条款一　指针与引用的区别

指针与引用看上去完全不同　指针用操作符　*　和　->　　引用使用操作符　.　但是它们似乎有相同的功能　指针与引用都是让你间接引用其他对象　你如何决定在什么时候使用指针　在什么时候使用引用呢

首先　要认识到在任何情况下都不能用指向空值的引用　一个引用必须总是指向某些对象　因此如果你使用一个变量并让它指向一个对象　但是该变量在某些时候也可能不指向任何对象　这时你应该把变量声明为指针　因为这样你可以赋空值给该变量　相反　如果变量肯定指向一个对象　例如你的设计不允许变量为空　这时你就可以把变量声明为引用

　但是　请等一下　　你怀疑地问　　这样的代码会产生什么样的后果

```
char *pc = 0;                  // 设置指针为空值

char& rc = *pc;                // 让引用指向空值
```

这是非常有害的　毫无疑问　结果将是不确定的　编译器能产生一些输出　导致任何事情都有可能发生　　应该躲开写出这样代码的人除非他们同意改正错误　如果你担心这样的代码会出现在你的软件里　那么你最好完全避免使用引用　要不然就去让更优秀的程序员去做　我们以后将忽略一个引用指向空值的可能性

因为引用肯定会指向一个对象　在 C 里　引用应被初始化

```
string& rs;                    // 错误　引用必须被初始化
string s("xyzzy");
string& rs = s;                // 正确　rs 指向 s
```

指针没有这样的限制

```
string *ps;                    // 未初始化的指针
                               // 合法但危险
```

不存在指向空值的引用这个事实意味着使用引用的代码效率比使用指针的要高　因为在使用引用之前不需要测试它的合法性

```
void printDouble(const double& rd)
{
      cout << rd;              // 不需要测试 rd,它
}                              // 肯定指向一个 double 值
```

相反　指针则应该总是被测试　防止其为空

```
void printDouble(const double *pd)
{
   if (pd) {                   // 检查是否为 NULL
      cout << *pd;
  }
}
```

指针与引用的另一个重要的不同是指针可以被重新赋值以指向另一个不同的对象　但是引用则总是指向在初始化时被指定的对象　以后不能改变

```
string s1("Nancy");
string s2("Clancy");

string& rs = s1;              // rs  引用  s1

string *ps = &s1;             // ps  指向  s1

rs = s2;                          // rs  仍旧引用 s1,
                                  //  但是  s1 的值现在是
                                  // "Clancy"

ps = &s2;                         // ps  现在指向  s2;
                                  // s1  没有改变
```

总的来说　在以下情况下你应该使用指针　一是你考虑到存在不指向任何对象的可能　在这种情况下　你能够设置指针为空　　二是你需要能够在不同的时刻指向不同的对象　在这种情况下　你能改变指针的指向　　如果总是指向一个对象并且一旦指向一个对象后就不会改变指向　那么你应该使用引用

还有一种情况　就是当你重载某个操作符时　你应该使用引用　最普通的例子是操作符[]　这个操作符典型的用法是返回一个目标对象　其能被赋值

```
vector<int> v(10);            //  建立整形向量　vector　　大小为 10;
                                  //  向量是一个在标准 C 库中的一个模板(见条款 35)
v[5] = 10;                        //  这个被赋值的目标对象就是操作符[]返回的值
```

