

# Chapter 20

## 標準樣板函式庫(四) 迭代器與泛型演算函式

# 迭代調整器

## ■ 插入迭代調整器 (insert iterator adaptor)

可讓使用者透過插入迭代調整器的協助輕易地將資料插入容器內

## ■ 逆向迭代調整器 (reverse iterator adaptor)

使用者可透過逆向迭代調整器的協助將一般迭代器由前往後的方向改為逆向前進

❖ 使用迭代調整器須加入 `iterator` 標頭檔

iterator adaptor

# 插入迭代調整器

## ■ 三種插入迭代調整器：

插入迭代器類別名稱	用途	介面函式
<code>insert_iterator</code>	任意位置插入	<code>inserter</code>
<code>front_insert_iterator</code>	前端插入	<code>front_inserter</code>
<code>back_insert_iterator</code>	末尾插入	<code>back_inserter</code>

- ❖ `front_insert_iterator` 使用容器的 `push_front` 函式插入資料
- ❖ `back_insert_iterator` 使用容器的 `push_back` 函式插入資料

insert iterator adaptor

# insert\_iterator 類別 (一)

## ■ 產生類別物件

```
list<double> a;
```

```
vector<int> b;
```

- (1) 容器名稱
  - (2) 物件名稱
  - (3) 迭代器插入位置

# insert\_iterator 類別 (二)

## ■ 使用 \* , ++ 與 = 覆載運算子：

```
list<int> a, b;
```

```
insert_iterator< list<int> > itr1(a,a.begin()) ;
```

```
// 將數字 1 到 5 依次插入串列中 a = 1 2 3 4 5
for ( int i = 1 ; i < 6 ; ++i , ++itr1 ) *itr1 = i ;
```

```
insert_iterator< list<int> > itr2(b,b.begin()) ;
```

```
// 將數字 1 到 5 依次插入串列中 b = 5 4 3 2 1
for ( int i = 1 ; i < 6 ; ++i ) *itr2 = i ;
```

❖ 考慮效率問題，空間緊鄰式容器不適合被當成插入的容器

# insert\_iterator 類別 (三)

## ■ 使用介面函式簡化插入步驟

```
template <class T , class Iter>
insert_iterator<T> inserter( T& foo , Iter iter ) ;
```

❖ 插入迭代調整器的最佳使用時機是與泛型函式一起使用

### ➤ 插入陣列元素到串列前

```
template <class Itr1 , class Itr2>
void insert( Itr1 from , Itr1 to , Itr2 j ) {
    for ( Itr1 i = from ; i != to ; ++i, ++j ) *j = *i ;
}

int a[3] = { 4 , 1 , 5 } ;
list<int> b(2,7) ;           // b = 7 7

// b = 4 1 5 7 7
insert( a , a+3 , inserter(b,b.begin()) ) ;
```



# 前(後)端插入迭代調整器

## ■ 產生與使用類別物件：

```
// foo = 1 1 1
deque<int>    foo(3,1) ;
```

```
// 產生前端與後端插入迭代調整器物件
front_iterator< deque<int> > fitr(foo) ;
back_iterator< deque<int> > bitr(foo) ;
```

```
// 由前端插入資料 foo = 8 2 3 1 1 1
*fitr = 3 ; *fitr = 2 ; *fitr = 8 ;
```

```
// 由後端插入資料 foo = 8 2 3 1 1 1 4 7 5
*bitr = 4 ; *bitr = 7 ; *bitr = 5 ;
```

# 前(後)端插入泛型函式

## ■ 前端插入泛型函式：

```
template <class T>
front_insert_iterator<T>
    front_insert( T& foo ) ;
```

## ■ 後端插入泛型函式：

```
template <class T>
back_insert_iterator<T>
    back_insert( T& foo ) ;
```

- ❖ 前後端插入迭代調整器的最佳使用時機是與泛型函式一起使用

# 使用前(後)端插入泛型函式

## ■ 使用插入泛型函式：

```

template <class Itr1 , class Itr2>
void insert( Itr1 a , Itr1 b , Itr2 j ) {
    for ( Itr1 i = a ; i != b ; ++i , ++j ) *j = *i ;
}
...
list<int> foo ;
vector<int> bar(4,1) ;
int no[8] = { 2 , 1 , 3 , 8 , 6 , 4 , 9 , 1 } ;
// foo = 6 8 3
insert( no+2 , no+5 , front_inserter(foo) ) ;
// foo = 6 8 3 1 1 1 1
insert( bar.begin() , bar.end() , back_inserter(foo) ) ;

```

❖ 上式等同使用 `STL` 內的 `copy` 泛型演算函式

```
copy( bar.begin() , bar.end() , back_inserter(foo) ) ;
```

# 逆向迭代調整器

## ■ 逆向迭代調整器：

```
list<int> a ;
// a = 0 1 2 3 4 5 6 7 8 9
for ( int i = 0 ; i < 10 ; ++i ) a.push_back(i) ;
```

```
// 產生逆向迭代調整器物件
reverse_iterator< list<int>::iterator > ritr(a.end()) ;
```

```
// 逆向列印 a : 9 8 7 6 5 4 3 2 1 0
for ( ; ritr.base() != a.begin() ; ++ritr )
    cout << *ritr << " " ;
```

