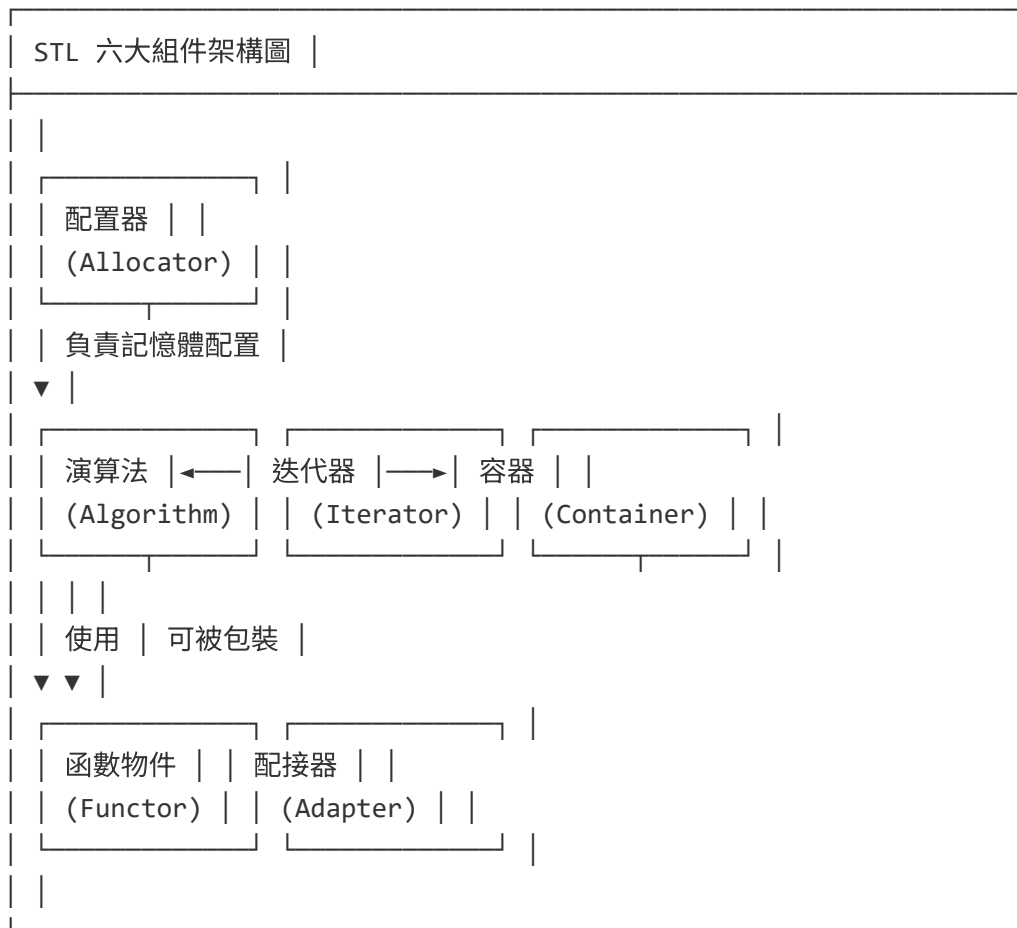


第三課：STL 的六大組件概覽

第三課：STL 的六大組件概覽

3.1 STL 的整體架構

STL 由六大組件構成，它們彼此協作，形成一個強大而靈活的系統：



讓我們逐一認識這六大組件。

3.2 組件一：容器 (Containers)

容器是用來儲存和管理資料的類別模板。

容器的分類

容器分類總覽
序列容器 (Sequence Containers)
array 固定大小陣列
vector 動態陣列 (最常用)
deque 雙端佇列
list 雙向鏈結串列
forward_list 單向鏈結串列
關聯容器 (Associative Containers)
set 有序集合 (不重複)
multiset 有序集合 (可重複)
map 有序鍵值對 (不重複鍵)
multimap 有序鍵值對 (可重複鍵)
無序容器 (Unordered Containers)
unordered_set 雜湊集合 (不重複)
unordered_multiset 雜湊集合 (可重複)
unordered_map 雜湊鍵值對 (不重複鍵)
unordered_multimap 雜湊鍵值對 (可重複鍵)
容器配接器 (Container Adapters)
stack 後進先出 (LIFO)
queue 先進先出 (FIFO)
priority_queue 優先佇列

容器快速示範

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

```

int main() {
// 序列容器：vector
std::vector<int> vec = {1, 2, 3, 4, 5};
std::cout << "vector: ";
for (int n : vec) std::cout << n << " ";
std::cout << std::endl;
// 序列容器：list
std::list<int> lst = {10, 20, 30};
lst.push_front(5); // list 可以高效地在前端插入
std::cout << "list: ";
for (int n : lst) std::cout << n << " ";
std::cout << std::endl;
// 關聯容器：set（自動排序、不重複）
std::set<int> s = {30, 10, 20, 10, 30}; // 重複的會被忽略
std::cout << "set: ";
for (int n : s) std::cout << n << " ";
std::cout << std::endl;
// 關聯容器：map（鍵值對）
std::map<std::string, int> ages;
ages["Alice"] = 25;
ages["Bob"] = 30;
ages["Charlie"] = 35;
std::cout << "map: ";
for (const auto& pair : ages) {
std::cout << pair.first << "=" << pair.second << " ";
}
std::cout << std::endl;
return 0;
}

