Chapter 18

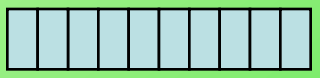
標準樣板函式庫(二)
序列容器

序列容器(一)：簡介

- 序列容器

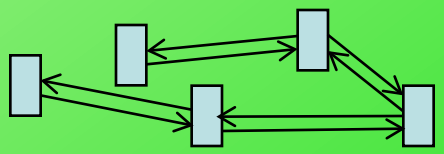
容器內的資料以如同線條的方式依次儲存，資料間的空間儲存型式可以是緊鄰的或是分開的

陣列式序列容器 (array-based)



元素的空間儲存型式是緊鄰在一起，適合使用下標運算子直接跳到指定的位置存取元素，對記憶空間的完整性要求較高。容器包含向量陣列(vector)，佇列陣列(deque)

節點式序列容器 (node-based)



元素的空間儲存型式是分散的，無法使用下標運算子直接跳到指定的位置存取元素，但對記憶空間的完整性要求較低。容器包含串列容器(list)

sequence container

序列容器(二)：建構物件

- 以直接方式設定物件初值

```
vector<int>  a1 ;              // a1 : 空向量陣列
list<float>  a2 ;              // a2 : 空串列

deque<int>   b1(10,3) ;        // b1 : 10 個整數 3
list<float>  b2(10) ;          // b2 : 10 個未設定初值的元素
```

- 以複製方式設定物件初值

```
int   c[10] = { 3 , 2 , 7 , 4 , 8 , 2 , 1 , 5 , 9 , 6 } ;

vector<int>  c1(c+1,c+7) ;                    // c1 = 2 7 4 8 2 1
deque<int>   c2(c1.begin()+1,c1.begin()+4) ; // c2 = 7 4 8
list<int>    c3(c+4,c+8) ;                    // c3 = 8 2 1 5
list<int>    c4(c2.begin(),c2.end()) ;        // c4 = 7 4 8
```

序列容器(三)：建構物件

■ 複製與指定物件

```
list<int>    d1(3,2) ;     // 3 個 2
list<int>    d2(d1) ;      // 複製 d1 到 d2
list<int>    d3 = d1 ;     // 同上
vector<int>  d4 = d1 ;     // 錯誤，須改成 d4(d1.begin(),d1.end())
...

d1 = d2 ;                  // 使用指定運算子複製 d2 到 d1
```

■ 序列容器陣列

```
vector<int>    foo1[5] ;    // 5 個向量陣列
deque<int>     foo2[6] ;    // 6 個佇列陣列
```

❖ 序列容器無法使用傳統陣列的初值方式設定初值

```
vector<int>  foo3    = { 3 , 2 , 5 } ;   // 錯誤
deque<int>   foo4[3] = { 3 , 2 , 5 } ;   // 錯誤
```

序列容器(四)：設定物件

- 同容器物件間的設定：使用指定運算子

```
deque<int>  a(3,5) , b ;
list<int>   c ;
...
b = a ;     // a 資料複製給 b
c = a ;     // 錯誤 , c 與 a 為不同類型容器
```

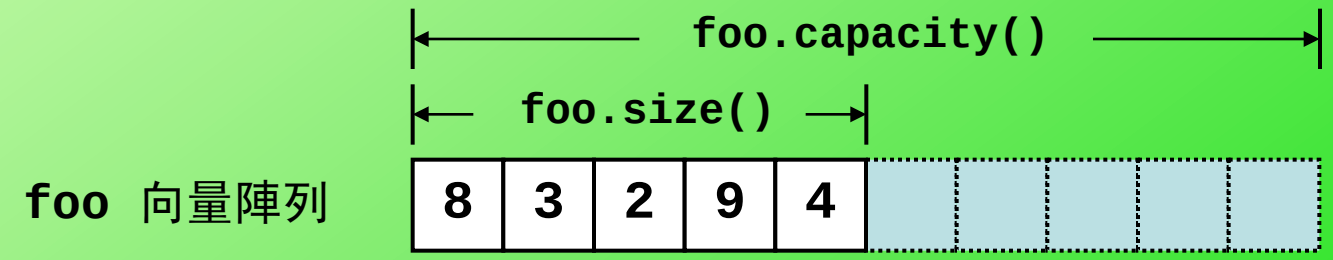
- 不同容器物件間的設定：使用 **assign** 成員函式

```
int  a[4] = { 2 , 3 , 5 , 1 } ;
deque<int>   b(a,a+4) , c ;          // b = 2 3 5 1 , c 為空佇列

b.assign(a+1,a+4) ;                  // b = 3 5 1
c.assign(a,a+3) ;                    // c = 2 3 5
...
list<int>  d ;                       // d  為空串列
d.assign(b.begin(),b.end()) ;        // d = 3 5 1
```

向量陣列(一)：空間管理

- 空間配置：
  - 預留空間：**capacity**
  - 陣列長度：**size**
- 成員函式：

`size()`	回傳陣列長度， 也就是元素個數
`resize( n , v )`	調整陣列長度到 **n** 個， 若 **n** 比現有的長度長，則補上 **n-size()** 個元素，且其值為 **v**。若較小則直接去除第 **n+1** 個元素之後的元素
`capacity()`	回傳預留的元素個數
`reserve( n )`	將陣列預留元素個數設為 **n**



向量陣列(二)：空間管理

■ 預留空間長度與陣列長度

```
vector<int>   foo ;      // foo 為空陣列，起始預留長度為 1
foo.reserve(100) ;       // foo 陣列的預留長度設為 100
foo( int i = 0 ; i < 100 ; ++i ) foo[i] = i ;         // 錯誤 陣列長度為 0
foo( int i = 0 ; i < 100 ; ++i ) foo.push_back(i) ; // 陣列長度增到 100
```

