Chapter 17

標準樣板函式庫(一)
簡介

標準樣板函式庫：主要類型

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

container, iterator, algorithm, functor class

標準樣板函式庫：範例(一)

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

using namespace std ;

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

// (5)定義一泛型函式，用以列印容器資料
template <class T>
void  print_container( const T& a , const T& b , char *sep ) {
    for ( T i = a ; i != b ; ++i ) cout << *i << sep ;
}
```

標準樣板函式庫：範例(二)


```
int main() {

    vector<int>  foo(20) ;      // (1)定義可存 20 個整數的向量陣列容器

    vector<int>::iterator  i ; // (2)定義處理整數向量陣列的迭代器物件

    // 利用迭代器物件將整數 1 到 20 一一存入 foo 向量陣列內
    int no = 1 ;
    for ( i = foo.begin() ; i != foo.end() ; ++i ) *i = no++ ;

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

    // (5)利用自定函式印出向量陣列的元素 且元素之間以空白分開
    print_container( foo.begin() , foo.end() , " " ) ;
    cout << endl ;

    return 0 ;
}
```

容器（一）

■ 序列容器

- 資料以序列方式儲存，資料在記憶空間的儲存方式可以是緊鄰或是分散
- 資料元素的儲存位置與資料值無任何相關
- 包含向量陣列(vector)、佇列陣列(deque)、串列(list)

■ 關聯容器

- 資料元素的儲存位置與資料值有所關聯
- 資料被放置在樹狀結構(tree structure)的節點(node)內
- 當程式須要存取容器資料時，都要在樹狀結構內依次比較相關節點內的索引(key)，才能找出資料在樹狀結構的存放位置
- 容器的資料都是經過比較後才存入樹狀結構，用以節省資料的搜尋時間
- 包含集合(set)、複集合(multiset)、映射(map)、複映射(multimap)

sequence container, associative container

容器（二）

名稱	容器	標頭檔	儲存方式
向量	vector	vector	緊鄰
佇列	deque	deque	緊鄰
串列	list	list	分散
集合	set	set	分散
複集合	multiset	set	分散
映射	map	map	分散
複映射	multimap	map	分散

❖ 各容器因儲存方式的不同，各有其最佳使用條件

容器的使用（一）

- 定義容器物件

```
vector<int>   a(10) ;       // 可存 10 個整數的向量陣列
vector<char>  b(20,'m') ;   // 可存 20 個字元，每個字元初值為 'm'
deque<double> c(5) ;        // 可存 5 個浮點數的佇列陣列 c
```

- 使用容器物件

```
// a 無任何元素
vector<int>   a ;

// 連續 10 次將整數 5 存入向量陣列末尾
for ( int i = 0 ; i < 10 ; ++i ) a.push_back(5) ;

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

❖ **push_back(5)**成員函式可將整數 **5** 由末尾加入容器內
size() 成員函式回傳向量陣列的元素個數

容器的使用（二）

- 序列容器可使用的成員函式

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

❖ V 代表可使用，X 代表不能使用

容器的使用（三）

- 使用 **deque**

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

- 使用 **list**

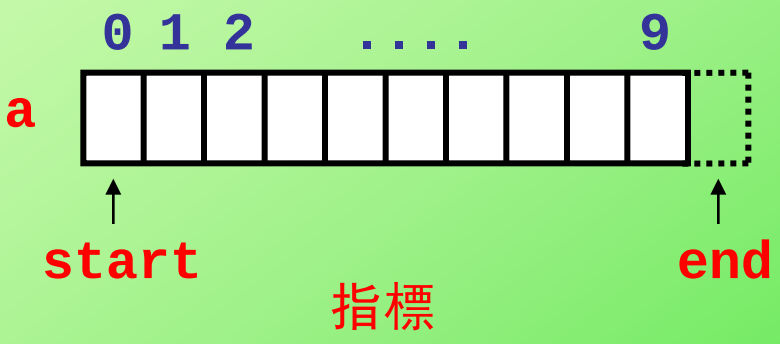
```
list<int>   b ;
b.push_back(3) ;  b.push_back(8) ;          //  b = 3 8
b.push_front(5) ; b.push_front(7) ;         //  b = 7 5 3 8
b.pop_front() ; b.pop_back() ;              //  b = 5 3
b.front() = 9 ;                             //  b = 9 3
cout << b.front() << ' ' << b.back() << endl ;
```

❖ 以上的串列物件 **b** 也可改用佇列容器 **deque<int>** 定義

迭代器(一)

