Section 17 Smart Pointer

🧨 Issues with Raw Pointers

在 C++ 中，傳統的 raw pointers（裸指標）提供了對記憶體管理的極大靈活性，開發者可以：

- ✳️ **自行配置記憶體（Allocation）**：例如使用 `new` 配置動態記憶體。
- ♻️ **自行釋放記憶體（Deallocation）**：例如使用 `delete` 或 `delete[]` 釋放資源。
- ⏳ **自行管理生命週期（Lifetime management）**：由開發者負責追蹤每一塊記憶體的存活時間。

❗潛在且嚴重的問題

這樣的自由度雖然強大，但也導致許多程式錯誤與難以除錯的問題：

- 🔍 **未初始化的指標（Wild pointers）**
  沒有初始化的指標指向未定義的記憶體位置，任何存取操作都可能導致 segmentation fault。
- 🕳️ **記憶體洩漏（Memory leaks）**
  若 `new` 了某個物件卻忘了 `delete`，那塊記憶體就永遠無法被釋放，造成記憶體洩漏（尤其在大型應用中危害更大）。
- ☠️ **懸掛指標（Dangling pointers）**
  指標所指向的物件已被釋放，但指標本身仍存在並指向該位置，再次使用會導致未定義行為。
- 💣 **例外安全問題（Not exception safe）**
  如果在 `new` 和 `delete` 的過程中發生例外（exception），而沒有妥善管理資源，可能會導致記憶體資源洩漏或程式錯誤。

🤔 所有權（Ownership）的問題

使用 raw pointers 也會產生一個核心問題：**這個指標的生命週期由誰負責？**

- 誰是這塊資源的「擁有者」？
- 什麼時候應該要 `delete` 指向的記憶體？
- 會不會一塊記憶體被多個地方 `delete` 兩次（double delete）？

C++ 的裸指標讓這些問題變得非常模糊，導致程式變得複雜又難維護。這也是為什麼 C++11 引入了 **Smart Pointers（智能指標）** 來解決這些問題，並結合 RAII 概念自動管理資源。

🧠 What Are Smart Pointers?

✅ 他們是什麼？

Smart Pointers 是 **物件（objects）**，不是單純的指標。它們是對傳統 raw pointers 的封裝，解決記憶體管理常見的問題，具有以下特性：

- 📦 **只能指向 heap 配置的記憶體**（透過 `new` 配置）
- 🧹 **當不再需要時會自動釋放記憶體（呼叫 `delete`）**
- 🔒 **遵循 RAII（Resource Acquisition Is Initialization）原則**，當 smart pointer 生命週期結束時，自動釋放其所擁有的資源

🔧 技術上他們是什麼？

- 🔃 **由 `class template` 所定義**，如 `std::unique_ptr<T>`、`std::shared_ptr<T>` 等
- 📦 **包裝一個 raw pointer**，讓你可以像使用普通指標一樣使用它，但背後包含完整的所有權與生命週期管理
- 🕹️ **需要 `#include <memory>`**
- 🔁 **提供重載的運算子：**
  - `*` 解引用（deref）
  - `->` 存取成員
- 🚫 **不支援指標運算（如 `++`、`--` 等）**
- 🛠️ **可以自定義 deleter（析構策略）**，這對於管理非記憶體資源（如檔案、socket）特別有用

📘 RAII — Resource Acquisition Is Initialization

RAII 是 C++ 中非常重要的一種設計模式，smart pointers 正是其代表應用之一。

📌 什麼是 RAII？

RAII 的基本理念是：

> 「資源的取得即是初始化，釋放則由物件生命週期決定。」

換句話說，只要物件存在，資源就保持存在；當物件離開作用域（scope），資源就會被自動釋放。

🎯 兩大階段：

1. 📥 Resource Acquisition（資源獲得）

- 例如：打開一個檔案、配置一塊記憶體、取得一個鎖
- 這些動作發生在「建構子（constructor）」中

2. ♻️ Resource Release（資源釋放）

- 例如：關閉檔案、釋放記憶體、釋放鎖
- 這些動作發生在「解構子（destructor）」中
- **你不必手動釋放**，物件離開作用域時自動釋放資源

🧩 Smart pointers 與 RAII

smart pointer 就是一種典型的 RAII 物件：

- 建構時綁定一塊 `new` 出來的 heap 資源
- 解構時自動 `delete` 掉該資源
- 不必手動管理資源，減少記憶體洩漏與 dangling pointer 問題

🔒 Unique Pointers (`std::unique_ptr<T>`)

✅ Simple smart pointer - very efficient!

