**13、以对象管理资源（Use objects to manage resources）**

在一个区域开始的时候动态申请一块内存，在最后delete它。

如果在中间有break、continue、return、异常等，就会导致程序运行不到delete那就退出了。这样就会造成资源泄露。单纯依赖手动释放内存是行不通的。

本条款的思想：把资源放进对象内，利用C++析构函数自动调用的机制来确保资源释放。

“以对象管理资源的观念”常被称为“资源获得时机便是初始化时机” **RAII**（Resource Acquisition Is Initialization）。在获得资源的同一语句内用它初始化一个对象。

auto\_ptr用来管理指针，但是由于其拷贝和复制的行为诡异（复制后源对象变空），所以这个方法不好。

auto\_ptr的替代方案是引用计数型智能指针（RCSP reference-counting smart pointers）。其实就是shared\_ptr。但是shaerd\_ptr无法解决循环引用的问题。

**总结：**

1. 为防止资源泄露，请只用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。
2. 书上写的常被使用的是boost::shared\_ptr和std::auto\_ptr，但是现在auto\_ptr已经不行了，而boost::shared\_ptr已经是加入到std中了。另外还有weak\_ptr, unique\_ptr。

**14、在资源管理类中小心copying行为（Thinking carefully about copying behavior in resource-managing class）**

并不是所有的资源都是在堆上申请，用delete释放。

问题：当RAII对象发生复制时会发生什么？

1. 禁用copy。如果该RAII对象的复制操作不合理，那么就要禁止对它的复制操作，例如继承noncopyable。
2. 引用计数（又来到了shared\_ptr）。Shared\_ptr可以指定删除器。类的析构函数（无论是编译器生成还是手写的）会自动调用类中non-static成员的析构函数。
3. 复制底层资源。即进行深拷贝。
4. 转移底层资源的拥有权。例如auto\_ptr。

除非编译器生成的copying函数就是你想要的，否则请手写或删除它。

**总结：**

1. 资源的copying行为决定了RAII管理对象的copying行为。
2. 常见的RAII对象的copying行为是禁用和引用计数，但是上面说的其他两种也是可以的。

**15、在资源管理类中提供对原始资源的访问。（Provide access to raw resources in resource-managing class）**

很多情况下，资源管理类不得不提供对原始资源的访问，例如很多API只认原始资源。

一般有两种方式：

1. 显示转换。例如shared\_ptr的.get()方法。（但是如果一直要调用get方法来获得原始资源对象，程序员可能会变得烦，然后不想用RAII类）。
2. 隐示转换。例如shared\_ptr重载了->和\*操作符访问资源的方法。
3. 隐示转换。重载资源管理类的类型转换运算符。但是这种方法一定要慎用或者不用。

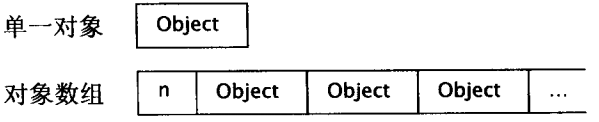
其实最好还是用显示转换。

**总结：**

1. 调用API往往要求使用原生资源，所以RAII类一般要提供访问原生资源的接口。
2. 获得原生资源的方法一般有两种：显示转换和隐示转换，一般显示转换比较安全，隐示转换可能更受使用者喜欢。

**16、成对使用new和delete时要采取相同的格式（Use the same form in corresponding uses of new and delete）**

一个指针指向单一对象还是数组是有区别的。一般而言单一对象的内存布局不同于数组的内存布局，数组可能还包含数组大小的记录。很多编译器是这样做的：



typedef string AddressLine[4]; //AddressLine是一个包含4个string的数组结构

string\*pal = new AddressLine; //应该用delete[]来释放。

应该要避免上面这种写法。

**总结：**

new和delete对应，new[]和 delete[]对应，千万不能错用。

**17、以独立语句将newed对象置入智能指针（Store newed objects in smart pointers in standalone statements）**

编译器对同一语句内的各个函数（或者说各项操作）的调用顺序可能会重新排列。

**总结：**

请单独用一行语句来将new的对象放入智能指针中，否则可能导致难以察觉（例如new一个对象和把对象放入智能指针之间发生异常）的资源泄露。

**18、让接口容易被正确使用，不易被误用（Make interfaces easy to use correctly and hard to use incorrectly）**

如果客户通过使用某个接口而没有得到他预期的结果，那这份代码就不应该编译通过。

明智而审慎的导入新类型对预防接口误用有奇效。

除非有好理由，否则应该尽量让你的types和内置的types的行为一样。一旦怀疑就拿int做样本想一下。

提供行为一致的接口。避免无端与内置类型不兼容。例如STL的各种接口就很一致。

任何接口如果要求客户记着做什么事，就有着不正确使用的倾向。

**总结：**

1. 好的接口容易被正确使用，不容易被误用。
2. 促进正确使用的办法包括接口的一致性，以及内置类型的行为兼容。
3. 阻止误用的办法有建立新类型，限制类型上的操作，束缚对象的值，消除客户对资源管理的责任。
4. Shared\_ptr支持定制删除器，很好的设计。可以防范跨DLL问题。