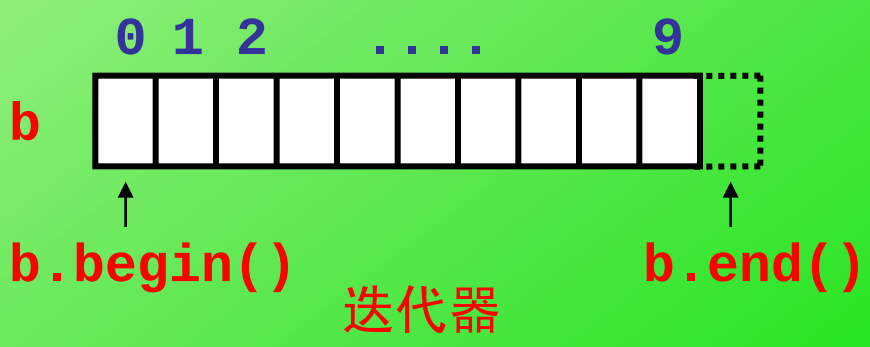
■ 指標

```
int a[10] ;
int *start = a ;
int *end = a+10 ;
int *ptr ;

ptr = start ;
while ( ptr != end ) {
    *ptr = rand()%100 ;
    ++ptr ;
}
```



■ 迭代器

```
vector<int>  b(10) ;


vector<int>::iterator  itr ;

itr = b.begin() ;
while( itr != b.end() ) {
    *itr = rand()%100 ;
    ++itr ;
}
```



迭代器(二)

■ 五種迭代器

- **輸入迭代器** `input`
  只可用來取得資料，僅能以單步方式向前移動
  `x=*i  ++i  i++  …`
- **輸出迭代器** `output`
  只可用來存入資料，移動方式同上
  `*i=x  ++i  i++  …`
- **向前迭代器** `forward`
  可取得或存入資料，但移動方式同上
  `x=*i  *i=x  ++i  i++  …`
- **雙向迭代器** `bidirectional`
  可取得或存入資料，但可以單步方式前後移動
  `x=*i  *i=x  ++i  i++  --i  i--  …`
- **隨意存取迭代器** `random-access`
  可取得或存入資料，迭代器可以跳躍式前後移動
  `x=*i  *i=x  ++i  i++  --i  i--`
  `i+n  i-n  i+=n  i-=n`
  `i<j  i>j  i<=j  i>=j …`

❖ 以上 `i , j` 為迭代器，`n` 為無號整數， `x` 為迭代器所指向的物件

迭代器(三)

- 容器因內部資料處理機制不同，各有其適用迭代器類型

名稱	容器	儲存方式	迭代器類型
向量	vector	緊鄰	隨意存取 (random-access)
佇列	deque	緊鄰	隨意存取 (random-access)
串列	list	分散	雙向 (bidirectional)
集合	set	分散	雙向 (bidirectional)
複集合	multiset	分散	雙向 (bidirectional)
映射	map	分散	雙向 (bidirectional)
複映射	multimap	分散	雙向 (bidirectional)

迭代器(四)

- 使用隨意取用迭代器的容器才有定義下標運算子

```
vector<int>  foo(3,5) ;        // 儲存 3 個整數 5 的向量陣列

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

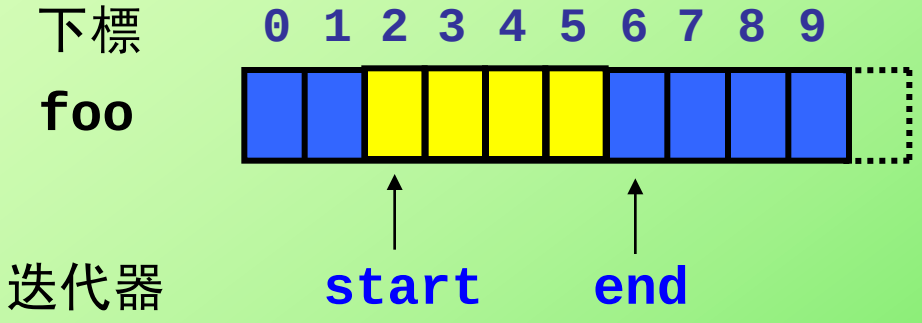
- 雙向迭代器可以被所有基本資料容器所使用

```
list<int>  foo ;              // foo 為串列容器物件
...
list<int>::iterator  i ;
for ( i = foo.begin() ; i != foo.end() ; ++i )
    cout << '>' << *i << endl ;
```

❖ 以上 **foo** 物件也可以是其它類型的容器

設定容器範圍(一)

- 指定範圍：利用兩個迭代器指定元素的前後端點



指定元素：[start , end)

元素個數： end - start

容器範圍：[foo.begin() , foo.end())

- start 指向範圍的第一個元素
  end 指向範圍末尾元素的後一個空間位置

- 以上 foo 容器的元素之間不見得須緊鄰在一起

設定容器範圍(二)

- 指定範圍內的資料經常利用迴圈來迭代處理

