第四課：迭代器（Iterator）的核心概念

第四課：迭代器（Iterator）的核心概念

4.1 什麼是迭代器？

迭代器（Iterator）是 STL 中最核心的概念之一。簡單來說：

迭代器是一種「泛化的指標」，提供統一的方式來遍歷任何容器中的元素。

從指標說起

你在 C 語言中已經很熟悉指標了：

```
#include <iostream>

int main() {
int arr[] = {10, 20, 30, 40, 50};
// 用指標遍歷陣列
for (int* ptr = arr; ptr < arr + 5; ++ptr) {
std::cout << *ptr << " ";
}
std::cout << std::endl;
return 0;
}
```

輸出：

```
10 20 30 40 50
```

指標的操作：

- `*ptr` ：取得指向的值（解參考）
- `++ptr` ：移動到下一個元素
- `ptr < end` ：比較位置

迭代器就是把這些操作「抽象化」，讓它能用在任何容器上。

4.2 為什麼需要迭代器？

問題：不同容器，不同遍歷方式

```
┌──────────────────────────────────────────────────────────────────────┐
│ 沒有迭代器的世界 │
├──────────────────────────────────────────────────────────────────────┤
│ │
│ 陣列： │
│ for (int i = 0; i < size; ++i) │
│ process(arr[i]); │
│ │
│ 鏈結串列： │
│ for (Node* p = head; p != nullptr; p = p->next) │
│ process(p->data); │
│ │
│ 二元樹： │
│ void traverse(Node* n) { │
│ if (n == nullptr) return; │
│ traverse(n->left); │
│ process(n->data); │
│ traverse(n->right); │
│ } │
│ │
│ 問題：每種資料結構都要寫不同的遍歷邏輯！ │
│ │
└──────────────────────────────────────────────────────────────────────┘
```

解決方案：迭代器統一介面

```
┌──────────────────────────────────────────────────────────────────────┐
│ 有迭代器的世界 │
├──────────────────────────────────────────────────────────────────────┤
│ │
│ 不管什麼容器，遍歷方式都一樣： │
│ │
│ for (auto it = container.begin(); it != container.end(); ++it)│
│ process(*it); │
│ │
│ 或更簡潔的： │
│ │
│ for (auto& element : container) │
│ process(element); │
```

```
│ 底層的複雜性被迭代器封裝了！ │
│ │
└──────────────────────────────────────────────────────────────────────┘
```

4.3 迭代器的基本操作

每個迭代器都支援以下基本操作：

操作	說明	類似的指標操作
`*it`	取得迭代器指向的元素	`*ptr`
`++it`	移動到下一個元素	`++ptr`
`it++`	移動到下一個元素（後置）	`ptr++`
`it1 == it2`	比較兩個迭代器是否相等	`ptr1 == ptr2`
`it1 != it2`	比較兩個迭代器是否不相等	`ptr1 != ptr2`

基本操作示範

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

int main() {
std::vector<int> vec = {10, 20, 30, 40, 50};
std::cout << "=== 迭代器基本操作 ===" << std::endl;
// 取得起始迭代器
std::vector<int>::iterator it = vec.begin();
// *it：解參考
std::cout << "*it = " << *it << std::endl;
// ++it：前進
++it;
std::cout << "++it 後，*it = " << *it << std::endl;
// it++：後置前進
std::cout << "*it++ = " << *it++ << std::endl; // 先取值，再前進
```

```
std::cout << "現在 *it = " << *it << std::endl;
// 比較
std::cout << "\n=== 迭代器比較 ===" << std::endl;
auto begin = vec.begin();
auto end = vec.end();
std::cout << "begin == end? " << (begin == end ? "是" : "否") << std::endl;
std::cout << "begin != end? " << (begin != end ? "是" : "否") << std::endl;
return 0;
}
```

輸出：

```
=== 迭代器基本操作 ===
*it = 10
++it 後，*it = 20
*it++ = 20
現在 *it = 30

=== 迭代器比較 ===
begin == end? 否
begin != end? 是
```

4.4 begin() 與 end()

每個 STL 容器都提供 `begin()` 和 `end()` 成員函數：

```
┌──────────────────────────────────────────────────────────────┐
│ begin() 與 end() 圖解 │
├──────────────────────────────────────────────────────────────┤
│ │
│ 容器內容： [ 10 | 20 | 30 | 40 | 50 ] │
│ ↑ ↑ │
│ begin() end() │
│ │
│ 重點： │
│ • begin() 指向第一個元素 │
│ • end() 指向「最後一個元素的下一個位置」(past-the-end) │
│ • end() 不指向任何有效元素，不能解參考 │
```

