## 第一部分 语法篇

### 第1章   从C继承而来的

**建议0：不用让main函数返回void**

         main函数的返回类型是int，不是void或其它类型。

**建议1：区分0的4种面孔**

(1)、整型0；(2)、空指针NULL，指针与int类型所占空间是一样的，都是32位；(3)、字符串结束标志’\0’；(4)、逻辑FALSE/false，FALSE/TRUE是int类型，而false/true是bool类型。

**建议2：避免那些由运算符引发的混乱**

         不要混淆=和==、&和&&、|与||这三对运算符之间的差别，针对=和==之间的问题，可以这样做：if (0 == nValue)。

**建议3：对表达式计算顺序不用想当然**

(1)、针对操作符优先级，建议多写几个括号；(2)、注意函数参数和操作数的评估求值顺序问题。

**建议4：小心宏#define使用中的陷阱**

(1)、用宏定义表达式时，要使用完备的括号：如 #define ADD(a, b) ((a)+(b))；(2)、使用宏时，不允许参数发生变化；(3)、用大括号将宏所定义的多条表达式括起来。

**建议5：不要忘记指针变量的初始化**

(1)、可以将其初始化为空指针0(NULL)；(2)、对于全局变量来说，在声明的同时，编译器会悄悄完成对变量的初始化。

**建议6：明晰逗号分隔表达式的奇怪之处**

(1)、在使用逗号分隔表达式时，C++会确保每个表达式都被执行，而整个表达式的值则是最右边表达式的结果；(2)、在C++中，逗号分隔表达式既可以用作左值，也可以用作右值。

**建议7：时刻提防内存溢出**

在调用C语言字符串金典函数(strcpy、strcat、gets等)时，要从源代码开始就提高警惕，尽量追踪传入数据的流向，向代码中的每一个假设提出质疑。在访问数据时，注意对于边界数据要特殊情况特殊处理，还要对杜绝使用未初始化指针和失败后未置NULL的“野指针”。

**建议8：拒绝晦涩难懂的函数指针**

         函数指针在运行时的动态调用(例如函数回调)中应用广泛。但是直接定义复杂的函数指针会由于有太多的括号而使代码的可读性下降。使用typedef可以让函数指针更直观和易维护。

**建议9：防止重复包含头文件**

         为了避免重复包含头文件，建议在声明每个头文件时采用“头文件卫士”加以保护，比如采用如下的形式：

**[cpp]** [view plaincopy](http://blog.csdn.net/fengbingchun/article/details/9193577)

1. #ifndef\_\_SOMEFILE\_H\_\_
2. #define\_\_SOMEFILE\_H\_\_
3. ……//声明、定义语句
4. #endif

**建议10：优化结构体中元素的布局**

把结构体中的变量按照类型大小从小到大依次声明，尽量减少中间的填充字节。

**建议11：将强制转型减到最少**

         (1)、const\_cast<T\*>(a)：它用于从一个类中去除以下这些属性：const、volatile和\_\_unaligned；(2)、dynamic\_cast<T\*>(a)：它将a值转换成类型为T的对象指针，主要用来实现类层次结构的提升；(3)、reinterpret\_cast<T\*>(a)：它能够用于诸如One\_class\*到Unrelated\_class\*这样的不相关类型之间的转换，因此它是不安全的；(4)、static\_cast<T\*>(a)：它将a的值转换为模板中指定的类型T，但是，在运行时转换过程中，它不会进行类型检查，不能确保转换的安全性。

**建议12：优先使用前缀操作符**

对于整型和长整型的操作，前缀操作和后缀操作的性能区别通常是可以忽略的。对于用户自定义类型，优先使用前缀操作符。因为与后缀操作符相比，前缀操作符因为无须构造临时对象而更具性能优势。

**建议13：掌握变量定义的位置与时机**

在定义变量时，要三思而后行，掌握变量定义的时机与位置，在合适的时机于合适的位置上定义变量。尽可能推迟变量的定义，直到不得不需要该变量为止；同时，为了减少变量名污染，提高程序的可读性，尽量缩小变量的作用域。

**建议14：小心typedef使用中的陷阱**

         区分typedef与#define之间的不同；不要用理解宏的思维方式对待typedef，typedef声明的新名称具有一定的封装性，更易定义变量。同时还要注意它是一个无“现实意义”的存储类关键字。

**建议15：尽量不要使用可变参数**

编译器对可变参数函数的原型检查不够严格，所以容易引起问题，难于查错，不利于写出高质量的代码。所以应当尽量避免使用C语言方式的可变参数设计，而用C++中更为安全的方式来完美代替之(如多态等)。

**建议16：慎用goto**

过度使用goto会使代码流程错综复杂，难以理清头绪。所以，如果不熟悉goto，不要使用它；如果已经习惯使用它，试着不去使用。

**建议17：提防隐式转换带来的麻烦**

提防隐式转换所带来的微妙问题，尽量控制隐式转换的发生；通常采用的方式包括：(1)、使用非C/C++关键字的具名函数，用operator as\_T()替换operator T()(T为C++数据类型)。(2)、为单参数的构造函数加上explicit关键字。

**建议18：正确区分void与void\***

Void是“无类型”，所以它不是一种数据类型；void\*则为“无类型指针”，即它是指向无类型数据的指针，也就是说它可以指向任何类型的数据。Void发挥的真正作用是限制程序的参数与函数返回值：(1)、如果函数没有返回值，那么应将其声明为void类型；(2)、如果函数无参数，那么声明函数参数为void。对于void\*，(1)、任何类型的指针都可以直接赋值给它，无须强制转型；(2)、如果函数的参数可以是任意类型指针，那么应声明其参数为void\*。

### 第2章   从C到C++，需要做出一些改变

**建议19：明白在C++中如何使用C**

         若想在C++中使用大量现成的C程序库，就必须把它放到extern“C” {/\* code \*/}中，extern “C”的作用就是告诉C++链接器寻找调用函数的符号时，采用C的方式。要实现在C++代码中调用C的代码，具体方式有以下几种：(1)、修改C代码的头文件，当其中含有C++代码时，在声明中加入extern “C”；(2)、在C++代码中重新声明一下C函数，在重新声明时添加上extern “C”；(3)、在包含C头文件时，添上extern “C”。

**建议20：使用memcpy()系列函数时要足够小心**

要区分哪些数据对象是POD(传统C风格的数据类型，C的所有对象都是POD，对于任何POD对象，我们都可以放心大胆地使用memset()、memcpy()、memcmp()等函数对对象的内存数据进行操作)，哪些是非POD(C++的对象可能并不是一个POD，如动多态)，由于非POD对象的存在，在C++中使用memcpy()系列函数时要保持足够的小心。

**建议21：尽量用new/delete代替malloc/free**

malloc与new之间的区别：(1)、new是C++运算符，而malloc则是C标准库函数；(2)、通过new创建的东西是具有类型的，而malloc函数返回的则是void\*，需要进行强制转型；(3)、new可以自动调用对象的构造函数，而malloc不会；(4)、new失败是会调用new\_handler处理函数，而malloc失败则直接返回NULL。Free与delete之间的区别：(1)、delete是C++运算符，free是C标准库函数；(2)、delete可以自动调用对象的析构函数，而malloc不会。另外，new/delete必须配对使用，malloc/free也一样。