```

輸出：

```

vector: 1 2 3 4 5
list: 5 10 20 30
set: 10 20 30
map: Alice=25 Bob=30 Charlie=35

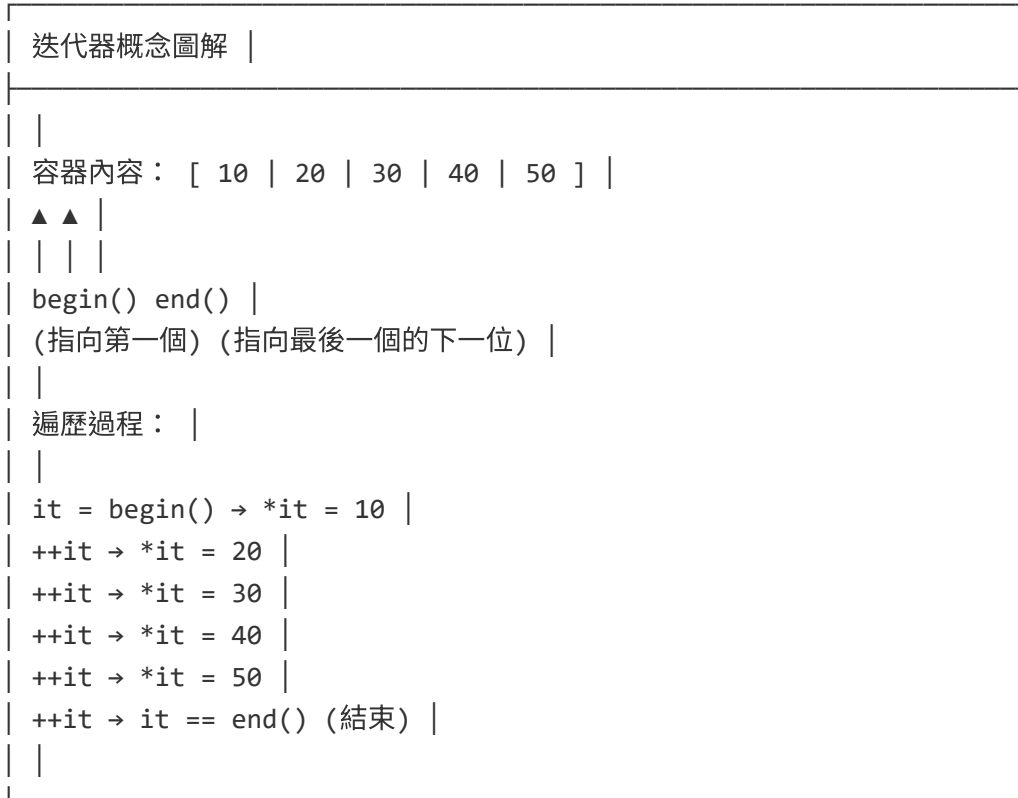
```

3.3 組件二：迭代器 (Iterators)

迭代器是連接容器與演算法的橋樑，它提供了一種統一的方式來遍歷容器中的元素。

迭代器的概念

你可以把迭代器想像成「泛化的指標」：



迭代器基本操作

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

int main() {
    std::vector<int> vec = {10, 20, 30, 40, 50};
    // 方法一：使用迭代器遍歷
    std::cout << "使用迭代器： ";
    for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " "; // *it 取得迭代器指向的值
    }
    std::cout << std::endl;
    // 方法二：使用 auto 簡化
```

```

std::cout << "使用 auto: ";
for (auto it = vec.begin(); it != vec.end(); ++it) {
std::cout << *it << " ";
}
std::cout << std::endl;
// 方法三：範圍 for（底層也是迭代器）
std::cout << "範圍 for: ";
for (int n : vec) {
std::cout << n << " ";
}
std::cout << std::endl;
return 0;
}

```

輸出：

使用迭代器: 10 20 30 40 50

使用 auto: 10 20 30 40 50

範圍 for: 10 20 30 40 50

迭代器的五種類別

迭代器類別層次	
功能最強	
▲	
Random Access Iterator（隨機存取迭代器）	
• 支援 +n, -n, [], <, >	
• 代表: vector, deque, array	
Bidirectional Iterator（雙向迭代器）	
• 支援 ++, --	
• 代表: list, set, map	
Forward Iterator（前向迭代器）	
• 只支援 ++	
• 代表: forward_list, unordered_set	

		Input Iterator (輸入迭代器)	
		• 只能讀取，只能前進一次	
		• 代表：istream_iterator	
		Output Iterator (輸出迭代器)	
		• 只能寫入，只能前進一次	
		• 代表：ostream_iterator	
	▼		
		功能最弱	

不同迭代器的能力差異

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

int main() {
    // vector 有 Random Access Iterator
    std::vector<int> vec = {10, 20, 30, 40, 50};
    auto vit = vec.begin();
    std::cout << "vector 迭代器可以：" << std::endl;
    std::cout << " vit[2] = " << vit[2] << std::endl; // 隨機存取
    std::cout << " *(vit + 3) = " << *(vit + 3) << std::endl; // 算術運算
    // list 只有 Bidirectional Iterator
    std::list<int> lst = {10, 20, 30, 40, 50};
    auto lit = lst.begin();
    std::cout << "\nlist 迭代器可以：" << std::endl;
    ++lit; // 前進
    std::cout << " ++lit → *lit = " << *lit << std::endl;
    --lit; // 後退
    std::cout << " --lit → *lit = " << *lit << std::endl;
    // 但 list 迭代器不能：
    // lit[2]; // 編譯錯誤！
    // lit + 3; // 編譯錯誤！
    return 0;
}
```

輸出：

vector 迭代器可以：

vit[2] = 30

*(vit + 3) = 40

list 迭代器可以：

++lit → *lit = 20

--lit → *lit = 10

3.4 組件三：演算法 (Algorithms)

演算法是一系列對容器元素進行操作的函數模板。它們透過迭代器來操作容器，與容器本身解耦。

演算法的分類

演算法分類總覽
非修改序列操作
├─ find, find_if 查找元素
├─ count, count_if 計數
├─ for_each 對每個元素執行操作
└─ all_of, any_of 條件判斷
修改序列操作
├─ copy, copy_if 複製元素
├─ transform 轉換元素
├─ fill, fill_n 填充
├─ replace 替換
└─ remove, remove_if 移除
排序相關操作
├─ sort, stable_sort 排序
├─ partial_sort 部分排序
├─ nth_element 找第 n 個元素
└─ binary_search 二分搜尋

