Chapter 19

標準樣板函式庫(三)
關聯容器

關聯容器(一)：簡介

- 關聯容器

  容器內的元素數值與其在容器內所儲存的位置有所相關

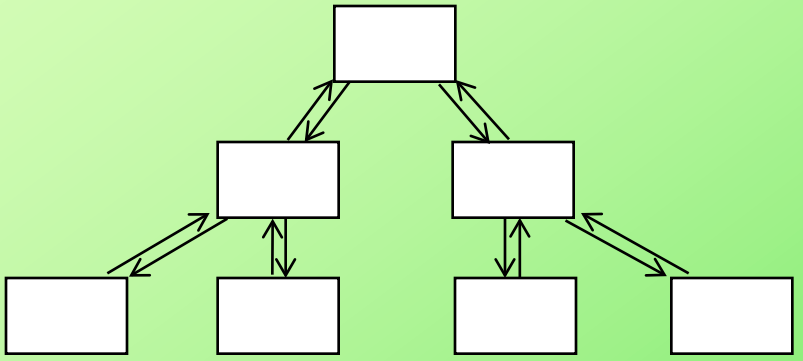
  - 所有的元素在存入關聯容器時都要經過一連串的比較動作，找出適當的位置才存入容器內，因此儲存元素的效率比序列容器差

  - 由於元素資料的大小與儲存位置有關，如果要找尋某元素是否在容器內，則只要搜尋元素所在的相關範圍，即可確認容器是否有此元素，因此搜尋元素的效率高

associative container

關聯容器(二)：樹狀結構

- 元素儲存在二元樹狀結構的節點內



- 節點式容器：使用雙向迭代器
- 分散式儲存：記憶空間的利用較有效率

關聯容器(三)：索引

- 關聯容器根據樹狀結構內每個節點內的索引值(key)來決定資料擺放的位置
  - 集合(set)，複集合(multiset)
    每個節點包含一筆資料，此筆資料即為索引
  - 映射(map)，複映射(multimap)
    每個節點包含兩筆資料，即(索引，映射值)

節點內的索引用來決定資料在樹狀結構的位置，因此不得直接更改索引資料，以免造成索引在節點間的次序錯亂。若使用者必須更改索引值，則須先去除索引所在的節點，然後再加入含有新索引的資料

❖ 複集合或複映射表示樹狀結構內的不同節點可以擁有相當的索引值

關聯容器(四)：雙向迭代器

- 關聯容器僅能使用雙向迭代器

關聯容器	迭代器	迭代器型態
`set<T>`	`set<T>::iterator` `set<T>::const_iterator`	常數型雙向迭代器 同上
`multiset<T>`	`multiset<T>::iterator` `multiset<T>::const_iterator`	常數型雙向迭代器 同上
`map<key,T>`	`map<key,T>::iterator` `map<key,T>::const_iterator`	雙向迭代器 常數型雙向迭代器
`multimap<key,T>`	`multimap<key,T>::iterator` `multimap<key,T>::const_iterator`	雙向迭代器 常數型雙向迭代器

關聯容器禁止使用者直接修改節點內的索引值，因此在容器中與索引有關的迭代器都是使用常數型雙向迭代器

關聯容器(五)：相當與相等

- 相等 (equal) ： 利用 operator==

  x 相等於 y ⟺ x == y

- 相當 (equivalent) ： 利用 operator<

  x 相當於 y ⟺ !(x<y) && !(y<x)

關聯容器利用兩資料是否「相當」來檢查樹狀結構節點內的索引是否重複。使用者在建構關聯容器物件時，可選擇提供小於運算子函式，即operator<，此運算子除可以用來比較兩索引的大小，同時也可用來檢驗索引是否重複

- 關聯容器內的搜尋成員函式都是利用小於運算子函式，但STL的搜尋演算函式則是利用相等運算子，兩者的搜尋方式以前者較有效率

關聯容器(六)：成對結構

- 成對結構：儲存成對資料

```
template <class S, class T>
struct  pair {
    S  first ;   // 第一筆資料
    T  second ;  // 第二筆資料

    pair() : first(S()) , second(T()) {}
    pair( const S& a , const T& b ) : first(a) , second(b) {}
} ;

template <class S, class T>
pair<S,T>  make_pair( const S& x , const T& y ) {
    return  pair<S,T>(x,y) ;
}

...

pair<string,double>   foo("PI",3.14) ;     // first 為 string 型別
foo.second = 3.141592654 ;                 // second 為 double 型別

pair<string,int>      bar ;                // first 為 string 型別
bar = make_pair("John",17) ;               // second 為 int 型別
```