- ❖ `ritr.base()` 回傳 `ritr` 逆向迭代器指向的位址
- ❖ `*ritr` 回傳 `ritr.base()` 指向位址的下一筆資料

reverse iterator adaptor

# 資料輸入/輸出串流迭代器

## ■ 四種高低階資料輸入/輸出串流迭代器：

類型	輸入/輸出串流迭代器類別	名稱
高階	<b>istream_iterator</b>	輸入串流迭代器 <b>input stream iterator</b>
	<b>ostream_iterator</b>	輸出串流迭代器 <b>output stream iterator</b>
低階	<b>istreambuf_iterator</b>	輸入緩衝區迭代器 <b>input streambuf iterator</b>
	<b>ostreambuf_iterator</b>	輸出緩衝區迭代器 <b>output streambuf iterator</b>

❖ 使用以上的迭代調整器須加入 **iterator** 標頭檔

# 格式化輸入/輸出串流迭代器

## ■ 格式化輸入/輸出串流迭代器

直接將迭代器指向資料串流物件，使得資料串流物件可透過其協助間接地讀取/寫入格式化的資料，如字元，字串，整數，浮點數或自定的資料型別

## ■ 格式化輸入/輸出串流迭代器類別

- 輸入串流迭代器類別：**istream\_iterator**
- 輸出串流迭代器類別：**ostream\_iterator**

# 輸入串流迭代器 (一)

## ■ `istream_iterator` 類別物件

```
// 針對 cin 產生一輸入串流迭代器處理整數的輸入  
istream_iterator<int> fiter1(cin) ;  
  
// 針對檔案產生一輸入串流迭代器處理浮點數的輸入  
ifstream infile("data") ;  
istream_iterator<double> fiter2(infile) ;
```

## ■ 使用輸入串流迭代器類別物件

使用覆載的 `->`, `*`, `++`, `==`, `!=` 等運算子控制  
迭代器的讀入動作

# 輸入串流迭代器 (二)

## ■ 計算遇到終止字元前的整數和

```
istream_iterator<int>    itr(cin) ;
int    sum = 0 ;
for ( ; itr != istream_iterator<int>() ; ++itr )
    sum += *itr ;
cout << "整數和 : " << sum << endl ;
```

- ❖ `istream_iterator<T>()` 代表所讀入的 `T` 型別資料的終止
- ❖ 終止字元在 `UNIX` 為 `ctrl-d`，在 `windows` 為 `ctrl-z`

## ■ 讀取複變數型別

```
istream_iterator< complex<double> >    itr(cin) ;
```



# 輸出串流迭代器 (一)

## ■ `ostream_iterator` 類別物件

```
// 針對 cout 產生一輸出串流迭代器處理整數的輸入  
ostream_iterator<int> iter1(cout) ;
```

```
// 輸出字元，並以 "\t" 作為間隔字串  
ostream_iterator<char> iter2(cout, "\t") ;
```

```
// 輸出浮點數到檔案 data 並以兩個空白分開  
ofstream ofile("data") ;  
ostream_iterator<double> out(ofile, " ") ;
```

# 輸出串流迭代器 (二)

## ■ 使用類別物件

使用覆載的 `->`, `*`, `++`, `==`, `!=` 等運算子控制  
迭代器的輸出動作

➤ 列印 `a**b**c**...y**z**`

```
ostream_iterator<char>      out(cout,"**") ;
for ( char ch = 'a' ; ch <= 'z' ; ++ch , ++out )
    *out = ch ;
```

❖ 以上輸出方式並沒有比直接使用 `cout` 方便，事實上與泛型  
函式一起合用才是使用輸入/輸出串流迭代器的最佳場合

# 輸出串流迭代器（三）

## ■ 與泛型函式合用

### ➤ 逆向列印檔案資料

```
list<int>    foo ;
ifstream      infile("data.db") ;
istream_iterator<int>  in(infile) ;
```

```
// 將檔案資料插入串列的前端
copy( in, istream_iterator<int>(),
      front_inserter(foo) ) ;
```

```
// 列印串列
copy( foo.begin() , foo.end() ,
      ostream_iterator<int>(cout," " ) ) ;
```

# 低階輸入/輸出緩衝區迭代器

- 輸入緩衝區迭代器  
`istreambuf_iterator`
- 輸出緩衝區迭代器  
`ostreambuf_iterator`
- 低階輸入/輸出緩衝區迭代器直接存取字元緩衝區資料，無型別轉換，故輸入/輸出效率快
- 低階輸入/輸出緩衝區迭代器的基本使用方式與格式化輸入/輸出串流迭代器類似，以下直接以例題說明使用方法

# 輸入緩衝區迭代器

## ■ 小寫字母轉大寫

```

struct to_upper {
    char operator() ( char c ) const {
        return ( 'a' <= c && c <= 'z' ? c-'a'+'A' : c );
    }

template <class S, class T, class Fn>
void convert( S a , S b , T c , Fn fn ) {
    for ( S i = a ; i != b ; ++i , ++c ) *c = fn(*i) ;
}
...
istreambuf_iterator<char> in(cin) ; // 低階
ostream_iterator<char> out(cout) ; // 高階

```

// 將輸入的小寫字母轉為大寫字母後輸出

```

convert( in , istreambuf_iterator<char>() , out ,
    to_upper() ) ;

```

# 輸出緩衝區迭代器

## ■ 檔案輸入/輸出：小寫字母轉大寫

```

struct to_upper { ... };

template <class S, class T, class Fn>
void convert( S a , S b , T c , Fn fn ) { ... }

ifstream infile("episodes.db") ; // 輸入檔
ofstream outfile("EPISODES.db") ; // 輸出檔

istreambuf_iterator<char> in(infile) ; // 低階
ostreambuf_iterator<char> out(outfile) ; // 低階

// 將輸入的小寫字母轉為大寫字母後輸出
convert( in , istreambuf_iterator<char>() , out ,
to_upper() ) ;

