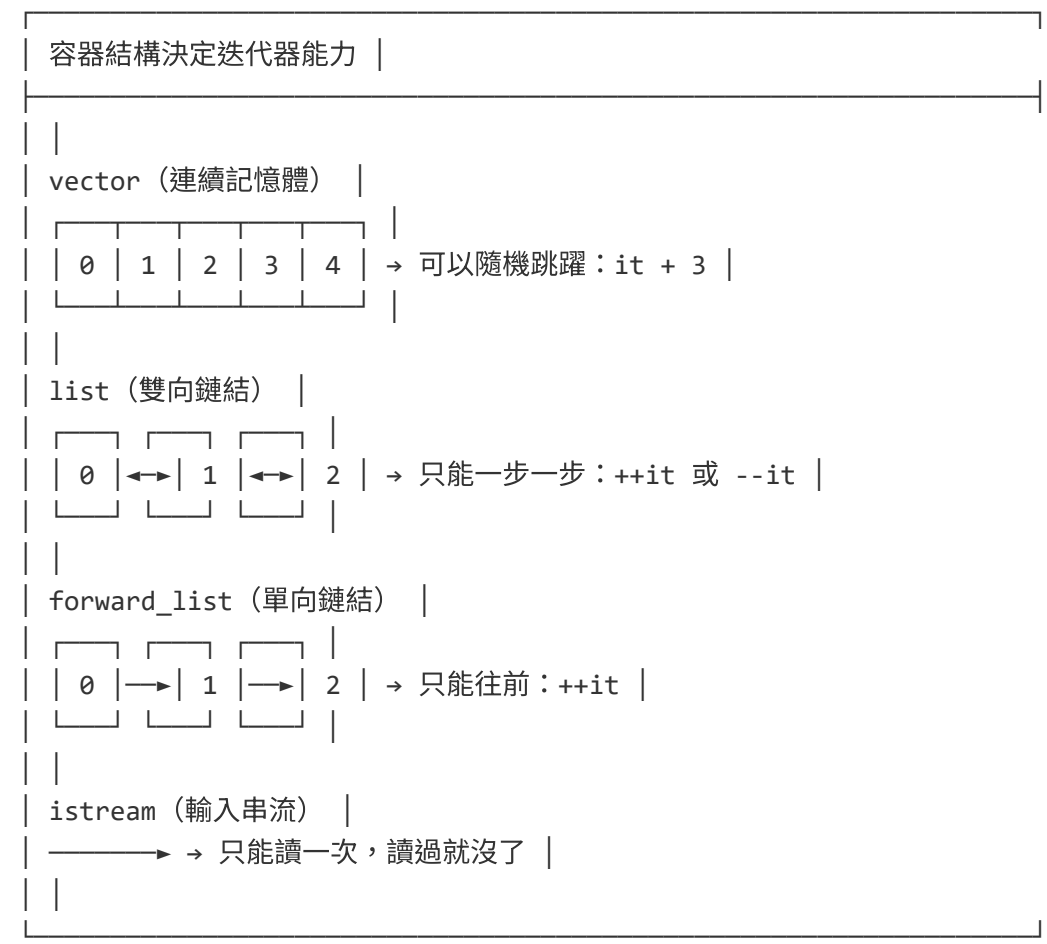
第五課：迭代器的五種分類

第五課：迭代器的五種分類

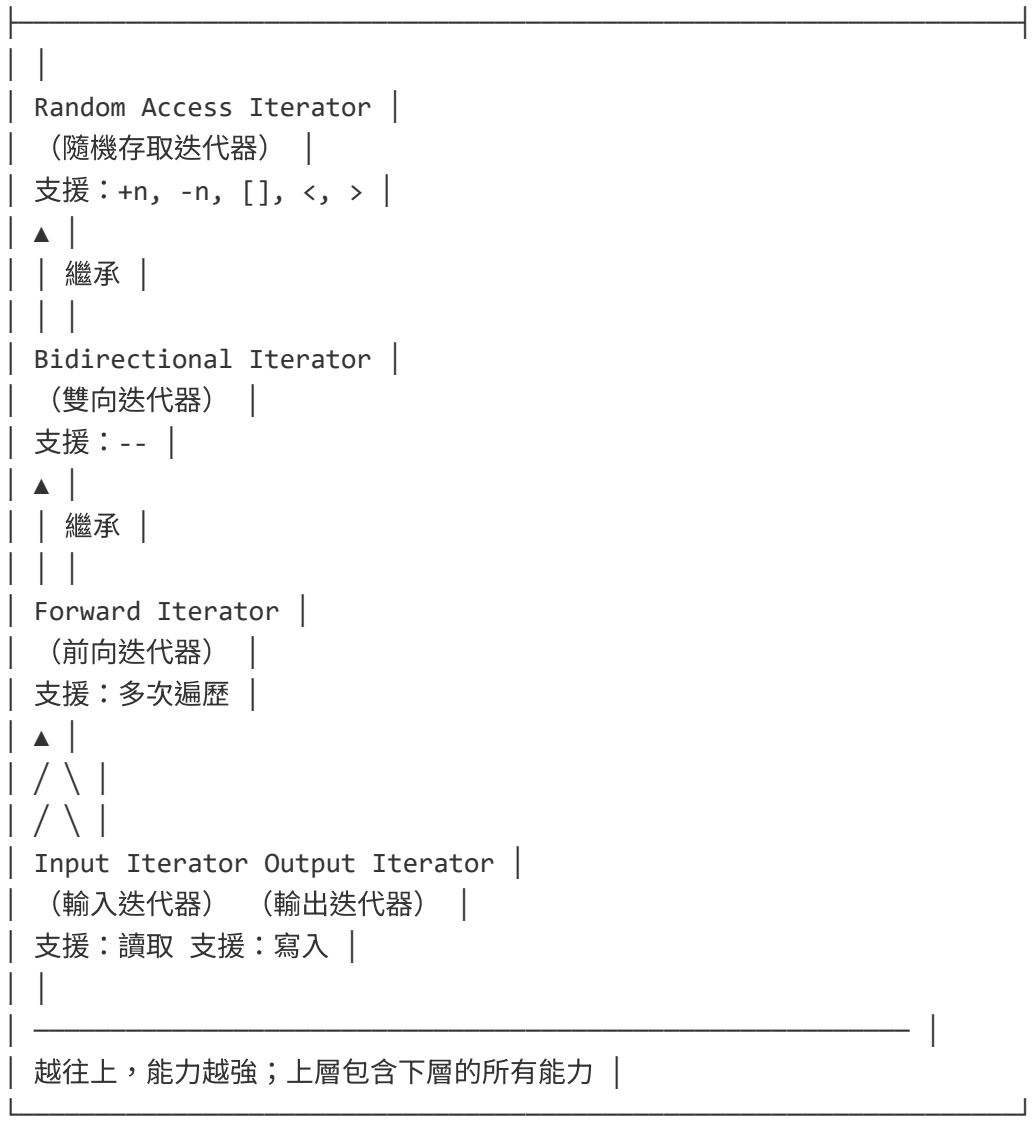
5.1 為什麼要分類？

不同的容器有不同的內部結構，能支援的迭代器操作也不同：



STL 把迭代器依「能力」分成五類，形成一個階層結構。

5.2 五種迭代器的階層



5.3 Input Iterator（輸入迭代器）

最基本的迭代器，只能讀取，只能往前，且只能遍歷一次。

特性

操作	支援	說明
`*it`	✓	讀取（但不能寫入）
`++it / it++`	✓	前進
`it1 == it2`	✓	相等比較
`it1 != it2`	✓	不等比較
`--it`	✗	不能後退
`it + n`	✗	不能跳躍
多次遍歷	✗	遍歷過的元素不能再訪問

典型代表：istream_iterator

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
int main() {
std::cout << "=== Input Iterator 示範 ===" << std::endl;
std::cout << "請輸入一些整數（輸入非數字結束）：" << std::endl;
// istream_iterator 是 Input Iterator
std::istream_iterator<int> input_begin(std::cin);
std::istream_iterator<int> input_end; // 預設建構 = 結束標記
// 讀取所有輸入到 vector
std::vector<int> numbers(input_begin, input_end);
std::cout << "你輸入了 " << numbers.size() << " 個數字：";
for (int n : numbers) {
std::cout << n << " ";
}
std::cout << std::endl;
return 0;
}
```

執行範例：

```
=== Input Iterator 示範 ===
```

```
請輸入一些整數（輸入非數字結束）：
10 20 30 40 quit
你輸入了 4 個數字：10 20 30 40
```

Input Iterator 的限制

```
#include <iostream>
#include <sstream>
#include <iterator>
int main() {
std::istringstream iss("10 20 30");
std::istream_iterator<int> it(iss);
// 可以讀取
std::cout << "第一次讀取: " << *it << std::endl;
// 可以前進
++it;
std::cout << "前進後讀取: " << *it << std::endl;
// 關鍵限制：不能回頭！
// --it; // 編譯錯誤：Input Iterator 不支援 --
// 也不能跳躍
// it + 1; // 編譯錯誤：Input Iterator 不支援 +
// 一旦遍歷過，就不能重新遍歷同一個串流
//  （除非重新建立串流）
return 0;
}
```

輸出：