集合與複集合(一)：物件

■ 物件產生方式

```
set<int>                    foo1 ;         // 利用預設由小到大的比較方式

set<int,less<int> >         foo2 ;         // 同上

set<int,greater<int> >      foo3 ;         // 利用由大到小的比較方式

set<int,greater<int> >      foo4(foo3) ; // 兩集合的索引比較方式要一樣


int a[] = { 3 , 1 , 2 , 5 , 2 , 6 } ;
set<int>                    foo5(a+1,a+5) ;    // foo5 = 1 2 5
set<int,less<int> >         foo6(a+1,a+5) ;    // 同上
set<int,greater<int> >      foo7(a+1,a+5) ;    // foo7 = 5 2 1
```

❖ 集合物件不會儲存重複的元素(即索引值)，但複集合物件則可以重複儲存

集合與複集合(二)：物件

- 集合元素可以為另一類別物件

```
struct  Student {
    string  name ;     // 姓名
    int     id ;       // 學號
    int     age ;      // 年齡
} ;

bool  operator< ( const Student& a , const Student& b ) {
    return  a.id < b.id ;
}

struct  by_age {
    bool  operator() ( const Student& a , const Student& b ) const {
        return  a.age > b.age ;
    }
} ;

set<Student>           foo ;   // 使用 operator< 以學號由小到大方式儲存
set<Student,by_age>    bar ;   // 使用 by_age 以年齡由大到小儲存
```

集合與複集合(三)：迭代器

- 仍有 **begin(), end(), rbegin(), rend()**

```
char                          a[4] = { 'b' , 'o' , 'r' , 'g' } ;
multiset<char>                b(a,a+4) ;

multiset<char>::iterator  i = b.begin() ;
for ( ; i != b.end() ; ++i ) cout << *i << ' ' ;   // 列印 : b g o r

multiset<char>::reverse_iterator  j = b.rbegin() ;
for ( ; j != b.rend() ; ++j ) cout << *j << ' ' ;  // 列印 : r o g b
```

- 所有的迭代器皆為常數型

```
char  a[4] = { 'b' , 'o' , 'r' , 'g' } ;
multiset<char>  foo(a,a+4) ;
multiset<char>::iterator   iter ;       // iter 仍為常數型迭代器
iter = foo.begin() ;
*iter = 'B' ;                           // 錯誤
```

❖ 可使用 **typedef** 簡化語法，如 **typedef multiset<int>::iterator Miter ;**

集合與複集合(四)：加入元素

成員函式	用途
`pair<iterator,bool>` `  insert( const T& foo )`	將資料 **foo** 加入集合(僅適用於集合容器)內。如果成功，則回傳 **pair<iter,true>**，**iter** 指向新元素位置，若失敗，則回傳 **pair<iter,false>**，**iter** 指向集合內已有的「相當」元素位置
`iterator` `  insert( const T& foo )`	將資料 **foo** 加入複集合(僅適用於複集合容器)，回傳指向新元素的迭代器

`iterator`
`  insert( iterator iter ,`
`  const T& foo )`
	將資料 **foo** 加入(複)集合內，且由 **iter** 起找尋適當的插入位置，函式回傳指向新元素的迭代器

`void  insert( iterator a ,` `  iterator b )`	將[a,b)之間的元素加入(複)集合內，對集合而言，重複的元素會被忽略

集合與複集合(五)：加入元素

- 加入元素

```
set<int>  foo ;

typedef  set<int>::iterator    SetIter ;   // 簡化語法
pair< SetIter , bool >   i ;
SetIter                  j ;

foo.insert(9) ;  foo.insert(5) ;           // 加入 9 5
i = foo.insert(3) ;                        // 加入 3
foo.insert(i.first,8) ;                    // 由 i.first 起加入 8

int       no[5] = { 9 , 2 , 5 , 7 , 3 } ;
foo.insert( no , no+5 ) ;                  // 加入 9 2 5 7 3

// 列印 : 2 3 5 7 8 9
for ( j = foo.begin() ; j != foo.end() ; ++j ) cout << *j << ' ' ;
```

