Section 12 Pointer

📌 為什麼要使用指標（Why use Pointers?）

指標（Pointer）是一種變數，儲存的是記憶體位置，它讓你可以更靈活、高效、低階地操作資料，是 C/C++ 語言的核心特性之一。

❓我不能直接用變數或函式本身嗎？（Can't I just use the variable or function itself?）

✅ 有些情況可以
🚫 但並非所有情況都可以，只使用變數或函式本身在某些場景會受限，這些限制就是為什麼我們需要指標。

✅ 可以，但不是總可以（Yes, but not always）
指標能突破變數作用域、記憶體分配限制，並支援動態行為，是解決以下幾種情況的關鍵：

📍函式內存取外部變數（Inside functions...）

✅ 用途
在函式中修改或存取外部變數
因為傳值呼叫（pass by value）不會改到原始變數，必須用指標來傳地址。

🧠 範例

```
void update(int* ptr) {
  *ptr = 10;
}
```

⚡操作陣列更有效率（Pointers can be used to operate on arrays very efficiently）

✅ 用途
陣列與指標緊密相關可透過指標運算快速存取或修改陣列元素
🧠 範例

```
int arr[3] = {1, 2, 3};
int* p = arr;
printf("%d", *(p + 1)); // 輸出 2
```

🧠 動態記憶體配置（We can allocate memory dynamically...）

✅ 用途
在程式執行時配置記憶體（heap）無變數名稱，只能透過指標使用
🧠 範例

```
int* arr = malloc(sizeof(int) * 5); // 配置 5 個 int 空間
```

🧬 物件導向中的多型（With OO, pointers are how polymorphism works!）

✅ 用途
在 C++ 中使用虛擬函式（virtual function）實現多型（Polymorphism）必須用指標（或參考）來呼叫

🧠 範例

```
Base* ptr = new Derived();
ptr->speak(); // 執行 Derived 的函式
```

📜 直接操作記憶體位址（Can access specific addresses in memory）

✅ 用途
直接存取特定記憶體位置（ex: 硬體暫存器），常見於嵌入式系統與底層開發
🧠 範例

```
#define REG *((volatile int*)0x40021018)
REG = 0x01; // 控制裝置（如 LED）
```

🧠 一、宣告指標時記憶體的情況

📌 宣告指標是什麼意思？

當你宣告一個指標變數時，你是在建立一個變數，用來儲存「某個記憶體位置的位址」。

🧪 範例：

```
int* ptr;
```

這行程式代表：

- ptr 是一個「指向 int 的指標」。
- 它會佔據一塊記憶體空間，用來儲存一個記憶體位址（通常是4或8 bytes）。

🧠 記憶體畫面示意（ASCII Art）：

```
int x = 42;
int* ptr = &x;
```

記憶體分佈（假設 x 位於位置 1000）：

```
地址   值    說明
------ ----  ----------------------
1000   42    x 儲存的值
2000   1000  ptr 儲存的是 x 的位址（&x）
```

🔍 二、什麼是 Dereferencing a Pointer（解參考指標）

📌 Dereferencing 是什麼？

所謂 dereferencing（解參考），是指透過指標取得它所指向的那塊記憶體中的值。
🧪 範例：

```
int x = 42;
int* ptr = &x;
printf("%d", *ptr); // 會印出 42
//這裡 *ptr 的意思就是「去 ptr 指的地方（1000），拿出那裡的值（42）」。
```

🧠 記憶體示意圖：

```
int x = 42;
int* ptr = &x;
```

記憶體狀態：

```
ptr --> [地址: 2000] = 1000
x   --> [地址: 1000] = 42
*ptr == 42 // 去地址 1000 拿值
```

🧪 更進一步的例子：修改變數內容

```
void update(int* p) {
   *p = 99;
}

int main() {
   int x = 5;
   update(&x);
   printf("%d", x); // 印出 99
}
```

這裡：

&x 把 x 的地址傳進函式。

*p = 99; 就是對那塊地址的內容（也就是 x）進行修改。

🔗 什麼是動態記憶體配置（Dynamic Memory Allocation）？

📌 定義：

在程式執行期間（runtime），**動態地**從「堆區（heap）」中向系統**申請記憶體空間**。
這段空間**沒有固定變數名稱，只能透過指標操作！**

💡 為什麼要用動態記憶體配置？

傳統宣告	動態配置
`int arr[10];`	陣列大小在編譯時決定
`int* arr = malloc(...);`	大小可在執行時決定

👉 適合當你不知道要使用多少記憶體，或者需要**在執行時根據輸入、條件決定大小**的情況！

🧠 C 語言中的 Dynamic Memory Allocation（動態記憶體配置）

📚 常見配置函式（來自 `<stdlib.h>`）

函式	說明
`malloc()`	分配未初始化的記憶體
`calloc()`	分配並初始化為 0
`realloc()`	調整已分配記憶體大小

函式	說明
free()	釋放記憶體

🧪 範例：使用 malloc()

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* arr = (int*) malloc(sizeof(int) * 5);  // 配置 5 個 int 的空間

    if (arr == NULL) {
        printf("記憶體配置失敗\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        arr[i] = i * 10;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    free(arr); // 釋放記憶體
    return 0;
}
```

🧠 記憶體示意圖（ASCII Art）

arr → [地址: 3000] = 指向 heap 上的空間（假設從 5000 開始）

Heap 區域內容：

