Effective C++ 改善程序设计的55个具体做法

Emmm，从标题来看，读这本书的目的应是掌握C++程序设计的优良做法，了解其背后的原理

本书内容是所谓轻薄短小高密度的“专家经验积累”

导读：

本书的目的是告诉你如何有效使用C++，这里提供C++这个语言的使用导引，使你的软件易理解，易维护，可移植，可扩充，高效并且有着你所预期的行为。

两类忠告：一般性的设计策略，集中讨论“如何在两个不同做法中择一完成某项任务”；带有具体细节的特定语言特性，比如new无法找到足够内存时怎么行事。

第一章：让自己习惯C++

条款1：C++是语言联邦

理解C++要认识主要的次语言：C，Object-Oriented C++，模板C++，标准库STL

C：区块blocks 语句 预处理器preprocessor 内置数据类型 数组 指针

面向对象C++：类，构造析构，封装，继承，多态，虚函数（动态绑定）

模板C++：泛型编程，TMP，模板元变成

STL：C++标准库，模板程序库

从一个次语言切换到另一个，高效编程守则要求你改变策略。例如对C内置类型而言，passbyvalue通常比passbyreference高效，但当从cpartofc++转移到面向对象C++，由于用户自定义的构造函数和析构函数存在，passbyreferencetoconst往往更好。STL的迭代器和仿函数都是在C指针上构造出来的，所以用到了STL，又是passbyvalue更加高效

C++高效编程守则视状况而定，取决于你使用C++的哪一部分

条款2：尽量以const，enum，inline替换#define

所谓宁可以编译器替换预处理器

Define定义的量有可能没有进入记号表，会给解决问题增加难度，用const定义一个常量替换掉上面的宏（但是条件编译还是要用预处理）

define无法用来给一个类定义一个专属常量，可以用enum{ NumTurns = 5 };来定义，类似于#define NumTurns = 5，

define定义宏，宏看起来是函数，但是不会带来函数调用带来的额外开销，但是会带来很多优先级的问题，用template inline来代替，

对单纯常量，用const对象或enums来替换#define，对于形似函数的宏，用inline函数替换defines

条款3：尽可能使用const

除非有需要改动参数或者local对象，否则应该将变量声明为const，只不过多打6个字符，可以减少很多错误

const成员函数：使类接口容易被理解，那个函数可以改动对象内容而哪个函数不行，它们使操作const对象成为可能

在const成员函数中可以修改mutable修饰的成员属性

编译器强制使用bitwise constness，编写程序应还使用“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版本可以避免代码重复

条款4：确定对象使用前已被初始化过

为内置对象进行手工初始化，主要涉及到cpartofc++，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，排列次序应该和他们在class中声明相同。

为避免“跨编译单元之初始化次序”问题，应该用local static对象替换non-local static对象。也就是设计模式中的单例模式，返回引用函数十分的单纯，第一行定义并初始化一个local static 对象，第二行返回它。将使用这个对象替换为使用这个返回引用函数，就不会出现不同编译单元中初始化次序的不同而在调用时发生问题的情况。

第二章：构造/析构/赋值运算

条款5：了解C++默默编写并调用哪些函数

编译器会为空类声明一个拷贝构造函数，拷贝赋值运算符，一个析构函数，如果没有声明任何构造函数，编译器还会为你声明一个默认构造函数，并且这些函数都是public和inline的。编译器产生的析构函数是非虚的，除非这个类的基类声明有虚析构函数。Copy assignment操作符和copy 构造函数编译器创建的版本只是单纯的将来源对象的每一个non-static成员变量拷贝到目标对象。

如果类成员包含引用或其他的不可改指向不同对象的成员，赋值语句经C++编译就会报错，此时需要自己实现拷贝赋值运算符

条款6：若不想使用编译器自动生成的函数，就应该明确拒绝

如果你不希望class支持某一特定机能，只要不声明对应函数就可以了，但是copy assignment 函数和运算符却不吃这一套，因为如果有人调用，编译器会为你声明他们。

