第二課：泛型編程（Generic Programming）概念

第二課：泛型編程（Generic Programming）概念

2.1 什麼是泛型編程？

泛型編程（Generic Programming）是一種程式設計範式，核心思想是：
「撰寫與型別無關的程式碼，讓同一段邏輯可以處理多種不同的資料型別。」
在 C++ 中，泛型編程主要透過 **template**（模板）來實現。

一個簡單的問題引入

假設你需要寫一個「交換兩個變數」的函數：

```
#include <iostream>

// 交換兩個 int
void swap_int(int& a, int& b) {
int temp = a;
a = b;
b = temp;
}

// 交換兩個 double
void swap_double(double& a, double& b) {
double temp = a;
a = b;
b = temp;
}

// 交換兩個 string... 還要再寫一次？

int main() {
int x = 10, y = 20;
swap_int(x, y);
std::cout << "x = " << x << ", y = " << y << std::endl;
double a = 3.14, b = 2.72;
swap_double(a, b);
```

```
std::cout << "a = " << a << ", b = " << b << std::endl;
return 0;
}
```

輸出：

```
x = 20, y = 10
a = 2.72, b = 3.14
```

問題很明顯：**邏輯完全相同，只是型別不同**，卻要重複寫多次。這違反了 DRY 原則（Don't Repeat Yourself）。

2.2 Template：泛型編程的基石

C++ 的 **template** 讓我們可以寫出「型別參數化」的程式碼：

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

// 泛型版本：一次搞定所有型別
template <typename T>
void my_swap(T& a, T& b) {
T temp = a;
a = b;
b = temp;
}

int main() {
// 交換 int
int x = 10, y = 20;
my_swap(x, y); // 編譯器自動推導 T = int
std::cout << "x = " << x << ", y = " << y << std::endl;
// 交換 double
double a = 3.14, b = 2.72;
my_swap(a, b); // 編譯器自動推導 T = double
std::cout << "a = " << a << ", b = " << b << std::endl;
// 交換 string
std::string s1 = "Hello", s2 = "World";
my_swap(s1, s2); // 編譯器自動推導 T = std::string
std::cout << "s1 = " << s1 << ", s2 = " << s2 << std::endl;
```

```
return 0;
}
```

輸出：

```
x = 20, y = 10
a = 2.72, b = 3.14
s1 = World, s2 = Hello
```

Template 的運作機制

└──┘

Template 是**編譯期**的機制：

1. 編譯器看到 `my_swap(x, y)` 時，推導出 `T = int`
2. 編譯器根據模板**生成**一個 `void my_swap(int&, int&)` 函數
3. 這個過程叫做 **template instantiation**（模板實例化）

因為是編譯期展開，所以**沒有執行期的效能損失。**

2.3 函數模板（Function Template）

基本語法

```
template <typename T> // 或 template <class T>，兩者在這裡等價
返回型別 函數名稱(參數列表) {
// 函數主體
}
```

範例：找最大值

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

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

int main() {
std::cout << "max(10, 20) = " << find_max(10, 20) << std::endl;
std::cout << "max(3.14, 2.72) = " << find_max(3.14, 2.72) << std::endl;
std::cout << "max('a', 'z') = " << find_max('a', 'z') << std::endl;
// 字串比較（字典序）
std::string s1 = "apple", s2 = "banana";
std::cout << "max(\"apple\", \"banana\") = " << find_max(s1, s2) << std::endl;
return 0;
}
```

輸出：

```
max(10, 20) = 20
max(3.14, 2.72) = 3.14
max('a', 'z') = z
max("apple", "banana") = banana
```

明確指定型別

有時候編譯器無法自動推導，或你想強制使用特定型別：

```
#include <iostream>

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

int main() {
// 問題：10 是 int，3.14 是 double，型別不一致
// std::cout << find_max(10, 3.14); // 編譯錯誤！
// 解法一：明確指定型別
std::cout << find_max<double>(10, 3.14) << std::endl;
// 解法二：讓兩個參數型別一致
std::cout << find_max(10.0, 3.14) << std::endl;
return 0;
}
```

輸出：

```
10
10
```

多個型別參數

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

template <typename T1, typename T2>
void print_pair(T1 first, T2 second) {
std::cout << "(" << first << ", " << second << ")" << std::endl;
```

```
int main() {
print_pair(1, 3.14); // int, double
print_pair("Name", 42); // const char*, int
print_pair(std::string("Hello"), 'W'); // string, char
return 0;
}
```

輸出：

