Section 21 Lambda Expression

🧬 What is a Lambda Expression?

Lambda 表達式是 C++11 引入的語法，用來定義**匿名函式（anonymous function）**，允許你**就地定義可呼叫物件**（callable object），並可傳遞至演算法或儲存在變數中使用。

語法形式如下：

```
[capture](parameters) -> return_type {
    // function body
}
```

🎯 Motivation

在 C++ 中，常需定義簡短的函式來傳入 STL 演算法，如 `std::sort()`、`std::for_each()` 等。若使用傳統函式或函式物件（functor）方式撰寫，會顯得冗長且不直觀。

Lambda 表達式提供了一種**簡潔且易於閱讀**的方式來撰寫 inline 函式邏輯。

🧰 Review of Function Objects (Functors)

函式物件（functor）是**定義了 `operator()` 的 class 或 struct**，可像函式一樣呼叫：

```
struct Adder {
    int operator()(int a, int b) const {
        return a + b;
    }
};

Adder add;
int result = add(3, 5); // == 8
```

缺點： 必須先定義類別，才能使用。

🔗 Relation between Lambdas and Function Objects

Lambda expressions 背後實作其實就是**compiler 自動生成的匿名 functor class**。

```
auto lambda = [int a, int b](int%20a,%20int%20b) { return a + b; };
```

會被轉換為類似這樣的東西：

```
struct __Lambda_1 {
  int operator()(int a, int b) const {
    return a + b;
  }
};

__Lambda_1 lambda;
```

🧱 Structure of a Lambda Expression

Lambda 表達式的基本語法結構：

```
[capture_list](parameter_list) -> return_type {
  function_body
};
```

- `capture_list`：要從外部環境擷取的變數（by value、by reference 等）
- `parameter_list`：函式參數（可以省略）
- `return_type`：回傳型別（可省略，讓 compiler 自動推導）
- `function_body`：函式本體

範例：

```
int x = 10;
auto addX = [x](int%20y) { return x + y; };
std::cout << addX(5); // 輸出 15
```

🧬 Types of Lambda Expressions

📪 Stateless Lambda Expression

定義： 沒有 capture 外部變數的 lambda，稱為 stateless，轉型為 function pointer 是合法的。

```
auto square = [int x](int%20x) { return x * x; };
std::cout << square(4); // 16

// 可轉換為 function pointer
int (*fptr)(int) = square;
std::cout << fptr(5); // 25
```

- 無狀態 Lambda 指的是 **不捕獲外部變數的 Lambda**，也就是 capture list 為空 `[]` 。
- 這類 Lambda 不會保有外部狀態，只能使用參數或函式內部宣告的變數。
- 無狀態 Lambda 的類型是唯一且可轉換成函式指標。

📦 範例及語法詳解

範例 1 — 直接呼叫無參數無狀態 Lambda

```
[] () { std::cout << "Hi" << std::endl; }();
```

- `[]` 表示不捕獲任何外部變數。
- `()` 是函式參數列表，這裡是空參數。
- `{ std::cout << "Hi" << std::endl; }` 是函式本體。
- `()` 在 Lambda 後面立即呼叫該函式。

解說： 這是一個匿名的無參數 Lambda，立即執行並輸出 `Hi` 。

範例 2 — 帶參數的 Lambda 並立即呼叫

```
[] (int x) { std::cout << x << std::endl; }(100);
```

- `[]` 無捕獲。
- `(int x)` 帶一個整數參數 `x` 。
- Lambda 本體輸出參數 `x` 。
- 最後的 `(100)` 是立即呼叫 Lambda，傳入 `100` 。

解說： 此 Lambda 接受一個整數並輸出它，透過立即呼叫輸出 `100` 。

範例 3 — 多參數 Lambda

```
[int x, int y](int%20x,%20int%20y) { std::cout << x + y << std::endl; }(100, 200);
```

- 兩個參數 `x` 、 `y` 。
- 輸出兩參數相加的結果。
- 立即呼叫，傳入 `100` 和 `200` 。

解說： 輸出 `300` 。

範例 4 — 將 Lambda 指派給變數後呼叫

```
auto l1 = [] () { std::cout << "Hi" << std::endl; };
l1();
```