**19、设计class犹如设计type（Treat class design as type design）**

每设计一个class你都要尝试回答以下问题：

新type的对象应该如何创建和销毁？--影响构造和析构函数、new和delete。

对象的初始化和对象的赋值应该有什么区别？--影响构造函数和赋值运算符。

新type的对象如果pass-by-value意味着什么?--Copy构造函数用来定义一个type的传值

什么是新type的合法值？对于class的成员变量通常只有部分数值是有效的。

你的新type需要配合某个继承图系么？会受基类影响，virtual函数影响

新type需要什么样的转换？影响单参构造函数和转换运算符。

什么样的操作符和函数对新type是合理的？决定你该声明哪些函数

什么样的标准函数应该驳回？将其声明为private

谁用新type？决定访问控制权限。

未声明接口？

新type有多一般化？是否需要定义为template

真的需要这个type吗？

**20、以pass-by-const-reference替换pass-by-value（Prefer pass-by-const-reference to pass-by-value）**

参数切割问题：如果一个派生类对象传值给一个基类对象，那么它派生类的部分会被丢掉。

**总结：**

1. 传常引用通常比传值高效，还可以避免切割问题
2. 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。对他们而言传值更妥当。

**21、必须返回对象时，别妄想返回其reference（Don’t try to return a reference when you must return an object）**

任何时候看到一个reference声明式，你都应该立刻问自己，它的另一个名称是什么。它一定是另一个物体的名称。

函数创建对象的途径有二：在stack或者heap空间创建。（应该还有全局数据段吧。）

不该返回引用时，就不要返回，无论是在函数内部如何创建返回对象：stakc、heap、静态变量都不行。

在返回reference和object之间抉择时，挑选行为最正确的那个。

**总结：**

绝对不要返回指针或引用指向一个local-stack对象、heap-allocated对象或者local-static对象、

**22、将成员变量声明为private（Declare data members private）**

将成员变量隐藏在函数接口背后，可以为“所有可能的实现”提供弹性。

对客户隐藏成员变量，保留了日后变更实现的权力。

Public意味着不封装，不封装意味着不可改变。

Protected成员变量的封装性并不高过public成员变量。

Protected和public一样缺乏封装性，如果这种访问权限的一个成员变量被取消，那会有大量的代码被破坏。

其实只有两种访问权限：private（封装）和其他（不提供封装）。

**总结：**

1. 切记将成员变量声明为private，这会带来很多好处。
2. Protected并不比public好到哪去。

**23、宁以non-member、non-friend替换member函数（Prefer non-member、non-friend functions to member functions）**

面向对象守则要求数据应尽可能的封装。

越多函数可以修改成员变量，封装性就越低。

能够访问private成员变量的只有member函数和friend函数而已。

Namespace可以跨越多个源码文件，而class不可以。

C++标准库有数十个头文件，每个头文件提供std的某些机能。

**总结：**

如果可以的话，请用non-member和non-friend函数替换member和friend函数。

**24、若所有参数都需要类型转换，请为此采用non-member函数（Declare non-member functions when type conversions should apply to all parameters）。**

让Class支持隐式类型转换是非常不好的主意。

如果可以friend函数就要避免。

例如operator\*（const Rational& a , const Rational& b）;重载乘法运算符放在函数外面（non-member）和放在函数里面（member）要好的多。因为如果Rational（有理数）支持从整数隐示类型转换的话，2\*Rational（3）和Rational（3）\*2应该是都可以的。如果放在函数里面，前者就不行了。因为函数内部重载运算符时，只能传第二个参数。

**总结：**

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函数。

**25、考虑写出一个不抛异常的swap函数（Consider support for a non-throwing swap function）**

全特化模板编程。

正常情况下，我们是不可能修改C++的标准库的，但是我们被允许为标准模板库的内容提供特化版本。像下面这样：

template<>

void swap<CLASS\_NAME>(CLASS\_NAME& a, CLASS\_NAME& b);

所以STL容器都提供有public的swap成员函数和std的特化swap版本来调用容器的swap成员函数。

C++只允许对类模板偏特化，不能对函数模板偏特化。

我们可以全特化std内的模板，但是不能往std里加新的东西。

Std的内容是由C++标准委员会定好的，他们禁止修改已经声明好的东西。你修改了可能编译会通过，但是以后的运行效果未必如你预期。

C++的名称查找法则确保将找到的global作用域或T所在之命名空间内的任何T专属的swap。

先找global作用域或者T所在的作用域，然后才会去std中找。

Swap的缺省版本是以copying函数为基础的，这两者一般情况下都允许抛出异常。

**总结：**

1. 当std:swap对你的类型效率不高时，提供一个public的swap成员函数，并确定这个函数不抛出异常（关于抛出异常的事情后面再说，现在先记住就行了）。
2. 如果你提供一个成员函数swap，那么也请提供一个非成员函数的swap来调用前者。对于类，也请特化std:swap()
3. 调用swap时应针对std:swap()使用using声明式，使得以后可以赤裸裸的调用swap，将名字查找的工作交给编译器。
4. 为用户定义类型进行std全特化是很好的，但千万不要尝试在std中加入某些对std而言是全新的东西。简而言之不要往std中加东西。

**26、尽可能延后变量定义式的出现时间（Postpone variable definitions as long as possible）**

跳过毫无意义的default构造函数。例如先default构造再复制就不如直接拷贝构造。

要尽量避免构造、析构、赋值等带来的不必要消耗。

**总结：**

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、尽量少做转型动作（Minimize casting）**

