Section 20 STL

🌲 C++ STL 概述

STL 是 C++ 提供的一套**泛型資料結構與演算法工具庫**，支援：

1. **容器 Containers**：用來儲存資料
2. **演算法 Algorithms**：操作容器中的資料（如排序、搜尋等）
3. **疊代器 Iterators**：在容器中移動與存取資料的工具
4. **函式物件 Function Objects（函數物件）**
5. **配接器 Adapters**：變更容器或函式的行為
6. **配接器記憶體配置器 Allocators（進階）**

STL 的核心精神是「**泛型程式設計**」（Generic Programming）：使用模板（template）來寫不依賴特定型別的程式。

🧩 什麼是 Function Template？

Function Template 是一種允許函式在**不指定具體資料型別**下撰寫的機制，透過 C++ 的模板（template）功能來實現 **泛型函式**。

👉 它的目的是「**讓函式可以作用於多種型別**」，而不用為每個型別都手動重複寫一次。

🧪 基本語法

```
template <typename T>
T functionName(T a, T b) {
    // 使用型別 T 撰寫通用邏輯
}
```

或：

```
template <class T>  // class 和 typename 等價
T functionName(T a, T b) {
    // ...
}
```

範例：取得兩數的最大值

```
#include <iostream>
using namespace std;

template <typename T>
T getMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << getMax(10, 20) << endl;      // int
    cout << getMax(3.14, 2.71) << endl;  // double
    cout << getMax('a', 'z') << endl;    // char
}
```

🔗 函式模板的推導（Type Deduction）

你通常 **不需要手動指定型別**，編譯器會自動從參數推斷模板型別：

```
getMax(10, 20);   // 推斷 T 為 int
getMax(1.5, 3.2); // 推斷 T 為 double
```

但你也**可以手動指定型別**：

```
getMax<double>(2, 3); // 將 int 轉成 double，強制使用 double 模板
```

🌈 多個模板參數

```
template <typename T1, typename T2>
class Pair {
   T1 first;
   T2 second;
public:
   Pair(T1 f, T2 s) : first(f), second(s) {}
   void print() { std::cout << first << " - " << second << '\n'; }
};

Pair<int, string> p(1, "apple");
```

⚠ 函式模板與 Overload（多載）並存

函式模板和普通函式可以共存，編譯器會根據「最符合的」來選擇：

```
int getMax(int a, int b) {
   cout << "Regular function\n";
   return (a > b) ? a : b;
}

template <typename T>
T getMax(T a, T b) {
   cout << "Template function\n";
   return (a > b) ? a : b;
}

int main() {
   getMax(3, 4);    // 呼叫 regular 版本（完全匹配）
   getMax(3.0, 4.0); // 呼叫 template 版本
}
```

🛠 限制型別：`requires`（C++20）或 `enable_if`（C++11）

例如限制只能對可比較大小的型別使用：

```
#include <concepts>

template <typename T>
requires std::totally_ordered<T>
T getMax(T a, T b) {
   return (a > b) ? a : b;
}
```

🧿 模板實例化（Instantiation）

當你呼叫一個模板函式時，編譯器會「產生」對應型別的版本：

```
getMax(1, 2);      // 產生 getMax<int>(int, int)
getMax(1.5, 2.5);  // 產生 getMax<double>(double, double)
```

這叫做 **template instantiation**。

🧵 小結：Function Template 的優點

優點	說明
節省程式碼重複	寫一次就能支援多種型別
增強型別安全	編譯時檢查型別錯誤
易於維護	像標準函式 `std::swap`、`std::max` 就是模板寫成的

🏗️ 什麼是 Class Template？

Class Template 是 C++ 提供的一種機制，讓你可以為任意型別建立「通用類別」，寫一份類別程式碼，就可以對不同型別的資料使用。

✅ 目的

類似於函式模板的「型別泛化」，但針對的是**類別層級**。

📐 基本語法

```
template <typename T>
class ClassName {
public:
    T data;

    ClassName(T val) : data(val) {}

    void display() {
        std::cout << data << std::endl;
    }
};
```

🧪 使用範例

```
#include <iostream>
using namespace std;

template <typename T>
class Box {
    T value;
public:
    Box(T val) : value(val) {}
    void show() const {
        cout << "Value: " << value << endl;
    }
};

int main() {
    Box<int> intBox(123);
    Box<string> strBox("Hello");

    intBox.show();   // Value: 123
    strBox.show();   // Value: Hello
}
```

🧬 使用模板參數的成員函式

你可以在類別內部使用模板型別 `T`，也可以定義模板成員函式：

```
template <typename T>
class Wrapper {
  T item;
public:
  Wrapper(T val) : item(val) {}

  void print() {
    std::cout << "Item = " << item << '\n';
  }

  T get() {
    return item;
  }
};
```

🧠 模板類別的實例化（Instantiation）

當你使用 `Box<int>` 時，編譯器就會根據模板產生一個專門的類別 `Box<int>`。

你也可以顯式指定：

```
Box<double> b(3.14);
```

🔗 分離宣告與定義（在 `.h` / `.cpp` 分開）

❌ 錯誤方式（這會造成 linker 錯誤）

Box.h

```
template <typename T>
class Box {
public:
    Box(T val);
    void show();
};
```

Box.cpp

```
template <typename T>
Box<T>::Box(T val) { ... } // 錯誤：無法編譯模板定義於 .cpp
```

✅ 正確方式（都寫在 `.h`）

因為 template 是在編譯期實體化的，**必須提供完整定義**。

```
// Box.h
template <typename T>
class Box {
    T value;
public:
    Box(T val) : value(val) {}
    void show() const { std::cout << value << std::endl; }
};
```

🧩 Class Template 與 Function Template 比較

功能	Function Template	Class Template
作用範圍	單一函式	整個類別
常見應用	`max`, `swap`	`vector`, `stack`, `map` 等 STL 類別
實例化時間	呼叫時	宣告時
模板參數使用位置	函式參數與回傳型別	成員變數與函式

🎨 Template Specialization（模板特化）

🧊 全特化：針對特定型別寫特殊版本

```
template <>
class Box<string> {
public:
    Box(string s) { std::cout << "Specialized: " << s << std::endl; }
};
```

🧵 小結：Class Template 的優點

優點	說明
泛型設計	支援任意型別
重複利用	可減少撰寫多型別版本的類別
類似 STL 類別結構	可模擬 `vector<T>`，`stack<T>` 等結構

📦 STL 中的例子：vector 就是 class template

```
std::vector<int> v1;
std::vector<double> v2;
std::vector<std::string> v3;
```

🧩 什麼是 Generic Array class？

「Generic Array class」是一個**可以存任意型別的類別化陣列容器**，透過 template 實作，像這樣使用：

```
Array<int> intArr(5);    // 建立一個 int 型態的陣列
Array<string> strArr(3); // 建立一個 string 型態的陣列
```

🏗️ 範例實作：簡易版 Generic Array class

🔧 功能目標

- 建立陣列
- 設定元素、取得元素
- 自動管理記憶體（避免記憶體洩漏）

- 支援索引運算子（operator[]）

🧪 程式碼

```
#include <iostream>
using namespace std;

template <typename T>
class Array {
private:
    T* data;
    size_t size;

public:
    // Constructor
    Array(size_t n) : size(n) {
        data = new T[n];
    }

    // Destructor
    ~Array() {
        delete[] data;
    }

    // Copy constructor
    Array(const Array& other) : size(other.size) {
        data = new T[size];
        for (size_t i = 0; i < size; ++i)
            data[i] = other.data[i];
    }

    // Assignment operator
    Array& operator=(const Array& other) {
        if (this == &other) return *this; // 防止自己指派自己
        delete[] data;
        size = other.size;
        data = new T[size];
```

```
        for (size_t i = 0; i < size; ++i)
            data[i] = other.data[i];
        return *this;
    }

    // 索引運算子
    T& operator[size_t index](size_t%20index) {
        if (index >= size) throw out_of_range("Index out of range");
        return data[index];
    }

    // const 版本
    const T& operator[size_t index](size_t%20index) const {
        if (index >= size) throw out_of_range("Index out of range");
        return data[index];
    }

    // 取得大小
    size_t getSize() const { return size; }
};
```

使用範例

```
int main() {
    Array<int> nums(5);
    for (int i = 0; i < nums.getSize(); ++i)
        nums[i] = i * i;

    for (int i = 0; i < nums.getSize(); ++i)
        cout << nums[i] << " ";
    cout << endl;

    Array<string> words(3);
    words[0] = "Hello";
    words[1] = "World";
    words[2] = "!";

    for (int i = 0; i < words.getSize(); ++i)
```

```
    cout << words[i] << " ";
}
```