```
list<int>  foo ;
for ( int n = 1 ; n < 10 ; ++n ) foo.push_back(n) ;

list<int>::iterator  i ;

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

- 指定常數容器內的範圍要使用常數迭代器

```
const  vector<int>               foo(5,3) ;
vector<int>::const_iterator      i ;          // i 為常數迭代器

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

列印容器資料

■ 樣板函式：列印容器元素

```
template <class T>
void  print_container( const T& foo ) {
    typename  T::const_iterator  iter ;
    for ( iter = foo.begin() ; iter != foo.end() ; ++iter )
        cout << *iter << ' ' ;
    cout << endl ;
}
...
vector<int>  foo ;
list<char>   bar ;
...
print_container( foo ) ;  // 列印 foo 向量，樣板參數 T 為 vector<int>
print_container( bar ) ;  // 列印 bar 串列，樣板參數 T 為 list<char>
```

❖ 以上 **typename** 後的 **T::const_iterator** 代表資料型別，不可加以省略

泛型演算函式（一）

- 樣板函式：列印指定範圍元素

```
template <class T>
void  print_items( T start , T end ) {
    for ( T i = start ; i != end ; ++i ) cout << *i << ' ' ;
    cout << endl ;
}
...
vector<int>   a(10,3) ;
vector<int>::iterator  itr1 , itr2 ;      // 定義 vector 的迭代器物件
...                                       // 設定 a, itr1 , itr2
print_items( itr1 , itr2 ) ;

deque<double>    b(10) ;
deque<double>::iterator  itr3 , itr4 ;    // 定義 deque 的迭代器物件
...                                       // 設定 b , itr3 ,itr4
print_items( itr3 , itr4 ) ;

list<char>   c ;
list<char>::iterator  itr5 , itr6 ;       // 定義 list 的迭代器物件
...                                       // 設定 c , itr5 , itr6
print_items( itr5 , itr6 ) ;
```

泛型演算函式（二）

■ 排序演算函式：sort

```
vector<int>   a(10) ;
for ( int i = 0 ; i < 10 ; ++i )          // 存入 10 個亂數
    a[i] = rand() ;
sort( a.begin()+3 , a.begin()+8 ) ;       // 由小到大重排 a[3] .. a[7]
```

排序範圍 [a.begin()+3 , a.begin()+8)

■ 取代演算函式：replace

```
deque<char>   b(5,'x') ;                              // b = x x x x x
replace( b.begin()+3 , b.end() , 'x' , 'o' ) ;        // b = x x x o o
```

取代範圍 [b.begin()+3 , b.end())

❖ 須加入 algorithm 標頭檔

泛型演算函式（三）

- 泛型演算函式一樣可以處理傳統陣列元素

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

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

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

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

❖ 泛型演算函式：演算函式內的程式碼與所處理的資料型別無關

泛型演算函式（四）

- 演算函式會因函式所使用演算方法的限制而無法適用於所有的容器上

```
vector<int>   a ;
for ( int i = 0 ; i < 10 ; ++i ) a.push_back( rand() ) ;

sort( a.begin() , a.end() ) ;                    // 正確

list<int>   b ;
for ( int i = 0 ; i < 10 ; ++i )
b.push_back( rand() ) ;

sort( b.begin() , b.end() ) ;                    // 錯誤
```

- 以上錯誤的原因在於 **sort** 演算函式無法對元素非緊緊相鄰的串列容器排序

泛型演算函式（五）

- 搜尋演算函式：**find**

```
// 預設的 find 演算函式程式碼
template <class InputIter , class T>
InputIter find( InputIter start , InputIter end , const T& val ) {
   while ( start != end && *start != val ) ++start ;
   return start ;
}
```

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

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

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

函式物件類別（一）

■ 函式物件類別：類別或結構內有定義函式成員運算子，即 **operator()**

```
template <class T>
struct  Square {
    T  operator() ( const T& a ) const { return  a * a ; }
} ;

...
Square<int>  foo ;                                  // 產生 foo 平方函式物件

cout << foo(2) << endl ;                            // 列印整數 2 的平方
cout << foo.operator()(2) << endl ;                 // 同上
cout << Square<int>()(2) << endl ;                  // 不使用物件名稱但效果同上
cout << Square<int>().operator()(2) << endl ; // 同上
```

❖ **Square** 類別為函式物件類別
Square 類別物件為函式物件

functor class, function object or functor

函式物件類別（二）

- 函式物件類別可定義單參數或多參數的函式運算子