如果操作符[]返回一个指针　那么后一个语句就得这样写

```
*v[5] = 10;
```

但是这样会使得 v 看上去象是一个向量指针　因此你会选择让操作符返回一个引用　　这有一个有趣的例外　参见条款 30

当你知道你必须指向一个对象并且不想改变其指向时 或者在重载操作符并为防止不必要的语义误解时　你不应该使用指针　而在除此之外的其他情况下　则应使用指针

条款 2　尽量使用 C++风格的类型转换

仔细想想地位卑贱的类型转换功能 cast 其在程序设计中的地位就象 goto 语句一样令人鄙视 但是它还不是无法令人忍受 因为当在某些紧要的关头 类型转换还是必需的 这时它是一个必需品
不过 C 风格的类型转换并不代表所有的类型转换功能 一来它们过于粗鲁 能允许你在任何类型之间进行转换 不过如果要进行更精确的类型转换 这会是一个优点 在这些类型转换中存在着巨大的不同 例如把一个指向 const 对象的指针 pointer-to-const-object 转换成指向非 const 对象的指针 pointer-to-non-const-object (即一个仅仅去除 cosnt 的类型转换) 把一个指向基类的指针转换成指向子类的指针 即完全改变对象类型 传统的 C 风格的类型转换不对上述两种转换进行区分 这一点也不令人惊讶 因为 C 风格的类型转换是为 C 语言设计的 而不是为 C++语言设计的
二来 C 风格的类型转换在程序语句中难以识别 在语法上类型转换由圆括号和标识符组成 而这些可以用在 C 中的任何地方 这使得回答象这样一个最基本的有关类型转换的问题变得很困难 在这个程序中是否使用了类型转换
这是因为人工阅读很可能忽略了类型转换的语句 而利用象 grep 的工具程序也不能从语句构成上区分出它们来
C++通过引进四个新的类型转换操作符克服了 C 风格类型转换的缺点 这四个操作符是, static_cast, const_cast, dynamic_cast, 和 reinterpret_cast 在大多数情况下 对于这些操作符你只需要知道原来你习惯于这样写
(type) expression
而现在你总应该这样写
static_cast<type>(expression)

例如 假设你想把一个 int 转换成 double 以便让包含 int 类型变量的表达式产生出浮点数值的结果 如果用 C 风格的类型转换 你能这样写
int firstNumber, secondNumber;
...
double result = ((double)firstNumber)/secondNumber
如果用上述新的类型转换方法 你应该这样写
double result = static_cast<double>(firstNumber)/secondNumber;
这样的类型转换不论是对人工还是对程序都很容易识别
static_cast 在功能上基本上与 C 风格的类型转换一样强大 含义也一样 它也有功能上限制 例如 你不能用 static_cast 象用 C 风格的类型转换一样把 struct 转换成 int 类型或者把 double 类型转换成指针类型 另外 static_cast 不能从表达式中去除 const 属性 因为另一个新的类型转换操作符 const_cast 有这样的功能

其它新的 C++类型转换操作符被用在需要更多限制的地方 const_cast 用于类型转换掉表达式的 const 或 volatileness 属性 通过使用 const_cast 你向人们和编译器强调你通过类型转换想做的只是改变一些东西的 constness 或者 volatileness 属性 这个含义被编译器所约束 如果你试图使用 const_cast 来完成修改 constness 或者 volatileness 属性之外的事情 你的类型转换将被拒绝 下面是一些例子

class Widget { ... };

```
class SpecialWidget: public Widget { ... };
void update(SpecialWidget *psw);
SpecialWidget sw;                          // sw 是一个非 const 对象
const SpecialWidget& csw = sw;     // csw 是 sw 的一个引用
                                                  // 它是一个 const 对象

update(&csw);   // 错误!不能传递一个 const SpecialWidget* 变量
                    // 给一个处理 SpecialWidget*类型变量的函数

update(const_cast<SpecialWidget*>(&csw));
                                  // 正确   csw 的 const 被显示地转换掉
                                 // csw 和 sw 两个变量值在 update
//函数中能被更新

update((SpecialWidget*)&csw);
                                  // 同上   但用了一个更难识别
//的 C 风格的类型转换

Widget *pw = new SpecialWidget;

update(pw);              // 错误   pw 的类型是 Widget*   但是
                            // update 函数处理的是 SpecialWidget*类型

update(const_cast<SpecialWidget*>(pw));
                            // 错误   const_cast 仅能被用在影响
                            // constness or volatileness 的地方上   ,
                            // 不能用在向继承子类进行类型转换
```

到目前为止 const_cast 最普通的用途就是转换掉对象的 const 属性

第二种特殊的类型转换符是 dynamic_cast 它被用于安全地沿着类的继承关系向下进行类型转换 这就是说 你能用 dynamic_cast 把指向基类的指针或引用转换成指向其派生类或其兄弟类的指针或引用 而且你能知道转换是否成功 失败的转换将返回空指针 当对指针进行类型转换时 或者抛出异常 当对引用进行类型转换时

```
Widget *pw;
...
update(dynamic_cast<SpecialWidget*>(pw));
          // 正确   传递给 update 函数一个指针
          // 是指向变量类型为 SpecialWidget 的 pw 的指针
                                  // 如果 pw 确实指向一个对象,
                                  // 否则传递过去的将使空指针
void updateViaRef(SpecialWidget& rsw);
updateViaRef(dynamic_cast<SpecialWidget&>(*pw));
```

```
                                        //正确    传递给 updateViaRef 函数
                                        // SpecialWidget pw  指针   如果 pw
                                        //  确实指向了某个对象
                                        //  否则将抛出异常
```

dynamic_casts 在帮助你浏览继承层次上是有限制的 它不能被用于缺乏虚函数的类型上 参见条款 24   也不能用它来转换掉 constness:

```
int firstNumber, secondNumber;
...
double result = dynamic_cast<double>(firstNumber)/secondNumber;
                                        //  错误   没有继承关系
const SpecialWidget sw;
...
update(dynamic_cast<SpecialWidget*>(&sw));
                                        //  错误! dynamic_cast 不能转换
                                        //  掉 const
```