尽量保证你的程序很干净的通过编译。

在C++中转型是一个你会想带着极大尊重去亲近的一个特性。

C风格的转型动作（**旧式风格**）：

1. Expression

T（Expression）

两种形式没什么差别只是小括号位置不同而已。

C++风格的四种**新式风格**转型:

const\_cast<T>(expression)：通常用来将对象的**常量性转除**。也是唯一有此能力的C++风格转型操作符。

dynamic\_cast<T>(expression)：主要用于**安全向下转型**，例如基类向派生类转换，如果转换失败会返回null。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）取决于编译器，**意味着它不可移植**。例如将pointer to int转换为int。

staic\_cast<T>(expression)：强迫隐式转换，例如non-const转换为const，int转换为double。这个其实就和C风格的转型差不多。

使用新式转型比较好：1）明显可以看出是转型，2）为编译器帮我们检查错误提供帮助。

我们要避免假设对象在C++中是如何布局的这种假设。

Dynamic\_cast的许多实现版本相当慢。每一次dynamic cast可能会导致多达四次的strcmp调用，用以比较class名字。深度继承或者多重继承的成本更高。

绝对必须避免的一件事是所谓的连串dynamic casts。这种代码又大又慢。

优良的C++代码很少使用转型。

**总结：**

1. 尽量避免转型，特别是在注重效率的代码中避免使用dynamic cast。如果设计导致需要转型动作，请试着修改设计。
2. 如果非得转型，请试着把他们放到函数里，在客户代码中尽量不要出现转型代码。
3. 尽量使用C++转型，不适用旧转型。

**28、避免返回handles指向对象内部成分（Avoid returning “handles” to object internals）**

成员变量的封装性最多只等于“返回其reference”的函数的访问级别。

如果const成员函数传出一个reference，后者所指数据与对象自身有关联，而它又被存储于对象之外，那么函数的调用者可以修改那个数据。

虽然调用const成员函数，却造成对象状态被更改（返回reference）。

绝对不该令成员函数返回一个指针指向访问级别较低的成员函数。

一旦返回一个handle代表对象内部成分，就有风险：handle比其所指对象更长寿。

**总结：**

尽量避免返回handle（reference、pointer、迭代器）指向对象内部。遵守这个条款可以增加封装性，帮助const成员函数的行为更像个const。

**29、为异常安全而努力是值得的（Strive for exception-safe code）**

当安全抛出时，带有异常安全性的函数会：

1. 不泄露任何资源
2. 不允许数据败坏

异常安全函数：

1. 基本承诺：如果异常被抛出，程序内的任何事物仍然保持在有效状态下。
2. 强烈保证：如果异常被抛出，程序状态不改变。函数成功就是成功；如果失败程序会回复到调用函数之前的状态。
3. 不抛掷保证：承诺决不抛出异常，因为它们总是能够完成他们原先承诺的功能。例如作用于内置类型的操作都提供nothrow保证。

异常安全代码必须保证以上三点之一。否则就不具备异常安全性。

任何使用动态内存的东西（例如STL的所有容器）如果无法找到足够内存，就会抛出bad\_alloc异常。

可能的话，提供nothrow保证，但大部分函数往往是提供基本承诺和强烈保证。

强烈保证并非在任何时刻都显得实际。

一个软件系统要不就具备异常安全性，要不就全然否定，没有所谓的局部异常安全系统。

**总结：**

1. 异常安全函数即使发生异常也不会泄露资源或者允许任何数据结构的败坏。这样的函数区分为三种可能：基本型、强烈型和nothrow型
2. 强烈保证型一般能以copy-and-swap形式实现，但强烈保证型并非对所有函数都具备现实意义。
3. 函数提供的异常安全保证，通常只能等于所调用的各个函数的异常安全中的最弱者。

**30、透彻理解inlining的里里外外（Understand the ins and outs of the inling）**

除了免除函数调用成本外，编译器可能还会对inline函数执行语境相关最优化。

Inline函数的缺点是会增大程序体积。

如果inline函数的本体很小的话，编译器针对函数本体所生成的代码可能比函数调用还小，这样就完美了。

Inline只是对编译器的一个申请或者说建议，不是强制命令。这个申请可以通过两种方式提出：

1. 隐喻方式：将函数定义放在类内。
2. 明确方式：在函数定义式前加上inline关键字。

Inline函数通常一定被置于头文件内，因为C++（大多数语言）在编译过程中进行inlining。

如果template具现出来的函数都应该inlined，就将template声明为inline。否则不应该声明为inline。

一个表面上看似inline的函数是否真实inline，主要取决于编译器。

编译器提供了一个诊断级别：如果无法将你要求的函数inline化，编译器会给你一个警告信息。

编译器即使想对一个函数进行inlining，但是当它发现有通过函数指针调用该函数的话，还是会生成outlined的函数体。

即使空白的构造函数至少会调用成员变量和基类的构造函数。所以是否将派生构造函数inline化，并非是个轻松的决定。

如果inline函数是在程序库用，那么函数被修改后，则用到该函数的程序要重新编译。而非inline函数只要重新链接就好。

大部分调试器多inline函数都束手无策。因为你无法在一个不存在的函数内设立断点。或者是在debug版程序中禁止发生inline。

一开始先不要将任何函数声明为inline，最多将那种一定能成为inline的inlining。

**总结：**

1. 将inlining限制在小型被频繁调用的函数上。这样使得程序调试和升级更容易，也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。
2. 不要只因为函数模板出现在头文件，就把他们声明为inline。