```
第一次讀取: 10
前進後讀取: 20
```

5.4 Output Iterator（輸出迭代器）

只能寫入，不能讀取，只能往前，且只能遍歷一次。

特性

操作	支援	說明
`*it = value`	✓	寫入
`++it / it++`	✓	前進
`*it`（讀取）	✗	不能讀取
`--it`	✗	不能後退
`it + n`	✗	不能跳躍

典型代表：ostream_iterator、back_inserter

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
int main() {
std::vector<int> numbers = {10, 20, 30, 40, 50};
// ostream_iterator 是 Output Iterator
std::cout << "=== ostream_iterator ===" << std::endl;
std::cout << "輸出: ";
std::copy(numbers.begin(), numbers.end(),
std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;
// back_inserter 也是 Output Iterator
std::cout << "\n=== back_inserter ===" << std::endl;
std::vector<int> source = {1, 2, 3};
std::vector<int> dest;
// back_inserter 讓你可以「寫入」到 vector 尾端
std::copy(source.begin(), source.end(),
std::back_inserter(dest));
std::cout << "dest: ";
for (int n : dest) std::cout << n << " ";
std::cout << std::endl;
return 0;
}
```

輸出：

```
=== ostream_iterator ===
輸出: 10 20 30 40 50
=== back_inserter ===
dest: 1 2 3
```

Output Iterator 的限制

```
#include <iostream>
#include <iterator>
int main() {
std::ostream_iterator<int> out(std::cout, " ");
// 可以寫入
*out = 10; // 輸出 "10 "
++out;
*out = 20; // 輸出 "20 "
++out;
*out = 30; // 輸出 "30 "
std::cout << std::endl;
// 但不能讀取！
// int x = *out; // 這不會如你預期的工作
// 也不能回頭或跳躍
// --out; // 不支援
// out + 1; // 不支援
return 0;
}
```

輸出：

```
10 20 30
```

5.5 Forward Iterator（前向迭代器）

可以多次遍歷，但仍只能往前。

特性

操作	支援	說明
`*it` （讀寫）	✓	可讀可寫
`++it / it++`	✓	前進
`it1 == it2`	✓	相等比較
多次遍歷	✓	可以保存迭代器，之後再用
`--it`	✗	不能後退
`it + n`	✗	不能跳躍

典型代表：forward_list、unordered_set

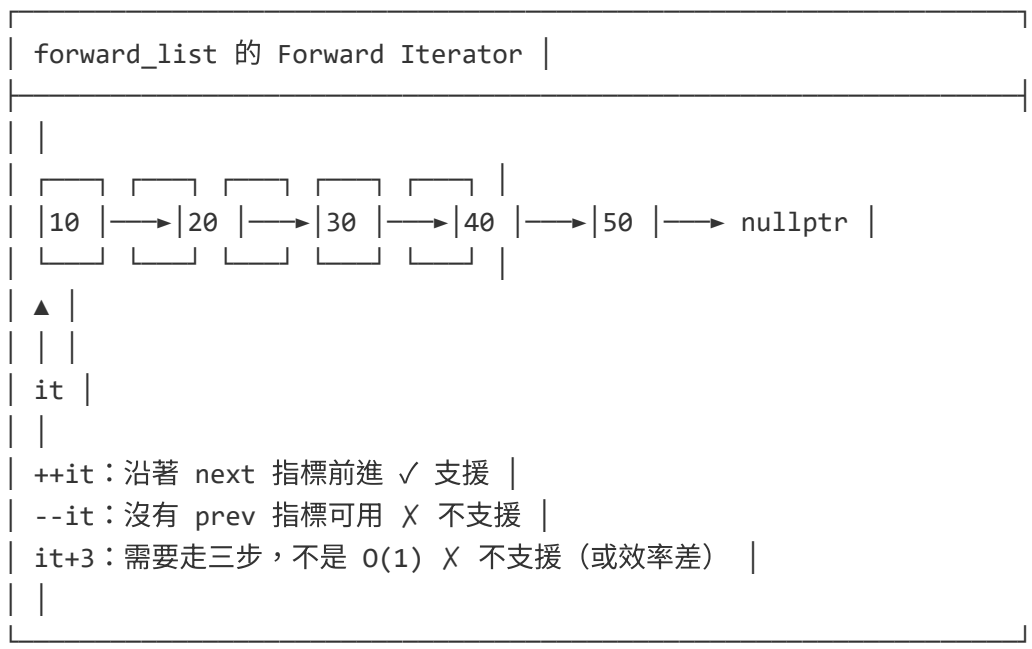
```
#include <iostream>
#include <forward_list>
#include <unordered_set>
int main() {
// forward_list 的迭代器是 Forward Iterator
std::cout << "=== forward_list ===" << std::endl;
std::forward_list<int> flist = {10, 20, 30, 40, 50};
// 保存一個迭代器
auto saved = flist.begin();
++saved; // 指向 20
// 先遍歷一次
std::cout << "第一次遍歷: ";
for (auto it = flist.begin(); it != flist.end(); ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
// 可以再遍歷一次（Input Iterator 不行）
std::cout << "第二次遍歷: ";
for (auto it = flist.begin(); it != flist.end(); ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
// 之前保存的迭代器還有效
std::cout << "保存的迭代器指向: " << *saved << std::endl;
// 但不能後退
```

```
// --saved; // 編譯錯誤！
return 0;
}
```