```
| |
| 這是「半開區間」[begin, end) |
| • 包含 begin |
| • 不包含 end |
| |
└──────────────────────────────────────────────────────────────────────┘
```

為什麼用半開區間？

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

int main() {
std::vector<int> vec = {10, 20, 30, 40, 50};
// 半開區間的好處：
// 1. 空容器的判斷很簡單
std::vector<int> empty_vec;
if (empty_vec.begin() == empty_vec.end()) {
std::cout << "空容器：begin() == end()" << std::endl;
}
// 2. 遍歷邏輯統一
std::cout << "\n遍歷非空容器: ";
for (auto it = vec.begin(); it != vec.end(); ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
// 3. 計算元素個數
std::cout << "\n元素個數: " << (vec.end() - vec.begin()) << std::endl;
// 4. 子區間表示
auto start = vec.begin() + 1; // 指向 20
auto finish = vec.begin() + 4; // 指向 50（不包含）
std::cout << "子區間 [1, 4): ";
for (auto it = start; it != finish; ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
return 0;
}
```

輸出：

```
空容器：begin() == end()

遍歷非空容器: 10 20 30 40 50

元素個數: 5
子區間 [1, 4): 20 30 40
```

4.5 使用 auto 簡化迭代器宣告

迭代器的完整型別很長，使用 `auto` 可以大幅簡化：

```
#include <iostream>
#include <vector>
#include <list>
#include <map>
#include <string>

int main() {
// 完整型別宣告（冗長）
std::vector<int> vec = {1, 2, 3};
std::vector<int>::iterator it1 = vec.begin();
// 使用 auto（簡潔）
auto it2 = vec.begin();
std::cout << "兩者等價：*it1 = " << *it1 << ", *it2 = " << *it2 << std::endl;
// 對於複雜型別，auto 更加重要
std::map<std::string, int> scores;
scores["Alice"] = 95;
scores["Bob"] = 87;
// 不用 auto，型別非常長
std::map<std::string, int>::iterator map_it1 = scores.begin();
// 用 auto，清爽多了
auto map_it2 = scores.begin();
std::cout << "map 第一個元素：" << map_it2->first
<< " = " << map_it2->second << std::endl;
return 0;
}
```

輸出：

```
兩者等價：*it1 = 1, *it2 = 1
map 第一個元素：Alice = 95
```

4.6 const_iterator：唯讀迭代器

當你只想讀取元素，不想修改時，使用 `const_iterator`：

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

void print_vector(const std::vector<int>& vec) {
// const 容器只能用 const_iterator
// 使用 cbegin() 和 cend() 明確取得 const_iterator
for (auto it = vec.cbegin(); it != vec.cend(); ++it) {
std::cout << *it << " ";
// *it = 100; // 編譯錯誤！不能透過 const_iterator 修改
}
std::cout << std::endl;
}

int main() {
std::vector<int> vec = {10, 20, 30, 40, 50};
// 一般迭代器：可讀可寫
std::cout << "=== 一般迭代器 ===" << std::endl;
for (auto it = vec.begin(); it != vec.end(); ++it) {
*it *= 2; // 可以修改
}
print_vector(vec);
// const_iterator：只能讀
std::cout << "\n=== const_iterator ===" << std::endl;
// 方法一：從 const 容器取得
const std::vector<int>& const_ref = vec;
for (auto it = const_ref.begin(); it != const_ref.end(); ++it) {
std::cout << *it << " ";
}
```

```
std::cout << std::endl;
// 方法二：使用 cbegin() / cend()
for (auto it = vec.cbegin(); it != vec.cend(); ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
return 0;
}
```

輸出：

```
=== 一般迭代器 ===
20 40 60 80 100

=== const_iterator ===
20 40 60 80 100
20 40 60 80 100
```

迭代器型別總覽

```
┌──────────────────────────────────────────────────────────────────┐
│ 迭代器型別總覽 │
├──────────────────────────────────────────────────────────────────┤
│ │
│ 成員函數 回傳型別 用途 │
│ ──────────────────────────────────────────────────────── │
│ begin() iterator 可讀寫，正向 │
│ end() iterator │
│ │
│ cbegin() const_iterator 唯讀，正向 │
│ cend() const_iterator │
│ │
│ rbegin() reverse_iterator 可讀寫，反向 │
│ rend() reverse_iterator │
│ │
│ crbegin() const_reverse_iterator 唯讀，反向 │
│ crend() const_reverse_iterator │
│ │
└──────────────────────────────────────────────────────────────────┘
```

4.7 reverse_iterator：反向迭代器

反向迭代器讓你從後往前遍歷容器：

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

int main() {
std::vector<int> vec = {10, 20, 30, 40, 50};
// 正向遍歷
std::cout << "正向: ";
for (auto it = vec.begin(); it != vec.end(); ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
// 反向遍歷
std::cout << "反向: ";
for (auto rit = vec.rbegin(); rit != vec.rend(); ++rit) {
std::cout << *rit << " ";
}
std::cout << std::endl;
return 0;
}
```

