12-迭代器(Iterator)

12-1 為什麼需要 Iterator？

在學 STL 之前，我們通常用「下標」存取陣列：

```
int a[] = {3, 1, 4, 1, 5};
for (int i = 0; i < 5; i++)
{
    cout << a[i] << " ";
}
```

容器種類很多（vector、list、set、map……），它們的內部結構不同。傳統的作法是針對不同的容器撰寫其存取元素的成員函數。但是這樣一來，每種容器都有自己的介面，對於寫程式的人來說是個很大的困擾。

Iterator（迭代器）就是一個「統一的指標」，讓你**用相同的寫法走訪不同的容器**。

12-2 Iterator 的基本使用

一、取得 Iterator

函式	說明
`v.begin()`	指向第一個元素
`v.end()`	指向最後一個元素的下一位（不可解參考！）
`v.rbegin()`	反向，指向最後一個元素
`v.rend()`	反向，指向第一個元素的前一位

ℹ️ v.end() 不是最後一個元素，是「越界哨兵(Sentry)」，只能用來判斷是否走完，不能 *v.end()！

```
vector<int> v = {10, 20, 30, 40, 50};

vector<int>::iterator it;   // 宣告 iterator
it = v.begin();             // 指向第一個元素

cout << *it << "\n";        // 取值：10
++it;                       // 向前移一位
cout << *it << "\n";        // 20
it += 2;                    // 向前移兩位（vector iterator 支援）
cout << *it << "\n";        // 40
```

二、使用 Iterator 走訪 vector

方法一：傳統 iterator 寫法

缺點是 iterator 的宣告很長，撰寫或閱讀都會很困擾。

```
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    cout << *it << " ";
}
// 輸出：10 20 30 40 50
```

方法二：auto（C++11 起推薦）

自 C++11 起，auto 可以幫我們自動推導變數的型別。以下面程式碼為例，若 v 的型別是 `vector<int>`，則由 `v.begin()` 可以推導出 it 的型別為 `vector<int>::iterator`，這讓我們輕鬆不少。

```
for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it << " ";
}
```

方法三：範圍式 for（底層也是 iterator）

auto 和 範圍式for(range-based for loop) 在競程中都很常使用，因為非常簡潔方便。

```
for (int x : v) {
    cout << x << " ";
}
```

ℹ️ 學 iterator 時建議先練習方法一，理解底層原理

三、反向走訪

vector 還有一組 `r`開頭的 iterator，`rbegin()`、`rend()`。可以用來反向走訪。

```
vector<int> v = {10, 20, 30, 40, 50};
for (auto it = v.rbegin(); it != v.rend(); ++it) {
    cout << *it << " ";
}
// 輸出：50 40 30 20 10
```

四、Iterator 的算術運算

由於 `vector` 是具備隨機存取特性的容器，其 iterator 屬於隨機存取迭代器（Random Access Iterator），支援加減法運算。

```
vector<int> v = {10, 20, 30, 40, 50};
auto it = v.begin();

auto it2 = it + 3;          // 指向第 4 個元素
cout << *it2 << "\n";       // 40

cout << it2 - it << "\n";   // 兩個 iterator 的距離：3
```

⚠ 並非所有容器都支援 +、-。像 list 的 iterator 就只能 ++、--。這是 vector 的優勢之一。

五、修改元素

Iterator 不只能讀，也能寫：

```
vector<int> v = {1, 2, 3, 4, 5};
for (auto it = v.begin(); it != v.end(); ++it) {
    *it *= 2;   // 每個元素乘以 2
}
// v 變成 {2, 4, 6, 8, 10}
```

容器的 iterator 通常還會有一組配套的 const iterator，用來確保我們在走訪的過程中只能讀取，無法修改容器內的元素。

與 `begin()`、`end()` 配套的是 `cbegin()`、`cend()`。

與 `rbegin()`、`rend()` 配套的是 `crbegin()`、`crend()`。

若程式碼嘗試使用 const iterator 來修改容器內的元素，會被編譯器擋下來。

```
for (auto it = v.cbegin(); it != v.cend(); ++it) {
    // *it = 0;  // 編譯錯誤！
    cout << *it << " ";
}
```

