1. 视C++语言为一个语言联邦

* C++高效编程守则视状况而变化，取决于你使用C++的哪一部分。
* C++总共分为四个次语言：C、OO C++、Template C++、STL。

1. 尽量以const，enum，inline替换#define

* 对于单纯常量，最好使用const对象或enum替换#define
* 对于形似函数的宏，最好使用inline函数替换#define

1. 尽可能使用const

* 将某些东西声明为const可帮助编译器侦测出错误用法。const可被施加与任何作用域内的对象、函数参数、函数返回类型、成员函数本体。
* 编译器强制实施bitwise constness，但你编写程序时应该使用“概念上的常量性”.
* 当const和non-const成员函数有着实质等价的实现时，令non-const版本调用const版本可避免代码重复。

1. 确定对象被使用前已被初始化（RAII）

* 为内置类型进行手工初始化，应为C++不保证初始化它们。
* 构造函数最好使用成员初值列，而不要在构造函数本体内使用赋值操作。初值列列出的成员变量，其排列次序应该和它们的class中的声明次序相同。
* 为避免“跨编译单元初始化次序”问题，请以local static对象替换non-local static对象。

1. 了解C++默默编写并调用哪些函数

* 编译器可以暗自为class创建default构造函数、copy构造函数、copy assignment操作符，以及析构函数。

1. 若不想使用编译器自动生成的函数，就该明确拒绝

* 为驳回编译器自动提供的机能，可将相应的成员函数声明为private并且不予实现。使用像uncopyable这样的base class也是一种做法。

1. 为多态基类声明virtual析构函数

* 带多态性质（polymorphic base class）应该声明一个virtual析构函数。如果class带有任何virtual函数，他就应该有一个virtual析构函数。
* classes的设计目的如果不是作为base class使用，或不是为了具备多态性，就不该声明virtual析构函数。

1. 别让异常逃离析构函数

* 析构函数绝不要吐出异常，如果一个被析构函数调用的函数可能抛出异常，析构函数应该捕捉任何异常，然后吞下他们或结束程序。
* 如果客户需要对某个操作函数运行期间抛出的异常做出反应，那么class应该提供一个普通函数执行该操作。

1. 决不在构造或析构函数中调用virtual函数

* 在构造和析构期间不要调用virtual函数，因为这类调用从不下降至derived class。

1. 令operator= 返回一个reference to \*this

* 令赋值操作符返回一个reference to \*this

1. 在operator=中处理”自我赋值“

* 确保当对象自我赋值时operator=有良好的行为。其中包括比较”来源对象“和”目标对象“的地址、精心周到的语句顺序、以及copy-and-swap。
* 确定任何函数如果操作一个以上的对象，而其中多个对象时同一个对象时，其行为仍然正确。

1. 复制对象时勿忘其每一个成分

* Copying函数应该确保复制”对象内的所有成员变量及“所有base class成分”。
* 不要尝试以某种copying函数实现另一个copying函数。应该将共同的机能放进第三个函数中，并由两个copying函数共同调用。

1. 以对象管理资源

* 为防止资源泄露，请使用RAII对象，他们在构造函数中获得资源并在析构函数中释放资源。
* 两个常被使用的RAII classes分别为shared\_ptr和weak\_ptr。

1. 在资源管理类中小心copying行为

* 复制RAII对象必须一并复制他所管理的资源，所以资源的copying行为决定RAII对象的copying行为。
* 普遍而常见的RAII class copying行为是：禁止copying（uncopyable）、施行引用计数法。

1. 在资源管理类中提供对原始资源的访问

* API往往要求访问原始资源，所以每一个RAII class应该提供一个“取得其所管理资源”的办法。
* 对原始资源的访问可能经由显式转换或隐式转换。一般而言显示转换比较安全，但隐式转换对客户比较方便。