數值操作 (<numeric>)
— accumulate 累加
— inner_product 內積
— partial_sum 部分和

演算法示範

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

int main() {
    std::vector<int> vec = {5, 2, 8, 1, 9, 3, 7, 4, 6};
    // 排序
    std::sort(vec.begin(), vec.end());
    std::cout << "排序後: ";
    for (int n : vec) std::cout << n << " ";
    std::cout << std::endl;
    // 查找
    auto it = std::find(vec.begin(), vec.end(), 7);
    if (it != vec.end()) {
        std::cout << "找到 7，位置: " << (it - vec.begin()) << std::endl;
    }
    // 計數
    std::vector<int> data = {1, 2, 2, 3, 2, 4, 2, 5};
    int count = std::count(data.begin(), data.end(), 2);
    std::cout << "2 出現次數: " << count << std::endl;
    // 累加
    int sum = std::accumulate(vec.begin(), vec.end(), 0);
    std::cout << "總和: " << sum << std::endl;
    // 最大最小
    auto minmax = std::minmax_element(vec.begin(), vec.end());
    std::cout << "最小值: " << *minmax.first << std::endl;
    std::cout << "最大值: " << *minmax.second << std::endl;
    return 0;
}
```

輸出：

排序後： 1 2 3 4 5 6 7 8 9

找到 7，位置： 6

2 出現次數： 4

總和： 45

最小值： 1

最大值： 9

演算法與容器的獨立性

同一個演算法可以用在不同容器上：

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

template <typename Container>
void print_container(const std::string& name, const Container& c) {
    std::cout << name << ": ";
    for (const auto& elem : c) std::cout << elem << " ";
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec = {3, 1, 4, 1, 5};
    std::list<int> lst = {9, 2, 6, 5, 3};
    std::deque<int> deq = {5, 8, 9, 7, 9};
    // 同一個 sort 演算法用在 vector 和 deque
    std::sort(vec.begin(), vec.end());
    std::sort(deq.begin(), deq.end());
    // list 有自己的 sort (因為 std::sort 需要 Random Access Iterator)
    lst.sort();
    print_container("vector", vec);
    print_container("list", lst);
    print_container("deque", deq);
    // 同一個 find 演算法用在所有容器
    std::cout << "\n尋找元素 5：" << std::endl;
```

```

auto vit = std::find(vec.begin(), vec.end(), 5);
std::cout << " vector: " << (vit != vec.end() ? "找到" : "沒找到") <<
std::endl;
auto lit = std::find(lst.begin(), lst.end(), 5);
std::cout << " list: " << (lit != lst.end() ? "找到" : "沒找到") << std::endl;
auto dit = std::find(deq.begin(), deq.end(), 5);
std::cout << " deque: " << (dit != deq.end() ? "找到" : "沒找到") << std::endl;
return 0;
}

```

輸出：

```

vector: 1 1 3 4 5
list: 2 3 5 6 9
deque: 5 7 8 9 9

```

```

尋找元素 5：
vector: 找到
list: 找到
deque: 找到

```

3.5 組件四：函數物件（Function Objects / Functors）

函數物件是重載了 `operator()` 的類別實例，它可以像函數一樣被呼叫。

為什麼需要函數物件？

演算法常常需要一個「策略」來決定如何操作。例如：

- `sort` 需要知道如何比較兩個元素
- `find_if` 需要知道什麼條件算「找到」
- `transform` 需要知道如何轉換元素

函數物件 vs 函數指標

```

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

// 方法一：普通函數
bool is_even_func(int n) {

```

```

return n % 2 == 0;
}

// 方法二：函數物件
class IsEven {
public:
bool operator()(int n) const {
return n % 2 == 0;
}
};

// 方法三：帶狀態的函數物件
class IsDivisibleBy {
int divisor;
public:
IsDivisibleBy(int d) : divisor(d) {}
bool operator()(int n) const {
return n % divisor == 0;
}
};

int main() {
std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// 使用普通函數
int count1 = std::count_if(vec.begin(), vec.end(), is_even_func);
std::cout << "偶數個數（函數）： " << count1 << std::endl;
// 使用函數物件
int count2 = std::count_if(vec.begin(), vec.end(), IsEven());
std::cout << "偶數個數（函數物件）： " << count2 << std::endl;
// 使用帶狀態的函數物件
int count3 = std::count_if(vec.begin(), vec.end(), IsDivisibleBy(3));
std::cout << "3的倍數個數： " << count3 << std::endl;
int count4 = std::count_if(vec.begin(), vec.end(), IsDivisibleBy(5));
std::cout << "5的倍數個數： " << count4 << std::endl;
return 0;
}

```

輸出：

偶數個數（函數）： 5

偶數個數（函數物件）： 5

3的倍數個數： 3

5的倍數個數： 2

STL 內建的函數物件

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // 內建函數物件

int main() {
    std::vector<int> vec = {5, 2, 8, 1, 9};
    // 預設排序（升序）
    std::sort(vec.begin(), vec.end());
    std::cout << "升序: ";
    for (int n : vec) std::cout << n << " ";
    std::cout << std::endl;
    // 使用 greater<int> 降序排序
    std::sort(vec.begin(), vec.end(), std::greater<int>());
    std::cout << "降序: ";
    for (int n : vec) std::cout << n << " ";
    std::cout << std::endl;
    // 其他內建函數物件
    std::cout << "\n內建函數物件示範：" << std::endl;
    std::cout << "plus<int>()(3, 4) = " << std::plus<int>()(3, 4) << std::endl;
    std::cout << "minus<int>()(10, 3) = " << std::minus<int>()(10, 3) <<
    std::endl;
    std::cout << "multiplies<int>()(5, 6) = " << std::multiplies<int>()(5, 6) <<
    std::endl;
    std::cout << "logical_and<bool>()(true, false) = "
    << std::logical_and<bool>()(true, false) << std::endl;
    return 0;
}
```

輸出：

升序: 1 2 5 8 9

降序: 9 8 5 2 1

內建函數物件示範：

```
plus<int>()(3, 4) = 7
minus<int>()(10, 3) = 7
multiplies<int>()(5, 6) = 30
logical_and<bool>()(true, false) = 0
```

Lambda 表達式：現代的函數物件

C++11 引入了 Lambda，讓函數物件的撰寫更簡潔：

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    // Lambda 表達式
    int count = std::count_if(vec.begin(), vec.end(),
    [](int n) { return n % 2 == 0; } // Lambda
    );
    std::cout << "偶數個數: " << count << std::endl;
    // 帶捕獲的 Lambda
    int divisor = 3;
    int count3 = std::count_if(vec.begin(), vec.end(),
    [divisor](int n) { return n % divisor == 0; } // 捕獲外部變數
    );
    std::cout << "3的倍數個數: " << count3 << std::endl;
    // 用 Lambda 做 transform
    std::vector<int> squared;
    squared.resize(vec.size());
    std::transform(vec.begin(), vec.end(), squared.begin(),
    [](int n) { return n * n; }
    );
    std::cout << "平方: ";
    for (int n : squared) std::cout << n << " ";
    std::cout << std::endl;
    return 0;
}
```

