**一、让自己习惯C++**

**TIPS2:尽量以const,enum,inline 替换#define**

2.1对于单纯常量,用const替换#defines

2.2对于形似函数的宏,用inline替换#defines

**TIPS3:尽可能使用const**

3.1编译器强制执行bitwise constness

3.2当const和non-const成员函数有着等价的实现时,令non-const版本调用const版本可以避免重复

**TIPS4:确保对象被使用前已经初始化**

4.1为内置对象手动初始化,因为C++不保证初始化它们

4.2构造函数用成员初值列而不要在构造函数内部用赋值进行初始化,初值列的成员变量,排列次序应该与之在class中的声明次序相同

4.3为了免除“跨编译单元初始化次序”问题,用local static变量替换non-local static变量！

**TIPS5:C++默默编写并调用的函数**

5.1编译器可以自行为class创造default构造函数,copy构造函数,copy assignment操作符以及析构函数

**二、构造/析构/赋值运算**

**TIPS6:若不想使用编译器自动生成的函数,就该明确拒绝**

6.1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,使用如uncopyable这样的based classe也是一种做法

**TIPS7:为多态基类声明virtual析构函数**

7.1带多态性质的的基类应该声明一个virtual析构函数,如果class带有任何virtual函数,它就应该有一个virtual析构函数

7.2类的设计目的如果不是作为基类使用,或不是为了具备多态性,就不应该声明virtual析构函数

**TIPS8:别让异常逃离析构函数**

8.1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,然后吞下她们或结束程序

8.2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操作

**TIPS9:绝对不要在构造和析构过程中调用虚函数**

9.1在构造和析构期间不要调用虚函数,因为这类调用从不下降至子类

**TIPS10:令operator=返回一个reference to \*this**

10.1令赋值操作符返回一个\*this的引用

**TIPS11:在拷贝操作符operator=中处理自我赋值**

实现方法(1)证同测试

Widget & widget::operator=(const Widget& rhs)

{

If (this == &rhs) return \*this (证同测试)

…

}

实现方法(2)让operator=具备异常安全性

Widget& widget::operator=(const Widget& rhs)

{

Bitmap\* pOrig = pb; (先记住原pb,再让pb指向\*pb的一个副本,再删除原pb)

pb = new Bitmap(\*rhs.pb);

delete pOrig;

return \*this;

}

实现方法(3)利用swap

Widget& Widget::operator=(const Widget& rhs)

{

Widget temp(rhs); (为rhs创建一个副本temp,在operator操作符结束后自动析构)

swap(temp); (将\*this的数据和副本的数据交换)

return \*this;

}

如果没有适当处理自我赋值，可能会出现函数内的\*this和rhs指向同一个对象,这样如果delete就不止销毁当前对象的bitmap,也会销毁rhs的bitmap,而原本对象不该被自我赋值动作而改变！

11.1确保当前对象自我赋值时,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、语句顺序以及copy-and-swap;

11.2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！

**TIPS12:复制对象时勿忘其每一个成分**

设计良好的OO-system会将对象内部封装起来，只留两个函数负责对象拷贝（即拷贝构造函数和拷贝赋值操作符）！

注:任何时候，只要为子类撰写拷贝构造函数，都应该注意赋值其基类的成员，如果是private类型则应该调用相应的基类函数进行复制！

12.1拷贝构造函数应当确保对象内的所有成员变量以及所有的基类成分！

12.2勿尝试以某个拷贝构造函数来实现另一个拷贝构造函数，应当将共同机制放到第三个函数中，并由两个拷贝构造函数共同调用

**三、资源管理**

**TIPS13:以对象管理资源**

13.1为防止资源泄露,应当使用RAII对象(资源获取的时机就是初始化时机,Resource Acquisition Is Initialization;RAII).

13.2常用的RAII 类分别为shared\_ptr和auto\_ptr,前者通常为较佳选择，如果选择后者，复制动作会使得其指向null(现auto\_ptr已经被替换,auto\_ptr是C++11前的指针。此后为unique\_ptr)

auto\_ptr的问题：

复制和赋值会改变资源的所有权，不符合人的直觉；

在 STL 容器中无法使用auto\_ptr ，因为容器内的元素必需支持可复制（copy constructable）和可赋值（assignable）。

unique\_ptr特性

拥有它所指向的对象

无法进行复制构造，也无法进行复制赋值操作

保存指向某个对象的指针，当它本身离开作用域时会自动释放它指向的对象。

unique\_ptr可以：

为动态申请的内存提供异常安全

将动态申请内存的所有权传递给某个函数

从某个函数返回动态申请内存的所有权

在容器中保存指针

**TIPS14:在资源管理类中警惕拷贝行为**

14.1复制RAII对象必须一并复制其所管理的资源，所以资源的拷贝行为决定RAII对象的拷贝行为

14.2普通而常见的RAII类的拷贝行为是：抑制拷贝、施行引用计数器(reference counting)。

referecen count 的核心思维：使用一个计数器来标识当前指针指向的对象被多少类的对象所使用（即记录指针指向对象被引用的次数）

构造函数中创建类的新对象时，初始化引用计数为1；

拷贝构造函数复制指针，并使相应的引用计数增加1；

赋值操作减少左操作数所值对象的引用计数，增加右操作数所指对象的引用计数；

析构函数使引用计数减少1，并且当引用计数为1时，释放指针说指向的对象

**TIPS15:在资源管理类中提供对原始资源的访问**

15.1APIs往往要求访问原始资源，所以每一个RAII类应该提供一个“取得其所管理之资源的方法”（shared\_ptr提供了一个get方法，用来执行显式转换，即其会返回智能指针内部的原始指针）