✨ 可加強的功能（進階）

功能	技術
範圍檢查	自訂錯誤處理或例外拋出
動態調整大小	加入 `resize()` 或改為 vector-like
疊代器支援	提供 `begin()`，`end()`
移動建構子與移動指派	C++11 `&&` 搭配 `std::move()`
初始化列表建構子	`Array(std::initializer_list<T>)`

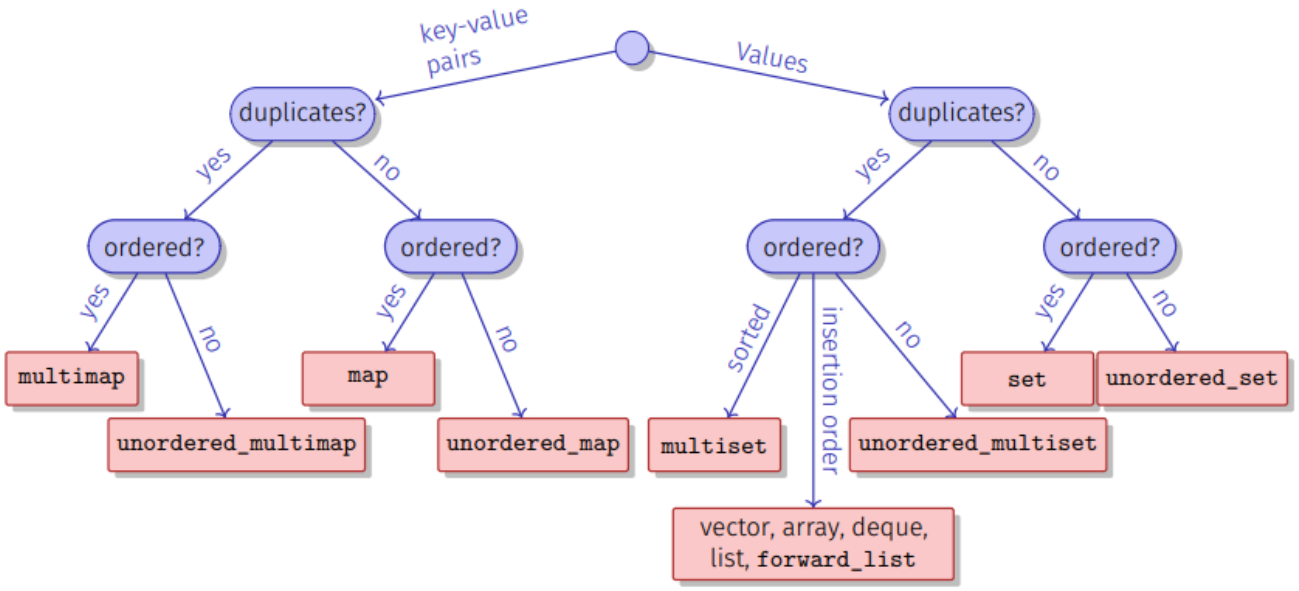
🧠 與 `std::vector` 的比較

特性	Generic Array class	`std::vector`
記憶體管理	手動 `new/delete`	自動處理
動態擴展	不支援（可手動加）	支援
操作簡易性	學習用	生產環境推薦
效率	好控制	最佳化已完成

📘 小結

- Generic Array class 是訓練 **template**、**記憶體管理**、**物件導向設計**的好例子。
- 你可以從這個簡單的例子延伸出：
  - **自訂容器（如 Stack、Queue）**
  - **理解 STL 的底層結構**
  - **學習 C++ 11/14/17 的現代技術如 move semantics、initializer_list**

📦 STL 容器 Containers



🔢 Types of STL Containers

STL 容器分為以下四大類型，每一類都依據其儲存與操作方式設計，以適用不同的資料結構需求。

📚 Sequence Containers（序列式容器）

用途：維持元素插入順序，類似陣列或串列。
特性：元素的排列順序依插入位置而定，可透過索引或 iterator 訪問。

容器名稱	特點簡述
`vector`	動態大小陣列，支援隨機存取，擴展效率佳
`deque`	雙端佇列，支援前後插入刪除，亦支援隨機存取
`list`	雙向鏈結串列，支援中間插入刪除但不支援隨機存取
`forward_list`	單向鏈結串列，記憶體使用更省但功能較少
`array`	固定大小的靜態陣列容器
`string`	字元序列的封裝，類似 `vector<char>`（實際上是個特殊容器）

🔗 **典型使用場景**：需要頻繁訪問（如 `vector`）、頻繁插入刪除（如 `list`）

🗂️ Associative Containers（關聯式容器）

用途：以鍵值（key）為基礎排序與存取元素。
特性：內部以平衡二元搜尋樹（如 Red-Black Tree）實作，會自動排序。

容器名稱	特點簡述h
`set`	唯一且排序的鍵集合，元素即為 key
`multiset`	可重複元素的排序集合
`map`	鍵值對 (key-value pair)，key 唯一，會自動排序
`multimap`	可重複 key 的鍵值對集合

🔍 **典型使用場景**：需要自動排序與快速查找元素的場景。

🌐 Unordered Containers（非排序關聯式容器）

用途：提供基於雜湊（hash）的快速查找，不保證元素順序。
特性：內部以 Hash Table 實作，平均 O(1) 查找與插入效率。

容器名稱	特點簡述
`unordered_set`	唯一鍵的 hash-based 集合
`unordered_multiset`	可重複鍵的 hash-based 集合
`unordered_map`	鍵值對集合，key 唯一
`unordered_multimap`	可重複鍵的鍵值對集合

⚡ **典型使用場景**：需要最快速查找且不在乎順序的情況。

📦 Container Adapters（容器配接器）

用途：提供特定抽象資料結構（如 stack、queue）的接口，實際以其他容器為底層實作。
特性：限制介面（只能 push/pop/top 等），以 `deque` 或 `vector` 作為底層儲存。

容器名稱	特點簡述
stack	後進先出（LIFO），僅可從頂端操作
queue	先進先出（FIFO），從尾部插入、前端取出
priority_queue	元素自動排序的 queue，預設為大根堆（最大值優先）

🔁 **典型使用場景**：需要模擬資料結構，如 DFS 使用 stack 、BFS 使用 queue 、模擬事件處理使用 priority_queue 。

📚 Sequence Containers

📦 std::vector — 動態陣列容器

📘 概要

std::vector 是一種動態大小的一維陣列容器，支援隨機存取（random access），元素會按照插入順序排列。它的大小可在執行時根據需求自動擴張。

🧱 內部實作

- 底層是連續記憶體區段（contiguous memory），類似 C-style 陣列。
- 當超出容量時，會重新配置更大的記憶體並搬移舊元素。

🔧 常見操作與成員函式

方法	說明
vector<T> v;	建立空的 vector
vector<T> v(n);	建立有 n 個預設值的 vector
v.push_back(val);	將 val 加入 vector 末尾
v.pop_back();	移除最後一個元素
v.size();	回傳目前元素個數
v.capacity();	回傳目前記憶體配置容量
v.reserve(n);	提前配置至少 n 個元素的空間
v.resize(n);	調整元素個數為 n，可能會加新元素或移除尾部
v.clear();	移除所有元素但不釋放記憶體

方法	說明
`v.empty();`	回傳是否為空容器
`v[i]` / `at(i)`	存取第 i 個元素，`at(i)` 有界限檢查
`v.begin()`, `v.end()`	iterator 起始與結尾
`v.insert(pos, val)`	插入元素到指定 iterator 位置
`v.erase(pos)`	移除指定 iterator 的元素

📌 範例程式碼

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

int main() {
    std::vector<int> nums;

    // 加入元素
    nums.push_back(10);
    nums.push_back(20);
    nums.push_back(30);

    // 使用索引訪問
    std::cout << "第二個元素: " << nums[1] << '\n';

    // 使用範圍 for loop
    std::cout << "所有元素: ";
    for (int x : nums)
        std::cout << x << " ";
    std::cout << '\n';

    // 移除最後一個元素
    nums.pop_back();

    // 查看大小與容量
    std::cout << "大小: " << nums.size() << ", 容量: " << nums.capacity() << '\n';