为了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这样的基类也是一种做法。

条款7：为多态基类声明virtual析构函数

如果派生类的对象经由一个基类指针被删除，而该基类带着一个非虚的析构函数，那通常对象的派生成分不会被销毁。

因此需要将析构函数设计为虚函数

如果一个类不含有虚函数，那么将它的析构函数设计为虚函数就要付出vptr和vtbl的空间，对象体积会增加。

只有当class内含至少一个vitrual函数，才为它声明virtual析构函数

总之，带有多态性质的基类应当声明一个虚析构函数，如果一个类带有任何虚函数，它也应该拥有一个虚析构函数；如果类的设计目的不是作为基类使用，不是为了具备多态性，就不该声明虚析构函数。

条款8：别让异常逃离析构函数

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（不让其传播）或者结束程序

如果客户需要对某个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9：绝不在构造和析构过程中调用virtual函数

虚函数在构造和析构期间不是虚函数，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到派生类，还是使用当前执行构造函数和析构函数的那一层

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

令赋值操作符返回一个reference to \*this，从而可以实现连续赋值

条款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，其中技术包括比较“来源对象”和“目标对象”的地址，测同验证；精心周到的语句顺序，复制对象所指东西之前别先删除此对象；以及copy-and-swap，就是传递值而非引用，然后交换，避免delete再new

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，其行为依然正确。

条款12：复制对象时勿忘其每一个成分

设计良好的面向对象系统会将对象的内部封装起来，只留两个函数负责对象拷贝复制。

Copying函数应确保复制“对象内的所有成员变量”及“所有base class”成分，因为base类中的很多的成员变量是private，所以调用base class的复制函数来复制base class部分的成员更好。

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。

第三章：资源管理

所谓资源，就是你用了将来就必须还给系统，

C++程序中最常使用的资源就是动态分配内存，分配了不归还将会导致内存泄漏，但内存只是必须管理的众多资源之一，其他如文件描述器，互斥锁，图形界面中的字型和笔刷，数据库连接，网络socket等。

不管哪一种资源，不再使用它时，都必须将它还给系统。

条款13：以对象管理资源

关键想法之一：获得资源后立刻放进管理对象，（智能指针在构造期间获得资源）

关键想法之二：管理对象运用析构函数确保资源被释放，（智能指针在析构期间删除资源）

此处可以借助智能指针，智能指针在被销毁之后会自动删除它所指之物，作为管理资源的对象确保资源被释放。

工厂模式如果返回Raw pointer很容易忘记delete，就造成了资源泄漏

智能指针如shared\_ptr会在引用计数为0时销毁资源。

Heap\_base 的资源用智能指针管理很合适，因为智能指针析构函数内部默认做delete。

条款14：在资源管理类中小心coping行为

如果自己实现资源管理对象类，这个类管理资源的方式是在构造期间获得资源，在析构期间释放资源，就实现了一个管理资源的类

上面这个类的对象如果被复制，会发生什么？

两种解决方案：禁止复制；对底层资源祭出“引用计数法”；

禁止复制：继承Uncopyable，

引用计数法：使用shared\_ptr来代替资源管理类中指向资源的raw pointer，这样可以增加计数功能，并且shared\_ptr当引用计数为0时析构函数删除所指对象这个操作是一个仿函数，因此可以用函数对象来指定引用为0时的操作，而不仅仅是删除。

复制底部资源：针对一份资源拥有任意数量的复件，而你需要“资源管理类”的唯一理由是，当你不再需要某个复件时确保它被释放。在此情况下复制资源管理对象，应该同时也复制其包含的资源。

转移底部资源的拥有权：某些罕见场合下只能有一个管理类对象指向未加工资源，发生复制了资源的所有权会从被复制物转移到目标物。

条款15：在资源管理类中提供对原始资源的访问

一些API直接处理原始资源，你用资源管理类对象将无法满足这样的API

例如share\_ptr保存一个资源，而一个API要用这个资源类型的指针，那么shared\_ptr就不能满足要求，解决这个问题的办法有显式转换和隐式转换

显式转换：用shared\_ptr提供的get成员函数。

隐式转换：shared\_ptr重载了指针取值操作符（\*和->）它们允许隐式转换为底部原始指针

APIs往往要求访问原始资源，所以每一个资源管理类应该提供一个“取得其所管理的资源”的办法。

对原始资源的访问可能经由显式转换或隐式转换，显式转换比较安全，隐式转换对用户比较方便。

条款16：成对使用new和delete时要采用相同形式

在new表达式中使用[]，必须在相应的delete表达式中也使用[]，如果在new表达式中不使用[]，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

