第38条：为按值传递设计函子类。

“为按值传递设计函子类”意味着在设计函子类时，应该考虑以下两个方面：

1. **无状态或状态可复制**：如果函子类是无状态的，或者它的状态是可复制的（例如，它只包含基本数据类型或没有动态分配的资源），那么它可以安全地通过值传递。这是因为按值传递会创建函子对象的副本，如果状态不可复制，这可能会导致问题。
2. **性能考虑**：按值传递可能会带来性能开销，因为需要复制对象。但如果函子类很小，这种开销通常是可以接受的。如果函子类很大或者复制成本很高，那么按值传递可能不是最佳选择。

第39条：使谓词成为纯函数。

1. **谓词（Predicate）**：返回布尔值或可以隐式转换为布尔值的函数。在STL中广泛使用，特别是在关联容器的比较操作和作为算法（如 find\_if 和各种排序算法）的参数。
2. **纯函数（Pure Function）**：一个函数的输出仅依赖于其输入参数，并且没有副作用。对于相同的输入，纯函数的输出不会改变，也不会影响任何外部状态。
3. **谓词类（Predicate Class）**：一个重载了 operator() 的仿函数类，其中 operator() 函数的行为类似于谓词，返回布尔值或可转换为布尔值的类型。在STL需要谓词的地方，可以使用谓词类。

第40条：使仿函数类具有适应性。

在STL中，适配性指的是一个仿函数类能够被STL算法和组件识别和使用的能力。为了使仿函数类具有适配性，你需要确保它们符合STL对某些特定属性的要求。

以下是一些使仿函数类具有适配性的方法：

1. **类型成员**：某些STL算法和组件需要仿函数类提供特定的类型成员，例如：
   * result\_type：表示 operator() 返回的类型。
   * argument\_type（对于一元仿函数）或 first\_argument\_type 和 second\_argument\_type（对于二元仿函数）：表示 operator() 的参数类型。
2. **标准特性**：你可以使用STL提供的标准特性类（如 std::function\_traits 或 std::result\_of）来帮助STL算法推断仿函数类的属性。
3. **继承**：你可以让仿函数类继承自标准库中的适配器类，如 std::unary\_function 或 std::binary\_function，这些适配器类提供了必要的类型成员。

第41条：理解ptr\_fun、puncture\_fun和puncture\_fun\_ref的原因。

以下是对这三个适配器的解释：

* **ptr\_fun**：这是一个函数适配器，它将普通函数（非成员函数）转换成一个可以用于STL算法的仿函数。这样，你就可以将普通函数作为参数传递给需要仿函数的STL算法。
* **mem\_fun**：这是一个成员函数适配器，它将成员函数（非静态成员函数）转换成一个可以用于STL算法的仿函数。当你需要在STL算法中调用对象的成员函数时，可以使用 mem\_fun。
* **mem\_fun\_ref**：与 mem\_fun 类似，但是它是用于引用传递的对象。当你传递的是对象的引用而不是指针时，应该使用 mem\_fun\_ref。

这三个适配器存在的原因包括：

1. **兼容性**：在STL算法中，通常需要仿函数作为参数。如果你的函数或成员函数不符合仿函数的签名，你可以使用这些适配器来使其兼容。
2. **灵活性**：它们增加了在STL算法中使用不同类型函数的灵活性。例如，你可以使用 find\_if 算法结合 ptr\_fun 来查找第一个满足某个条件的元素。
3. **重用性**：通过使用这些适配器，你可以重用已有的函数或成员函数，而不必为STL算法专门编写新的仿函数。

第42条：确保less<T>表示运算符&lt;。

使用STL编程

条款43：优先使用算法调用而不是手写循环。

原因有三：效率：算法通常比程序员产生的循环更有效。

正确性：编写循环比调用算法更容易出错。

■可维护性：算法调用通常会产生比相应的显式循环更清晰、更直接的代码。

如果您需要一个执行相当简单的操作的循环，但需要令人困惑的绑定器和适配器，或者如果您要使用算法则需要一个单独的仿函数类，那么您可能最好只编写循环。最后，如果你需要在循环中做一些相当长和复杂的事情，尺度会向算法倾斜，因为长时间的复杂计算通常应该被转移到单独的函数中。一旦你把循环体移到一个单独的函数中，你几乎可以肯定地找到一种方法把这个函数传递给一个算法（通常是for\_each），这样得到的代码是直接的。

第44条：优先使用成员函数而不是同名的算法。

以下是一些原因说明为什么应该优先使用成员函数：

1. **性能优化**：成员函数通常是为了特定容器类型而设计的，因此它们可能比通用的STL算法更高效。
2. **接口一致性**：成员函数的接口可能与容器本身更加一致，使得代码更加易于理解和维护。
3. **减少错误**：使用成员函数可以减少因参数不匹配或算法使用不当而导致的错误。

以下是对每个算法的解释：

* **count**：计算序列中等于给定值的元素个数。它对整个序列进行线性搜索，时间复杂度为 O(n)。
* **find**：查找序列中第一个等于给定值的元素。如果找到，返回一个指向该元素的迭代器；如果没有找到，返回结束迭代器。它也是线性搜索，时间复杂度为 O(n)。
* **binary\_search**：在已排序的序列中查找给定值。如果找到，返回 true；否则返回 false。它使用二分查找，时间复杂度为 O(log n)。
* **lower\_bound**：在已排序的序列中查找第一个不小于（即大于或等于）给定值的元素的位置。它返回一个指向该元素的迭代器，如果所有元素都小于给定值，则返回结束迭代器。它使用二分查找，时间复杂度为 O(log n)。
* **upper\_bound**：在已排序的序列中查找第一个大于给定值的元素的位置。它返回一个指向该元素的迭代器，如果所有元素都不大于给定值，则返回结束迭代器。它也使用二分查找，时间复杂度为 O(log n)。
* **equal\_range**：在已排序的序列中查找等于给定值的所有元素的范围。它返回一个 pair，其中包含指向第一个等于给定值的元素的迭代器和指向第一个大于给定值的元素的迭代器。它实际上结合了 lower\_bound 和 upper\_bound 的功能，时间复杂度为 O(log n)。

条款46：考虑函数对象而不是函数作为算法参数。

1. **状态保持**：函数对象可以包含成员变量，这些变量可以在调用之间保持状态。这意味着函数对象可以在多次调用中记住信息，而普通函数则不能。
2. **可定制性**：函数对象可以在创建时通过构造函数参数进行定制，这使得它们比普通函数更加灵活。
3. **性能**：对于小型函数对象，调用它们通常比调用普通函数更快，因为它们可以内联展开。
4. **适配器兼容性**：函数对象可以与STL中的函数适配器（如 std::bind1st, std::bind2nd, std::not1, std::not2 等）一起使用，从而创建更复杂的操作。

第47条：避免产生只写代码。

第48条：始终#包含适当的标题。

任何时候你在头文件中使用任何组件时，都要确保提供相应的#include指令，即使你的开发平台允许你不使用它。当你发现自己移植到不同的平台时，你的勤奋会得到回报，减少压力。

项目49：学习破译STL相关的编译器诊断。

项目50：熟悉STL相关网站。