    // 插入與刪除
    nums.insert(nums.begin() + 1, 99);  // 在第二個位置插入 99
```

```
    nums.erase(nums.begin());       // 移除第一個元素

    // 輸出修改後
    std::cout << "修改後: ";
    for (int x : nums)
        std::cout << x << " ";
}
```

🚀 適用情境

- 須頻繁**隨機存取**元素
- 元素數量**會動態變化**且通常新增在末尾
- 追求**緊湊、高效的記憶體配置**

⚠️ 注意事項

- **插入或刪除前面的元素**效率差，因為會導致所有後面元素搬移。
- 重新配置（reallocation）會**使所有 iterator 與指標失效**。

📦 `std::deque` — 雙端佇列容器（Double-Ended Queue）

📘 概要

`std::deque`（**Double-Ended QUEue**）是支援兩端插入與刪除的容器，類似 `vector`，但在前端操作效率更好。它同樣支援隨機存取。

🧱 內部實作

不像 `vector` 使用連續記憶體，`deque` 是一組指向多個固定大小區塊的陣列，實現「彈性前後擴張」的效果。

🔧 常見操作與成員函式

方法	說明
`deque<T> d;`	建立空的 deque
`d.push_back(val);`	在尾端加入元素

方法	說明
d.push_front(val);	在前端加入元素
d.pop_back();	移除尾端元素
d.pop_front();	移除前端元素
d.front(); / d.back();	存取最前/最後一個元素
d[i] / d.at(i)	隨機存取元素（O(1)）
d.size();	回傳元素個數
d.clear();	清空所有元素
d.empty();	回傳是否為空
d.insert(pos, val)	插入元素到指定 iterator 位置
d.erase(pos)	移除指定 iterator 的元素
d.begin(), d.end()	Iterator 起迄位置

📌 範例程式碼

```
#include <iostream>
#include <deque>

int main() {
    std::deque<int> dq;

    // 前後插入
    dq.push_back(10);
    dq.push_front(5);
    dq.push_back(20);

    // 現在順序為 5 10 20
    std::cout << "目前內容: ";
    for (int x : dq)
        std::cout << x << " ";
    std::cout << '\n';

    // 移除兩端
    dq.pop_front(); // 移除 5
    dq.pop_back();  // 移除 20
```

```
    std::cout << "中間只剩: " << dq.front() << '\n'; // 印出 10

    // 插入與刪除
    dq.push_back(99);
    dq.insert(dq.begin() + 1, 42); // 在中間插入

    std::cout << "插入後內容: ";
    for (int x : dq)
        std::cout << x << " ";
}
```

🚀 適用情境

- 需要在**前後都能高效插入/刪除**的場景（如：模擬滑動視窗、BFS）
- 比 `list` 更省空間，且仍提供常數時間的前後操作

⚠️ 注意事項

- 雖支援隨機存取，但其記憶體不是連續配置，不適用於需要原始指標的函式（如 `memcpy`）。
- 相比 `vector`，不保證 cache locality（記憶體連續性），對某些應用效能略差。

🔗 `std::list` — 雙向鏈結串列容器（Doubly Linked List）

📘 概要

`std::list` 是一種**雙向鏈結串列**，每個節點包含資料與前後指標，允許在任何位置**常數時間插入與刪除**。
它不支援隨機存取（無法用 `[]` 來訪問第 i 個元素）。

📦 內部實作

每個元素（節點）都有指向前一個與下一個元素的指標，便於雙向走訪與修改。不同於 `vector` 或 `deque`，其元素**不連續配置**。

🔧 常見操作與成員函式

方法	說明
`list<T> lst;`	建立空的 list
`lst.push_back(val);`	在尾端插入元素
`lst.push_front(val);`	在前端插入元素
`lst.pop_back();` / `lst.pop_front();`	移除尾端 / 前端元素
`lst.front();` / `lst.back();`	存取最前 / 最後元素
`lst.insert(pos, val);`	插入元素到指定 iterator 前
`lst.erase(pos);`	移除指定 iterator 的元素
`lst.begin(), lst.end()`	Iterator 起迄位置
`lst.rbegin(), lst.rend()`	反向 iterator
`lst.sort();`	就地排序（用 operator<）
`lst.reverse();`	反轉元素順序
`lst.remove(val);`	移除所有與 `val` 相等的元素
`lst.unique();`	移除連續重複元素（排序後使用）
`lst.merge(otherList);`	合併已排序的兩個 list

📌 範例程式碼

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> lst;

    // 插入元素
    lst.push_back(10);
    lst.push_front(5);
    lst.push_back(20);

    std::cout << "目前內容: ";
    for (int x : lst)
        std::cout << x << " ";
    std::cout << '\n';
```

```
    // 插入在第二個位置
    auto it = lst.begin();
    ++it; // 移到第二個元素
    lst.insert(it, 99); // 插在 10 前面

    std::cout << "插入後: ";
    for (int x : lst)
        std::cout << x << " ";
    std::cout << '\n';

    // 排序與反轉
    lst.sort();
    lst.reverse();

    std::cout << "排序反轉後: ";
    for (int x : lst)
        std::cout << x << " ";
}
```

🚀 適用情境

- 經常需要在**中間插入或刪除元素**的場景
- 資料量大、排序與去重需求多（搭配 `sort()` 與 `unique()`）

⚠️ 注意事項

- 無法透過 `lst[i]` 訪問特定位置（只能使用 iterator 慢慢走）
- 相比 `vector`，**每個節點額外多佔兩個指標空間**
- 無法與 C-style API（如 `memcpy`）配合

🧵 `std::forward_list` — 單向鏈結串列容器（Singly Linked List）

📘 概要

`std::forward_list` 是一種**單向鏈結串列**，每個節點只有一個指向「下一個節點」的指標。它是 C++11 新增的容器，設計目的是為了**節省記憶體空間**與提高某些場景下的

效率。

內部實作

每個元素只持有 `next` 指標（不像 `list` 有 `prev` 和 `next`），不支援尾端操作，也不支援雙向走訪。

常見操作與成員函式

方法	說明
`forward_list<T> fl;`	建立空的 forward_list
`fl.push_front(val);`	加入元素到前端
`fl.pop_front();`	移除前端元素
`fl.front();`	取出最前端元素
`fl.insert_after(pos, val);`	在 `pos` 後插入元素
`fl.erase_after(pos);`	移除 `pos` 後的元素
`fl.before_begin()`	特殊 iterator：用於插入第一個元素之前
`fl.begin()`, `fl.end()`	正常走訪用
`fl.sort();`	排序元素
`fl.reverse();`	反轉元素順序
`fl.remove(val);`	移除所有與 val 相等的元素
`fl.unique();`	移除連續重複元素
`fl.merge(other);`	合併已排序的 forward_list

範例程式碼

```cpp
#include <iostream>
#include <forward_list>

int main() {
    std::forward_list<int> fl = {3, 6, 9};

    // 在前面插入
    fl.push_front(1); // [1, 3, 6, 9]

    // 插入在第一個元素後
```

```
    auto it = fl.begin(); // 指向 1
    fl.insert_after(it, 2);  // [1, 2, 3, 6, 9]

    // 移除某個元素後的節點
    fl.erase_after(it); // [1, 2, 6, 9]

    // 遍歷
    std::cout << "內容: ";
    for (int x : fl)
        std::cout << x << " ";
    std::cout << '\n';

    // 排序與反轉
    fl.sort();
    fl.reverse();

    std::cout << "排序反轉後: ";
    for (int x : fl)
        std::cout << x << " ";
}
```

🚀 適用情境

- 節省記憶體空間（比 `list` 少一個指標）
- 頻繁進行**前端插入或刪除**
- 只需要**單向走訪**

⚠️ 注意事項

- 沒有 `push_back()` 或 `pop_back()`（只能操作前端）
- 沒有 `size()` 成員函式（你要手動計算）
- 插入刪除時需使用 `insert_after` / `erase_after`，稍微不直覺
- iterator 無法倒退（不像 `list` 有 bidirectional iterator）

📏 `std::array` — 固定大小的靜態陣列容器（C++11 起）

📘 概要

`std::array` 是一種 **固定大小、靜態配置** 的容器，提供 C-style 陣列的記憶體效率與 STL 容器的介面。
大小必須在**編譯期間**決定，不能動態調整。

```
#include <array>
std::array<int, 5> arr; // 宣告一個有 5 個 int 的 array
```

🧱 內部實作

本質上就是包裝 C-style 陣列 (`T[N]`) 並提供：

- STL 相容的 iterator
- 成員函式（如 `at()` 、 `size()` ）
- 拷貝與賦值操作（完整 value semantics）

🔧 常見操作與成員函式

方法 / 成員	說明
`array<T, N>`	宣告固定大小為 `N` 的陣列
`a[i]` / `a.at(i)`	存取第 `i` 個元素； `at()` 會檢查越界
`a.front()` / `a.back()`	存取第一或最後一個元素
`a.fill(val)`	將所有元素設為 `val`
`a.size()`	傳回元素個數（永遠是 N）
`a.begin()` / `a.end()`	iterator 起迄位置
`a.data()`	傳回底層原始陣列指標 `T*` （可與 C API 互通）
`a.swap(b)`	與另一個同型別 `array` 交換內容