**建议22：灵活地使用不同风格的注释**

C风格的注释/\* \*/与C++风格的注释//在C++语言中同时存在。

**建议23：尽量使用C++标准的iostream**

建议使用#include <iostream>

**建议24：尽量采用C++风格的强制转型**

**建议25：尽量用const、enum、inline替换#define**

对于简单的常量,应该尽量使用const对象或枚举类型数据，避免使用#define；对于形似函数的宏，尽量使用内联函数，避免使用#define。总之一句话，尽量将工作交给编译器，而不是预处理器。

**建议26：用引用代替指针**

与指针不同，引用与地址没有关联，甚至不占任何存储空间。

### 第3章   说一说“内存管理”的那点事儿

在VC中，栈空间未初始化的字符默认是-52，补码是0xCC，两个0xCC，即0xCCCC在GBK编码中就是“烫”；堆空间未初始化的字符默认是-51，两个-51在GBK编码中就是“屯”。两者都是未初始化的内存。

**建议27：区分内存分配的方式**

一个程序要运行，就必须先将可执行的程序加载到计算机内存里，程序加载完毕后，就可以形成一个运行空间，按照代码区、数据区、堆区、栈区进行布局。代码区存放的是程序的执行代码；数据区存放的是全局数据、常量、静态变量等；堆区存放的则是动态内存，供程序随机申请使用；而栈区则存放着程序中所用到的局部数据。这些数据可以动态地反应程序中对函数的调用状态，通过其轨迹也可以研究其函数机制。

         在C++中，数据区又被分成自由存储区、全局/静态存储区和常量存储区，再加上堆区、栈区，也就是说，内存被分成了5个区。(1)、栈区：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元将自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是所分配的内存容量有限；(2)、堆区：堆就是那些由new分配的内存块，其释放编译器不会管它，而是由我们的应用程序控制它，一般一个new就对应于一个delete，如果程序员没有释放掉，那么在程序结束后，操作系统就会自动回收；(3)、自由存储区：是那些由malloc等分配的内存块，它和堆十分相似，不过它是用free来结束自己生命的；(4)、全局/静态存储区：全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量又分为初始化的和未初始化的，在C++里面没有作此区分，它们共同占用同一块内存区；(5)、常量存储区：这是一块比较特殊的存储区，里面存放的是常量，不允许修改。

         堆与栈的区别：(1)、管理方式不同：对于栈来讲，它是由编译器自动管理的，无须我们手工控制；对于堆来说，它的释放工作由程序员控制，容易产生memory leak；(2)、空间大小不同：一般来讲在32位系统下，堆内存可以达到4GB的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定空间大小的；(3)、碎片问题：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而产生大量的碎片，使程序效率降低。对于栈来讲，则不存在这个问题，其原因还要从栈的特殊数据结构说起。栈是一个具有严明纪律的队列，其中的数据必须遵循先进后出的规则，相互之间紧密排列，绝不会留给其他数据可插入之空隙，所以永远都不可能有一个内存块从栈中间弹出，它们必须严格按照一定的顺序一一弹出；(4)、生成方向：对于堆来讲，其生长方向是向上的，也就是向着内存地址增加的方向增长；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长的；(5)、分配方式：堆都是动态分配的，没有静态分配的堆。栈有两种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数完成，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放的，无须我们手工实现；(6)、分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：它会分配专门的寄存器存放栈的地址，而且压栈出栈都会有专门的指令来执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制很复杂，例如为了分配一块内存，库函数会按照一定的算法在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间(可能是由于内存碎片太多)，则可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存了，然后返回。显然，堆的效率比栈要低得多。

**建议28：new/delete与new[]/delete[]必须配对使用**

由于内置数据类型没有构造、析构函数，所以在针对内置数据类型时，释放内存使用delete或delete[]的效果都是一样的。

**建议29：区分new的三种形态**

         (1)、如果是在堆上建立对象，那么应该使用new operator，它会为你提供最为周全的服务；(2)、如果仅仅是分配内存，那么应该调用operator new，但初始化不在它的工作职责之内。如果你对默认的内存分配过程不满意，想单独定制，重载operator new是不二选择；(3)、如果想在一块已经获得的内存里建立一个对象，那就应该用placement new。但是通常情况下不建议使用，除非是在某些对时间要求非常高的应用中，因为相对于其他两个步骤，选择合适的构造函数完成对象初始化是一个时间相对较长的过程。

**建议30：new内存失败后的正确处理**

当使用new申请一块内存失败时，抛出异常std::bad\_alloc是C++标准中规定的标准行为，所以推荐使用try{p=new int[SIZE];} catch(std::bad\_alloc) {…} 的处理方式。但是在一些老旧的编译器中，却不支持该标准，它会返回NULL，此时具有C传统的Test\_for\_NULL代码形式便起了作用。所以，要针对不同的情形采取合理的处置方式。

**建议31：了解new\_handler的所作所为**

         在使用operatornew申请内存失败后，编译器并不是不做任何的努力直接抛出std::alloc异常，在这之前，它会调用一个错误处理函数(这个函数被称为new-handler)，进行相应的处理。通常，一个好的new-handler函数的处理方式必须遵循以下策略之一：(1)、使更大块内存有效；(2)、装载另外的new-handler；(3)、卸载new-handler；(4)、抛出异常；(5)、无返回。

**建议32：借助工具检测内存泄露问题**

内存泄露一般指的是堆内存的泄露。检测内存泄露的关键是能截获对分配内存和释放内存的函数的调用。通过截获的这两个函数，我们就能跟踪每一块内存的生命周期。每当成功分配一块内存时，就把它的指针加入一个全局的内存链中；每当释放一块内存时，再把它的指针从内存链中删除。这样当程序运行结束的时候，内存链中剩余的指针就会指向那些没有被释放的内存。这就是检测内存泄露的基本原理。

         检测内存泄露的常用方法有如下几种：(1)、MS C-Runtime Library内建的检测功能，要在非MFC程序中打开内存泄露的检测功能非常容易，只须在程序的入口处添加以下代码：

\_CrtSetDbgFlag(\_CrtSetDbgFlag(\_CRTDBG\_REPORT\_FLAG)| \_CRTDBG\_LEAK\_CHECK\_DF);

(2)、外挂式的检测工具：MS下BoundsChecker或Insure++；Linux下RationalPurify或Valgrind.