輸出：

偶數個數：5

3的倍數個數：3

平方：1 4 9 16 25 36 49 64 81 100

3.6 組件五：配接器 (Adapters)

配接器是用來修改或包裝其他組件介面的工具。STL 有三類配接器：

容器配接器

```
#include <iostream>
#include <stack>
#include <queue>

int main() {
    // stack：後進先出 (LIFO)
    std::stack<int> stk;
    stk.push(1);
    stk.push(2);
    stk.push(3);
    std::cout << "stack (LIFO): ";
    while (!stk.empty()) {
        std::cout << stk.top() << " ";
        stk.pop();
    }
    std::cout << std::endl;
    // queue：先進先出 (FIFO)
    std::queue<int> que;
    que.push(1);
    que.push(2);
    que.push(3);
    std::cout << "queue (FIFO): ";
    while (!que.empty()) {
        std::cout << que.front() << " ";
        que.pop();
    }
    std::cout << std::endl;
    // priority_queue：優先佇列（預設最大值優先）
    std::priority_queue<int> pq;
```

```

pq.push(30);
pq.push(10);
pq.push(50);
pq.push(20);
std::cout << "priority_queue: ";
while (!pq.empty()) {
    std::cout << pq.top() << " ";
    pq.pop();
}
std::cout << std::endl;
return 0;
}

```

輸出：

```

stack (LIFO): 3 2 1
queue (FIFO): 1 2 3
priority_queue: 50 30 20 10

```

迭代器配接器

```

#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    // reverse_iterator：反向迭代
    std::cout << "反向迭代: ";
    for (auto rit = vec.rbegin(); rit != vec.rend(); ++rit) {
        std::cout << *rit << " ";
    }
    std::cout << std::endl;
    // back_inserter：自動在尾端插入
    std::vector<int> src = {10, 20, 30};
    std::vector<int> dest;
    std::copy(src.begin(), src.end(), std::back_inserter(dest));
    std::cout << "back_inserter 結果: ";
    for (int n : dest) std::cout << n << " ";
}

```

```

std::cout << std::endl;
// ostream_iterator：輸出到串流
std::cout << "ostream_iterator: ";
std::copy(vec.begin(), vec.end(),
std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;
return 0;
}

```

輸出：

```

反向迭代: 5 4 3 2 1
back_inserter 結果: 10 20 30
ostream_iterator: 1 2 3 4 5

```

函數配接器 (C++11 後主要用 Lambda 取代)

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

int main() {
std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// std::bind：綁定參數
auto is_greater_than_5 = std::bind(std::greater<int>(),
std::placeholders::_1, 5);
int count = std::count_if(vec.begin(), vec.end(), is_greater_than_5);
std::cout << "大於5的個數: " << count << std::endl;
// 等價的 Lambda 寫法 (更推薦)
int count2 = std::count_if(vec.begin(), vec.end(),
[](int n) { return n > 5; }
);
std::cout << "大於5的個數 (Lambda) : " << count2 << std::endl;
return 0;
}

```

輸出：

大於5的個數： 5

大於5的個數 (Lambda) : 5

3.7 組件六：配置器 (Allocators)

配置器負責容器的記憶體配置與釋放。大多數情況下，你不需要直接使用它。

預設配置器

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

int main() {
    // 所有容器都有一個預設的配置器
    // vector<int> 實際上是 vector<int, allocator<int>>
    std::vector<int> vec1; // 使用預設配置器
    std::vector<int, std::allocator<int>> vec2; // 明確指定 (效果相同)
    vec1.push_back(1);
    vec2.push_back(2);
    std::cout << "vec1[0] = " << vec1[0] << std::endl;
    std::cout << "vec2[0] = " << vec2[0] << std::endl;
    // allocator 的基本用法
    std::allocator<int> alloc;
    // 配置記憶體 (但不建構物件)
    int* ptr = alloc.allocate(5); // 配置 5 個 int 的空間
    // 建構物件
    for (int i = 0; i < 5; ++i) {
        std::allocator_traits<std::allocator<int>>::construct(alloc, ptr + i, i * 10);
    }
    std::cout << "手動配置的陣列: ";
    for (int i = 0; i < 5; ++i) {
        std::cout << ptr[i] << " ";
    }
    std::cout << std::endl;
    // 解構物件
    for (int i = 0; i < 5; ++i) {
        std::allocator_traits<std::allocator<int>>::destroy(alloc, ptr + i);
    }
}
```

```
// 釋放記憶體
alloc.deallocate(ptr, 5);
return 0;
}
```

輸出：

```
vec1[0] = 1
vec2[0] = 2
手動配置的陣列: 0 10 20 30 40
```

配置器在進階應用中很有用，例如：

- 記憶體池（Memory Pool）
- 追蹤記憶體使用
- 特殊的記憶體區域（如共享記憶體）

目前你只需要知道它的存在，我們在第十六階段會深入探討。

3.8 六大組件的協作範例

讓我們用一個完整的例子展示六大組件如何協同工作：

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

int main() {
// 【容器】儲存資料
std::vector<int> numbers = {64, 25, 12, 22, 11, 90, 42};
std::cout << "原始資料: ";
// 【迭代器配接器】 ostream_iterator 輸出
std::copy(numbers.begin(), numbers.end(),
std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;
// 【演算法】 sort + 【函數物件】 greater
std::sort(numbers.begin(), numbers.end(), std::greater<int>());
std::cout << "降序排序: ";
std::copy(numbers.begin(), numbers.end(),
std::ostream_iterator<int>(std::cout, " "));
```