```
// 單參數函式運算子
template <class T>
struct  Cubic {
    T  operator() ( const T& a ) const { return  a * a * a ; }
} ;
```

```
cout << Cubic<double>()(1.1) << endl ;    // 計算 1.1 的立方
```

```
// 多參數函式運算子
template <class T>
struct  Small {
    bool  operator() ( const T& a , const T& b) const {
        return  a < b ;
    }
} ;
```

```
cout << Small<int>()(1,-5) << endl ;      // 列印 1 < -5 的結果
```

函式物件（一）

- 函式物件經常與泛型演算函式結合一起使用

```
template <class T>
struct  Square {
    T  operator() ( const T& a ) const { return  a * a ; }
} ;

template <class T>
struct  Cubic {
    T  operator() ( const T& a ) const { return  a * a * a ; }
} ;

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

```
vector<int>  c(3,2) ;                                          // c = 2 2 2
print_items( c.begin() , c.end() , Square<int>() ) ;   // 列印 4 4 4
print_items( c.begin() , c.end() , Cubic<int>() ) ;    // 列印 8 8 8
```

函式物件（二）

- 之前的泛型演算函式也可以處理傳統陣列

```
int  foo[4] = { -1 , -2 , 3 , 4 } ;
print_items( foo , foo+4 , Square<int>() ) ;    // 印出 1 4 9 16
print_items( foo+1 , foo+3 , Cubic<int>() ) ;   // 印出 -8 27
```

- 函式運算子也可設定回傳布林值

```
template <class T>
struct  Odd {
    bool  operator() ( const T& foo ) const {
        return  ( foo%2 ? true : false ) ;     // 檢查 foo 是否為奇數
    }
} ;
```

- 樣板函式經常與回傳布林值的函式物件類別一起使用

```
template <class S , class T>
void  print_items ( S iter1 , S iter2 , T fn ) {
    for ( S i = iter1 ; i != iter2 ; ++i )
        if ( fn(*i) ) cout << *i << ' ' ;      // 僅列印滿足條件的資料
    cout << endl ;
} ;
```

函式物件（三）

- 函式物件類別內也可以設定建構函式儲存資料成員

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

vector<int>  foo(10) ;
for ( int i = 0 ; i < 10 ; ++i ) foo[i] = rand() ;  // 產生10個亂數
```

```
// 僅列印 foo 容器內大於 50 的元素
print_items(foo.begin(),foo.end(),Greater<int>(50)) ;

// 也可僅列印 foo 容器內中的奇數
print_items(foo.begin(),foo.end(),Odd<int>()) ;
```

函式物件

函式物件

樣板函式與函式物件（一）

- 樣板函式與函式物件結合一起可增加函式運用的自由度

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

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

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

樣板函式與函式物件（二）

- 樣板函式的樣板參數可以為不同型式的函式指標或函式物件

```
// 計算 fn(item) 之值
template <class F, class T>
T  compute( F fn , T item ) {
    return fn(item) ;
}
```

```
// 樣板參數為一樣板函式指標
cout << compute( square<int> , 2 ) << endl ;          // 印出 4

// 樣板參數為一函式指標
cout << compute( cubic ,       2 ) << endl ;          // 印出 8

// 樣板參數為一函式物件
cout << compute( Abs<double>() , -2.2 ) << endl ;     // 印出 2.2
```

樣板函式與函式物件（三）

- **sort** 演算函式也可選擇性地透過樣板參數輸入比較函式，設定排序標準

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

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

template <class T>
struct  small_square {
    bool  operator() ( T a , T b ) const { return  a*a < b*b ; }
} ;

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

sort( a , a + 5 ) ;                          // a = -7 -5 3 4 8
sort( a , a + 5 , large_abs() ) ;            // a = 8 -7 -5 4 3
sort( a , a + 5 , small_square<int>() ) ;    // a = 3 4 -5 -7 8
```

比較兩數的絕對值，大者優先

比較兩數平方值，小者優先

❖ 輸入 **sort** 演算函式內的比較函式須要兩個參數

樣板函式與函式物件（四）

- 分數排序

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

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

struct  Bigger_First {
    bool operator()( const Fraction& a , const Fraction& b ) const {
        return  a.num * b.den > a.den * b.num ;
    }
} ;
```

比較兩分數，大者優先

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

// 輸出 : 7/5 2/3 3/5
for ( int i = 0 ; i < 3 ; ++i ) cout << foo[i] << ' ' ;
```