**建议33：小心翼翼地重载operator new/operator delete**

通过重载operator new 和 operatordelete的方法，可以自由地采用不同的分配策略，从不同的内存池中分配不同的类对象。但是是否选择重载operator new/delete 一定要深思熟虑。

**建议34：用智能指针管理通过new创建的对象**

智能指针auto\_ptr，要使用它，需要包含memory头文件：(1)、auto\_ptr对象不可作为STL容器的元素；(2)、auto\_ptr缺少对动态配置而来的数组的支持；(3)、auto\_ptr在被复制的时候会发生所有权转移。

**建议35：使用内存池技术提高内存申请效率与性能**

经典的内存池技术，是一种用于分配大量大小相同的小对象的技术。通过该技术可以极大地加快内存分配/释放过程。内存池技术通过批量申请内存，降低了内存申请次数，从而节省了时间。

### 第4章   重中之重的类

**建议36：明晰class与struct之间的区别**

C中的struct是一种数据类型。C++中struct被看成是一个对象，它可以包含函数，可以拥有构造函数、析构函数，同样拥有继承等能力。

         C++中的class与struct差别：(1)、class和struct如果定义了构造函数，就都不能用大括号进行初始化了；如果没有定义，struct可以用大括号初始化，而class只有在所有成员变量全是public的情况下，才可以用大括号进行初始化；(2)、关于默认访问权限：class中默认的成员访问权限是private的，而struct中则是public的；(3)、关于继承方式：class继承默认是private继承，而struct默认是public继承

**建议37：了解C++悄悄做的那些事**

         对于类，编译器会悄悄地完成很多事：隐士产生一个类的默认构造函数，拷贝构造函数，拷贝赋值运算符和析构函数。

**建议38：首选初始化列表实现类成员的初始化**

类成员的初始化可采用两种形式来完成：在构造函数体重赋值完成和用初始化类成员列表完成：(1)、const成员变量只能用成员初始化列表来完成初始化，而不能在构造函数内被赋值；(2)、如果类B中含有A类型的成员变量，而类A中又禁止了赋值操作，此时要想顺利地完成B中成员变量的初始化，就必须采用初始化列表方式。即使没有禁用赋值操作，还是不推荐采用函数体内的赋值初始化方式。因为这种方式存在着两种问题。第一，比起初始化列表，此方式效率偏低；第二，留有错误隐患。

         对于初始化列表，初始化的顺序与构造函数中的赋值方式不同，初始化列表中成员变量出现的顺序并不是真正初始化的顺序，初始化的顺序取决于成员变量在类中的声明顺序。只有保证成员变量声明的顺序与初始化列表顺序一致才能真正保证其效率。

**建议39：明智地拒绝对象的复制操作**

在某些需要禁止对象复制操作的情形下，可以将这个类相应的拷贝构造函数、赋值操作符operator = 声明为private，并且不要给出实现。或者采用更简单的方法：使用boost::noncopyable作为基类。

**建议40：小心，自定义拷贝函数**

如果类内部出现了动态配置的资源，我们就不得不自定义实现其拷贝函数了。在自定义拷贝函数时，应该保证拷贝一个对象的All Parts:所有数据成员及所有的基类部分。

**建议41：谨防因构造函数抛出异常而引发的问题**

判断构造对象成功与否，解决办法：抛出一个异常。构造函数抛出异常会引起对象的部分构造，因为不能自动调用析构函数，在异常发生之前分配的资源将得不到及时的清理，进而造成内存泄露问题。所以，如果对象中涉及了资源分配，一定要对构造之中可能抛出的异常做谨慎而细致的处理。

**建议42：多态基类的析构函数应该为虚**

虚函数的最大目的就是允许派生类定制实现。所以，用基类指针删除一个派生类对象时，C++会正确地调用整个析构链，执行正确的行为，以销毁整个对象。在实际使用虚析构函数的过程中，一般要遵守以下规则：当类中包含至少一个虚函数时，才将该类的析构函数声明为虚。因为一个类要作为多态基类使用时，它一定会包含一个需要派生定制的虚函数。相反，如果一个类不包含虚函数，那就预示着这个类不能作为多态基类使用。同样，如果一个类的析构函数非虚，那你就要顶住诱惑，决不能继承它，即使它是“出身名门”。比如标准库中的string、complex、以及STL容器。

         多态基类的析构函数应该是virtual的，也必须是virtual的，因为只有这样，虚函数机制才会保证派生类对象的彻底释放；如果一个类有一个虚函数，那么它就该有一个虚析构函数；如果一个类不被设计为基类，那么这个类的析构就应该拒绝为虚。

**建议43：绝不让构造函数为虚**

虚函数的工作机制：虚函数的多态机制是通过一张虚函数表来实现的。在构造函数调用返回之前，虚函数表尚未建立，不能支持虚函数机制，所以构造函数不允许设为虚。

**建议44：避免在构造/析构函数中调用虚函数**

成员函数、包括虚成员函数，都可以在构造、析构的过程中被调用。当一个虚函数被构造函数(包括成员变量的初始化函数)或者析构函数直接或间接地调用时，调用对象就是正在构造或者析构的那个对象。其调用的函数是定义于自身类或者其基类的函数，而不是其派生类或者最低派生类的其他基类的重写函数。

如果在构造函数或析构函数中调用了一个类的虚函数，那它们就变成普通函数了，失去了多态的能力。

**建议45：默认参数在构造函数中给你带来的喜与悲**

合理地使用默认参数可以有效地减少构造函数中的代码冗余，让代码简洁而有力。但是如果不够小心和谨慎，它也会带来构造函数的歧义，增加你的调试时间。

**建议46：区分Overloading、Overriding、Hiding之间的差异**

1、重载(Overloading)：是指同一作用域的不同函数使用相同的函数名，但是函数的参数个数或类型不同；2、重写(Overriding)：是指在派生类中对基类中的虚函数重新实现，即函数名和参数都一样，只是函数的实现体不一样，派生类对基类中的操作进行个性化定制就是重写。重写需要注意的问题：(1)、函数的重写与访问层级(public、private、protected)无关；(2)、const可能会使虚成员函数的重写失效；(3)重写函数必须和原函数具有相同的返回类型；3、隐藏(Hiding)：是指派生类中的函数屏蔽基类中具有相同名字的非虚函数。

**建议47：重载operator=的标准三步走**

1、不要让编译器帮你重载赋值运算符；2、一定要检查自赋值；3、赋值运算符重载需返回\*this的引用，引用之于对象的优点在于效率，为了能够更加灵活地使用赋值运算符，选择返回引用绝对是明智之举；4、赋值运算符重载函数不能被继承。如果需要给类的数据成员动态分配空间，则必须实现赋值运算符。

**建议48：运算符重载，是成员函数还是友元函数**

运算符重载的四项基本原则：(1)、不可臆造运算符；(2)、运算符原有操作数的个数、优先级和结合性不能改变；(3)、操作数中至少一个是自定义类型；(4)、保持重载运算符的自然含义。

         运算符的重载可采用两种形式：成员函数形式和友元函数形式。(1)、重载为成员函数时，已经隐含了一个参数，它就是this指针；对于双目运算符，参数仅有一个；(2)、当重载友元函数时，将不存在隐含的参数this指针；如果运算符被重载为友元函数，那么它就获得一种特殊的属性，能够接受左参数和右参数的隐式转换，如果是成员函数版的重载则只允许右参数的隐式转换。一般说来，建议遵守一个不成文的规定：对双目运算符，最好将其重载为友元函数，因为这样更方便些；而对于单目运算符，则最好重载为成员函数。

