

變動序列操作

表 G.14 概括變動序列操作。沒有顯示引數，多載的函數只列出一次。在表格之後是較完整的描述和原型。因此，您可以瀏覽表格，大致瞭解函數的功能，再閱讀感興趣的函數細節。

現在我們來仔細看一下有關變動序列操作。對於每一個函數的討論，我們都會顯示其原型，並且後面接著一段簡短的解釋。如之前的用法，成對的迭代器表示區間，使用精選的樣版參數名稱表示迭代器的型態。一般的區間形式為 [first, last)，表示從 first 至 last 但不包括 last。將函數作為傳遞的引數是為函數物件，這可以是函數指標或是已定義 () 操作的物件。如第 16 章所述，判斷函數是具有一個引數的 Boolean 函數，而二元判斷函數是具有兩個引數的 Boolean 函數（這函數不一定是 bool 型態，只要回傳 0 值表 false，回傳非 0 值表 true 即可）。還有與第 16 章一樣，一元函數物件是有單一引數的函數，二元函數是有兩個引數的函數。

copy()

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

copy() 函數複製區間 [first, last) 的元素至區間 [result, result + (last - first)) 中。它回傳 result + (last - first)，也就是指向最後複製元素之後一個位置的迭代器。這函數要求 result 不能在區間 [first, last) 中，也就是說目的區間不能與原始區間重疊。

表 G.14 變動序列操作

函數	說明
copy()	從區間複製元素至迭代器所指定的位置。
copy_backward()	從區間複製元素至迭代器所指定的位置。從區間的尾端開始往後進行複製。
swap()	交換兩個 reference 所指定位置的內容。
swap_ranges()	交換兩個區間的對應元素。
iter_swap()	交換兩個迭代器指定之位置所儲存的值。
transform()	將一個函數物件應用在區間的每一個元素上（或是一對區間的每對元素上），將回傳值複製到另一個區間的對應位置中。
replace()	將區間中某一值的出現處都用另一值取代。
replace_if()	在一區間中，將判斷函數物件用在原始值上，若回傳 true，則用另一值取代此值。
replace_copy()	將一個區間複製到另一個區間，將某一特定值的出現處用另一值取代。
replace_copy_if()	將一個區間複製到另一個區間，若每一個值其判斷函數物件之值為 true 則用指定的值取代之。
fill()	將區間中的每一個值都指定為指定的值。
fill_n()	將 n 個連續元素都設為指定值。
generate()	將區間的每一個值都指定為 generator 的回傳值，這是無引數的函數物件。
generate_n()	將區間的前 n 個值都指定為 generator 的回傳值，這是無引數的函數物件。
remove()	移除區間中等於指定值的所有元素，並回傳指向結果區間之 past-the-end 的迭代器。
remove_if()	移除區間中使判斷函數物件回傳 true 的元素，並回傳指向結果區間之 past-the-end 的迭代器。

函數	說明
remove_copy()	將一區間的元素複製到另一個區間，略過等於指定值的元素。
remove_copy_if()	將一區間的元素複製到另一個區間，略過判斷函數物件回傳 true 的元素。
unique()	將區間中兩個以上的相等元素序列減少為一個元素。
unique_copy()	將一區間的元素複製到另一個區間，將兩個以上的相等元素減少為一個元素。
reverse()	反轉區間中的元素。
reverse_copy()	以相反的順序將一區間複製到另一區間。
rotate()	將區間視為環狀排序，元素往左旋轉。
rotate_copy ()	以旋轉的順序將一區間複製到另一區間。
random_shuffle()	隨意的重新排列區間中的元素。
reverse()	倒轉區間的元素。
reverse_copy()	以相反方向的順序複製一區間的元素於另一區間。
rotate()	由左至右以圓形順序看待一區間
rotate_copy()	以旋轉的順序複製一區間的元素於另一區間。
random_shuffle()	隨意排列區間元素。
shuffle()	利用一函數物件型態滿足 C++11 需求的均勻分配產生器，在區間隨機安排元素的順序。(C++11)
is_partitioned()	若一區間被一給定的預測所分割，則此函數回傳 true 。
partition()	將滿足判斷函數物件的所有元素置於不滿足的元素之前。
stable_partition()	將滿足判斷函數物件的所有元素置於不滿足的元素之前。在每一群中，保留元素的相對順序。
partition_copy()	複製滿足預測函數物件的所有元素於輸出區間，其餘的元素複製到另一區間。 (C++11)
partition_point()	在一給定的預測下分割區間，其回傳一迭代器，它指向不滿足預測的第一個元素。

copy_n() (C++11)

```
template<class InputIterator, class Size class OutputIterator>
OutputIterator copy(InputIterator first, Size n,
                    OutputIterator result);
```

copy_n() 函數從 first 位置開始，複製 n 個元素至[result, result + (last - first))。元素至區間中。它回傳 result + (last - first)，也就是指向最後複製元素之後一個位置的迭代器。這函數要求 result 不能在區間 [first, last) 中，也就是說目的區間不能與原始區間重疊。

copy_if() (C++11)

```
template<class InputIterator, class OutputIterator,
         class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
                       OutputIterator result, Predicate pred);
```