集合與複集合(六)：去除元素

成員函式	用途
`void erase( iterator iter )`	將 `iter` 指向的元素去除
`size_type erase( const T& foo )`	去除資料值(或索引值)為 `foo` 的元素，回傳所去除的元素個數
`void erase( iterator a , iterator b )`	去除在 `[a,b)` 範圍之間的所有元素
`void clear()`	清除所有元素

```
string              a = "brokenback mountain" ;
multiset<char>      b(a.begin(),a.end()) ;

string   vowel = "aeiou" ;
for ( int i = 0 ; i < vowel.size() ; ++i )
   cout << vowel[i] << " : " << b.erase(vowel[i])
        << endl ;
```

```
a : 2
e : 1
i : 1
o : 2
u : 1
```

集合與複集合(七)：搜尋元素

成員函式	用途
`iterator find( const T& foo ) const`	找尋元素 **foo** 。若找到，回傳指向的迭代器，否則回傳 **end()**
`iterator lower_bound( const T& foo ) const`	找尋元素 **foo** 。若找到，回傳下限迭代器，否則回傳一迭代器指向第一個超出 **foo** 的元素
`iterator upper_bound( const T& foo ) const`	回傳一迭代器指向第一個超出 **foo** 的元素，即所謂的上限迭代器
`pair<iterator,iterator> equal_range( const T& foo ) const`	同時回傳下限，上限成對物件迭代器

對複集合而言，若所找到的「相當」元素不只一個，則須用一個範圍 [a,b)來界定元素所在區域， a 即為下限迭代器，b 為上限迭代器

- 所有的搜尋都是利用輸入的比較方式找尋「相當」的元素，同時以上所有的搜尋成員函式都適用於集合容器與複集合容器

集合與複集合(八)：搜尋元素

■ 集合搜尋

```
int  n , no[10] = { 3 , 2 , 7 , 9 , 9 , 7 , 5 , 2 , 7 , 3 } ;

typedef  set<int>::iterator  SetIter ;

set<int>   foo(no,no+10) ;

cin >> n ;
```

```
// 第一類型搜尋
SetIter   i = foo.find(n) ;
cout << n << ( i != foo.end() ? " 不" : " " ) << "在集合內" << endl ;
```

```
// 找尋包含 n 的範圍
pair<SetIter,SetIter>  iter = foo.equal_range(n) ;
cout << n << ( iter.first == iter.second ? " 不" : " " )
     << "在集合內" << endl ;
```

集合與複集合(九)：搜尋元素

■ 複集合搜尋

```
struct  myless {
    bool operator() ( int i , int j ) const { return i%10 < j%10 ; }
} ;
...
int  n , no[10] = {13,52,27,19,9,87,45,2,7,23} ;
typedef  multiset<int,myless>  SetType ;
typedef  SetType::iterator     SetIter ;

SetType  foo(no,no+10) ;
SetIter  i ;
pair<SetIter,SetIter>  m ;

cin >> n ;
cout << "-> " ;
m = foo.equal_range(n) ;
if ( iter.first != iter.second )
    for ( i = m.first ; i != m.second ; ++i ) cout << *i << ' ' ;
else
    cout << " 沒找到" ;
```

```
57
-> 27 87 7
```

集合與複集合(十)：其他函式

成員函式	用途
`size_type count( const T& foo ) const`	回傳元素 **foo** 在容器內出現的個數
`bool empty() const`	檢查(複)集合是否為空集合
`void swap( set<T>& foo )` `void swap( multiset<T>& foo )`	與另一個同型別的(複)集合容器對調所有元素
`key_compare key_comp() const`	回傳(複)集合內比較元素大小的函式(物件)

