智能指针

1.[unique\_ptr](mk:@MSITStore:E:\\c++-c-Qt资料集合\\学习文件(c or c++)\\c和c++参考手册.chm::/chmhelp/cpp-memory-unique_ptr.html) 拥有独有对象所有权语义的智能指针

提供独有所有权，适合管理唯一拥有者的资源。

2.[shared\_ptr](mk:@MSITStore:E:\\c++-c-Qt资料集合\\学习文件(c or c++)\\c和c++参考手册.chm::/chmhelp/cpp-memory-shared_ptr.html) 拥有共享对象所有权语义的智能指针

提供共享所有权，适合多个对象或线程共享资源的场景。

3.[weak\_ptr](mk:@MSITStore:E:\\c++-c-Qt资料集合\\学习文件(c or c++)\\c和c++参考手册.chm::/chmhelp/cpp-memory-weak_ptr.html) 到 [std::shared\_ptr](mk:@MSITStore:E:\\c++-c-Qt资料集合\\学习文件(c or c++)\\c和c++参考手册.chm::/chmhelp/cpp-memory-shared_ptr.html) 所管理对象的弱引用

提供弱引用，解决 std::shared\_ptr 的循环引用问题。

4.[auto\_ptr](mk:@MSITStore:E:\\c++-c-Qt资料集合\\学习文件(c or c++)\\c和c++参考手册.chm::/chmhelp/cpp-memory-auto_ptr.html) 拥有严格对象所有权语义的智能指针

由于不安全的所有权转移机制，已被弃用，建议使用 std::unique\_ptr 代替。

C++ 中的智能指针是用于自动管理动态内存分配的对象指针，避免了手动管理内存的复杂性和潜在的内存泄漏问题。以下是对 std::unique\_ptr、std::shared\_ptr、std::weak\_ptr 和 std::auto\_ptr 的分析、解释以及设计背后的原因。

### 1. std::unique\_ptr

std::unique\_ptr 是一个拥有**独有对象所有权语义**的智能指针。它确保一个指针只有一个所有者，因此它不能被复制，只能通过移动语义（move semantics）进行转移所有权。

**特点：**

* **独占所有权**：std::unique\_ptr 通过禁止拷贝构造和拷贝赋值，确保一个对象在任一时刻只有一个所有者。
* **自动释放内存**：当 std::unique\_ptr 离开作用域时，它所管理的对象会自动被销毁，并释放其内存。
* **移动语义**：std::unique\_ptr 支持移动语义，可以通过 std::move 将所有权转移给另一个 std::unique\_ptr。

**示例：**

std::unique\_ptr<int> p1(new int(10)); // p1 拥有 int 对象

std::unique\_ptr<int> p2 = std::move(p1); // p1 的所有权被转移给 p2，p1 不再拥有对象

**设计原因：** std::unique\_ptr 的设计旨在提供一个轻量级、低开销的智能指针，适合管理独占资源。当我们确定某个资源不应被多个指针共享时，std::unique\_ptr 是理想的选择。

### 2. std::shared\_ptr

std::shared\_ptr 是一个拥有**共享对象所有权语义**的智能指针。它允许多个 std::shared\_ptr 实例共享同一个对象，并通过引用计数来管理对象的生命周期。

**特点：**

* **共享所有权**：多个 std::shared\_ptr 可以共享同一个对象，当最后一个 std::shared\_ptr 被销毁时，所管理的对象才会被释放。
* **引用计数**：std::shared\_ptr 内部维护一个引用计数，每当有新的 std::shared\_ptr 指向同一对象时，引用计数加 1；当 std::shared\_ptr 被销毁或重置时，引用计数减 1。当引用计数为 0 时，自动释放对象内存。

**示例：**

std::shared\_ptr<int> p1(new int(10)); // 引用计数为 1

std::shared\_ptr<int> p2 = p1; // 引用计数为 2

p1.reset(); // p1 不再指向对象，引用计数为 1

**设计原因：** std::shared\_ptr 旨在解决需要共享所有权的场景，如多个对象或线程需要访问和管理同一个资源。它通过引用计数自动管理对象的生命周期，减少手动管理内存的复杂性。

### 3. std::weak\_ptr

std::weak\_ptr 是一个到 std::shared\_ptr 所管理对象的**弱引用**。它不会影响对象的引用计数，因此不能独立管理对象的生命周期。

**特点：**

* **不增加引用计数**：std::weak\_ptr 不会增加引用计数，因此它不拥有对象的所有权。
* **避免循环引用**：在使用 std::shared\_ptr 时，容易出现循环引用的问题，导致内存泄漏。std::weak\_ptr 可以打破这种循环引用。
* **访问对象**：std::weak\_ptr 需要通过 lock() 方法提升为 std::shared\_ptr 才能访问对象，只有在对象仍然存在时，lock() 才会成功。

**示例：**

std::shared\_ptr<int> p1(new int(10));

std::weak\_ptr<int> wp = p1; // wp 是 p1 的弱引用，不影响引用计数

if (std::shared\_ptr<int> p2 = wp.lock()) {

// 如果对象仍然存在，p2 将指向该对象

std::cout << \*p2 << std::endl;

}

**设计原因：** std::weak\_ptr 的设计目的是解决 std::shared\_ptr 的循环引用问题。在某些情况下，一个对象需要知道另一个对象的存在，但不应该影响它的生命周期，此时 std::weak\_ptr 是理想的选择。

### 4. std::auto\_ptr（已弃用）

std::auto\_ptr 是 C++98 中引入的智能指针，拥有严格对象所有权语义，即在 std::auto\_ptr 被复制时，所有权会转移给新的 std::auto\_ptr 对象，而旧的指针会被置为 nullptr。

**特点：**

* **严格所有权**：当 std::auto\_ptr 被拷贝时，所有权会从原始指针转移到新的指针，旧指针不再拥有对象。
* **易导致问题**：这种设计容易导致意外的所有权转移，使得 std::auto\_ptr 在容器中使用时非常危险。

**示例：**

std::auto\_ptr<int> p1(new int(10));

std::auto\_ptr<int> p2 = p1; // p1 的所有权被转移给 p2，p1 变为 nullptr

**设计原因及弃用原因：** std::auto\_ptr 的设计初衷是提供一种简单的智能指针来管理动态内存，但由于它的所有权转移机制太过严格，容易导致内存管理问题和意外的资源丢失。为了更安全和明确地管理所有权，C++11 引入了 std::unique\_ptr 和 std::shared\_ptr，并在 C++11 中正式弃用了 std::auto\_ptr。