`unique_ptr` 是一種輕量且效率極高的智能指標，用於表達 **「唯一所有權」** 的語意。它提供自動記憶體管理，但不具備 shared_ptr 的參考計數開銷，因此執行效率更佳。

🔷 `unique_ptr<T>` points to a heap object of type `T`

當我們使用 `unique_ptr`，我們通常是使用 `std::make_unique<T>()` 將一個物件配置到 heap 上：

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

int main() {
    std::unique_ptr<int> ptr = std::make_unique<int>(100);
    std::cout << *ptr << std::endl; // 輸出: 100
}
```

🔷 It is unique - only one `unique_ptr<T>` can own the object

不能有多個 `unique_ptr` 指向同一個物件，這樣可以避免 double delete 的風險。

```
#include <memory>

int main() {
    std::unique_ptr<int> p1 = std::make_unique<int>(42);
    // std::unique_ptr<int> p2 = p1; // ❌ 編譯錯誤：不能複製 unique_ptr
}
```

🔷 Owns what it points to

`unique_ptr` 擁有它指向的物件，因此一旦該指標被銷毀，物件也會自動被釋放。

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

struct MyClass {
    MyClass()  { std::cout << "Constructed\n"; }
    ~MyClass() { std::cout << "Destroyed\n"; }
};
```

```
int main() {
  {
    std::unique_ptr<MyClass> ptr = std::make_unique<MyClass>();
  } // 離開區塊時自動呼叫 destructor，輸出: "Destroyed"
}
```

◆ Cannot be assigned or copied

`unique_ptr` **不允許複製**，因為那會違反「唯一擁有者」的原則。

```
std::unique_ptr<int> p1 = std::make_unique<int>(10);
// std::unique_ptr<int> p2 = p1; // ❌ 無法複製
```

◆ CAN be moved

你可以使用 `std::move` 將擁有權「搬移」給另一個 `unique_ptr`。

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

int main() {
  std::unique_ptr<int> p1 = std::make_unique<int>(77);
  std::unique_ptr<int> p2 = std::move(p1); // p1 不再擁有該記憶體

  if (!p1)
    std::cout << "p1 is now null\n";
  std::cout << *p2 << std::endl; // 輸出: 77
}
```

◆ When the pointer is destroyed, the object is also destroyed

這是 RAII 原則的實踐例子，當 `unique_ptr` 被銷毀時，記憶體會被自動釋放。

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

struct Test {
  Test()  { std::cout << "Acquired\n"; }
  ~Test() { std::cout << "Released\n"; }
};

int main() {
  std::unique_ptr<Test> t = std::make_unique<Test>();
} // 離開作用域自動釋放，輸出: Acquired 然後 Released
```

🤝 Shared Pointers (`std::shared_ptr<T>`)

📌 Provides shared ownership of heap objects

`shared_ptr` 是一種支援 **多個指標共享同一個 heap 物件的智能指標**。它背後透過 **引用計數（reference count）** 管理記憶體的釋放時機。

🔷 `shared_ptr<T>` points to an object of type T on the heap

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

int main() {
  std::shared_ptr<int> sp = std::make_shared<int>(42);
  std::cout << *sp << std::endl; // 輸出: 42
}
```

🔷 It is **not unique** — many `shared_ptr` can point to the same heap object

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

int main() {
    std::shared_ptr<int> sp1 = std::make_shared<int>(100);
    std::shared_ptr<int> sp2 = sp1; // OK: 共享同一個資源
    std::cout << *sp2 << std::endl; // 輸出: 100
}
```

◆ Establishes shared ownership relationship

每個 `shared_ptr` 都會增加使用次數（use count），直到所有 `shared_ptr` 都離開 scope，才會真正釋放資源。

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

int main() {
    std::shared_ptr<int> sp1 = std::make_shared<int>(123);
    {
        std::shared_ptr<int> sp2 = sp1;
        std::cout << "Use count inside: " << sp1.use_count() << std::endl; // 2
    }
    std::cout << "Use count outside: " << sp1.use_count() << std::endl; // 1
}
```

◆ CAN be assigned and copied ✅

```
std::shared_ptr<int> a = std::make_shared<int>(5);
std::shared_ptr<int> b = a; // ✅ 合法的複製操作
```

◆ CAN be moved ✅

```
std::shared_ptr<int> a = std::make_shared<int>(77);
std::shared_ptr<int> b = std::move(a); // a 被移轉後不再持有所有權

if (!a) {
```

```
    std::cout << "a is now null" << std::endl;
}
std::cout << *b << std::endl; // 輸出: 77
```

🔷 Doesn't support managing arrays by default

`shared_ptr` 不直接支援 `new[]` 配置的陣列，你應使用 `std::shared_ptr<T[]>(...)` 並搭配自定 deleter，或改用 `std::vector<T>` 更安全。

