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。

2. 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

3. 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。

4. 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。

5. 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。

6. 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。

7. 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。

8. 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。

9. 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。

10.令 operator= 返回一个 reference to *this

- 令赋值操作符返回一个 reference to *this

11.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。

12.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。

13.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。

14.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。

15.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。

16.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。

17.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。

18.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。

19.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？

20.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。

21.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。

22.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。

23.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。

24.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。

25.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。

26.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。

27.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。

28.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。

29.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。

30.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。

31.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。

32.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。

33.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。

34.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。

35.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。

36.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

37.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。

38.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）。

39.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。

40.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。

41.了解隐式接口和编译期多态

- classes 和 templates 都支持接口和多态。
- 对 classes 而言接口是显示的（explicit），以函数签名为中心，多态则是通过 virtual 函数发生在运行期。
- 对 templates 参数而言，接口是隐式的（implicit），奠基于有效表达式。多态则是通过 templates 具现化和函数重载解析发生于编译期。

42.了解 typename 的双重意义

- 声明 template 参数时，前缀关键字 class 和 typename 可互换。
- 请使用关键字 typename 标识嵌套从属类型名称；但不得在 base class lists（积累列）或 member initialization list（成员初值列）内以它作为 base class 修饰符。

43.学习处理模块化基类的名称

- 可在 derived class templates 内通过”this->”指涉 base class templates 内的成员名称（vs2013 没有必要，可直接调用），或藉由一个明白写出的“base class 资格修饰符（g++ 4.8 无效）”完成。

44.将与参数无关代码抽离 templates

- templates 生成多个 classes 和多个函数，所以任何 template 代码都不该与某个造成膨胀的 template 参数产生相依关系。
- 因非类型模板参数而造成的代码膨胀，往往可以消除，做法是以函数参数或 class 成员变量替换 template 参数。
- 因类型参数而造成的代码膨胀，往往可降低，做法是让带有完全相同二进制表述的具体类型共享实现码。

45.运用成员函数模板接受所有兼容类型

- 请使用成员函数模板生成“可接受所有兼容类型”的函数。
- 如果你声明 member template 用于泛化 copy 函数或泛化 assignment 操作，你还是声明正常的 copy 构造函数和 copy assignment 操作符。

46.需要类型转换时请为模板定义非成员 friend 函数

- 当我们编写一个 class template，而它所提供之“从此 template 相关的”函数支持“所有参数之隐式类型转换”时，请将那些函数定义为“class template 内部的 friend 函

数”。（这一点和定义一个具体的 class 不同）。

47.请使用 traits classes 表现类型信息

- Traits classes 使得“类型相关信息“在编译器可用。它们以 templates 和 templates 特化完成实现。
- 整合重载技术后，traits classes 有可能在编译器对类型执行 if…else 测试。

48.认识 template 元编程

- template 元编程（TMP）可将工作由运行期移往编译器，因而得以实现早期错误侦测和更高的执行效率。
- TMP 可被用来生成“基于政策选择组合“的客户定制代码，也可用来避免生成对某些特殊类型并不合适的代码。

49.了解 new-handler 的行为

- set_new_handler 允许客户定义一个函数，在内存分配无法获得满足时被调用。
- Nothrow new 是一个颇为局限的工具，因为它只适合于内存分配；后继的构造函数调用还是可能抛出异常。

50.了解 new 和 delete 的合理替换时机

- 有许多理由需要写个自己的 new 和 delete，包括改善效能、对 heap 运用错误进行调试、收集 heap 使用信息。

51.编写 new 和 delete 时需要固守常规

- operator new 应该内含一个无线循环，并在其中尝试分配内存，如果它无法满足内存需求，就该调用 new-handler。他也应该有能力处理 0bytes 申请。class 专属版本则应该处理”比正确大小更大的（错误）申请“。
- operator delete 应该在收到 null 指针时不做任何事情。class 专属版本还应该处理“比正确大小还大的（错误）申请“。

52.写了 placement new 也要写 placement delete

- 当你写了一个 placement operator new，请确定也写了一个对应的 placement operator delete。如果没有这样做，你的程序可能会发生隐微而时断时续的内存泄露。
- 当你声明 placement new 和 placement delete，请确定不要无意识地遮掩了它们的正常版本。

53.不要轻忽编译器的警告

54.让自己熟悉标准程序库

55.让自己熟悉 boost 程序库