📌 範例程式碼

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> arr = {1, 2, 3, 4, 5};
```

```
    // 訪問元素
    std::cout << "第一個元素: " << arr.front() << '\n';
    std::cout << "最後一個元素: " << arr.back() << '\n';

    // 改變內容
    arr[2] = 99;

    // 使用範圍 for loop
    std::cout << "所有內容: ";
    for (int x : arr)
        std::cout << x << " ";
    std::cout << '\n';

    // 使用 fill()
    arr.fill(0);
    std::cout << "全部清為 0: ";
    for (int x : arr)
        std::cout << x << " ";
}
```

🚀 適用情境

- 已知大小且不需變動的序列資料（例如 RGB、座標、固定大小表格等）
- C-style 陣列需要安全封裝與 STL 相容介面
- 搭配 template 用於編譯期最佳化場景（靜態配置更快）

⚠️ 注意事項

- **大小固定，不可擴張**，否則請用 `vector`
- 元素數量需於 **編譯時指定**
- 雖為 STL 容器，但不是動態資料結構（不含記憶體管理邏輯）

🔐 Associative Containers（關聯式容器）

🌲 `std::set` — 唯一且自動排序的集合容器

📘 概要

`std::set` 是一個**關聯式容器**，用來儲存**唯一元素**，且會根據元素大小**自動排序（預設用 `operator<`）**。
其底層是 **紅黑樹（Red-Black Tree）**，所以所有操作時間複雜度皆為 O(log n)。

```
#include <set>
std::set<int> s;
```

🧱 內部結構

- 使用**平衡二元搜尋樹**維持排序（通常為紅黑樹）
- 插入與查找皆為 O(log n)
- 所有元素皆視為「key」，因此不可重複（`==` 的元素只能出現一次）

🔧 常見操作與成員函式

方法 / 成員	說明
`s.insert(val)`	插入元素，若已存在則忽略
`s.erase(val)`	移除指定值的元素
`s.erase(it)`	移除指定 iterator 的元素
`s.find(val)`	回傳指向該值的 iterator，若不存在則為 `end()`
`s.count(val)`	回傳該值的出現次數（對 `set` 而言為 0 或 1）
`s.lower_bound(val)`	回傳第一個 ≥ val 的 iterator
`s.upper_bound(val)`	回傳第一個 > val 的 iterator
`s.begin()`, `s.end()`	取得正向 iterator
`s.rbegin()`, `s.rend()`	取得反向 iterator
`s.empty()` / `s.size()`	是否為空、元素數量
`s.clear()`	清空集合

📌 範例程式碼

```
#include <iostream>
#include <set>
```

```cpp
int main() {
    std::set<int> s;

    // 插入元素
    s.insert(3);
    s.insert(1);
    s.insert(4);
    s.insert(1); // 重複元素被忽略

    std::cout << "排序後內容: ";
    for (int x : s)
        std::cout << x << " ";
    std::cout << '\n';  // 1 3 4

    // 查找
    if (s.count(3))
        std::cout << "找到 3\n";

    // 使用 lower_bound / upper_bound
    auto it = s.lower_bound(2); // >= 2 的第一個
    if (it != s.end())
        std::cout << "第一個 ≥ 2 的元素是: " << *it << '\n';

    // 移除元素
    s.erase(3);

    std::cout << "移除後內容: ";
    for (int x : s)
        std::cout << x << " ";
}
```

🧠 關於 `insert` 回傳值

```cpp
auto result = s.insert(10);
if (result.second) {
    std::cout << "插入成功！位置：" << *result.first << '\n';
} else {
```

```
    std::cout << "已存在於 set 中\n";
}
```

- `insert()` 回傳 `pair<iterator, bool>`，第二個值代表是否真的插入成功。

🚀 適用情境

- 需要維持元素**唯一性與自動排序**
- 查找、插入、刪除效率要求 O(log n)
- 可當作**去重過的排序集合**（如統計獨特輸入、字典、索引等）

⚠️ 注意事項

- 不支援 `operator[]`（因為沒有 value 的概念）
- 插入相同元素不會覆蓋也不會報錯，只是忽略
- 若需儲存重複元素，請改用 `multiset`

🌿 `std::multiset` — 允許重複元素的排序集合

📘 概要

`std::multiset` 和 `std::set` 幾乎一模一樣，唯一的差別是它**允許多個重複的元素**。
底層一樣使用 **紅黑樹（Red-Black Tree）**，操作效率為 O(log n)。

```
#include <set>
std::multiset<int> ms;
```

🧱 內部結構

- 元素自動排序（使用 `operator<`）
- 每個元素視為 key，可重複
- 資料結構為自動平衡的搜尋樹，插入與刪除皆為 O(log n)

🔧 常見操作與成員函式

方法 / 成員	說明
`ms.insert(val)`	插入元素（允許重複）

方法 / 成員	說明
`ms.erase(val)`	移除所有該值的元素
`ms.erase(it)`	移除指定位置的元素
`ms.find(val)`	回傳第一個等於該值的 iterator
`ms.count(val)`	回傳該值出現的次數
`ms.lower_bound(val)`	第一個 ≥ val 的 iterator
`ms.upper_bound(val)`	第一個 > val 的 iterator
`ms.equal_range(val)`	回傳 `pair<lower_bound, upper_bound>` 範圍
`ms.begin()`, `ms.end()`	正向 iterator
`ms.rbegin()`, `ms.rend()`	反向 iterator
`ms.empty()` / `ms.size()`	是否為空、元素個數
`ms.clear()`	清空容器

📌 範例程式碼

```cpp
#include <iostream>
#include <set>

int main() {
    std::multiset<int> ms;

    // 插入元素
    ms.insert(10);
    ms.insert(20);
    ms.insert(10);  // 重複插入
    ms.insert(30);

    std::cout << "內容: ";
    for (int x : ms)
        std::cout << x << " ";
    std::cout << '\n'; // 輸出: 10 10 20 30

    // 查找某元素的次數
    std::cout << "10 出現次數: " << ms.count(10) << '\n';
```

```
    // 使用 equal_range
    auto [begin, end] = ms.equal_range(10);
    std::cout << "10 的範圍: ";
    for (auto it = begin; it != end; ++it)
        std::cout << *it << " ";
    std::cout << '\n';

    // 移除單一個 10
    auto it = ms.find(10);
    if (it != ms.end())
        ms.erase(it);  // 只刪掉一個 10

    std::cout << "刪除一個 10 後: ";
    for (int x : ms)
        std::cout << x << " ";
}
```

🚀 適用情境

- 需要儲存重複元素並且**維持自動排序**
- 統計出現次數、維護有序資料（例如：多筆成績、同分者排序）
- 與 `map` 搭配可實作 multimap 功能的鍵集合

⚠ 注意事項

- **插入的所有元素皆會保留**，不會去重
- `erase(val)` 會刪除**所有**該值，若只想刪一個需用 `find()` + `erase(it)`
- 沒有 `operator[]`

📄 std::map — 唯一鍵的關聯式鍵值對容器

📘 概要

`std::map` 是一個儲存**鍵值對（key-value pairs）**的關聯式容器，
會根據 key 自動排序，並保證每個 key **唯一且僅出現一次**。
底層使用 **紅黑樹（Red-Black Tree）**，支援 O(log n) 插入、刪除與查找。

```
#include <map>
std::map<std::string, int> age;
```

🧱 資料結構與 pair

每個元素是 `std::pair<const Key, T>`，例如：