15.2对原始资源的访问可能经由显示转换或者隐式转换，一般而言显示比较安全，但隐式对客户比较方便。

注：C++可通过operator进行隐式转换，形如：

operator 类型T ()

即将对象隐式转换为函数对象！

Class Font{

Public:

…

**Operator FontHandle() const{ （本行定义了隐式转换函数）**

return f;

}

}；

Void changeFontSize(FontHandle f, int newsize);

Font f(getFont());

Int newFontsize;

…

…

changeFontSize(f,newFontsize) 🡪这里入口参数为FontHandle类型，而传入的是Font类型，故进行了隐式类型转换，进而调用了隐式转换函数

**TIPS16:成对使用new和delete的时候要采取相同的形式**

问题的关键在于,即将被删除的指针，所指的是单一对象或是对象数组？单一对象的内存布局一般不同数组的内存布局，更明确地说，数组所用内存还包括数组大小的记录，以便于delete知道应该调用多少次析构函数。单一的内存对象则没有这个记录！

单一对象：| Object |

对象数组：| n | Object | Object | Object |……

当我们对着一个指针进行delete时，唯一可以让delete知道内存中是否存在一个“数组大小记录“的方法是，由我们告诉它！比如，在delete中加上[ ]，delete就知道指向一个数组！

delete stringPtr1 -----删除单一对象

delete [ ] stringPtr2 ----删除一个由对象构成的数组

结论：如果new时使用了[]，那么delete时也需要[]，如果new时没有[]，则delete时一定不要有[]。

**TIPS17:以独立语句将newed对象置入智能指针**

以独立语句将newed对象存储于智能指针内，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。

比如：processWidget(std1::tr1::shared\_ptr<Widget>(new Widget),priority());

这种方式可能存在资源泄露，因为在调用前，编译器会核实函数的各个实参，如果执行这样的操作序列：

1. 执行new Widget
2. 执行priority
3. 调用shared\_ptr的构造函数

如果步骤2出现异常，则会导致new Widget返回的指针遗失，因为其尚未置入shared\_ptr内，解决方法很简单，即写成下述形式：

std1::tr1::shared\_ptr<Widget> pw(new Widget)；

processWidget(pw,priority());

**四、设计与声明**

**TIPS18:让接口容易被正确使用，不容易被误用**

18.1好的接口容易被正确使用，不容易被误用，故应当在所有接口中努力达成这一性质！

18.2促进正确使用的方法包括接口的一致性，以及与内置类型行为的兼容

18.3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

18.4tr1::shared\_ptr支持定制删除器，可以防范dll问题，可以被用来自动解除互斥锁

18.5好的代码可以防止无效代码通过编译！

**TIPS19:设计class犹如设计type**

设计高效的class应当直面的问题：

19.1新type的对象如何被创建与销毁？

19.2对象的初始化和赋值改由怎样的差别？

19.3新type对象如果被值传递，意味着什么？

19.4什么是type的合法值？

19.5新type需要配合某个继承图系吗？

19.6新type需要怎样的转换？

19.7什么样的操作符和函数对此新type是合理的？

19.8什么样的标准函数应该驳回？

19.9该取谁作为新type的成员？

19.10什么是新type的未声明接口？

19.11新type有多么地一般化？

**TIPS20:宁以“传常量引用”的方式替代“传值”的方式**

缺省的情况下C++以传值的方式传递对象至函数，除非我们另外指定！否则函数参数都是以实际实参的副本为初值，而调用端所获得的亦是函数返回值的一个副本，这些副本系由对象的拷贝构造函数产出，这可能使得传值成为昂贵（耗时）的操作！

一般而言，可以合理假设“传值并不昂贵”的唯一对象就是内置类型和STL的迭代器和函数对象，至于其他任何东西都可遵循“以常量引用传递”的方式！

**TIPS21:必须返回对象时勿返回其引用**

勿返回一个指针或引用指向局部栈对象，或返回引用指向堆分配的对象，或返回指针指向一个局部静态对象而有可能同时需要多个这样的对象。

**TIPS22:将成员变量声明为private！**

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！

如果成员变量被声明于public中，而最终将其取消，可能造成所有客户代码破坏！同样，如果声明于protected中，一旦发生改变，则其所有子类都会被破坏！

22.1切记将成员变量声明为私有，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，并提供类的作者以充分弹性

22.2protected并不比public更具有封装性（封装性与当其内容改变时可能造成的代码破坏量成反比---public可被任意实体访问，可造成破坏性最大，protected可以被本类和子类成员函数访问到，因此破坏性也很大，而private只允许本类成员函数访问，所以破坏性小）

**TIPS23:宁以non-member、non-friend替换成员函数**

比如一个类

class WebBrowser{

public:

void clearCache();

void clearHistroy();

void removeCookies();

};