真假判斷式（一）

- 真假判斷式：函式或是函式物件類別內的函式運算子回傳一真假值

➢ 單元判斷式 **(unary predicate)** ：須要一個參數

```
// 普通函式：判斷 i 是否大於 0
bool  positive( int i ) { return i > 0 ; }
```

➢ 雙元判斷式 **(binary predicate)** ：須要兩個參數

```
// 函式物件類別：判斷 a 是否大於 b
template <class T>
struct  Larger {
    bool  operator() ( const T& a , const T& b ) const {
        return a > b ;
    }
} ;
```

STL 預設的函式物件類別

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

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

❖ 使用以上的函式物件類別須加入 **functional** 標頭檔

STL 函式物件類別：範例一(一)

- 計算陣列元素之間

連加 : $a_0+a_1+a_2+...+a_{n-1}$　　連減 : $a_0-a_1-a_2-...-a_{n-1}$

連乘 : $a_0*a_1*a_2*... *a_{n-1}$　　連除 : $a_0/a_1/a_2/.../a_{n-1}$

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

using namespace std ;

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

STL 函式物件類別：範例一(二)


```
int  main() {

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

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

    // 計算陣列元素相減 : (2*24) - 24 - 2 - 1 - 3 - 1 => 印出 : 17
    cout << compute( a , a+5 , 2*a[0] , minus<int>() ) << endl ;

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

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

    return 0 ;
}
```

STL 函式物件類別：範例二

- 內積計算

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

...

int        foo[5] = { 2 , 3 , 1 , 2 , 5 } ;     // foo = 2 3 1 2 5
list<int>  bar(foo+2,foo+5) ;                   // bar = 1 2 5

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

// 計算 (3,1,2)*(1,2,5) 的數值，印出 : 15
cout << inner_product(foo+1,foo+4,bar.begin(),0) << endl ;
```

束縛函式與否定函式（一）

- 雙變數函數與單變數函數

若有一雙變數函數

$$f(x,y) = x^2 - 2xy + y^2$$

若讓 $y = c$, c 為一固定數值，則可定義一單變數函數

$$g(x) \overset{df}{=} f(x,c) = x^2 - 2cx + c^2$$

束縛函式與否定函式（二）

- 束縛函式：對雙元函式的某一個參數設限使得此雙元函式經過參數束縛後如同一單元函式

```
bind1st : 束縛雙元函式的第一個參數
bind2nd : 束縛雙元函式的第二個參數
```

```
less<T>()(a,b)                  :  比較 a 是否比 b 小
bind1st( less<T>() , 50 )       :  限制 a 為 50 ， 比較 50 < b
bind2nd( less<T>() , 90 )       :  限制 b 為 90 ， 比較 a < 90
```

```
plus<T>()(a,b)                  :  計算 a + b 之值
bind1st( plus<T>() , 50 )       :  計算 50 + b
```

- 使用束縛函式或是之後的否定函式須加入 **functional** 標頭檔

binder

束縛函式與否定函式（三）

- 束縛函式：計算滿足一個輸入條件的數量

```
// 計算某段容器範圍內 [itr1,itr2) 滿足某個輸入條件 fn 的個數
template <class Iter , class Fn>
int  count_no( Iter itr1 , Iter itr2 , Fn fn ) {
    int no = 0 ;
    for ( ; itr1 != itr2 ; ++itr1 ) if ( fn(*itr1) ) ++no ;
    return no ;
}
...

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

```
// 計算陣列元素 x 不大於 8 的個數, 即滿足 8 > x 條件式的總個數, 印出 4
cout << count_no( a , a+10 , bind1st(greater<int>(),8) ) << endl ;
```

```
// 計算陣列元素 x 不能整除 5 的個數, 即滿足 x % 5 不為零的總個數, 印出 8
cout << count_no( a , a+10 , bind2nd(modulus<int>(),5) ) << endl ;
```

束縛函式與否定函式（四）

- 束縛函式：計算滿足兩個輸入條件的數量

```
// 計算某段容器範圍內 [itr1,itr2) 滿足兩個輸入條件 fn2 , fn3 的個數
template <class Iter , class Fn1 , class Fn2 , class Fn3>
int  count_no( Iter itr1 , Iter itr2 , Fn3 logical_fn
               Fn2 fn2 , Fn3 fn3 ) {
    int no = 0 ;
    for ( ; itr1 != itr2 ; ++itr1 )
        if ( logical_fn( fn2(*itr1) , fn3(*itr1) ) ) ++no ;
    return no ;
}
...

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