❖ 預留空間長度不能比陣列長度小，當陣列使用 **push_back** 時，陣列會將新元素加入陣列末尾，同時陣列長度自動加一。當預留長度比陣列長度小時，陣列會自動搬家，陣列所有元素會複製到新的記憶空間，之後原有陣列所佔用的空間將會歸還系統，搬家後的預留長度通常會被設為新陣列長度的兩倍

```
vector<int>   foo ;          // foo 為空陣列
for ( int i = 0 ; i < 100 ; ++i ) {
    foo.push_back(i) ;
    if ( foo.size() == foo.capacity() ) cout << foo.size() << endl ;
}
```

➢ 輸出 **1 2 4 8 16 32 64**

向量陣列(三)：向量陣列物件

- 物件產生方式

```
vector<int>   foo1 ;           // foo1 為空陣列
vector<char>  foo2(3) ;        // foo2 為三個字元的陣列
vector<char>  foo3(5,'a') ;    // foo3 為五個 'a' 字元的陣列

vector<char>  foo4(9,'b') ;    // foo4 為九個 'b' 字元的陣列

// 將 foo4[1] 到末尾的所有元素複製到 bar1
vector<char>  bar1(foo4.begin()+1,foo4.end()) ;

int  foo5[] = { 2 , 4 , 3 , 7 } ;

// 將 foo5[1] 到 foo5[3] 複製到 bar2
vector<int>   bar2(foo5+1,foo5+4) ;


vector<char>  bar3(foo3) ;     // 將 foo3 複製給 bar3
vector<int>   bar4(foo5) ;     // 錯誤 foo5 不能為傳統陣列
```

向量陣列(四)：使用向量陣列

■ 利用下標運算子：不會檢驗下標範圍

```
vector<int>  foo ;                          // foo 為空陣列
foo.reserve(10) ;                           // 預留長度設為 10 但陣列長度仍為 0
for ( int i = 0 ; i < 10 ; ++i )            // 錯誤  下標超出陣列長度範圍
    foo[i] = i ;
```

❖ 下標運算子不會改變陣列長度或預留長度，使用者要自行檢驗使用的下標須在陣列長度的範圍內，即 **[0,foo.size()-1]**

■ 使用 at 下標成員函式：會檢驗下標範圍

```
vector<int>  foo(10) ;                      // 陣列長度為 10
for ( int i = 0 ; i < 10 ; ++i )            // 正確
    foo.at(i) = i ;
```

❖ at 成員函式與下標運算子用法一樣，但若下標超出範圍會擲出錯誤訊息

向量陣列(五)：對調與前後

- 對調兩陣列

```
vector<int>  a(3,5) , b(5,2) ;   // a = 5 5 5
                                 // b = 2 2 2 2 2

b.swap(a) ;                      // a = 2 2 2 2 2
                                 // b = 5 5 5
```

- 前後元素

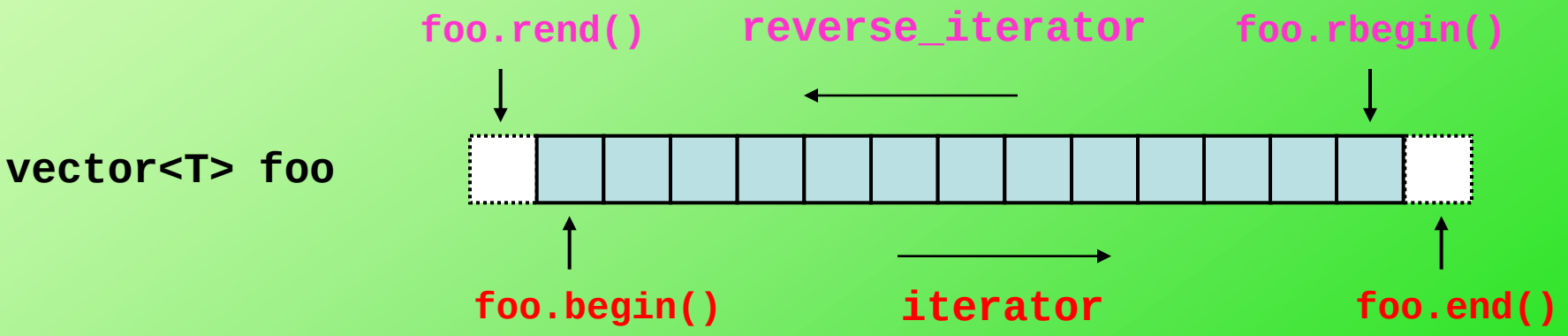
```
vector<int>  c(4,3) ;            // c = 3 3 3 3

foo.front() = 8 ;                // c = 8 3 3 3
foo.back() = 5 ;                 // c = 8 3 3 5

cout << foo.front() << ' '       // 印出 8 5
     << foo.back() << endl ;
```

向量陣列(六)：正逆向迭代器

- 四種迭代器

非常數型正逆向迭代器	常數型正逆向迭代器
`vector<T>::iterator`	`vector<T>::const_iterator`
`vector<T>::reverse_iterator`	`vector<T>::const_reverse_iterator`

❖ 當向量物件為常數時，須使用常數型迭代器才能取用向量陣列內的元素



❖ `foo.end()` 與 `foo.rend()` 所指向的空間都不在陣列內

向量陣列(七)：使用迭代器

- 使用正逆向迭代器

```
int     dat[] = { 1 , 2 , 3 , 4 , 5 } ;
vector<int>  foo(dat,dat+5) ;
vector<int>::iterator  i ;

// 由前向後列印出 1 2 3 4 5
for ( i = foo.begin() ; i != foo.end() ; ++i ) cout << *i << ' ' ;

vector<int>::reverse_iterator  j ;

//由後向前逆向列印出 5 4 3 2 1
for ( j = foo.rbegin() ; j != foo.rend() ; ++j ) cout << *j << ' ' ;
```