为了执行上述一些列动作，WebBrowser类也提供一个函数：

class WebBrowser{

public:

…

…

Void clearEverything();

};

在这里，clearEverything是成员函数。

但是，也可以以非成员函数的形式实现：

Void clearEverything(WebBrowser& wb)

{

wb.clearCacher();

wb.clearHistroy();

wb.removeCookies();

}

后者比前者更好，因为后者封装性比前者好，因为前者函数体内可以调用私有成员。且后者弹性比前者高！

C++标准程序库的组织方式，将使用某个类的不同便利函数声明于不同的头文件（但命名空间相同），每个头文件声明std的某些机能。

类似的组织方式如下：

头文件webrowser.h内

namespace WebBrowserStuff{

Class webBroser{…};

… //non-member函数

…

}

头文件webbrowserbookmarks.h内

namespace WebBrowserStuff{

… //与书签相关的便利函数

…

}

头文件webbrowsercookies.h内

namespace WebBrowserStuff{

… //与cookie相关的便利函数

…

}

**TIPS24:若所有参数皆需类型转换，应采用non-member函数**

24.1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一个non-member！

注：只有参数被列于参数列内，这个参数才是隐式类型转换的合格参与者！

**TIPS25:考虑写出一个不抛出异常的swap函数**

如果swap的缺省代码对实现class或class template提供可接受的效率，则开发者不需做任何额外的事情。如果swap缺省效率不足，试着做以下事情：

1. 提供一个public swap函数，让其高效地置换某个类型的两个对象值
2. 在class或template所在的命名空间提供一个non-member swap，并令其调用上述swap成员函数
3. 如果正编写一个class，为之特化std::swap，并令其调用swap成员函数
4. 如果调用swap，务必确定包含一个using声明式，以便让std::swap在函数内曝光可见，然后不加任何修饰符，赤裸裸地调用swap
5. 为“用户定义类型”进行std template全特化是好的，但请勿在std内加入某些对std而言全新地东西

**五、实现**

**TIPS26:尽可能延后变量定义式的出现时间**

不应该只延后变量的定义，直到非得使用该变量的前一刻为止，甚至应该尝试延后这份定义直到能够给它初值实参为止。这样能够避免构造和析构非必要对象，还可以避免无意义的default构造行为。更深一层说，以“更明显意义之初值”将变量初始化，还可以附带说明变量的目的。

对于循环而言，如果变量只在循环内使用，有两种定义方法

1. 定义于循环外，每次循环迭代时赋值
2. 定义于循环内

如果类的一个赋值成本低于一组构造+析构成本，A方法大体而言比较高效，尤其当n很大的时候。否则B做法更好！

**TIPS27:尽量少做转型操作**

C++的四类新式转型

const\_cast<T>(expression) 用于将对象的常量性移除

dynamic\_cast<T>(expression) 用于执行“安全向下转型”

reinterpret\_cast<T>(expression) 用于执行低级转型，实际动作取决于编译器，可能无法移植。比如将一个int的指针转型为int

static\_cast<T>(expression) 用于强迫隐式转换，比如将非常量对象转型为常量对象，或者将int转为double等

旧式转型

(T)expression

T(expression)

均是将对象转型为T类型。

旧式转型依旧合法，但是新式比较受欢迎！而唯一使用旧式转型的时机是，当需要调用一个explicit构造函数将一个对象传递给另一个对象时

在C++中，单一对象可能拥有一个以上的地址，实际上一旦使用多重继承，这就一直在发生着。即使在单一继承中也可能发生。

之所以需要使用dynamic\_cast，通常是因为想在一个被认定为是子类的对象上执行子类的操作函数，但却只有一个指向基类的指针，或者引用。有两个一般性做法可以避免这个问题：

1. 使用容器并在其中存储指向子类对象的指针（通常是智能指针）
2. 在基类内提供虚函数做想对各个子类做的事

27.1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

27.2如果转型是必要的，试着将其隐藏于某个函数背后，客户随后可以调用该函数而无需将转型放入自己的代码内

27.3宁可使用C++的新式转型，不要使用旧式转型，因为前者很容易被辨识出来

**TIPS28:避免返回handles指向对象的内部成分**

28.1避免返回handles(包括引用，指针，迭代器)指向对象内部，遵循这个条约可以增加封装性，帮助const成员函数的行为像一个const。

**TIPS29:为“异常安全”而努力是值得的**

当有异常抛出时，带有异常安全性的函数会：

1. 不泄露任何资源
2. 不允许数据破坏

异常安全函数提供三个保证之一

基本承诺：如果异常被抛出，程序内的任何事物仍然保持在有效状态下

强烈保证：如果异常被抛出，程序状态不会改变

不抛掷保证：承诺绝不抛出异常

异常安全码比必须提供上述三种保障之一，如果不这么做就不具备异常安全性。

若能够提供不抛掷保证是最好的，但大部分情况下抉择往往落在基本保证和强烈保证之间。

Copy-and-swap的策略很容促成强烈保证！即为打算修改的对象(原件)做一个副本，然后再副本身上做出一切必要修改，若没有任何修改动作抛出异常，原对象仍保持未改变的状态。待所有改变都成功后，再将修改过的那个副本和原对象再一个不抛出异常的操作中置换（swap）

