Item01：视C++为一个语言联邦

记住：

* C++可以分为4种次语言，包括（C、Object-Oriented C++、Template C++、STL），C++高效编程守则视情况而改变，取决于你使用C++的哪一部分。

Item02：尽量以const，enum，inline替换 #define

记住：

* 对于单纯常量，最好以const对象或enums替换#defines。
* 对于形似函数的宏（macros），最好改用inline函数替换#defines。

Item03：尽可能使用const

记住：

* 将某些东西声明为const可帮助编译器侦测出错误用法。Const可被施加于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。

Item04：确定对象被使用前已先被初始化

记住：

* 为内置型对象进行手工初始化，因为C++不保证初始化它们。
* 构造函数最好是很好成员初值列（member initialization list），而不要在构造函数本体内使用赋值操作（assignment），初始列列出的成员变量，其排列次序应该和它们在class中的声明次序相同。
* 为免除“跨编译单元之初始化次序”问题，请以local static对象替换non-local static对象。

Item05：了解C++默默编写并调用哪些函数

记住：

* 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。

Item06：若不想使用编译器自动生成的函数，要明确拒绝。

记住：

* 为驳回编译器自动（暗自）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。

Item07：为多态基类声明virtual析构函数

记住：

* Polymorphic（多态性质的）base classes应该声明一个virtual析构函数。如果class带有任何virtual函数，它就应该拥有一个virtual析构函数。
* Classes的设计目的如果不是作为base classes使用，或不是为了具备多态性质（polymorphically），就不应该声明virtual析构函数。

Item08：别让异常逃离析构函数

记住：

* 析构函数绝对不要吐出异常。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，然后吞下它们（不传播）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（而非在析构函数中）执行该操作。

Item09：绝不在构造和析构过程中调用virtual函数

记住：

* 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（比起当前执行构造函数和析构函数的那层）。

Item10：令operator=返回一个reference to \*this

记住：

* 令赋值（assignment）操作符返回一个reference to \*this。

Item11：在operator=中处理“自我赋值”

记住：

* 确保当对象自我赋值时operator=有良好行为。其中技术包括比较“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，而其中多个对象是同一个对象时，其行为仍然正确。

Item12：复制对象时勿忘记其每一个成分

记住：

* 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”。
* 不要尝试以某个copying函数实现另一个copying函数。应该将共同机能放进第三个函数中，并由两个copying函数共同调用。

Item13：以对象管理资源

记住：

* 为了防止资源泄露，请使用RAII对象，它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分别是tr1：shared\_ptr和atuo\_ptr。前者通常是较佳选择，因为其copy行为比较直观。若选择auto\_ptr，复制动作会使它（被复制物）指向null。

Item14：在资源管理类中小心copying行为

记住：

* 复制RAII对象必须一并复制它所管理的资源，所以资源的copying行为决定RAII对象的copying行为。
* 普通而常见的RAII class copying行为是：抑制copying、施行引用计数法（reference counting）。不过其他行为也都可能被实现。

Item15：在资源管理类中提供对原始资源的访问

记住：

* APIs往往要求访问原始数据（raw resources），所以每一个RAII class应该提供一个“取得其所管理之资源”的办法。
* 对原始资源的访问可能经由显式转换或隐式转换。一般而言显式转换比较安全（个人觉得一般情况下能显式就显式），但隐式转换对客户比较方便。

Item16：成对使用new和delete时要采取相同形式

记住：

* 如果你在new表达式中使用[ ]，必须在相应的delete表达式中也是用[ ]。如果你在new表达式中没有用[ ]，那么一定不要在delete中使用[ ]。

Item17：以独立语句将new**ed（new过了的）**对象置入智能指针

记住：

* 以独立语句将newed对象存储于（置于）智能指针内。如果不这样做，一旦异常被抛出，有可能导致难以察觉的资源泄露。

Item18：让接口容易被正确使用，不易被误用

记住：

* 好的接口很容易被正确使用，不容易被误用。应该在所有接口中努力达成这些性质。
* “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。
* 这tr1::shared\_ptr支持定制型删除器（custom deleter）。这可以防范DLL问题，可被用来自动解除互斥锁（mutexes）等等。