**建议49：有些运算符应该成对实现**

         为了更好地适应使用习惯，很多运算符重载时最好成对实现，比如==与!=、<与>、<=与>=、+与+=、-与-=、\*与\*=、/与/=。

**建议50：特殊的自增自减运算符重载**

后缀操作重载时返回值应该为一个const对象。

**建议51：不要重载operator&&、operator||以及operator,**

“&&”、“||”、“,”(逗号运算符)都具有较为特殊的行为特性，重载会改变运算符的这些特性，进而影响原有的习惯，所以不要去重载这三个可以重载的运算符。

**建议52：合理地使用inline函数来提高效率**

内联函数具有与宏定义相同的代码效率，但在其他方面却要优于宏定义。因为内联函数还遵循函数的类型和作用域规则。内联函数一般情况下都应该定义在头文件中。内联函数的定义分为两种方式：(1)、显示方式：在函数定义之前添加inline关键字，内联函数只有和函数体声明放在一起时inline关键字才具有效力；(2)、隐式方式：将函数定义于类的内部。一个给定的函数是否得到内联，很大程度上取决于你正在使用的编译器。

         使用内联函数应该注意：(1)、内联函数的定义必须出现在内联函数第一次被调用之前。所以，它一般会置于头文件中；(2)、在内联函数内不允许用循环语句和开关语句，函数不能过于复杂；(3)、依据经验，内联函数只适合于只有1~5行的小函数；(4)、对于内存空间有限的机器而言，慎用内联。过分地使用内联会造成函数代码的过度膨胀，会占用太多空间；(5)、不要对构造/析构函数进行内联；(6)、大多开发环境不支持内联调试，所以为了调试方便，不要将内联优化放在调试阶段之前。

**建议53：慎用私有继承**

私有继承会使基类的所有东西(包括所有的成员变量与成员函数)在派生类中变成private的，也就是说基类的全部在派生类中都只能作为实现细节，而不能成为接口。私有继承意味着“只有implementation 应该被继承，interface应该被忽略”，代表着是“is-implemented-in-terms-of”的内在关系。通常情况下，这种关系可以采用组合的方式来实现，并提倡优先使用组合的方案。但是如果存在虚函数和保护成员，就会使组合方案失效，那就应使用私有继承。

**建议54：抵制MI的糖衣炮弹**

MI(多重继承)意味着设计的高复杂性、维护的高难度性，尽量少使用MI。

**建议55：堤防对象切片**

多态的实现必须依靠指向同一类族的指针或引用。否则，就可能出现著名的对象切片(Object Slicing)问题。所以，在既有继承又有虚函数的情况下，一定要提防对象切片问题。

**建议56：在正确的场合使用恰当的特性**

(1)、虚函数：虚函数机制的实现是通过虚函数表和指向虚函数表的指针来完成的。关键字virtual告诉编译器该函数应该实现晚绑定，编译器对每个包含虚函数的类创建虚函数表VTable，以放置类的虚函数地址。编译器密码放置了指向虚函数表的指针VPtr，当多态调用时，它会使用VPtr在VTable表中查找要执行的函数地址；(2)、多重继承：对于多重继承来说，对象内部会有多个VPrt，所以这就使偏移量计算变得复杂了，而且会使对象占用的空间和运行时开销都变大；(3)、虚基类：它与多重继承的情况类似，因为虚基类就是为了多重继承而产生的；(4)、运行时类型检测(RTTI)：是我们在程序运行时得到对象和类有关信息的保证。

**建议57：将数据成员声明为private**

将数据成员声明为private是具有相当充分的理由的：(1)、实现数据成员的访问控制；(2)、在将来时态下设计程序，为之后的各种实现提供弹性；(3)、保持语法的一致性。

**建议58：明晰对象的构造与析构的顺序**

(1)、对象的构造都是从类的最根处开始的，由深及浅，先基类后子类，层层构造，这个顺序不能改变。如果含有多个基类，那么就按照声明顺序由前及后执行。析构函数则严格按照构造的逆序执行；(2)、成员对象构造函数的调用顺序与成员对象的声明顺序严格一致，析构顺序是构造顺序的严格逆序。这是因为类的声明是绝对唯一的，而类的构造函数可以有多个，所以按照声明才会使析构函数得到唯一的逆序；(3)、如果继承遇到成员对象，基类构造函数依然会被首先调用，然后调用成员对象的构造函数。

**建议59：明了如何在主调函数启动前调用函数**

如果想在主程序main启动之前调用某些函数，调用全局对象的构造函数绝对是一个很不错的方法。因为从概念上说，全局对象是在程序开始前已经完成了构造，而在程序执行之后才会实施析构。

### 第5章   用好模板，向着GP(泛型编程)开进

**建议60：审慎地在动、静多态之间选择**

         虚函数机制配合继承机制，生效于运行期，属于晚绑定，是动多态；而模板将不同的行为和单个泛化记号相关联发生在编译期，属于早绑定，被称为静多态。(1)、动多态：它的技术基础是继承机制和虚函数，它在继承体系之间通过虚函数表来表达共同的接口；(2)、静多态：它的技术基础是模板。与动多态相比，静多态始终在和参数“较劲儿”，它适用于所有的类，与虚函数无关。从应用形式上看，静多态是发散式的，让相同的实现代码应用于不同的场合；动多态是收敛式的，让不同的实现代码应用于相同的场合。从思维方式上看，前者是泛型式编程风格，它看重的是算法的普适性；后者是对象式编程风格，它看重的是接口与实现的分离度。两者区别：(1)、动多态的函数需要通过指针或引用传参，而静多态则可以传值、传指针、传引用等，“适应性”更强；(2)、在性能上，静多态优于动多态，因为静多态无间接访问的迂回代码，它是单刀直入的；(3)、因为实现多态的先后顺序不同，所以如果出现错误，它们抛出错误的时刻也不一样，动多态会在运行时报错，而静多态则在编译时报错。

**建议61：将模板的声明和定义放置在同一个头文件里**

模板类型不是一种实类型，它必须等到类型绑定后才能确定最终类型，所以在实例化一个模板时，必须要能够让编译器“看到”在哪里使用了模板，而且必须要看到模板确切的定义，而不仅仅是它的声明，否则将不能正常而顺利地产生编译代码。函数模板、类模板不同于一般的函数、类，它们不能像一般的方式那样进行声明与定义，标准要求模板的实例化与定义体必须放在同一翻译单元中。实现这一目标有三种方法(将模板的声明和定义都放置在同一个.h文件中；按照旧有的习惯性做法来处理，声明是声明，实现是实现，二者相互分离，但是需要包含头文件的地方做一些改变，如，在使用模板时，必须用#include “Temp.cpp”替换掉#include “Temp.h”；使用关键字export来定义具体的模板类对象和模板函数)，但是最优策略还是：将模板的声明和定义都放置在同一个.h文件中，虽然在某种程度上这破坏了代码的优雅性。

**建议62：用模板替代参数化的宏函数**

参数化的宏函数有着两个致命缺点：(1)、缺乏类型检查；(2)、有可能在不该进行宏替换的时候进行了替换，违背了作者的意图。模板是实现代码复用的一种工具，它可以实现类型参数化，达到让代码真正复用的目的。

**建议63：区分函数模板与模板函数、类模板与模板类**

函数模板的重点在于“模板”两个字，前面的“函数”只是一个修饰词。其表示的是一个专门用来生产函数的模板。而模板函数重点在“函数”，表示的是用模板所生成的函数。函数模板的一般定义形式为：

