# Effective C++

1.让自己习惯C++

**条款01：视C++为一个语言联邦。**

1. C++高效编程守则视状况而变化，取决于你十一C++的哪一部分。
2. 四个次语言：C、Object-Oriented C++、Template C++、STL。

**条款02：尽量以const, enum, inline替换#define，即以编译器替换预处理器**

1）使用#define在出现编译错误信息时无法追踪，因为定义的常量名没有进入记号表。

2）无法利用#define创建一个class专属常量，因为#define不重视作用域，也不能够提供封装性。

3）一个枚举类型（enumerated type）的数值可权充int被使用。enum hack的行为比较像#define而不像const。

4）对于形似函数的宏（marcos），最好改用inline函数替换#define。

**条款03：尽可能使用const**

1）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。

2）编译器强制实施bitwise constness，但编写程序时应该使用“概念上的常量性”（conceptual constness）

bitwise constness：成员函数只有在不更改对象任何成员变量（static除外），才可以说是const

3）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。（利用const\_cast和static\_cast强制类型转换）

**条款04：确定对象在被使用前已先被初始化**

1）读取未初始化的值会导致不明确的行为（undefined behavior）,尤其是指针。为内置型对象进行手工初始化，因为C++不保证初始化它们。

2）构造函数最好使用成员初值列（member initialization list），而不要在构造函数本体内使用赋值操作（assignment）。

3）成员初始化次序：base classes更早于derived classes被初始化，而class的成员变量总是以其声明的次序被初始化。

4）C++对定义于不同编译单元内的non-local static对象（非定于函数内的static对象）的初始化相对次序并无明确定义

解决办法：C++保证函数内的local static对象会在该函数被调用起见，首次遇上该对象定义式时被初始化。

将non-local static对象搬到自己的专属函数内（该对象在此函数内被声明为static），然后返回一个reference指向它所包含的对象。

**2.构造/析构/赋值运算**

**条款05：了解C++默默编写并调用哪些函数**

1）编译器为class创建default构造函数、copy构造函数、copy assignment操作符，以及析构函数。

**条款06：若不想使用编译器自动生成的函数，就明确拒绝**

1）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。

**条款07：为多态基类声明virtual析构函数**

1）当derived class对象经由一个base class指针删除，而该base class带着一个non-virtual析构函数，其结果未定义——通常造成“局部销毁”对象，形成资源泄漏。

2）如果class不含virtual函数，通常表示并不被用作一个base class。

3）每一个带有virtual函数的class都有一个相应的vptr（virtual table pointer），指向一个由函数指针构成的数组vptl（virtual table）。当对象调用某一virtual函数，实际被调用的函数取决于该对象的vptr所指的那个vptl——编译器在其中寻找适当的函数指针。

4）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（plymorphically），就不该声明virtual析构函数（class体积会由于vptr而会增加）。

5）析构函数的运作方式：最深层派生的class其析构函数最先被调用，然后其每一个base class的析构函数被调用。

**条款08：别让异常逃离析构函数**

1）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）或者结束程序。

2）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。

**条款09：绝不在构造和析构过程中调用virtual函数**

1）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）

**条款10：令operator=返回一个reference to \*this**

1）为了实现“连锁赋值”，赋值操作符必须返回一个reference指向操作符的左侧实参。这也适用于所有赋值相关运算，例如+=、-=。

**条款11：在operator=中处理“自我赋值”**

1）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。

2）确定任何函数如果操作一个以上的对象，而其中有多个对象是同一个对象时，其行为仍然正确。

**条款12：复制对象时勿忘其每一个成分**

1）copying函数应该确保复制“对象内所有成员变量”及“所有base class成分”。

2）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。

**3.资源管理**

**条款13：以对象管理资源**

1）获得资源后立刻放进管理对象（managing object）（资源取得时机便是初始化时机，Resource Acquisition Is Initialization;RAII），管理对象运用析构函数确保资源被释放。

