第十章复习材料：动态内存分配与智能指针

1. 原始内存分配（`malloc` / `free` 和 `new` / `delete`）

- **`malloc` 和 `free`**：传统的C语言函数用于在堆上分配和释放内存，`malloc` 只是分配内存，但不会调用构造函数，`free` 只是释放内存，也不会调用析构函数。
  - **风险**：开发人员需要手动管理内存，容易造成内存泄漏或指针悬挂问题（使用未释放或重复释放的内存）。
- **`new` 和 `delete`**：C++ 提供的内存分配方法，`new` 不仅分配内存，还会调用对象的构造函数，`delete` 不仅释放内存，还会调用析构函数。
  - **风险**：如果 `new` 和 `delete` 不成对使用，会导致内存泄漏或重复释放内存的问题。

  示例：

```
int* ptr = new int(5);  // 使用 new 分配内存并初始化
delete ptr;  // 使用 delete 释放内存
```

- **常见错误**：
  - **内存泄漏**：没有正确释放动态分配的内存。

```
int* ptr = new int(10);
// 忘记 delete ptr; 造成内存泄漏
```

  - **重复删除**：多次调用 `delete`。

```
int* ptr = new int(10);
delete ptr;
delete ptr;  // 错误，导致未定义行为
```

  - **重新分配内存**：指针指向新分配的内存，旧内存未释放。

```
int* ptr = new int(10);
ptr = new int(20);  // 错误，之前的内存没有 delete
```

2. 智能指针： shared_ptr 、 unique_ptr 和 weak_ptr

shared_ptr （共享所有权指针）

- **作用**： shared_ptr 是一个引用计数的智能指针，可以有多个指针共享同一个对象。当最后一个指向对象的 shared_ptr 被销毁时，内存自动释放。
- **引用计数**：每当一个新的 shared_ptr 被复制或赋值时，引用计数增加；当 shared_ptr 被销毁时，引用计数减少。当引用计数为0时，自动调用 delete 释放内存。
  示例：

```
std::shared_ptr<int> ptr1 = std::make_shared<int>(10);  // 引用计数 = 1
std::shared_ptr<int> ptr2 = ptr1;  // 引用计数 = 2
ptr2.reset();  // ptr2 销毁，引用计数 = 1
```

unique_ptr （独占所有权指针）

- **作用**： unique_ptr 拥有对象的唯一所有权，不能复制和赋值，只能通过移动语义转移所有权。这确保了某块动态内存只被一个指针管理。
- **移动语义**： unique_ptr 可以通过 std::move 将所有权从一个 unique_ptr 转移到另一个，但不能复制。
  示例：

```
std::unique_ptr<int> ptr1 = std::make_unique<int>(10);
// std::unique_ptr<int> ptr2 = ptr1;  // 错误，不能复制
std::unique_ptr<int> ptr2 = std::move(ptr1);  // 所有权转移
```

weak_ptr （弱引用指针）

- **作用**： weak_ptr 不参与对象的引用计数。它用于解决 shared_ptr 的**循环引用**问题， weak_ptr 可以安全地引用 shared_ptr 管理的对象，但不会延长对象的生命周期。
- **使用 lock()**：在使用 weak_ptr 时，需要通过 lock() 方法获取 shared_ptr ，以确保对象还存在。
  示例：

```
std::shared_ptr<int> sp = std::make_shared<int>(10);
std::weak_ptr<int> wp = sp;  // wp 不增加引用计数
if (std::shared_ptr<int> locked = wp.lock()) {
    std::cout << *locked << std::endl;  // 确保对象还存在
}
```

循环引用与 `weak_ptr` 的应用：

- **循环引用问题**：如果两个对象通过 `shared_ptr` 互相引用，它们的引用计数永远不会变为 0，导致内存泄漏。

```
class A;
class B;
std::shared_ptr<A> a = std::make_shared<A>();
std::shared_ptr<B> b = std::make_shared<B>();
a->ptrB = b;
b->ptrA = a;  // 循环引用，无法释放内存
```

- **解决方案**：将其中一个指针改为 `weak_ptr`，从而打破循环。

```
class A {
public:
    std::weak_ptr<B> ptrB;
};
class B {
public:
    std::shared_ptr<A> ptrA;
};
```

3. 使用 `make_shared` 和 `make_unique`

- **推荐使用**：使用 `make_shared` 和 `make_unique` 来动态分配内存。它们在分配内存时提供了更好的性能，并且避免了手动调用 `new` 和 `delete` 带来的错误。
- **性能优化**：`make_shared` 通过一次性分配对象和引用计数器的内存，减少了内存分配的次数。

示例：

```
std::shared_ptr<int> sp = std::make_shared<int>(10);  // 推荐
std::unique_ptr<int> up = std::make_unique<int>(20);  // 推荐
```

4. 智能指针的赋值与复制语义

- **`shared_ptr` 的赋值与复制**：
  - `shared_ptr` 支持复制与赋值，每次复制都会增加引用计数，直到所有 `shared_ptr` 离开作用域，引用计数变为 0 时，自动释放内存。

- **`unique_ptr` 的移动语义**：
  - `unique_ptr` 只能通过移动语义来转移所有权，防止内存被多个指针管理。
  - 复制和赋值是不允许的，因为 `unique_ptr` 保证了独占所有权。

5. 总结：智能指针与安全内存管理

- 使用 `shared_ptr` 来实现对象的共享所有权，使用 `unique_ptr` 来实现独占所有权，使用 `weak_ptr` 来避免循环引用。
- 使用 `make_shared` 和 `make_unique` 来创建智能指针，避免手动管理动态内存。
- 通过智能指针的引用计数机制，确保动态分配的内存在不再需要时自动释放，从而避免内存泄漏。

考题示例：

1. **简答题**：解释 `shared_ptr` 和 `weak_ptr` 的区别，并说明 `weak_ptr` 是如何解决循环引用问题的。
2. **填空题**：在 `shared_ptr` 的引用计数为 0 时，智能指针会自动调用 _____ 来释放内存。
3. **代码阅读题**：给定一段代码，指出其中的内存泄漏问题，并说明如何使用智能指针进行改进。