第 2 项：与 #defines 相比，更喜欢 const、枚举和内联。

✦ 对于简单常量，优先使用 const 对象或枚举而不是 #defines。

✦ 对于类似函数的宏，优先使用内联函数而不是#defines。

**原因**：

* **类型安全**：#define 是预处理器指令，它只是简单的文本替换，没有类型信息。而 const 对象和 enum 是类型安全的，编译器可以检查类型错误。
* **调试友好**：#define 定义的常量在编译时被替换，调试器中无法看到其符号名。而 const 对象和 enum 在调试器中是可见的。
* **作用域控制**：#define 是全局的，容易造成命名冲突。const 对象和 enum 可以限制在特定的作用域内（如命名空间、类或函数内）。
* **避免重复计算**：#define 定义的常量可能会在代码中多次替换，而 const 对象只会初始化一次
* **类型安全**：函数式宏（如 #define MAX(a, b) ((a) > (b) ? (a) : (b))）没有类型检查，容易导致错误。而内联函数是类型安全的。
* **避免副作用**：宏的参数可能会被多次求值，导致意外的副作用。例如：

**逻辑常量性的背景**

在C++中，const 成员函数承诺不会修改对象的“物理状态”（即对象的成员变量）。然而，有时我们需要在 const 成员函数中修改某些内部状态，而这些修改对客户端代码是不可见的。

需要记住的事情

✦ 声明 const 有助于编译器检测使用错误。 const 可以应用于任何范围的对象、函数参数和返回类型以及整个成员函数。

✦ 编译器强制执行按位常量，但您应该使用逻辑常量进行编程。

✦ 当 const 和非常量成员函数具有本质上相同的实现时，可以通过让非常量版本调用 const 版本来避免代码重复。

第 4 项：确保对象在使用之前已初始化。

 C++ 的规则规定，对象的数据成员在进入构造函数主体之前初始化。

C++提供了成员初始化列表（member initializer list），可以在构造函数体执行之前直接初始化成员变量，从而避免不必要的默认构造和赋值操作。

同样，不同翻译单元中定义的非局部静态对象的初始化的相对顺序是未定义的

 **解决初始化顺序问题**：

* 局部静态对象在第一次调用函数时初始化，确保依赖关系正确。
* 避免了非局部静态对象初始化顺序不确定的问题。

 **延迟初始化**：

* 对象只有在第一次使用时才会初始化，节省了启动时间和资源。

 **线程安全性（C++11 之后）**：

* C++11 标准规定，局部静态对象的初始化是线程安全的。

 **单例模式的实现**：

* 这种方法实际上是单例模式的一种常见实现方式。

需要记住的事情

✦ 手动初始化内置类型的对象，因为 C++ 有时只会自行初始化它们。

✦ 在构造函数中，优先使用成员初始化列表而不是构造函数体内的赋值。 按照在类中声明的顺序在初始化列表中列出数据成员。 ✦

通过用本地静态对象替换非本地静态对象，避免跨翻译单元的初始化顺序问题。

要记住的事情 ✦ 编译器可能会隐式生成类的默认构造函数、复制构造函数、复制赋值运算符和析构函数。

第 6 项：明确禁止使用您不想要的编译器生成的函数。

要记住的事情 ✦ 要禁止编译器自动提供的功能，请将相应的成员函数声明为私有并且不给出任何实现。 使用像 Uncopyable 这样的基类是实现此目的的一种方法。

第 7 项：在多态基类中将析构函数声明为 virtual。

**3. 虚析构函数的工作原理**

* **虚函数表（vtable）**：
  + 当一个类包含虚函数时，编译器会为该类生成一个虚函数表（vtable），其中存储了虚函数的地址。
  + 每个对象会包含一个指向虚函数表的指针（vptr）。
* **动态绑定**：
  + 在运行时，通过对象的 vptr 查找虚函数表，从而确定调用哪个函数。
  + 对于虚析构函数，编译器会生成一个特殊的析构函数链，确保从派生类到基类的析构函数依次被调用。
* **调用 delete ptr**：
  + 由于基类的析构函数是虚函数，delete 操作会通过虚函数表（vtable）找到实际的析构函数。
  + 首先调用派生类的析构函数（~Derived()）。
  + 然后调用基类的析构函数（~Base()）。