輸出：

```
正向: 10 20 30 40 50
反向: 50 40 30 20 10
```

反向迭代器圖解

```
┌──────────────────────────────────────────────────────────────┐
│ 反向迭代器圖解 │
├──────────────────────────────────────────────────────────────┤
│ │
│ 容器內容： [ 10 | 20 | 30 | 40 | 50 ] │
```

```
| |
| 正向： ↑ ↑ |
| begin() end() |
| (指向 10) (past-the-end) |
| |
| 反向： ↑ ↑ |
| rbegin() rend()|
| (指向 50) (past-the- |
| beginning)|
| |
| rbegin() 的 ++ 會往左移動（邏輯上的「下一個」是前一個元素） |
| |
└──────────────────────────────────────────────────────────────┘
```

4.8 迭代器與演算法

迭代器的真正威力在於它讓演算法與容器解耦：

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

int main() {
std::vector<int> vec = {5, 2, 8, 1, 9};
std::list<int> lst = {5, 2, 8, 1, 9};
// 同一個 find 演算法，用在不同容器
std::cout << "=== std::find ===" << std::endl;
auto vec_it = std::find(vec.begin(), vec.end(), 8);
if (vec_it != vec.end()) {
std::cout << "在 vector 中找到 8" << std::endl;
}
auto lst_it = std::find(lst.begin(), lst.end(), 8);
if (lst_it != lst.end()) {
std::cout << "在 list 中找到 8" << std::endl;
}
// 同一個 count 演算法
std::cout << "\n=== std::count ===" << std::endl;
```

```
std::vector<int> data = {1, 2, 2, 3, 2, 4, 2};
int count = std::count(data.begin(), data.end(), 2);
std::cout << "2 出現了 " << count << " 次" << std::endl;
// 在子區間上操作
std::cout << "\n=== 子區間操作 ===" << std::endl;
std::vector<int> nums = {10, 20, 30, 40, 50, 60, 70};
// 只在 [begin+1, begin+5) 範圍內查找，即 {20, 30, 40, 50}
auto sub_it = std::find(nums.begin() + 1, nums.begin() + 5, 40);
if (sub_it != nums.begin() + 5) {
std::cout << "在子區間中找到 40" << std::endl;
}
return 0;
}
```

輸出：

```
=== std::find ===
在 vector 中找到 8
在 list 中找到 8

=== std::count ===
2 出現了 4 次

=== 子區間操作 ===
在子區間中找到 40
```

4.9 迭代器失效問題

這是使用迭代器時最容易出錯的地方：**當容器被修改時，迭代器可能失效。**

問題示範

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

int main() {
std::vector<int> vec = {1, 2, 3, 4, 5};
// 危險！在遍歷時修改容器
/*
```

```
for (auto it = vec.begin(); it != vec.end(); ++it) {
if (*it == 3) {
vec.erase(it); // 刪除後，it 失效！
// 繼續使用 it 是未定義行為
}
}
*/
// 正確做法：使用 erase 的回傳值
std::cout << "原始: ";
for (int n : vec) std::cout << n << " ";
std::cout << std::endl;
for (auto it = vec.begin(); it != vec.end(); /* 不在這裡 ++ */) {
if (*it == 3) {
it = vec.erase(it); // erase 回傳下一個有效迭代器
} else {
++it;
}
}
std::cout << "刪除 3 後: ";
for (int n : vec) std::cout << n << " ";
std::cout << std::endl;
return 0;
}
```

輸出：

```
原始: 1 2 3 4 5
刪除 3 後: 1 2 4 5
```

迭代器失效規則

```
┌──────────────────────────────────────────────────────────────┐
│ 迭代器失效規則 │
├──────────────────────────────────────────────────────────────┤
│ │
│ vector： │
│ ├── 插入：可能導致所有迭代器失效（若發生重新配置） │
│ ├── 刪除：被刪元素之後的所有迭代器失效 │
│ └── push_back：可能導致所有迭代器失效（若發生重新配置） │
│ │
```

```
| deque： |
| ├── 頭尾插入：所有迭代器失效（但指標和參考不失效） |
| └── 中間插入/刪除：所有迭代器失效 |
| |
| list / forward_list： |
| ├── 插入：不影響現有迭代器 |
| └── 刪除：只有被刪元素的迭代器失效 |
| |
| set / map（樹狀結構）： |
| ├── 插入：不影響現有迭代器 |
| └── 刪除：只有被刪元素的迭代器失效 |
| |
| unordered_set / unordered_map： |
| ├── 插入：可能導致所有迭代器失效（若發生 rehash） |
| └── 刪除：只有被刪元素的迭代器失效 |
| |
└──────────────────────────────────────────────────────────────┘
```