```
5000: 0
5004: 10
5008: 20
5012: 30
5016: 40
```

✅ 注意事項

1 分配失敗要檢查（`malloc()` 可能回傳 `NULL`）

```
if (arr == NULL) {
    // 記憶體不足
}
```

2 記得 `free()`，否則會「記憶體洩漏（memory leak）」

```
free(arr); // 用完要釋放
```

3 配置陣列時記得 `sizeof(...)`

```
int* arr = malloc(sizeof(int) * N);
```

4 可搭配 `realloc()` 調整大小

```
arr = realloc(arr, sizeof(int) * 10);
```

🧠 小結表格

操作	意義
`malloc(n)`	分配 n bytes 記憶體，未初始化
`calloc(m, n)`	分配 m 個 n bytes 並初始化為 0
`realloc(ptr, size)`	調整指標指向的記憶體大小
`free(ptr)`	釋放記憶體

🧠 C++ 中的動態記憶體配置（Dynamic Memory Allocation）

✅ C++ 與 C 的差異

操作	C 風格	C++ 風格
分配記憶體	`malloc()`	`new`
釋放記憶體	`free()`	`delete`
配置陣列記憶體	`malloc()`	`new[]`
釋放陣列記憶體	`free()`	`delete[]`

🧪 使用 `new` 配置記憶體

📘 範例：配置單一整數

```
int* ptr = new int;  // 配置一個 int，未初始化
*ptr = 42;

std::cout << *ptr << std::endl;  // 印出 42

delete ptr;  // 釋放記憶體
```

🧪 使用 `new[]` 配置陣列

📘 範例：配置 int 陣列

```
int* arr = new int[5];  // 配置 5 個 int 的陣列

for (int i = 0; i < 5; i++) {
    arr[i] = i * 10;
}
```

```
for (int i = 0; i < 5; i++) {
    std::cout << arr[i] << " ";
}

delete[] arr;  // 釋放陣列記憶體
```

✅ 使用 `new` 配置物件

📘 範例：

```
class Dog {
public:
    void bark() { std::cout << "Woof!" << std::endl; }
};

Dog* d = new Dog();  // 配置 Dog 物件
d->bark();           // 呼叫成員函式
delete d;            // 刪除物件（呼叫 destructor）
```

🧠 記憶體配置圖示（ASCII Art）

```
Dog* d = new Dog();

d → [記憶體位址: 0x1234] → Dog 物件
```

⚠ 注意事項

1️⃣ `new` 配置的記憶體不會自動釋放

- 一定要使用 `delete` 或 `delete[]`

2️⃣ `delete[]` 用來釋放陣列

- `delete` 和 `delete[]` 不可混用！

```
int* arr = new int[5];
delete arr;   // ❌ 錯誤！未定義行為
delete[] arr; // ✅ 正確
```

3️⃣ 建議使用智能指標（`std::unique_ptr`，`std::shared_ptr`）來避免記憶體洩漏

```
#include <memory>

std::unique_ptr<int> p = std::make_unique<int>(100);
// 自動釋放，不需要寫 delete
```

🧠 小結表格

語法	說明
`new T`	配置一個 T 物件
`new T[n]`	配置一個 T 陣列
`delete ptr`	釋放用 `new` 配置的記憶體
`delete[] ptr`	釋放用 `new[]` 配置的陣列記憶體
`std::make_unique<T>`	建立智能指標，自動釋放

📌 建議學習順序

1. 先了解 `new` / `delete` 的基本語法
2. 熟悉 `new[]` / `delete[]` 用法與陣列
3. 練習 class 配置與刪除（建構與解構）
4. 學習 `RAII` 與 `智能指標（Smart Pointer）` 的使用

📌 為什麼要使用動態配置（`new`）而不是直接宣告變數？

✅ 什麼時候需要用 `new`？（動態記憶體配置）

1️⃣ 執行時才知道需要多少空間

```
int n;
std::cin >> n;
int* scores = new int[n]; // 動態配置「n 個 int」
```

> 如果用 `int scores[100];` 就可能爆掉，因為使用者輸入可能超過。

2️⃣ 需要讓資料在函式結束後仍然存在

```
int* makeValue() {
    int* p = new int(42);
    return p; // 回傳 heap 上的資料，還活著
}

int main() {
    int* q = makeValue();
    std::cout << *q << std::endl;
    delete q;
}
```

> 如果你用 `int x = 42;` 則 x 在函式結束後會被釋放，無法使用。

3️⃣ 建立執行時決定的「物件」或「陣列」

```
std::string* name = new std::string("John");
Dog* d = new Dog();
```

> 常見於需要在執行時建立 class 物件，或交由其他函式、結構管理。

4️⃣ 建構複雜資料結構：鏈結串列、樹、圖、堆疊...

```
struct Node {
  int data;
  Node* next;
};

Node* head = new Node{1, nullptr}; // 建立第一個節點
```

> 節點數量不固定，需要在「需要時」用 `new` 產生。

❌ 為什麼不要總是用 `new`？

```
int x = 10;          // ✅ 簡單直接，高效
int* p = new int(10);  // ❌ 複雜又容易漏掉 delete
```

✅ 使用原則：

狀況	使用方式
大小明確、生命週期可預期	◆ **直接宣告（stack）**
數量不定、跨函式、複雜結構	◆ **動態配置（heap）** 用 `new`

🧠 記憶比喻

宣告方式	類比
`int x = 5;`	書放在書桌上，隨時拿得到
`int* p = new int(5);`	書放在倉庫裡，要拿鑰匙（指標）去找