輸出：

```
=== forward_list ===
第一次遍歷: 10 20 30 40 50
第二次遍歷: 10 20 30 40 50
保存的迭代器指向: 20
```

Forward Iterator 與單向鏈結



5.6 Bidirectional Iterator（雙向迭代器）

可以雙向移動，但仍不能隨機跳躍。

特性

操作	支援	說明
`*it` （讀寫）	✓	可讀可寫
`++it / it++`	✓	前進
`--it / it--`	✓	後退
多次遍歷	✓	可以保存迭代器
`it + n`	✗	不能跳躍
`it1 < it2`	✗	不能比較大小（只能比較相等）

典型代表：list、set、map

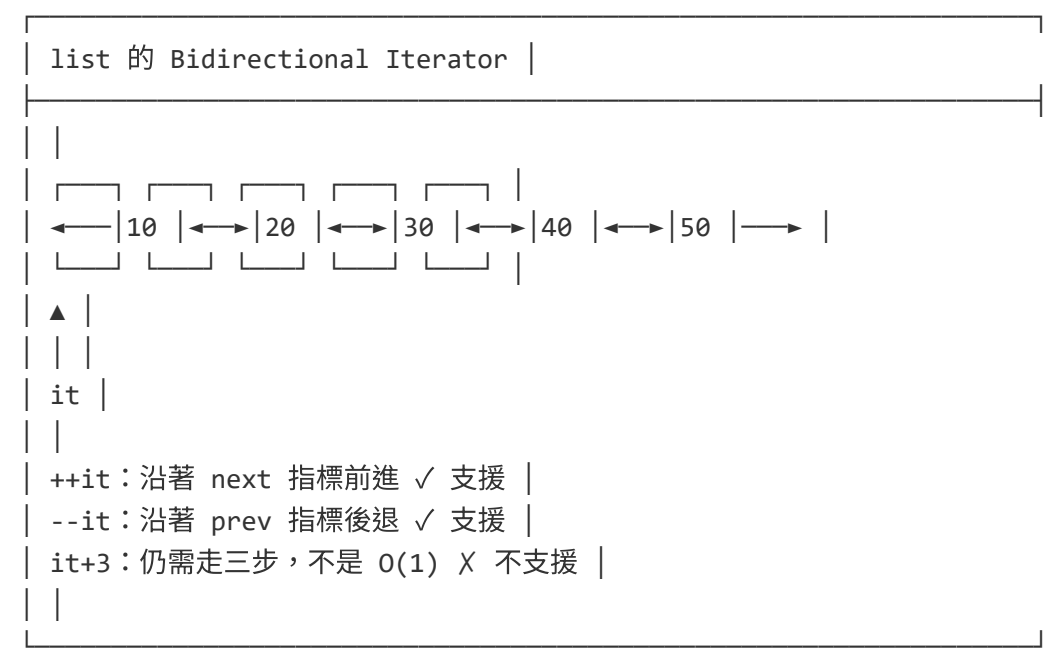
```
#include <iostream>
#include <list>
#include <set>
int main() {
// list 的迭代器是 Bidirectional Iterator
std::cout << "=== list ===" << std::endl;
std::list<int> lst = {10, 20, 30, 40, 50};
auto it = lst.begin();
std::cout << "起始: " << *it << std::endl;
++it;
std::cout << "++it: " << *it << std::endl;
++it;
std::cout << "++it: " << *it << std::endl;
--it; // Bidirectional 可以後退！
std::cout << "--it: " << *it << std::endl;
// 但不能跳躍
// it + 2; // 編譯錯誤！
// set 也是 Bidirectional Iterator
std::cout << "\n=== set ===" << std::endl;
std::set<int> s = {50, 10, 30, 20, 40}; // 自動排序
std::cout << "正向: ";
for (auto it = s.begin(); it != s.end(); ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
```

```
std::cout << "反向: ";
for (auto it = s.rbegin(); it != s.rend(); ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
return 0;
}
```

輸出：

```
=== list ===
起始: 10
++it: 20
++it: 30
--it: 20
=== set ===
正向: 10 20 30 40 50
反向: 50 40 30 20 10
```