如你想在没有继承关系的类型中进行转换 你可能想到 static_cast 如果是为了去除 const 你总得用 const_cast

这四个类型转换符中的最后一个是 reinterpret_cast 这个操作符被用于的类型转换的转换结果几乎都是实现时定义 implementation-defined 因此 使用 reinterpret_casts 的代码很难移植

reinterpret_casts 的最普通的用途就是在函数指针类型之间进行转换 例如 假设你有一个函数指针数组

```
typedef void (*FuncPtr)();          // FuncPtr is  一个指向函数
                                                //  的指针   该函数没有参数
                                                //  也返回值类型为 void
FuncPtr funcPtrArray[10];           // funcPtrArray  是一个能容纳
                                                // 10 个 FuncPtrs 指针的数组
```

让我们假设你希望 因为某些莫名其妙的原因 把一个指向下面函数的指针存入 funcPtrArray 数组

```
int doSomething();
```

你不能不经过类型转换而直接去做 因为 doSomething 函数对于 funcPtrArray 数组来说有一个错误的类型 在 FuncPtrArray 数组里的函数返回值是 void 类型 而 doSomething 函数返回值是 int 类型

```
funcPtrArray[0] = &doSomething;       //  错误   类型不匹配
```

reinterpret_cast 可以让你迫使编译器以你的方法去看待它们

```
funcPtrArray[0] =                               // this compiles
```

```
reinterpret_cast<FuncPtr>(&doSomething);
```

转换函数指针的代码是不可移植的　C++不保证所有的函数指针都被用一样的方法表示　在一些情况下这样的转换会产生不正确的结果　参见条款 31　所以你应该避免转换函数指针类型　除非你处于着背水一战和尖刀架喉的危急时刻　一把锋利的刀　一把非常锋利的刀

如果你使用的编译器缺乏对新的类型转换方式的支持 你可以用传统的类型转换方法代替 static_cast, const_cast, and reinterpret_cast　也可以用下面的宏替换来模拟新的类型转换语法

```
#define static_cast(TYPE,EXPR)          ((TYPE)(EXPR))
#define const_cast(TYPE,EXPR)           ((TYPE)(EXPR))
#define reinterpret_cast(TYPE,EXPR)    ((TYPE)(EXPR))
```

你可以象这样使用使用

```
double result = static_cast(double, firstNumber)/secondNumber;
update(const_cast(SpecialWidget*, &sw));
funcPtrArray[0] = reinterpret_cast(FuncPtr, &doSomething);
```

这些模拟不会象真实的操作符一样安全 但是当你的编译器可以支持新的的类型转换时它们可以简化你把代码升级的过程

没有一个容易的方法来模拟 dynamic_cast 的操作　但是很多函数库提供了函数安全地在派生类与基类之间的进行类型转换 如果你没有这些函数而你有必须进行这样的类型转换 你也可以回到 C 风格的类型转换方法上 但是这样的话你将不能获知类型转换是否失败　当然　你也可以定义一个宏来模拟 dynamic_cast 的功能　就象模拟其它的类型转换一样

```
#define dynamic_cast(TYPE,EXPR)        (TYPE)(EXPR)
```

请记住　这个模拟并不能完全实现 dynamic_cast 的功能　它没有办法知道转换是否失败

我知道　是的　我知道　新的类型转换操作符不是很美观而且用键盘键入也很麻烦　如果你发现它们看上去实在令人讨厌　C 风格的类型转换还可以继续使用并且合法 然而正是因为新的类型转换符缺乏美感才能使它弥补了在含义精确性和可辨认性上的缺点　并且使用新类型转换符的程序更容易被解析　不论是对人工还是对于工具程序　它们允许编译器检测出原来不能发现的错误　这些都是放弃 C 风格类型转换方法的强有力的理由　还有第三个理由　也许让类型转换符不美观和键入麻烦是一件好事