1. 成对使用new和delete时要采用相同形式

* 如果你在new表达式中使用[]，必须在delete表达式中使用[]。没有在new中使用，delete中也不能使用。

1. 以独立语句将newed对象置入智能指针

* 以独立的语句将newed对象存储于（置入）只能指针内。如果不这么做，一旦异常被抛出，有可能造成难以察觉的资源泄露。

1. 让接口容易被使用，不易别误用

* 好的接口很容易被正确使用，不容易别误用。你应该在你的所以接口中能力达成这种性质。
* “促进正确使用”的办法包括接口的一致性，以及与内置类型的行为兼容。
* “阻止误用”的办法包括建立新类型、限制类型上的操作，束缚对象值，以及消除客户的资源管理责任。
* shared\_ptr支持定制删除器，这可防范DLL问题，可被用来自动解除互斥器等等。

1. 设计class犹如设计type

* class的设计就是type的设计。在定义一个新的type之前，请确定你已经考虑过以下所有主题：
* 新type的对象应该如何被创建和销毁？
* 对象的初始化和对象的赋值该有什么样的差别？
* 新type的对象如果被pass by value，意味着什么？
* 什么是新type的合法值？
* 新的type需要配合某个继承图系吗？
* 新的type需要什么样的转换？
* 什么样的操作符和函数对此新type而言是合理的？
* 什么样的标准函数应该驳回？
* 谁该取用新type的成员？
* 什么是新type的“未声明接口”？
* 新的type有多么一般化？
* 你真的需要一个新type吗？

1. 宁以pass-by-reference-to-const替换pass-by-value

* 尽量以pass-by-reference-to-const替换pass-by-value。前提是通常比较高效，并可避免切割问题（derived class被切割为base class）。
* 以上规则并不适用与内置类型，以及STL的迭代器和函数对象。对他们而言，pass-by-value往往比较合适。

1. 必须返回对象时，别妄想返回其reference

* 绝不要返回pointer或reference指向的一个local stack对象，或返回reference指向的heap-allocated对象，或返回pointer或reference指向一个local static对象，而有可能同时需要多个这样的对象。

1. 将成员变量声明为private

* 切记将成员变量声明为private。这可赋予客户访问数据的一致性、可细微划分访问控制、允诺约束条件获得保证，并提供class作者并充分实现弹性。
* protected并不比public更具有封装性。

1. 宁以non-member、non-friend替换member函数

* 宁以non-member、non-friend替换member函数，这样做可以增加封装性、包裹弹性和机能扩充性。

1. 若所有参数都需要类型转换，请为此采用non-member函数

* 如果你需要为某个函数的所有参数进行类型转换，那么这个函数必须是个non-member。

1. 考虑写出一个不抛出异常的swap函数

* 当std::swap对你的类型效率不高时，提供一个swap成员函数，并确定这个函数不抛出异常。
* 如果你提供一个member swap，也该提供一个non-member swap函数用来调用前者。对于classes，也请特例化std::swap。
* 调用swap时应针对std::swap使用using声明式，然后调用swap并且不带任何“命名空间资格修饰”。
* 为“用户定义类型”进行std templates全特化是好的，但千万不要尝试在std内加入对std而言全新的东西。

1. 尽量延后变量定义式的出现时间

* 尽可能延后变量定义式的出现。这样做可增加程序的清晰度并改善程序效率。

1. 尽量稍作转型动作

* 如果可以尽量避免转型，特别是在注重代码效率的代码中避免dynamic\_cast。如果有个设计需要转型动作，试着发展无需转型的替代设计。
* 如果转型是必要的，试着将它隐藏于某个函数背后。客户随后可以调用该函数，而不需要将转型放进自己的代码中。
* 宁可使用C++-style(新式)转型，不要使用旧式转型。前者很容易辨识出来，而且也比较有着分门别类的执掌。

1. 避免返回handles指向对象内部成分

* 避免返回handles（包括references、指针、迭代器）指向对象内部。遵守这个条款可增加封装性，帮助const成员函数的行为像个const，并将发生“虚假号码牌”的可能性将至最低。