这个类有一个纯虚函数，所以它是抽象的，并且它有一个虚析构函数，所以你不必担心析构函数问题。

需要记住的事情

✦ 多态基类应该声明虚拟析构函数。 如果一个类有任何虚函数，它应该有一个虚析构函数。

✦ 未设计为基类或未设计为多态使用的类不应声明虚拟析构函数。

第 8 项：防止异常离开析构函数。

要记住的✦析构函数永远不应该发出异常。如果析构函数中调用的函数可能抛出异常，则析构函数应该捕获任何异常，然后吞噬它们或终止程序。

✦如果类客户端需要能够对操作过程中引发的异常做出反应，则类应该提供执行该操作的常规(即，非析构函数)函数。

第 9 条：切勿在构造或销毁期间调用虚函数。

您不应该在构造或销毁期间调用虚拟函数，因为这些调用不会按照您的想法执行，并且如果它们执行了，您仍然会不高兴。

在C++中，对象的构造顺序是从基类到派生类：

1. **基类构造函数**：首先调用基类的构造函数。
2. **派生类构造函数**：然后调用派生类的构造函数。

在基类构造函数执行时，派生类的数据成员尚未初始化

如果在基类构造函数中调用虚函数，并且该虚函数在派生类中被重写，那么理论上应该调用派生类的版本。然而，**此时派生类的数据成员尚未初始化**，如果派生类的虚函数访问了这些未初始化的数据成员，会导致未定义行为。

**基类构造函数中调用虚函数**：

* 不会调用派生类的重写版本，因为派生类的数据成员尚未初始化。
* 这是为了避免未定义行为

一旦派生类析构函数运行，对象的派生类数据成员就会采用未定义的值，因此 C++ 将它们视为不再存在。

要记住的事情 ✦ 不要在构造或销毁期间调用虚函数，因为此类调用永远不会转到比当前执行的构造函数或析构函数更派生的类。

第 10 项：让赋值运算符返回对 \*this 的引用。

Widget& Widget::operator=(const Widget& rhs)

{

Widget temp(rhs);

// make a copy of rhs’s data

swap(temp);

// swap \*this’s data with the copy’s

return \*this;

}

需要记住的事情

✦ 确保当一个对象被分配给它自己时，operator= 表现良好。 技术包括比较源对象和目标对象的地址、仔细的语句排序以及复制和交换。

✦ 如果两个或多个对象相同，请确保在多个对象上运行的任何函数都能正确运行。

第 12 项：复制对象的所有部分。

结论很明显：如果向类添加数据成员，则需要确保也更新复制函数。

任何时候您自己为派生类编写复制函数时，都必须注意复制基类部分。

尝试相反的方式——让复制构造函数调用复制赋值运算符——同样是荒谬的。

将拷贝构造函数和拷贝赋值运算符中的公共代码提取到一个私有成员函数中（通常命名为 init），然后在拷贝构造函数和拷贝赋值运算符中调用该函数。

要记住的事情 ✦ 复制函数应该确保复制对象的所有数据成员及其所有基类部分。

✦ 不要尝试用其中一种复制功能来实现另一种复制功能。 相反，将通用功能放在两者都调用的第三个函数中。

第 13 项：使用对象来管理资源。

事实上，这就是这个 Item 背后的一半想法：通过将资源放入对象中，我们可以依靠 C++ 的自动析构函数调用来确保资源被释放。

这个简单的示例演示了使用对象管理资源的两个关键方面：

■ 获取资源并立即将其移交给资源管理对象。 上面，createInvestment 返回的资源用于初始化将管理它的 auto\_ptr。 事实上，使用对象来管理资源的想法通常被称为资源获取即初始化（RAII），因为在同一条语句中获取资源和初始化资源管理对象是很常见的。 有时，获取的资源会分配给资源管理对象，而不是初始化它们，但无论哪种方式，每个资源都会在获取资源时立即移交给资源管理对象。

■ 资源管理对象使用其析构函数来确保资源得到释放。 因为当对象被销毁时（例如，当对象超出范围时）会自动调用析构函数，因此无论控制如何离开块，资源都会正确释放。 当释放资源的行为可能导致抛出异常时，事情可能会变得棘手，但这是第 8 条解决的问题，因此我们在这里不担心它。

RAII的核心思想是：**将资源的获取与对象的初始化绑定在一起，资源的释放与对象的析构绑定在一起**。

要记住的事情 ✦ 为了防止资源泄漏，请使用在构造函数中获取资源并在析构函数中释放资源的 RAII 对象。

✦ 两个常用的 RAII 类是 tr1::shared\_ptr 和 auto\_ptr。 tr1::shared\_ptr 通常是更好的选择，因为它在复制时的行为很直观。 复制 auto\_ptr 将其设置为 null。