若 pred(*i)爲真，copy_if() 函數將經由在迭代器 i，複製區間 [first, last) 的元素至區間 [result, result + (last - first))。它回傳 result + (last - first)，也就是指向最後複製元素之後一個位置的迭代器。這函數要求 result 不能在區間 [first, last) 中，也就是說目的區間不能與原始區間重疊。

copy_backward()

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

copy_backward() 函數複製區間 [first, last) 的元素至區間 [result - (last - first), result) 中。從元素 last - 1 開始複製到位置 result - 1，並從此往後進行至 first。它回傳 result - (last - first)，也就是指向最後複製元素之後一個位置的迭代器。這函數要求 result 不能在區間 [first, last)中，但是因為複製是往後執行，所以目的區間和原始區間可能重疊。

move() (C++11)

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

move() 函數使用 std::move() 將區間 [first, last) 的元素移至區間 [result, result + (last - first)) 中。它回傳 result + (last - first)，也就是指向最後複製元素之後一個位置的迭代器。這函數要求 result 不能在區間 [first, last) 中，也就是說目的區間不能與原始區間重疊。

move_backward() (C++11)

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

move_backward() 函數移動區間 [first, last) 的元素至區間 [result - (last - first), result) 中。從元素 last - 1 開始複製到位置 result - 1，並從此往後進行至 first。它回傳 result - (last - first)，也就是指向最後複製元素之後一個位置的迭代器。這函數要求 result 不能在區間 [first, last)中，但是因為複製是往後執行，所以目的區間和原始區間可能重疊。

swap()

```
template<class T> void swap(T& a, T& b);
```

swap() 函數交換兩個由 reference 所指定位置的值。

swap_ranges()

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

swap_ranges() 函數交換區間 [first1, last1) 和以 first2 起始區間的對應元素。這兩個區間不能重疊。

iter_swap()

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

iter_swap() 函數交換兩個由迭代器所指定位置的值。

transform()

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
OutputIterator result, UnaryOperation op);
```

```
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryOperation binary_op);
```

第一個版本的 transform() 將一元函數物件 op 應用在區間 [first, last) 的每個元素上，並將回傳值指定至以 result 起始之區間的對應元素。所以 *result 會設為 op(*first)，等依此類堆。它回傳 result + (last - first)，也就是目的區間之 past-the-end 的值。

第二個版本的 transform() 將二元函數物件 op 應用在區間 [first1, last1) 以及 [first2, last2) 的每個元素以上，並將回傳值指定至以 result 起始之區間的對應元素上。所以 *result 會指定為 op(*first1, *first2)，等依此類推。它回傳 result + (last - first)，也就是目的區間之 past-the-end 的值。

replace()

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

replace() 函數將區間 [first, last) 中出現 old_value 之處都用 new_value 值取代。

replace_if()

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);
```

replace_if() 函數將區間 [first, last) 中每個使 pred(old) 回傳 true 的 old 值用 new_value 值取代。

replace_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
     OutputIterator result,const T& old_ value, const T& new_ value);
```

replace_copy() 函數將區間 [first, last) 中的元素複製到以 result 起始的區間，但是用 new_value 取代每一個 old_value 值。它回傳 result + (last - first)，即目的區間的 past-the-end 值。

replace_copy_if()

```
template<class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
    OutputIterator result, Predicate pred, const T& new_ value);
```

replace_copy_if() 函數將區間 [first, last) 中的元素複製到以 result 起始的區間，但是用 new_value 取代每一個使 pred(old) 回傳 true 的 old 值用 new_value 值取代。它回傳 result + (last - first)，即目的區間的 past-the-end 值。

fill()

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

fill() 函數將區間 [first, last) 中的每個元素設為 value。

fill_n()

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value);
```

fill_n() 函數將從位置 first 開始的 n 個元素都設為 value。

generator()