Item19：设计class犹如设计type

记住：

* Class的设计就是type的设计。有很多需要讨论的主题都在本Item。

Item20：宁以pass-by-reference-to-const替换pass-by-value

记住：

* 尽量以pass-by-reference-to-const替换pass-by-value。前者通常比较高效，并可避免切割问题。
* 以上规则并不适用于内置类型，以及STL的迭代器和函数对象。对它们而言pass-by-value比较适当。

Item21：必须返回对象时，别妄想返回其reference

记住：

* 绝不要返回pointer或reference指向一个local stack对象，或返回reference指向一个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。条款4已经为“在单线程环境中合理返回reference指向一个local static对象”提供了一份设计实例。

Item22；将成员变量声明为private

记住：

* 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作者以充分的实现弹性。
* Protected并不比public更具有封装性。

Item23：宁以non-member、non-friend替换member函数

记住：

* 宁可拿non-member、non-friend函数替换member函数。这样做可以增加封装性、包裹弹性（packaging flexibility）和机能扩展性。

Item24：若所有参数皆需类型转换，请为此采用non-member函数

记住：

* 如果你需要为某个函数的所有参数（包括被this指针所指的那个隐喻参数）进行类型转换，那么这个函数必须是个non-member

Item25：考虑写出一个不抛异常的swap函数

记住：

* 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。
* 如果提供了一个member swap，也该提供一个non-member swap用来调用前者。对于classes（而非templates），也请特化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内加入某些对std而言全新的东西。

Item26：尽可能延后变量定义式的出现时间

记住：

* 这样可以增加程序的清晰度并改善程序效率

Item27：尽量少做转型动作

记住：

* 尽量避免转型，特别是在注重效率的代码中避免dynamic\_casts。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。客户随后可以调整该函数，而不需要转型放进他们自己的代码内。
* 宁可使用C++-style（新式）转型，不要使用旧时转型，前者很容易辨识出来，而且也比较有着分门别类的执掌。

Item28：避免返回handles指向对象内部成分

记住：

* 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“虚吊号码牌（dangling handles）”的可能性降至最低。

Item29：为“异常安全”而努力是值得的

记住：

* 异常安全函数（Exception-safe functions）即使发生异常也不会泄露资源或允许任何数据结构败坏。

Item30：透彻了解inlining的里里外外

记住：

* 将大多数inlining限制在小型、被频繁调用的函数身上。这可使日后的调试过程和二进制升级（binary upgradability）更容易，也可使潜在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 templates出现在头文件，就将它们声明为inline。

Item31：将文件间的编译依存关系降至最低

记住：

* 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。
* 程序库头文件应该以“完全且仅有声明式”的形式存在。不论是否涉及templates都适用。

Item32：确定你的public继承塑模出is-a关系

记住：

* Public继承意味着：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对象也都是一个base class对象。

Item33：避免遮掩继承而来的名称

记住：

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

Item34：区分接口继承和实现继承

记住：

* 接口继承和实现继承不同。在public继承下，derived classes总是继承base class的接口。
* Pure virtual函数只具体指定接口继承。
* Impure virtual函数具体指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。

Item35：考虑virtual函数以外的其他选择

记住：

* Virtual函数的替代方案包括NVI方案及Strategy设计模式的多种形式。NVI手法自身是一个特殊形式的Template Method设计模式。
* 将机能从成员函数移到class外部函数，带来的一个缺点是，非成员函数无法访问class的non-public成员。
* Tr1::function对象的行为就像一般函数指针。这样的对象可接纳“与给定之目标签名式（target signature）兼容”的所有可调用物。

Item36：绝不重新定义继承而来的non-virtual函数

Item37：绝不重新定义继承而来的缺省参数值

记住：

* 绝不重新定义一个继承而来的缺省参数值，因为缺省参数值都是静态绑定，而virtual函数——你唯一应该覆写的东西——却是动态绑定的。

Item38：通过复合塑模出has-a或“根据某物实现出”