❖ foo[i] 相當於 *(foo.begin()+i)

向量陣列(八)：變更陣列長度

■ 變更長度：resize

```
vector<int>  foo(10,3) ;      // 10 個 3

foo.resize(6) ;               // 去除末尾 4 個元素，變成 6 個 3
foo.resize(15,4) ;            // 共 15 個元素，前 6 個元素為原有元素 3，
                              // 後 9 個為新加入的元素 4
```

❖ 當變更後的長度比預留長度大時，則整個陣列會自動搬家進而影響執行效率

■ 清空陣列：clear

```
vector<int>   foo(10) ;                          // 陣列長度為 10
foo.clear() ;                                    // 陣列長度歸零
if ( foo.empty() )  cout << "空陣列" ;           // 印出空陣列
```

➢ empty() 成員函式用來檢查向量陣列是否為空陣列

❖ 清空後的陣列長度為零，但陣列的預留長度仍保持不變

向量陣列(九)：插入元素

■ insert 成員函式

```
vector<int>  foo(5,3) ;                    // foo : 3 3 3 3 3

// (1) 插入數值 7 到 foo.begin()+1 所指向的位置元素之前
foo.insert( foo.begin()+1 , 7 ) ;          // foo : 3 7 3 3 3 3

// (2) 插入 2 個整數 4 到 foo.end()-2 所指向的位置元素之前
foo.insert( foo.end()-2 , 2 , 4 ) ;        // foo : 3 7 3 3 4 4 3 3
```

■ insert 泛型函式

```
vector<int>  a(2,7) , b(2,6) ;                    // a = 7 7 , b = 6 6
int  c[] = { 9 , 8 , 3 , 2 } ;

insert( a.begin()+1 , b.begin() , b.end() ) ; // a = 7 6 6 7
insert( a.begin() , c+1 , c+3 ) ;                 // a = 8 3 7 6 6 7
```

❖ 所有插入函式的第一個參數皆為插入點的位置，欲插入的元素置放在插入點之前，由於插入元素須連帶搬動其他元素的位置，是個昂貴的函式

向量陣列(十)：移除元素

- 去除單一元素

```
vector<int>  foo(6) ;

for ( int i = 0 ; i < 6 ; ++i ) foo[i] = i ;    // foo = 0 1 2 3 4 5

foo.erase( foo.begin()+2 ) ;                    // foo = 0 1 3 4 5
```

指定元素位置

- 去除範圍內的元素

```
vector<int>  foo(6) ;

for ( int i = 0 ; i < 6 ; ++i ) foo[i] = i ;    // foo = 0 1 2 3 4 5

// 去除 [foo.begin()+1,foo.end()-1) 內的元素
foo.erase( foo.begin()+1 , foo.end()-1 ) ;      // foo = 0 5
```

向量陣列(十一)：多重陣列

■ 二維陣列：陣列元素也是陣列

```
int  i , j ;
int  row = 3 , col = 4 , val = 5 ;  // row : 列 , col : 行

// 定義 matrix[row][col] 矩陣，且其內的元素初值皆為 val
vector< vector<int> >  matrix(row,vector<int>(col,val)) ;
```

須保留至少一個空格

```
for ( i = 0 ; i < row ; ++i ) {
    for ( j = 0 ; j < col ; ++j ) cout << matrix[i][j] << ' ' ;
    cout << endl ;
}

cout << "列數 : " << matrix.size() << endl ;
cout << "行數 : " << matrix[0].size() << endl ;
```

- 矩陣的總列數 : `matrix.size()`
- 第 **i+1** 列元素個數 : `matrix[i].size()`

向量陣列(十二)：多重陣列

- 雙重矩陣內的各列元素個數可以不相等

```
int  i , j , n = 6 ;
vector< vector<int> >  pascal ;

for ( i = 0 ; i < n ; ++i ) {

    pascal.push_back( vector<int>(1,1) ) ;

    for ( j = 1 ; j < i ; ++j )
        pascal[i].push_back( pascal[i-1][j-1] + pascal[i-1][j] ) ;
    if ( i > 0 ) pascal[i].push_back(1) ;

}

for ( i = 0 ; i < pascal.size() ; ++i ) {
    for ( j = 0 ; j < pascal[i].size() ; ++j )
        cout << setw(4) << pascal[i][j] ;
    cout << endl ;
}
```

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
```

向量陣列(十三)：多重陣列

■ 三重陣列

```
typedef  vector<int>       array1d ;      // 一維陣列
typedef  vector<array1d>   array2d ;      // 二維陣列
typedef  vector<array2d>   array3d ;      // 三維陣列

int  i , j , k ;
i = j = k = 3 ;

// foo 為 ixjxk 的三維陣列，所有元素的初值皆為零
array3d  foo(i,array2d(j,array1d(k,0))) ;

// 設定三維陣列元素數值為其下標和
for ( i = 0 ; i < foo.size() ; ++i )
    for ( j = 0 ; j < foo[i].size() ; ++j )
        for ( k = 0 ; k < foo[i][j].size() ; ++k )
            foo[i][j][k] = i + j + k ;
```

❖ 多重陣列物件在離開了其定義領域後，其所佔用的記憶空間會自然消失

向量陣列(十四)：多重陣列

- 容器內的元素可以為其他種容器