```

# 輸入/輸出緩衝區迭代器

- 可直接利用此類型迭代器存取在輸入/輸出緩衝區的數值資料，避免資料轉型所須耗費的時間，大幅增加輸入/輸出效率

```

const int S = 3 ;
int no[S] = { 12 , 3 , -4 } ;

ofstream outfile("number") ;
ostreambuf_iterator<char> out(outfile) ; // 低階

char *p = reinterpret_cast<char*>(no) ;

// 以字元方式直接將整數陣列輸出至檔案 number
copy( p , p+S*sizeof(int) ,
      ostreambuf_iterator<char>(out) ) ;

outfile.close() ;

```



# advance 與 distance

- **advance** : 移動迭代器指定數量
- **distance** : 計算兩迭代器的距離

```
string Q = "Must you be so linear , Jean-Luc" ;  
  
list<char> startrek(Q.begin(),Q.end()) ;  
list<char>::iterator itr1 , itr2 ;  
  
itr1 = itr2 = startrek.begin() ;  
advance(itr1,5) ;  
advance(itr2,12) ;  
cout << *itr1 << " " << *itr2 << endl ; // y s  
cout << distance(itr1,itr2) << endl ; // 7
```

# 泛型演算函式

■ 多數的泛型演算函式被定義在 `algorithm` 標頭檔

簡寫名稱	英文名稱	中文名稱
<code>BiIter</code>	<code>bidirectional iterator</code>	雙向迭代器
<code>ForIter</code>	<code>forward iterator</code>	向前迭代器
<code>InIter</code>	<code>input iterator</code>	輸入迭代器
<code>OutIter</code>	<code>output iterator</code>	輸出迭代器
<code>RandIter</code>	<code>random access iterator</code>	隨意存取迭代器
<code>UnPred</code>	<code>unary predictor</code>	單元判斷式
<code>BinPred</code>	<code>binary predictor</code>	雙元判斷式
<code>Comp</code>	<code>comparison function</code>	比較函式
<code>Fn</code>	<code>unary function</code>	單元函式
<code>BinFn</code>	<code>binary function</code>	雙元函式

# 複製，填滿，對調

函式名稱	作用
<code>copy</code>	將某序列元素複製到另一序列內
<code>copy_backward</code>	同上，但指定複製的終止位址
<code>fill</code>	在輸入序列範圍內填滿某資料
<code>fill_n</code>	同上，但填滿指定數目的資料
<code>iter_swap</code>	對調兩迭代器所指位址的資料
<code>swap</code>	對調兩元素
<code>swap_ranges</code>	對調兩等長序列內的所有元素

# COPY

## ■ 複製函式

```
template <class InIter , class OutIter>
OutIter copy( InIter a , InIter b ,
               OutIter c ) ;
```

## ■ 功用

將  $[a,b)$  範圍內的元素一一複製到另一個由  $c$  起始的序列。複製完畢後，函式回傳第二個序列最後一個複製元素的後一個位址



# copy\_backward

## ■ 末端複製函式

```
template <class BiIter1 , class BiIter2>
BiIter2 copy_backward( BiIter1 a , BiIter1 b ,
                      BiIter2 d ) ;
```

## ■ 功用

將  $[a,b)$  範圍內的元素一一複製到以  $d$  指到的前一個元素為止。複製完畢後，函式回傳第二個序列最後一個複製元素的後一個位址



- ❖ 此函式並非逆向複製元素，而是設定複製的末端元素位置

# fill 與 fill\_n

## ■ 填滿

```
template <class ForIter , class T>
void  fill( ForIter a , ForIter b , const T& val ) ;

template <class ForIter , class S , class T>
void  fill_n( ForIter a , S n , const T& val ) ;
```

## ■ 功用

<b>fill</b>	填入 <code>val</code> 數值資料於 $[a,b)$ 內
<b>fill_n</b>	填入 <code>n</code> 筆 <code>val</code> 資料於由 <code>a</code> 起點的序列內



- ❖ `fill_n` 函式的參數 `n` 不可超過由 `a` 起始的序列長度

# iter\_swap 與 swap

## ■ 對調單一元素

```
template <class ForIter1 , class ForIter2>
void  iter_swap( ForIter1 a , ForIter2 c ) ;

template <class T>
void  swap( T& foo , T& bar ) ;
```

## ■ 功用

**iter\_swap** 對調兩迭代器所指向的資料值  
**swap** 對調兩同型別的資料值



# Swap\_ranges

## ■ 對調序列元素

```
template <class ForIter1 , class ForIter2>
ForIter2 swap_ranges( ForIter1 a , ForIter2 b ,
                      ForIter2 c ) ;
```

## ■ 功用

將  $[a,b)$  序列與另一個由迭代器  $c$  起始的等數量元素序列對調



# 搜尋序列容器元素

函式名稱	作用
<code>adjacent_find</code>	找尋相鄰且相等的元素
<code>equal</code>	檢查兩序列是否完全相等
<code>find</code>	在序列內找尋指定資料值的元素位址
<code>find_if</code>	在序列內找尋滿足判斷式的元素位址
<code>find_first_of</code>	在序列內找出首位出現在另一序列元素位址
<code>find_end</code>	在序列內找出末位出現在另一序列元素位址
<code>search</code>	在序列內找尋另一序列出現的起始位址
<code>search_n</code>	在序列內找尋指定數量的相鄰元素位址
<code>mismatch</code>	在兩序列內，以一對一方式找尋首位不同元素的個別位址

# adjacent\_find

## ■ 搜尋相鄰且相同元素位置

```
template <class ForIter>
ForIter adjacent_find( ForIter a , ForIter b ) ;

template <class ForIter, class BinPred>
ForIter adjacent_find( ForIter a , ForIter b ,
                      BinPred fn ) ;
```