记住：

* 复合（composition）的意义和public继承完全不同。
* 在应用域，复合意味has-a（有一个）。在实现域，复合意味着is-implemented-in-terms-of（根据某物实现出）。

Item39：明智而审慎地使用private继承

记住：

* Private继承意味着is-implemented-in-terms-of（根据某物实现出）。它通常比复合地级别低。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（composition）不同，private继承可以造成empty base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。

Item40：明智而审慎地使用多重继承

记住：

* 多重继承比单一继承复杂。它可能导致新的歧义性，以及对virtual继承的需要。
* Virtual继承会增加大小、速度、初始化（及赋值）复杂度等等成本。如果virtual base classes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”的两项组合。

Item41：了解隐式接口和编译期多态

记住；

* Classes和templates都支持接口（interfaces）和多态（polymorphism）
* 对classes而言接口是显式的（explicit），以函数签名为中心。多态则是通过virtual函数发生于运行期。
* 对template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template具体化和函数重载解析发生于编译期。

Item42：了解typename的双重意义

记住：

* 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（基类列）或member initialization list（成员初始列）内以它作为base class修饰符。

Item43：学习处理模板化基类内的名称

记住：

* 可在derived class templates内通过“this->”指涉base class templates内的成员名称，或藉由一个明白写出的“base class资格修饰符”完成。

Item44：将与参数无关的代码抽离templates

记住：

* Template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（non-type template parameters）而造成的代码膨胀，往往可消除，做法是以函数参数或class成员变量替换template参数
* 因类型参数type parameters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代码。

Item45：运用成员函数模板接受所有兼容类型

记住：

* 请使用member function templates（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 templates用于“泛化copy构造”或“泛化assignment操作”，你还是需要声明正常的copy构造函数和copy assignment操作符。

Item46：需要类型转换时请为模板定义非成员函数

记住：

* 当编写一个class template，而它所提供之“与此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”。

Item47：请使用traits classes表现类型信息

记住：

* Traits classes使得“类型相关信息”在编译期可用。它们以templates和“templates特化”完成实现。
* 整合重载即使后，traits classes有可能在编译期对类型执行if…else测试。

Item48：认识template元编程

记住：

* Template metaprogramming（TMP，模板元编程）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适合的代码。

Item49：了解new-handler的行为

记住：

* Set\_new\_handler允许客户指定一个函数，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，因为它只适用于内存分配；后继的构造函数调用还是可能抛出异常。

Item50：了解new和delete的合理替换时机

记住：

* 有许多理由需要写自定的new和delete，包括改善能效、对heap运用错误进行调试、收集heap使用信息。

Item51：编写new和delete是需固守常规

记住：

* Operator new应该内含一个无穷循环，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。它也应该有能力处理0bytes申请。Class专属版本则还应该处理“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。Class专属版本则还应该处理“比正确大小更大的（错误）申请”。

Item52：写了placement new也要写placement delete

记住：

* 当你写一个placement operator new，请确定也写出了对应的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（非故意）地遮掩了它们的正常版本。

Item53：不要忽略编译器的警告

记住：

* 严肃对待编译器发出的警告信息。努力在你的编译器的最高（最严苛）警告级别下争取“无警告”的荣誉。
* 不要过度依赖编译器的报警能力，因为不同的编译器对待事情的态度并不相同。一旦移植到另一个编译器上，原本依赖的警告信息有可能消失。

Item54：让自己熟悉包括TR1在内的标准程序库

记住：

* C++标准程序库的主要机能由STL、iostreams、locales组成。并包含C99标准程序库。
* TR1添加了智能指针（tr1::shared\_ptr）、一般化函数指针（tr1::function）、hash-based容器、正则表达式以及另外10个组件的支持。
* TR1自身只是一份规范。为获得TR1提供的好处，你需要一份实物。一个好的实物来源是Boost。

Item55：让自己熟悉Boost

记住：

* Boost是一个社群，也是一个网站。致力于免费、源码开放、同僚复审的C++程序库开发。Boost在C++标准化过程中扮演深具影响力的角色。
* Boost提供许多TR1组件实现品，以及其他许多程序库。