```
(1, 3.14)
(Name, 42)
(Hello, W)
```

2.4 類別模板（Class Template）

Template 不只能用在函數，也能用在類別。這是 STL 容器的基礎。

基本語法

```
template <typename T>
class ClassName {
// 類別成員可以使用 T
};
```

範例：簡易的泛型容器

讓我們實作一個超簡化版的「盒子」容器：

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

template <typename T>
class Box {
private:
T content;
bool has_content;
public:
Box() : has_content(false) {}
```

```
void put(const T& item) {
content = item;
has_content = true;
}
T get() const {
if (!has_content) {
throw std::runtime_error("Box is empty!");
}
return content;
}
bool is_empty() const {
return !has_content;
}
};

int main() {
// 裝 int 的盒子
Box<int> int_box;
int_box.put(42);
std::cout << "int_box contains: " << int_box.get() << std::endl;
// 裝 string 的盒子
Box<std::string> string_box;
string_box.put("Hello, STL!");
std::cout << "string_box contains: " << string_box.get() << std::endl;
// 裝 double 的盒子
Box<double> double_box;
double_box.put(3.14159);
std::cout << "double_box contains: " << double_box.get() << std::endl;
return 0;
}
```

輸出：

```
int_box contains: 42
string_box contains: Hello, STL!
double_box contains: 3.14159
```

STL 的 vector 就是類別模板

當你寫 `std::vector<int>` 時，就是在**實例化**一個類別模板：

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

int main() {
// vector<int> 是一個型別
// vector<double> 是另一個型別
// 它們是從同一個 template 實例化出來的不同類別
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::vector<std::string> words = {"Hello", "World"};
std::cout << "numbers[0] = " << numbers[0] << std::endl;
std::cout << "words[0] = " << words[0] << std::endl;
return 0;
}
```

輸出：

```
numbers[0] = 1
words[0] = Hello
```

2.5 泛型編程的核心概念：概念（Concepts）

泛型編程有一個重要的問題：**不是所有型別都適用於所有模板**。

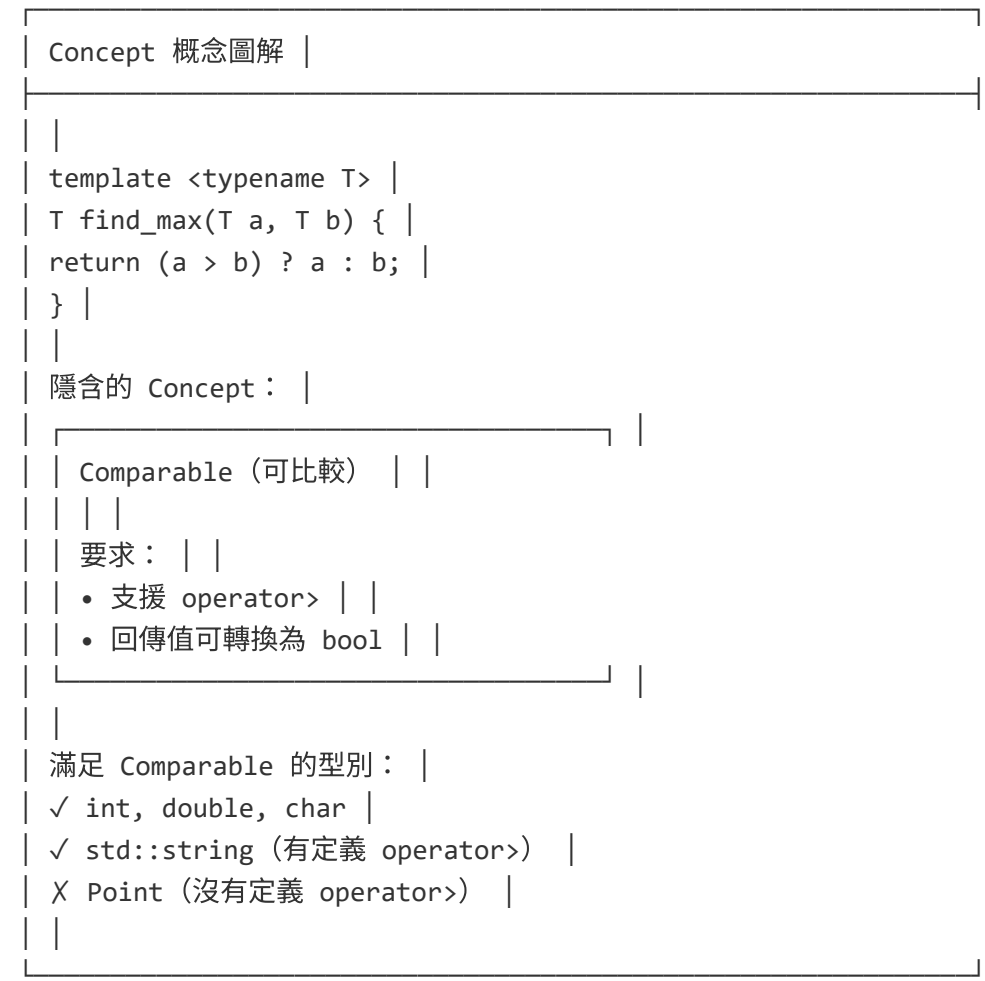
問題示範