29.1强烈保证往往可以用copy-and-swap的方式实现，但强烈保证并非对所有函数都可实现或具备现实意义（如果函数只操作局部状态，则容易施以强烈保证，但函数对非局部数据有连带影响时，强烈保证便很困难）

29.2函数提供的异常安全保证通常最高只等于其所调用之各个函数的“异常安全保证”中的最弱者。

**TIPS30:透彻了解inline**

Inline的本质是将“对此函数的每一个调用”都以函数本体替换之，如果函数本体很小，那么编译器对函数本体所产出的码可能比函数调用所产生的码更小。

Inline的隐喻方式是将函数定义于类的定义式内（成员函数），friend函数也可以被定义于内的内部，如果这样，它们也将被隐喻声明为inline。

而明确的声明方式则式再定义式前加上inline关键字。

Inline函数通常一定被置于头文件内，因为大多构建环境都是在编译过程中进行inlining，而为了将一个调用替换为本体，编译器必须知道那个函数长什么样。Inlinging在大多C++程序中都是编译期行为。、

大部分编译器拒绝将过于复杂的函数(比如带有循环和递归)inlining，而对所有的虚函数的调用也会使inlining落空。事实上构造函数和析构函数都不适合inlining。

所以，一开始先不要将任何函数声明为inline，或至少将inline局限于那些“一定成为inline“或者”十分平淡无奇“的函数身上。

30.1将大多数inlining限制在小型、被频繁调用过的函数身上，可以使得日后的调试过程和二进制升级更容易，也可以使得潜在的代码膨胀问题最小化，使得程序的速度提升机会最大化。

30.2不要因为function template出现在头文件，就将它们声明为inline

**TIPS31:将文件间的编译依存关系降到最低**

31.1把接口从实现中分离！

31.2采用pimpl idiom设计，即把一个类分成两个类，一个负责提供接口，一个负责实现接口，然后在提供接口的类里包含一个pImp指针（pImp即pointer to implementation，实现指针），指向负责实现的类！这样的设计下，就把接口从实现中分离了。

31.3上述方法实质是以“声明的依存性”替换“定义的依存性”，正式编译器最小化依存性的本质：现实中让头文件尽可能自我满足，万一做不到则让它与其他文件内的声明式（而非定义式）相依

31.4如果使用object的引用或者指针可以完成任务，就不要直接使用object

31.5如果能够，尽量以类的声明式替换类的定义式，如果将提供类的定义式的义务（通过#include完成）从函数声明之头文件转移到内含函数调用之客户文件，即可将“并非真正之必要之类型定义“与客户端之间的编译依存性去掉

31.6使用pImp的类往往称作handle class，实现handle class也可以使用抽象基类的方法，这种方法下，基类也称作interface class。

31.7支持编译依存性最小化的方法的一般构想是：相依于声明式而非定义式，基于此构想的两个手段式Handle classes 和 Interface classes

31.8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！

**六、继承与面向对象设计**

**TIPS32:确定你的public继承塑模出“is-a”关系**

32.1 is-a 表示 public继承意味着“是一种”(is-a)”的关系.比如让class D以public方式继承了class B，便是告诉C++编译器，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。即B比D表现出更一般化的概念，而D比B更为特殊化。B可以用上场的地方，D一样可以用上场，反之并不成立！C++对于public继承严格遵循这样的准则！！！

32.2public继承是一种is-a，表示适用于基类身上的每一件事一定也适用于派生类身上，因为每一个派生类对象也是一个基类对象

**TIPS33:避免遮掩继承而来的名称**

33.1内层作用域的名称会遮掩外围作用域的名称。

33.2编译器查找的方式是，先从local作用域内，查找是否有名称x的声明式，如果没有，则再查找其外围，最后查找到global！对于类对象而言，会先从local作用域，然后查找子类覆盖的作用域，再到基类覆盖的作用域，最后到global。

33.3子类的名称会遮掩基类内的名称，在public继承下无人希望如此

33.4为了让遮掩的名称重见天日，可以使用using声明式(比如在子类中using 基类名::成员，这样也会使得基类的成员在子类中是可见的，而不会被覆盖)或转交函数。

**TIPS34:区分接口继承和实现继承**

34.1接口继承和实现继承不同，在public继承之下，子类总是继承基类的接口

34.2纯虚函数只具体指定接口继承

34.3非纯虚函数具体指定接口继承和缺省的实现继承

34.4非虚函数具体指定接口继承以及强制性实现继承

**TIPS35:考虑虚函数以外的其他选择**

当为解决问题而寻找某个设计方案时，可以考虑虚函数的替代方案。

包括：

1. 使用NVI手法，以共有的非虚函数包裹访问性较低(protect和private)的虚函数，此为Template Method设计模式的一种特殊形式
2. 将虚函数替换为函数指针成员变量，此为strategy设计模式的某种形式
3. 以tr1::function成员变量代替虚函数
4. 将继承体系内的虚函数替换为另一个继承体系内的虚函数，这是strategy设计模式的传统实现手法