- Lambda 不再立即呼叫，而是先存入變數 `l1` 。
- 呼叫 `l1()` 時才執行 Lambda 內的程式碼。

範例 5 — Lambda 作為函式參數傳遞

```
std::vector<int> nums {10, 20, 30, 40, 50, 60};

filter_vector(nums, [int x](int%20x) { return x > 30; });
```

- 這裡 `filter_vector` 接收一個 `std::function<bool(int)>` ，而 Lambda 是一個無狀態的布林條件函式。
- Lambda 將元素 `x` 與 `30` 比較，回傳 `true` 或 `false` ，用於篩選。

範例 6 — 返回 Lambda

```
auto make_lambda() {
  return [] () { std::cout << "This lambda was made using the make_lambda function!"
<< std::endl; };
}
```

```
auto l5 = make_lambda();
l5();
```

- 函式 `make_lambda()` 回傳一個無狀態 Lambda。
- 呼叫 `l5()` 執行回傳的 Lambda。

範例 7 — Lambda 參數使用 auto（C++14 以上）

```
auto l6 = [auto x, auto y](auto%20x,%20auto%20y) {
  std::cout << "x: " << x << " y: " << y << std::endl;
};

l6(10, 20);
l6(100.3, 200);
```

- 使用 `auto` 讓 Lambda 參數支援泛型。
- 同一 Lambda 可接受不同型態的參數。

🧠 Stateful Lambda Expression (Capturing Context)

定義： 擷取（capture）外部變數的 lambda，即為 stateful，無法隱式轉換為 function pointer。

1. 捕獲外部變數 (Capture List) 簡介

- Lambda 的捕獲列表 `[]` 用來指定如何從外部環境「捕獲」變數。
- 捕獲方式分兩種：
  - **捕獲值（by value）**：捕獲變數的副本，Lambda 內修改不影響外部變數。
  - **捕獲參考（by reference）**：捕獲變數的參考，Lambda 內修改會影響外部變數。
- **捕獲全域變數**不需要特別捕獲，可直接使用。

2. 範例說明

範例 1：捕獲值（By Value）

```
int global_x {1000};

void test1() {
  int local_x {100};
  auto l = [local_x](.md) {
    std::cout << local_x << std::endl;  // 捕獲 local_x 的值副本
    std::cout << global_x << std::endl; // 全域變數可直接使用，不需捕獲
  };
  l();
}
```

- `[local_x]` 捕獲 `local_x` 的「值」。
- Lambda 內的 `local_x` 是副本，不能修改外部的 `local_x`。
- `global_x` 是全域變數，可直接讀取。

範例 2：捕獲值 + `mutable` 修飾符

```
void test2() {
  int x {100};
  auto l = [x](.md) mutable {
    x += 100;   // 修改的是 Lambda 內部的副本 x，不影響外部
    std::cout << x << std::endl;
  };
  l(); // 輸出 200
  std::cout << x << std::endl; // 外部 x 仍是 100
  l(); // 300
  std::cout << x << std::endl; // 100
}
```

- 捕獲值後，若想在 Lambda 內修改捕獲變數，需要加 `mutable`。
- 修改僅影響 Lambda 內的捕獲副本，不會改變外部變數。

capture 是在 lambda *建立時* 發生，而不是在第一次執行時才發生。

1. `auto l = [x](.md) mutable { ... };`
   - 此行建立 `lambda` 時，**複製了外部變數 `x` 的值（此時為 100）** 進到 lambda 內部。
   - 因為 `mutable`，你可以修改這份副本。

- 所以每次 `l()` 執行時，是在改變這個 lambda object **自己內部的 `x` 副本**。

2. **lambda 執行會改變這份內部副本，不會影響外部的 `x`**。但如果你多次執行 `l()`，它會繼續改變這份副本，形成累積效果。

✅ 重點整理：

項目	說明
capture 發生時機	**lambda 被建立的當下**
是否每次執行都重新 capture？	❌ 不是，每次執行都是用 lambda object 自己的內部狀態
`mutable` 的意義	允許你改變 lambda 內部捕捉來的值（預設是 `const`）
是否影響外部變數？	❌ `[x]` 是 **by value** capture，不會影響外部 `x`