**31、将文件间的编译依存关系降至最低（Minimize compilation dependencies between files）**

std:string只是一个typedef，原型是basic\_string<char>。如果前置声明只是class string；是错的。并且也不应该前置声明标准程序库的东西。标准头文件不太可能成为编译瓶颈。

编译器必须在编译期间知道对象的大小。

前置声明后，在类内只能使用指针或者引用该类的对象，如果在类内直接定义该类的对象会提示incomplete type。

让头文件尽可能自我满足，实在做不到在包含其他头文件。

一个设计策略：

1. 如果使用引用或者指针就可以完成任务，那就不要使用对象。
2. 如果可能，尽量以class声明式替换class定义式。只前置声明某类，也可以在声明成员函数时的参数或者返回值上使用该类。
3. 为声明式和定义式提供不同的头文件。例如为一个类Person提供两个头文件Person.h和PersonImpl.h

Handle classes和Interface classes两派解除了接口和实现之间的耦合。Handle classes是在类内拥有一个实现的指针。Interface是基类给定纯虚函数接口，由工厂函数创建真实对象，返回接口类指针。

**总结：**

1. 支持编译依存最小化的构想是：依赖声明式而不依赖定义式。两个方法：Handle classes和Interface classes。
2. 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。
3. **继承与面向对象设计**

**32、确定你的public继承塑模出is-a关系（Make sure public inheritance models “is-a”）**

C++面向对象编程中，最重要的一条规则就是：**公开继承意味着 is-a的关系**。

私有继承的意义完全不同。

受保护的继承少用，作者说他都不是很清楚有什么用。

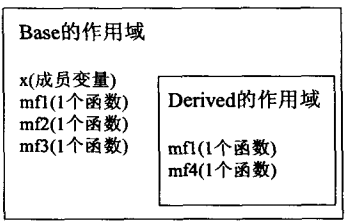
Public继承主张：能够实施于基类对象的每件事情，都可以实施于派生类上。

**总结：**

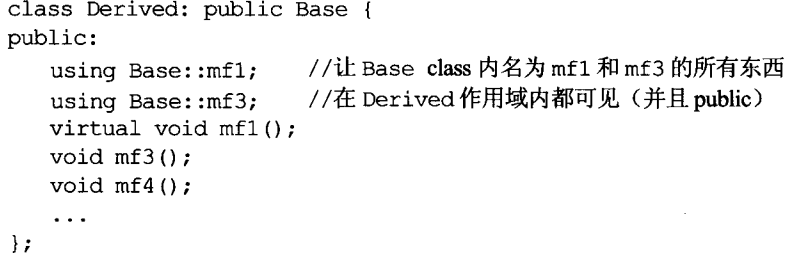
公开继承意味着is-a。适用于基类上的每一件事情一定也适用于派生类上，因为派生类就是一个基类对象。

**33、避免遮掩继承而来的名称（Avoid hiding inherited names）**

派生类的作用域其实是嵌套在基类的作用域中的。



在mf4中调用mf2()的名称查找规则，首先查找mf4函数中的**local作用域**，再找**Derived类的作用域**，再找**Base类的作用域**，再找Base类所在**namespace的作用域**，再找**global作用域**。



Using只是把名字的使用权拿了过来。如果不用using，那么派生类中和基类重名的函数会被遮掩。

**总结：**

1. 派生类的名称会遮掩基类的名称。在公开继承下从来没有人希望这样。
2. 为了让被遮掩的名称再次可用，可以使用using或者转交函数。（在函数实现里显式调用基类的该函数）。

**34、区分接口继承和实现继承（Differentiate between inheritance of interface and inheritance of implementation）**

公开继承又分为两种：函数接口继承和函数实现继承。

我们竟然可以为纯虚函数提供定义！但是调用时还是要通过类名调用。pBase->Base::func();

纯虚函数的目的是让派生类**只继承函数接口**。

普通虚函数的目的是让派生类**继承该函数的接口**和**缺省实现**。

非虚函数的目的是让派生类**继承函数接口**以及**一份强制性实现**。

接口和缺省实现应该分开，例如：接口定义为纯虚函数，缺省实现单独定义一个受保护的非虚函数。

如果成员函数是个非虚函数，意味着它并不打算在派生类中有不同的行为。

**总结：**

1. 接口继承和实现继承不同。在公开继承下，派生类总是继承基类的接口。
2. 纯虚函数只具体指定接口继承
3. 普通虚函数具体指定接口继承及缺省实现继承
4. 非虚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（Consider alternatives to virtual functions）**

通过公开非虚成员函数调用私有虚函数，成为non-virtual interface（NVI）手法，就是**Template Method设计模式**的一个独特表现形式。可以把它叫做virtual函数的外覆器（这就有点python装饰器的味道了）。