以独立语句将newd对象存储到管理资源的对象（如智能指针）中。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漏，如资源被创建和资源被转换为资源管理对象两个时间点之间有可能会发生干扰。

第四章：设计与声明

让接口容易被正确使用，不容易被误用

条款18：让接口容易被正确使用，不易被误用

好的接口很容易被使用，不容易被误用，你应该在你的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

“组织误用“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支持定制型删除器，可以防范跨DLL的new和delete问题，可被用来自动解除互斥锁

条款19：设计class犹如设计type

C++犹如其他OOP语言一样，当定义新class，也就定义了新type。

设计class应带着语言设计者当初设计语言内置类型时一样的谨慎来研讨class的设计

几乎每一个class都要求你面对以下提问：

新type的对象如何被创建和销毁？：这涉及到你的class的构造函数和析构函数以及内存分配函数和释放函数

对象的初始化和对象的赋值该有什么样的差别？：这个问题的答案决定你的构造函数和赋值操作符的行为，以及其间的差异。很重要的是别混淆了“初始化”和“赋值”，因为他们对应不同的函数调用

新type的对象如果被passed by value（以值传递），意味着什么？：对class，copy构造函数用来定义一个type的pass by value该如何实现

什么是新的type的“合法值”？：对class的成员变量而言，通常只有某些数值集是有效的。那些数值集决定了你的class必须维护的约束条件，也就决定了你的成员函数“特别是构造函数，赋值操作符和所谓setter函数”必须进行的错误检查工作。

你的新type需要配合某个继承图系吗？如果你继承自某些既有的classes，你就受到那些classes的设计的舒服，特别是受到他们的函数时virtual或non-virtual的影响，如果你允许其他classes继承你的class，那会影响你所声明的函数特别是析构函数，是否为virtual

你的新type需要什么样的转换？如果你希望允许类型T1之物被隐式的转换为类型T2之物，就必须在class T1内写一个类型转换函数，或者在T2内写一个non-explicit-one-argument的构造函数。如果你只允许explicit构造函数存在，就得写出专门负责执行转换的函数，且不得为类型转换操作符或non-explicit-one-argument构造函数

什么样的操作符和函数对此新type而言是合理的？这个问题决定你应该为你的class声明哪些函数，其中某些该是member函数，某些则不是

什么样的标准函数应该驳回？声明为private的，如条款6

谁该用新type的成员？这个提问可以帮助你决定哪个成员为public，哪个成员为private或protected。它也帮助你决定哪一个classes和/或functions应该是friends，以及将他们嵌套于另一个之内是否合理。

什么是新type的“未声明接口”？它对效率，异常安全性以及资源运用提供何种保证？你在这方面提供的保证将为你的class实现代码加上相应的约束条件

你的新type有多么一般化？或许你定义的是一个type家族，果真如此你就应该定义一个class template而非一个class

你真的需要一个新type吗？如果只是定义新的derived class以便为既有的class添加技能，那么说不定单纯定义一或多个nonmember函数或templates，更能达到目标。

Class的设计就是type的设计，请在定义一个新type之前，请确定你已经考虑过本条款覆盖的所有讨论主题

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

Pass by value会产生副本，然后要使用copy构造函数，如果是class的对象进行传递的话，那类的成员有可能也要调用自身的拷贝构造函数，为了产生副本，然后函数结束后副本还要调用析构函数

Pass by referencetoconst 传递效率就会高很多，没有构造函数析构函数的调用。Const是因为不是副本了，避免改变原来的对象

Pass by referencetoconst也可以避免传递参数时发生slicing对象切割问题，当一个派生类对象以byvalue的方式传递并被视为一个基类对象时，基类的拷贝构造函数就会被调用，派生类独有的部分就会被切割掉，只留下base class对象，解决切割问题的办法，就是用by referencetoconst来传递参数，利用了多态的机制，传进来什么类型，就表现什么类型