範例 3：捕獲參考（By Reference）

```
void test3() {
  int x {100};
  auto l = [&x](.md) mutable {
    x += 100;   // 直接修改外部的 x
    std::cout << x << std::endl;
  };
  l();            // 輸出 200
  std::cout << x << std::endl;  // 外部 x 變成 200
}
```

- `[&x]` 捕獲 `x` 的參考，Lambda 內修改影響外部變數。
- `mutable` 在此可有可無，因為參考可直接修改。

範例 4：預設捕獲值 `[=]` + `mutable`

```
void test4() {
  int x {100}, y {200}, z {300};
  auto l = [=](.md) mutable {
    x += 100; // 修改 Lambda 內副本
    y += 100;
    std::cout << x << std::endl; // 200
```

```
    std::cout << y << std::endl; // 300
  };
  l();
  std::cout << x << std::endl;  // 外部 x 仍是 100
  std::cout << y << std::endl;  // 外部 y 仍是 200
}
```

- `[=]` 表示「預設捕獲所有用到的外部變數值副本」。
- 未使用的變數（例如 `z`）不會被捕獲。
- 只能修改副本，外部變數不受影響。

範例 5：預設捕獲參考 `[&]`

```
void test5() {
  int x {100}, y {200}, z {300};
  auto l = [&](.md) {
    x += 100;
    y += 100;
    z += 100;
    std::cout << x << std::endl; // 200
    std::cout << y << std::endl; // 300
    std::cout << z << std::endl; // 400
  };
  l();
  std::cout << x << std::endl;  // 外部 x 變成 200
  std::cout << y << std::endl;  // 外部 y 變成 300
  std::cout << z << std::endl;  // 外部 z 變成 400
}
```

- `[&]` 預設捕獲所有外部變數的參考。
- Lambda 內修改會直接反映在外部變數。

範例 6：預設捕獲值，特定捕獲參考 `[=, &y]`

```
void test6() {
  int x {100}, y {200}, z {300};
  auto l = [=, &y](.md) mutable {
    x += 100;
```

```
    y += 100; // y 是參考捕獲，會修改外部 y
    z += 100;
    std::cout << x << std::endl; // 200 (Lambda 內副本)
    std::cout << y << std::endl; // 300 (外部 y 被修改)
    std::cout << z << std::endl; // 400 (Lambda 內副本)
  };
  l();
  std::cout << x << std::endl; // 外部 x = 100 不變
  std::cout << y << std::endl; // 外部 y = 300
  std::cout << z << std::endl; // 外部 z = 300 不變
}
```

- `[=, &y]`：預設以「值」捕獲，但 `y` 以參考捕獲。
- 此例中 `y` 被修改會反映到外部。

範例 7：預設捕獲參考，特定捕獲值 `[&, x, z]`

```
void test7() {
  int x {100}, y {200}, z {300};
  auto l = [&, x, z](.md) mutable {
    x += 100; // x 是值捕獲（Lambda 內副本）
    y += 100; // y 是參考捕獲
    z += 100; // z 是值捕獲
    std::cout << x << std::endl; // 200
    std::cout << y << std::endl; // 300
    std::cout << z << std::endl; // 400
  };
  l();
  std::cout << x << std::endl; // 外部 x = 100 不變
  std::cout << y << std::endl; // 外部 y = 300 改變
  std::cout << z << std::endl; // 外部 z = 300 不變
}
```

- `[&, x, z]`：預設參考捕獲，`x`、`z` 是值捕獲。
- 這種混合捕獲方式可依需求彈性控制。

3. 特殊捕獲 `this` 與物件成員變數

範例 8：使用 [this] 捕獲物件成員

```
class Person {
 private:
  std::string name;
  int age;
 public:
  Person(std::string n, int a) : name{n}, age{a} {}

  auto change_person1() {
    return [this](std::string%20new_name,%20int%20new_age) {
      name = new_name; // 透過 this 指標操作成員變數
      age = new_age;
    };
  }
};

void test8() {
  Person person("Larry", 18);
  auto change_person1 = person.change_person1();
  change_person1("Moe", 30);
  std::cout << person << std::endl; // [Person: Moe : 30]
}
```