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

generator() 函數將區間 [first, last) 中的每個元素都設為 gen()，此處 gen 是 generator 函數物件，也就是沒有引數的函數。例如，gen 可以是指向 rand() 的指標。

generator_n()

```
template<class OutputIterator, class Size, class Generator>
void generate_n(OutputIterator first, Size n, Generator gen);
```

generator_n() 函數將以 first 開始之區間的 n 個元素都設為 gen()，此處 gen 是 generator 函數物件，也就是沒有引數的函數。例如，gen 可以是指向 rand() 的指標。

remove()

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& value);
```

remove() 函數移除區間 [first, last) 中所有的 value，並回傳指向結果區間之 past-the-end 的迭代器。這函數是穩定的，意思是未移除元素的順序是不變的。

因為各種 remove() 和 unique() 函數都不是成員函數，而且不受限於 STL 的收納器，所以它們不會重設收納器的大小。相反的，它們會回傳迭代器指向新的 past-the-end 的位置。一般而言移除的元素只是移往收納器的尾端。但是對於 STL 收納器您可以用回傳的迭代器，以及 eras() 成員函數重設 end()。

remove_if()

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

remove_if() 函數移除區間 [first, last) 中所有使 pred(val) 回傳 true 的 val 值，並回傳指向結果區間之 past-the-end 的迭代器。這函數是穩定的，意思是未移除元素的順序是不變的。

remove_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& value);
```

remove_copy() 函數將區間 [first, last) 中的值複製至以 result 為起始的區間，在複製時會略過 value 值的元素。它會回傳指向結果區間之 past-the-end 的迭代器。這函數是穩定的，意思是未移除元素的順序是不變的。

remove_copy_if()

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred);
```

remove_copy_if() 函數將區間 [first, last) 複製至以 result 為起始的區間，在複製時會略過使 pred(val) 回傳 true 的 val 值。它會回傳指向結果區間之 past-the-end 的迭代器。這函數是穩定的，意思是未移除元素的順序是不變的。

unique()

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate pred);
```

unique() 函數將區間 [first, last) 中含有兩個以上相等元素的序列減少為單一元素，並回傳指向新區間之 past-the-end 的迭代器。第一個版本用 == 運算子比較元素。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說若 pred(*it1, *it2) 回傳 true，表示 it1 和 it2 所指的元素相互匹配。

unique_copy()

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

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryPredicate pred);
```

unique_copy() 將區間 [first, last) 中的元素複製至以 result 起始的區間，並將含有兩個以上相等元素的每一序列減少為單一元素，它回傳指向新區間之 past-the-end 的迭代器。第一個版本用 == 運算子作為值型態的比較方式。第二個版本用二元的判斷函數物件 pred 來比較元素。也就是說若 pred(*it1, *it2) 回傳 true，表示 it1 和 it2 所指的元素相互匹配。

reverse()

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

reverse() 函數反轉區間 [first, last) 中的元素順序，作法是呼叫 swap(first, last - 1)，等依此類推。

reverse_copy()

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result);
```

reverse_copy() 函數以相反的順序將區間 [first, last) 中的元素複製至以 result 起始的區間。這兩個區間不應重疊。

rotate()

```
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);
```

rotate() 函數對區間 [first, last) 中的元素執行向左旋轉。在 middle 的元素移至 first，在 middle + 1 的元素移至 first + 1，等等。在 middle 之前的元素會繞回收納器的尾端，所以在 first 的元素會在之前是 last - 1 的元素之後。

rotate_copy()

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);
```

rotate_copy() 函數對區間 [first, last) 中的元素以 rotate() 執行旋轉，並將結果複製至以 result 起始的區間。

random_shuffle()

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

這版本的 random_shuffle() 函數會打亂區間 [first, last) 中的元素。這分佈是均勻的，也就是說原始順序的各種排列組合都有同等的機會。

random_shuffle()

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

這版本的 random_shuffle() 函數會打亂區間 [first, last] 中的元素。函數物件決定分佈的方式。若有 n 個元素，運算式 random(n) 應會回傳區間 [0, n] 之間的值。在 C++98，random 參數是 lvalue reference，而在 C++11，它是 rvalue reference。

shuffle()

```
template<class RandomAccessIterator, class Uniform RandomNumberGenerator>
void shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    UniformRandomNumberGenerator&& rgen);
```

這版本的 shuffle() 函數會打亂區間 [first, last] 中的元素。函數物件 rgen 的型態應和 C++11 標準所定義的均勻亂數產生器的需求一致。rgen 決定分佈的方式。若有 n 個元素，運算式 rgen(n) 應會回傳區間 [0, n] 之間的值。

is_partitioned() (C++11)