```
#include <iostream>

template <typename T>
T find_max(T a, T b) {
return (a > b) ? a : b; // 使用了 > 運算子
}

// 自訂類別，沒有定義 > 運算子
class Point {
public:
int x, y;
Point(int x, int y) : x(x), y(y) {}
```

```
};

int main() {
Point p1(1, 2), p2(3, 4);
// Point max_point = find_max(p1, p2); // 編譯錯誤！
// 錯誤訊息：no match for 'operator>'
return 0;
}
```

隱含的要求：Concept

`find_max` 函數對型別 `T` 有一個**隱含的要求：** `T` 必須支援 `>` 運算子。
在泛型編程中，這種「型別必須滿足的條件」稱為 **Concept（概念）**。



讓自訂型別滿足 Concept

```
#include <iostream>

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

class Point {
public:
int x, y;
Point(int x, int y) : x(x), y(y) {}
// 定義 > 運算子，以距離原點的平方和比較
bool operator>(const Point& other) const {
return (x*x + y*y) > (other.x*other.x + other.y*other.y);
}
};

// 為了輸出
std::ostream& operator<<(std::ostream& os, const Point& p) {
return os << "(" << p.x << ", " << p.y << ")";
}

int main() {
Point p1(1, 2), p2(3, 4);
Point max_point = find_max(p1, p2); // 現在可以了！
std::cout << "max point: " << max_point << std::endl;
return 0;
}
```

輸出：

```
max point: (3, 4)
```

2.6 STL 中的 Concept（C++20 正式支援）

在 C++20 之前，Concept 只是「文件上的約定」。C++20 正式引入了 concept 關鍵字：

```
#include <iostream>
#include <concepts>

// 定義一個 Concept：可比較的型別
template <typename T>
concept Comparable = requires(T a, T b) {
{ a > b } -> std::convertible_to<bool>;
{ a < b } -> std::convertible_to<bool>;
};

// 使用 Concept 約束模板
template <Comparable T>
T find_max(T a, T b) {
return (a > b) ? a : b;
}

int main() {
std::cout << find_max(10, 20) << std::endl; // OK
std::cout << find_max(3.14, 2.72) << std::endl; // OK
return 0;
}
```

輸出：

```
20
3.14
```

現在你只需要知道 Concept 的**概念**，我們在後續課程會更深入探討。

2.7 泛型編程 vs 物件導向的多型

你可能會問：「這跟物件導向的繼承、虛擬函數有什麼不同？」

物件導向多型（執行期）

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

// 基底類別
```

```
class Shape {
public:
virtual double area() const = 0;
virtual ~Shape() = default;
};

// 衍生類別
class Circle : public Shape {
double radius;
public:
Circle(double r) : radius(r) {}
double area() const override {
return 3.14159 * radius * radius;
}
};

class Rectangle : public Shape {
double width, height;
public:
Rectangle(double w, double h) : width(w), height(h) {}
double area() const override {
return width * height;
}
};

int main() {
std::vector<std::unique_ptr<Shape>> shapes;
shapes.push_back(std::make_unique<Circle>(5.0));
shapes.push_back(std::make_unique<Rectangle>(4.0, 3.0));
for (const auto& shape : shapes) {
std::cout << "Area: " << shape->area() << std::endl; // 執行期決定
}
return 0;
}
```

輸出：

```
Area: 78.5397
Area: 12
```

泛型編程多型（編譯期）

```
#include <iostream>

class Circle {
double radius;
public:
Circle(double r) : radius(r) {}
double area() const {
return 3.14159 * radius * radius;
}
};

class Rectangle {
double width, height;
public:
Rectangle(double w, double h) : width(w), height(h) {}
double area() const {
return width * height;
}
};

// 泛型函數：不需要繼承關係
template <typename Shape>
void print_area(const Shape& shape) {
std::cout << "Area: " << shape.area() << std::endl; // 編譯期決定
}

int main() {
Circle c(5.0);
Rectangle r(4.0, 3.0);
print_area(c); // 編譯期展開為 print_area(const Circle&)
print_area(r); // 編譯期展開為 print_area(const Rectangle&)
return 0;
}
```