一般而言可以合理假设内置类型和STL的迭代器和函数对象passbyvalue不昂贵，passbyvalue比较适当

条款21：必须返回对象时，别妄想返回其reference

绝不要试图返回pointer或reference指向一个local stack对象，因为local stack所在代码区间的代码结束后，这个对象就被销毁了，也绝不要返回pointer或reference指向一个local static对象，因为有可能同时需要多个这样的对象。该返回对象就返回对象吧，只是调用一次copy构造函数而已

条款22：将成员变量声明为private

Public成员变量可被用户通过对象直接访问，如果public接口内的每一样都是函数，那用户就不用思考通过对象调用成员时要不要加小括号[成员函数加而成员变量不加]，

将成员变量隐藏在函数接口的背后，可以为“所有可能的实现”提供弹性

将成员变量声明为private，这可以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更具封装性

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

封装开始讨论：如果某些东西被封装，它就不可再见，越多东西被封装，越少人可以看到它，越少人看到它，我们就有越大的弹性去改变它，因为我们的改变仅仅直接影响看到改变的那些人事物。越多东西被封装，我们改变那些东西的能力就越大。推崇封装，它使我们能够改变事物而只影响有限客户

能够访问private成员变量的函数只有class的member函数加上friend函数而已。

导致较大封装性的是non-member和non-friend函数，因为她们并不增加“能够访问class内之private成分”的函数数量。

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、机能扩充性

条款24：若所有参数皆需要类型转换，请为此采用non-member函数

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

条款25：考虑写出一个不抛异常的swap函数

如果swap缺省实现版的效率不足，尝试以下方式修改：

1. 提供一个public swap成员函数，让它高效地置换你的类型的两个对象值，绝不会抛出异常，因为高效版应该是基于内置类型，而内置类型是不会抛出异常的
2. 在你的class或template所在的命名空间内提供一个non-member swap，并令它调用上述swap成员函数
3. 如果你正编写一个class（非class template），为你的class特化std::swap。并用它调用你的swap成员函数
4. 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。
5. 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
6. 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
7. 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

第五章：实现

条款26：尽可能延后变量定义式的出现时间

只要你定义了一个变量而其类型带有一个构造函数或析构函数，那么当程序的控制流到达这个变量定义式时，你便得承受构造成本，当这个变量离开作用域时，你便得承受析构成本，即使这个变量最终未被使用也是如此

尽可能延后变量定义式的出现，可以增加程序的清晰度并且改善程序的效率

定义新的变量可以在构造的时候直接初始化，避免default构造然后又赋值，降低效率

条款27：尽量少做转型动作

旧式转型 (T)expression T(expression)

新式转型 :

const\_cast<T>( expression) 常量性转除

dynamic\_cast<T>(expression) 安全向下转型，用来决定某对象是否属于继承体系中的某个类型

reinterpret\_cast<T>(expression) 低级转型，直接作用内存，结果取决于编译器

static\_cast<T>(expression)强迫隐式转换，例如将non-const对象转换为const对象，无法将const转为non-const

用一个基类指针指向派生类对象可以说明单一对象可能拥有一个以上的地址，一旦使用多重继承，这种事就几乎一直发生着

使用dynamic\_cast通常是逆向在一个你认定为derived class对象身上指向derived class操作函数，但你的手上只有一个指向base的pointer或reference，有两个办法可以避免这个问题

1. 使用容器并在其中存储直接指向derivedclass对象的指针，
2. 使用虚函数，如果base中不做操作添加一个空的虚函数也是有必要的

绝对应该避免的应该是连串的dynamic\_cast

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\_casts。如果有个设计需要转型动作，试着发展无须转型的替代设计

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，而不需将转型放入它们自己的代码中

宁可使用新式转型，不要使用旧式转型。前者很容易辨识出来，而且也有分门别类的执掌

条款28：避免返回handles指向对象内部成分

Handles：指针，引用，迭代器，三者都是所谓号码牌

如果一个const的成员函数返回了指向private成员的引用给一个外部数据，那就破坏了封装性，通过这个引用就可以更改private成员。还有如果这个引用被得到了，而这个对象被销毁后这个引用还存在，就会出现这个引用指向一个不存在的对象。引用就空悬，虚吊了