派生类可以重新定义继承的私有虚函数，合情合理。

由**函数指针**实现**Strategy设计模式**。

唯一能够解决需要以非成员函数访问class的非公开成员的办法就是弱化封装：例如友元函数、提供public属性接口。

由**std::function**实现**Strategy设计模式**。

使用function模板比单纯的函数指针的一个好处是，函数模板允许隐式转换，而函数指针不行。例如short转为int。

解决了一个长久以来的疑问：

关于std::bind成员函数时，经常见到需要传this指针的情况。答案是这样的，如果你要通过function模板传一个可调用对象，如果可调用对象是类的成员函数，那么在外部调用的时候不仅需要知道该函数声明时的各个参数，而且该函数还有一个隐式参数没有写出来，那就是放在真正第一位的this指针。但是如果你在外部调用时，并不知道调用的是哪个对象，所以，就可以通过std::bind(&Class:memFunc, obj, \_1)来绑定一个该类对象。

**古典Strategy设计模式**

将计算方法单独写成一个继承体系，然后作为指针放在使用它的函数里。

**总结：**

1. virtual函数的替代方案包括NVI手法以及Strategy设计模式的多种形式。NVI手法是Template Mehtod设计模式的一种形式。
2. 从成员函数转移到class外部函数的一个缺点是非成员函数无法访问class的非公开成员。
3. function模板对象的行为就像一般的函数指针。这种对象可以接受所有的与它给定的目标签名式/可调用格式 所兼容的可调用物。

**36、绝不重新定义继承而来的非虚函数（Never redefine an inherited non-virtual function）**

虚函数是动态绑定，非虚函数是静态绑定。

如果派生类重新定义了基类的非虚函数，那指向基类和派生类的指针或者引用的调用该函数的行为就不一样了。而且也违背了基类不变形凌驾派生类特异性的设计性质。

**总结：**

绝对不要重新定义继承来的非虚函数。

**37、绝不重新定义继承而来的缺省参数值（Never redefine a function’s inherited default parameter value）**

虚函数是动态绑定，但是其缺省参数值却是静态绑定。意思是只要你用基类指针调用虚函数，无论该真实对象是什么，该函数的参数都是基类该函数的缺省参数。

不论基类指针真正指向什么，它们的静态类型都是基类指针。

如果缺省参数要在运行期动态绑定，那么会很复杂很慢，所以C++才保持了这样的方式。简而言之就是为了运行期效率。

一般而言，virtual函数不要提供缺省参数，如果非要不可，那就用虚函数的替代设计，像35中说的那些。

**总结：**

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，而虚函数是你唯一应该重写的东西，却是动态绑定。

**38、通过复合塑模出has-a或者根据某物实现出（Model “has-a” or “is-implemented-in-terms-of through composition”）**

Set的实现往往导致每个元素用三个指针的额外开销。Set通常用AVL实现。如果**速度比空间重要**，用它很合理。如果相反，那么自己去实现set。

如果D是一种B，那么对B为真的每一件事情，对D也都应该为真。

自己实现set的话，继承list是不合理的，因为list中可以有重复元素，如果set继承list那么set中也可以有重复元素。

所以应该用has-a关系，set中包含一个list来实现。

**总结：**

1. 复合的意义和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什么。

**39、明智而审慎地使用private继承（Use private inheritance judiciously）**

Private继承并不意味着is-a关系，而是意味着implemented-in-terms-of（根据某物实现出）。

1. Private继承，编译器不会自动将一个派生类对象转为基类对象。
2. 由private继承而来的所有成员，在派生类中都会变成private的，无论它在基类中是什么样的。

让类D私有继承类B，只是因为类B内已经具备了某些特性，而不是因为B对象和D对象有什么实际意义上的关系。 意思是D对象根据B对象实现而得，再没有其他含义了。

Private继承纯粹只是用于**实现**的一种技术，在设计层面上没有意义。

尽可能使用**复合**，必要时才使用**private继承**（例如涉及到virtual函数）。

Private继承通常可以用**public继承+复合**（内部类）来替换。

独立（非附属）对象的大小一定不为零，C++规定如果类时空的，则默认放一个char进去，这就是sizeof（空类）==1的原因。

**空白基类最优化 EBO**(empty base optimization)。例如继承一个空白类不会增加派生类的大小，而复合的话空类对象是会增加被加入到那个类的大小的。

当并不存在is-a关系的两个类，其中一个需要访问另一个的protected成员，或需要重新定义其一或多个virtual函数，private继承通常可以派上用场。

**明智而审慎**指的是考虑过所有方案后，private仍然是最佳方案的话，那就用它。

**总结：**

1. private继承意味着is-implemented-in-terms-of（根据某物实现出）。通常比复合级别更低。但是当派生类需要访问基类protected成员或者需要重新定义继承而来的virtual函数时，private继承可以派上用场。
2. 和复合不同，private继承可以造成EBO（空白基类最优化）。
3. **明智而审慎地使用多重继承（Use multiple inheritance judiciously）**

一个派生类继承两个基类，如果两个基类中同时存在两个同名成员，那么当派生类对象调用该成员时就会有歧义。即使两个成员具有不同的访问权限。

可以通过指定是哪个基类的方法来解决。

通过虚继承可以解决菱形继承/钻石继承的问题。

但是虚继承为带来一定的代价。例如虚继承的类的对象往往比非虚继承的大，访问速度也慢。

虚继承 基类的初始化责任是由继承体系中最底层的类负责。

1. 非必要时不适用虚继承。
2. 如果必须要虚基类，那么久尽可能避免在其中放数据。

多重继承只是面向对象工具箱里的一个工具而已。

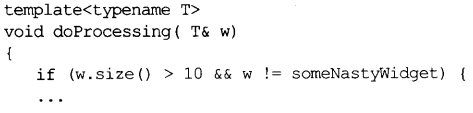
**总结：**

1. 多重继承比单一继承复杂。他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）复杂度等等成本。如果virtual base classes 不带任何数据，那virtual继承僵尸最具实用价值的。
3. 多重继承的确有正当用途。例如设计public继承某个Interface类+private继承某个协助实现的类时。
4. **模板与泛型编程**