**TIPS36:绝不重定义继承来的非虚函数**

非虚函数是静态绑定(前期绑定,early binding)的，即使声明一个指向派生类的基类指针，其调用的非虚函数也是也永远是基类的版本。

而虚函数是动态绑定(后期绑定,late binding)的，所以无论声明的是基类还是派生类指针，其调用的虚函数只与指针所指向的对象有关。

**TIPS37:绝不重新定义继承而来的缺省参数值**

所谓静态类型,即程序中被声明时所采取的类型。

所谓动态类型即”目前所指对象的类型”

虚函数系动态绑定而来，意为调用一个虚函数时，究竟调用哪一份函数的实现代码，取决于发出调用的那个对象的动态类型！

但是,缺省参数值是静态绑定，意思是可能会在“调用一个定义于派生类内的虚函数”的同时，却使用基类为它指定的缺省参数值。

37.1请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---唯一应当覆写的东西，是动态绑定！

**TIPS38:通过复合塑模出has-a或”根据某物实现出”**

复合是类型间的一种关系,当某种类型的对象内涵它种类型的对象，便是这种关系。当复合发生于应用域(比如表示人、物类似的对象)的对象间，表现出“has a”(有)的关系，而发生于实现域(比如互斥器、缓冲区、查找树等的对象)内则是表现“根据某物实现出”的关系。

系统自带的set通常采用平衡查找树实现。

38.1复合的意义和public继承完全不同

**TIPS39:明智而谨慎地使用private继承**

private继承意味着”根据某物实现出”，是一种纯粹地实现技术，而与基类没有任何观念上的联系。所以将基类内部所有成员都转变为了private：因为这些都是实现的枝节而已。如果D继承B，意思是D对象根据B对象实现而得，再没有其他意涵了。故private继承在软件设计上没有意义，其意义仅停留于软件实现层面。

private继承的功能与复合类似，故尽可能使用复合而少使用private继承，除非必要(所谓必要即当protected成员和/或虚函数牵扯进来时)。

39.1private继承意味着“根据某物实现出”。它通常比复合的优先度低，但是其当派生类要访问protected base class的成员，或者需要重新定义继承来的虚函数时，这么设计是合理的。

**TIPS40:明智而谨慎地使用多重继承**

40.1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(比如出现菱形继承时)

40.2virtual继承会增加大小、速度、初始化(赋值)复杂度等等成本，如果virtual base classes不带任何数据，将会是最实用的情况

40.3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”的两相组合。

**七、模板与泛型编程**

**TIPS41:了解隐式接口和编译器多态**

通常显式接口由函数的签名式(函数名、参数类型、返回类型)构成，隐式接口则不同，其并不基于签名式，而是基于有效表达式。

例如以下表达式:

template<typename T>

void doProcessing(T& w)

{

If(w.size()>10 && w !=someNastyWidget )

…

}

T的隐式接口看起来好像要有一些约束：

必须提供一个名为size的成员函数，该函数返回一个整数值

必须支持一个operator!=函数，用来比较两个T对象，这里假设someNastyWidget的类型为T

但由于操作符重载带来的可能性，这两个约束其实都不必要满足。尽管T必须支持size成员函数，但其可以由基类继承而来。这个成员函数不需要返回一个整数值，甚至无需返回一个数值类型，就此而言，其甚至不需要返回一个定义由operator>的类型，它唯一需要做的仅仅是返回一个类型为X的对象，而对象加上一个int必须能够调用一个operator>.这个operator>不需要非得取得一个类型为X的参数不可，因为其也可以取得类型Y的参数，只要存在一个隐式转换能把类型Y的对象转换为类型X的对象！

同样道理，T不需要支持operator!=。因为以下这样也是可以的:operator1=接受一个类型为x的对象和一个类型为y的对象，T可以被转换为x而someNastyWidget可以被转换为y，这样就可以有效调用operator!=.

41.1类和模板都支持接口与多态

41.2对类而言接口式显示的(explicit),以函数签名为中心，多态则是通过虚函数发生于运行期。

41.3对模板参数而言，接口式隐式的，基于有效表达式。多态则通过模板的具现化和函数重载解析发生于编译期。

41.4运行期多态和编译期多态的区别是，“哪一个虚函数该被绑定”和“哪一个重载函数该被调用”

**TIPS42:了解typename的双重含义**

template<class T>和template<typename T>在声明式中完全相同。

但有时候C++并不总是把class和typename等价，这时候一定要使用typename！

Template内部出现的名称如果相依于某个template参数，称之为从属名称。

如果从属名称在class内呈现嵌套状，则是为嵌套从属名称(如C::const\_iterator,C是template<typename C>中的template参数)。

因为在知道C是什么之前，没有任何办法知道C::const\_iterator是否为一个类型，因此C++解析器在遇到嵌套从属名时，便假设其不是一个类型，除非手动告诉编译器它是！一般规则很简单:任何时候想要在template中指涉一个嵌套从属类型名，就必须在紧邻它的前一个位置放上关键字typename。

typename必须作为嵌套从属名的前缀词 这一规则的例外是，typename不可以出现在base classes list内的嵌套从属名称前，也不可以出现在成员初值列中作为基类的修饰符(很怪异，但是勉强接受把！)。