```
template<class InputIterator, class Predicate>
bool is_partitioned(InputIterator first,
                    InputIterator last, Predicate pred);
```

若區間是空的或它是經由 pred 加以劃分，is_partition() 函數將回傳 true。否則，此函數回傳 false。

partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred);
```

partition() 函數將可使 pred(val) 回傳 true 的 val 值元素，置於所有不能符合此測試的元素之前。它會回傳迭代器，指向最後一個使判斷函數物件回傳 true 之值的下一個位置。

stable_partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
                                       BidirectionalIterator last,
                                       Predicate pred);
```

stable_partition() 函數將可使 pred(val) 回傳 true 的 val 值元素，置於所有不能符合此測試的元素之前。在這兩群元素中，這函數保留元素原來的相對順序。它會回傳迭代器，指向最後一個使判斷函數物件回傳 true 之值的下一個位置。

partition_copy() (C++11)

```
template<class InputIterator, class OutputIterator1,
         classs OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2> partition_copy(
     InputIterator first, InputIterator last,
     OutputIterator1 out_true, OutputIterator2 out_false
                               Predicate pred);
```

partition_copy() 函數將複製 pred(val) 回傳 true 的 val 值元素，到以 out_true 為首的區間。並將其餘的元素複製到以 out_false 為首的區間。此函數回傳一組物件，一為指向區間尾端的迭代器，它是以 out_true 為首回傳迭代器； 二為一為指向區間尾端的迭代器，它是以 out_false 為首回傳迭代器。

partition_point() (C++11)

```
template<class ForwardIterator, class Predicate>
ForewardIterator partition_point(ForwardIterator first,
                                 ForwardIterator last,
                                 Predicate pred);
```

partition_point() 函數需要經由 pred 分割的區間。它會回傳迭代器，指向最後一個使判斷函數物件回傳 true 的位置。

排序和關係操作

表 G.15 概括排序和關係操作。沒有顯示引數，多載的函數只列出一次。每個函數都有一個版本使用 < 來對元素排序，以及一個版本用比較函數物件作元素排序。在表格之後是較完整的描述和原型。因此，您可以瀏覽表格，大致瞭解函數的功能，再閱讀感興趣的函數細節。

表 G.15 排序和關係操作

函數	說明
sort()	對一區間排序。
stable_sort()	對一區間排序，保留等值元素的相對順序
partial_sort()	對一區間作部分排序，對前 n 個元素作完整排序。
partial_sort_copy()	將部分排序的區間複製到另一個區間。
nth_element()	從迭代器指定的區間中，找出若區間排序後元素應該放置的位置，並將元素置於該處。
lower_bound()	已知一值，為了維持順序，找出在排序區間中此值可以插入的第一個位置。
upper_bound()	已知一值，為了維持順序，找出在排序區間中此值可以插入的最後一個位置。
equal_range()	已知一值，找出已排序區間的最大子區間，使得此值可以放在此子區間的所有元素之前，而不會破壞此順序。
binary_search()	若排序的區間包含一值等於指定的值，則回傳 true，否則回傳 false。

函數	說明
merge()	將兩個排序的區間合併成第三個區間。
Inplace_merge()	就地合併兩個連續的排序區間。
includes()	若一集合的每個元素都可在另一個集合中找到，則回傳 true。
set_union()	建構兩個集合的聯集，這是包含兩個集合所有元素的集合。
set_intersection()	建構兩個集合的交集，這是包含兩個集合共同元素的集合。
set_difference()	建構兩個集合的差集，這是包含在第一個集合但不在第二個集合之元素的集合。
set_symmetric_difference()	建構一個集合包含在一個集合中但不在另一集合中的元素。
make_heap()	將區間轉成 heap。
push_heap()	將元素加入 heap 中。
pop_heap()	從 heap 中移出最大的元素。
sort_heap()	對 heap 排序。
is_heap()	若一區間是一 heap，則回傳 true 。 (C++11)
is_heap_until()	若區間是一 heap，則回傳最後的迭代器。 (C++11)
min()	回傳兩值中較小者。
max()	回傳兩值中較大者。
minmax()	回傳一 pair 物件，包含兩個參數值，為了增加大小或回傳在 initializer_list 參數的最大和最小項目。 (C++11)
min_element()	在區間中找尋最小值第一次出現之處。
max_element()	在區間中找尋最大值第一次出現之處。
minmax_element()	回傳一 pair 物件，包含一迭代器，其指向區間第一次出現的最小值，以及另一迭代器，指向區間最後出現的。 (C++11)

函數	說明
lexicographic_compare()	以字典編纂的方式比較兩個序列，若第一個序列小於第二個則回傳 true，否則回傳 false。
next_permutation()	對一序列產生下一個排列組合。
previous_permutation()	對一序列產生前一個排列組合。

本節的函數是用定義於元素的 < 運算子或是由樣版型態 Compare 設計的比較物件決定兩個元素的順序。若 comp 是型態 Compare 的物件，則 comp(a, b) 是 a < b 的一般化，在此排序方法中若 a 在 b 之前，則回傳 true。若 a < b 是 false 且 b < a 也是 false，則 a 和 b 等值。比較物件必須提供至少是**嚴格弱排序**（strict weak ordering）。這意義如下：

- 運算式 comp(a, a) 必須是 false，這一般化的觀念是一值不能小於自己（這是嚴格的部分）。
- 若 comp(a, b) 為 true 且 comp(b, c) 為 true，則 comp(a, c) 也是 true（這意思是比較有遞移性）。
- 若 a 和 b 等值，且 b 和 c 等值，則 a 和 c 等值（這意思是等值有遞移性）。

若要將 < 用在整數上，則等值（equivalency）隱含相等（equality），但是對於較一般性的情況則不一定成立。例如，您可以定義一個具有數個描述郵寄地址的結構，並定義 comp 比較物件根據郵遞區號排序結構。則兩個具有相同郵遞區號的位址是等值而非相同。

現在我們來仔細看一下有關排序和關係的操作。對於每一個函數的討論，我們都會顯示其原型，並且後面接著一段簡短的解釋。我們將此節分成數小節。如之前的用法，成對的迭代器表示區間，使用精選的樣版參數名稱表示迭代器的型態。一般的區間形式為 [first, last)，表示從 first 至 last 但不包括 last。將函數作為傳遞的引數是為函數物件，這可以是函數指標或是已定義 () 操作的物件。如第 16 章所述，判斷函數是具有一個引數的 Boolean 函數，而二元判斷函數是具有兩個引數的 Boolean 函數（這函數不一定是 bool 型態，只要回傳 0 值表 false，回傳非 0 值表 true 即可）。還有與第 16 章一樣，一元函數物件是有單一引數的函數，二元函數是有兩個引數的函數。

排序

首先是排序演算法。

sort()

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

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

sort() 函數用值型態的 < 運算子，以遞增的方式排序區間 [first, last) 中的元素。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

stable_sort()

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

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

stable_sort() 函數對區間 [first, last) 排序，但保留等值元素的相對順序。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

partial_sort()

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

template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
RandomAccessIterator last, Compare comp);
```