```
std::map<std::string, int> m;
m["Alice"] = 30;  // 插入 pair<const string, int>("Alice", 30)
```

- `key` 是第一個成員（不可重複）
- `value` 是第二個成員（可修改）

🔧 常見操作與成員函式

方法 / 成員	說明
`m[key] = val`	插入或更新 `key` 對應的值
`m.at(key)`	存取值，若 key 不存在會拋出例外
`m.insert({key, val})`	插入新的鍵值對（若 key 已存在則無效）
`m.emplace(key, val)`	原地建構鍵值對（效率更高）
`m.find(key)`	回傳 iterator 指向該 key，或 `end()`
`m.count(key)`	回傳該 key 是否存在（為 0 或 1）
`m.erase(key)` / `m.erase(it)`	移除元素（透過 key 或 iterator）
`m.lower_bound(key)`	第一個 ≥ key 的元素 iterator
`m.upper_bound(key)`	第一個 > key 的元素 iterator
`m.begin()`, `m.end()`	iterator 起迄位置（照 key 排序）
`m.empty()` / `m.size()` / `m.clear()`	通用成員函式

📌 範例程式碼

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

int main() {
```

```
    std::map<std::string, int> scores;

    // 插入與更新
    scores["Alice"] = 95;
    scores["Bob"] = 89;
    scores["Charlie"] = 77;

    // 更新 Bob 的分數
    scores["Bob"] = 92;

    // 使用迴圈輸出
    for (const auto& [name, score] : scores)
        std::cout << name << ": " << score << '\n';

    // 查找特定 key
    if (scores.count("Alice"))
        std::cout << "Alice 的分數是 " << scores.at("Alice") << '\n';

    // 移除 Charlie
    scores.erase("Charlie");

    std::cout << "目前剩下 " << scores.size() << " 筆資料。\n";
}
```

🧠 關於 `m[key]` 的隱含行為

```
std::map<std::string, int> m;
std::cout << m["nonexistent"] << '\n'; // ❗會自動插入 "nonexistent": 0
```

使用 `[]` 存取不存在的 key，**會自動插入該 key 並給一個預設值**。
若不想自動插入，請使用 `find()` 或 `at()`。

🚀 適用情境

- 鍵值唯一、需自動排序的資料（例如：統計字頻、查詢 ID 對應資訊）
- 需要高效 O(log n) 查找與插入
- 可用來模擬「dictionary」、「symbol table」、「phonebook」等結構

⚠ 注意事項

- 若需儲存重複 key，請使用 `multimap`
- 若不需要排序，追求更快效率，可改用 `unordered_map`
- 使用 `operator[]` 時要留意「**不小心插入不該有的 key**」問題

📚 `std::multimap` — 允許重複鍵的關聯式鍵值對容器

📘 概要

`std::multimap` 與 `std::map` 類似，都是儲存**鍵值對（key-value pairs）**，但允許多個元素擁有相同的 key。
底層一樣使用平衡二元搜尋樹（通常是紅黑樹），所以插入、查找、刪除的時間複雜度均為 O(log n)。

🔧 常見操作與成員函式

方法 / 成員	說明
`mm.insert({key, val})`	插入元素，允許重複鍵
`mm.emplace(key, val)`	原地建構插入元素
`mm.erase(key)`	移除所有匹配 key 的元素
`mm.erase(it)`	移除指定 iterator 的元素
`mm.find(key)`	找到第一個匹配 key 的 iterator
`mm.count(key)`	回傳該 key 出現的次數
`mm.lower_bound(key)`	第一個 ≥ key 的 iterator
`mm.upper_bound(key)`	第一個 > key 的 iterator
`mm.equal_range(key)`	回傳 pair(lower_bound, upper_bound)
`mm.begin()`, `mm.end()`	正向 iterator
`mm.rbegin()`, `mm.rend()`	反向 iterator

📌 範例程式碼

```cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::multimap<std::string, int> mm;

    mm.insert({"Alice", 90});
    mm.insert({"Bob", 85});
    mm.insert({"Alice", 95});  // 重複鍵

    std::cout << "內容:\n";
    for (const auto& [name, score] : mm) {
        std::cout << name << ": " << score << '\n';
    }

    // 查詢 Alice 的所有分數
    auto range = mm.equal_range("Alice");
    std::cout << "Alice 的分數:\n";
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << it->second << '\n';
    }

    // 刪除所有 Bob
    mm.erase("Bob");

    std::cout << "刪除 Bob 後內容:\n";
    for (const auto& [name, score] : mm) {
        std::cout << name << ": " << score << '\n';
    }
}
```

🚀 適用情境

- 需要同一 key 對應多個 value 的情況（例如：分類資料、多筆交易記錄）
- 需要排序且允許重複鍵的關聯資料儲存

⚠ 注意事項

- 不支援用 operator[]（因為 key 可重複，無法唯一索引）
- erase(key) 會刪除所有匹配的元素
- 插入時不會覆蓋現有資料

🌪️ Unordered Containers（非排序關聯式容器）

⚡ std::unordered_set — 唯一元素的哈希集合容器

📘 概要

std::unordered_set 是一種基於**哈希表（hash table）**實作的集合容器，
用來儲存**唯一元素**，但元素**不會自動排序**，而是依照哈希值分佈。
查找、插入、刪除的平均時間複雜度接近 O(1)。

🧱 內部結構

- 使用哈希函式 (std::hash<T>) 來計算元素位置
- 透過鏈結法或開放位址法解決哈希碰撞
- 不保證元素的遍歷順序，每次執行可能不同

🔧 常見操作與成員函式

方法 / 成員	說明
us.insert(val)	插入元素，重複元素會被忽略
us.erase(val)	移除指定元素
us.find(val)	查找元素，回傳 iterator 或 end()
us.count(val)	回傳元素是否存在（0 或 1）
us.bucket_count()	獲取桶數（hash table size）
us.load_factor()	載入因子，衡量填充程度
us.rehash(n)	重新配置桶數，至少為 n
us.begin(), us.end()	迭代器

📌 範例程式碼

```cpp
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_set<int> us;

    us.insert(10);
    us.insert(20);
    us.insert(10);  // 重複會被忽略

    std::cout << "內容: ";
    for (int x : us)
        std::cout << x << " ";
    std::cout << '\n';

    if (us.count(20))
        std::cout << "20 存在於集合中\n";

    us.erase(10);
    std::cout << "刪除 10 後內容: ";
    for (int x : us)
        std::cout << x << " ";
}
```

🚀 適用情境

- 只關心元素是否存在，不需排序
- 追求快速插入與查找（平均 O(1)）
- 元素類型必須可用於哈希（標準類型都支援）

⚠ 注意事項

- 不保證遍歷順序
- 哈希函式衝突會影響效能
- 需要合適的哈希函式與相等比較器

⚡ std::unordered_multiset — 允許重複元素的哈希集合

📘 概要

`std::unordered_multiset` 和 `unordered_set` 類似，
但允許多個相同元素（重複元素）存在於集合中。
底層依然是哈希表結構，操作平均時間複雜度為 O(1)。

🔧 常見操作與成員函式

方法 / 成員	說明
`ums.insert(val)`	插入元素，允許重複
`ums.erase(val)`	移除所有匹配元素
`ums.erase(it)`	移除指定 iterator 的元素
`ums.find(val)`	查找第一個匹配元素的 iterator
`ums.count(val)`	回傳該元素出現的次數
`ums.begin(), ums.end()`	迭代器範圍
`ums.bucket_count()`	桶數
`ums.load_factor()`	載入因子
`ums.rehash(n)`	重新配置桶數

📌 範例程式碼

```
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_multiset<int> ums;

    ums.insert(10);
    ums.insert(20);
    ums.insert(10);  // 重複元素會被保留

    std::cout << "內容: ";
    for (int x : ums)
```

```
    std::cout << x << " ";
  std::cout << '\n';

  std::cout << "10 出現次數: " << ums.count(10) << '\n';

  ums.erase(10); // 移除所有 10

  std::cout << "刪除 10 後內容: ";
  for (int x : ums)
    std::cout << x << " ";
}
```

🚀 適用情境

- 需要快速儲存多筆可能重複的資料
- 只需判斷元素存在與統計，不關心順序

⚠️ 注意事項

- 不保證元素遍歷順序
- `erase(val)` 會移除所有符合的元素
- 類型須可用於哈希與相等比較

⚡ `std::unordered_map` — 唯一鍵的哈希鍵值對容器

📘 概要

`std::unordered_map` 是一種基於哈希表的關聯式容器，儲存**鍵值對（key-value pairs）**，
提供**平均 O(1)** 的插入、查找與刪除速度，且 key **不可重複**。
它不像 `std::map` 會排序 key，而是依照 key 的 hash 值來組織資料。

```
#include <unordered_map>
std::unordered_map<std::string, int> um;
```

🧱 資料結構與特性

- 採用 **哈希表（hash table）**
- 每個元素是 `std::pair<const Key, T>`
- 不保證遍歷順序，但效能通常比 `map` 快