Bidirectional Iterator 與雙向鏈結



5.7 Random Access Iterator（隨機存取迭代器）

功能最強大的迭代器，支援所有操作。

特性

操作	支援	說明
`*it` （讀寫）	✓	可讀可寫
`++it / --it`	✓	前進/後退
`it + n / it - n`	✓	跳躍
`it += n / it -= n`	✓	跳躍賦值
`it[n]`	✓	下標存取
`it1 - it2`	✓	計算距離
`it1 < it2`	✓	比較大小

典型代表：vector、deque、array、原生指標

```
#include <iostream>
#include <vector>
#include <deque>
int main() {
std::vector<int> vec = {10, 20, 30, 40, 50};
std::cout << "=== Random Access Iterator 完整功能 ===" << std::endl;
auto it = vec.begin();
// 基本操作
std::cout << "*it = " << *it << std::endl;
// 前進後退
++it;
std::cout << "++it: *it = " << *it << std::endl;
--it;
std::cout << "--it: *it = " << *it << std::endl;
// 跳躍！
it = it + 3;
std::cout << "it + 3: *it = " << *it << std::endl;
it = it - 2;
std::cout << "it - 2: *it = " << *it << std::endl;
// 下標存取
it = vec.begin();
```

```
std::cout << "it[2] = " << it[2] << std::endl;
std::cout << "it[4] = " << it[4] << std::endl;
// 計算距離
auto begin = vec.begin();
auto end = vec.end();
std::cout << "end - begin = " << (end - begin) << std::endl;
// 比較大小
auto mid = vec.begin() + 2;
std::cout << "begin < mid? " << (begin < mid ? "是" : "否") << std::endl;
std::cout << "end > mid? " << (end > mid ? "是" : "否") << std::endl;
return 0;
}
```

輸出：

```
=== Random Access Iterator 完整功能 ===
*it = 10
++it: *it = 20
--it: *it = 10
it + 3: *it = 40
it - 2: *it = 20
it[2] = 30
it[4] = 50
end - begin = 5
begin < mid? 是
end > mid? 是
```

5.8 五種迭代器的能力比較表

```
┌──────────────────────────────────────────────────────────────────────┐
│ 迭代器能力比較表 │
├──────────────┬──────────┬──────────┬──────────┬──────────────┬──────────────┤
│ 操作 │ Input │ Output │ Forward │ Bidirectional│ Random Access │
├──────────────┼──────────┼──────────┼──────────┼──────────────┼──────────────┤
│ 讀取 *it │ ✓ │ ✗ │ ✓ │ ✓ │ ✓ │
├──────────────┼──────────┼──────────┼──────────┼──────────────┼──────────────┤
│ 寫入 *it=v │ ✗ │ ✓ │ ✓ │ ✓ │ ✓ │
├──────────────┼──────────┼──────────┼──────────┼──────────────┼──────────────┤
│ ++it │ ✓ │ ✓ │ ✓ │ ✓ │ ✓ │
```

--it	✗	✗	✗	✓	✓
it + n	✗	✗	✗	✗	✓
it - n	✗	✗	✗	✗	✓
it[n]	✗	✗	✗	✗	✓
it1 - it2	✗	✗	✗	✗	✓
it1 < it2	✗	✗	✗	✗	✓
多次遍歷	✗	✗	✓	✓	✓
代表容器	istream iterator	ostream iterator	forward_ list, map	list, set, map	vector, deque, array, 指標

5.9 演算法對迭代器的要求

STL 演算法會根據需要的功能，要求特定類別的迭代器：

```
#include <iostream>
#include <vector>
#include <list>
#include <forward_list>
#include <algorithm>
int main() {
std::vector<int> vec = {5, 2, 8, 1, 9};
std::list<int> lst = {5, 2, 8, 1, 9};
std::forward_list<int> flst = {5, 2, 8, 1, 9};
// find：只需要 Input Iterator
// 所有容器都能用
std::cout << "=== find（需要 Input Iterator）===" << std::endl;
auto v_it = std::find(vec.begin(), vec.end(), 8);
auto l_it = std::find(lst.begin(), lst.end(), 8);
auto f_it = std::find(flst.begin(), flst.end(), 8);
```