条款 3　不要使用多态性数组

类继承的最重要的特性是你可以通过基类指针或引用来操作派生类 这样的指针或引用具有行为的多态性 就好像它们同时具有多种形态 C++允许你通过基类指针和引用来操作派生类数组 不过这根本就不是一个特性 因为这样的代码根本无法如你所愿地那样运行

假设你有一个类 BST 比如是搜索树对象 和继承自 BST 类的派生类 BalancedBST
```
class BST { ... };

class BalancedBST: public BST { ... };
```
在一个真实的程序里 这样的类应该是模板类 但是在这个例子里并不重要 加上模板只会使得代码更难阅读 为了便于讨论 我们假设 BST 和 BalancedBST 只包含 int 类型数据

有这样一个函数 它能打印出 BST 类数组中每一个 BST 对象的内容
```
void printBSTArray(ostream& s,
                   const BST array[],
                   int numElements)
{
   for (int i = 0; i < numElements; ) {
      s << array[i];          //假设 BST 类
   }                                //重载了操作符<<
}
```
当你传递给该函数一个含有 BST 对象的数组变量时 它能够正常运行
```
BST BSTArray[10];

...

printBSTArray(cout, BSTArray, 10);          // 运行正常
```
然而 请考虑一下 当你把含有 BalancedBST 对象的数组变量传递给 printBSTArray 函数时 会产生什么样的后果
```
BalancedBST bBSTArray[10];

...

printBSTArray(cout, bBSTArray, 10);          // 还会运行正常么
```
你的编译器将会毫无警告地编译这个函数 但是再看一下这个函数的循环代码
```
for (int i = 0; i < numElements; ) {
   s << array[i];
}
```
这里的 array[I]只是一个指针算法的缩写 它所代表的是*(array) 我们知道 array 是

一个指向数组起始地址的指针　但是 array 中各元素内存地址与数组的起始地址的间隔究竟有多大呢　它们的间隔是 i*sizeof(一个在数组里的对象)　因为在 array 数组[0]到[I]间有 I 个对象　编译器为了建立正确遍历数组的执行代码　它必须能够确定数组中对象的大小　这对编译器来说是很容易做到的　参数 array 被声明为 BST 类型　所以 array 数组中每一个元素都是 BST 类型　因此每个元素与数组起始地址的间隔是 be i*sizeof(BST)

至少你的编译器是这么认为的　但是如果你把一个含有 BalancedBST 对象的数组变量传递给 printBSTArray 函数　你的编译器就会犯错误　在这种情况下　编译器原先已经假设数组中元素与 BST 对象的大小一致 但是现在数组中每一个对象大小却与 BalancedBST 一致 派生类的长度通常都比基类要长 我们料想 BalancedBST 对象长度的比 BST 长　如果如此的话　printBSTArray 函数生成的指针算法将是错误的 没有人知道如果用 BalancedBST 数组来执行 printBSTArray 函数将会发生什么样的后果　不论是什么后果都是令人不愉快的

如果你试图删除一个含有派生类对象的数组　将会发生各种各样的问题　以下是一种你可能的不正确的做法

```
//删除一个数组，但是首先记录一个删除信息
void deleteArray(ostream& logStream, BST array[])
{
   logStream << "Deleting array at address "
                  << static_cast<void*>(array) << '\n';

delete [] array;
}

BalancedBST *balTreeArray =                          // 建立一个 BalancedBST 对象数组
   new BalancedBST[50];

...

deleteArray(cout, balTreeArray);                     // 记录这个删除操作
```

这里面也掩藏着你看不到的指针算法　当一个数组被删除时　每一个数组元素的析构函数也会被调用　当编译器遇到这样的代码

```
delete [] array;
```

它肯定象这样生成代码

```
// 以与构造顺序相反的顺序来
// 解构 array 数组里的对象
for ( int i =  数组元素的个数  1; i >= 0;--i)
```

```
{
    array[i].BST::~BST();                          // 调用 array[i]的
}                                                  // 析构函数
```

因为你所编写的循环语句根本不能正运行　所以当编译成可执行代码后　也不可能正常运行　语言规范中说通过一个基类指针来删除一个含有派生类对象的数组　结果将是不确定的　这实际意味着执行这样的代码肯定不会有什么好结果　多态和指针算法不能混合在一起来用　所以数组与多态也不能用在一起

值得注意的是如果你不从一个具体类　concrete classes　例如 BST　派生出另一个具体类　例如 BalancedBST　那么你就不太可能犯这种使用多态性数组的错误　正如条款 33 所解释的　不从具体类派生出具体类有很多好处　我希望你阅读一下条款 33 的内容