## ■ 功用

在  $[a,b)$  序列內搜尋相鄰且相同的元素位置。  
如果找到則回傳第一個元素位置，否則回傳  $b$



- ❖ 函式的第二式使用雙元判斷式  $fn$  自行設定相等的涵義

# equal

## ■ 判斷兩序列是否相等

```
template <class InIter1 , class InIter2>
bool equal( InIter1 a , InIter1 b , InIter2 c ) ;

template <class InIter1, class InIter2, class BinPred>
bool equal( InIter1 a , InIter1 b , InIter2 c ,
            BinPred fn ) ;
```

## ■ 功用

判斷  $[a, b)$  序列與由  $c$  開始的序列的個別元素  
是否相等



❖ 函式的第二式使用雙元判斷式  $fn$  自行設定相等的涵義

# find 與 find\_if

## ■ 搜尋序列元素

```
template <class InIter , class T>
InIter find( InIter a , InIter b , const T& val ) ;

template <class InIter , class UnPred>
InIter find_if( InIter a , InIter b , UnPred fn ) ;
```

## ■ 功用

**find** 搜尋 $[a,b)$ 內元素值為  $val$  的位置  
**find\_if** 搜尋 $[a,b)$ 內滿足單元判斷式  $fn$  的元素位置

- ❖ 若序列內並無滿足搜尋條件的元素，則回傳迭代器  $b$



# find\_first\_of

## ■ 搜尋第一個出現在序列的元素

```
template <class ForIter1 , class ForIter2>
ForIter1 find_first_of( ForIter1 a , ForIter1 b ,
                        ForIter2 c , ForIter2 d ) ;

template <class ForIter1, class ForIter2, class BinPred>
ForIter1 find_first_of( ForIter1 a , ForIter1 b ,
                        ForIter2 c , ForIter2 d ,
                        BinPred fn ) ;
```

## ■ 功用

在 $[a,b)$ 序列內搜尋第一個出現在 $[c,d)$ 序列內的元素位置，若無法找到則回傳迭代器  $b$

- ❖ 函式的第二式使用雙元判斷式  $fn$  自行設定相等的涵義



# find\_end

## ■ 逆向搜尋第一個出現在序列的元素

```
template <class ForIter1 , class ForIter2>
ForIter1 find_end( ForIter1 a , ForIter1 b ,
                    ForIter2 c , ForIter2 d ) ;

template <class ForIter1, class ForIter2, class BinPred>
ForIter1 find_end( ForIter1 a , ForIter1 b ,
                    ForIter2 c , ForIter2 d ,
                    BinPred fn ) ;
```

## ■ 功用

在  $[a, b)$  序列內逆向搜尋第一個出現在  $[c, d)$  序列內的元素位置，若無法找到則回傳迭代器  $b$

- ❖ 函式的第二式使用雙元判斷式  $fn$  自行設定相等的涵義



# search

## ■ 搜尋子序列

```
template <class ForIter1 , class ForIter2>
ForIter1 search( ForIter1 a , ForIter1 b ,
                  ForIter2 c , ForIter2 d ) ;

template <class ForIter1, class ForIter2, class BinPred>
ForIter1 search( ForIter1 a , ForIter1 b ,
                  ForIter2 c , ForIter2 d ,
                  BinPred fn ) ;
```

## ■ 功用

在  $[a, b)$  序列內搜尋含有  $[c, d)$  子序列的位置，  
若無法找到則回傳迭代器  $b$

- ❖ 函式的第二式使用雙元判斷式  $fn$  自行設定相等的涵義



# search\_n

## ■ 搜尋 n 個相鄰且相等的元素

```
template <class ForIter , class Size , class T>
ForIter search_n( ForIter a , ForIter b ,
                  Size n , const T& foo ) ;

template <class ForIter , class Size , class T ,
          class BinPred>
ForIter search_n( ForIter a , ForIter b ,
                  Size n , const T& foo ,
                  BinPred fn ) ;
```

## ■ 功用

在  $[a,b)$  序列內搜尋 n 個相鄰的元素值為 foo 的位置，若無法找到則回傳迭代器 b

- ❖ 函式的第二式使用雙元判斷式 fn 自行設定相等的涵義



# mismatch

## ■ 搜尋兩序列第一個不相等的元素

```
template <class InIter1 , class InIter2>
pair<InIter1,InIter2>
    mismatch( InIter1 a , InIter1 b , InIter2 c ) ;

template <class InIter1 ,class InIter2 ,class BinPred>
pair<InIter1,InIter2>
    mismatch( InIter1 a , InIter1 b , InIter2 c ,
              BinPred fn ) ;
```

## ■ 功用

在  $[a,b)$  序列與由迭代器  $c$  開始的序列內，兩兩相比，直到找到第一個不相等的元素位置為止

- ❖ 函式的第二式使用雙元判斷式  $fn$  自行設定相等的涵義



# 取代與搬移元素

函式名稱	作用
<code>replace</code>	將指定元素以新元素取代
<code>replace_copy</code>	同上，取代後序列會複製到新序列，原序列不變
<code>replace_if</code>	將滿足判斷函式的元素以新元素取代
<code>replace_copy_if</code>	同上，取代後序列會複製到新序列，原序列不變
<code>remove</code>	將指定的元素搬到序列末端
<code>remove_copy</code>	同上，未搬移的元素會複製到新序列，原序列不變
<code>remove_if</code>	將滿足判斷函式的元素搬到序列末端
<code>remove_copy_if</code>	同上，未搬移的元素會複製到新序列，原序列不變
<code>unique</code>	移除相鄰且重複元素
<code>unique_copy</code>	同上，結果會複製到新序列，原序列不變