```

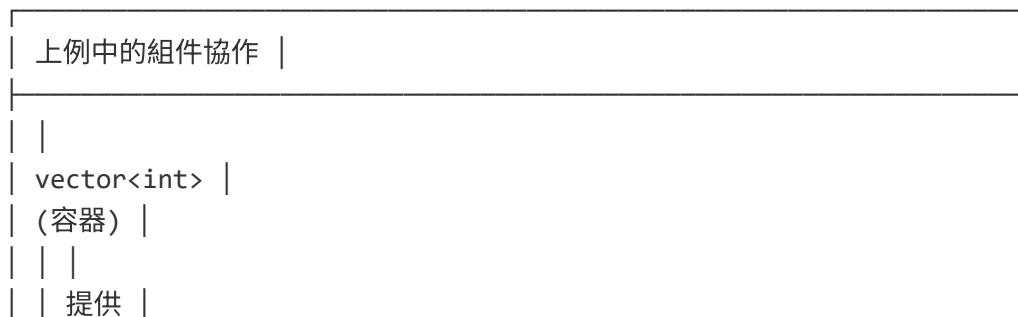
std::cout << std::endl;
// 【演算法】copy_if + 【Lambda (函數物件)】
std::vector<int> even_numbers;
std::copy_if(numbers.begin(), numbers.end(),
std::back_inserter(even_numbers), // 【迭代器配接器】
[](int n) { return n % 2 == 0; }); // 【Lambda】
std::cout << "偶數: ";
std::copy(even_numbers.begin(), even_numbers.end(),
std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;
// 【演算法】transform + 【Lambda】
std::vector<int> doubled;
doubled.resize(even_numbers.size());
std::transform(even_numbers.begin(), even_numbers.end(),
doubled.begin(),
[](int n) { return n * 2; });
std::cout << "偶數加倍: ";
std::copy(doubled.begin(), doubled.end(),
std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;
return 0;
}

```

輸出：

原始資料: 64 25 12 22 11 90 42
 降序排序: 90 64 42 25 22 12 11
 偶數: 90 64 42 22 12
 偶數加倍: 180 128 84 44 24

組件關係圖



▼
begin() / end()
(迭代器)
傳給
▼
sort / copy_if / transform ← greater<int> / Lambda
(演算法) (函數物件)
輸出到
▼
back_inserter / ostream_iterator
(迭代器配接器)
※ 整個過程中，allocator 在背後默默配置記憶體

3.9 本課重點整理

第三課 重點整理
<div> <div>STL 六大組件：</div> <div> <div>1. 容器 (Containers)</div> <div>→ 儲存資料：vector, list, set, map...</div> </div> <div> <div>2. 迭代器 (Iterators)</div> <div>→ 遍歷容器的泛化指標</div> <div>→ 五種類別：Input, Output, Forward, Bidirectional, Random</div> </div> <div> <div>3. 演算法 (Algorithms)</div> <div>→ 操作資料：sort, find, copy, transform...</div> <div>→ 透過迭代器與容器解耦</div> </div> </div>

	4. 函數物件 (Functors)	
	→ 可呼叫的物件：重載 operator()	
	→ 現代 C++ 常用 Lambda 取代	
	5. 配接器 (Adapters)	
	→ 容器配接器：stack, queue, priority_queue	
	→ 迭代器配接器：reverse_iterator, back_inserter	
	6. 配置器 (Allocators)	
	→ 管理記憶體配置與釋放	
	→ 通常使用預設，進階才需自訂	
	核心設計：	
	→ 容器與演算法透過迭代器連接，達到正交性	
	→ 函數物件讓演算法行為可客製化	

3.10 課後思考

1. **思考題：**為什麼 `std::sort` 不能直接用在 `std::list` 上，而 `list` 要有自己的 `sort` 成員函數？
2. **觀察題：**在你目前或過去的專案中，有哪些地方可以用 STL 的容器和演算法來簡化程式碼？

準備好進入**第四課：迭代器 (Iterator) 的核心概念**了嗎？下一課我們會深入探討迭代器這個「連接容器與演算法的橋樑」，理解它的設計精髓。

很好，這兩個概念確實比較進階。讓我為你詳細解釋。

補充課程：Allocators 與 `std::bind` 深入解析

第一部分：Allocators (配置器)

1.1 為什麼需要 Allocator？

在 C 語言中，你用 malloc 和 free 來管理記憶體。在 C++ 中，STL 容器需要一個統一的方式來處理記憶體配置，這就是 Allocator 的角色。

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

int main() {
    // 你寫的程式碼
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    // 實際上 vector 內部做的事：
    // 1. 需要記憶體 → 呼叫 allocator.allocate(n)
    // 2. 在記憶體上建構物件 → 呼叫 allocator.construct(ptr, value)
    // 3. 不需要時解構物件 → 呼叫 allocator.destroy(ptr)
    // 4. 釋放記憶體 → 呼叫 allocator.deallocate(ptr, n)
    return 0;
}
```

1.2 Allocator 的基本介面

```
#include <iostream>
#include <memory>

int main() {
    // 建立一個 int 型別的配置器
    std::allocator<int> alloc;
    std::cout << "=== Allocator 基本操作 ===" << std::endl;
    // Step 1: allocate - 配置原始記憶體（不呼叫建構子）
    int* ptr = alloc.allocate(3); // 配置 3 個 int 的空間
    std::cout << "配置了 3 個 int 的空間，位址: " << ptr << std::endl;
    // 此時記憶體是「原始的」，裡面是垃圾值
    // 直接使用可能有未定義行為
    // Step 2: construct - 在原始記憶體上建構物件
    std::allocator_traits<std::allocator<int>>::construct(alloc, ptr, 100);
    std::allocator_traits<std::allocator<int>>::construct(alloc, ptr + 1, 200);
    std::allocator_traits<std::allocator<int>>::construct(alloc, ptr + 2, 300);
    std::cout << "建構後的值: " << ptr[0] << ", " << ptr[1] << ", " << ptr[2] <<
    std::endl;
    // Step 3: destroy - 解構物件（對 int 來說沒什麼作用，但對複雜型別很重要）
}
```

```

std::allocator_traits<std::allocator<int>>::destroy(alloc, ptr);
std::allocator_traits<std::allocator<int>>::destroy(alloc, ptr + 1);
std::allocator_traits<std::allocator<int>>::destroy(alloc, ptr + 2);
std::cout << "物件已解構" << std::endl;
// Step 4: deallocate - 釋放記憶體
alloc.deallocate(ptr, 3);
std::cout << "記憶體已釋放" << std::endl;
return 0;
}