輸出：

```
Area: 78.5397
```

```
Area: 12
```

比較表

物件導向多型 vs 泛型編程多型		
面向	物件導向多型	泛型編程多型
決定時機	執行期（Runtime）	編譯期（Compile-time）
關係要求	需要繼承關係	不需要繼承關係
效能	虛擬函數表查詢開銷	無額外開銷
彈性	可存放異質物件	同一容器只能放同型別
錯誤時機	執行期才發現	編譯期就能發現
代表	virtual function	template

STL 選擇了泛型編程，原因是效能和編譯期錯誤檢查。

2.8 完整範例：泛型的 Stack

讓我們用所學的知識，實作一個泛型的 Stack：

```
#include <iostream>
#include <stdexcept>
#include <string>

template <typename T>
class Stack {
private:
static const int MAX_SIZE = 100;
T data[MAX_SIZE];
int top_index;
public:
Stack() : top_index(-1) {}
```

```
void push(const T& value) {
if (top_index >= MAX_SIZE - 1) {
throw std::overflow_error("Stack overflow");
}
data[++top_index] = value;
}
T pop() {
if (is_empty()) {
throw std::underflow_error("Stack underflow");
}
return data[top_index--];
}
T& top() {
if (is_empty()) {
throw std::underflow_error("Stack is empty");
}
return data[top_index];
}
bool is_empty() const {
return top_index < 0;
}
int size() const {
return top_index + 1;
}
};

int main() {
// int Stack
Stack<int> int_stack;
int_stack.push(10);
int_stack.push(20);
int_stack.push(30);
std::cout << "=== Int Stack ===" << std::endl;
std::cout << "Size: " << int_stack.size() << std::endl;
std::cout << "Top: " << int_stack.top() << std::endl;
while (!int_stack.is_empty()) {
std::cout << "Pop: " << int_stack.pop() << std::endl;
}
// string Stack（同一份程式碼，不同型別）
Stack<std::string> string_stack;
```

```
string_stack.push("World");
string_stack.push("STL");
std::cout << "\n=== String Stack ===" << std::endl;
std::cout << "Size: " << string_stack.size() << std::endl;
while (!string_stack.is_empty()) {
std::cout << "Pop: " << string_stack.pop() << std::endl;
}
return 0;
}
```

輸出：

```
=== Int Stack ===
Size: 3
Top: 30
Pop: 30
Pop: 20
Pop: 10

=== String Stack ===
Size: 3
Pop: STL
Pop: World
Pop: Hello
```

2.9 本課重點整理

```
┌──────────────────────────────────────────────────────────────┐
│ 第二課 重點整理 │
├──────────────────────────────────────────────────────────────┤
│ │
│ 1. 泛型編程 = 撰寫與型別無關的程式碼 │
│ → 核心工具：template │
│ │
│ 2. 函數模板 │
│ → template <typename T> │
│ → 編譯器自動推導型別，或明確指定 func<Type>(...) │
```

```
| |
| 3. 類別模板 |
| → template <typename T> class ClassName { ... }; |
| → 使用時必須明確指定：ClassName<Type> |
| |
| 4. 模板實例化 |
| → 編譯期展開，生成特定型別的程式碼 |
| → 無執行期效能損失 |
| |
| 5. Concept（概念） |
| → 型別必須滿足的條件 |
| → 例如：支援 operator>、可複製、可比較 |
| |
| 6. 泛型 vs 物件導向多型 |
| → 泛型：編譯期、無繼承要求、零開銷 |
| → 物件導向：執行期、需繼承、有虛擬表開銷 |
| |
| 7. STL 的基礎 |
| → vector<T>、list<T> 都是類別模板 |
| → sort、find 都是函數模板 |
| |
└──────────────────────────────────────────────────────────┘
```

2.10 課後練習

試著思考或實作：

1. **思考題**：`std::vector<int>` 和 `std::vector<double>` 是同一個類別嗎？
2. **實作題**：寫一個泛型函數 `print_array`，可以印出任何型別的陣列內容。

提示：

```
template <typename T>
void print_array(const T* arr, int size) {
// 你的實作
}
```

準備好進入**第三課：STL 的六大組件概覽**了嗎？下一課我們會鳥瞰整個 STL 的架構，了解容器、迭代器、演算法、函數物件、配置器、配接器這六大組件如何協同工作。