对象内部一般指对象的成员变量和不被公开使用的成员函数

避免返回handles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“虚吊号码牌”的可能性降到最低。

条款29：为“异常安全”而努力是值得的

异常安全有两个条件

不泄露任何资源：

不允许数据败坏：

异常安全函数即使发生异常也不会泄漏资源或允许任何数据结构的败坏。这样的函数可以分为三种可能的保证：基本型，强烈性，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”并非对所有函数都可实现或具备现实意义

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

inline只是对编译器的一个申请，不是强制命令，这项申请可以隐喻提出，也可以明确提出，隐喻提出的方式是将函数定义在class定义式内。明确声明inline函数的做法则是在其定义式前加上关键字inline

inline无法伴随程序库升级而升级，如果f是程序库的一个inline函数，客户将“f函数本体”编进程序中，一旦程序设计者决定改变f，所有用到f的客户端程序都必须重新编译。

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

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

条款31：将文件之间的编译依存关系降到最低

当你修改了一点点的内容，结果一编译，发现整个世界都被编译了、、

问题在于C++没有很好地把接口从实现中分离

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 class和interface classes（通常不带成员变量，也没有构造函数，只有一个virtual析构函数以及一组pure virtual函数，用来描述整个接口）

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

第六章：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

一定要记住公开继承（public inheritance）意味“是一种（is-a）”的关系

如果你让class D（“derived”）以public 形式继承class B（“Base”），你便是告诉C++编译器说，每一个类型为D的对象同时也是一个类型为B的对象，反之不成立。

D比B表现出更特殊化的概念，B比D表现出更一般化的概念。你主张“B对象可以派上用场的地方，D对象一样可以派上用场”，每一个D对象都是一种B对象，反之如果你需要一个D对象，B对象无法效劳。

适用于B身上的每一件事情一定适用于D身上，因为每一个D对象也是一个B对象

条款33：避免遮掩继承而来的名称

Derived class内的名称会遮掩base class内的名称。在public继承下从来没有人希望如此，一个函数调用会现在derivedclass中找，找不到了才会在base class去找，被遮掩了的话，base class有也找不到

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，using声明式可以让derived class看到base class内的被using声明式声明过的所有同名函数。转交函数是在derived class内部将要使用的函数明确写出来

条款34：区分接口继承和实现继承

纯虚函数两个特性：它们必须被任何“继承了它们”的具象class重新声明，而且它们在抽象class中通常没有定义。因此：

声明一个pure virtual函数的目的是为了让derived classes只继承函数接口。

Impure virtual函数可以让derived classes继承其函数接口，但impure virtual函数会提供一份实现代码，derived classes可能覆写它。因此：

声明一个impure virtual函数的目的，是让derived class继承该函数的接口和缺省实现。

如果成员函数是一个non-virtual函数，意味是它并不打算在derived classes中有不同的行为。它表示不论derived class变得多么特异化，它的行为都不可以改变。

声明non-virtual函数的目的是为了令derived class继承函数的接口及一份强制性实现。

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

条款35：考虑virtual函数以外的其他选择

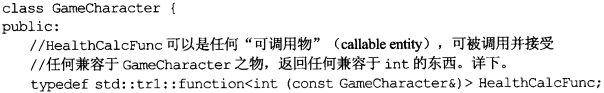
C++的derived classes可重新定义继承来的private virtual函数、

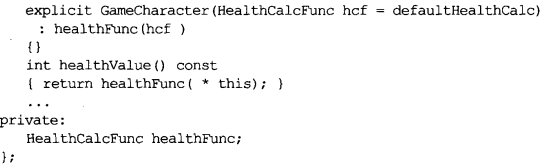
NVI手法：将virtual函数设为private，然后在一个non-virtual函数中调用这个virtual函数，这样的好处是可以在non-virtual中调用virtual函数的前后添加处理其他事物（事前工作比如锁定互斥器，记录日志，事后工作比如解锁互斥器，验证函数的事后条件等。）non-virtual函数被继承后也是相同的实现，就可以让用户专心修改virtual中的内容而设计者专心改non-virtual中的内容