```

輸出：

```

=== Allocator 基本操作 ===
配置了 3 個 int 的空間，位址：0x...
建構後的值：100, 200, 300
物件已解構
記憶體已釋放

```

1.3 對複雜型別的重要性

對 `int` 來說，`construct` 和 `destroy` 似乎多此一舉。但對有建構子/解構子的型別，這非常重要：

```

#include <iostream>
#include <memory>
#include <string>

class Person {
public:
    std::string name;
    int age;
    Person(const std::string& n, int a) : name(n), age(a) {
        std::cout << " [建構] " << name << std::endl;
    }
    ~Person() {
        std::cout << " [解構] " << name << std::endl;
    }
};

int main() {

```

```

std::allocator<Person> alloc;
std::cout << "=== 配置記憶體 ===" << std::endl;
Person* ptr = alloc.allocate(2);
// 此時只有記憶體，還沒有 Person 物件
std::cout << "\n=== 建構物件 ===" << std::endl;
std::allocator_traits<std::allocator<Person>>::construct(
alloc, ptr, "Alice", 25);
std::allocator_traits<std::allocator<Person>>::construct(
alloc, ptr + 1, "Bob", 30);
std::cout << "\n=== 使用物件 ===" << std::endl;
std::cout << ptr[0].name << " is " << ptr[0].age << std::endl;
std::cout << ptr[1].name << " is " << ptr[1].age << std::endl;
std::cout << "\n=== 解構物件 ===" << std::endl;
std::allocator_traits<std::allocator<Person>>::destroy(alloc, ptr);
std::allocator_traits<std::allocator<Person>>::destroy(alloc, ptr + 1);
std::cout << "\n=== 釋放記憶體 ===" << std::endl;
alloc.deallocate(ptr, 2);
std::cout << "完成" << std::endl;
return 0;
}

```

輸出：

=== 配置記憶體 ===

=== 建構物件 ===

[建構] Alice

[建構] Bob

=== 使用物件 ===

Alice is 25

Bob is 30

=== 解構物件 ===

[解構] Alice

[解構] Bob

=== 釋放記憶體 ===

完成

1.4 allocate vs new 的差異

allocate vs new 比較
new 運算子：
<div>Person* p = new Person("Alice", 25);</div> <div>→ 配置記憶體 + 呼叫建構子（一步完成）</div>
allocator：
<div>allocator<Person> alloc;</div> <div>// Step 1: 只配置記憶體</div> <div>Person* p = alloc.allocate(1);</div> <div>// Step 2: 在記憶體上建構物件</div> <div>construct(alloc, p, "Alice", 25);</div> <div>→ 兩步分開，更精細的控制</div>
為什麼 STL 需要這種分離？
→ vector 預留空間時，只需配置記憶體，不需建構物件
→ push_back 時才在那個位置建構

1.5 理解 vector 的 capacity 與 size

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

```
class Widget {
public:
```

```

int id;
Widget(int i) : id(i) {
    std::cout << " Widget(" << id << ") 建構" << std::endl;
}
~Widget() {
    std::cout << " Widget(" << id << ") 解構" << std::endl;
}
};

int main() {
    std::cout << "=== 建立 vector 並 reserve ===" << std::endl;
    std::vector<Widget> vec;
    vec.reserve(5); // 配置 5 個 Widget 的記憶體，但不建構任何 Widget
    std::cout << "size: " << vec.size() << std::endl; // 0
    std::cout << "capacity: " << vec.capacity() << std::endl; // 5
    std::cout << "\n=== push_back 3 個元素 ===" << std::endl;
    vec.push_back(Widget(1)); // 在位置 0 建構
    vec.push_back(Widget(2)); // 在位置 1 建構
    vec.push_back(Widget(3)); // 在位置 2 建構
    std::cout << "\nsize: " << vec.size() << std::endl; // 3
    std::cout << "capacity: " << vec.capacity() << std::endl; // 5
    std::cout << "\n=== 離開作用域 ===" << std::endl;
    // vector 解構時，只解構 3 個已建構的 Widget
    // 然後釋放全部 5 個位置的記憶體
    return 0;
}

```

輸出：

```

=== 建立 vector 並 reserve ===
size: 0
capacity: 5

=== push_back 3 個元素 ===
Widget(1) 建構
Widget(1) 解構
Widget(2) 建構
Widget(2) 解構
Widget(3) 建構
Widget(3) 解構

```

```
size: 3
capacity: 5
```

=== 離開作用域 ===

Widget(1) 解構

Widget(2) 解構

Widget(3) 解構

(注意：額外的建構/解構是因為 push_back 參數傳遞和移動語意)

1.6 自訂 Allocator (進階)

這是一個簡單的追蹤記憶體使用的 Allocator：

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

template <typename T>
class TrackingAllocator {
public:
    using value_type = T;
    // 追蹤總共配置了多少記憶體
    static size_t total_allocated;
    static size_t total_deallocated;
    TrackingAllocator() = default;
    template <typename U>
    TrackingAllocator(const TrackingAllocator<U>&) {}
    T* allocate(size_t n) {
        size_t bytes = n * sizeof(T);
        total_allocated += bytes;
        std::cout << "[Allocator] 配置 " << n << " 個 "
        << typeid(T).name() << " (" << bytes << " bytes)" << std::endl;
        return static_cast<T*>(::operator new(bytes));
    }
    void deallocate(T* ptr, size_t n) {
        size_t bytes = n * sizeof(T);
        total_deallocated += bytes;
        std::cout << "[Allocator] 釋放 " << n << " 個 "
        << typeid(T).name() << " (" << bytes << " bytes)" << std::endl;
        ::operator delete(ptr);
    }
};
```