- [this] 捕獲目前物件指標，可在 Lambda 中修改成員。
- **注意**：C++20 中 [=] 捕獲 this 已被棄用，改用 [this] 或 [*this]。
- [*this]（C++20）為值捕獲整個物件。

---

4. Lambda 與函式物件（Functor）比較

範例 9：Lambda 等價於自訂 Functor

```
class Lambda {
 private:
  int y;
 public:
```

```
    Lambda(int y) : y{y} {};
    void operator()(int x) const {
        std::cout << x + y << std::endl;
    };
};

void test9() {
    int y {100};
    Lambda lambda1(y);
    auto lambda2 = [y](int%20x) { std::cout << x + y << std::endl; };

    lambda1(200);  // 輸出 300
    lambda2(200);  // 輸出 300
}
```

- Lambda 本質上是匿名函式物件，包含捕獲的資料成員（此例中是 `y`）。
- Functor 類別可模擬 Lambda 行為。

---

5. 複合捕獲示範 — 捕獲 `this` 與引用混合

範例 10：複合捕獲 `this` 與引用

```
class People {
    std::vector<Person> people;
    int max_people;
  public:
    People(int max = 10) : max_people(max) {}

    void add(std::string name, int age) {
        people.emplace_back(name, age);
    }

    int get_max_people() const { return max_people; }
    void set_max_people(int max) { max_people = max; }

    std::vector<Person> get_people(int max_age) {
        std::vector<Person> result;
```

```
    int count{0};
    std::copy_if(people.begin(), people.end(), std::back_inserter(result),
      [this, &count, max_age](const%20Person%20&p) {
        return p.get_age() > max_age && ++count <= max_people;
      });
    return result;
  }
};

void test10() {
  People friends;
  friends.add("Larry", 18);
  friends.add("Curly", 25);
  friends.add("Moe", 35);
  friends.add("Frank", 28);
  friends.add("James", 65);

  auto result = friends.get_people(17);
  for (const auto &p : result)
    std::cout << p << std::endl;
}
```

- Lambda 捕獲 `[this, &count, max_age]`：
  - `this` 捕獲物件指標，允許讀取成員變數 `max_people`。
  - `count` 以參考捕獲，能在 Lambda 內累計已處理人數。
  - `max_age` 以值捕獲，常數條件。
- 用 `std::copy_if` 篩選年齡大於 `max_age` 且數量不超過 `max_people` 的成員。

小結

捕獲方式	說明	Lambda 內是否可修改外部變數	是否影響外部變數
`[x]`	值捕獲，Lambda 內有副本	可（需 mutable）	否
`[&x]`	參考捕獲，Lambda 內直接使用外部變數參考	可	是

捕獲方式	說明	Lambda 內是否可修改外部變數	是否影響外部變數
[=]	預設值捕獲（捕獲用到的變數）	可（需 mutable）	否
[&]	預設參考捕獲（捕獲用到的變數）	可	是
[=, &y]	預設值捕獲，y 參考捕獲	可（需 mutable）	只有 y 是是
[&, x, z]	預設參考捕獲，x 、z 值捕獲	可（需 mutable）	只有參考捕獲變數是
[this]	捕獲物件指標，操作成員變數	可	是

🧠 Lambda 與 STL 函式

📘 std::function 說明

🔷 1. 概述 std::function

```
template<class R, class... Args> class std::function<R(Args...)>;
```

✅ 功能：

std::function 是一個泛型函式封裝器，用於封裝**任何符合特定函式簽章**的「可呼叫物件（callable object）」。

> 它的主要目的是讓函式、Lambda、函式物件（functor）和 std::bind 的結果可以用統一的方式儲存與呼叫。

🔷 2. 語法與語意

```
std::function<ReturnType(ParameterTypes...)> funcName;
```

🔶 Template 參數說明：

參數	說明
`ReturnType`	此 callable object 的回傳型態
`ParameterTypes...`	此 callable object 所需的參數型態（可為 0 個）