# replace    replace\_copy

## ■ 取代元素

```
template <class ForIter , class T>
void  replace( ForIter a , ForIter b ,
                const T& old_val , const T& new_val ) ;

template <class ForIter , class OutIter , class T>
OutIter replace_copy( ForIter a, ForIter b, OutIter c,
                      const T& old_val ,
                      const T& new_val ) ;
```

## ■ 功用

在 $[a,b)$ 序列內搜尋 `old_val`，全部以 `new_val` 取代

❖ 函式的第二類型將取代後的序列複製到迭代器 `c` 指向的序列

# replace\_if replace\_copy\_if

## ■ 依條件取代元素

```
template <class ForIter , class UnPred , class T>
void replace_if( ForIter a , ForIter b ,
                  UnPred fn , const T& new_val ) ;

template <class ForIter , class OutIter ,
          class UnPred , class T>
OutIter replace_copy_if( ForIter a, ForIter b, OutIter c,
                         UnPred fn , const T& new_val ) ;
```

## ■ 功用

在 $[a,b)$ 序列內搜尋滿足單元判斷式  $fn$  的元素，之後將之全部以新元素  $new\_val$  取代。

❖ 第二式將取代後的序列複製到迭代器  $c$  指向的序列



# remove      remove\_copy

## ■ 刪除元素

```
template <class ForIter , class T>
ForIter remove( ForIter a , ForIter b ,
                  const T& val ) ;
```

```
template <class ForIter , class OutIter , class T>
OutIter remove_copy( ForIter a , ForIter b ,
                      OutIter c , const T& val ) ;
```

## ■ 功用

在 $[a,b)$ 序列內刪除  $\text{val}$  元素

❖ 函式的第二類型將刪除後的序列複製到迭代器  $c$  指向的序列

# **remove\_if      remove\_copy\_if**

## ■ 依條件刪除元素

```
template <class ForIter , class UnPred>
ForIter remove_if( ForIter a , ForIter b , UnPred fn );

template <class ForIter , class OutIter , class UnPred>
OutIter remove_copy_if( ForIter a , ForIter b ,
                        OutIter c , UnPred fn ) ;
```

## ■ 功用

在 $[a,b)$ 序列內刪除滿足單元判斷式  $fn$  的元素

❖ 第二類型將刪除後的序列複製到迭代器  $c$  指向的序列



# unique

## ■ 刪除相鄰且相同的重複元素

```
template <class ForIter>
ForIter unique( ForIter a , ForIter b ) ;

template <class ForIter , class BinPred>
ForIter unique( ForIter a , ForIter b , BinPred fn ) ;
```

## ■ 功用

在 $[a, b)$ 序列內刪除相鄰且相同的重複元素

❖ 函式的第二式使用單元判斷式  $fn$  自行設定相等的涵義

# unique\_copy

## ■ 刪除相鄰且相同的重複元素

```
template <class ForIter , class OutIter>
OutIter unique_copy( ForIter a , ForIter b ,
                     OutIter c ) ;

template <class ForIter, class OutIter, class BinPred>
OutIter unique_copy( ForIter a , ForIter b ,
                     OutIter c , BinPred fn ) ;
```

## ■ 功用

在 $[a,b)$ 序列內刪除相鄰且相同的重複元素，刪除後的序列複製到以  $c$  指向的序列

- ❖ 函式的第二式使用單元判斷式  $fn$  自行設定相等的涵義



# 最大與最小元素

函式名稱	作用
<code>max</code>	回傳兩元素的最大值
<code>max_element</code>	回傳序列內的最大元素位址
<code>min</code>	回傳兩元素的最小值
<code>min_element</code>	回傳序列內的最小元素位址

# max 與 max\_element

## ■ 回傳兩數或序列的最大元素

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

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

template <class ForIter>
ForIter max_element( ForIter a , ForIter b ) ;

template <class ForIter , class Comp>
ForIter max_element( ForIter a, ForIter b, Comp fn ) ;
```

- ❖ 兩函式的第二類型利用雙元比較判斷式 `fn` 決定兩數的大小

# min 與 min\_element

## ■ 回傳兩數或序列的最小元素

```

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

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

template <class ForIter>
ForIter min_element( ForIter a , ForIter b ) ;

template <class ForIter , class Comp>
ForIter min_element( ForIter a, ForIter b, Comp fn ) ;

```

- ❖ 兩函式的第二類型利用雙元比較判斷式 `fn` 決定兩數的大小



# 資料的重新排列

函式名稱	作用
<code>rotate</code>	旋轉序列
<code>rotate_copy</code>	同上，旋轉後的結果複製到新序列，原序列不變
<code>reverse</code>	逆轉序列
<code>reverse_copy</code>	同上，逆轉後的結果複製到新序列，原序列不變
<code>random_shuffle</code>	隨機重新設定序列元素位置
<code>partition</code>	將滿足判斷式的元素搬到序列前端
<code>stable_partition</code>	同上，但滿足判斷式的元素仍保持原有的前後次序

# rotate 與 rotate\_copy

## ■ 旋轉序列元素

```
template <class ForIter>
void rotate( ForIter a , ForIter m , ForIter b ) ;

template <class ForIter , class OutIter>
OutIter rotate_copy( ForIter a , ForIter m ,
                     ForIter b , OutIter c ) ;
```

## ■ 功用

旋轉[**a,b**)序列元素直到迭代器 **m** 指向的  
元素為首位元素

- ❖ 函式的第二類型將旋轉後的序列複製到迭代器 **c**  
指向的序列



# reverse 與 reverse\_copy

## ■ 逆向排列序列元素