六、插入與刪除

vector 的 insert 和 erase 都使用 iterator 指定位置：

```
vector<int> v = {1, 2, 4, 5};

// 在 index 2 插入 3
v.insert(v.begin() + 2, 3);
// v = {1, 2, 3, 4, 5}

// 刪除 index 1 的元素
v.erase(v.begin() + 1);
// v = {1, 3, 4, 5}

// 刪除 index 1 到 3（不含 3）
v.erase(v.begin() + 1, v.begin() + 3);
```

```
// v = {1, 5}
```

思考一個問題：如果在 erase 之前就有一些 iterator 指向這個 vector 裡的某些元素。那麼在 erase 之後，這些 iterator 還有用嗎？

12-3 Iterator 與 STL 演算法

和容器一樣，STL 也提供了很多通用演算法，適用於不同的資料型別和容器。

在使用 STL 的演算法時，除了要記得引入 `<algorithm>` 檔頭檔外，通常也需要提供一對 iterator，用來表示演算法作用的範圍。

用來表示範圍的 iterator，是左閉右開區間 `[first, last)`

一、常用的演算法

1. 排序 sort

排序所有元素

```
#include <algorithm>
vector<int> v = {5, 3, 1, 4, 2};

sort(v.begin(), v.end());           // 升冪
// v = {1, 2, 3, 4, 5}

sort(v.begin(), v.end(), greater<int>());  // 降冪
// v = {5, 4, 3, 2, 1}
```

只排序部分元素

```
vector<int> v = {5, 3, 1, 4, 2};
sort(v.begin() + 1, v.begin() + 4);  // 只排 index 1~3
// v = {5, 1, 3, 4, 2}
```

2. 線性搜尋 find

對於容器內沒排序的資料，可以使用線性搜尋(就是循序搜尋)，在 `[first, last)` 中尋找指定的元素值。

若找到了，將回傳指向第一個與目標值相同元素的 iterator；若沒找到會回傳指向 `last` 的 iterator。

```
vector<int> v = {10, 20, 30, 40, 50};
auto it = find(v.begin(), v.end(), 30);

if (it != v.end()) {
    cout << "找到！位於 index " << (it - v.begin()) << endl;  // 2
} else {
    cout << "找不到" << endl;
}
```

3. 二分搜尋 lower_bound, upper_bound

若 vector 裡的元素有排序過，可以使用二分搜尋這個高效率的搜尋演算法。

`lower_bound(first, last, target)` 會回傳第一個>=target元素的位置，若找不到則回傳 last。

`upper_bound(first, last, target)` 會回傳第一個>target元素的位置，若找不到則回傳 last。

```
vector<int> v = {1, 3, 3, 5, 7, 9};

// lower_bound：第一個 >= target 的位置
auto lo = lower_bound(v.begin(), v.end(), 3);
// 1, 3, 3, 5, 7, 9
//    ^
//    |
//    lo
cout << (lo - v.begin()) << endl;  // 1 (index 1)
```

```
// upper_bound：第一個 > target 的位置
auto hi = upper_bound(v.begin(), v.end(), 3);
// 1, 3, 3, 5, 7, 9
//          ^
//          |
//         hi
cout << (hi - v.begin()) << endl;  // 3 (index 3)

// 值 3 出現的次數：
cout << hi - lo << endl;  // 2

// 確認是否找到指定值
lo = lower_bound(v.begin(), b.end(), 4);
// lo = lower_bound(v.begin(), b.end(), 5);
if(lo!=v.end() && *lo==4)
{
  cout << "找到！位於 index " << (lo-v.begin()) << endl;
}
else
{
  cout << "找不到" << endl;
}
```

4. 最大/最小值 max_element/min_element

```
vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6};

auto mx = max_element(v.begin(), v.end());
cout << *mx << "\n";                    // 9
cout << (mx - v.begin()) << "\n";      // index 5

auto mn = min_element(v.begin(), v.end());
cout << *mn << "\n";                    // 1
```

5. 反轉 reverse

```
vector<int> v = {1, 2, 3, 4, 5};
reverse(v.begin(), v.end());
// v = {5, 4, 3, 2, 1}

// 反轉部分：
reverse(v.begin() + 1, v.begin() + 4);
// v = {1, 4, 3, 2, 5}
```