42.1声明template参数时，前缀关键字class和typename可以互换

42.2请使用关键字typename标识嵌套从属名称，但也有例外。

**TIPS43:学习处理模板化基类内的名称**

由于基类模板可能被特化，而那个特化版本可能不提供和一般模板相同的接口，所以编译器往往拒绝在模板化基类中寻找继承而来的名称！

比如下例:

template<typename Company>

class LoggingMsgSender:public MsgSender<Company>{

public:

…

Void sendClearMsg(const MsgInfo& info)

{

…

sendClear(info)

…

}

};

这里,sendClear就是继承自基类模板的函数，但如果基类模板被特化，这个接口可能不存在！某种意义而言，当从OOC跨越进了Template C++继承就不像以前那般畅通无阻了！

为了解决这一问题，可以采用三种方式

1:在基类函数调用动作前加上this->

This->sendClear(info)   告诉编译器，假设sendClear将被继承

2:使用using声明式

Using MsgSender<Company>::sendClear 告诉编译器，假设sendClear位于基类内

3:明确指出被调用的函数位于基类内

MsgSender<Company>::sendClear(info) 假设sendClear将被继承下来(但这往往是糟糕的做法，如果调用的是虚函数，上述明确修饰就会关闭虚函数的绑定行为)

为了省点力气…emmm还是第一种方式吧…

**TIPS44:将与参数无关的代码抽离templates**

44.1templates生成多个类和多个函数，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

44.2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类成员变量替换template参数

44.3因参数类型而造成的代码膨胀，往往可降低，做法是让带有完全相同的二进制表述的具现类型共享实现代码。

**TIPS45:运用成员函数模板接受所有的兼容类型**

所谓智能指针，是指行为像指针的对象，并提供指针所没有的机能。

就原理而言，一个template可以被无限量具现化，以致生成无限量函数。故，通过函数模板也可以写出模板构造函数，如下：

template<typename T>

class SmartPtr{

public:

template<typename U>

SmartPtr(const SmartPtr<U>& other);

……

};

以上代码的含义是，对任何类型T和类型U，这里可以根据SmartPtr<U>生成一个SmartPtr<T>---因为SmartPtr<T>有个构造函数接受一个SmartPtr<U>参数。这一类构造函数根据对象u创建对象t，而u和v的类型是同一个template的不同具现化，有时我们称之为泛化拷贝构造函数。

这里的构造函数之所以没有被声明为explicit，那是蓄意的，因为原始指针类型之间的转换(比如从派生类指针转换为基类指针)是隐式转换，无需明白写出转型动作。

但有时，泛化拷贝构造函数提供的东西比我们需要的的多。比如我们不希望根据一个SmartPtr<double>去创建一个SmartPtr<int>，因为现实中并没有将int\*转换为double\*的隐式转换。那么我们必须为之加上限制！这点，可以遵循智能指针，提供一个get成员函数，返回智能指针对象所持有的那个原始指针的副本，那么就可以在构造模板中实现代码中过的约束转换行为，使之符合期望。

如下:

template<typename T>

class SmartPtr{

public:

template<typename U>

SmartPtr(const SmartPtr<U>& other)

: heldPtr(other.get()){…}

T\* get() const {return heldPtr}

…

private:

T \*heldPtr;

};

通过使用成员初值列来初始化SmartPtr<T>内部类型为T\*的成员变量，并以类型U\*的指针作为初值，这个行为只有当”存在某个隐式转换可将一个U\*类型的指针转换为一个T\*类型的指针”时才能通过编译，而这正是所需要的。最终效益是SmartPtr<T>有了一个泛化拷贝构造函数，这个函数只有在其所获得实参适当(兼容)时才可通过编译。

45.1请使用成员模板函数生成可接受所有兼容那类型的函数

45.2如果声明成员模板函数用于泛化拷贝构造或者泛化拷贝操作符，还是需要声明正常的拷贝构造函数和正常的拷贝操作符。

**TIPS46:需要类型转换时请为模板定义非成员函数**

当编写一个模板类，而为它提供之“与此模板相关的”函数支持“所有参数之隐式转换”时，请为那些函数定义为“模板类”内部的友元函数。

**TIPS47:请使用traits classes表现类型信息**

C++的5种迭代器：

“最弱的”：Input、output迭代器，智能向前移动，只能读/写。对应istream\_iterators、ostream\_iterators.

“中间威力”:forward迭代器,可以提供上述两种迭代器所做的任何事，

“次级威力”:Bidirectional迭代器，可以向前，也可以向后移动，STL的list迭代器就属于这一分类，set、multiset、map、multimap的迭代器也是如此

“最强大”:random access迭代器，其相比之前的，更强的地方在于可执行“迭代器算术”，即其可以在常量时间内移动任意距离，而不像之前的只能一步步++或者--。vector、deque、string的迭代器均时此分类

对于这5种分类，C++提供专门的结构加以确认：

Struct input\_iterator\_tag{}

Struct output\_iterator\_tag{}

Struct forward\_iterator\_tag:public input\_iterator\_tag{}

Struct bidrectional\_iterator\_tag:public forward\_iterator\_tag{}

Struct random\_access\_iterator\_tag:public bidirectional\_iterator\_tag{}

实际上，这些迭代器是有效is-a的继承关系，所以forward迭代器都是input迭代器，以此类推。

Traits并非C++的关键字或者一个预先定义好的结构体，而时一种技术，程序员所共同遵循的协议。类型的traits必须位于类型自身之外，因为traits必须也能够适用于内置类型，而原始指针内无法嵌套信息。标准技术时把它放进一个模板以及其一个或多个的特化版本中。这样的模板在程序库内有若干个，其中针对迭代器者被命名为iterator\_traits.

template<typename IterT>

struct iterator\_traits;

尽管其是个结构体，习惯上将之称作traits class。

其运作方式时是，针对每一个类型IterT,在结构内声明某个typedef名为iterator\_category,用来确认IterT的迭代器分类。

如下

template<typename IterT>

struct iterator\_traits{

typedef typename IterT::iterator\_category iterator\_category.