由函数指针实现的Strategy模式：要求每一个人物的构造函数接受一个指针，指向一个健康计算函数，而我们可以调用该函数进行实际计算，这样就是说同一个类的不同对象处理健康的函数也可能不同，处理健康函数从成员函数变成了非成员函数。

由trl::function完成的Strategy模式：函数指针会带来参数、返回值类型的要求，降低了弹性；如果不再使用函数指针，而是改用一个类型为trl::function的对象，这些约束就挥发不见了。

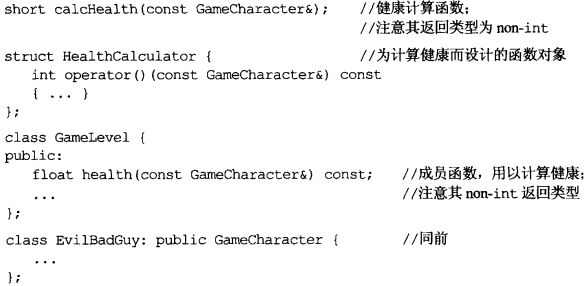
如std::trl::function<int (const GameCharacter&)>接受一个reference指向const GameCharacter，并返回int，不同于函数指针，任何的数据结构或是函数只要能找到这样的引用就可以被当做是这个参数，看下面截图（本书第一次截图）

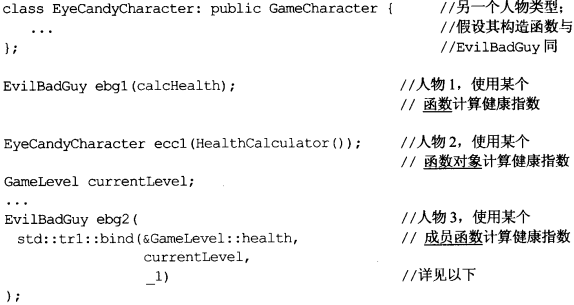




typedef std::trl::function<int (const GameCharacter&)> HealthCalFunc;

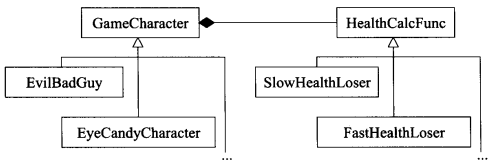
其中HealthCalFunc类型的成员变量在构造函数初始化时可以用各种各样的参数，只要它们有一个指向const GameCharacter并返回可以隐式转换为int类型的引用。也就是这个trl::function类型产生的对象是可以持有任何与此签名式兼容的可调用物。所谓兼容，意思是可调用物的参数可被隐式转换为const GameCharacter& ，而其返回类型可被隐式转换为int。如下：





最后一个bind是适配器，可以绑定函数对象，参数，将其组合起来。

设计模式中的Strategy：传统的实现手法是将继承体系内的virtual函数替换为另一个继承体系内的virtual函数。这是Strategy设计模式传统实现手法



每一个GameCharacter对象都含有一个指针，指向一个来自HealthCalcFunc继承体系的对象。

Virtual函数的替代方案包括NVI手法以及Strategy设计模式中的多种形式，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。

Trl::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

绝对不要重新定义继承而来的non-virtual函数。

绝对不要重新定义继承而来的non-virtual函数。

绝对不要重新定义继承而来的non-virtual函数。

条款37：绝不重新定义继承而来的缺省参数值

对象的所谓静态类型是它在程序中被声明时所用的类型

如一个指针被声明为基类，那它的静态类型就是基类，不管它真正指向什么，他们的静态类型都是基类。

而所谓的动态类型则是指“目前所指对象的类型”。

绝对不要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---你唯一该覆写的东西----却是动态绑定

Virtual函数是动态绑定（运行时绑定）的，而缺省参数值却是静态绑定（编译时绑定）的

如果你改变了虚函数的base类缺省参数值，当你调用derived class的虚函数时如果使用默认参数会发现，参数并没有改变，仍然是虚函数在base类中的缺省参数值。