```

}
};

template <typename T>
size_t TrackingAllocator<T>::total_allocated = 0;

template <typename T>
size_t TrackingAllocator<T>::total_deallocated = 0;

template <typename T, typename U>
bool operator==(const TrackingAllocator<T>&, const TrackingAllocator<U>&) {
return true;
}

template <typename T, typename U>
bool operator!=(const TrackingAllocator<T>&, const TrackingAllocator<U>&) {
return false;
}

int main() {
std::cout << "=== 使用自訂 Allocator 的 vector ===" << std::endl;
{
std::vector<int, TrackingAllocator<int>>> vec;
std::cout << "\n--- push_back 1 ---" << std::endl;
vec.push_back(1);
std::cout << "\n--- push_back 2 ---" << std::endl;
vec.push_back(2);
std::cout << "\n--- push_back 3, 4, 5 ---" << std::endl;
vec.push_back(3);
vec.push_back(4);
vec.push_back(5);
std::cout << "\n--- 離開作用域 ---" << std::endl;
}
std::cout << "\n=== 統計 ===" << std::endl;
std::cout << "總共配置: " << TrackingAllocator<int>::total_allocated << "
bytes" << std::endl;
std::cout << "總共釋放: " << TrackingAllocator<int>::total_deallocated << "
bytes" << std::endl;
return 0;
}

```

可能的輸出：

```
=== 使用自訂 Allocator 的 vector ===
```

```
--- push_back 1 ---
```

```
[Allocator] 配置 1 個 i (4 bytes)
```

```
--- push_back 2 ---
```

```
[Allocator] 配置 2 個 i (8 bytes)
```

```
[Allocator] 釋放 1 個 i (4 bytes)
```

```
--- push_back 3, 4, 5 ---
```

```
[Allocator] 配置 4 個 i (16 bytes)
```

```
[Allocator] 釋放 2 個 i (8 bytes)
```

```
[Allocator] 配置 8 個 i (32 bytes)
```

```
[Allocator] 釋放 4 個 i (16 bytes)
```

```
--- 離開作用域 ---
```

```
[Allocator] 釋放 8 個 i (32 bytes)
```

```
=== 統計 ===
```

```
總共配置：60 bytes
```

```
總共釋放：60 bytes
```

這展示了 vector 如何在容量不足時重新配置更大的記憶體。

1.7 Allocator 使用時機總結

何時需要自訂 Allocator？	
大多數情況：不需要，預設的 <code>std::allocator</code> 就夠了	
可能需要的場景：	
1. 記憶體池 (Memory Pool)	
→ 避免頻繁的 <code>malloc/free</code> ，提高效能	
2. 追蹤記憶體使用	

	→ 除錯、效能分析	
	3. 特殊記憶體區域	
	→ 共享記憶體、GPU 記憶體、對齊要求	
	4. 嵌入式系統	
	→ 固定大小的記憶體區塊	
	5. 遊戲開發	
	→ 每幀重置的暫存配置器	

第二部分：std::bind

2.1 什麼是 std::bind？

std::bind 是一個函數配接器，可以：

1. 綁定參數：預先填入函數的某些參數
2. 調整參數順序：改變參數的位置
3. 產生新的可呼叫物件：把函數「包裝」成另一個函數

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

// 一個普通函數
int add(int a, int b) {
    return a + b;
}

int main() {
    // 不用 bind 的呼叫
    std::cout << "add(3, 5) = " << add(3, 5) << std::endl;
    // 用 bind 綁定第一個參數為 10
    auto add_10 = std::bind(add, 10, std::placeholders::_1);
    // 現在 add_10 只需要一個參數
    std::cout << "add_10(5) = " << add_10(5) << std::endl; // 10 + 5
    std::cout << "add_10(20) = " << add_10(20) << std::endl; // 10 + 20
    // 綁定兩個參數
```

```

auto always_15 = std::bind(add, 10, 5);
std::cout << "always_15() = " << always_15() << std::endl; // 10 + 5
return 0;
}

```

輸出：

```

add(3, 5) = 8
add_10(5) = 15
add_10(20) = 30
always_15() = 15

```

2.2 placeholders (佔位符) 詳解

`std::placeholders::_1`、`_2`、`_3` ... 代表「呼叫時傳入的第 1、2、3... 個參數」：

```

#include <iostream>
#include <functional>

void show_three(int a, int b, int c) {
    std::cout << "a=" << a << ", b=" << b << ", c=" << c << std::endl;
}

int main() {
    using namespace std::placeholders;
    std::cout << "=== 原始函數 ===" << std::endl;
    show_three(1, 2, 3);
    std::cout << "\n=== 綁定第一個參數為 100 ===" << std::endl;
    auto f1 = std::bind(show_three, 100, _1, _2);
    f1(2, 3); // show_three(100, 2, 3)
    f1(20, 30); // show_three(100, 20, 30)
    std::cout << "\n=== 綁定中間的參數為 200 ===" << std::endl;
    auto f2 = std::bind(show_three, _1, 200, _2);
    f2(1, 3); // show_three(1, 200, 3)
    std::cout << "\n=== 交換參數順序 ===" << std::endl;
    auto f3 = std::bind(show_three, _3, _1, _2);
    f3(1, 2, 3); // show_three(3, 1, 2)
    std::cout << "\n=== 重複使用同一個參數 ===" << std::endl;
    auto f4 = std::bind(show_three, _1, _1, _1);
    f4(7); // show_three(7, 7, 7)
}

```

```
}
```

輸出：

=== 原始函數 ===

a=1, b=2, c=3

=== 綁定第一個參數為 100 ===

a=100, b=2, c=3

a=100, b=20, c=30

=== 綁定中間的參數為 200 ===

a=1, b=200, c=3

=== 交換參數順序 ===

a=3, b=1, c=2

=== 重複使用同一個參數 ===

a=7, b=7, c=7

2.3 bind 與成員函數

綁定類別的成員函數時，需要額外提供物件：

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

class Calculator {
public:
    int value;
    Calculator(int v) : value(v) {}
    int add(int x) const {
        return value + x;
    }
    int multiply(int x) const {
        return value * x;
    }
};

int main() {
    using namespace std::placeholders;
```

```

Calculator calc(10);
// 綁定成員函數
// 第一個參數是成員函數指標
// 第二個參數是物件（或物件指標）
// 之後是函數參數
auto add_to_10 = std::bind(&Calculator::add, &calc, _1);
std::cout << "10 + 5 = " << add_to_10(5) << std::endl;
std::cout << "10 + 20 = " << add_to_10(20) << std::endl;
auto multiply_10 = std::bind(&Calculator::multiply, &calc, _1);
std::cout << "10 * 3 = " << multiply_10(3) << std::endl;
// 也可以綁定固定值
auto always_add_5 = std::bind(&Calculator::add, &calc, 5);
std::cout << "always_add_5() = " << always_add_5() << std::endl;
return 0;
}

```

輸出：