**41、了解隐式接口和编译器多态（Understand implicit interfaces and compile-time polymorphism）**

面向对象编程总是以显式接口和运行期多态解决问题。

模板与泛型编程与面向对象有根本上的不同。在这里隐式接口和编译期多态比显式接口和运行期多态更重要。



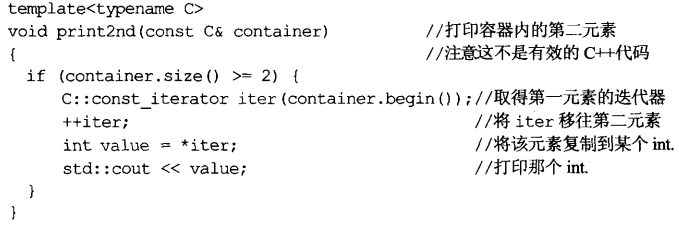
隐式接口仅仅是由一组有效表达式构成，表达式自身看起来很复杂，但他们要求的约束条件一般而言相当直接又明确。

**总结：**

1. 类和模板都支持接口和多态
2. 对类而言，接口是显式的，以函数签名为中心。多态则是通过virtual函数发生于运行期
3. 对模板而言，接口是隐式的，奠基于有效表达式。多态则是通过模板具现化和函数重新解析发生于编译期。

**42、了解typename的双重意义（Understand the two meanings of typename）**

声明模板时，关键字class和typename没有不同，只是习惯问题。



从属名称：依赖于模板参数的名称。例如上面的C。

嵌套从属名称：例如上面的C::const\_iterator。

非从属名称：不依赖于任何模板参数的名称。例如上面的int。

上面的代码中，iter只有在C::const\_iterator是一种类型时，才是有效的代码，但是如果没有明确指出，C::const\_iterator被默认为不是类型。**明确指出**的关键字是就typename。像这样：



只有在嵌套从属类型名称上才能使用typename指明。

在继承的基类列表和类初始化的成员初始化列表上不能使用typename。

std::iterator\_traits<IterT>::value\_type

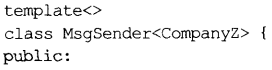
可以萃取出IterT指针指向的元素类型。



**总结：**

1. 声明模板时，class和typename可以互换。
2. 用typename来标识嵌套从属类型名称，但是不能在继承的基类列表和类构造函数的初始化列表中使用它。

**43、学习处理模板化基类内的名称（Know how to access names in templatized base classes）**



这种template语法叫做全特化。

**当我们从面相对象C++跨进模板C++时，继承就不像以前那样畅行无阻了。**

C++拒绝在模板化基类中寻找继承而来的名称。

**总结：**

解决上面这个问题的方法是，对编译器承诺：模板化基类的任何特化版本都将支持一般（泛化）版本所提供的接口。三种方法:

1. this调用
2. using声明
3. 明确指定基类调用

**47、请使用traits classes表现类型信息（Use traits classes for information about types）**

STL共有5种迭代器，对应于它们支持的操作：

1. Input迭代器，只能向前移动，一次一步。只能读取一次它所指的东西。模拟输入文件的读指针。例如istream\_iterators。
2. Output迭代器，只能向前移动，一次一步。只能写一次它所指的东西。模拟文件的写指针。例如ostream\_iterators。
3. Forward迭代器，可以做上述两个迭代器所能做的每一件事，可以读写所指物一次以上。
4. Bidirectional迭代器，可以前后移动，例如STL的list、set、map、multiset、multimap的迭代器。
5. Random access迭代器，可以再常量时间内向前或向后跳跃任意距离。例如vector、deque、string的迭代器。

Traits并不是C++的关键字或者某种预先定义好的构件；它是一种技术，是一个C++程序员共同遵守的协议。

Traits总是被实现为struct，但是总是被成为traits classes。

**总结：**

1. traits classes使得类型相关信息在编译器可用。他们以template和template特化完成实现。
2. 整合重载技术后，使得traits classes可能在编译器执行if…else…测试。

**48、认识template元编程（Be aware of template metaprogramming）**

TMP（template meta programming）模板元编程。

TMP已被证明是一个图灵完全机器，意思是它威力大到足以计算任何事物。

**总结：**

1. 模板元编程可以将工作由编译器移往编译期，因为得以实现早期错误的侦测和更高的执行效率
2. Tmp可被用于“基于政策选择组合”的客户定制代码，也可以用来避免生成对某些特殊类型并不合适的代码。
3. **定制new和delete**

多线程下内存分配很可能有问题。

STL使用的heap内存是由容器所拥有的内存分配器对象管理，不是被new和delete直接管理的。

**49、了解new-handler的行为（Understand the behavior of the new-handler）**

现代C++当new操作符无法满足某一内存分配需求时，它会抛出异常。旧版本的会返回一个null指针。

在<new>头文件里包含了new操作符和delete操作符，已经set\_new\_handler函数，用于设置当new操作符出错时执行的回调。

一个设计良好的new-handler应该满足：

1. 让更多内存可被使用。因为new-handler可能被多次调用，当出错回调给出更多内存后，之后可能使得下一次new操作符成功。
2. 设置另一个new-handler。
3. 删除new-handler。一旦没有设置new-handler，在分配内存出现异常时，就会抛出异常
4. 抛出bad\_alloc异常，或者继承自bad\_alloc的异常。
5. 不反悔。调用abort或者exit退出程序。