partial_sort() 函數對區間 [first, last) 作部分排序。排序區間的前 middle – first 個元素置於區間 [first, middle) 中，剩餘的元素則未排序。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

partial_sort_copy()

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first,
                        InputIterator last,
```

```
                        RandomAccessIterator result_first,
                        RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last,
                  Compare comp);
```

partial_sort_copy() 函數將排序的區間 [first, last) 複製前 n 個元素至區間 [result_first, result_first + n)。n 值是 last – first 和 result_last – result_first 中較小者。這函數回傳 result_first + n。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

is_sorted() (C++11)

```
template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last
               Compare comp);
```

若區間 [first, last) 已排序好，is_sorted() 函數將回傳 true；否則，回傳 false。其中第一版本使用 < ，而第二版本使用比較物件 comp 來決定小順序。

is_sorted_until() (C++11)

```
template<class ForwardIterator>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last
                                Compare comp);
```

若區間 [first, last) 小於兩個元素，is_sorted_until() 函數將回傳 last；否則，回傳最後的迭代器 it，而且若區間 [first, last) 是已排序好的。第一版本使用 < ，而第二版本使用比較物件 comp 來決定小順序。

nth_element()

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

```
template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
RandomAccessIterator last, Compare comp);
```

nth_element() 函數找出在區間 [first, last) 中排好順序後的第 n 個元素，然後將此元素置於第 n 個位置。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

二元搜尋

二元搜尋的演算法是假設區間已經排序。它們只要求一個往前的迭代器，但是隨機迭代器則最有效率。

lower_bound()

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

lower_bound() 函數在排序的區間 [first, last) 中找出第一個可將 value 插入的位置，而不破壞順序。它回傳迭代器指向此位置。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

upper_bound()

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
```

upper_bound() 函數在排序的區間 [first, last) 中找出最後一個可將 value 插入的位置，而不破壞順序。它回傳迭代器指向此位置。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

equal_range()

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first, ForwardIterator last, const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first, ForwardIterator last, const T& value,
    Compare comp);
```

equal_range() 函數在排序的區間 [first, last) 中找出最大的子空區間 [it1, it2)，使得 value 可以放在此區間的任何迭代器之前，都不會破壞順序。這函數回傳由 it1 和 it2 形成的 pair。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

binary_search()

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value, Compare comp);
```