         Template<class数据类型参数标识符>

         返回类型标识符 函数名(数据类型参数标识符 形参)

         {  //… …}

         将函数模板的模板参数实例化后会生成具体的函数，此函数就是模板函数。由函数模板所生成的模板函数的一般形式为：

         函数名<数据类型参数标识符>(数据类型参数标识符 形参)

         类模板是为类定义的一种模式，它使类中的一些数据成员和成员函数的参数或返回值可以取任意的数据类型。在类定义中，凡是采用标准数据类型的数据成员、成员函数的参数前面都要加上类型标识符，在返回类型前也要进行同样的处理。如果类中的成员函数要在类的声明之外定义，则它必须是模板函数。将类模板的模板参数实例化后生成的具体类，就是模板类。函数模板和类模板处于实例化之前，而模板函数或模板类则在实例化之后。

**建议64：区分继承与模板**

模板的长处在于处理不同类型间“千篇一律”的操作。相较于类继承，这些类不必具有什么相同的性质。

### 第6章   让神秘的异常处理不再神秘

**建议65：使用exception来处理错误**

异常能：(1)、增强程序的健壮性；(2)、使代码变得更简洁优美、更易维护；(3)、错误信息更灵活、丰富。

**建议66：传值throw异常，传引用catch异常**

throw byvalue, catch by reference

**建议67：用“throw;”来重新抛出异常**

对于异常的重新抛出，需要注意：(1)、重新抛出的异常对象只能出现在catch块或catch调用的函数中；(2)、如果在处理代码不执行时碰到“throw ;”语句，将会调用terminate函数。

**建议68：了解异常捕获与函数参数传递之间的差异**

异常与函数参数的传递之间差异：(1)、控制权；(2)、对象拷贝的次数；(3)、异常类型转换；(4)、异常类型匹配。

**建议69：熟悉异常处理的代价**

异常处理在带来便利的同时，也会带来时间和空间上的开销，使程序效率降低，体积增大，同时会加大代码调试和管理的成本。

**建议70：尽量保证异常安全**

如果采用了异常机制，请尽量保证异常安全：努力实现强保证，至少实现基本保证。

### 第7章   用好STL这个大轮子

**建议71：尽量熟悉C++标准库**

C++标准库主要包含的组件：(1)、C标准函数库；(2)、输入/输出(input/output)；(3)、字符串(string)；(4)、容器(containers)；(5)、算法(algorithms)；(6)、迭代器(iterators)；(7)、国际化(internationalization)；(8)、数值(numerics)；(9)、语言支持(languagesupport)；(10)、诊断(diagnostics)；(11)、通用工具(general utilities)。字符串、容器、算法、迭代器四部分采用了模板技术，一般被统称为STL(Standard Template Library,即标准模板库)。

         在C++标准中，STL被组织成了13个头文件：<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack>、<utility>。

**建议72：熟悉STL中的有关术语**

(1)、容器：是一个对象，它将对象作为元素来存储；(2)、泛型(Genericity)：泛型就是通用，或者说是类型独立；(3)算法：就是对一个对象序列所采取的某些操作，例如std::sort()、std::copy()、std::remove()；(4)、适配器(Adaptor)：是一个非常特殊的对象，它的作用就是使函数转化为函数对象，或者是将多参数的函数对象转化为少参数的函数对象；(5)、O(h)：它是一个表示算法性能的特殊符号，在STL规范中用于表示标准库算法和容器操作的最低性能极限；(6)、迭代器：是一种可以当做通用指针来使用的对象，迭代器可以用于元素遍历、元素添加和元素删除。

**建议73：删除指针的容器时避免资源泄露**

STL容器虽然智能，但尚不能担当删除它们所包含指针的这一责任。所以，在要删除指针的容器时须避免资源泄露：或者在容器销毁前手动删除容器中的每个指针，或者使用智能引用计数指针对象(比如Boost的shared\_ptr)来代替普通指针。

**建议74：选择合适的STL容器**

容器分为：(1)、标准STL序列容器：vector、string、deque和list；(2)、标准STL关联容器：set、multiset、map和multimap；(3)、非标准序列容器：slist(单向链表)和rope(重型字符串)；(4)、非标准关联容器：hash\_set、hash\_multiset、hash\_map和hash\_multimap；(5)、标准非STL容器：数组、bitset、valarray、stack、queue和priority\_queue。

**建议75：不要在STL容器中存储auto\_ptr对象**

auto\_ptr是C++标准中提供的智能指针，它是一个RAII对象，它在初始化时获得资源，析构时自动释放资源。C++标准中规定：STL容器元素必须能够进行拷贝构造和赋值操作。禁止在STL容器中存储auto\_ptr对象原因有两个：(1)、auto\_ptr拷贝操作不安全，会使原指针对象变NULL；(2)、严重影响代码的可移植性。

**建议76：熟悉删除STL容器中元素的惯用法**

(1)、删除容器中具有特定值的元素：如果容器是vector、string或deque，使用erase-remove的惯用法(remove只会将不应该删除的元素前移，然后返回一个迭代器，该迭代器指向的是那个应该删除的元素，所以如果要真正删除这一元素，在调用remove之后还必须调用erase)；如果容器时list，使用list::remove；如果容器是标准关联容器，使用它的erase成员函数；(2)、删除容器中满足某些条件的所有元素：如果容器是vector、string或deque，使用erase-remove\_if惯用法；如果容器是list，使用list::remove\_if；如果容器是标准关联容器，使用remove\_copy\_if & swap组合算法，或者自己写一个遍历删除算法。

**建议77：小心迭代器的失效**

         迭代器是一个对象，其内存大小为12(sizeof(vector<int>::iterator)，vs2010,32bit)。引起迭代器失效的最主要操作就是插入、删除。对于序列容器(如vector和deque)，插入和删除操作可能会使容器的部分或全部迭代器失效。因为vector和deque必须使用连续分配的内存来存储元素，向容器中添加一个元素可能会导致后面邻接的内存没有可用的空闲空间而引起存储空间的重新分配。一旦这种情况发生，容器中的所有的迭代器就会全部失效。

**建议78：尽量使用vector和string代替动态分配数组**

相较于内建数组，vector和string具有几方面的优点：(1)、它们能够自动管理内存；(2)、它们提供了丰富的接口；(3)、与C的内存模型兼容；(4)、集众人智慧之大成。

**建议79：掌握vector和string与C语言API的通信方式**

使用vector::operator[]和string::c\_str是实现STL容器与C语言API通信的最佳方式。

**建议80：多用算法调用，少用手写循环**

用算法调用代替手工编写的循环，具有几方面的优点：(1)、效率更高；(2)、不易出错；(3)、可维护性更好。

## 第二部分 编码习惯和规范篇

### 第8章   让程序正确执行

**建议81：避免无意中的内部数据裸露**

         对于const成员函数，不要返回内部数据的句柄，因为它会破坏封装性，违反抽象性，造成内部数据无意中的裸露，这会出现很多“不可思议”的情形，比如const对象的非常量性。