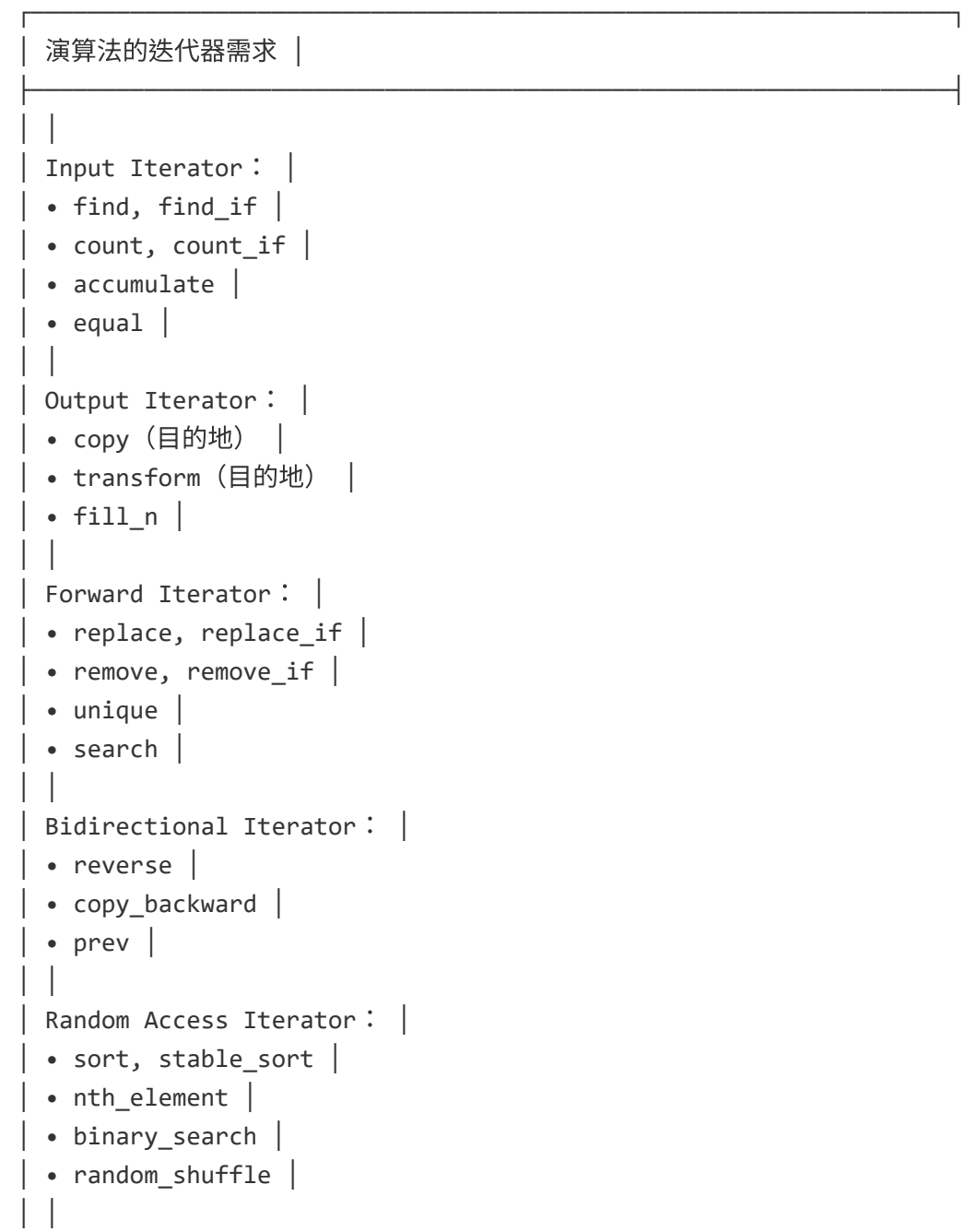
```
std::cout << "vector: " << (v_it != vec.end() ? "找到" : "沒找到") <<
std::endl;
std::cout << "list: " << (l_it != lst.end() ? "找到" : "沒找到") << std::endl;
std::cout << "forward_list: " << (f_it != flst.end() ? "找到" : "沒找到") <<
std::endl;
// reverse：需要 Bidirectional Iterator
// forward_list 不能用！
std::cout << "\n=== reverse（需要 Bidirectional Iterator）===" << std::endl;
std::reverse(vec.begin(), vec.end());
std::reverse(lst.begin(), lst.end());
// std::reverse(flst.begin(), flst.end()); // 編譯錯誤！
std::cout << "vector 反轉: ";
for (int n : vec) std::cout << n << " ";
std::cout << std::endl;
std::cout << "list 反轉: ";
for (int n : lst) std::cout << n << " ";
std::cout << std::endl;
// sort：需要 Random Access Iterator
// 只有 vector 能用 std::sort！
std::cout << "\n=== sort（需要 Random Access Iterator）===" << std::endl;
std::sort(vec.begin(), vec.end());
// std::sort(lst.begin(), lst.end()); // 編譯錯誤！
// std::sort(flst.begin(), flst.end()); // 編譯錯誤！
// list 和 forward_list 有自己的 sort 成員函數
lst.sort();
flst.sort();
std::cout << "vector 排序: ";
for (int n : vec) std::cout << n << " ";
std::cout << std::endl;
std::cout << "list 排序: ";
for (int n : lst) std::cout << n << " ";
std::cout << std::endl;
return 0;
}
```