```
vector< list<double> >  a ;   // 向量內含雙精確度浮點數串列

deque< vector<int> >    b ;   // 佇列內含整數向量陣列

list< deque<char> >     c ;   // 串列內含字元佇列


// 設定 a 陣列共有 n 個串列，每一個串列有 n+1 個元素，元素值為 3
int i , j , n = 6 ;
for ( i = 0 ; i < n ; ++i ) {
    a.push_back( list<int>(1,3) ) ;
    for ( j = 0 ; j < n ; ++j ) a[i].push_back(3) ;
}
```

❖ 使用者可根據問題需求自行選擇適當的容器加以合成使用

佇列陣列(一)：使用方式

- 可由前端加入或移除元素

```
// 由前端加入元素
void  push_front( const T& ) ;

// 由前端取出元素
void  pop_front() ;
```

- 不用預留空間：沒有 **reserve** 與 **capacity** 函式
- 迭代器類型

```
deque<T>::iterator               iter1 ;   // 正向迭代器
deque<T>::const_iterator         iter2 ;   // 正向常數迭代器

deque<T>::reverse_iterator       iter3 ;   // 逆向迭代器
deque<T>::const_reverse_iterator iter4 ;   // 逆向常數迭代器
```

❖ 佇列陣列使用方式幾乎與向量陣列相同，在此只敘述其間差異

佇列陣列(二)：與向量陣列比較

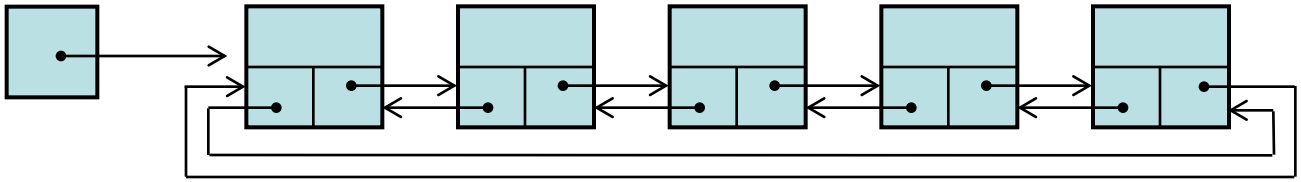
- 佇列陣列與向量陣列比較

	向量陣列	佇列陣列
空間管理	元素相鄰	元素分段相鄰
由前端加入或移除元素	不可	可以
執行效率	快	稍差但相近不遠

❖ 佇列陣列與向量陣列兩者使用方式與效率幾乎相同，因此選用的主要標準是以程式是否須由陣列前端加入或移除元素而定

串列(一)：雙鏈節結構

■ 雙鏈節串列資料結構



- 資料以節點為單位，節點間以分散方式儲存
- 節點資料不緊鄰在一起，沒有預留空間儲存機制
- 若有多筆資料須存入串列，則須以一筆一筆方式依次向作業系統取得空間儲存
- 每個節點都透過前後兩指標連結前後的節點，節點不能直接跳到非連結在一起的其他節點

doubly-linked list

串列(二)：迭代器

■ 雙向迭代器

串列的迭代器類型為雙向迭代器，僅能在節點間前後一步步移動

```
int  a[] = { 2 , 4 , 9 , 3 } ;
list<int>   foo(a,a+4) ;
list<int>::iterator  i ;    // 串列迭代器為雙向迭代器

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

■ 迭代器種類

`list<T>::iterator`	正向迭代器
`list<T>::const_iterator`	正向常數型迭代器
`list<T>::reverse_iterator`	逆向迭代器
`list<T>::const_reverse_iterator`	逆向常數型迭代器

❖ 雙向迭代器不能用加減運算式 i = i + 1 或 i = i - 1 往前或往後移動迭代器，因此不能用來定義下標運算子

串列(三)：使用正逆向迭代器

■ 使用正逆向迭代器

```
list<int>                    a ;
list<int>::iterator          i ;     // i 為正向迭代器
list<int>::reverse_iterator  j ;     // j 為逆向迭代器

// 依次存入 1 2 3 ... 8 9 10 數字於串列內
for ( int i = 0 ; i < 10 ; ++i ) a.push_back(i+1) ;


// 使用正向迭代器由前往後列印 1 2 3 ... 8 9 10
for ( i = a.begin() ; i != a.end() ; ++i ) cout << *i << ' ' ;
cout << endl ;


// 使用逆向迭代器由後往前列印 10 9 8 ... 3 2 1
for ( j = a.rbegin() ; j != a.rend() ; ++j ) cout << *j << ' ' ;
cout << endl ;
```

❖ **begin()** 與 **end()** 為正向迭代器，**rbegin()** 與 **rend()** 為逆向迭代器

串列(四)：空間管理

■ 串列空間管理成員函式

成員函式	用途
`size_type size()`	回傳串列元素個數
`void resize( size_type n , T val = T() )`	調整串列元素個數為 **n**，若 **n** 比現有個數大，則補上 **n-size()** 個元素，初值為 **val**。若 **n** 較小，則去除第 **n+1** 個元素之後的所有元素
`void clear()`	去除串列所有元素
`bool empty()`	檢查串列是否為空串列

❖ **size_type** 為長度單位，通常為無號整數

串列(五)：串列排序

- 兩種排序成員函式：**sort**

```
int  a[10] = { 14 , 3 , -25 , 15 , 33 , -49 , 9 , 72 , -11 , 89 } ;

list<int>            foo(a,a+10) ;
list<int>::iterator  i ;
```

```
// (1) : 無參數模式，預設排序方式為由小排到大
foo.sort() ;
```

```
// 輸出 : -49 -25 -11 3 9 ... 33 72 89
for ( i = foo.begin() ; i != foo.end() ; ++i ) cout << *i << ' ' ;
cout << endl ;
```

```
// (2) : 輸入排序函式或函式物件，用以定義排序方式
foo.sort( greater<int>() ) ;                    // greater<int> 數字大者優先
```

```
// 輸出 : 89 72 33 ... 9 3 -11 -25 -49
for ( i = foo.begin() ; i != foo.end() ; ++i ) cout << *i << ' ' ;
cout << endl ;
```