```
template <class BiIter>
void reverse( BiIter a , BiIter b ) ;

template <class BiIter , class OutIter>
OutIter reverse_copy( BiIter a , BiIter b ,
                      OutIter c ) ;
```

## ■ 功用

逆向排列[a,b)序列元素

- ❖ 函式的第二類型將逆向排列後的序列複製到迭代器  
c 指向的序列



# random\_shuffle

## ■ 隨機排列序列

```
template <class RandIter>
void random_shuffle( RandIter a , RandIter b ) ;

template <class RandIter , class Fn>
void random_shuffle( RandIter a , RandIter b ,
                     Fn fn ) ;
```

## ■ 功用

對  $[a, b)$  序列元素重新隨機排列

❖ 函式的第二式可自行設定隨機函式  $fn$



# partition stable\_partition

## ■ 切割篩選序列元素

```
template <class BiIter , class UnPred>
BiIter partition( BiIter a , BiIter b , UnPred fn ) ;

template <class BiIter , class UnPred>
BiIter stable_partition( BiIter a , BiIter b ,
                         UnPred fn ) ;
```

## ■ 功用

利用 **fn** 判斷式篩選滿足的元素並將之移到序列前端，篩選後函式回傳第一個不滿足的元素位址

- ❖ **stable\_partition** 在篩選後仍會保持元素在篩選前的前後次序



# 排列

函式名稱	作用
<code>next_permutation</code>	找出序列的下一個排列方式
<code>prev_permutation</code>	找出序列的上一個排列方式

# next\_permutation

## ■ 找尋序列下一個排列方式

```
template <class RandIter>
bool next_permutation( RandIter a , RandIter b ) ;

template <class RandIter , Comp fn>
bool next_permutation( RandIter a , RandIter b ,
                      Comp fn ) ;
```

## ■ 功用

找出序列的下一個排列方式，如果找到回傳真值，否則回傳假。

❖ 函式的第二式可自行設定元素比較函式 `fn`

程式

輸出

# prev\_permutation

## ■ 找尋序列上一個排列方式

```
template <class RandIter>
bool prev_permutation( RandIter a , RandIter b ) ;  
  
template <class RandIter , Comp fn>
bool prev_permutation( RandIter a , RandIter b ,
                      Comp fn ) ;
```

## ■ 功用

找出序列的前一個排列方式，如果找到回傳真值，否則回傳假。

❖ 函式的第二式可自行設定元素比較函式 `fn`

# 排序相關函式

函式名稱	作用
<code>sort</code>	由小到大排序
<code>stable_sort</code>	同上，但相當的元素其順序保持不變
<code>partial_sort</code>	對整個序列排序，然而僅有在指定位 置之前的元素是由小到大排列，之後 的元素則無特定大小順序
<code>partial_sort_copy</code>	同上，但排序後的資料被複製到新序 列，原序列保持不變
<code>nth_element</code>	將元素擺放到適當位置，使得在此位 置之前的資料皆小於此元素

# sort

## ■ 排序

```
template <class RandIter>
void sort( RandIter a , RandIter b ) ;

template <class RandIter , Comp fn>
void sort( RandIter a , RandIter b , Comp fn ) ;
```

## ■ 功用

對 [a,b) 序列由小到大排序

- ❖ 函式的第二式可自行設定元素比較函式 fn

# stable\_sort

## ■ 穩定排序

```
template <class RandIter>
void stable_sort( RandIter a , RandIter b ) ;

template <class RandIter , Comp fn>
void stable_sort( RandIter a , RandIter b ,
                  Comp fn ) ;
```

## ■ 功用

對  $[a,b)$  序列由小到大排序，但此排序法對資料相當的元素在排序前後仍會保持原先的次序

- ❖ 函式的第二式可自行設定元素比較函式 `fn`



# partial\_sort

## ■ 部份排序

```
template <class RandIter>
void partial_sort( RandIter a , RandIter m ,
RandIter b ) ;

template <class RandIter , Comp fn>
void partial_sort( RandIter a , RandIter m ,
RandIter b , Comp fn ) ;
```

## ■ 功用

對  $[a,b)$  序列排序，但只有  $[a,m)$  範圍內的元素是由小到大排列。 $m$  之後的元素都會比之前元素大，但卻不會依大小排列

- ❖ 函式的第二式可自行設定元素比較函式 `fn`



# partial\_sort\_copy

## ■ 複製部份排序

```
template <class InIter , class RandIter>
RandIter partial_sort_copy( InIter a , InIter b ,
                           RandIter c , RandIter d ) ;

template <class InIter , class RandIter , class Comp>
RandIter partial_sort_copy( InIter a , InIter b ,
                           RandIter c , RandIter d ,
                           Comp fn ) ;
```

## ■ 功用

對[a,b)序列排序，但僅複製對應的數量到  
[c,d)序列儲存，此外原序列保持不變

- ❖ 函式的第二式可自行設定元素比較函式 fn



程式



輸出

# nth\_element

## ■ 指定位置方式排序

```
template <class RandIter>
void nth_element( RandIter a , RandIter nth ,
                  RandIter b ) ;

template <class RandIter , class Comp>
void nth_element( RandIter a , RandIter nth ,
                  RandIter b , Comp fn ) ;
```

## ■ 功用

對  $[a,b)$  序列排序，使得  $nth$  指向元素之前  
的元素都小於此元素

- ❖ 函式的第二式可自行設定元素比較函式  $fn$

程式

輸出

# 已排序序列的合併

函式名稱	作用
<code>merge</code>	合併兩段經過排序的序列到新序列
<code>inplace_merge</code>	將序列前後兩段已排序的元素依資料大小重新組合

# merge

## ■ 合併兩段經過排序的序列