**建议82：积极使用const为函数保驾护航**

const的真正威力体现在几个方面：(1)、修饰函数形式的参数：const只能修饰输入参数，对于内置数据类型的输入参数，不要将“值传递”的方式改为“const 引用传递”；(2)、修饰函数返回值；(3)、修饰成员函数：用const修饰成员函数的目的是提高程序的健壮性。const成员函数不允许对数据成员进行任何修改。

         关于const成员函数，须遵循几个规则：(1)、const对象只能访问const成员函数，而非const对象可以访问任意的成员函数；(2)、const对象的成员是不可修改的，然而const对象通过指针维护的对象却是可以修改的；(3)、const成员函数不可以修改对象的数据，不管对象是否具有const性质。

**建议83：不要返回局部变量的引用**

局部变量的引用是一件不太靠谱的事儿，所以尽量避免让函数返回局部变量的引用。同时也不要返回new生成对象的引用，因为这样会让代码层次混乱，让使用者苦不堪言。

**建议84：切忌过度使用传引用代替传对象**

相较于传对象，传引用的优点：它减少了临时对象的构造与析构，所以更具效率。但须审慎地使用传引用替代传对象，必须传回内部对象时，就传对象，勿传引用。

**建议85：了解指针参数传递内存中的玄机**

用指针参数传回一块动态申请的内存，是很常见的一种需求。然而如果不甚小心，就很容易造成严重错误：程序崩溃+内存泄露，解决之道就是用指针的指针来传递，或者换种内存传递方式，用返回值来传递。

**建议86：不要讲函数参数作为工作变量**

工作变量，就是在函数实现中使用的变量。应该防止将函数参数作为工作变量，而对于那些必须改变的参数，最好先用局部变量代替之，最后再将该局部变量的内容赋给该参数，这样在一定程度上保护了数据的安全。

**建议87：躲过0值比较的层层陷阱**

(1)、0在不在该类型数据的取值范围内？(2)、浮点数不存在绝对0值，所以浮点零值比较需特殊处理；(3)区分比较操作符==与赋值操作符=，切忌混淆。

**建议88：不要用reinterpret\_cast去迷惑编译器**

reinterpret\_cast，简单地说就是保持二进制位不变，用另一种格式来重新解释，它就是C/C++中最为暴力的类型转换，所实现的是一个类型到一个毫不相关、完全不同类型的映射。reiterpret\_cast仅仅重新解释了给出对象的比特模型，它是所有类型转换中最危险的。尽量避免使用reinterpret\_cast，除非是在其他转换都无效的非常情形下。

**建议89：避免对动态对象指针使用static\_cast**

在类层次结构中，用static\_cast完成基类和子类指针(或引用)的下行转换是不安全的。所以尽量避免对动态对象指针使用static\_cast，可以用dynamic\_cast来代替，或者优化设计，重构代码。

**建议90：尽量少应用多态性数组**

多态性数组一方面会涉及C++时代的基类指针与派生类指针之间的替代问题，同时也会涉及C时代的指针运算，而且常会因为这二者之间的不协调引发隐蔽的Bug。

**建议91：不要强制去除变量的const属性**

在C++中，const\_cast<T\*>(a)一般用于从一个类中去除以下这些属性：const、volatile和\_unaligned.强制去除变量的const属性虽然可以带来暂时的便利，但这不仅增加了错误修改变量的几率，而且还可能会引发内存故障。

### 第9章   提高代码的可读性

**建议92：尽量使代码版面整洁优雅**

(1)、避免代码过长；(2)、代码缩进和对齐；(3)、空行分隔段落；(4)、使用空格；(5)、语句行。

**建议93：给函数和变量起一个“能说话”的名字**

(1)、名称必须直观，可望文生义，不必解码；(2)、长度要符合“min\_length && max\_information”(最小名长度最大信息量)的原则；(3)、与整体风格保持一致；(4)、变量名称应该是一个“名词”，或者是“形容词+名词”；而函数名称应该是“动词+名词”的组合；(5)、杜绝仅靠大小写来区分的名称标示符；(6)、变量名之前附加前缀用来识别变量类型；(7)、C++类或结构的成员变量附加前缀“m\_”；全局变量名称附加前缀“g\_”；(8)、单字符变量只能用作循环变量；(9)、类名采用“C+首字母大写的单词”形式来命名。

**建议94：合理地添加注释**

(1)、使用统一的注释方法为每个层次的代码块添加注释；(2)、避免不必要的注释；(3)、掌握代码注释量的一个度；(4)、边写代码加边注释；(5)、注释要简明而准确；(6)、注意特有标签的特有作用。

**建议95：为源代码设置一定的目录结构**

如果一个软件所涉及的文件数目比较多，通常要将其进行划分，为其设置一定的目录结构，以便于维护，如include、lib、src、doc、release、debug。

**建议96：用有意义的标识代替Magic Numbers**

用宏或常量替代信息含量较低的MagicNumbers，绝对是一个好习惯，这样可提高代码的可读性与可维护性。

**建议97：避免使用“聪明的技巧”**

**建议98：运算符重载时坚持其通用的含义**

**建议99：避免嵌套过深与函数过长**

**建议100：养成好习惯，从现在做起**

### 第10章   让代码运行得再快些

**建议101：用移位实现乘除法运算**

在大部分的C/C++编译器中，用移位的方法比直接调用乘除法子程序生成代码的效率要高。只要是乘以或除以一个整数常量，均可用移位的方法得到结果，如a=a\*9可以拆分成a=a\*(8+1)，即a=a(a<<3)+a。移位只对整数运算起作用。

**建议102：优化循环，提高效率**

应当将最长的循环放在最内层，最短的循环放在最外层，以减少CPU跨切循环层的次数，提高效率。

**建议103：改造switch语句**

对于case的值，推荐按照它们发生的相对频率来排序，把最可能发生的情况放在第一位，最不可能的情况放在最后。

**建议104：精简函数参数**

函数在调用时会建立堆栈来存储所需的参数值，因此函数的调用负担会随着参数列表的增长而增加。所以，参数的个数会影响进栈出栈的次数，当参数很多的时候，这样的操作就会花费很长的时间。因此，精简函数参数，减少参数个数可以提高函数调用的效率。如果精简后的参数还是比较多，那么可以把参数列表封装进一个单独的类中，并且可以通过引用进行传递。

**建议105：谨慎使用内嵌汇编**

汇编语言与其他高级语言相比，更接近机器语言，效率更高，所以在应用程序中如果碰到那些对时间要求苛刻的部分，可以采用汇编语言来重写。

**建议106：努力减少内存碎片**

经常性地动态分配和释放内存会造成堆碎片，尤其是应用程序分配的是很小的内存块时。避免堆碎片：(1)、尽可能少地使用动态内存，在大多数情况下，可以使用静态或自动储存，或者使用STL容器，减少对动态内存的依赖；(2)、尽量分配和重新分配大块的内存块，降低内存碎片发生的几率。内存碎片会影响程序执行的效率。