📌 使用方式：

```
std::function<int(int, int)> add;
```

這代表 `add` 可以儲存任何「可接受兩個 `int` 並回傳一個 `int`」的可呼叫物件。

🔷 3. 支援的 Callable 類型

類型	範例
普通函式	`void greet();`
函式指標	`void (*ptr)();`
Lambda（具/無 capture）	`[int x](int%20x) { return x*x; }`
函式物件（定義了 `operator()`）	`struct MyFunctor { void operator()(); };`
經由 `std::bind` 包裝的呼叫物件	`std::bind(func, args...)`

🔷 4. 常見成員與操作

函式或運算子	說明
`operator()`	呼叫封裝的 callable，語法如呼叫函式
`operator=`	可以將其他符合簽章的 callable 指派進來
`bool()`	可轉型為 `bool` 判斷是否有有效 callable
`.target<T>()`	回傳指向內部 callable 的指標（若類型正確）
`.swap()`	與另一個 `std::function` 交換內容

🔷 5. 使用範例

🧪 (A) 封裝普通函式

```
#include <iostream>
#include <functional>
void greet() {
   std::cout << "Hello!\n";
}
int main() {
   std::function<void()> f = greet;
   f();  // 呼叫 greet }
```

🧪 (B) 封裝 lambda（無 capture）

```
std::function<int(int)> square = [int x](int%20x) { return x * x; };
std::cout << square(5);  // 輸出 25
```

🧪 (C) 封裝 lambda（有 capture）

```
int factor = 3;
std::function<int(int)> scale = [factor](int%20x) { return x * factor; };
std::cout << scale(10);  // 輸出 30
```

🧪 (D) 使用 `std::bind`

```
#include <functional>
int add(int a, int b) {
   return a + b;
}

int main() {
   std::function<int(int)> add5 = std::bind(add, 5, std::placeholders::_1);
   std::cout << add5(3);  // 輸出 8
}
```

🧪 (E) 使用 Functor 類別

```
struct Printer {
   void operator()(const std::string& s) const {
```

```
        std::cout << "Message: " << s << "\n";
    }
};

std::function<void(std::string)> f = Printer{}; f("Hello"); // 輸出 Message: Hello
```

🔷 6. 與函式指標比較

特性	`std::function`	函式指標
可儲存 callable 類型	幾乎所有類型（含 lambda、有狀態函式物件）	僅支援普通函式與靜態函式
是否支援狀態（capture）	✅（可儲存 lambda 與閉包）	❌
是否可重新指派不同 callable	✅	只有同簽章函式可重新指派
型別安全性	高（template 控制）	較低，需自行小心
呼叫方式	`obj(args...)`	`ptr(args...)`
效能	較低（type-erased、可能用 heap）	較高（指標直接呼叫）
空狀態檢查	`.operator bool()`	檢查是否為 null 指標

🔷 7. 注意事項

- 預設建構的 `std::function` 是 **empty**，若呼叫 `operator()` 會丟出 `std::bad_function_call`。
- 可以使用 `if (f)` 判斷是否指向有效 callable。
- 有些高效能場合（如 HFT、嵌入式系統）會避免使用 `std::function`，因其效能不如函式指標或 inline lambda。

🧩 Lambda 作為 return value

Lambda 表示式是匿名函式，可以像一般函式物件一樣 return，常用於建立「自訂邏輯的函式」。

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

std::function<int(int)> make_adder(int x) {
    return [x](int%20y) {
        return x + y;
    };
}

int main() {
    auto add10 = make_adder(10);
    std::cout << add10(5) << std::endl; // 輸出 15
}
```

✨ 為什麼不用直接寫 member function？

1. **需要封裝環境中的變數（state）**：lambda 可 capture 外部變數
2. **可動態產生不同邏輯的函式**：例如根據輸入決定行為
3. **簡潔，可內嵌用於高階函式傳入**
4. **適合搭配 STL 演算法使用**

🔁 STL 函式與 Lambda 實例

std::find_if

- 語法：

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

- 功能：傳回第一個使得 `p(*it)` 為 true 的 iterator。
- 使用 lambda 判斷條件：

```
std::vector<int> v = {1, 3, 5, 7, 10};
auto it = std::find_if(v.begin(), v.end(), [int x](int%20x) { return x > 6; });
if (it != v.end()) std::cout << *it << std::endl; // 輸出 7
```

std::remove_if

- 語法：