2）auto\_ptr是个“类指针（pointer-like）对象”，即所谓“智能指针”，其析构函数自动对其所指对象调用delete。若通过copy构造函数或者copy赋值操作符复制，旧指针变成null，而复制所得的指针将取得资源的唯一拥有权。

引用计数型智慧指针（reference-counting smart pointer，RCSP）shared\_ptr，持续追踪共有多少对象指向某笔资源，并在无人指向它时自动删除该资源。shared\_ptr允许指定所谓的“删除器”。

PS：无法用于C++动态分配数组情形（不支持delete[]）。

**条款14：在资源管理类中小心copying行为**

1）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。

2）普遍而常见的RAII class copying行为时：抑制copying、施行引用计数法。

**条款15：在资源管理类中提供对原始资源的访问**

1）APIs往往要求访问原始资源，所以每一个RAII class应该提供一个“取得其所管理之资源的”方法。

2）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，但隐式转换（重载()操作符）对客户比较方便。

**条款16：成对使用new和delete时要采取相同形式**

1）如果你在new表达式中使用[]，必须在相应的delete表达式中也使用[]。如果不使用[]，一定不要在相应的delete表达式中使用[]。

**条款17：以独立语句将newed对象置入智能指针**

1）举例：processWidget(shared\_ptr<Widget>(new Widget), priority());

以上语句可能出现：在new Widget之后执行priority函数而出现异常，导致内存泄漏

避免这类问题的办法：使用分离语句，以独立语句将newed对象存储于智能指针内。

**4.设计与声明**

**条款18：让接口容易被正确使用，不易被误用**

1）“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。

2）“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。

3）shared\_ptr支持定制型删除器（custom deleter）。这可防范DLL问题，被用来自动解除互斥锁。所谓的“cross-DLL problem”，发生于“对象在动态链接程序库中被new创建，却在另一个DLL内被delete销毁”，在许多平台上，这一类“跨DLL之new/delete成对运用”会导致运行期错误。

**条款19：设计class犹如设计type**

如何设计高效的classes？首先必须了解面对的问题：

1）新type的对象应该如何创建和销毁？

2）对象的初始化和对象的赋值该有什么差别？

3）新type的对象如果被passed by value，意味着什么？

4）什么是新type的“合法值”？

5）新type需要配合某个继承图系（inheritance graph）吗？

6）新type需要什么样的转换？

7）什么样的操作符和函数对此新type而言是合理的？

8）什么样的标准函数应该驳回？

9）谁该取用新type的成员？

10）什么是新type的“未声明接口”？

11）新type有多么一般化？

12）你真的需要一个新type吗？

**条款20：宁以pass-by-reference-to-const替换pass-by-value**

1）pass-by-reference-to-const效率高得多：没有任何构造函数或析构函数被调用，因为没有新对象被创建。

2）以by reference方式传递参数可以避免slicing（对象切割）问题。

3）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往往比较适当。

**条款21：必须返回对象时，别妄想返回其reference**

1）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22：将成员变量声明为private**

1）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。

2）封装性与“当其内容改变时可能造成的代码破坏量”成反比。protected并不比public更具封装性。

条款23：宁以non-member、non-friend替换member函数

1）面向对象守则要求数据应该尽可能被封装。non-member non-friend函数比member函数更具封装性。

2）这样的做法可以增加包裹弹性（packaging flexibility）和机能扩充性。

条款24：若所有参数皆需类型转换，请为此采用non-member函数

1）如果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。

**条款25：考虑写出一个不抛异常的swap函数**

1）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。

2）如果你提供一个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。

3）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”。

4）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。

**5.实现**

**条款26：尽可能延后变量定义式的出现时间**

1）增加程序的清晰度并改善程序效率

**条款27：尽量少做转型动作**

1）const\_cast<T>(expression)：通常用来将对象的常量性移除，也是唯一有此能力的C++转型操作符

dynamic\_cast<T>(expression)：主要用来执行“安全向下转型”，即用来决定某对象是否归属继承体系中的某个类型，是唯一无法由旧式语法执行的动作，也是唯一可能耗费重大运行成本的转型动作。

reinterpret\_cast<T>(expression)：意图执行低级转型，实际动作（及结果）可能取决于编译器，不可移植。

static\_cast<T>(expression)：用来强迫隐式转换，例如将non-const对象转换为const对象等。也可以用来执行多种转换的反向转换，如将pointer-to-base转为pointer-to-derived。