```
// 計算陣列元素介於 (5,15) 之間的個數, 即滿足 (5<x)&&(x<15) 條件的總個數,
// 印出 6
cout << count_no( a , a+10 , logical_and<bool>() ,
                  bind1st(less<int>(),5) ,
                  bind2nd(less<int>(),15) ) << endl ;
```

束縛函式與否定函式（五）

- 否定函式：用來逆轉真假判斷式的結果

not1 ： 逆轉單元真假判斷式的結果

not2 ： 逆轉雙元真假判斷式的結果

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

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

```
// 計算陣列元素可被 3 整除但不被 2 整除的元素個數,
// 即滿足 !( x%3 != 0 ) && ( x%2 != 0 ) 條件的總個數, 印出 4
cout << count_no( a , a+10 , logical_and<bool>() ,
                  not1(bind2nd(modulus<int>(),3)) ,
                  bind2nd(modulus<int>(),2) ) << endl ;
```

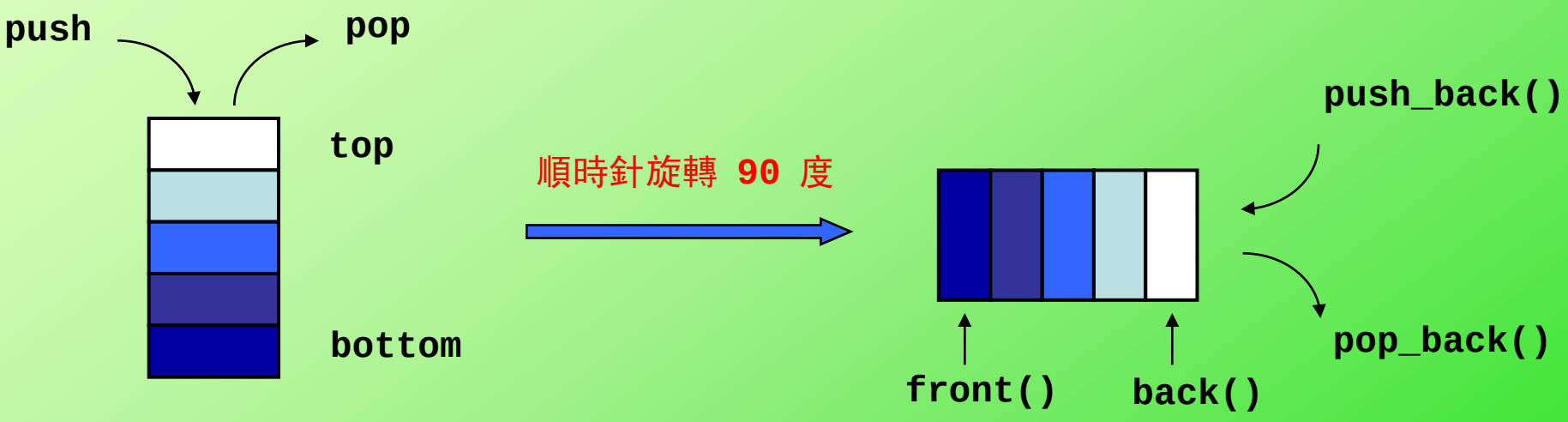
```
// 逆轉排序條件，改為由大排到小，相當於 greater_equal<int>()
// a = 21 12 10 9 9 8 7 5 4 3
sort( a , a+10 , not2(less<int>()) ) ;
```

negator

範例：堆疊類別

堆疊儲存機制

向量陣列



- 堆疊儲存機制可以直接利用向量陣列來模擬

程式

輸出

範例：向量運算

- 數學向量：直接利用向量陣列模擬

$\vec{a}$ = (2 3 4 1)

$\vec{b}$ = (1 2 0 3)

$\vec{a}$ + $\vec{b}$ = (2 3 4 1) + (1 2 0 3) = (3 5 4 4)

$\vec{a}$ - $\vec{b}$ = (2 3 4 1) - (1 2 0 3) = (1 1 4 -2)

$\vec{a}$ * $\vec{b}$ = 2*1 + 3*2 + 4*0 + 1*3 = 11

程式

輸出

範例：進階排序(一)

- 在資料庫(database)中，通常是以記錄(record)為資料最小單位，一筆資料通常包含若干個別細目(field)

```
// 學生資料庫
typedef  string    NAME ;
typedef  unsigned  AGE ;
enum  GENDER { female , male } ;

class  Student {
    private :
        NAME   name ;   // 姓名
        AGE    age ;    // 年齡
        GENGER gender ; // 性別
    public :
        ...
        NAME   get_name() const { return name ; }
        AGE    get_age() const { return age ; }
        GENDER get_gender() const { return gender ; }
} ;