```
#include <memory>

int main() {
    // ⚠ 錯誤用法：delete 不適用於陣列
    // std::shared_ptr<int> sp(new int[5]); // ❌

    // 正確做法（建議用 vector）
    auto sp = std::shared_ptr<int[]>(new int[5]); // ✅ 但注意不能用 `make_shared`
}
```

🔚 When the use count is zero, the object is destroyed

當所有指向該資源的 `shared_ptr` 被銷毀時，use count 變為 0，自動釋放記憶體。

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

struct MyClass {
    MyClass() { std::cout << "Constructed\n"; }
    ~MyClass() { std::cout << "Destroyed\n"; }
};

int main() {
    std::shared_ptr<MyClass> p1;
    {
        std::shared_ptr<MyClass> p2 = std::make_shared<MyClass>();
        p1 = p2;
        std::cout << "Use count: " << p2.use_count() << std::endl; // 2
```

```
    } // p2 離開 scope，但 p1 仍存在
    std::cout << "Still alive\n";
} // p1 離開 scope，use count 變 0，觸發 destructor
```

☒ Weak Pointers (std::weak_ptr<T>)

📌 Provides a non-owning "weak" reference

weak_ptr 是一種 **不擁有資源的智能指標**。它可以「觀察」某個 shared_ptr 所指的物件，但不會干涉其生命週期。

🔷 weak_ptr<T> points to a heap object of type T

它的行為類似指標，但沒有所有權，也不能直接解參考（*）使用。

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

int main() {
    std::shared_ptr<int> sp = std::make_shared<int>(10);
    std::weak_ptr<int> wp = sp;

    std::cout << "Shared value: " << *sp << std::endl; // OK
    // std::cout << *wp << std::endl; // ❌ 錯誤：不能直接解參考 weak_ptr
}
```

🔷 Does NOT participate in the owning relationship

weak_ptr 不會影響 shared_ptr 的使用次數（use count），不會阻止資源釋放。

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

int main() {
    std::shared_ptr<int> sp = std::make_shared<int>(20);
    std::weak_ptr<int> wp = sp;
```

```
    std::cout << "use_count: " << sp.use_count() << std::endl; // 1
}
```

◆ Always created from a `shared_ptr`

不能用 `new` 或 `make_shared` 直接創建 `weak_ptr`，**必須從 `shared_ptr` 建立**。

```
std::shared_ptr<int> sp = std::make_shared<int>(30);
std::weak_ptr<int> wp = sp; // ✅ 正確建立方式
```

◆ Used to prevent strong reference cycles

當兩個或多個物件互相用 `shared_ptr` 指向對方，就會造成 **循環引用（reference cycle）**，導致 use count 永遠無法歸零，物件永遠不會被釋放。

`weak_ptr` 就是用來打破這種循環的：

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

Class B;

Class A {
public:
    std::shared_ptr<B> b_ptr;
    A() {std::cout << "A constructed";}
    ~A() { std::cout << "A destroyed\n"; }
};

Class B {
public:
    std::weak_ptr<A> a_ptr; // 用 weak_ptr 打破循環
    B() {std::cout << "B constructed";}
    ~B() { std::cout << "B destroyed\n"; }
};

int main() {
    std::shared_ptr<A> a = std::make_shared<A>();
    std::shared_ptr<B> b = std::make_shared<B>();
```

```
    a->b_ptr = b;
    b->a_ptr = a; // 如果這裡用 shared_ptr，A 和 B 就永遠不會被釋放

    // 程式結束時 A 和 B 都能正確釋放
}
```

🔁 如何使用 `weak_ptr` 取值？`lock()` 方法

由於 `weak_ptr` 不能直接解參考，若想使用其指向的值，需先呼叫 `.lock()` 方法，這會回傳一個可能為空的 `shared_ptr` ：