宁可使用以上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职掌。

2）如果可以，尽量避免转型，特别是在注重效率的代码中避免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，例如使用类型安全容器，或者将virtual函数往继承体系上方移动。

3）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，而不需将转型放进他们自己的代码内。

**条款28：避免返回handles指向对象内部成分**

1）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”的可能性降至最低。

**条款29：为“异常安全”而努力是值得的**

1）“异常安全”的两个条件：不泄漏任何资源、不允许数据败坏。

2）异常安全函数（Exception-safe functions）提供以下三个保证之一：

基本承诺：如果异常被抛出，程序内的任何事物仍能保持在有效状态下，没有任何对象或数据结构会因此而败坏。

强烈保证：如果异常被抛出，程序状态不改变。如果函数成功就是完全成功，如果函数失败程序会回复到“调用函数之前”的状态。

不抛掷（nothrow）保证：承诺绝不抛出异常，因为总能完成它们原先承诺的功能。所有内置类型身上的所有操作都提供nothrow保证。

3）“强烈保证”往往能够以copy-and-swap实现出来，但“强烈保证”并非对所有函数都可实现或具备现实意义。

copy-and-swap策略：为你打算修改的对象做出一份副本，然后在那副本身上做一切必要修改。若有任何修改动作抛出异常，原对象保持未改变状态。待所有改变都成功后，再将修改过的副本和原对象在一个不抛出异常的操作中置换（swap）。

4）函数提供的“异常安全保证”通常最高只等于其所调用的各个函数的“异常安全保证”中的最弱者。

**条款30：透彻了解inlining的里里外外**

1）inline函数的观念：对此函数的每一个调用都以函数本体替换之。只是对编译器的一个申请，并不是强制命令。这项申请可以隐喻提出，即将函数定义于class定义式内，也可以明确提出。

2）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。

3）不要只因为function templates出现在头文件，就将它们声明为inline。

4）inline函数无法随着程序库的升级而升级，一旦程序库设计者改变该函数，所有用到该函数的程序都必须重新编译。

5）大部分调试器面对inline都束手无策。慎重使用inline便是对日后使用调试器带来帮助。

6）策略：一开始先不要将任何函数声明为inline，或至少将inlining施行范围局限在那些“一定成为inline”或“十分平淡无奇”的函数身上。

**条款31：将文件间的编译依存关系降至最低**

1）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。基于此构想的两个手段是Handle classes和Interface classes。

2）如果使用object reference或object pointers可以完成任何，就不要使用objects；可以只靠一个类型声明式就定义出指出该类型的reference和pointer。

如果能够，尽量以class声明式替换class定义式。当声明一个函数而用到某个class时，并不需要该class的定义。

为声明式和定义式提供不同的头文件。

2）程序头文件应该以“完全且仅有声明式”的形式存在。这种做法不论是否涉及template都适用。

**6.继承与面向对象设计**

**条款32：确定你的public继承塑模出is-a关系**

1）“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 class对象也都是一个base class对象。

**条款33：避免遮掩继承而来的名称**

1）derived classes内的名称会遮掩base classes的名称。在public继承下从来没有人希望如此，违背了is-a的关系。

2）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。

**条款34：区分接口继承和实现继承**

1）接口继承和实现继承不同。在public继承之下，derived classes总是继承base classes的接口。

2）pure virtual函数只具体指定接口继承。也可以为pure virtual函数提供定义，但调用它的唯一途径是调用时明确指出其class名称。

3）impure virtual具体指定接口继承及缺省实现继承。

4）non-virtual函数具体制定接口继承以及强制性实现继承。

5）由于不同类型的声明意味根本意义不相同的事情，因此声明成员函数时必须谨慎选择。避免错误：将所有函数声明为non-virtual，或者将所有成员函数声明为virtual。