```
template <class InIter1 , class InIter2 ,
          class OutIter>
OutIter merge( InIter1 a , InIter1 b ,
                InIter2 c , InIter2 d , OutIter e ) ;

template <class InIter1 , class InIter2 ,
          class OutIter , class Comp>
OutIter merge( InIter1 a , InIter1 b ,
                InIter2 c , InIter2 d ,
                OutIter e , Comp fn ) ;
```

## ■ 功用

合併兩段經過排序的序列到新序列

❖ 函式的第二式可自行設定元素比較函式 `fn`

程式

輸出

# inplace\_merge

## ■ 重新組合兩段在同一序列內並經過排序的子序列

```
template <class BiIter>
void inplace_merge( BiIter a , BiIter m ,
                    BiIter b ) ;

template <class BiIter , class Comp>
void inplace_merge( BiIter a , BiIter m ,
                    BiIter b , Comp fn ) ;
```

## ■ 功用

合併兩段已經排序完成的子序列

❖ 函式的第二式可自行設定元素比較函式 `fn`

程式

輸出

# 已排序完全序列的搜尋

函式名稱	作用
<code>binary_search</code>	使用二分搜尋方式找尋元素
<code>lower_bound</code>	找出序列內不小於輸入資料的第一個元素位址
<code>upper_bound</code>	找出序列內不小於輸入資料的末筆元素後一個位址
<code>equal_range</code>	回傳以上兩個前後位址

# binary\_search

## ■ 二分搜尋法找尋元素

```
template <class ForIter , class T>
bool binary_search( ForIter a , ForIter b ,
                     const T& val ) ;

template <class BiIter , class T , class Comp>
bool binary_search( ForIter a , ForIter b ,
                     const T& val , Comp fn ) ;
```

## ■ 功用

對一已排序完成的  $[a, b)$  序列，利用二分搜尋法找尋元素值  $val$  是否在序列內

- ❖ 函式的第二式可自行設定元素比較函式  $fn$

程式

輸出

# lower\_bound

## ■ 找尋元素插入序列的下界位置

```
template <class ForIter , class T>
ForIter lower_bound( ForIter a , ForIter b ,
                      const T& val ) ;

template <class ForIter , class T , class Comp>
ForIter lower_bound( ForIter a , ForIter b ,
                      const T& val , Comp fn ) ;
```

## ■ 功用

在已排序完成的[a,b)序列內找尋元素值 val 插入的下界位置，使得插入後的序列仍可保持相同的大小順序

- ❖ 函式的第二式可自行設定元素比較函式 fn

# upper\_bound

## ■ 找尋元素插入序列的上界位置

```
template <class ForIter , class T>
ForIter upper_bound( ForIter a , ForIter b ,
                      const T& val ) ;

template <class ForIter , class T , class Comp>
ForIter upper_bound( ForIter a , ForIter b ,
                      const T& val , Comp fn ) ;
```

## ■ 功用

在已排序完成的 $[a,b)$ 序列內找尋元素值  $val$  插入的上界位置，使得插入後的序列仍可保持相同的大小順序

- ❖ 函式的第二式可自行設定元素比較函式  $fn$

# equal\_range

## ■ 同時找尋元素插入序列的上界與下界位置

```
template <class ForIter , class T>
pair<ForIter,ForIter>
equal_range( ForIter a , ForIter b , const T& val ) ;

template <class ForIter , class T , class Comp>
pair<ForIter,ForIter>
equal_range( ForIter a , ForIter b , const T& val ,
    Comp fn ) ;
```

## ■ 功用

在已排序完成的 $[a,b)$ 序列內同時找尋 $val$ 插入的上下界位置，使得新序列仍可保持相同的大小順序

❖ 函式的第二式可自行設定元素比較函式  $fn$



# 集合運算函式

函式名稱	作用
<code>includes</code>	檢查甲集合是否包含乙集合
<code>set_difference</code>	找出元素是在甲集合內但不在乙集合內
<code>set_intersection</code>	找出兩集合的交集
<code>set_symmetric_difference</code>	找出不在兩集合交集的元素
<code>set_union</code>	找出兩集合的聯集

- ❖ 所有集合序列都須經過排列

# includes

## ■ 判斷一集合序列是否包含另一集合序列

```
template <class InIter1 , class InIter2>
bool includes( InIter1 a , InIter1 b ,
                InIter2 c , InIter2 d ) ;

template <class InIter1 , class InIter2 , class Comp>
bool includes( InIter1 a , InIter1 b ,
                InIter2 c , InIter2 d , Comp fn ) ;
```

## ■ 功用

判斷[a,b)序列是否包含依次[c,d)序列的所有元素，此兩序列都須已排序完成

- ❖ 此兩序列都須為排序完成的序列
- ❖ 函式的第二式可自行設定元素比較函式 `fn`

# 四種集合演算函式 (一)

## ■ `set_difference`

找出元素是在甲集合內但不在乙集合內

## ■ `set_intersection`

找出兩集合的交集

## ■ `set_symmetric_difference`

找出不在兩集合交集內的元素

## ■ `set_union`

找出兩集合的聯集

# 四種集合演算函式 (二)

## ■ 函式基本型式

```
template <class InIter1 , class InIter2 ,
          class OutIter>
OutIter 集合函式名稱( InIter1 a , InIter1 b ,
                         InIter2 c , InIter2 d ,
                         OutIter e ) ;
```

  