第 14 项：仔细考虑资源管理类中的复制行为

RAII对象需要复制它所管理的资源，因此资源的复制行为决定了RAII对象的复制行为

。常见的RAII类复制行为是不允许复制和执行引用计数，但其他行为也是可能的。

要记住的事情

RAII API通常需要访问原始资源，因此每个RAII类都应该提供一种方法来获取它所管理的资源。访问可以通过显式转换或隐式转换。

一般来说，显式转换更安全，但隐式转换对客户端更方便。

第16项：在new和delete的相应用法中使用相同的形式。

规则很简单：如果你在一个new表达式中使用[]，你必须在相应的delete表达式中使用[]。如果你不在new表达式中使用[]，那么在匹配的delete表达式中也不要使用[]。

如果你在new表达式中使用[]，你必须在相应的delete表达式中使用[]。如果你没有在new表达式中使用[]，你也不能在相应的delete表达式中使用[]。

条款17：在独立语句中的智能指针中存储新对象。

**解决方案**：

1. 将智能指针的创建与函数调用分离。
2. 使用 std::make\_shared 或 std::make\_unique（C++14 及以上）来避免显式使用 new

将新对象存储在独立语句中的智能指针中。如果不这样做，在引发异常时可能会导致细微的资源泄漏。

第18条：使界面易于正确使用，难以错误使用。

**如果接口要求客户端记住某些操作（如手动释放资源），那么这种接口很容易被误用，因为客户端可能会忘记执行这些操作**

好的界面易于正确使用，难以错误使用。您应该在所有接口中努力实现这些特性。

促进正确使用的方法包括接口的一致性和与内置类型的行为兼容性。

防止错误的方法包括创建新类型、限制对类型的操作、约束对象值以及消除客户端资源管理责任。

shared\_ptr支持自定义删除器。这可以防止crossDLL问题，可以用来自动解锁互斥锁（见第14条）等。

第19条：把类设计当作类型设计。

要记住的事情类设计是类型设计。在定义新类型之前，请务必考虑本条目中讨论的所有问题。

第20条：偏好pass-by-reference-to-const而不是pass-byvalue。

图形用户界面, 文本

描述已自动生成

只有内置类型和STL迭代器以及函数对象类型才能合理地假设按值传递是廉价的。

要记住的事情更喜欢通过引用传递到常量而不是通过值传递。它通常更有效，并且避免了切片问题。

该规则不适用于内置类型和STL迭代器和函数对象类型。对于他们来说，按值传递通常是合适的。

第21条：当你必须返回一个对象时，不要试图返回一个引用。

1. **优先按值返回对象**：
   * 对于小型对象（如 Rational），按值返回是安全且高效的选择。
2. **避免返回局部对象的引用或指针**：
   * 任何返回局部对象引用或指针的函数都是错误的，必须避免。
3. **依赖编译器优化**：
   * 现代编译器会通过 RVO 和移动语义优化返回对象的性能。

永远不要返回一个指向本地堆栈对象的指针或引用，一个指向堆分配对象的引用，或者一个指向本地静态对象的指针或引用，如果有可能需要多个这样的对象的话。(Item 4提供了一个设计示例，其中返回对本地静态的引用是合理的，至少在单线程环境中是如此。

第22项：Decorated数据成员私有。

1. **始终将数据成员设为私有**：
   * 这是实现封装和保护数据的基本要求。
2. **提供必要的访问函数**：
   * 根据需求提供 getter 和 setter 函数，但不要为每个数据成员都提供。
3. **封装计算逻辑**：
   * 如果数据成员的值可以通过计算得到，将其封装在函数中，而不是直接存储。
4. **保持接口一致性**：
   * 所有公开接口都应该是函数，避免客户端混淆。

从封装的角度来看，实际上只有两个访问级别：private（提供封装）和everything else（不提供封装）。

将Declaration数据成员设置为私有。它为客户端提供了语法上统一的数据访问，提供了细粒度的访问控制，允许强制执行不变量，并为类作者提供了实现灵活性。

受保护的封装性并不比公共的封装性更强。

第23条：非成员非友元函数优于成员函数。

**1. 面向对象设计的核心原则**

* **封装性**：
  + 面向对象设计的核心原则之一是封装性，即将数据和实现细节隐藏起来，只暴露必要的接口。
  + 封装性的目的是降低模块之间的耦合性，提高代码的可维护性和可扩展性。
* **误解**：
  + 有些人认为面向对象设计就是简单地将数据和操作数据的函数捆绑在一起（即设计为成员函数）。
  + 这种理解是片面的，面向对象设计更强调数据的封装性，而不是形式上的捆绑。