6）80-20法则：一个典型的程序有80%的执行时间花费在20%的代码上。

**条款35：考虑virtual函数以外的其他选择**

1）使用non-virtual interface（NVI）手法，是Template Method设计模式的一种特殊形式。它以public non-virtual成员函数（接口）包裹较低访问性（private或protected）的virtual函数（具体实现）。

将virtual函数替换为“函数指针成员变量”，这是Strategy设计模式的一种分解表现形式，意味着对象具体的方法实现与类型无直接关系。

以tr1::function成员变量替换virtual函数，因而允许使用任何可调用搭配一个兼容于需求的签名式。也是Strategy设计模式的某种形式。

将继承体系内的virtual函数替换为另一个继承体系内的virtual函数。Strategy设计模式的传统实现手法。

2）将机能从成员函数移到class外部函数，带来的缺点是，非成员函数无法访问class的non-public成员。

3）tr1::function对象的行为就行一般函数指针，这样的对象可接纳“与给定之目标签名式兼容”的所有可调用物。

**条款36：绝不重新定义继承而来的non-virtual函数**

1）non-virtual函数都是静态绑定，函数调用与静态类型相关；virtual函数是动态绑定，函数调用与动态相关。

2）derived classes绝对不应该重新定义一个继承而来的non-virtual函数，否则便是出现设计矛盾。

**条款37：绝不重新定义继承而来virtual函数的缺省参数值**

1）静态类型：在程序中被声明时所采用的类型

动态类型：目前所指对象的类型，可在程序执行过程中经由赋值动作改变。

2）缺省参数值都是静态绑定，而virtual函数却是动态绑定。

**条款38：通过复合塑模出has-a或“根据某物实现出”**

1）复合（composition）的意义和public继承完全不同。

2）在应用域，复合意味着has-a。在实现域，复合意味着根据某物实现出。

**条款39：明智而审慎地使用private继承**

1）private继承：编译器不会自动将一个derived class对象转换为一个base class对象；base class的所有成员，在derived class都变成private属性。

2）private继承在软件设计层面没有意义，其意义只及于软件实现层面。

private继承意味is-implemented-in-term of(根据某物实现出)。它通常比复合的级别低。但当derived class需要访问protected base class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。

3）和复合不同，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。

**条款40：明智而审慎地使用多重继承**

1）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。

2）virtual继承会增加大小、速度、初始化（及赋值）复杂度等成本。如果virtual base classes不带任何数据，将是最具实用价值的情况。非必要不使用virtual bases，如果必须使用virtual base classes，尽可能避免在其中放置数据。

3）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”。

**7.模板与泛型编程**

**条款41：了解隐式接口和编译期多态**

1）classes和templates都支持接口（interfaces）和多态（polymorphism）。

2）对classes而言接口是显式的，以函数声明为中心。多态则是通过virtual函数发生于运行期。

3）对template参数而言，接口是隐式的，基于有效表达式。多态则通过template具现化和函数重载解析发生于编译器。

**条款42：了解typename的双重意义**

1）声明template参数时，前缀关键字class和typename可以互换。

2）请使用关键字typename标识嵌套从属类型名称（template内出现的相依于某个template参数的名称，例如C::const\_iterator）；但不得在base class lists（基类列）或member initialization list（成员初值列）内以它作为base class修饰符。

**条款43：学习处理模板化基类内的名称**

1）模板全特化：template针对类型特化，而且特化是全面性的，一旦类型参数被定下来，再没有其他template参数可供变化。

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

**条款44：将与参数无关的代码抽离templates**

1）templates生成多个class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。

2）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。

3）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现类型共享实现码（例如list<int\*>和list<const long\*>）。