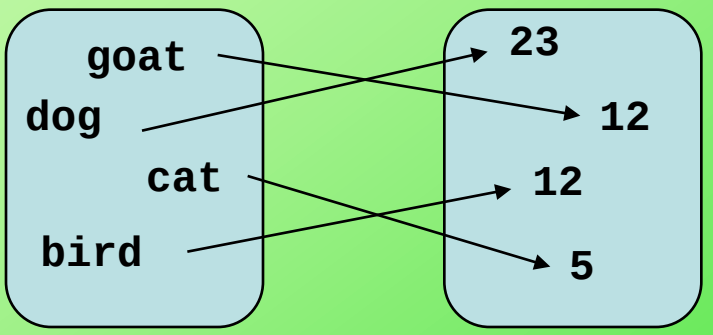
```
struct  myless {
    bool operator() ( int i , int j ) const { return i%10 < j%10 ; }
} ;
...
int  no[10] = {13,52,27,19,9,87,45,2,7,23} ;
multiset<int,myless>    foo(no,no+10) ;
cout << foo.key_comp()(24,33) << endl ;   // 印出 0
cout << foo.key_comp()(24,15) << endl ;   // 印出 1
```

映射與複映射(一)：基礎

- 索引與映射值

```
typedef   string      animal ;
typedef   int         quantity ;
map<animal,quantity> foo ;
```

索引　映射值



索引：動物　　映射值：數量

容器	中文名稱	差別
map	映射	索引不能重複
multimap	複映射	索引可重複

❖ (複)映射容器以索引大小決定節點儲存位置，使用者不能自由變更索引

映射與複映射(二)：物件

■ 物件產生方式

```
// 索引為字元，映射值為整數
map<char,int>                          a ;

// 同上，但以 ASCII 碼中次序較大的字元排在前端
map< char , int , greater<char> >  b ;

// 索引為字串，映射值為整數
multimap< string , int >   c ;

// 利用複製建構函式
multimap< string , int >   d(c) ;

// 儲存在範圍產生物件，索引相當的資料不重複儲存
map< string , int >        e( d.begin() , d.end() ) ;
```

❖ 儘量使用 **typedef** 簡化語法

```
typedef  multimap<string,double,greater<string> >  Str_Double ;
```

映射與複映射(三)：迭代器

- 迭代器為一成對物件，成對物件的第一個元素(first)為索引，第二個元素(second)為映射值

```
typedef  multimap<int,char>   i2cmmap ;
i2cmmap                       foo ;
ic2mmap::reverse_iterator     iter ;
...
// 依索引排列由後往前列印
for ( iter = foo.rbegin() ; iter != foo.rend() ; ++iter )
    cout << iter->first << " : " << iter->second << endl ;
```

- 索引為常數型別，不能透過迭代器更改索引值

```
iter = foo.rbegin() ;
iter->first = 25 ;              // 錯誤，不能直接修改索引值
iter->second = 'b' ;            // 正確
```

映射與複映射(四)：下標運算子

- 僅定義於映射容器，方便直接利用索引存取映射值

```
map<char,int>               no ;      // 儲存各個 DNA 出現數量
map<char,int>::iterator     iter ;

string  DNA = "AACGTCCGTTACGTT" ;

for ( int i = 0 ; i < DNA.size() ; ++i )
    no[DNA[i]] += 1 ;

// 依索引排列列印
for ( iter = no.begin() ; iter != no.end() ; ++iter )
    cout << iter->first << " : " << iter->second << endl ;
```

```
A : 3
C : 4
G : 3
T : 5
```

- 若下標運算子所使用的索引並不存在，則映射容器會自動插入以此索引建構而成的新節點，新節點的映射值則以其型別的預設建構函式產生。若映射型別為內定型別，其預設值就設為零

映射與複映射(五)：下標運算子

- 常數映射物件不能使用下標運算子

```
class  FOO {
  private :
    map<int,int>   bar ;
    ...
  public :

    // (1) 錯誤：若索引 a 不在 bar 內，則 a 會被加入 bar 內成新節點
    int  fn( int a ) const { return bar[a] ; } ;

    // (2) 錯誤：s 為常數物件，但 s 使用 bar 映射的下標運算子可能會更改
    //           bar 的資料，造成 s 不是常數
    friend  ostream& operator<< (
                    ostream& out , const FOO& s ) {
        return  out << s.bar[...] ;
    }

} ;
```