🔧 常見操作與成員函式

方法 / 成員	說明
`um[key] = val`	插入或更新 key 對應的值（若不存在會插…
`um.at(key)`	存取值，不存在則拋例外
`um.insert({key, val})`	插入鍵值對，若 key 存在則不變動
`um.emplace(key, val)`	原地建構鍵值對（效能較佳）
`um.find(key)`	找到 key 對應的 iterator
`um.count(key)`	key 是否存在（為 0 或 1）
`um.erase(key)` / `um.erase(it)`	移除元素
`um.begin()`, `um.end()`	迭代器（無排序）
`um.bucket_count()` / `load_factor()`	哈希表狀態資訊
`um.rehash(n)`	重新配置桶數

📌 範例程式碼

```cpp
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<std::string, int> scores;

    scores["Alice"] = 90;
    scores["Bob"] = 80;
    scores["Charlie"] = 85;

    // 使用範圍 for loop
    for (const auto& [name, score] : scores)
        std::cout << name << ": " << score << '\n';
```

```
    // 查找與更新
    if (scores.count("Bob"))
        scores["Bob"] += 5;

    // at() 會拋例外
    try {
        std::cout << "Eve: " << scores.at("Eve") << '\n';
    } catch (const std::out_of_range& e) {
        std::cout << "找不到 Eve\n";
    }
}
```

🧠 特別注意：`[]` 的隱含插入

```
int x = um["NewKey"]; // 如果 NewKey 不存在，會插入 { "NewKey", 0 }
```

與 `std::map` 相同，`[]` 操作**會插入預設值**，可能造成非預期的鍵出現。

🧠安全查詢（不建立 key）

```
if (um.find("Charlie") != um.end())
    std::cout << um["Charlie"];  // safe
```

🚀 適用情境

- 需要快速查找 / 插入 / 更新 key-value 配對資料
- 不需要自動排序（反而避免排序開銷）
- 類似「字典」、「hash table」、「資料庫索引」應用

⚠ 注意事項

- 不保證輸出順序，每次執行可能不同
- key 類型需支援 `std::hash<Key>` 與 `==` 比較
- 避免過度使用 `[]` 插入無意義的預設值

⚡ std::unordered_multimap — 允許重複鍵的哈希鍵值對容器

📘 概要

std::unordered_multimap 是 unordered_map 的變種，允許插入**多個相同的 key**。它不會自動排序 key，而是根據哈希值儲存資料。每筆資料仍是 pair<const Key, T>，但 key 可重複。

```
#include <unordered_map>
std::unordered_multimap<std::string, int> umm;
```

🧱 資料結構

- 底層為 **哈希表（hash table）**
- 插入、刪除、查找的平均時間複雜度為 O(1)
- 多個相同 key 的值彼此無順序關聯

🔧 常見操作與成員函式

方法 / 成員	說明
umm.insert({key, val})	插入鍵值對，允許重複 key
umm.emplace(key, val)	原地建構插入
umm.find(key)	回傳其中一個該 key 的元素 iterator
umm.count(key)	回傳該 key 出現的次數
umm.equal_range(key)	回傳 pair，範圍內包含所有相同 key 的元素
umm.erase(key) / umm.erase(it)	移除所有匹配 key / 指定位置
umm.begin(), umm.end()	遍歷所有元素

📌 範例程式碼

```
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_multimap<std::string, int> umm;
```

```
    umm.insert({"Alice", 90});
    umm.insert({"Bob", 80});
    umm.insert({"Alice", 85}); // 重複 key

    std::cout << "內容:\n";
    for (const auto& [name, score] : umm)
        std::cout << name << ": " << score << '\n';

    // 查找 Alice 的所有分數
    auto range = umm.equal_range("Alice");
    std::cout << "Alice 所有分數:\n";
    for (auto it = range.first; it != range.second; ++it)
        std::cout << it->second << '\n';
}
```

🚀 適用情境

- 一個 key 對應多個 value，例如：
  - 一位老師對應多門課
  - 一個分類對應多個產品
  - 多次交易記錄、log 資料等

⚠️ 注意事項

- 不支援 `operator[]`（因 key 不唯一）
- 元素遍歷順序不保證
- 重複 key 的存取需透過 `equal_range` 或迴圈 `find()` 多次找

✅ 非排序容器總結（Unordered Containers）：

容器類型	是否排序	是否允許重複	儲存內容
`unordered_set`	❌	❌	唯一元素
`unordered_multiset`	❌	✅	可重複元素
`unordered_map`	❌	❌	唯一 key 的鍵值對
`unordered_multimap`	❌	✅	可重複 key 的鍵值對

🔁 Container Adapters（容器配接器）

Container Adapter 是一類封裝現有容器（如 `deque` 或 `vector`）的容器，
提供特定資料結構的行為，例如「堆疊」、「佇列」等。
它們不直接提供 iterator，不支援遍歷，也沒有複雜的成員函式，
僅暴露出簡化的操作介面。

📦 常見的 Container Adapters：

Adapter 類型	行為	預設底層容器
`std::stack`	後進先出（LIFO）	`deque`
`std::queue`	先進先出（FIFO）	`deque`
`std::priority_queue`	優先順序佇列	`vector` + heap

📦 `std::stack` — 後進先出堆疊容器（LIFO）

📘 概要

`std::stack` 是一個**後進先出（LIFO）**的容器配接器，
封裝底層容器（通常是 `deque`），讓你只能從一端進出資料（頂端 top）。

```
#include <stack>
std::stack<int> s;
```

🔧 常見操作與成員函式

方法 / 成員	說明
`s.push(val)`	將元素推入堆疊
`s.pop()`	移除頂端元素（不回傳）
`s.top()`	取得頂端元素（不移除）
`s.empty()`	是否為空
`s.size()`	元素個數

📌 範例程式碼

```cpp
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;

    s.push(10);
    s.push(20);
    s.push(30);  // 現在堆疊是 [10, 20, 30]

    std::cout << "頂端元素: " << s.top() << '\n';  // 30

    s.pop();  // 移除 30
    std::cout << "現在頂端: " << s.top() << '\n';  // 20

    std::cout << "剩餘元素數: " << s.size() << '\n';
}
```

⚙️ 底層容器選擇

```cpp
std::stack<int, std::vector<int>> s1;
std::stack<int, std::deque<int>> s2;  // 預設
```

`std::stack` 可自訂底層容器，但必須支援：

- `back()`、`push_back()`、`pop_back()` 等操作

🚀 適用情境

- 模擬堆疊結構（呼叫堆疊、括號比對、DFS 等）
- 不需要隨機存取或遍歷，只需控制頂端元素

⚠️ 注意事項

- **不能遍歷！** 沒有 iterator
- `pop()` **不會回傳值**（如需同時取出請搭配 `top()`）

- 若需要隨機存取或排序，請使用其他容器（如 `vector` 或 `deque`）

📭 std::queue — 先進先出佇列容器（FIFO）

📘 概要

`std::queue` 是一種**先進先出（FIFO）**的容器配接器，
只允許從一端插入（`back`），從另一端移除（`front`）。
預設底層使用 `std::deque`，但也可使用 `list` 等支援雙端操作的容器。

```
#include <queue>
std::queue<int> q;
```

🔧 常見操作與成員函式

方法 / 成員	說明
`q.push(val)`	將元素加入隊尾（enqueue）
`q.pop()`	移除前端元素（dequeue），不回傳
`q.front()`	存取隊首元素（最早進來的）
`q.back()`	存取隊尾元素（最新進來的）
`q.empty()`	檢查是否為空
`q.size()`	傳回元素數量

📌 範例程式碼

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

int main() {
    std::queue<std::string> q;

    q.push("apple");
    q.push("banana");
    q.push("cherry");
```

```
    std::cout << "前端: " << q.front() << '\n'; // apple
    std::cout << "後端: " << q.back() << '\n';  // cherry

    q.pop(); // 移除 apple

    std::cout << "移除後前端: " << q.front() << '\n'; // banana
    std::cout << "剩餘數量: " << q.size() << '\n';
}
```

⚙️ 底層容器自訂（可選）

```
std::queue<int, std::deque<int>> q1; // 預設
std::queue<int, std::list<int>> q2;  // 也可使用 list
```

底層容器必須支援：

- `push_back()` 、 `pop_front()` 、 `front()` 、 `back()`

🚀 適用情境

- 模擬排隊、資源等待、BFS（廣度優先搜尋）
- 資料按時間順序處理，從最早到最晚
- 任務佇列、事件處理器

⚠️ 注意事項

- **不能隨機存取或遍歷！** 沒有 iterator
- `pop()` 不會回傳值（取值前請先 `front()` ）
- 若需優先處理順序不同的資料，請用 `priority_queue`

⛰️ `std::priority_queue` — 優先佇列（預設為最大堆）

📘 概要

`std::priority_queue` 是一種 **堆積（heap）結構** 的容器配接器，
提供**每次取出最大值（或最小值）**的功能，常用於排程、最佳化演算法（如

Dijkstra）。
底層實作為 `std::vector` 加上 `make_heap`、`push_heap`、`pop_heap` 等 heap 操作。