**条款45：运用成员函数模板接受所有兼容类型**

1）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。

2）如果你声明member templates用于“泛化copy构造”或“泛化赋值操作”，你还是需要声明正常的copy构造函数和copy赋值操作符。

**条款46：需要类型转换时请为模板定义非成员函数**

1）当我们编写一个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”。

**条款47：请使用traits classes表现类型信息**

1）STL迭代器分类：

Input迭代器：只能向前移动，一次一步，只能读取一次

Output迭代器：只能向前移动，一次一步，只能涂写一次

Forward迭代器：前述两种分类所能做的每一件事，读或写其所指物一次以上。

Bidirectional迭代器：相比Forward迭代器还可以向后移动

random access迭代器：可以在常量时间内向前或向后跳跃任意距离

2）Traits是一个C++程序员共同遵守的协议，技术要求之一就是，它对内置类型和用户自定义类型的表现必须一样好。Traits classes使得“类型相关信息”在编译器可用，它们以templates和“templates特化”实现。

3）如何设计并实现一个traits class：

确认若干你希望将来可取得的类型相关信息。例如对迭代器而言，我们希望将来可取得其分类。

为该信息选择一个名称（例如iterator\_category）。

提供一个template和一组特化版本（例如iterator\_traits），内涵你希望支持的类型相关信息。

4）如何使用一个traits class：

建立一组重载函数（身份像劳工）或函数模板（例如doAdvance），彼此间的差异仅在于各自的traits参数。令每个函数实现码与其接受之traits信息相应和。

建立一个控制函数（身份像工头）或函数模板（例如advance），它调用上述那些“劳工函数”并传递traits class所提供的信息。

**条款48：认识template元编程**

1）Template metaprograming（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。

2）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不适合的代码。

**8.定制new和delete**

**条款49：了解new-handler的行为**

1）当operator new抛出异常以反映一个未获满足的内存需求之前，它会先调用一个客户指定的错误处理函数，一个所谓的new-handler。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。

2）一个良好设计的new-handler必须做以下事情：

让更多内存可被使用；安装另一个new-handler；卸除new-handler；抛出bad\_alloc（或派生自bad\_alloc）异常；不返回。

2）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。

**条款50：了解合理定制替换编译器提供的new和delete的时机**

1）检测运用错误。

收集动态分配内存之使用统计信息。

增加分配和归还的速度。

降低缺省内存管理器带来的空间额外开销。

弥补缺省分配器中的非最佳齐位。

将相关对象成簇集中。

获得非传统的行为。

**条款51：编写new和delete时需固守常规**

1）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0 bytes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”。

2）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”。

**条款52：写了placement new 也要写placement delete**

1）Widget \*pw = new Widget;共调用两个函数：用以分配的operator new，和Widget的default构造函数。

2）placement new是指operator new接受的参数除了一定会有的size\_t之外还有其他，例如接受一个指针“void\*”指向对象被构造之处。

operator delete如果接受额外参数，便成为placement delete。

3）如果一个带额外参数的operator new没有对应版的带相同额外参数的operator delete，那么new的内存分配动作需要取消并恢复旧观时就没有任何operator delete被调用，而出现内存泄露。

4）当声明placement new和placement delete，请确定不要无意识地遮掩了它们的正常版本。一个简单做法就是建立一个内含所有正常形式的new和delete的base class。

**9.杂项讨论**

**条款53：不要轻忽编译器的警告**

1）严肃对待编译器发出的警告信息，努力在编译器的最高警告级别下争取“无任何警告”的程序。

2）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由可能消失。

**条款54：让自己熟悉包括TR1在内的标准程序库**

1）C++标准程序库的主要机能由STL、iostream、locales组成，并包含C99标准程序库。

2）TR1代表“Technical Report 1”，是C++程序库工作小组对该份文档的称呼。TR1添加了智能指针（例如tr1::shared\_ptr）、一般化指针（tr1::function）、hash-based容器、正则表达式（regular expression）以及另外10个组件的支持。

3）TR1自身只是一份规范。为过得TR1提供的好处，你需要一份实物。一个号的实物来源是Boost。

**条款55：让自己熟悉Boost**

1）Boost是一个C++开发者集结的社群，网址是http://boost.org，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。

2）Boost提供许多TR1组件实现品，以及其他许多程序库。