6. 計數 count

```
vector<int> v = {1, 2, 2, 3, 2, 4};
cout << count(v.begin(), v.end(), 2) << endl;  // 3
```

7. 加總 accumulate

需 `include <numeric>`。

```
#include <numeric>
vector<int> v = {1, 2, 3, 4, 5};
int sum = accumulate(v.begin(), v.end(), 0);  // 初始值 0
cout << sum << endl;  // 15
```

8. 去除相鄰重複 unique

特別注意「**相鄰**」這兩個字，unique 會移除相鄰出現的元素。 它的作法是略過相鄰相同的元素，把後續相異元素往前搬，覆蓋掉相鄰相同的元素，所以最後會回傳你新的結尾 iterator。要記得把「尾巴」刪掉。

```
vector<int> v = {1, 3, 2, 2, 3, 1};
```

```
auto newEnd = unique(v.begin(), v.end());  // 去重，回傳新的 end
// 1, 3, 2, 2, 3, 1
//   uniquer 之後
// 1, 3, 2, 3, 1, 1
//                ^
//                |
//              newEnd
v.erase(newEnd, v.end());       // 刪掉尾巴
// v = {1, 3, 2, 3, 1}
```

為了移除容器內的相同元素，通常我們會先 sort 再 unique，最後 erase 掉多餘元素。

```
vector<int> v = {1, 3, 2, 2, 3, 1};
sort(v.begin(), v.end());       // {1, 1, 2, 2, 3, 3}
auto newEnd = unique(v.begin(), v.end());  // 去重，回傳新的 end
v.erase(newEnd, v.end());       // 刪掉尾巴
// v = {1, 2, 3}
```

二、Compare

在使用 sort 排序時，你可能注意到了，除了用 `[first, end)` 標定排序範圍外，在尾巴我們還有提供一個 `greater<int>()` 或 `less<int>()` 來決定是由大到小亦或由小到大排序。

由於 STL 的 sort 是藉由比較兩個元素的大小來決定誰要排在前面，所以需要我們告訴它要怎麼決定誰排在前面。

1. Compare function

最簡單的方式，就是提供一個函數讓 sort 叫用，它會呼叫這個函數並傳入兩個值 a, b，如果 a 應該排在 b 前面，我們就回傳 `true`，否則回傳 `false`。

以下是個實例

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// 小的排前面
bool comp1(const int &a, const int &b)
{
    return a<b;
}

// 大的排前面
bool comp2(const int &a, const int &b)
{
    return a>b;
}

int main()
{
    vector<int> v = {1, 3, 5, 6, 4, 2};

    sort(v.begin(), v.end(), comp1);

    for(int i: v) {
        cout << i << " ";
    }
    cout << endl;
    // 1 2 3 4 5 6

    sort(v.begin(), v.end(), comp2);

    for(int i: v) {
        cout << i << " ";
```

```
    }
    cout << endl;
 // 6 5 4 3 2 1

    return 0;
}
```

2. lambda 函數

在 C++11 之後，在這種場合我們可以直接在 sort 的參數列裡建立一個 lambda funtion 讓 sort 叫用如下。

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector<int> v = {1, 3, 5, 6, 4, 2};

    // 小的排前面
    sort(v.begin(), v.end(), [] (const int &a, const int &b)->bool {
            return a<b;
        }
    );

    for(int i: v) {
        cout << i << " ";
    }
    cout << endl;

    // 大的排前面
    sort(v.begin(), v.end(), [] (const int &a, const int &b)->bool {
            return a>b;
        }
    );

    for(int i: v) {
        cout << i << " ";
    }
    cout << endl;


    return 0;
}
```

lambda 函數是一個匿名的函數，可以直接寫在程式碼裡。

我們來拆解檢視一下這個 lambda 函數。

```
[] (const int &a, const int &b)->bool {
    return a<b;
}
```

