复制构造函数/赋值运算符、析构函数

- **关键概念**：可以通过将复制构造函数或赋值运算符声明为 `= delete` 来禁止类的复制构造或赋值。
  - **示例**： `std::unique_ptr` 无法被复制，因为它独占资源管理。
- **三法则**：如果一个类有非空的析构函数，那么它通常需要一个复制构造函数和复制赋值运算符（反之亦然），这就是所谓的“三法则”。
  - **示例**：管理资源的类（如文件句柄或动态内存）通常需要这三个函数。
- **声明复制构造函数**：这将禁止编译器生成默认构造函数。如果你还需要一个默认构造函数，应该声明 `= default` 。
  - **示例**： `class MyClass { MyClass(const MyClass&) = delete; MyClass() = default; }`

预计考题：

- 声明复制构造函数为 `= delete` 会有什么效果？
- 为什么具有非空析构函数的类需要复制构造函数？
- 如果声明了复制构造函数但没有默认构造函数会发生什么？

像值类型和指针类型的类

- **像值类型的类**：管理自己的资源，需要复制构造函数、赋值运算符和析构函数。
  - **示例**：表示一副牌的类，自己管理数据。
- **像指针类型的类（有复制语义）**：可能允许资源的浅复制，类似于 `shared_ptr` ，它使用引用计数。
  - **示例**： `std::shared_ptr` 在复制时增加其引用计数。
- **像指针类型的类（有移动语义）**：不支持复制，但允许所有权转移（移动语义），如 `unique_ptr` 。
  - **示例**： `std::unique_ptr` 转移所有权后会使原指针无效。

预计考题：

- 在什么情况下你会使用具有移动语义的类而不是复制语义？
- 为什么 `shared_ptr` 需要管理引用计数？

交换函数 vs. `std::swap`

- **重排序算法**：像 `std::sort` 这样的算法使用交换操作来重新排列元素。

- **类型特定的交换**：类可以定义自己的交换函数来进行浅复制或所有权转移，通常比 `std::swap` 性能更好。
  - **示例**：在赋值运算符中使用 `swap(this, other)` 。
- **在赋值操作中的使用**：可以在赋值运算符中使用交换来避免不必要的复制。例如，类的复制赋值运算符可以通过交换其数据成员来实现。
  - **示例**： `MyClass& operator=(MyClass rhs) { swap(*this, rhs); return *this; }`

预计考题：

- 为什么在赋值操作中使用类型特定的交换有利?
- 交换如何提高重排序算法的性能?

lvalues、rvalues 和引用的语义

- **lvalues**：是持久性的实体，出现在赋值运算符的左侧，例如变量。
  - **示例**： `int a = 5; int& ref = a;`
- **rvalues**：是临时的实体，出现在赋值运算符的右侧，用于表达式中。
  - **示例**： `int&& rref = 5 * 3;`
- **值传递**：值传递使用 rvalues，而引用传递使用 lvalues。

预计考题：

- lvalue 和 rvalue 引用有什么区别?
- 在函数签名中什么时候使用 rvalue 引用?

移动语义

- `std::move` ： 将 lvalue 视为 rvalue，从而允许其资源被转移（移动）。
  - **示例**： `int && rref = std::move(j); // j 的资源现在可以被移动`
- **移动构造函数和移动赋值运算符**：这些函数转移资源的所有权，而不是复制。移动后，源对象处于“已移动”状态，但仍然必须是可销毁的。
  - **示**
    例：

```
class MyClass { MyClass(MyClass&& other) { data = other.data; other.data = nullptr; } }
```

预计考题：

- `std::move` 的作用是什么?
- 为什么移动赋值运算符对于性能至关重要?
- 移动后的对象在移动操作后应确保什么?