```
#include <queue>
std::priority_queue<int> pq; // 預設為 max-heap
```

🔧 常見操作與成員函式

方法 / 成員	說明
`pq.push(val)`	插入元素（自動維持 heap）
`pq.pop()`	移除頂端元素（最大或最小）
`pq.top()`	存取頂端元素
`pq.empty()`	是否為空
`pq.size()`	元素數量

📌 範例：最大堆（預設行為）

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

int main() {
    std::priority_queue<int> pq;

    pq.push(10);
    pq.push(30);
    pq.push(20);

    std::cout << "最大元素: " << pq.top() << '\n'; // 30

    pq.pop();
    std::cout << "下一個最大: " << pq.top() << '\n'; // 20
}
```

📌 範例：最小堆（min-heap）

```
#include <queue>
#include <vector>
#include <functional>

std::priority_queue<int, std::vector<int>, std::greater<>> minHeap;
```

這裡 `greater<>` 是比較函式，讓堆頂是最小值。

> ⓘ **Info**
>
> 為什麼 `std::priority_queue` 加上 `greater` 就會變成 **min-heap**？
> 它跟 `heapify` 的排序原理有什麼關係？

🧩 `std::priority_queue` 排序是怎麼決定的？

📦 priority_queue 預設定義：

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

> ✅ 它的排序規則不是「大放前面」或「小放前面」，而是由 `Compare` 來決定「哪個元素優先」
> ❗你可以把 `Compare(a, b)` 理解為：「a 是否應該排在 b 的**後面**？」

🧠 了解 `std::less` 與 `std::greater`

Comparator	表達意義	排序結果	對應堆
`std::less<T>`	a < b 成立 → b 比 a 優先	最大值在頂部	Max-Heap（預設）
`std::greater<T>`	a > b 成立 → b 比 a 優先	最小值在頂部	Min-Heap

🔍 換句話說，`priority_queue` 是這樣判斷順序的：

```
if (Compare(a, b)) {
  // 那麼 b 優先 → b 放在 heap 更高的位置（靠近 root）
}
```

✅ 舉例：

```
std::priority_queue<int, std::vector<int>, std::greater<>> pq;
```

插入順序：`5, 3, 7`

插入後的順序會根據 `std::greater` 來建堆：

- 比較時會問：「a > b 嗎？」
  - 若是，那麼 b 要優先，b 排在前面 → 形成「最小值在頂部」的 heap

✅ 視覺例子：

```
std::priority_queue<int, std::vector<int>, std::greater<>> pq;
pq.push(5);
pq.push(3);
pq.push(7);
```

形成的 heap 是這樣的 min-heap：

```
  3
 / \
5   7
```

📌 範例：`pair` 自訂比較（例如：Dijkstra）

```
#include <queue>
#include <vector>
#include <functional>

using pii = std::pair<int, int>;  // {cost, node}
std::priority_queue<pii, std::vector<pii>, std::greater<>> dijkstra;
```

`greater<>` 會根據 pair 第一個元素（cost）從小排到大。

🚀 適用情境

- 每次都要快速取得最大或最小值
- 演算法：Dijkstra、A*、K-th largest/smallest
- 排程系統、任務優先權管理

⚠️ 注意事項

- **不能遍歷！** 沒有 iterator
- 不支援中間刪除，只能操作 top
- 若需自訂排序，需提供自定 comparator 或 `greater<>`

📦 Container Adapters 總結：

類型	行為	主要操作	底層容器（預設）
`stack`	LIFO	`push`, `pop`, `top`	`deque`
`queue`	FIFO	`push`, `pop`, `front`, `back`	`deque`
`priority_queue`	heap	`push`, `pop`, `top`	`vector`

⚙️ 演算法通常需要額外資訊來執行工作

許多 STL 演算法需要額外提供**條件**或**行為**，來決定如何處理資料，例如比較大小、過濾條件等。這些可透過以下方式提供：

🧱 1. Functors（函式物件）

✅ 定義

Functor 是一個**重載了 `operator()` 的類別或結構體（struct）**，讓其物件可以像函式一樣被呼叫。它的優點包括：

- 可以封裝狀態（stateful）
- 支援 template 化與複雜邏輯
- 多用於需要高效的演算法中

📌 語法範例

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

struct IsOdd {
    bool operator()(int x) const {
        return x % 2 != 0;
    }
};

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    auto it = std::find_if(vec.begin(), vec.end(), IsOdd());

    if (it != vec.end()) {
        std::cout << "First odd number: " << *it << '\n';
    }
}
```

🧠 優點

- 可以儲存內部狀態（e.g. 累積計數、指定閾值）
- 編譯器優化友善（inline expansion）

🧭 2. Function Pointers（函式指標）

✅ 定義

Function pointer 是一個指向普通函式的指標，可以將一個已有的函式傳給 STL 演算法使用。適用於簡單、不需額外狀態的比較或邏輯。

📌 語法範例

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

```cpp
#include <algorithm>

bool descending(int a, int b) {
    return a > b;
}

int main() {
    std::vector<int> vec = {4, 2, 5, 1, 3};

    std::sort(vec.begin(), vec.end(), descending);

    for (int x : vec) std::cout << x << ' ';
    std::cout << '\n';
}
```

🧠 優點

- 語法簡單，容易閱讀
- 適合無狀態的邏輯
- 可重用已定義函式

⚠️ 缺點

- 無法封裝狀態
- 無法內嵌簡單邏輯（需額外定義函式）

⚡ 3. Lambda Expressions(C++11 之後)

Lambda Expression

🧮 STL 演算法 Algorithms

STL 提供了許多常見且實用的演算法（algorithms），可操作於各種容器元素的序列（sequences of container elements），這些序列是透過 **iterator** 提供給演算法使用的。

- STL 中的演算法數量眾多，無法在此一一詳述。

- 演算法涵蓋了排序（sorting）、搜尋（searching）、修改（modifying）、計算（numeric operations）等操作。
- 詳細列表可參考官方文檔：
  🔗 cppreference.com/w/cpp/algorithm

📌 std::sort — 排序元素範圍

🔷 說明

`std::sort` 是 `<algorithm>` 標頭檔中的排序演算法，用於**將範圍內的元素重新排列為遞增順序**，或透過自訂比較條件排序。使用隨機存取迭代器（如 `vector`、`deque`）運作，基於 **introsort**（混合 quicksort、heapsort、insertionsort）以達到高效排序。

🔗 語法

```
// 預設使用 operator< 排序
template< class RandomIt >
void sort( RandomIt first, RandomIt last );

// 使用自訂比較函式
template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

🧠 參數說明

參數	說明
`first`, `last`	要排序的範圍 `[first, last)`，需為 Random Access Iterator。
`comp`	比較函式（可為 function pointer, functor, 或 lambda），回傳 `bool`。如果 `comp(a, b)` 回傳 `true`，表示 `a` 會排在 `b` 前面。

📈 時間複雜度

- 最佳情況：`O(n log n)`
- 平均情況：`O(n log n)`
- 最差情況：`O(n log n)`（因為使用 introsort）

🧪 範例 1：排序 primitive type（int）

✅ 使用預設 `<`

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

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

    std::sort(v.begin(), v.end());

    for (int x : v) std::cout << x << ' ';
    // 輸出：1 2 3 4 5
}
```

✅ 使用 lambda 自訂遞減排序

```
std::sort(v.begin(), v.end(), [int a, int b](int%20a,%20int%20b) {
    return a > b;
});
// 輸出：5 4 3 2 1
```

🧪 範例 2：排序 user-defined type（struct）

🧱 定義資料結構

```
struct Student {
    std::string name;
    int score;
};
```

✅ 使用 functor 依照 score 遞減排序

```
struct CompareByScoreDesc {
    bool operator()(const Student& a, const Student& b) const {
```

```
        return a.score > b.score;
    }
};

std::vector<Student> students = {
    {"Alice", 90}, {"Bob", 75}, {"Charlie", 85}
};

std::sort(students.begin(), students.end(), CompareByScoreDesc{});
```

✅ 使用 lambda 表達式依照 name 排序