輸出：

```
=== find（需要 Input Iterator）===
vector: 找到
list: 找到
forward_list: 找到
```

```
=== reverse（需要 Bidirectional Iterator）===
vector 反轉: 9 1 8 2 5
list 反轉: 9 1 8 2 5
=== sort（需要 Random Access Iterator）===
vector 排序: 1 2 5 8 9
list 排序: 1 2 5 8 9
```

常見演算法的迭代器需求



└──┘

5.10 std::advance 與 std::distance

這兩個工具函數可以「屏蔽」迭代器類別的差異：

```
#include <iostream>
#include <vector>
#include <list>
#include <iterator>
int main() {
// std::advance：移動迭代器
std::cout << "=== std::advance ===" << std::endl;
std::vector<int> vec = {10, 20, 30, 40, 50};
std::list<int> lst = {10, 20, 30, 40, 50};
auto vit = vec.begin();
auto lit = lst.begin();
// 對於 Random Access Iterator，advance 內部用 it + n
std::advance(vit, 3);
std::cout << "vector advance(it, 3): " << *vit << std::endl;
// 對於 Bidirectional Iterator，advance 內部用 ++it 三次
std::advance(lit, 3);
std::cout << "list advance(it, 3): " << *lit << std::endl;
// std::distance：計算距離
std::cout << "\n=== std::distance ===" << std::endl;
auto vbegin = vec.begin();
auto vend = vec.end();
std::cout << "vector distance: " << std::distance(vbegin, vend) << std::endl;
auto lbegin = lst.begin();
auto lend = lst.end();
std::cout << "list distance: " << std::distance(lbegin, lend) << std::endl;
// std::next 和 std::prev（C++11）：更方便的移動
std::cout << "\n=== std::next / std::prev ===" << std::endl;
auto it = vec.begin();
auto next_it = std::next(it, 2); // 前進 2 步，不修改 it
auto prev_it = std::prev(vec.end(), 1); // 後退 1 步
std::cout << "*it = " << *it << std::endl;
std::cout << "*next(it, 2) = " << *next_it << std::endl;
std::cout << "*prev(end, 1) = " << *prev_it << std::endl;
```

```
return 0;
}
```

輸出：

```
=== std::advance ===
vector advance(it, 3): 40
list advance(it, 3): 40
=== std::distance ===
vector distance: 5
list distance: 5
=== std::next / std::prev ===
*it = 10
*next(it, 2) = 30
*prev(end, 1) = 50
```

advance 與 distance 的效率

```
┌──────────────────────────────────────────────────────────────────┐
│ advance 與 distance 的時間複雜度 │
├──────────────────────────────────────────────────────────────────┤
│ │
│ advance(it, n) distance(it1, it2) │
│ ──────────────────────────────────────────────────────── │
│ Random Access Iterator O(1) O(1) │
│ Bidirectional Iterator O(n) O(n) │
│ Forward Iterator O(n) O(n) │
│ Input Iterator O(n) O(n) │
│ │
│ 說明： │
│ • Random Access 可以直接 it + n，所以是 O(1) │
│ • 其他迭代器必須一步一步走，所以是 O(n) │
│ • 使用 advance/distance 讓程式碼通用，效率由迭代器類別決定 │
│ │
└──────────────────────────────────────────────────────────────────┘
```

5.11 迭代器類別標籤（Iterator Tags）

STL 用「標籤」來識別迭代器類別，讓演算法可以選擇最佳實作：

```
#include <iostream>
#include <iterator>
#include <vector>
#include <list>
#include <forward_list>
// 查詢迭代器類別
template <typename Iterator>
void print_iterator_category(const std::string& name) {
using category = typename std::iterator_traits<Iterator>::iterator_category;
std::cout << name << " 的迭代器類別: ";
if constexpr (std::is_same_v<category, std::input_iterator_tag>) {
std::cout << "Input Iterator";
} else if constexpr (std::is_same_v<category, std::output_iterator_tag>) {
std::cout << "Output Iterator";
} else if constexpr (std::is_same_v<category, std::forward_iterator_tag>) {
std::cout << "Forward Iterator";
} else if constexpr (std::is_same_v<category,
std::bidirectional_iterator_tag>) {
std::cout << "Bidirectional Iterator";
} else if constexpr (std::is_same_v<category,
std::random_access_iterator_tag>) {
std::cout << "Random Access Iterator";
}
std::cout << std::endl;
}
int main() {
print_iterator_category<std::vector<int>::iterator>("vector");
print_iterator_category<std::list<int>::iterator>("list");
print_iterator_category<std::forward_list<int>::iterator>("forward_list");
print_iterator_category<int*>("原生指標");
print_iterator_category<std::istream_iterator<int>>("istream_iterator");
print_iterator_category<std::ostream_iterator<int>>("ostream_iterator");
return 0;
}
```