```
template<class ForwardIt, class UnaryPredicate>
ForwardIt remove_if(ForwardIt first, ForwardIt last, UnaryPredicate p);
```

- 功能：將滿足條件的元素移至 vector 後方，回傳剩餘有效範圍的尾 iterator。
- 使用 lambda 判斷移除條件：

```
std::vector<int> v = {1, 2, 3, 4, 5};
auto it = std::remove_if(v.begin(), v.end(), [int x](int%20x) { return x % 2 == 0; });
v.erase(it, v.end()); // 移除偶數
```

std::sort

- 語法：

```
template<class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);
```

- 功能：根據比較函式 comp 排序範圍內的元素
- 使用 lambda 指定排序邏輯：

```
std::vector<int> v = {3, 1, 4, 2};
std::sort(v.begin(), v.end(), [int a, int b](int%20a,%20int%20b) { return a > b; });
// 變成降冪排序
```

🔒 lambda 補獲 this 的行為

當 lambda 宣告在 class 成員函式內，並 capture `this`，表示可以在 lambda 中存取該物件的所有成員。

```
class Person {
public:
  std::string name;
  int age;

  auto get_filter() {
    return [this](int%20min_age) {
      return age >= min_age; // 使用 this->age
    };
```

```
    }
};
```

> 注意：這是因為 lambda 只能存取其 scope 中的變數，若要存取 class 成員，就必須捕獲 this。

📄 STL 搭配 Lambda 實例說明

範例 1: `std::for_each` - 非修改操作

功能： 對容器中的每個元素執行動作

```
std::for_each(nums.begin(), nums.end(), [int num](int%20num) {
    std::cout << num << " ";
});
```

範例 2: `std::is_permutation` - 判斷兩序列是否為排列

功能： 判斷兩個 triangle 向量是否包含相同的點（順序無關）

```
std::is_permutation(vec1.begin(), vec1.end(), vec2.begin(), [Point lhs, Point rhs]
(Point%20lhs,%20Point%20rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
});
```

範例 3: `std::transform` - 修改序列

功能： 對每個元素加上 bonus 值

```
std::transform(scores.begin(), scores.end(), scores.begin(), [bonus](int%20x) {
    return x + bonus;
});
```

範例 4: `std::remove_if + vector::erase` - erase-remove idiom

功能： 移除所有偶數元素

```
nums.erase(std::remove_if(nums.begin(), nums.end(), [int num](int%20num) {
    return num % 2 == 0;
}), nums.end());
```

範例 5: `std::sort` - 使用自定義排序條件

功能： 對物件 vector 根據 name/age 排序

```
// 依 name 遞增排序
std::sort(people.begin(), people.end(), [Person a, Person b]
(Person%20a,%20Person%20b) {
    return a.get_name() < b.get_name();
});

// 依 age 遞減排序
std::sort(people.begin(), people.end(), [Person a, Person b]
(Person%20a,%20Person%20b) {
    return a.get_age() > b.get_age();
});
```

範例 6: `std::all_of` - 測試序列是否全部符合條件

功能： 檢查所有數字是否落在某區間內

```
std::all_of(nums.begin(), nums.end(), [start, end](int%20x) {
    return x >= start && x <= end;
});
```

範例 7: 結合 `all_of`, `none_of` 與 class 成員

功能： 驗證密碼中是否含有禁用字元

```
std::all_of(password.begin(), password.end(), [this](char%20c) {
    return c != restricted_symbol;
});

// 使用巢狀 lambda 判斷多個禁用符號：
std::all_of(password.begin(), password.end(), [this](char%20c) {
```

```
    return std::none_of(restricted_symbols.begin(), restricted_symbols.end(), [c]
(char%20s) {
        return c == s;
    });
});
```

✅ 小結：Lambda 適用場合

場景	使用 Lambda 好處
自定義排序、條件過濾	免寫額外函式、內嵌條件簡潔直觀
搭配 STL (`for_each` , `all_of` 等)	提高程式可讀性與靈活性
class 內部行為定義	使用 `[this]` 捕獲物件資料成員