封装的东西越多，我们改变它的能力就越强。这就是我们首先重视封装的原因：它为我们提供了改变事物的灵活性，只影响有限数量的客户端

图形用户界面, 文本, 应用程序

描述已自动生成

要记住的事情

优先使用非成员非友元函数而不是成员函数。这样做可以提高封装、打包灵活性和功能可扩展性。

第24条：当类型转换应用于所有参数时，Declare非成员函数。

图形用户界面, 文本, 应用程序

描述已自动生成

**如果需要对函数的所有参数进行类型转换（包括this指针指向的参数），则该函数必须是非成员函数。**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**当std：：swap对你的类型来说效率不高时，提供一个swap成员函数。确保你的swap不会抛出异常。**

**如果您提供会员交换，也提供调用会员的非会员交换。对于类（而不是模板），也要专门化std：：swap。**

**当调用swap时，使用std：：swap的using声明，然后不带命名空间限定地调用swap。**

**完全为用户定义类型专门化std模板是可以的，但永远不要尝试向std添加全新的东西。**

**第26条：尽可能长时间地使用后置变量定义。**

文本

中度可信度描述已自动生成

**尽可能长时间地使用变量定义。它增加了程序的清晰度，提高了程序的效率。**

**第27条：尽量减少强制转换。**

** 避免假设对象布局：**

* **不要假设对象在内存中的布局，因为不同编译器的实现可能不同。**
* **避免使用强制类型转换和指针算术操作。**

** 正确调用基类虚函数：**

* **在派生类的虚函数中，使用 BaseClass::function() 直接调用基类的函数，而不是通过强制类型转换。**

** 理解多重地址：**

* **在涉及继承时，理解对象可能具有多个地址，避免错误的类型转换。**

** 编写可移植代码：**

* **避免依赖特定编译器的对象布局，编写可移植的代码。**
* **尽量避免使用类型转换，尤其是在性能敏感的代码中。**
* **将类型转换封装在函数中，客户端代码调用该函数，而不是直接使用类型转换。**

文本, 信件

描述已自动生成

**第28条：避免返回对象内部的“句柄”。**

**引用、指针和迭代器都是句柄（获取其他对象的方法），返回对象内部的句柄总是冒着损害对象封装的风险。**

**避免返回对象内部的句柄（引用、指针或迭代器）。不返回句柄会增加封装性，帮助const成员函数执行const，并最大限度地减少悬空句柄的创建。**

**第29条：争取异常安全的代码。**

文本

中度可信度描述已自动生成

文本, 信件

中度可信度描述已自动生成

文本

低可信度描述已自动生成

文本

描述已自动生成

**项目30：了解内联的来龙去脉。**

**内联函数背后的思想是用它的代码体替换该函数的每个调用，它不需要博士学位。在统计数据中可以看到，这可能会增加目标代码的大小。在内存有限的机器上，过度的内联可能会导致程序太大空间即使使用虚拟内存，内联导致的代码膨胀也会导致额外的分页、降低的指令缓存命中率以及伴随这些情况的性能损失。**

文本

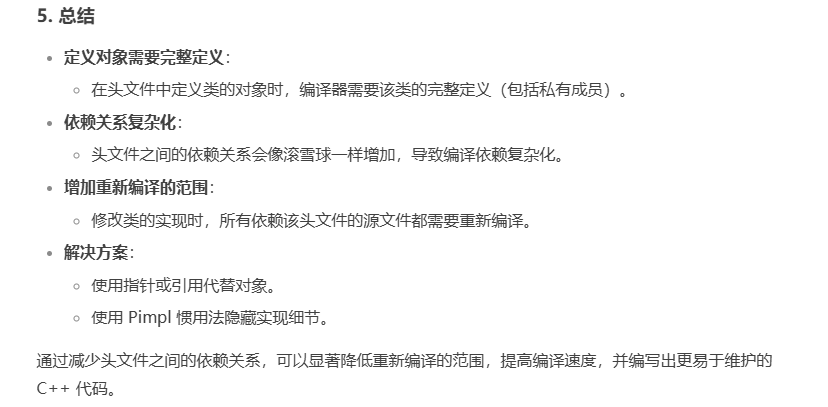
描述已自动生成

**要记住的事情**

**限制大多数内联到小的，经常调用的函数。这有利于调试和二进制可重复性，最大限度地减少潜在的代码膨胀，并最大限度地提高程序速度的机会。**

**不要仅仅因为函数模板出现在头文件中就声明它们内联。**

**项目31：最小化文件之间的编译依赖。**



图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**最小化编译依赖背后的一般思想是依赖于声明而不是定义。基于此思想的两种方法是Handle类和Interface类。**

**类库头文件应该以完整的和仅声明的形式存在。无论是否涉及模板，这都适用。**

文本

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成