安全的刪除模式

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

int main() {
// 方法一：使用 erase 的回傳值（適用於所有容器）
std::cout << "=== 方法一：erase 回傳值 ===" << std::endl;
std::vector<int> vec1 = {1, 2, 3, 2, 4, 2, 5};
for (auto it = vec1.begin(); it != vec1.end(); ) {
if (*it == 2) {
it = vec1.erase(it);
} else {
++it;
}
}
for (int n : vec1) std::cout << n << " ";
std::cout << std::endl;
// 方法二：使用 erase-remove 慣用法（適用於 vector）
std::cout << "\n=== 方法二：erase-remove 慣用法 ===" << std::endl;
```

```
std::vector<int> vec2 = {1, 2, 3, 2, 4, 2, 5};
vec2.erase(
std::remove(vec2.begin(), vec2.end(), 2),
vec2.end()
);
for (int n : vec2) std::cout << n << " ";
std::cout << std::endl;
// 方法三：對於 list，使用成員函數 remove（更高效）
std::cout << "\n=== 方法三：list::remove ===" << std::endl;
std::list<int> lst = {1, 2, 3, 2, 4, 2, 5};
lst.remove(2); // list 有自己的 remove 成員函數
for (int n : lst) std::cout << n << " ";
std::cout << std::endl;
return 0;
}
```

輸出：

```
=== 方法一：erase 回傳值 ===
1 3 4 5

=== 方法二：erase-remove 慣用法 ===
1 3 4 5

=== 方法三：list::remove ===
1 3 4 5
```

4.10 迭代器與指標的關係

迭代器被設計成「像指標一樣操作」，但它是更抽象的概念：

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

int main() {
std::vector<int> vec = {10, 20, 30, 40, 50};
// 取得底層指標
int* ptr = vec.data(); // C++11
```

```
// 或 int* ptr = &vec[0];
// 取得迭代器
auto it = vec.begin();
std::cout << "=== 相似的操作 ===" << std::endl;
std::cout << "指標解參考: *ptr = " << *ptr << std::endl;
std::cout << "迭代器解參考: *it = " << *it << std::endl;
std::cout << "\n指標算術: *(ptr + 2) = " << *(ptr + 2) << std::endl;
std::cout << "迭代器算術: *(it + 2) = " << *(it + 2) << std::endl;
// 迭代器可以轉換成指標（對於連續記憶體的容器）
std::cout << "\n=== 迭代器轉指標 ===" << std::endl;
int* from_iterator = &(*it); // 取得迭代器指向的元素的位址
std::cout << "*from_iterator = " << *from_iterator << std::endl;
// 但指標不能直接當迭代器用在演算法中（需要配合 size）
// 好消息是，原始指標本身就是一種迭代器！
std::cout << "\n=== 指標作為迭代器 ===" << std::endl;
int arr[] = {100, 200, 300};
// 指標可以直接用在 STL 演算法
auto found = std::find(arr, arr + 3, 200);
if (found != arr + 3) {
std::cout << "在陣列中找到 200" << std::endl;
}
return 0;
}
```

輸出：

```
=== 相似的操作 ===
指標解參考: *ptr = 10
迭代器解參考: *it = 10

指標算術: *(ptr + 2) = 30
迭代器算術: *(it + 2) = 30

=== 迭代器轉指標 ===
*from_iterator = 10

=== 指標作為迭代器 ===
在陣列中找到 200
```

指標就是 Random Access Iterator

```
┌──────────────────────────────────────────────────────────────────────┐
│ 指標與迭代器的關係 │
├──────────────────────────────────────────────────────────────────────┤
│ │
│ 原始指標（T*）滿足 Random Access Iterator 的所有要求： │
│ │
│ ✓ *ptr 解參考 │
│ ✓ ++ptr 前進 │
│ ✓ --ptr 後退 │
│ ✓ ptr + n 隨機存取 │
│ ✓ ptr - n 隨機存取 │
│ ✓ ptr[n] 下標存取 │
│ ✓ ptr1 - ptr2 計算距離 │
│ ✓ ptr1 < ptr2 比較 │
│ │
│ 所以 STL 演算法可以直接用在 C 風格陣列上！ │
│ │
└──────────────────────────────────────────────────────────────────────┘
```