类的static成员必须在类外被定义，除非是const整型。

C++不支持类专属new-handler的做法，但是我们可以自己实现出这种功能来：

1. 类提供自己的set\_new\_handler和new操作符。
2. Set\_new\_handler使得客户可以设置自己的new-handler，new操作符则在分配类需要的内存过程种，当出现错误时，调用类专属的new-handler替换掉global的new-handler。

mixin风格的**基类**，使得派生类继承单一特定能力。（这里是使得派生类具有设定类new-handler的能力）。

将基类声明为一个模板，但是类内容和模板参数无关，可以使得派生类继承的模板类中的静态变量互不相同，也叫做**实体互异**。

上面的这种技术叫做学名叫做：**CRTP**(curiosity recurring template pattern)怪异循环模板模式。

1993年之前new操作符还是要求返回null，后来的新标准要求抛出异常。为了兼容那种需要检测返回null的程序，new操作符提供了nothrow版本语法：Widget\* p = new(std::nothrow) Widget;

**总结：**

1. set\_new\_handler允许客户指定一个函数，在内存分配失败时被调用。
2. nothrow版的new只适合内存分配，因为new分配对象内存成功后，会调用该对象的构造函数，这时就无法保证在该构造函数里会不会调用nothrow的new了。

**50、了解new和delete的合理替换时机（Understand when it make senses to replace new and delete）**

替换编译器提供的new和delete的原因：

1. 用来检测运用上的错误。例如new和delete的次数不对应。或者我们想多分配一点内存，以实现在指定分配内存的头部或者尾部增加一个字段的功能。
2. 为了强化性能。编译器提供的new和delete一般为了满足所有人的使用，性能并不高。如果你专业一点，对程序的动态内存运用形态有深刻的了解，那么你实现出来的new和delete能比编译器提供的快很多。
3. 为了收集使用的统计数据。这里的统计指的就是数学统计的意义，例如分配内存块的大小分布情况，最大值等。

可以在申请内存的时候多申请两个int，在原申请内存的首尾分别放两个签名位，保证内存没有被篡改。

齐位：**部分**计算机体系结构（主要是CPU吧）要求某种类型的变量地址必须是其大小的倍数，例如int的地址必须是4的倍数，double的地址必须是8的倍数等。齐位的一个明显的好处就是可以让运行速度加快。

C++要求所有new操作符返回的指针，必须有适当的对齐，和数据类型有关。

Malloc返回的指针式对齐的，所以可以在new中调用malloc。

自己去写一套内存管理器大多是没有必要的，因为已经有很多开源代码实现的很好了。

**补充**上面替换编译器提供的new和delete的原因：

1. 增加分配和回收的速度。一般的泛用型分配器往往没有专门写的某个类型的内存分配其速度快。影响因素例如对象的内存分布、单/多线程
2. 降低缺省内存分配器带来的额外开销。泛用型分配器在分配内存的时候往往会额外分配一点内存做一些其他操作，自己定制可以消除了这些东西。
3. 弥补非最佳齐位。缺省的内存分配器，可能不会保证分配的内存有最佳的齐位。
4. 将对象成簇集中。将对象放到内存地址很相近的位置。这样在同种对象频繁访问时减少内存换页。
5. 获得非传统行为。例如共享内存只有C API，你可以用C++封装一下。

**总结：**

有许多理由需要自己定制new和delete，包括改善性能、对heap运用错误进行调试、收集heap的使用信息。

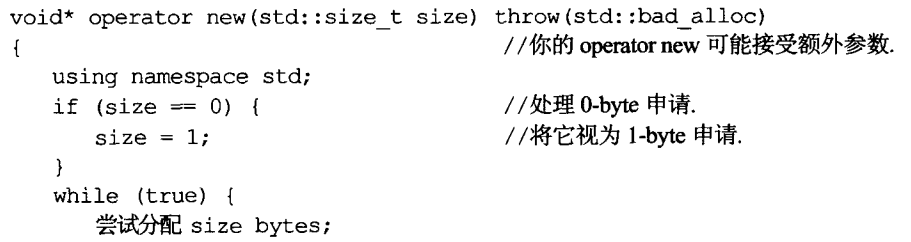
**51、编写new 和delete的时候需要固守常规（Adhere to convention when writing new and delete）**

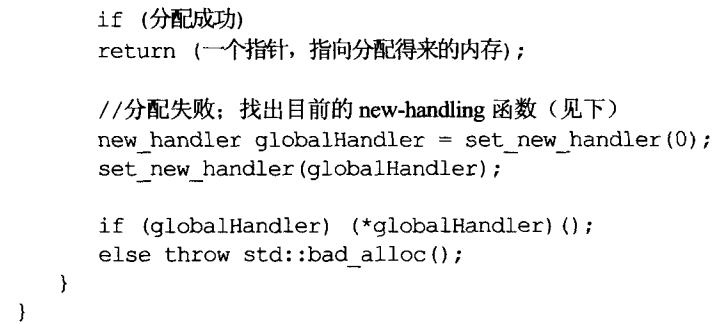
New操作符必须得有正确的返回值、内存不足时调用new-handler、必须有对付零需求的准备、避免不小心掩盖正常的new。