**建议107：正确地使用内联函数**

内联(inline)函数既能够去除函数调用所带来的效率负担，又能够保留一般函数的优点。只有当函数非常短小的时候使用inline才能得到预想中的效果。对于编译器来说，函数内联与否取决于以下关键性的因素：(1)、函数体的大小；(2)、是否有局部对象被声明；(3)、函数的复杂性(是否存在函数调用、递归等)。

**建议108：用初始化取代赋值**

以用户初始化代替赋值，可以使效率得到较大的提升，因为这样可以避免一次赋值函数operator =的调用。因此，当我们在赋值和初始化之间进行选择时，初始化应该是首选。需要注意的是，对基本的内置数据类型而言，初始化和赋值之间是没有差异的，因为内置类型没有构造和析构过程。

**建议109：尽可能地减少临时对象**

临时对象产生的主要情形及避免方法：(1)、参数：采用传常量引用或指针取代传值；(2)、前缀或后缀：优先采用前缀操作；(3)、参数转换：尽量避免这种转换；(4)、返回值：遵循single-entry/single-exit原则，避免同一个函数中存在多个return语句。

**建议110：最后再去优化代码**

在进行代码优化之前，需要知道：(1)、算法是否正确；(2)、如何在代码优化和可读性之间进行选择；(3)、该如何优化：代码分析(profiling)工具；(4)、如何选择优化方向：先算法，再数据结构，最后才是实现细节。

### 第11章   零碎但重要的其他建议

**建议111：采用相对路径包含头文件**

一个“点”(“.\”)代表的是当前目录所在的路径，两个“点”(“..\”)代表的是相对于当前目录的上一次目录路径。

         当写#include语句时，推荐使用相对路径；此外，要注意使用比较通用的正斜线“/”，而不要使用仅在Windows下可用的反斜线“\”。

**建议112：让条件编译为开发出力**

条件编译中的预处理命令主要包括：#if、#ifndef、#ifdef、#endif和#undef等，它们的主要功能是在程序编译时进行有选择性的挑选，注释掉一些指定的代码，以达到版本控制、防止对文件重复包含等目的。

**建议113：使用.inl文件让代码整洁可读**

.inl文件是内联函数的源文件，.inl文件还可用于模板的定义。.inl文件可以将头文件与内联函数的复杂定义隔离开来，使代码整洁可读，如果将其用于模板定义，这一优点更加明显。

**建议114：使用断言来发现软件问题**

断言assert，只会在调试模式下生成代码，而在Release版本中则直接无视之。

**[cpp]** [view plaincopy](http://blog.csdn.net/fengbingchun/article/details/9193577)

1. assert(pData!= NULL && “In Function DataProcess”)
2. assert(0&& “MSG\_HANDLE\_ERROR”)

**建议115：优先选择编译和链接错误**

静态检查：编译器必须检查源程序是否符合源语言规定的语法和语义要求，静态检查的主要工作就是语义分析，它是独立于数据和控制流的，可信度相对较高，而且不会增加程序的运行时开销。

         动态检查：是在运行时刻对程序的正确性、安全性等做检查，比如内存不足、溢出、数组越界、除0等，这类检查对于数据和控制流比较依赖。

         C/C++语言属于一种静态语言。一个设计较好的C++程序应该是较少地依赖动态检查，更多地依赖静态检查。

**建议116：不放过任何一条编译器警告**

强烈建议：(1)、把编译器的警告级别调至最高；(2)、不要放过编译器的任何一条警告信息。

**建议117：尽量减少文件之间的编译依赖**

不要在头文件中直接包含要使用的类的头文件(除了标准库)，直接包含头文件这种方式相对简单方便，但是会耗费大量的编译时间。推荐使用**类的前向声明**来减少文件直接的编译依赖。用对类声明的依赖替代对类定义的依赖，这是减少编译依赖的原则。

         为了加快编译进程，减少时间的浪费，我们应该尽量减少头文件依赖，其中的可选方案包括前向声明、柴郡猫技术等。

**建议118：不用在头文件中使用using**

名空间是C++提供的一种机制，可以有效地避免函数名污染。然而在应用时要十分注意：任何情况下都不应在头文件中使用“using namespace XXX”这样的语句，而应该在定义时直接用全称。

**建议119：划分全局名空间避免名污染**

使用自己的名空间将全局名空间合理划分，会有效地减少名污染问题，因为，不要简单地将所有的符号和名称统统扔进全局名空间里。

## 第三部分 程序架构和思想篇

### 第12章   面向对象的类设计

**建议120：坚持“以行为为中心”的类设计**

“以数据为中心”关注类的内部数据结构，习惯将private类型的数据写在前面，而将public类型的函数写在后面。

         “以行为为中心”关注的重心放在了类的服务和接口上，习惯将public类型的函数写在前面，而将private类型的数据写在后面。

**建议121：用心做好类设计**

在设计一个类时，首先，类的设计就是数据类型的设计，在数据类型的设计中，(1)、类应该如何创建和销毁呢？这会影响到类的构造函数和析构函数的设计。首先应该确定类是否需要分配资源，如果需要，还要确定这些资源又该如何释放。(2)、类是否需要一个无参构造函数？如果需要，而恰恰此时这个类已经有了构造函数，那么我们就得显示地写一个。(3)、类需要复制构造函数吗？其参数上加上了const修饰吗？它是用来定义这个类传值(pass-by-value)的具体实现的。(4)、所有的数据成员是不是都已经在构造函数中完成了初始化呢？(5)、类需要赋值操作符吗？赋值操作符能正确地将对象赋给对象本身吗？它与初始化有什么不同？其参数上加上了const修饰吗？(6)、类的析构函数需要设置为virtual吗？(7)、类中哪些值得组合是合法的？合法值的限定条件是什么？在成员函数内部是否对变量值得合法性做了检查？其次，类的设计是对现实对象进行抽象的一个过程。再次，数据抽象的过程其实是综合考虑各方面因素进行权衡的一个过程。

**建议122：以指针代替嵌入对象或引用**

设计类的数据成员时，可以有三种选择：(1)、嵌入对象；(2)、使用对象引用；(3)、使用对象指针。

         如果在类数据成员中使用到了自定义数据类型，使用指针是一个较为明智的选择，它有以下几方面的优点：(1)、成员对象类型的变化不会引起包含类的重编译；(2)、支持惰性计算，不创建不使用的对象，效率更高；(3)、支持数据成员的多态行为。

**建议123：努力将接口最小化且功能完善**

类接口的目标是完整且最小。精简接口函数个数，使每一个函数都具有代表性，并且使其功能恰好覆盖class的智能，同时又可以获得接口精简所带来的好处：(1)、利于理解、使用，维护成本也相对较低；(2)、可以缩小头文件长度，并缩短编译时间。

**建议124：让类的数据隐藏起来**

坚持数据封装，坚持信息隐藏，杜绝公有、保护属性的存在(数据成员私有、柴郡猫技术)。

**建议125：不要让成员函数破坏类的封装性**

小心类的成员函数返回属性变量的“直接句柄”，它会破坏辛辛苦苦搭建维护的封装性，一种方法，将函数的返回值加上const修饰。

**建议126：理解“virtual + 访问限定符”的深层含义**

virtual关键字是C++中用于实现多态的重要机制，其核心理念就是通过基类访问派生类定义的函数。

         (1)、基类中的一个虚拟私有成员函数，表示实现细节是可以被派生类修改的；(2)、基类中的一个虚拟保护成员函数，表示实现细节是必须被派生类修改的；(3)、基类中的一个虚拟公有成员函数，则表示这是一个接口，不推荐，建议用protected virtual 来替换。