```
std::sort(students.begin(), students.end(),
    [const Student& a, const Student& b]
(const%20Student&%20a,%20const%20Student&%20b) {
        return a.name < b.name;
    });
```

📘 補充說明

- `std::sort` **不穩定**（相同元素的順序不一定保留），若需要穩定排序請使用 `std::stable_sort`。
- 比較函式需符合**嚴格弱排序(Strict Weak Ordering)** 規則，否則結果未定義。
- 可用於 `std::vector`、`std::deque` 等，但 **不可用於** **`std::list`**（因其為 bidirectional iterator，請改用 `list::sort()`）

🔍 `std::find_if` — 找出符合條件的第一個元素

🔷 說明

`std::find_if` 是 `<algorithm>` 標頭檔中的搜尋演算法，用來**在範圍中尋找第一個滿足條件的元素**，條件由一個可呼叫對象（predicate）決定。

🔧 語法

```
template< class InputIt, class UnaryPredicate >
InputIt find_if( InputIt first, InputIt last, UnaryPredicate p );
```

🧠 參數說明

參數	說明
`first`, `last`	要搜尋的範圍 `[first, last)`，需為 Input Iterator。
`p`	一元述詞（unary predicate），可為 function pointer、functor 或 lambda，接收一個參數並回傳 `bool`。

📈 時間複雜度

- 最壞情況：最多呼叫 `p` 次數為距離 `last - first`
- 平均時間：線性 O(n)

🧪 範例 1：搜尋 primitive type

✅ 找出第一個奇數

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

int main() {
    std::vector<int> v = {2, 4, 6, 7, 8};

    auto it = std::find_if(v.begin(), v.end(), [int x](int%20x) {
        return x % 2 != 0;
    });

    if (it != v.end()) {
        std::cout << "First odd: " << *it << '\n'; // 輸出：7
    }
}
```

🧪 範例 2：搜尋 user-defined type

🧱 定義資料結構

```
struct Student {
    std::string name;
    int score;
};
```

✅ 找出第一個及格的學生（score >= 60）

```
std::vector<Student> students = {
    {"Alice", 55}, {"Bob", 40}, {"Charlie", 75}, {"David", 60}
};

auto it = std::find_if(students.begin(), students.end(), [const Student& s]
(const%20Student&%20s) {
    return s.score >= 60;
});

if (it != students.end()) {
    std::cout << "First passing student: " << it->name << '\n'; // Charlie
}
```

📘 補充說明

- `find_if` 回傳第一個使述詞 `p(x)` 為 `true` 的元素的 iterator，若無則回傳 `last`。
- 若搜尋固定值（非條件），可使用 `std::find`：

  ```
  std::find(v.begin(), v.end(), 42);
  ```

- 若需搜尋最後一個滿足條件的元素，請使用 `std::find_if` 結合 `std::reverse_iterator` 或考慮 `std::ranges::find_if`（C++20）。

🔁 `std::for_each` — 對範圍內每個元素執行指定操作

🔷 說明

`std::for_each` 會遍歷區間 `[first, last)` 中的每個元素，並將每個元素傳入一個可呼叫對象（函式、函式物件、或 lambda）中執行。常用於**執行副作用操作**（如列印、累加、修改等）。

🔧 語法

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
```

🧠 參數說明

參數	說明
`first`, `last`	要遍歷的範圍 `[first, last)`，需為 Input Iterator。
`f`	一元函式（unary function），接收元素參考或值，通常回傳 `void`，也可為函式物件（支持累積狀態）。

📈 時間複雜度

- 線性 O(n)，需遍歷所有元素

🧪 範例 1：primitive types — 列印所有元素

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

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

  std::for_each(v.begin(), v.end(), [int x](int%20x) {
    std::cout << x << ' ';
  });
  // 輸出：1 2 3 4 5
}
```

🧪 範例 2：user-defined type — 修改成員值並列印

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

struct Student {
```

```
    std::string name;
    int score;
};

int main() {
    std::vector<Student> students = {
        {"Alice", 70}, {"Bob", 85}, {"Charlie", 90}
    };

    // 提升每個學生 5 分
    std::for_each(students.begin(), students.end(), [Student& s](Student&%20s) {
        s.score += 5;
    });

    // 列印結果
    std::for_each(students.begin(), students.end(), [const Student& s]
(const%20Student&%20s) {
        std::cout << s.name << ": " << s.score << '\n';
    });
}
```

🧪 範例 3：stateful functor / lambda — 累加元素和

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

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

    int sum = 0;

    // 捕捉外部變數 sum，累加每個元素
    std::for_each(v.begin(), v.end(), [&sum](int%20x) {
        sum += x;
    });
```

```
    std::cout << "Sum = " << sum << '\n';  // 輸出：Sum = 15
}
```

或者使用自訂 functor：

```
struct Accumulator {
    int total = 0;
    void operator()(int x) { total += x; }
};

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    Accumulator acc = std::for_each(v.begin(), v.end(), Accumulator());
    std::cout << "Sum = " << acc.total << '\n';  // 輸出：Sum = 15
}
```

📘 補充說明

- `for_each` 會回傳傳入的函式物件（通常是用來累積狀態的 functor），方便後續使用。
- 對於不修改元素內容，只需副作用操作非常合適。
- C++20 起可使用範圍版本 `std::ranges::for_each`，語法更簡潔。

🧠 STL 使用優勢

- 節省時間：不用自己造輪子
- 節省錯誤：經過測試、效率高
- 泛型設計：支援任意型別
- 組合彈性：容器＋演算法＋疊代器 = 強大組合

🔥 Iterator Invalidation（疊代器失效）

📌 定義

疊代器（iterator）是一種指向容器中元素的物件，但在某些操作（例如插入、刪除、清空）後，這些 iterator **可能失效**，也就是它們**不再指向有效的元素位置**，若繼續使用，會導致 **未定義行為（undefined behavior）**。

⚠ 當你執行某些操作時，會發生什麼？

💥 Suppose we are iterating over a vector of 10 elements, and we `clear()` the vector while iterating?

這樣的程式碼會導致**嚴重錯誤**：

```
std::vector<int> v = {1,2,3,4,5,6,7,8,9,10};

for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it == 5) {
        v.clear();  // 錯誤：使所有 iterator 失效
    }
    std::cout << *it << " ";  // 未定義行為（可能閃退）
}
```

當 `clear()` 被呼叫後，`it` 就變成一個「野指標」，繼續解參會導致**記憶體錯誤或當機**。

📌 哪些操作會使 iterator 失效？

這取決於容器種類（以下是最常見容器）：

容器	會使 iterator 失效的操作	備註
`vector`	`insert()`，`erase()`，`resize()`，`clear()`，`push_back()`（如容量不足）	可能全部失效
`deque`	同上	同樣危險
`list`	`erase()`，`clear()`	只會使刪除的元素的 iterator 失效，其它 iterator 安全

容器	會使 iterator 失效的操作	備註
`map` / `set`	`erase()`，`clear()`	插入不會影響既有 iterator，但刪除會失效
`unordered_map` / `unordered_set`	`insert()`，`erase()`，`rehash()`	rehash 時會全部失效

✅ 怎麼避免？

方法一：不要在迴圈中直接修改容器大小

錯誤示範：

```
for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it % 2 == 0)
        v.erase(it);  // ❌ 會導致 iterator 失效
}
```

正確寫法：

```
for (auto it = v.begin(); it != v.end(); ) {
    if (*it % 2 == 0)
        it = v.erase(it);  // ✔ erase 會回傳下一個有效的 iterator
    else
        ++it;
}
```

🧠 小結

重點	說明
Iterator 是容器的「指標」	它們的有效性取決於容器的狀態
某些操作會使 iterator 失效	如：`clear()` 、`erase()` 、`insert()` 等
使用失效的 iterator = 未定義行為	可能閃退、資料錯亂

重點	說明
避免失效：使用 erase 回傳值或重設 iterator	或使用新容器儲存欲保留資料

STL 關聯式容器的「進一步說明」，包含：

1. 🔀 `unordered_map` / `unordered_set` 的介紹與比較
2. 🧪 用 `map` 實作字頻統計（文字分析常見應用）
3. ⚖️ 用 `set` 實作集合運算（交集、聯集、差集）

操作	set	multiset	map	multimap
宣告	`set<int>`	`multiset<int>`	`map<K, V>`	`multimap<K, V>`
插入	`insert(x)`	`insert(x)`	`m[k] = v` / `insert({k,v})`	`insert({k,v})`
重複 key	❌	✅	❌	✅
查值	`find(x)`	`find(x)`	`m[k]` / `at(k)`	`equal_range(k)`
刪除	`erase(val)`	`erase(val)` or iterator	`erase(k)`	`erase(it)`