若在排序的區間 [first, last) 中找到 value 的等值元素，則 binary_search() 函數回傳 true，否則回傳 false。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

若用 < 來排序，則當 a < b 和 b < a 均為 false 時，a 和 b 才是等值。對於一般數字，等值隱含相等，但是對於只根據一個成員作排序的結構則不成立。因此一個新值可能可以插入許多位置，仍維持資料的排序。同樣的，若用比較物件 comp 作排序，則等值表示 comp (a, b) 和 comp(b, a) 均為 false（這敘述的一般化是若 a 不小於 b 且 b 不小於 a，則 a 和 b 等值）。

合併

合併函數假設區間都已經排序。

merge()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator,
```

```
class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

merge() 函數合併排序區間 [first1, last1) 和 [first2, last2) 的元素，將結果放在以 result 為起始的區間。目的區間不應與這兩個區間重疊。在這兩區間若發現等值元素，則第一個區間的元素會置於第二個區間的元素之前。回傳值是指向結果區間之 past-the-end 的迭代器。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

inplace_merge()

```
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
     BidirectionalIterator middle, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
     BidirectionalIterator middle, BidirectionalIterator last,
     Compare comp);
```

inplace_merge() 函數合併兩個連續的排序區間 — [first1, middle) 和 [middle, last) — 成單一的排序序列存在區間 [first, last)。若有等值元素，則第一區間的元素會置於第二區間的元素之前。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

使用集合操作

集合操作是處理所有的排序序列，包括 set 和 multiset。對於儲存一值之一個以上實體的收納器，如 multiset，這定義都是一般化。兩個 multiset 的聯集是包含每個元素出現的較多次數，交集是包含每個元素出現的較少次數。例如，假設 multiset A 包含字串 "apple" 7 次，multiset B 包含相同字串 4 次。則 A 和 B 的聯集會包含 "apple" 的 7 個實體，而交集會包含 4 個實體。

includes()

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2, Compare comp);
```

若區間 [first2, last2) 的每個元素都可以在區間 [first1, last1) 中找到，則 includes() 函數回傳 true；否則回傳 false。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

set_union()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
```

set_union() 函數是將區間 [first1, last1) 和 [first2, last2) 的聯集複製到 result 所指的位置。這結果區間不應與任一個原始區間重疊。這函數回傳指向結果區間之 past-the-end 的迭代器。聯集是包含出現在兩個集合或任一集合之所有元素的集合。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

set_intersection()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
```

set_intersection() 函數是將區間 [first1, last1) 和 [first2, last2) 的交集複製到 result 所指的位置。結果區間不應與任一個原始區間重疊。這函數回傳指向結果區間之 past-the-end 的迭代器。交集是包含兩個集合之共同元素的集合。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

set_difference()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result, Compare comp);
```

set_difference() 函數是將區間 [first1, last1) 和 [first2, last2) 的差集複製到 result 所指的位置。結果區間不應與任一個原始區間重疊。這函數回傳指向結果區間之 past-the-end 的迭代器。差集是包含出現在第一個集合但是不在第二個集合的元素。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

set_symmetric_difference()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_symmetric_difference(
                        InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator,
         class Compare>
OutputIterator set_symmetric_difference(
                        InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result, Compare comp);
```

set_symmetric_difference() 函數將區間 [first1, last1) 和 [first2, last2) 的對稱差集複製到 result 所指的位置。結果區間不應與任一個原始區間重疊。這函數回傳指向結果區間之 past-the-end 的迭代器。對稱差集是包含出現在第一個集合但是不在第二個集合或是出現在第二個集合但不在第一個集合的元素。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

使用 Heap 操作

heap 是常見的資料形式，其性質是在 heap 中的第一個元素是最大的。當移除第一個元素時，或是加入任何新元素，則 heap 也許必須重新排列以維持此性質。heap 的設計是為了使這兩個操作可以有效率地完成。

make_heap()

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

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

make_heap() 函數使區間 [first, last) 成為 heap。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

push_heap()

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

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

push_heap() 函數假設區間 [first, last - 1) 是有效的 heap，並將此值新增至 heap 的位置 last - 1（也就是超過這有效 heap 的終點一個位置），使 [first, last) 成為有效的 heap。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

pop_heap()

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

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

pop_heap() 函數假設區間 [first, last) 是有效的 heap。它交換位置 last - 1 和 first 的值，並使區間 [first, last - 1) 成為有效的 heap。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

sort_heap()

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

template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

sort_heap() 函數假設區間 [first, last) 是有效的 heap 並排序之。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

is_heap() (C++11)