串列(六)：插入元素

- 插入元素於串列內

成員函式	用途
`iterator insert( iterater i , const T& val=T() )`	插入一元素 **val** 於迭代器 **i** 所指向的元素之前，回傳所指向的新元素迭代器
`void insert( iterator i , size_type no , const T& val )`	在迭代器 **i** 所指向元素之前插入 **no** 個值為 **val** 元素
`void insert( iterator i , iterator a , iterator b )`	將 **[a,b)** 範圍內的元素插入迭代器 **i** 所指向的元素之前

```
int  a[3] = { 7 , 4 , 9 } ;
list<int>            foo(a,a+3) ;          // foo = 7 4 9
list<int>::iterator  i = foo.begin() ;

++i ;                                      // i 指向 4

foo.insert( i , 2 , 3 ) ;                  // foo = 7 3 3 4 9
foo.insert( i , 8 ) ;                      // foo = 7 3 3 8 4 9
```

串列(七)：移除元素

■ 移除串列內的元素

成員函式	用途
`void remove( const T& val )`	移除串列所有資料值為 `val` 的元素
`void remove_if( un_pred p )`	移除滿足單元判斷式 `p` 的所有元素
`iterator erase( iterator i )`	去除迭代器所指向的元素，回傳所去除節點之後位置的迭代器
`iterator erase( iterate start , iterate end )`	去除範圍內的所有元素，回傳迭代器指向去除範圍後一個位置

```
int  a[6] = { 7 , 4 , 6 , 2 , 9 , 4 } ;
list<int>              foo(a,a+6) ;

// 去除元素值為 4 的所有元素
foo.remove( 4 ) ;                                  // foo = 7 6 2 9

// 去除所有比 5 大的元素
foo.remove_if( bind2nd(greater<int>(),5) ) ;  // foo = 2
```

串列(八)：串列合併

- 破壞性合併成員函式：**merge**

➢ 合併前

➢ 合併

```
foo.merge(bar) ;
```

➢ 合併後

❖ 合併過程會破壞併入的串列，同時若原有串列皆為排序完成的串列，則合併後，串列也會依大小排序。使用者也可以輸入合併的排序方式

串列(九)：串列合併

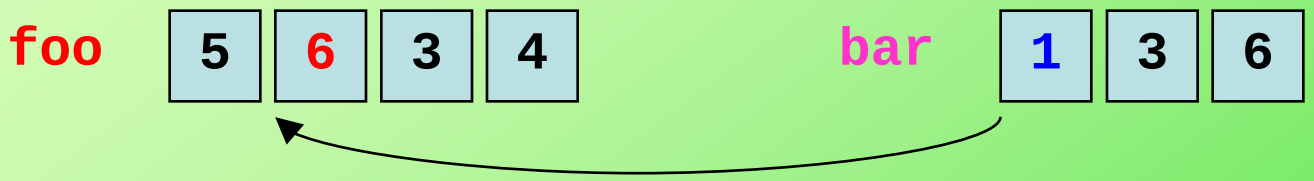
■ 進階合併

```
// 函式用來比較個位數大小，小者優先
bool  remainder( int m , int n ) { return m%10 < n%10 ; }

int  a[5] = { 20 , 45 , 25 , 37 , 9 } ;
list<int>  foo1(a,a+4) ;                    // foo1 = 20 45 25 37
list<int>  bar1(a+2,a+5) ;                  // bar1 = 25 37 9
list<int>  foo2 = foo1 , bar2 = bar1 ;

list<int>::iterator  i ;

// 將 bar1 併入 foo1 後 bar1 將變成空串列
foo1.merge(bar1) ;

// 印出 foo1 : 20 25 37 9 45 25 37
for( i = foo1.begin() ; i != foo1.end() ; ++i ) cout << *i << ' ' ;
cout << endl ;

// 利用函式依個位數大小由小到大方式將 bar2 併入 foo2 後 bar2 將變成空串列
foo2.merge(bar2,remainder) ;

// 印出 foo2 : 20 45 25 25 37 37 9
for( i = foo2.begin() ; i != foo2.end() ; ++i ) cout << *i << ' ' ;
cout << endl ;
```

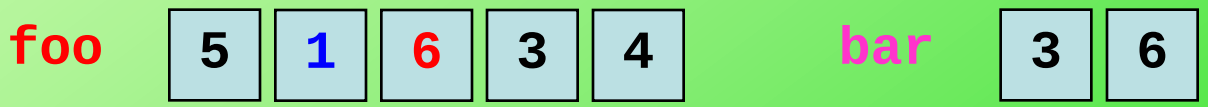
串列(十)：切割搬移串列

■ 切割搬移串列成員函式：**splice**

```
void  splice( iterator i , list<T>& foo , iterator j )
```




```
list<int>::iterator  iter = foo.begin() ;
++iter ;
foo.splice(iter,bar,bar.begin()) ;
```



❖ 兩串列經過切割搬移後，元素總數量仍保持不變

串列(十一)：切割搬移串列

■ 三種切割搬移串列成員函式：**splice**

成員函式	用途
`void splice( iterator i , list<T>& foo , iterator j )`	將迭代器 `j` 指到的 `foo` 串列內單一個元素搬到迭代器 `i` 所指向的元素之前
`void splice( iterator i , list<T>& foo , iterator a , iterator b )`	將 `foo` 串列在迭代器 `[a,b)` 範圍內的元素搬到迭代器 `i` 所指向的元素之前
`void splice( iterator i , list<T>& foo )`	將整串 `foo` 串列移到迭代器 `i` 所指到的元素之前，同時 `foo` 串列就成為空字串