如果你一定要让derived class有一个不同的缺省参数值，那么可以尝试用NVI手法，在public中的non-virtual中给不同的缺省参数值。

条款38：通过复合塑模出has-a或“根据某物实现出”

复合：当某种类型的对象内包含其他类型的对象，便是复合

Public 继承带有is-a的意义，复合也有它自己的意义，意味着has a

在应用域，复合意味has-a。在实现域，复合以为is-implemented-terms-of（根据某物实现出）

条款39：明智而审慎地使用private继承

由pr ivate继承而来的类有两条规则：

如果class之间继承关系是private，编译器不会自动将一个derived class对象转换为一个base class对象。

由private base class继承而来的所有成员在derived中都会变成private属性。

Private继承意味着is-implemented-in-terms of（根据某物实现出）。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 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

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（以及赋值）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”的两相组合

第七章：模板与泛型编程

C++templates的最初发展动机很直接：让我们得以建立“类型安全”的容器如vector，list和map。

后来，人们发现，容器很好，但泛型编程—写出的代码与其所处理的对象类型彼此独立---更好！

最终，人们发现，C++template机制自身是一部完整的图灵机：它可以被用来计算任何可计算的值。于是导出了模板元编程，创造出“在C++编译器内执行并于编译完成时停止执行”的程序。

容器反倒只成为C++template馅饼上的一小部分。

条款41：了解隐式接口和编译期多态

在模板中T类型的对象w必须支持哪一种接口，系由template中执行于w身上的操作来决定。w身上使用的这一组表达式便是类型T必须支持的一组隐式接口。

凡涉及w的任何函数调用，例如operator>和operator=!=，有可能造成template具现化，使这些调用得以成功。这样的具现行为发生在编译期。“以不同的template参数具现化function templates”会导致调用不同的函数，这便是所谓的编译期多态。