```
std::weak_ptr<int> wp = sp;

if (auto spt = wp.lock()) {
    std::cout << "Value: " << *spt << std::endl;
} else {
    std::cout << "Resource has been released.\n";
}
```

總結：

Feature	`shared_ptr`	`weak_ptr`
擁有資源	✅	❌
會增加引用計數	✅	❌
會自動釋放記憶體	✅	❌
用來打破循環引用（cycles）	❌	✅

🛠 Custom Deleters

❓ 為什麼需要 Custom Deleter？

有時候我們管理的資源 **不是純粹用 `new` 建立的記憶體**，例如：

- 打開的檔案（`FILE*`）

- 資料庫連線
- 網路 socket
- 陣列（`new[]`）
- 第三方 C API 的記憶體配置器（例如 malloc/free）

這時若直接用 `delete` 去釋放，會造成錯誤或 memory leak。**這就是 custom deleter 派上用場的地方。**

✅ C++ 的 smart pointers 允許你提供自定義 deleter

支援 custom deleter 的 smart pointer：

- `std::unique_ptr<T, Deleter>`
- `std::shared_ptr<T>`（透過建構子或 `std::shared_ptr<T>(p, deleter)`）

🔧 實現方式有很多種

1️⃣ 使用函式（Function）

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

void fileCloser(FILE* fp) {
    if (fp) {
        std::cout << "Closing file...\n";
        fclose(fp);
    }
}

int main() {
    std::unique_ptr<FILE, decltype(&fileCloser)> filePtr(fopen("data.txt", "r"),
fileCloser);

    if (filePtr) {
        std::cout << "File opened.\n";
    }
} // 離開作用域會自動呼叫 fileCloser，關閉檔案
```

2 使用 Lambda（最常見、最簡潔）

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

int main() {
    auto deleter = [int* ptr](int*%20ptr) {
        std::cout << "Deleting int pointer with value: " << *ptr << std::endl;
        delete ptr;
    };

    std::unique_ptr<int, decltype(deleter)> up(new int(99), deleter);
} // 自動呼叫 lambda，輸出: Deleting int pointer with value: 99
```

3 用於 `shared_ptr` 的情境（也可搭配 lambda）

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

struct MyResource {
    MyResource()  { std::cout << "Resource acquired\n"; }
    ~MyResource() { std::cout << "Should not be called!\n"; }
};

int main() {
    auto customDeleter = [MyResource* p](MyResource*%20p) {
        std::cout << "Custom deleter called\n";
        delete p;
    };

    std::shared_ptr<MyResource> ptr(new MyResource, customDeleter);
} // 離開作用域時會使用 customDeleter，而非呼叫 ~MyResource()
```

⚠ 注意事項

- `unique_ptr` 使用 custom deleter 時，**需要在類型中指定 deleter 的型別**（如 `decltype(...)`）

- `shared_ptr` 的 deleter **不會改變型別**，但只能在初始化時指定，**之後不可變更**
- Lambda 是最方便也最常用的方法

📦 應用場景範例

資源類型	建議用法
`FILE*` 檔案指標	`fclose()` via custom deleter
`malloc` 分配	`free()` via custom deleter
socket / handle	自定義 `close()` 函式
`new[]` 陣列	使用 `delete[]`

🧠 `smart pointer` 與 raw pointer的 解參考（dereferencing）行為差異

#觀念釐清

🧠 基本前提：什麼是解參考？

- **解參考**就是「*拿出指標所指向的資料來用*」。
- 對於 raw pointer 和 smart pointer，解參考語法都是 `*ptr`。
- **但背後行為、型別與安全性差異極大**。

📜 比較總表

特性項目	Raw Pointer (`T*`)	Smart Pointer (`std::shared_ptr<T>` 等)
語法	`*ptr`	`*sptr`
回傳值型別	`T&` （參考）	`T&` （參考）
能否使用 `->`	✅ 可以	✅ 可以
是否可以複製資料	❌ 解參考不會複製	❌ `.operator*()` 回傳 reference，不複製

特性項目	Raw Pointer (`T*`)	Smart Pointer (`std::shared_ptr<T>` 等)
是否自動釋放記憶體	❌ 否	✅ 是（RAII）
安全性	❌ 危險（可能是 nullptr / dangling）	✅ 有檢查、較安全
`operator*()` 實作	無（語言內建）	自行實作，通常回傳 `T&` ，避免值拷貝

🔍 差異一：是否會複製資料？

「為什麼 `smart pointer` 解參考會傳 reference 而不是值？」

✅ Smart pointer 的 `operator*()` 長這樣（簡化版）：

```
T& operator*() const noexcept {
    return *ptr_; // ptr_ 是內部的 raw pointer
}
```

這意味著：

- 解參考 `*sptr` 拿到的是 **原本物件的參考**
- 不會呼叫 copy constructor，不會產生「值的複製」

🧠 這樣做的好處：

- ✅ 避免效能浪費（避免不必要的拷貝）
- ✅ 保證操作的是原本的物件（特別重要於大型物件、不可複製的物件）

❌ 錯誤理解：以為 smart pointer 解參考後會複製一份

```
auto sptr = std::make_shared<std::vector<int>>();
auto v = *sptr; // ✅ 這是複製一份（v 是值，不是參考）
v.push_back(42); // 不會改到 sptr 裡的資料
```