映射與複映射(六)：加入元素

成員函式	用途
`pair<iterator,bool>` `  insert( const T& foo )`	將資料 **foo** 加入映射物件(僅適用於映射容器)內。如果成功，則回傳 **pair<iter,true>**，**iter** 指向新元素位置，若失敗，則回傳 **pair<iter,false>**，**iter** 指向映射內已有的相當元素位置
`iterator` `  insert( const T& foo )`	將資料 **foo** 加入複映射(僅適用於複映射容器)，回傳指向新元素的迭代器

`iterator`
`  insert( iterator iter ,`
`  const T& foo )`
	將資料 **foo** 加入(複)映射內，且由 **iter** 起找尋適當的插入位置，函式回傳指向新元素的迭代器

`void  insert( iterator a ,` `  iterator b )`	將[a,b)之間的元素加入(複)映射內，對映射容器而言，重複的索引資料會被忽略

- 函式型式與集合容器相同，但這裡的 **T** 代表包含索引與映射值的成對物件

映射與複映射(七)：加入元素

- 加入元素

```
string  foo[4] = { "李白 夜思 怨情" , "王維 送別 相思" ,
                   "杜牧 清明"      , "杜甫 八陣圖" } ;

typedef  string  POEM ;
typedef  string  POET ;

multimap<POET,POEM>  a ;
string               poet , poem ;
istringstream        istring(poem) ;
for ( int i = 0 ; i < 4 ; ++i ) {
    istring.str( foo[i] ) ;
    istring >> poet ;
    while ( istring >> poem ) a.insert(make_pair(poet,poem)) ;
    istring.clear() ;
}
```

```
王維 : 送別
王維 : 相思
李白 : 夜思
李白 : 怨情
杜甫 : 八陣圖
杜牧 : 清明
```

```
multimap<POET,POEM>::iterator  iter ;
for ( iter = a.begin() ; iter != a.end() ; ++iter )
    cout << iter->first << " : " << iter->second << endl ;
```

映射與複映射(八)：去除元素

成員函式	用途
`void erase( iterator iter )`	將 **iter** 指向的節點去除
`size_type erase( const T& foo )`	去除索引值為 **foo** 的節點，回傳所去除的節點個數
`void erase( iterator a , iterator b )`	去除在 **[a,b)** 範圍之間的所有節點
`void clear()`	清除所有節點

```
string          a = "brokenback mountain" ;
map<char,int>   b ;
for ( int i = 0 ; i < a.size() ; ++i ) b[a[i]] += 1 ;
string          vowel = "aeiou" ;
for ( int i = 0 ; i < vowel.size() ; ++i ) {
    cout << vowel[i] << " : " << b[vowel[i]] << endl ;
    b.erase( vowel[i] ) ;     // 去除母音的節點
}
```

```
a : 2
e : 1
i : 1
o : 2
u : 1
```

映射與複映射(九)：搜尋索引

成員函式	用途
`iterator find( const T& foo ) const`	找尋索引 **foo** 的節點 。若找到，回傳指向的迭代器，否則回傳 **end()**
`iterator lower_bound( const T& foo ) const`	找尋索引 **foo** 。若找到，回傳下限迭代器，否則回傳一迭代器指向第一個超出 **foo** 的節點
`iterator upper_bound( const T& foo ) const`	回傳一迭代器指向第一個超出索引 **foo** 的節點，即所謂的上限迭代器
`pair<iterator,iterator> equal_range( const T& foo ) const`	同時回傳下限，上限成對物件迭代器

對複映射而言，若所找到的「相當」元素不只一個，則須用一個範圍 [a,b)來界定元素所在區域， a 即為下限迭代器，b 為上限迭代器

- 所有的搜尋都是利用輸入的比較方式找尋「相當」的元素，同時以上所有的搜尋成員函式都適用於映射容器與複映射容器

映射與複映射(十)：搜尋範例


```
string  composers = "Vivaldi Handel Bach Haydn Mozart Beethoven "
                    "Schubert Berlioz Weber Mendelssohn Chopin "
                    "Schumann Liszt Bramhs Bruch Tchaikovsky " ;

multimap<char,string>   collection ;
string                  name ;
istringstream           istr(composers) ;

while( istr >> name ) collection.insert( make_pair(name[0],name) ) ;

typedef  multimap<char,string>::iterator  mmiter ;
char                    letter ;
mmiter                  iter ;
pair<mmiter,mmiter>     piter ;

while ( 1 ) {
    cout << "> 輸入大寫字母 : "
    cin >> letter ;
    piter = collection.equal_range(letter) ;
    for ( iter = piter.first ; iter != piter.second ; ++iter )
        cout << iter->second << " " ;
    cout << endl ;
}
```

```
> 輸入大寫字母 : M
Mozart Mendelssohn
> 輸入大寫字母 : H
Handel Haydn
> 輸入大寫字母 : C
Chopin
...
```