通常显式接口由函数的签名式（也就是函数名称、函数类型、返回类型）构成。

隐式接口而是由有效表达式组成，只要这个表达式能够有效，那这个隐式接口就是合法的

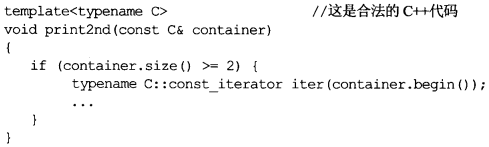
Classes和templates都支持接口和多态。

对classes而言接口使显式的，以函数签名为中心。多态则是通过virtual函数发生在运行期

对于templates参数而言，接口是隐式的，奠基于有效表达式。多态则是通过template具现化和函数重载解析发生于编译期。

条款42：了解typename的双重意义

任何时候你想在template中指涉一个嵌套从属类型名称，就必须在紧邻它的前一个位置放上关键字typename。



container不是从属类型名称，而C::const\_iterator是从属嵌套名称。

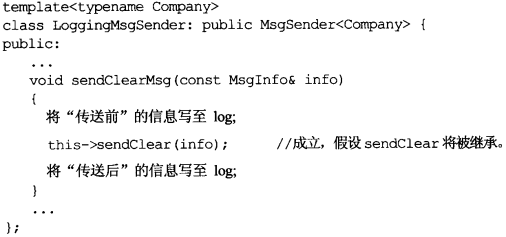
声明template参数时，前缀关键字class和typename可互换。

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

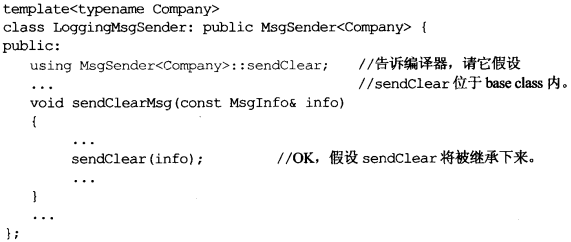
条款43：学习处理模板化基类内的名称

为了不让C++进入templatized base classes观察的行为失效。有三种办法，

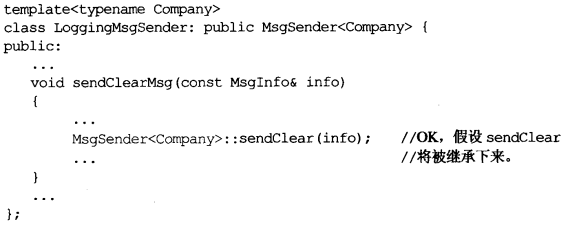
第一是在base class函数调用动作之前加上this->



第二是使用using声明式。条款33提到过的，使用后就基本全都用基类的函数了



第三是明确指出被调用的函数位于base class内：



条款44：将与参数无关的代码抽离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现类型共享实现码。

条款45：运用成员函数模板接受所有兼容类型

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

条款46：需要类型转化时请为模板定义非成员函数

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

涉及到traits编程技法，STL中使用过了，就是把不同迭代器内部把类型定义到一个统一的别名（typedef）中，然后在对使用到迭代器的算法中加一个迭代器类型的参数，获得这个参数的方法是用迭代器参数调用它自身的类型获取函数

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

整合重载技术（overloading）后，traits classes有可能在编译期对类型执行if…else测试

如何使用一个traits class：

建立一组重载函数（身份像劳工）或者函数模板，彼此间的差异只存在于各自的traits参数。令每个函数实现码与其接受的traits信息相应和。

建立一个控制函数（身份像工头）或函数模板，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

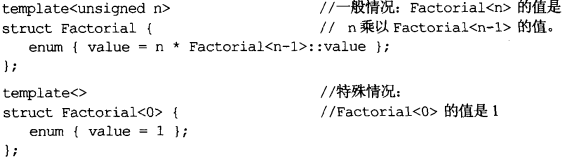
Template metaprogramming（TMP，模板元编程）是编写template-base C++程序并执行于编译期的过程。

所谓template metaprogram模板元程序是以C++写成、执行于C++编译器内的程序。一旦TMP程序结束执行，其输出，也就是从templates具现出来的若干C++源码，便会一如往常地被编译

TMP，模板元编程可将工作由运行期移往编译期，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，也可用来避免生产对某些特殊类型并不适合的代码

TMP起手程序，编译期计算阶乘



有了这个template metaprogram，只要你指涉Factorial<n>::value就可以得到n阶乘值。

循环发生在template具现体Factorial<n>内部指涉另一个template具现体Factorial<n-1>之时。和所有良好递归一样，我们需要一个特殊情况造成递归结束。这里的特殊情况是template特化体Factorial<0>

第8章：定制new和delete

C++可以手工管理内存。这样开发人员可以研究并学习他们的软件使用内存的行为特征，然后修改分配和归还工作，以求获得其所构建的系统的最佳效率（包括时间和空间）

本章焦点在于了解C++内存管理例程的行为。主角是分配例程和归还例程（也就是operator new和operator delete），配角是new-handler，这是当operator new无法满足客户的内存需求时所调用的函数

多线程环境下的内存管理，遭遇到单线程不曾有过的挑战。由于heap是一个可被改动的全局性资源，因此多线程系统充斥着发狂访问这一类资源的race conditions（竞速状态）出现机会。

operator new和operator delete只适合用来分配单一对象，Arrays所用的内存由operator new[]分配，并由operator delete[]归还

除了一些特殊情况，适用于operator new和operator delete的事也都适用于operator new[]和operator delete[]

STL容器使用的heap内存是由容器所拥有的分配器对象管理的，而不是由new和delete管理。

条款49：了解new-handler的行为

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。为了指定这个“用以内存处理不足”的函数，客户必须调用set\_new\_handler，那是声明于<new>的一个标准程序库函数

Namespace std {

typedef void (\*new\_handler)( );

new\_handler set\_new\_handler(new\_handler p) throw();

}

new\_handler是一个typedef，它可以定义出一个指针指向函数，该函数没有参数也不返回任何东西。

set\_new\_handler则是获得一个new\_handler的并返回一个new\_handler的函数、

set\_new\_handler的参数是个指针，指向operator new无法分配足够内存时该被调用的函数。

set\_new\_handler的返回值也是个指针，指向set\_new\_handler被调用前正在执行