輸出：

```
vector 的迭代器類別: Random Access Iterator
list 的迭代器類別: Bidirectional Iterator
forward_list 的迭代器類別: Forward Iterator
```

```
istream_iterator 的迭代器類別: Input Iterator
ostream_iterator 的迭代器類別: Output Iterator
```

5.12 各容器的迭代器類別總覽

5.13 實際應用：根據迭代器類別選擇演算法

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <iterator>
// 通用的「取得第 n 個元素」函數
template <typename Container>
auto get_nth_element(Container& c, size_t n)
-> typename Container::reference
{
auto it = c.begin();
std::advance(it, n); // 自動選擇最佳方式前進
return *it;
}
// 通用的「排序」函數
template <typename Container>
void sort_container(Container& c) {
using iterator = typename Container::iterator;
using category = typename std::iterator_traits<iterator>::iterator_category;
if constexpr (std::is_same_v<category, std::random_access_iterator_tag>) {
// Random Access：用 std::sort
std::sort(c.begin(), c.end());
std::cout << "使用 std::sort" << std::endl;
} else {
// 其他：用容器自己的 sort（如果有的話）
c.sort();
std::cout << "使用容器成員函數 sort" << std::endl;
}
}
int main() {
std::cout << "=== get_nth_element ===" << std::endl;
std::vector<int> vec = {10, 20, 30, 40, 50};
std::list<int> lst = {10, 20, 30, 40, 50};
std::cout << "vector[2] = " << get_nth_element(vec, 2) << std::endl;
std::cout << "list[2] = " << get_nth_element(lst, 2) << std::endl;
std::cout << "\n=== sort_container ===" << std::endl;
std::vector<int> v = {5, 2, 8, 1, 9};
std::list<int> l = {5, 2, 8, 1, 9};
std::cout << "排序 vector: ";
sort_container(v);
```

```
for (int n : v) std::cout << n << " ";
std::cout << std::endl;
std::cout << "排序 list: ";
sort_container(l);
for (int n : l) std::cout << n << " ";
std::cout << std::endl;
return 0;
}
```

輸出：

```
=== get_nth_element ===
vector[2] = 30
list[2] = 30
=== sort_container ===
排序 vector: 使用 std::sort
1 2 5 8 9
排序 list: 使用容器成員函數 sort
1 2 5 8 9
```

5.14 本課重點整理

```
┌──────────────────────────────────────────────────────────────────────┐
│ 第五課 重點整理 │
├──────────────────────────────────────────────────────────────────────┤
│ │
│ 1. 五種迭代器類別（由弱到強） │
│ ① Input Iterator：只能讀、只能往前、單次遍歷 │
│ ② Output Iterator：只能寫、只能往前、單次遍歷 │
│ ③ Forward Iterator：可讀寫、只能往前、可多次遍歷 │
│ ④ Bidirectional Iterator：可讀寫、可雙向、可多次遍歷 │
│ ⑤ Random Access Iterator：完整功能，支援跳躍和比較 │
│ │
│ 2. 容器的迭代器類別 │
│ • Random Access：vector, deque, array, 指標 │
│ • Bidirectional：list, set, map │
│ • Forward：forward_list, unordered_set, unordered_map │
│ │
│ 3. 演算法的迭代器要求 │
```

```
│ • find：Input Iterator（最寬鬆） │
│ • reverse：Bidirectional Iterator │
│ • sort：Random Access Iterator（最嚴格） │
│ │
│ 4. 通用工具函數 │
│ • std::advance(it, n)：移動迭代器 n 步 │
│ • std::distance(it1, it2)：計算兩迭代器的距離 │
│ • std::next(it, n) / std::prev(it, n)：取得移動後的迭代器 │
│ │
│ 5. 迭代器標籤（Iterator Tags） │
│ • 用於在編譯期識別迭代器類別 │
│ • 讓演算法可以根據迭代器類別選擇最佳實作 │
│ │
│ 6. 效能考量 │
│ • Random Access 的 advance/distance 是 O(1) │
│ • 其他迭代器的 advance/distance 是 O(n) │
│ │
└──────────────────────────────────────────────────────────────────────┘
```

5.15 課後思考

1. **思考題**：為什麼 `std::sort` 要求 Random Access Iterator，而不能用在 `list` 上？（提示：想想快速排序的運作方式）
2. **思考題**：unordered_set 的迭代器為什麼只是 Forward Iterator 而不是 Bidirectional？（提示：想想雜湊表的結構）

準備好進入**第六課：容器（Container）的概念與分類**了嗎？下一課我們會深入探討 STL 容器的設計原則和選擇策略。