映射與複映射(十一)：其他函式

成員函式	用途
`size_type count( const S& foo ) const`	回傳節點內的索引值 foo 在容器內出現的個數
`bool empty() const`	檢查(複)映射是否為空映射
`void swap( map<S,T,C>& foo )` `void swap( multimap<S,T,C>& foo )`	與另一個同型別的(複)映射容器對調所有元素
`key_compare key_comp() const`	回傳(複)映射內比較索引大小的函式(物件)

```
struct  myless {
    bool operator() ( int i , int j ) const { return i%10 < j%10 ; }
} ;
...
multimap<int,int,myless>    foo ;
cout << foo.key_comp()(24,33) << endl ;   // 印出 0
cout << foo.key_comp()(24,15) << endl ;   // 印出 1
```

範例：集合類別(一)

- 集合：將集合容器物件嵌入類別內使用

```
template <class T>
class  Set {
  private :
    set<T>  data ;
  public :
    void  insert( const T& a ) ;    // 加入元素
    int   erase( const T& a ) ;     // 去除元素   回傳去除的個數

    template <class S>                    // 交集
    friend Set<S>  operator* ( const Set<S>& a , const Set<S>& b ) ;

    template <class S>                    // 聯集
    friend Set<S>  operator+ ( const Set<S>& a , const Set<S>& b ) ;

    template <class S>                    // 差集
    friend Set<S>  operator- ( const Set<S>& a , const Set<S>& b ) ;
    ...
} ;
```

範例：集合類別(二)

- 集合類別運算子覆載

```
交集  :  operator*  (優先處理)
聯集  :  operator+
差集  :  operator-
```

則

A + B * C ⟺ A ∪ (B ∩ C)

A * B + C ⟺ (A ∩ B) ∪ C

程式

輸出

範例：HTML 課程表(一)

- 利用程式產生 HTML 課程表

HTML 標記

```
<b>粗體字</b>
<font color=red>紅字</font>
<br> : 換行
<hr> : 水平線
...
```

```
<font color=red><table>
<tr><th>春</th><th>夏</th><th>秋</th><th>冬</th>
<tr><th>20</th><th>32</th><th>23</th><th>12</th>
</table></font>
```

春	夏	秋	冬
20	32	23	12

範例：HTML 課程表(二)

- 輸入

```
國文     Mon : 3 4      Tue : 5
英文     Fri : 6 7 8
微積分   Tue : 1 2      Thu : 3 4  Fri : 1
物理     Wed : 1 2      Thu : 7 8
...
```

- 輸出

```
<html><head><title>課程表</title></head>
<body><table border=1 cellpadding=3>
<tr><th bgcolor=white width=5%><br></th>
<th width=16% bgcolor=cyan>星期一</th>
...
```

範例：HTML 課程表(三)

- 瀏覽器顯示

	星期一	星期二	星期三	星期四	星期五
1		微積分	物理		微積分
2		微積分	物理		
3	國文			微積分	
4	國文			微積分	
5		國文			
6	計概				英文
7	計概			物理	英文
8	計概			物理	英文

程式

輸出

範例：簡易資料庫

- 唐詩資料庫


```
typedef  string  Poet ;     // 詩人

// 唐詩作品結構
struct  Poem {
    Poet    poet ;          // 作者
    string  name ;          // 詩名
    string  content ;       // 內容
} ;

// 唐詩資料庫類別
class  Poem_db {
  private :

    multimap<Poet,Poem>   poem_db ;   // 根據詩人名字儲存資料

  public :
    // 由 datafile 資料檔讀入所有唐詩作品
    Poem_db( char* datafile ) ;

    // 搜尋某詩人在資料檔中的所有作品
    vector<Poem>  find( const Poet& poet ) const ;
} ;
```