};

这对用户自定义类型可行，但对于指针行不通，因此其特别为指针提供了一个偏特化版本。

现在，已经知道设计并实现一个traits class的步骤：

1. 确认若干希望未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
2. 为该信息选择一个名称(如iterator\_category)
3. 提供一个模板和一组特化版本，内涵所希望支持类型的相关信息。

为了在编译期完成类型判断，可以通过重载的方式即：

1. 建立一组重载函数(身份像劳工)或函数模板，彼此间的差异只在于各自的traits函数，令每个函数实现码与其接受之traits信息相应和。
2. 建立一个控制函数(身份像工头)或者函数模板，调用上述那些“劳工”并传递traits class所需要的信息

47.1Traits Class使得类型相关信息在编译期可用，它们以模板和模板特化实现

47.2整合重载技术后，traits class有可能在编译器对类型执行if…else测试

**TIPS48:认识模板元编程**

48.1模板元编程可以将工作由运行期移往编译期，因而得以实现早期侦测和更高的执行效率.

48.2模板元编程(TMP)可以被用来生成“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。

**八、定制new和delete**

**TIPS49:了解new-handler的行为**

当operator new无法满足某一内存分配需求时，它会抛出异常。以前它会返回一个null指针，某些旧式编译器也会这么做。

当operator new抛出异常以反映一个未获得满足的内存需求之前，它会调用一个客户指定的错误处理函数，即所谓的new-handler，为了指定这个用以处理内存不足的函数，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();

}

注:typedef函数指针的格式

typedef  返回类型A (\*新类型名B)(参数表)

顶以后，B就表示一个返回类型为A,且参数表与定义中相同的函数指针。

可以这样使用new\_handler:

Void outofMemory()

{

….

}

Int main()

{

Std::set\_new\_handler(outofMemory);

Int \*XXX = new int[100000000];

}

就上述例子而言，如果operator new无法为100000000个整数分配足够的空间，outofMemory就会被调用。

一个设计良好的new-handler必须能做到以下事情:

1. 让更多内存可以被使用
2. 安装另一个new-handler，如果目前这个new-handler无法区得更多可用内存，需要知道另外哪个具有此能力，果真如此，就可以安装那个新的new-handler代替自己
3. 卸除new-handler(将null传递给set\_new\_handler)
4. 抛出bad\_alloc
5. 不返回，通常调用abort或者exit

**TIPS50:了解new和delete的合理替换时机**

替换编译器提供的new和delete的原因：

1. 用来检测运行上的错误

如果将new所得的内存delete掉却不幸失败，会导致不确定行为。如果operator new有一连串动态分配所得地址，而operator delete将地址从中移走，则容易检测出上述错误用法。此外各种各样的编程错误可能导致数据overruns(写入点在分配点之后)或者underruns(写入点在分配点之前)。如果自行定义一个operator news，便可超额分配内存，以额外空间放置特定的byte patterns(签名)，operator delete便得以检测上述签名是否原封不动，如不是则表示在分配区的某个生命时间点发生了overruns或者underruns，这时候operator delete可以志记那个事实与指针。

1. 强化效能

编译器所带的operator new和operator delete主要用于一般目的，其必须考虑破碎问题，这最终会导致程序无法满足大区块的内存要求，即使彼时有总量足够但分散为许多小区块的自由内存。一般来说，定制版的delete和new的性能会胜过缺省的版本！

1. 收集使用上的统计数据

为收集软件如何使用其动态内存，比如分配区块的大小，寿命分布，FIFO次序还是LIFO次序还是随机次序来分配、归还，运用形态是否随时间改变，任何时刻所使用的最大动态分配量是多少，等等。自行定义operator new和operator delete使得我们可以轻松收集这些信息。

1. 增加内存分配归还的速度
2. 降低缺省内存管理器带来的额外空间开销
3. 弥补缺省分配器中的非最佳齐位
4. 将相关对象成簇集中
5. 获得非传统行为

在这个主题中，齐位(alignment)意义重大(比如double类型指针的地址必须是8的倍数，如果不是，可能会导致运行期硬件异常。而就算可以容忍其不为8的倍数，比如在intel x86平台下，但如果对齐为8倍数，会使得访问速度快很多！！)，因为C++要求所有operator news返回的指针有适当的对齐(取决于数据类型)，malloc就是在这样的条件下工作，所以命令operator new返回一个得自malloc的指针是安全的。