// 每個物件儲存一筆記錄   整個資料庫可儲存在傳統陣列或向量陣列
Student            students[100] ;
vector<Student>    student_list ;
```

細目

範例：進階排序(二)

- 簡單排序

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

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

const int NO = 100 ;
Student  foo[NO] = { .... } ;    // 設定資料庫
```

```
// 依學生姓名由小到大依次排序
sort( foo , foo+NO , by_name< less<NAME> >() ) ;
```

```
// 依學生年齡大小由大到小依次排序
sort( foo , foo+NO , by_age< greater<AGE> >() ) ;
```

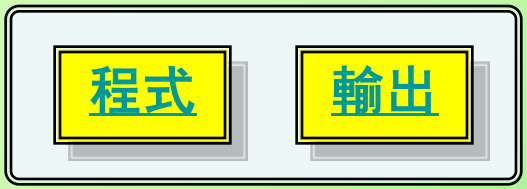
範例：進階排序(三)

- 進階排序

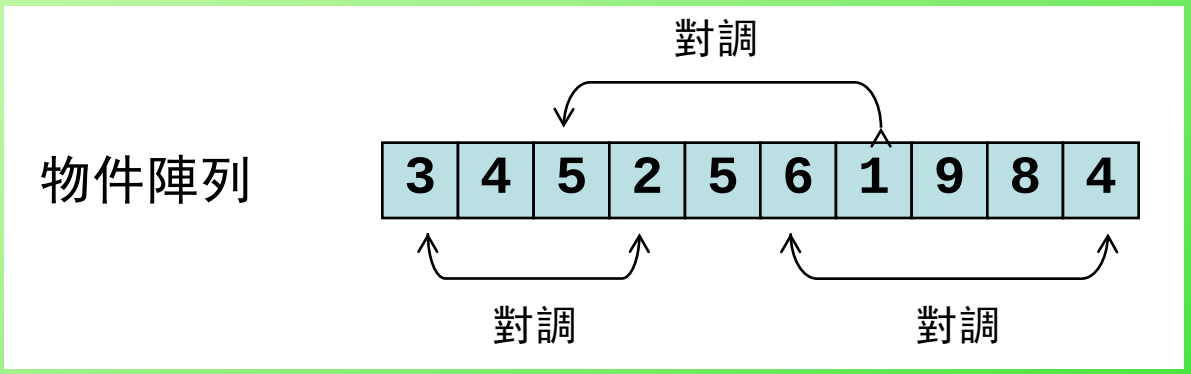
```
// 自定排序規則：先比較性別  再比較年齡  最後比較姓名
template <class Cmp_g , class Cmp_a , class Cmp_n>
struct  my_rule {
    bool  operator() ( const Student& a , const Student& b ) const {
        if ( a.get_gender() == b.get_gender() ) {
            if ( a.get_age() == b.get_age() )
                return  Cmp_n()( a.get_name() , b.get_name() ) ;
            else
                return  Cmp_a()( a.get_age() , b.get_age() ) ;
        } else
            return  Cmp_g()( a.get_gender() , b.get_gender() ) ;
    }
} ;

const int NO = 100 ;
Student  foo[NO] = { .... } ;    // 設定資料庫

// 依 (1) 性別[小->大] (2) 年齡[大->小] (3) 姓名[小->大] 規則排序
sort( foo , foo+NO ,
      my_rule< less<GENDER> , greater<AGE> , less<NAME> >() ) ;
```

範例：進階排序(四)