4.11 範圍 for 與迭代器

C++11 的範圍 for 迴圈底層就是使用迭代器：

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

int main() {
std::vector<int> vec = {10, 20, 30, 40, 50};
// 範圍 for（語法糖）
std::cout << "範圍 for: ";
for (int n : vec) {
std::cout << n << " ";
}
std::cout << std::endl;
// 等價的迭代器寫法
std::cout << "迭代器: ";
for (auto it = vec.begin(); it != vec.end(); ++it) {
```

```
std::cout << *it << " ";
}
std::cout << std::endl;
// 編譯器實際上把範圍 for 轉換成類似這樣：
/*
{
auto&& __range = vec;
auto __begin = __range.begin();
auto __end = __range.end();
for (; __begin != __end; ++__begin) {
int n = *__begin;
// 迴圈主體
}
}
*/
// 如果要修改元素，使用參考
std::cout << "\n修改元素（使用參考）: ";
for (int& n : vec) {
n *= 2;
}
for (int n : vec) {
std::cout << n << " ";
}
std::cout << std::endl;
return 0;
}
```

輸出：

```
範圍 for: 10 20 30 40 50
迭代器: 10 20 30 40 50

修改元素（使用參考）: 20 40 60 80 100
```

4.12 完整範例：自訂類別使用迭代器

讓我們看看如何讓自訂類別支援迭代器和範圍 for：

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

class IntRange {
private:
int start_;
int end_;
public:
// 內部迭代器類別
class Iterator {
private:
int current_;
public:
Iterator(int value) : current_(value) {}
int operator*() const { return current_; }
Iterator& operator++() {
++current_;
return *this;
}
bool operator!=(const Iterator& other) const {
return current_ != other.current_;
}
};
IntRange(int start, int end) : start_(start), end_(end) {}
Iterator begin() const { return Iterator(start_); }
Iterator end() const { return Iterator(end_); }
};

int main() {
std::cout << "=== 自訂 IntRange 類別 ===" << std::endl;
IntRange range(1, 6); // 代表 1, 2, 3, 4, 5
// 可以使用範圍 for！
std::cout << "範圍 for: ";
for (int n : range) {
std::cout << n << " ";
}
std::cout << std::endl;
// 也可以使用迭代器
std::cout << "迭代器: ";
```

```
for (auto it = range.begin(); it != range.end(); ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
return 0;
}
```

輸出：

```
=== 自訂 IntRange 類別 ===
範圍 for: 1 2 3 4 5
迭代器: 1 2 3 4 5
```

4.13 本課重點整理

```
┌──────────────────────────────────────────────────────────────────────┐
│ 第四課 重點整理 │
├──────────────────────────────────────────────────────────────────────┤
│ │
│ 1. 迭代器是什麼 │
│ → 泛化的指標，提供統一的容器遍歷方式 │
│ → 連接容器與演算法的橋樑 │
│ │
│ 2. 基本操作 │
│ → *it（解參考）、++it（前進）、it1 != it2（比較） │
│ │
│ 3. begin() 與 end() │
│ → 半開區間 [begin, end) │
│ → end() 指向「最後元素的下一個位置」 │
│ │
│ 4. 迭代器種類 │
│ → iterator：可讀寫 │
│ → const_iterator：唯讀（cbegin/cend） │
│ → reverse_iterator：反向（rbegin/rend） │
│ │
│ 5. 迭代器失效 │
│ → 容器修改時，迭代器可能失效 │
```

```
| → 使用 erase 的回傳值來安全刪除 |
| → 不同容器有不同的失效規則 |
| |
| 6. 迭代器與指標 |
| → 指標是一種 Random Access Iterator |
| → STL 演算法可以直接用在 C 風格陣列 |
| |
| 7. 範圍 for |
| → 語法糖，底層使用迭代器 |
| → 需要 begin() 和 end() |
| |
└──────────────────────────────────────────────────────────────┘
```

4.14 課後練習

1. **思考題**：為什麼 `end()` 不指向最後一個元素，而是指向「最後一個元素的下一個位置」？這種設計有什麼好處？
2. **實作題**：寫一個函數，接受一個 vector，使用迭代器將所有奇數刪除（注意迭代器失效問題）。

提示：

```
void remove_odds(std::vector<int>& vec) {
// 你的實作
}
```

準備好進入**第五課：迭代器的五種分類**了嗎？下一課我們會深入探討 Input、Output、Forward、Bidirectional、Random Access 這五種迭代器的能力差異。