```
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last
               Compare comp);
```

若區間 [first, last]是一堆積 (heap)，則 is_heap() 函數將回傳 true，否則，回傳 false。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

is_heap_until() (C++11)

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

template<class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(
             RandomAccessIterator first, RandomAccessIterator last
             Compare comp);
```

若區間 [first, last]是少於兩個元素，則 is_heap_until() 函數將回傳 last，否則回傳最後的迭代器 it，其區間 [first, last)是一堆積。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

尋找 Minimum 和 Maximum 數值

minimum 和 maximum 函數回傳成對值和序列值的最小值和最大值。

min()

```
template<class T> const T& min(const T& a, const T& b);

template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```

min() 函數回傳兩值中的較小者。若兩個值是等值，它回傳第一個值。第一個版本用 < 決定排序，第二個版本用 comp 比較物件。

```
template<class T> T min(initializer_list<T> t);

template<class T, class Compare>
T min(initializer_list<T> t), Compare comp);
```

這個 min() 函數版本 (C++11) 在回傳初始串列 t 最小值。若有兩個或多個是等值時，將回傳第一個值。第一個版本用 <，第二個版本用比較物件 comp 決定順序。

max()

```
template<class T> const T& max(const T& a, const T& b);

template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

max() 函數回傳兩值中的較大者。若兩個值是等值，它回傳第一個值。第一個版本用 < 決定排序，第二個版本用 comp 比較物件。

```
template<class T> T max(initializer_list<T> t);

template<class T, class Compare>
T max(initializer_list<T> t), Compare comp);
```

這個 max() 函數版本 （C++11）回傳初始串列 t 最大值。若兩個或多個是等值，它回傳第一個值。第一個版本用 < 決定排序，第二個版本用 comp 比較物件。

minmax() (C++11)

```
template<class T>
  pair<const T&,const T&> minmax(const T& a, const T& b);

template<class T, class Compare>
  pair<const T&,const T&> minmax(const T& a, const T& b,
                                 Compare comp);
```

這一 minmax() 函數的版本，若 b 小於 a，則回傳 (b,a)，反之，回傳 (a,b) 。第一個版本用 < 決定排序，第二個版本用 comp 比較物件。

```
template<class T> pair<T,T> minmax(initializer_list<T> t);

template<class T, class Compare>
pair<T,T> minmax(initializer_list<T> t), Compare comp);
```

這一 minmax() 函數的版本，回傳在初始串列最小和最大元素的複本。若有多個元素和最小值相等，則回傳第一次出現的值。若有多個和最大值相等，則回傳最後出現的值 。第一個版本用 < 決定排序，第二個版本用 comp 比較物件。

min_element()

```
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

min_element() 函數回傳指向區間 [first, last) 中第一個迭代器 it，使得此區間沒有其他元素小於 *it。第一個版本用 < 決定排序，第二個版本用 comp 比較物件。

max_element()

```
template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

max_element() 函數回傳指向區間 [first, last) 中第一個迭代器，使得 *it 不小於此區間的其他元素。第一個版本用 < 決定排序，第二個版本用 comp 比較物件。

minmax_element() (C++11)

```
template<class ForwardIterator>
  pair<ForwardIterator,ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  pair<ForwardIterator,ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last,
                   Compare comp);
```

minmax_element() 函數回傳一組物件，包含指向區間 [first, last] 中第一個迭代器 it1，使得 *it1 不小於此區間的其他元素。包含指向區間 [first, last] 中最後一個迭代器 it2，使得 *it2 不小於此區間的其他元素。第一個版本用 < 決定排序，第二個版本用 comp 比較物件。

lexicographical_compare()

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             Compare comp);
```

若區間 [first1, last1) 中的元素序列在字典編纂的順序小於區間 [first2, last2) 中的元素序列，則 lexicographical_compare() 函數回傳 true；否則回傳 false。字典編纂順序是比較第一個序列的第一個元素和第二個序列的第一個元素，亦即比較 *first1 和 *first2。若 *first1 小於 *first2，則函數回傳 true。若 *first2 小於 *first1，則函數回傳 false。若兩元素等值，則進行下一個元素的比較。這程序不斷執行直到兩個對應元素不等值或是一序列結束。若兩個序列直到一序列結束都是等值，則較短者較小。若兩個序列等值且長度相同，也不是小於，所以函數回傳 false。第一個版本的函數用 < 比較元素，第二個版本用 comp 比較物件。字典編纂的比較是子母比較的一般化。

使用排列組合

序列的排列組合是元素的重新排序。例如，3 個元素的序列有 6 種可能的排序方式，因為對於第一個位置有 3 個選擇，之後對於第二個位置只有 2 個選擇，在第三個位置就只有一個選擇。例如，數字 1，2，3 的 6 種排列如下：

123 132 213 232 312 321

一般來說，n 個元素的序列有 $n \times (n\text{-}1) \times .. \times 1$，或是 $n!$ 種可能的排列組合。

排列組合函數的前提是，所有排列組合形成的集合，能夠以字典編纂排序，如上面 6 個排列組合的例子。這意思是每一個排列組合的前面和後面都會有一個特定的排列組合。例如 213 在 232 之前，312 在 232 之後。但是第一個排列組合（如例子的 123）沒有前者，而最後一個排列組合（如例子的 321）沒有後者。

next_permutation()