**建议127：谨慎恰当地使用友元机制**

通常说来，类中的私有成员一般是不允许外面访问的。但是友元可以超脱这条禁令，它可以访问该类的私有成员。所带来的最大好处就是避免了类成员函数的频繁调用，节约了处理器的开销，提高了程序的效率。但是，通常，大家认为“友元破坏了类的封装性”。采用友元机制，一般是基于这样的需求：一个类的部分成员需要对个别其他类公开。

**建议128：控制对象的创建方式**

栈和堆是对象的主要分布区，它们对应着两种基本的对象创建方式：以new方式手动管理的堆创建和只需声明就可使用的栈创建。

         控制对象的创建方式：(1)、要求在堆中建立对象：为了执行这种限制，必须找到一种方法保证调用new是建立对象的唯一手段。非堆对象是在定义它时自动构造的，而且是在生存期结束时自动释放的。将析构函数声明为private，而构造函数保持为public；(2)、禁止在堆中建立对象：要禁止调用new来建立对象，可以通过将operator new函数声明为private来实现。

**建议129：控制实例化对象的个数**

当实例化对象唯一时，采用设计模式中的单件模式；当实例化对象为N(N>0)个时，设置计数变量是一个思路。

**建议130：区分继承与组合**

(1)、继承：C++的“继承”特性可以提高程序的可复用性。继承规则：若在逻辑上B是一种A，并且A的所有功能和属性对B而言都有意义，则允许B继承A的功能和属性。继承易于修改或扩展那些被复用的实现。但它的这种“白盒复用”却容易破坏封装性，因为这会将父类的实现细节暴露给子类。当父类实现更改时，子类也不得不随之更改，所以，从父类继承来的实现将不能在运行期间进行改变；(2)、组合：在逻辑上表示的是“有一个(Hase-A)”的关系，即A是B的一部分。组合属于“黑盒”复用，被包含对象的内部细节对外是不可见的。所以，它的封装性相对较好，实现上的相互依赖性比较小。并且可以通过获取指向其他的具有相同类型的对象引用，在运行期间动态地定义组合。而其缺点就是致使系统中的对象过多。Is-A关系用继承表达，Has-A关系用组合表达，优先使用(对象)组合。

**建议131：不要将对象的继承关系扩展至对象容器**

A是B的基类，B是一种A，但是B的容器却不能是这种A的容器。

**建议132：杜绝不良继承**

在继承体系中，派生类对象必须是可以取代基类对象的。

**建议133：将RAII作为一种习惯**

RAII(ResourceAcquisition Is Initialization)，资源获取即初始化，RAII是C++语言的一种管理资源、避免泄露的惯用方法。RAII的做法是使用一个对象，在其构造时获取资源，在对象生命周期中控制对象资源的访问，使之始终保持有效，最后再对象析构时释放资源。实现这种功能的类即采用了RAII方式，这样的类被称为封装类。

**建议134：学习使用设计模式**

设计模式是用来“封装变化、降低耦合”的工具，它是面向对象设计时代的产物，其本质就是充分运用面向对象的三个特性(即：封装、继承和多态)，并进行灵活的组合。

**建议135：在接口继承和实现继承中做谨慎选择**

在接口继承和实现继承之间进行选择时，需要考虑的一个因素就是：基类的默认版本。对于那些无法提供默认版本的函数接口我们选择函数接口继承；而对于那些能够提供默认版本的，函数实现继承就是最佳选择。

**建议136：遵循类设计的五项基本原则**

(1)、单一职责原则(SRP)：一个类，最好只做一件事。SRP可以看作是低耦合、高内聚在面向对象原则上的引申；(2)、开闭原则(OCP)：对扩展开放，对更改关闭，应该能够不用修改原有类就能扩展一个类的行为；(3)、替换原则(LSP )：子类应当可以替换父类并出现在父类能够出现的任何地方。反过来则不成立，子类可以替换基类，但是基类不一定能替换子类；(4)、依赖倒置原则(DIP)：高层模块不依赖于底层模块，而是二者都依赖于抽象，即抽象不依赖于具体，具体依赖于抽象。依赖一定会存在类与类、模块与模块之间。当两个模块之间存在紧密的耦合关系时，最好的方法就是分离接口和实现：在依赖之间定义一个抽象的接口使得高层模块调用接口，底层模块实现接口的定义，从而有效控制耦合关系，达到依赖于抽象的设计目的；(5)、接口分离原则(ISP)：使用多个小的专门的接口，而不要使用一个大的总接口。接口有效地将细节和抽象隔离开来，体现了对抽象编程的一切好处，接口隔离强调接口的单一性。分离的手段主要有两种方式：一个是利用委托分离接口，另一个是利用多重继承分离接口。

### 第13章  返璞归真的程序设计

**建议137：用表驱动取代冗长的逻辑选择**

表驱动法(Table drivenmethod)，是一种不必用很多的逻辑语句(if或case)就可以把表中信息找出来的方法。它是一种设计模式，可用来代替复杂的if/else或switch-case逻辑判断。

**建议138：为应用设定特性集**

对待C++高级特性的态度一定要谨慎，是否有必要使用多重继承、异常、RTTI、模板及模板元编程，一定要做一个审慎的评估，以便为应用选择合适的特征集，避免使用过分复杂的设计和功能，否则将会使得代码难于理解和维护。

**建议139：编码之前需三思**

在让电脑运行你的程序之前，先让你的大脑编译运行。

**建议140：重构代码**

重构无止境，重构你的代码，精雕细琢，千锤百炼。

**建议141：透过表面的语法挖掘背后的语义**

**建议142：在未来时态下开发C++程序**

在未来时态下开发C++程序，需要考虑代码的可重用性、可维护性、健壮性，以及可移植性。

**建议143：根据你的目的决定造不造轮子**

在编程语言中这些轮子表现为大量的通用类和库。在工程实践中，不要重复造轮子；而在学习研究中，鼓励重复造轮子。

**建议144：谨慎在OO与GP之间选择**

面向对象(OO)和泛型编程(GP)是C++提供给程序员的两种矛盾的思考模式。OO是我吗难以割舍的设计原则，世界是对象的，我们面向对象分析、设计、编程；而泛型编程则关注于产生通用的软件组件，让这些组件在不同的应用场合都能很容易的重用。

**建议145：让内存管理理念与时俱进**

学习STL allocator，更新内存管理理念。

**建议146：从大师的代码中学习编程思想与技艺**

阅读代码需要方法：刚开始不要纠结于代码的细节，将关注的重点放在代码的高层结构上，理解代码的构建过程；之后，再有重点的深入研究，理解对象的构造，明晰算法的步骤，尝试着深入理解其中的繁杂细节。

**建议147：遵循自然而然的C++风格**

**建议148：了解C++语言的设计目标与原则**

**建议149：明确选择C++的理由**