1. 为异常安全而努力

* 异常安全函数即使发生异常也不会泄露资源或允许任何数据结构败坏。这样的函数区分为三种可能的保证：基本型、强烈型、不抛异常型。
* “强烈保证”往往能够以copy-and-swap实现出来，但强烈保证并非对所有的函数都可实现或具备显示意义。
* 函数提供的“异常安全保证”通常最高只等于其所调用各个函数的“异常安全保证“中的最弱者。

1. 透彻了解inline的里里外外

* 将大多数inline限制在小型、被频繁调用的函数身上。这可是日后的调试过程和二进制升级更容易，也可是千载的代码膨胀问题最小化，使程序的速度提升机会最大化。
* 不要只因为function template出现在头文件，就将他们声明为inline。

1. 将文件间的编译依存关系降至最低

* 支持”编译依存性最小化“的一般构想是：相依于声明式，不要相依于定义式。基于此构想的两个构想手段是handle classes和interface classes。
* 程序库头文件应该以“完全且仅有的声明式“的形式存在。这种做法不论是否涉及template都适用。

1. 确定你的public继承塑模出is-a关系

* public继承意味is-a。适用于base classes身上的每一件事情一定也适用于derived classes身上，因为每一个derived classes对象都是一个base class对象。

1. 避免遮掩继承而来的名称

* derived classes内的名称会遮掩base classes内的相同名称。在public继承下从来没有人希望如此。
* 为了让被遮掩的名称重见天日，可使用using声明式或转交函数。

1. 区分接口继承和实现继承

* 接口继承和实现继承不同。在public继承下，derived class总是继承base class的接口。
* pure virtual函数之具体指定接口继承。
* 简朴的impure virtual函数具体制定接口继承及缺省实现继承。
* non-virtual函数具体指定接口继承以及强制性实现继承。

1. 考虑virtual函数以外的其他选择

* 不考虑virtual函数的几种替代方案：
* 使用non-virtual interface（NVI）手法，那是Template Method设计模式的一种特殊形式。它以public non-virtual 成员函数包裹较低访问性（private或protected）的virtual函数。
* 将virtual函数替换为”函数指针成员变量“，这是Strategy设计模式的一种表现形式。
* 以std::function 成员变量替换virtual函数，因为允许使用可调用物搭配一个兼容于需求的签名式。这也是strategy设计模式的某种形式。
* 将继承体系内的virual函数替换为另一个继承体系内的virtual函数。这是Strategy设计模式的传统实现手法。

1. 绝不重新定义继承而来的non-virtual函数

* 绝不重新定义继承而来的non-virtual函数

1. 绝不重新定义继承而来的缺省参数值

* 绝不重新定义继承而来的缺省参数值，因为缺省参数值都是静态绑定的，而virtual函数——你唯一应该覆写的东西——却是动态绑定的。

1. 通过符合塑模出has-a或”根据某物实现出“

* 复合的意义和public继承完全不同
* 在应用域（实际存在的事物），复合意味着has-a。在实现域（vector等抽象的数据结构），复合意味is-implemented-in-terms-of（根据什么实现出）。

1. 明智而审慎地使用private继承

* private继承意味is-implemented-in-terms-of（根据什么实现出）。它通常比复合的级别低。但是当derived class需要访问protected base class的成员，或需要重新定义继承而来的virtual函数时，这么设计是合理的。
* 和复合不同，private继承可以造成empty class最优化。这对致力于“对象尺寸最小化“的程序库开发者而言，可能很重要。

1. 明智而审慎地使用多重继承

* 多重继承比单一继承复杂。他可能导致新的歧义性，以及virtual继承的需要。
* virtual继承会增加大小、速度、初始化复杂度等等成本。如果virtual base class不带任何数据，将是最具实用价值的情况。
* 多重继承的确有正当用途。其中一个情节涉及”public继承某个interface class“和”private继承某个协助实现的class“的两相组合。