- 進階排序：

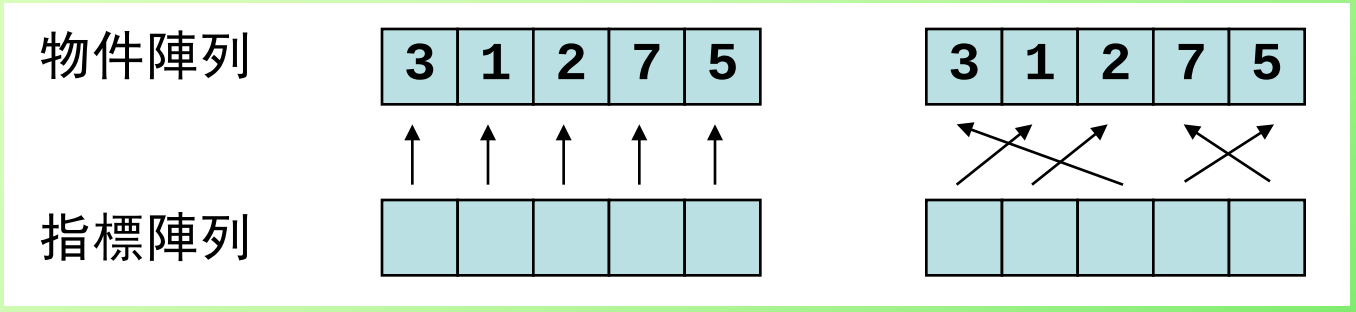


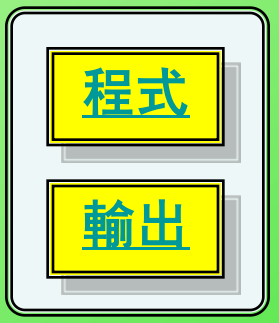
- 傳統排序在排序過程中會不斷的對調資料，因此須要耗費相當多的時間重複地複製資料



範例：進階排序(五)

- 增加排序效率：使用指標陣列輔助





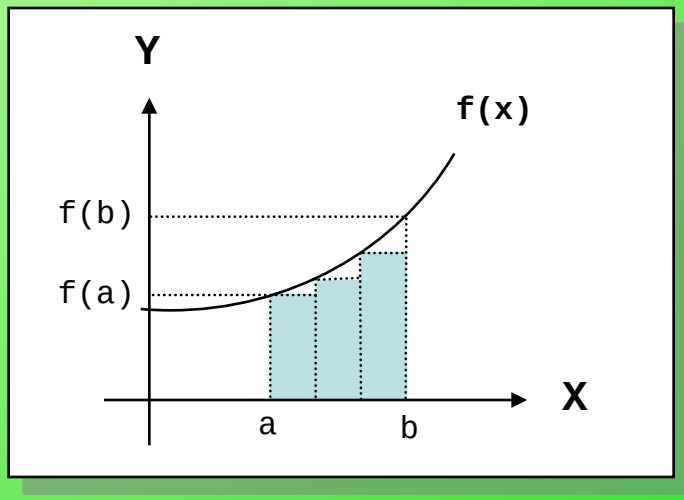

```
template <class Compare>
struct  by_age {
   bool  operator() ( const Student* a , const Student* b ) const {
      return Compare()( a->get_age() , b->get_age() ) ;
   }
} ;
...
vector<Student>   foo(NO) ;     // 設定資料庫
vector<Student*>  ptrs(NO) ;

// ptrs 儲存位址
for ( i = 0 ; i < NO ; ++i ) ptrs[i] = &foo[i] ;

// 依 年齡[大->小] 規則排序
sort( ptrs.begin() , ptrs.end() , by_age< greater<AGE> >() ) ;
```

範例：定積分的數值計算(一)

- 黎曼定積分

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

- n 個矩形面積近似定積分



$$n = 3 \qquad \Delta x = \frac{b-a}{n}$$

$$\int_a^b f(x)dx \approx \Delta x \left(f(a) + f(a+\Delta x) + f(a+2\Delta x) \right)$$

範例：定積分的數值計算(二)

- 積分泛型演算函式

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

template <class FN>
double  integral( FN fn , double a , double b , int n = 1000 ) {
    double  area = 0. , dx = (b-a)/n ;
    for ( int i = 0 ; i < n ; ++i ) area += fn(a+dx*i) ;
    return  dx*area ;
}

...
```

```
// 計算平方函數在 [2,10] 的定積分 ( n 為 1000 等份 )
cout << integral( Square() , 2 , 10 ) << endl ;
```

```
// 計算平方函數在 [2,10] 的定積分 ( n 為 500 等份 )
cout << integral( Square() , 2 , 10 , 500 ) << endl ;
```

範例：定積分的數值計算(三)

- 進階積分泛型演算函式

```
template <class T>
double  Cubic( T x ) { return x * x * x ; }

template <class FN1 , class FN2>
double  integral( FN1 fn , double a , double b ,
                  const FN2& fn2 , int n = 1000 ) {
    double  tmp , area = 0. , dx = (b-a)/n ;
    for ( int i = 0 ; i < n ; ++i ) {
        tmp = fn(a+dx*i) ;
        if ( fn2(tmp) ) area += tmp ;
    }
    return  dx*area ;
}

...
```

程式

輸出

```
// 計算立方函數在 [0,1] 且函數值須小於 0.25 的定積分 ( n 為 1000 等份 )
cout << integral( Cubic<double> , 0 , 1 ,
                  bind2nd(less<double>(),0.25) ) << endl ;
```