New操作符不会不不止一次的尝试分配内存，当内存不够时会调用new-handler，这是因为假设new-handler会释放更多的内存出来。如果new-handler是null会直接抛出bad\_alloc。

C++要求，即使要求分配大小为0的内存也要返回一个合法的指针。

目前来看一个new的实现可能是这样的：

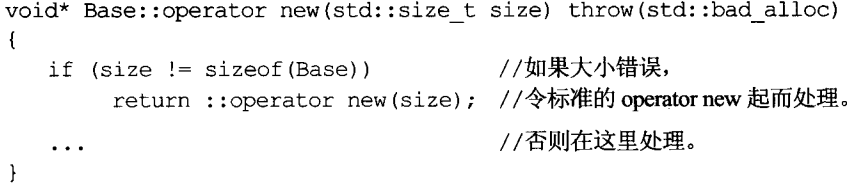




上面获取new-handler的方法是因为，没有办法直接获得new-handler。

定制内存管理器的一个常见理由是：为特定的类的对象分配行为提供最优化，却不是为了该类的任何派生类。

在基类的new操作符中最好加上这样的判断：



以防止派生类调用基类的new。

对于new[]操作符，需要的做的事情只是分配一块未加工的内存。

传递给new[]的size\_t参数，可能比实际对象占用的空间要大一点，例如有可能需要额外空间来存放数组大小。

Delete[]需要做的事情是保证delete null是安全的。

如果基类缺少虚析构函数，那么delete的参数的值可能是错误的。

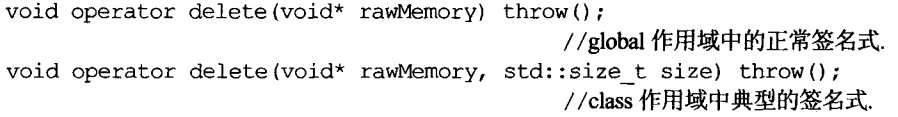
**总结：**

1. new中应该有一个无穷循环，不断尝试分配内存，无法满足要求时要调用new-handler.需要处理0 bytes申请.类版本还需要处理和类大小不同的错误申请(调用原生new).
2. Delete在删除null的时候不需要做任何事,类版本还应该处理删除大小和类大小不同时的错误申请.

**52、写了placement new也要写placement delete（Write placement delete if you write placement new）**

正常的new和delete的函数签名：





如果new接受的参数除了必要的size\_t之外还有其他的参数，这就是个placement new。

在众多版本的placement new中，特别有用的一个是**接受一个指针指向对象该被构造之处**。这种new的用途例如：在vector的未使用空间上创建对象。这个版本的new也是placement的来源，**一个特定位置上的new**。当说到placement new时，大都说的是这个版本的。

A a = new A；

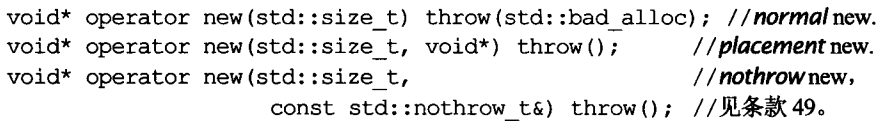
上面这条语句包含两个部分，new分配内存，然后调用构造函数。当new成功，但构造函数抛出异常时，需要回收new分配的内存，这个任务在C++运行时系统身上。

C++运行时系统，在出现上述的异常时，会去找和new有相同参数的delete，如果找到就调用它来恢复现场，否则就什么也不做。并且应该还有一个版本的delete在正常删除时被调用。

Placement delete只会在new对象后调用构造函数异常时调用，在正常delete一个指针时绝对不会调用placement delete。

New操作符同样会有名称遮掩问题。

缺省情况下，C++提供如下版本的new：



**总结：**

1. 当写了一个placement new时，还要记得写一个placement delete。否则可能会发生内存泄露。
2. 小心你写的new和delete遮掩了其他的new和delete（基类或者全局作用域的）。

**53、不要忽视编译器的警告（Pay attention to compiler warnings）**

**总结：**

1. 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警告级别下争取无任何警告的荣誉。
2. 不要过度依赖编译器警报能力，因为不同的编译器对待事情的态度不同。一旦移植到另一个编译器上，你原本依赖的警告信息有可能消失。

**54、让自己熟悉包括TR1在内的标准程序库（Familiarize yourself with the stand library，include TR1）**

TR1:Technical Report 1.

TR1提供的机能几乎对每一种程序库和每一种应用程序都带来利益。

**总结：**

1. 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。
2. TR1添加了智能指针、一般化函数指针、hash-based容器、正则表达式以及另外10个组件的支持。（这些在C++11里都有了）
3. TR1只是一个规范。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。

**55、让自己熟悉Boost（Familiarize yourself with Boost）**

Boost组织和C++标准委员会有着独一无二的密切关系，并且对委员会具有很深的影响力，boost由委员会成员创立。

Boost：一个可被加入标准C++之各种功能的测试场。

**总结：**

1. Boost是一个社区，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。
2. Boost提供许多TR1组件实现品，以及其他许多程序库。