❖ 所有切割函式的前兩個參數為所要插入的位置與欲切割的串列

串列(十二)：逆轉與對調

成員函式	用途
`void reverse()`	將串列元素前後次序倒過來
`void swap( list<T>& foo )`	與輸入的串列物件對調所有元素

```
char                    a[] = "abbcccabaa" ;

list<char>              foo(a,a+10) ;      // foo = abbcccabaa
list<char>              bar(a+6,a+10) ;    // bar = abaa

// 逆轉 foo
foo.reverse() ;                            // foo = aabacccbba

// bar 與 foo 對調
bar.swap(foo) ;                            // foo = abaa
                                           // bar = aabacccbba
```

串列(十三)：去除重複元素

成員函式	用途
void unique()	去除相鄰且數值相等的節點個數到一個
void unique(bin_pred p)	同上，但使用雙元判斷式 p 自行設定元素相等的涵義

```
// 若兩整數的個位數相等   則兩數相等
bool  equal( int a , int b ) { return a%10 == b%10 ; }
...
int  a[10] = { 23 , 24 , 63 , 23 , 14 , 14 , 4 , 1 , 91 , 2 } ;

list<int>   foo(a,a+10) ;    // foo : 23 24 63 23 14 14 4 1 91 2
foo.unique() ;               // foo : 23 24 63 23 14 4 1 91 2

list<int>   bar(a,a+10) ;    // bar : 23 24 63 23 14 14 4 1 91 2
bar.unique(equal) ;          // bar : 23 24 63 14 1 2
```

❖ 檢查兩元素是否相等的前題是此兩元素要緊鄰在一起

容器轉換器

- 容器轉換器

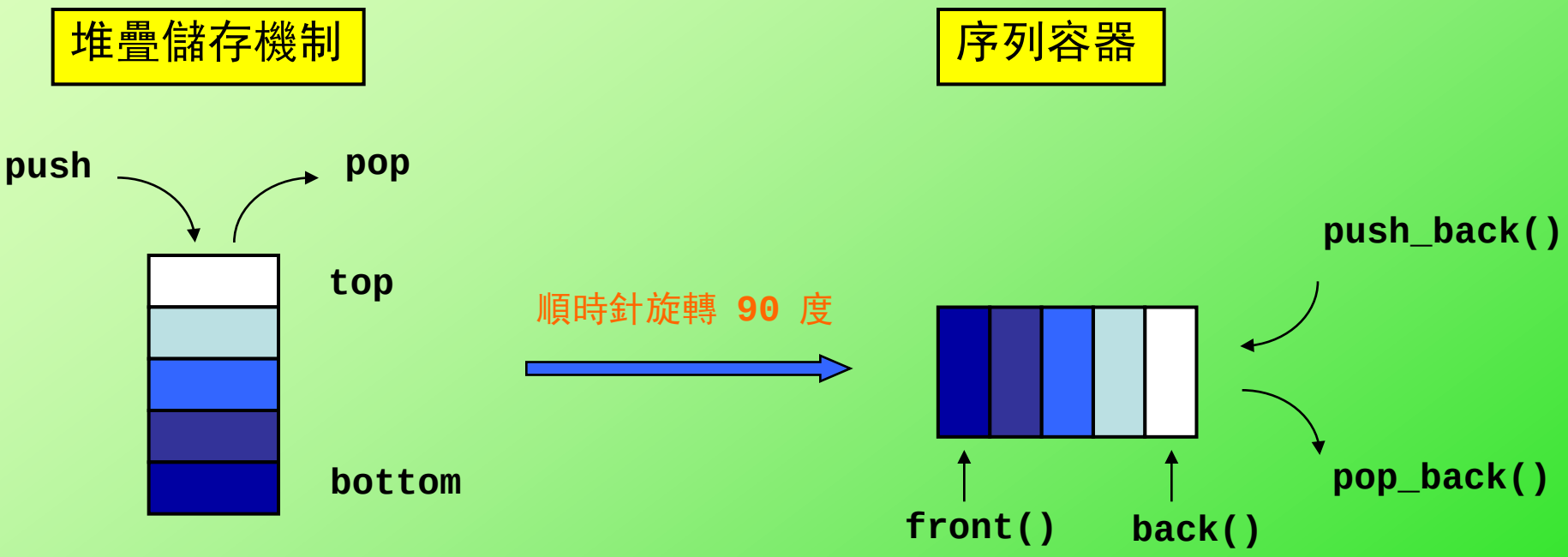
  利用序列容器所建構出來的特殊用途容器

容器轉換器	資料結構機制	標頭檔
stack	堆疊	stack
queue	佇列	queue
priority_queue	優先佇列	queue

container adaptor

堆疊轉換器(一)：儲存機制

■ 堆疊：先進後出資料模擬機制



➢ 堆疊儲存機制可以直接利用向量陣列，佇列陣列或者是串列容器來模擬

stack

堆疊轉換器(二)：產生物件

- 建構堆疊物件：

```
// 整數堆疊 使用預設的佇列陣列來儲存整數
stack< int >                          a ;

// 整數堆疊 使用向量陣列來儲存整數
stack< int , vector<int> >            b ;

// 字元堆疊 使用串列來儲存資料
stack< char , list<char> >            c ;
```

- 定義堆疊時，須將堆疊所處理的元素型別與其所使用的序列容器一起寫上，同時要將所使用容器的標頭檔加入程式碼一起編譯。使用者若不設定使用的容器，則以佇列陣列為堆疊預設使用的容器