`[]` 這個我們先不看，當 lambda 函數裡需要存取外面的變數時才需要它。

`(const int &a, const int &b)->bool` 這表示該 lambda 函數有兩個參數，回傳值是 bool 型別。

`{ ... }` 大括號裡是函數的實作部分。

3. functor

前面提到的 `greater<int>()` 和 `less<int>()` 是所謂的函式物件 `function object`，也叫 `functor`。

本質上它是一個「行為表現得像一般函式的物件（Object）」，只要一個類別（class）或結構（struct）重載了「函

式呼叫運算子」 `operator()` ，由它所建立出來的物件就是一個 Functor。

例如：以下是一個用來計算兩整數相加的 functor。

```
#include <iostream>

using namespace std;

class AplusB {
public:
    int operator()(int a, int b)
    {
        return a+b;
    }
};

int main()
{
    AplusB sum;

    cout << sum(2, 3) << endl;  // 5

    return 0;
}
```

用 functor 來做排序。

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class smallFirst {
public:
    bool operator()(const int &a, const int &b)
    {
        return a<b;
    }
};

class bigFirst {
public:
    bool operator()(const int &a, const int &b)
    {
        return a>b;
    }
};

int main()
{
    vector<int> v = {1, 3, 5, 6, 4, 2};

    // 小的排前面
    sort(v.begin(), v.end(),  smallFirst());

    for(int i: v) {
        cout << i << " ";
    }
    cout << endl;

    // 大的排前面
    sort(v.begin(), v.end(),  bigFirst());

    for(int i: v) {
        cout << i << " ";
    }
```

```
    cout << endl;


    return 0;
}
```

如果希望它可以用在多種型別的排序，可以結合 template 來使用。

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template<typename T>
class smallFirst {
public:
    bool operator()(const T &a, const T &b)
    {
        return a<b;
    }
};

template<typename T>
class bigFirst {
public:
    bool operator()(const T &a, const T &b)
    {
        return a>b;
    }
};

int main()
{
    vector<int> v = {1, 3, 5, 6, 4, 2};

    // 小的排前面
    sort(v.begin(), v.end(),  smallFirst<int>());

    for(int i: v) {
        cout << i << " ";
    }
    cout << endl;

    vector<double> u = {1.4, 3.5, 1.5, 6.5, 4.5, 2.5};
    // 大的排前面
    sort(u.begin(), u.end(),  bigFirst<double>());

    for(double i: u) {
        cout << i << " ";
    }
    cout << endl;


    return 0;
}
```

三、 Predicate

Predicate 和 Compare 一樣是一個用來被別人呼叫並回傳 bool 值的函數(或 functor)。只是語意上有差異而已。

例如:我們要刪除 vector 裡所有的奇數，我們可以這樣做。

1. 使用 function

```
#include <iostream>
```

```
#include <vector>
#include <algorithm>

using namespace std;

bool odd(const int &a)
{
    return (a%2==1);
}

int main()
{
    vector<int> v = {2, 3, 5, 6, 4, 1};

    auto new_end = remove_if(v.begin(), v.end(), odd);
    v.erase(new_end, v.end());  // 記得把尾巴刪掉

    for(int i: v) {
        cout << i << " ";
    }
    cout << endl;

    return 0;
}
```

2. 使用 lambda function

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector<int> v = {2, 3, 5, 6, 4, 1};

    auto new_end = remove_if(v.begin(), v.end(),
                             [] (const int &a)->bool
                             {
                                 return (a%2==1);
                             }
                         );
    v.erase(new_end, v.end());  // 記得把尾巴刪掉

    for(int i: v) {
        cout << i << " ";
    }
    cout << endl;

    return 0;
}
```

3. 使用 functor

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class odd {
public:
    bool operator()(const int &a) {
        return (a%2==1);
    }
};
```

```
int main()
{
    vector<int> v = {2, 3, 5, 6, 4, 1};

    auto new_end = remove_if(v.begin(), v.end(), odd());
    v.erase(new_end, v.end());  // 記得把尾巴刪掉

    for(int i: v) {
        cout << i << " ";
    }
    cout << endl;

    return 0;
}
```

四、練習題

使用自訂排序規則來解以下幾題有關排序的問題

- 打獵分配問題
- 物品堆疊(Stacking)
- 吸塵器機器人