```
template <class InIter1 , class InIter2 ,
          class OutIter , class Comp>
OutIter 集合函式名稱( InIter1 a , InIter1 b ,
                         InIter2 c , InIter2 d ,
                         OutIter e , Comp      fn ) ;
```

- ❖ [a,b),[c,d)序列都須為排序完成的序列
- ❖ 函式輸出到 e 指向的序列
- ❖ 函式的第二式可自行設定元素比較函式 fn



# 其他類型函式

函式名稱	作用
<code>count</code>	計算資料在序列內出現的個數
<code>count_if</code>	功能同上，但可自行設定相等的定義
<code>generate</code>	利用函式或函式物件設定序列元素值
<code>generate_n</code>	功能同上，但設定序列的 $n$ 個元素值
<code>for_each</code>	將序列內的每個元素輸入函式作處理
<code>transform</code>	功能同上，但結果送到另一序列

# count

# count\_if

## ■ 計算資料出現次數

```
template <class InIter , class T>
size_t count( InIter a , InIter b , const T& val ) ;

template <class InIter , class UnPred>
size_t count_if( InIter a , InIter b , UnPred fn ) ;
```

## ■ 功用

**count** 求 **val** 在  $[a,b)$  序列內出現的次數  
**count\_if** 求  $[a,b)$  序列中滿足 **fn** 的元素個數



# generate    generate\_n

## ■ 設定序列元素值

```
template <class ForIter , class Fn>
void generate( ForIter a , ForIter b , Fn fn ) ;

template <class OutIter , class Size , class Fn>
void generate_n( OutIter a , Size n , Fn fn ) ;
```

## ■ 功用

<b>generate</b>	利用 <b>fn</b> 設定[a,b)序列元素值
<b>generate_n</b>	利用 <b>fn</b> 設定序列以 <b>a</b> 起始的前 <b>n</b> 個元素值

❖ 本函式或函式物件 **fn** 不能使用任何參數



# for\_each

## ■ 轉換序列元素

```
template <class InIter , class Fn>
Fn  for_each( InIter a , InIter b , Fn fn ) ;
```

## ■ 功用

將 [a,b) 序列內的元素一一送入 fn 函式內處理

❖ 本函式設計簡單但應用卻相當廣泛



# transform (一)

## ■ 轉換序列元素至新序列

```
template <class InIter , class OutIter , class Fn>
OutIter transform( InIter a , InIter b ,
                    OutIter e , Fn fn ) ;

template <class InIter1 , class InIter2 ,
          class OutIter , class Fn>
OutIter transform( InIter1 a , InIter1 b ,
                    InIter2 c , OutIter e , Fn fn ) ;
```

## ■ 功用

將  $[a, b)$  元素送入單元函式  $fn$  處理後存入  $e$   
指向的序列

- ❖ 函式的第二式使用雙元函式一次處理兩個等長的序列



# transform (二)

- 可與輸出 / 輸入串流迭代器合用，簡化程式

```
ifstream  infile1("data1") , infile2("data2") ;  
  
// 由 data1 與 data2 兩整數檔分別讀入整數，計算其乘積後輸出  
copy( istream_iterator<int>(infile1) ,  
       istream_iterator<int>() ,  
       istream_iterator<int>(infile2) ,  
       ostream_iterator<int>(cout," ") ,  
       multiplies<int>() ) ;
```



# 數值演算函式

函式名稱	作用
<code>accumulate</code>	計算序列元素間的總和
<code>adjacent_difference</code>	計算兩兩相鄰元素的差距
<code>inner_product</code>	計算兩序列的內積
<code>partial_sum</code>	計算序列各項的部份和

❖ 此四個函式須使用 `numeric` 標頭檔

# accumulate

## ■ 計算序列元素總和

```
template <class InIter , class T>
T accumulate( InIter a , InIter b , T no ) ;

template <class InIter , class T , class BinFn>
T accumulate( InIter a , InIter b , T no ,
              BinFn fn ) ;
```

## ■ 功用

計算  $[a, b)$  序列元素總和， $no$  為傳入函式計算前的初值

- ❖ 函式的第二式可使用一雙元函式來定義加法的內容



# adjacent\_difference

## ■ 計算相鄰元素之差

```
template <class InIter , class OutIter>
OutIter adjacent_difference( InIter a , InIter b ,
                             OutIter c ) ;

template <class InIter , class OutIter , class BinFn>
OutIter adjacent_difference( InIter a , InIter b ,
                             OutIter c , BinFn fn ) ;
```

## ■ 功用

計算  $[a, b)$  序列兩兩相鄰元素之差，並將結果存入  $c$  所指向的序列內

❖ 函式的第二式可自行定義差距函式的內容



# inner\_product

## ■ 計算兩序列之內積

```
template <class InIter1 , class InIter2 , class T>
T inner_product( InIter1 a , InIter1 b , InIter2 c ,
                  T no ) ;

template <class InIter1 , class InIter2 , class T ,
          class BinFn1 , class BinFn2>
T inner_product( InIter1 a , InIter1 b , InIter2 c ,
                  T no , BinFn1 fn1 , BinFn2 fn2 ) ;
```

## ■ 功用

計算兩序列的內積和，no 為輸入的計算初值

- ❖ 函式的第二式可自行定義內積函式的內容，fn1 為加法函式，fn2 為乘法函式



# partial\_sum

## ■ 計算序列之部份和

```
template <class InIter , class OutIter>
OutIter partial_sum( InIter a , InIter b ,
                     OutIter c ) ;

template <class InIter , class OutIter , class BinFn>
OutIter partial_sum( InIter a , InIter b ,
                     OutIter c , BinFn fn ) ;
```

## ■ 功用

計算序列之所有前  $n$  項之部份和，結果存入  $c$   
所指向的序列

❖ 函式的第二式可自行定義累加函式的內容