堆疊轉換器(三)：成員函式

成員函式		用途
`bool`	`empty() const`	檢查是否為空堆疊
`size_type`	`size() const`	回傳堆疊元素個數
`void`	`pop()`	去除堆疊最上面的元素
`void`	`push(const T& val)`	將 `val` 元素推入到堆疊內
`T&`	`top()`	回傳最上面元素的參考
`const T&`	`top() const`	同上，但回傳常數參考

- 以上的 **pop** 函式並不會回傳所去除的資料。若要察看或是直接更換堆疊容器最上面的元素，須使用 **top** 成員函式

堆疊轉換器(四)：使用範例

- 迴文詩

```
char  p[] = "斜風遶徑曲  從石帶山連  花餘拂戲鳥  數密隱鳴蟬" ;

int  s = sizeof(p) / sizeof(char) ;    // 計算字串長度

stack<string>  poem ;                  // c++ 字串堆疊

// 由每兩個字元所形成的字串存入堆疊後再取出
for ( int i = 0 ; i < s ; i += 2 )
    poem.push( string(p+i,p+i+2) ) ;
```

```
while ( ! poem.empty() ) {
    cout << poem.top() ;
    poem.pop() ;
}
```

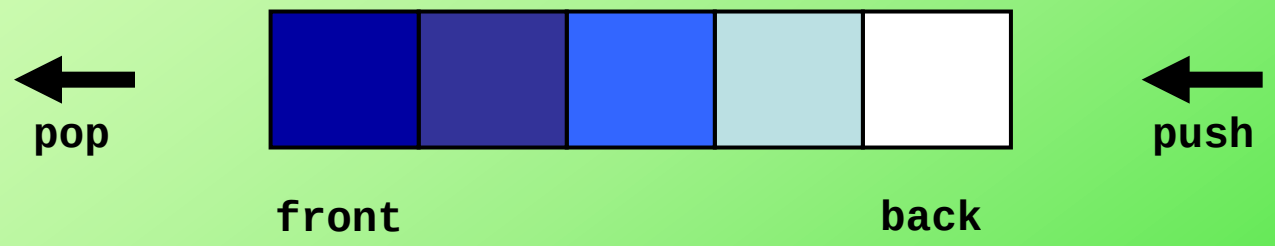
```
cout << endl ;
```

蟬鳴隱密數　鳥戲拂餘花　連山帶石從　曲徑遶風斜

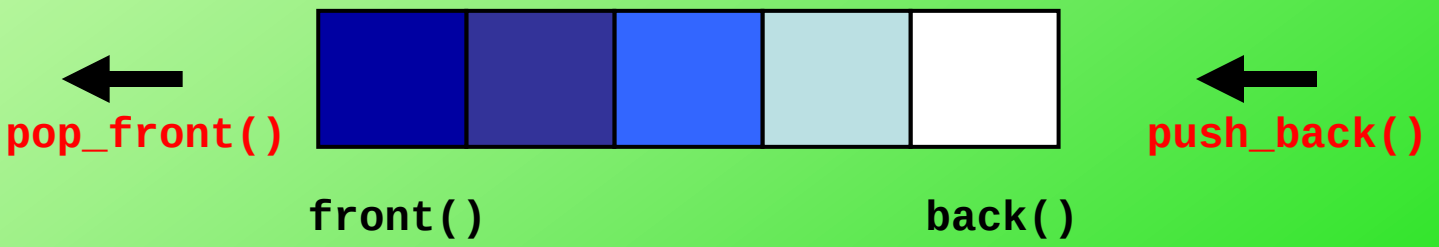
佇列轉換器(一)：儲存機制

- 佇列：先進先出資料模擬機制

佇列儲存機制



佇列陣列或串列



➢ 佇列可用佇列陣列或是串列來模擬，但不能使用向量陣列

queue

佇列轉換器(二)：產生物件

- 建構佇列轉換器物件：

```
// 整數佇列 使用預設的佇列陣列來儲存整數
queue< int >                          a ;

// 同上
queue< int , deque<int> >             b ;

// 字元佇列 使用串列來儲存資料
queue< char , list<char> >            c ;
```

- 定義佇列轉換器時，須將轉換器所處理的元素型別與其所使用的序列容器一起寫上，同時要將所使用容器的標頭檔加入程式碼一起編譯。使用者若不設定使用的容器，則以佇列陣列為轉換器預設使用的容器

佇列轉換器(三)：成員函式

成員函式		用途
`bool`	`empty() const`	檢查是否為空佇列
`size_type`	`size() const`	回傳佇列儲存元素個數
`void`	`pop()`	去除佇列最前端的元素
`void`	`push(const T& val)`	將 `val` 元素推入到佇列內
`T&`	`front()`	回傳最前端元素的參考
`const T&`	`front() const`	同上，但回傳常數參考
`T&`	`back()`	回傳最末端元素的參考
`const T&`	`back() const`	同上，但回傳常數參考

佇列轉換器(四)：使用範例

- 射箭成績

```
int  i ;
const int NO = 4 ;
string  contestant[NO] = { "趙" , "錢" , "孫" , "李" } ;

queue<int>  score ;
srand( static_cast<unsigned int>(time(NULL)) ) ;

// 以亂數方式產生成績
for ( i = 0 ; i < NO ; ++i ) score.push(rand()%11) ;

i = 0 ;
while ( ! score.empty() ) {
    cout << score.front() << " : " << contestant[i++] << "  " ;
    score.pop() ;
}
cout << endl ;
```

優先佇列轉換器(一)

- 優先佇列 ：

  存入的資料依其大小會自動移動到適當的位置儲存，資料並不會因存入的次序較慢而較晚被取出

  - 資料取出的順序完全是依照其大小而定
  - 優先佇列的資料結構是透過陣列來完成，元素的加入或是移除完全是在陣列的末尾進行
  - 加入的元素須透過一連串的比較與對調才能由末尾移動到適當的位置，同時對調的過程須使用到下標運算子
  - 使用者可以設定元素間的比較標準用以決定其優先順序