```
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);
```

next_permutation() 函數將區間 [first, last) 的序列轉換成字典編纂順序的下一個排列組合。若下一個排列組合存在，則函數回傳 true。若不存在（也就是說，此區間含有字典編纂順序的最後一個排列組合），則函數回傳 false，並將此區間轉換成字典編纂順序的第一個排列組合。第一個版本的函數用 < 比較元素，第二個版本用 comp 比較物件。

prev_permutation()

```
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);
```

prev_permutation() 函數將區間 [first, last) 的序列轉換成字典編纂順序的前一個排列組合。若前一個排列組合存在，則函數回傳 true。若不存在（也就是說，此區間含有字典編纂順序的第一個排列組合），則函數回傳 false，並將此區間轉換成字典編纂順序的最後一個排列組合。第一個版本的函數用 < 比較元素，第二個版本用 comp 比較物件。

數字操作

表 G.16 概括數字操作，這描述在 numeric 標頭檔。沒有顯示引數，多載的函數只列出一次。每個函數都有一個版本使用 < 來對元素排序，以及一個版本用比較函數物件作元素的排序。在表格之後是較完整的描述和原型。因此，您可以瀏覽表格，大致瞭解函數的功能，再閱讀感興趣的函數細節。

表 G.16 數字操作

函數	說明
accumulate()	計算區間中所有值的累計總和。
inner_product()	計算兩個區間的內積。
partial_sum()	將一個區間的部分和複製到第二個區間。
adjacent_difference()	計算一個區間中相鄰元素的差，將結果複製到第二個區間。
iota()	利用++運算子，指定連續值給一區間內的元素(C++11)。

現在我們來仔細看一下有關數字操作。對於每一個函數的討論，我們都會顯示其原型，並且後面接著一段簡短的解釋。

accumulate()

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

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);
```

accumulate() 函數將值 acc 初始化為 init，然後對區間 [first, last) 的每個迭代器 i 循序執行 acc = acc + *i（第一個版本），或是 acc = binary_op(acc, *i)（第二個版本）。然後回傳 acc 的結果值。

inner_product()

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);

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);
```

inner_product() 函數將值 acc 初始化為 init，然後對區間 [first1, last1] 的每個迭代器 i 和區間 [first2, first2 + (last1 - first1)] 的對應迭代器 j 循序執行 acc = *i * *j（第一個版本），或是 acc = binary_op(*i, *j)（第二個版本）。也就是從每一個序列的第一個元素計算值，從每一個序列的第二個元素計算值，其

餘依此類推，直到第一個序列結束 （此處第二個序列至少要和第一個序列一樣長）。這函數回傳 acc 的結果值。

partial_sum()

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

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result,
                           BinaryOperation binary_op);
```

partial_sum() 函數將 *first 設給 *result，*first + *(first + 1) 設給 *(result + 1)（第一個版本），或是將 binary_op(*first, *(first + 1)) 設給 *(result + 1)（第二個版本），其餘依此類推。也就是說以 result 起始的序列，其第 n 個元素包含以 first 起始之序列的前 n 個元素和（或是同義的 binary_op）。這函數回傳指向結果區間之 past-the-end 的迭代器。這演算法允許 result 等於 first，意思是若有需要結果可以複製到原始的序列上。

adjacent_difference()

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

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

adjacent_difference() 函數將 *first 設給位置 result (*result = *first)。在目的區間之後的位置都指定為原始區間之相鄰位置的差（或是 binary_op）。其意思是目的區間的下一個位置 (result + 1) 會設為 *(first + 1) - *first（第一個版本），或是 binary_op(*(first + 1), *first)（第二個版本），其餘依此類推。這函數回傳指向結果區間之 past-the-end 的迭代器。這演算法允許 result 等於 first，意思是若有需要結果可以複製到原始的序列上。

iota() (C++11)

```
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last,T value);
```

iota() 函數將值指定給*first，並將新值指定給下一元素，直到最後一元素為止。同時利用 ++value 將 value 值遞增。