如果你寫 `auto& v = *sptr;` ，那才會變成操作原本的資料。

📌 與 raw pointer 差在哪？

Raw pointer（例如 `T* ptr`）的解參考行為：

```
int a = 10;
int* ptr = &a;

int& ref = *ptr;  // ✅ 拿到 reference
int  val = *ptr;  // ✅ 複製一份值
```

- 語法上直接支援這兩種：值 or reference
- 沒有安全機制，容易踩到空指標、野指標（野生生物）

✅ 小範例比較

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

int main() {
    // raw pointer
    int a = 42;
    int* p = &a;
    int  val1 = *p;  // 拷貝
    int& ref1 = *p;  // reference

    // smart pointer
    auto sp = std::make_shared<int>(99);
    int  val2 = *sp;  // 拷貝 smart pointer 所管理的 int 值
    int& ref2 = *sp;  // reference：直接操作內部資料

    ref2 = 100;
    std::cout << *sp << "\n"; // 印出 100
}
```

🧠 補充：為什麼 smart pointer 要回傳 reference？

若 smart pointer 這樣寫：

```
T operator*() { return *ptr_; } // ❌ 回傳值（複製一份）
```

那麼你在這樣寫時：

```
(*sptr).set_name("Jorson");
```

你其實是改到**一份拷貝品**，而不是原物件 → 完全違反了 RAII 管理資源的目的！

✅ 結論總整理

比較點	Raw Pointer	Smart Pointer (`shared_ptr` , `unique_ptr`)
解參考回傳型別	`T&`	`T&` （透過 `.operator*()` ）
是否會自動複製	❌ 解參考不會	❌ 解參考也不會
`.operator*()` 實作	無（語言內建）	✅ 明確回傳 reference，避免值拷貝
使用目的	任意操作記憶體	安全、負責資源生命週期

「語法上解參考 `*ptr` 的表現」 和 「背後實際的型別」 之間的區別。

> Raw pointer 解參考後不是回傳那個記憶體的「值」嗎？
> 為什麼說它的型別是 `T&` ？

✅ 關鍵答案：

➡ `*ptr` **的型別是** `T&` **（T 的參考）**，但你可以選擇：

- ✅ 使用它的「值」 → `T val = *ptr;` （複製資料）
- ✅ 使用它的「參考」 → `T& ref = *ptr;` （直接參照原資料）

🔬 解釋 `*ptr` 是 `T&` 的原因（從語言規則來看）

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

int val = *p;    // ✅ 把 p 指到的內容複製出來 → val == 42
int& ref = *p;   // ✅ ref 是 x 的別名

*p = 99;       // ✅ 改變 p 指向的內容，也就是改變 x
std::cout << x;  // 👉 印出 99
```

👉 為什麼 `*p` 是 `int&`？

因為 C++ 規定：

> 解參考一個指標（`*p`）得到的是「該記憶體內容的別名」，也就是 **reference**，而不是一份值的拷貝。

🧠 具體記憶體圖：

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

```
[記憶體結構]
x:  [42]
p:  → x 的地址

*p == x
```

📜 編譯器角度：看型別推導

你可以加一段程式印出型別來驗證：

```
#include <iostream>
#include <type_traits>
```

```
int main() {
    int x = 5;
    int* p = &x;

    std::cout << std::boolalpha;
    std::cout << std::is_same<decltype(*p), int&>::value << "\n"; // ✅ true
}
```

這會印出 `true`，說明 `*p` 是 `int&`，不是 `int`。

✅ 小節：你看到的「值」其實是透過 reference 來的

表達式	型別	說明
`*p`	`T&`	解參考，結果是參考（不是複製值）
`T val = *p;`	`T`	把 reference 的值「複製」出來
`T& ref = *p;`	`T&`	把 reference 再綁到另一個 reference

🧪 舉例對照：raw pointer 與 smart pointer 的一致性

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

*p = 99;    // OK → 透過 T& 改變值
std::cout << x << "\n"; // 👉 99

auto sp = std::make_shared<int>(123);
*sp = 456;   // 也 OK，因為 *sp 是 int&
std::cout << *sp << "\n"; // 👉 456
```

✅ 因為 `*p` 和 `*sp` 都是 reference，才可以寫進去值！

🎯 最後總結

解參考來源	表達式	型別是什麼？	拿到的是什麼？
Raw pointer	`*p`	`T&`	指向物件的參考
Smart pointer	`*sptr`	`T&`	包裝的物件參考（透過 `operator*()`）
真正的值	`T val = *p`	`T`	複製出來的新資料