❖ 優先佇列轉換器只能利用向量陣列或是佇列陣列來模擬

priority queue

優先佇列轉換器(二)：物件

- 建構優先佇列轉換器物件：

```
// 使用 預設的向量陣列 與 less<double> 來儲存浮點數
// 數字越小，優先順序越低
priority_queue< double >       a ;


// 使用 佇列陣列 與 greater<int> 排定整數的優先順序
// 數字越大，優先順序越低
priority_queue< int , deque<int> , greater<int> >      b ;
```

- 在建構物件時，可以直接將某範圍內的元素存入

```
string  foo = "cats" ;

// 使用 佇列陣列 與預設的 less<char> 來儲存字元
// 字母順序越小，優先順序越低。 c 物件儲存字母的取出順序為 tsca
priority_queue< char , deque<char> >  c(foo.begin(),foo.end() ) ;
```

優先佇列轉換器(三)：函式

成員函式	用途
`bool empty() const`	檢查是否為空優先佇列
`size_type size() const`	回傳優先佇列儲存元素個數
`void pop()`	去除優先佇列內最優先的元素
`void push(const T& val)`	將 `val` 元素推入到優先佇列內
`const T& top() const`	回傳最優先元素的常數參考

- 優先佇列內的最優先元素要用 `top()` 取得，而非 `front()`。同時優先佇列內的最優先元素是經過比較後產生的，使用者不能直接更換最優先元素的值

優先佇列轉換器(四)：範例

- 射箭成績排序

```
struct  Record {
    string  name ;
    int     score ;
    Record( string n , int s ) : name(n) , score(s) {} ;
} ;
bool  operator< ( const Record& a , const Record& b ) {
    return  a.score < b.score ;
}
ostream&  operator<< ( ostream& out , const Record& foo ) {
    return  out << foo.name << " : " << foo.score ;
}
...
priority_queue<Record>  player ;      // 分數低者優先次序較低

player.push(Record("趙",7)) ;  player.push(Record("錢",5)) ;
player.push(Record("孫",9)) ;  player.push(Record("李",8)) ;
```

```
while ( ! player.empty() ) {
    cout << player.top() << endl ;
    player.pop() ;
}
```

孫 : 9
李 : 8
趙 : 7
錢 : 5

範例：矩陣類別(一)

- 矩陣：利用雙重向量陣列

可動態設定矩陣的行列數，同時不須煩惱記憶空間的管理問題

```
template <class T>
class  Matrix {
  private :
    vector< vector<T> >  mat ;
    int                  row , col ;
  public :
    Matrix( int r , int c , const T& val = 0 ) : row(r) , col(c) {
       mat = vector< vector<T> >(r,vector<T>(c,val)) ;
    }
} ;
...

int  r , c ;
cin >> r >> c ;
Matrix<int>  matrix(r,c) ;
```

範例：矩陣類別(二)

- 覆載矩陣下標運算子

```
template <class T>
class  Matrix {
  private :
    vector< vector<T> >  mat ;
    int                  row , col ;
  public :
    ...
    vector<T>&  operator[]( int i ) { return  mat[i] ; }
    const vector<T>&  operator[]( int i ) const { return mat[i] ; }
} ;
...
Matrix<int>  a(4,8,1) ;
```

則

向量陣列下標運算子

```
a[3][5]   ⟺   a.operator[](3).operator[](5)
```

矩陣下標運算子

程式

輸出

範例：多項式類別(一)

- 多項式函數

$$f(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

多項式的每一項包含次方數與係數，可用結構型別將其包裹在一起

```
template <class T>
struct  Item {
    int   deg ;        // 次方數
    T     coeff ;      // 係數
} ;
```

將每一項存入串列內，則多項式類別可以設計成

```
template <class T>
class  Polynomial {
  private :
    list< Item<T> >   items ;    // 儲存多項式的每一項
  ...
} ;
```

❖ 串列內的項式是由小次方儲存到大次方

範例：多項式類別(二)

- 串列資料成員只儲存係數非零的項式，避免空間的浪費

$$f(x) = 1 + x^{10000}$$

- 當新項式插入多項式時，新項式須依照項式的次方數由小到大插到適當的位置

$$f(x) = 1 + x^{10000} \quad \longleftarrow \quad 2\ x^{500}$$

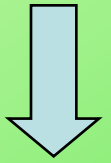
$$f(x) = 1 + 2\ x^{500} + x^{10000}$$

❖ 本類別並不適合使用優先串列儲存項式資料，因優先串列無法讓程式以不破壞方式自由地取得所有項式的資料

範例：課表列印(一)

- 課表資料

```
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
...
```



程式設計

	Mon	Tue	Wed	Thu	Fri
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	
...					

範例：課表列印(二)

- 程式設計

```
typedef  string  Course ;
typedef  int     Section ;

enum  Weekday  { Sun , Mon , Tue , Wed , Thu , Fri , Sat } ;

struct  Schedule {
    Course   name ;   // 課程名稱
    Weekday  day ;    // 授課時間
    Section  sct ;    // 授課節數
} ;
```

```
// 依列印的順序設定：節數小的優先，若相等，則授課日小的優先
bool  operator< ( const Schedule& a , const Schedule& b ) {
    return ( a.sct == b.sct ? b.day < a.day : b.sct < a.sct ) ;
}
```

```
class  Course_Schedule {
  private :
    // 依課程的順序，儲存課程資料
    priority_queue<Schedule>  courses ;
    ...
} ;
```

程式

輸出