**TIPS51:编写new和delete时需固守常规**

Operator new的返回值十分单纯，如果有能力提供客户所需求的内存，则返回一个指针指向那个内存，如果没有那个能力，则遵循TIPS49的描述，并抛出一个bad\_alloc异常。

new表达式即new operaotr，比如new + 类名！注意与operator new区分！

Operator new的作用相当于一个malloc，即分配内存并返回指针。

New表达式包括了operator new！

delete表达式同理！

一般而言 new表达式包括两个步骤(即new operator)：

1. 调用operator new分配内存
2. 调用类的构造函数

同理，delete表达式也包括两个步骤(即delete operator)：

1. 调用类的析构函数
2. 调用operator delete归还内存

撰写operator delete时要奉行“删除null指针永远安全!”比如下面这版本的非成员delete

Void operator delete(void \* rawMemory) throw()

{

If(rawMemory == 0 ) return;

…

归还内存

…

}

这个函数的成员版本也很简单，只需要多加一个动作检测删除数量，万一类专属的new将大小有误的内存分配行为转交：：operator new指向，那么也必须保证将大小有误的delete行为转交：：operaotr delete执行。

Void Base :: operator delete(void \*rawMemory,size\_t size) throw()

{

If(rawMemory == 0 ) return;

If(size != sizeof(Base)){

::operator delete(rawMemory);

Return;

}

…

归还内存

…

}

51.1operator new内部应该含有一个无穷循环，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的申请。Class专属版本的则应该处理“比正确大小更大(错误的)的申请“

51.2operator 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(错误)“申请。

**TIPS52:写placement new也要写placement delete**

如果operator new接受的参数除了一定会有的size\_t之外还有其他，这就是一个placement new(定位new)。

C++标准的 new 表达式能完成大部分的需求, 但不是全部, 比如: 如何在已有的内存空间上创建对象, 标准 new 表达式做不了, C++也不支持直接在raw内存上调用对象的构造函数, 于是placement new就产生了, 取名placement, 也说明了它的历史来源, 即在原地构建一个新对象。

众多定位new版本中特别有用的一个是“接受一个指针指向对象被构造之处“，其长相如下：

Void \*operator new(std::size\_t size,void \* pMemory) throw();

这个版本的new已经纳入C++标准程序库(#include <new>)，其用途之一是在vector未使用的空间上创建对象。也是最早的定位new版本。

考虑以下代码，其将动态创建一个Widget时将相关分配信息志记于cerr。

Widget \*pw = new (std::cerr) Widget；

如果内存分配成功，而Widget构造函数抛出异常，运行期系统有责任取消operator new的分配并遵循旧规。然而运行期系统无法知道真正被调用的那个operator new如何运作，所以上述做法行不通。取而代之的是，运行期系统寻找“参数个数和类型都与operaotr new相同“的某个operator delete。如果找到，那就是它的调用对象。既然这里的new是定位new，接受一个额外的实参，所以对应的operator delete要与之对应，这种delete便是定位delete。这样改变之后，对应的placement delete被调用，保证不泄露任何内存！

然而如果上述代码没有抛出异常，而客户代码中有个delete：

Delete pw；

这里会调用正常形式的operator delete而不是placement版本的，因为定位版本只有在“伴随定位new调用而触发的构造函数”出现异常时才会调用。

附带一提，由于成员函数的名称会掩盖其外围作用域中相同的名称，可能会导致class无法调用global版本的new或者base class的new。解决方法是：建立一个base class，内涵所有正常形式的new和delete，凡是想以自定义形式扩充标准形式的客户，可利用继承机制以及using声明式来取得标准形式！

52.1当声明定位new和定位delete时，请确定不要无意识地遮掩了其正常版本！

**九、杂项讨论**

**TIPS53:不要忽视编译器的警告**

53.1严肃对待编译器的警告信息。努力在编译器的最高警告等级下争取“无任何警告”的荣誉。

53.2不要过度依赖编译器的报警能力，因为不同编译器对事情的态度不同，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。

**TIPS54:让自己熟悉包括TR1在内的标准程序库**

C++98标准的程序库成分：

1. STL
2. Iostream
3. 国际化支持
4. 数值处理，包括复数模板(complex)和纯数值数组(valarray)
5. 异常阶层体系
6. C89程序库

TR1则详细叙述了14个新组建,统统都放在std命名空间内，更正确地说是在其嵌套命名空间tr1内，因此tr1组建shared\_ptr地全名是std::tr1::shared\_ptr,其组建包括：

1. 智能指针
2. Tr1::function，此物可以表示任何可调用物，也就是任何函数或函数对象
3. Tr1::bind,能够做STL绑定物
4. Hash tables
5. 正则表达式(Regular expressions)
6. Tuples(变量组)
7. Tr1::array
8. Tr1::mem\_fn
9. Tr1::reference\_wrapper
10. 随机数生成工具
11. 数学特殊函数
12. C99兼容扩充
13. Type traits
14. Tr1::result of

54.1C++程序库的主要机能由STL、iostream、locales组成，并包含C99标准程序库

54.2TR1添加了智能指针，一般化指针、hash-based容器、正则表达式以及另外10个组建支持

54.3TR1自身只是一份规范，为获得TR1提供的好处，需要一份实物，一个好的来源是Boost。

TIPS55:让自己熟悉Boost

Boost是一个C++社群，网址：<http://boost.org>

55.1Boost提供许多TR1组件的实现品，以及其他许多程序库