```

10 + 5 = 15
10 + 20 = 30
10 * 3 = 30
always_add_5() = 15

```

2.4 bind 與 STL 演算法

這是 `std::bind` 最常見的用途——與 STL 演算法搭配：

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

int main() {
using namespace std::placeholders;
std::vector<int> numbers = {1, 5, 10, 15, 20, 25, 30};
// 找出所有大於 12 的元素個數
// std::greater<int>() 需要兩個參數：greater(a, b) 返回 a > b
// 我們要 x > 12，所以綁定第二個參數為 12
auto greater_than_12 = std::bind(std::greater<int>(), _1, 12);
int count = std::count_if(numbers.begin(), numbers.end(), greater_than_12);
std::cout << "大於 12 的元素個數： " << count << std::endl;
}

```

```

// 找出所有小於 20 的元素
auto less_than_20 = std::bind(std::less<int>(), _1, 20);
std::cout << "小於 20 的元素: ";
for (int n : numbers) {
    if (less_than_20(n)) {
        std::cout << n << " ";
    }
}
std::cout << std::endl;
// 找出 10 到 25 之間的元素
auto in_range = [](int n) {
    return n >= 10 && n <= 25;
};
std::cout << "10 到 25 之間: ";
for (int n : numbers) {
    if (in_range(n)) {
        std::cout << n << " ";
    }
}
std::cout << std::endl;
return 0;
}

```

輸出：

```

大於 12 的元素個數: 4
小於 20 的元素: 1 5 10 15
10 到 25 之間: 10 15 20 25

```

2.5 bind vs Lambda：對比

現代 C++ 中，Lambda 通常是更好的選擇：

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

int main() {
    using namespace std::placeholders;
    std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
}

```

```

int threshold = 5;
// ===== 用 bind =====
auto greater_than_threshold_bind = std::bind(
std::greater<int>(), _1, threshold
);
int count_bind = std::count_if(
vec.begin(), vec.end(),
greater_than_threshold_bind
);
// ===== 用 Lambda =====
auto greater_than_threshold_lambda = [threshold](int n) {
return n > threshold;
};
int count_lambda = std::count_if(
vec.begin(), vec.end(),
greater_than_threshold_lambda
);
// ===== 更簡潔的 Lambda (直接內聯) =====
int count_inline = std::count_if(
vec.begin(), vec.end(),
[threshold](int n) { return n > threshold; }
);
std::cout << "bind 結果: " << count_bind << std::endl;
std::cout << "lambda 結果: " << count_lambda << std::endl;
std::cout << "inline lambda 結果: " << count_inline << std::endl;
return 0;
}

```

輸出：

```

bind 結果: 5
lambda 結果: 5
inline lambda 結果: 5

```

2.6 bind vs Lambda 比較表

bind vs Lambda 比較			
面向	std::bind	Lambda	

可讀性	較差	較好	
彈性	有限	完全自由	
複雜邏輯	很難表達	容易表達	
編譯器優化	可能較難優化	容易優化（通常內聯）	
錯誤訊息	很難讀	較易讀	
C++ 標準	C++11	C++11	
現代 C++ 推薦	不推薦	推薦	

2.7 何時還需要用 bind？

雖然 Lambda 通常更好，但有幾種情況 bind 可能有用：

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

class EventHandler {
public:
    void on_click(int x, int y) {
        std::cout << "Click at (" << x << ", " << y << ")" << std::endl;
    }
    void on_key(char key) {
        std::cout << "Key pressed: " << key << std::endl;
    }
};

int main() {
    using namespace std::placeholders;
    EventHandler handler;
    // 場景：需要把成員函數存到 std::function
    // 方法一：bind
    std::function<void(int, int)> click_handler_bind =
        std::bind(&EventHandler::on_click, &handler, _1, _2);
    // 方法二：Lambda（也可以，但稍微冗長）
```

```

std::function<void(int, int)> click_handler_lambda =
[&handler](int x, int y) { handler.on_click(x, y); };
// 兩者都可以正常呼叫
click_handler_bind(100, 200);
click_handler_lambda(300, 400);
return 0;
}

```

輸出：

```

Click at (100, 200)
Click at (300, 400)

```

2.8 std::bind 使用建議

std::bind 使用建議
<p>現代 C++ (C++11 以後) 的建議：</p> <ul style="list-style-type: none"> ✓ 優先使用 Lambda <ul style="list-style-type: none"> → 更清晰、更靈活、編譯器更容易優化 ✓ 了解 bind 的概念 <ul style="list-style-type: none"> → 因為你會在舊程式碼中遇到它 → 理解它有助於理解函數式編程概念 ✗ 不要在新程式碼中使用 bind <ul style="list-style-type: none"> → 幾乎所有 bind 能做的事，Lambda 都能做得更好 <p>例外情況：</p> <ul style="list-style-type: none"> • 極簡單的參數綁定（如 <code>bind(func, 10, _1)</code>） • 與某些框架的相容性需求

總結

本課總結 |

|
|
| Allocator (配置器): |

- 負責容器的記憶體配置與釋放 |
- 分離「配置記憶體」和「建構物件」兩個步驟 |
- 大多數情況使用預設的 `std::allocator` |
- 自訂 `allocator` 用於特殊記憶體管理需求 |

|
|
| `std::bind`: |

- 綁定函數參數，產生新的可呼叫物件 |
- 使用 `placeholders (_1, _2...)` 代表未綁定的參數 |
- 可以綁定普通函數和成員函數 |
- 現代 C++ 建議用 `Lambda` 取代 |

|
|
| 重要觀念: |

- 這兩個工具都是 `STL` 底層機制的一部分 |
- 理解它們有助於深入理解 `STL` 的設計 |
- 日常使用中，你很少需要直接用到它們 |

還有其他關於第三課的問題嗎？或者準備好進入**第四課：迭代器 (Iterator)** 的核心概念了？