条款 4　避免无用的缺省构造函数

缺省构造函数　指没有参数的构造函数　在 C++语言中是一种让你无中生有的方法　构造函数能初始化对象,而缺省构造函数则可以不利用任何在建立对象时的外部数据就能初始化对象　有时这样的方法是不错的　例如一些行为特性与数字相仿的对象被初始化为空值或不确定的值也是合理的　还有比如链表　哈希表　图等等数据结构也可以被初始化为空容器

但不是所有的对象都属于上述类型　对于很多对象来说　不利用外部数据进行完全的初始化是不合理的 比如一个没有输入姓名的地址簿对象 就没有任何意义　在一些公司里　所有的设备都必须标有一个公司 ID 号码　所以在建立对象以模型化一个设备时　不提供一个合适的 ID 号码　所建立的对象就根本没有意义

在一个完美的世界里　无需任何数据即可建立对象的类可以包含缺省构造函数　而需要数据来建立对象的类则不能包含缺省构造函数　唉　可是我们的现实世界不是完美的　所以我们必须考虑更多的因素　特别是如果一个类没有缺省构造函数　就会存在一些使用上的限制

请考虑一下有这样一个类　它表示公司的设备　这个类包含一个公司的 ID 代码　这个 ID 代码被强制做为构造函数的参数

```
class EquipmentPiece {
public:
    EquipmentPiece(int IDNumber);
    ...
};
```

因为 EquipmentPiece 类没有一个缺省构造函数　所以在三种情况下使用它　就会遇到问题　第一中情况是建立数组时　一般来说　没有一种办法能在建立对象数组时给构造函数传递参数　所以在通常情况下　不可能建立 EquipmentPiece 对象

数组

```
EquipmentPiece bestPieces[10];                  // 错误  没有正确调用
                                                        // EquipmentPiece 构造函数
 EquipmentPiece *bestPieces =
   new EquipmentPiece[10];                        // 错误  与上面的问题一样
```

不过还是有三种方法能回避开这个限制 对于使用非堆数组 non-heap arrays 即不在堆中给数组分配内存 译者注 的一种解决方法是在数组定义时提供必要的参数

```
int ID1, ID2, ID3, ..., ID10;                   // 存储设备 ID 号的
                                                        // 变量
...

EquipmentPiece bestPieces[] = {                 // 正确, 提供了构造
   EquipmentPiece(ID1),                           // 函数的参数
   EquipmentPiece(ID2),
   EquipmentPiece(ID3),
   ...,
   EquipmentPiece(ID10)
};
```

不过很遗憾 这种方法不能用在堆数组(heap arrays)的定义上

一个更通用的解决方法是利用指针数组来代替一个对象数组

```
typedef EquipmentPiece* PEP;                    //  PEP 指针指向
                                                        //一个 EquipmentPiece 对象

PEP bestPieces[10];                                 // 正确, 没有调用构造函数
PEP *bestPieces = new PEP[10];                  // 也正确
```

在指针数组里的每一个指针被重新赋值 以指向一个不同的 EquipmentPiece 对象

```
for (int i = 0; i < 10; ++i)
   bestPieces[i] = new EquipmentPiece( ID Number );
```

不过这中方法有两个缺点 第一你必须删除数组里每个指针所指向的对象 如果你忘了 就会发生内存泄漏 第二增加了内存分配量 因为正如你需要空间来容纳 EquipmentPiece 对象一样 你也需要空间来容纳指针

如果你为数组分配 raw memory 你就可以避免浪费内存 使用 placement new 方法 参见条款 8 在内存中构造 EquipmentPiece 对象

```
// 为大小为 10 的数组 分配足够的内存
// EquipmentPiece 对象; 详细情况请参见条款 8
// operator new[] 函数
void *rawMemory =
    operator new[](10*sizeof(EquipmentPiece));
// make bestPieces point to it so it can be treated as an
// EquipmentPiece array
EquipmentPiece *bestPieces =
    static_cast<EquipmentPiece*>(rawMemory);
// construct the EquipmentPiece objects in the memory
// 使用"placement new" (参见条款 8)
for (int i = 0; i < 10; ++i)
    new (&bestPieces[i]) EquipmentPiece( ID Number );
```

注意你仍旧得为每一个 EquipmentPiece 对象提供构造函数参数 这个技术 也称为数组到指针的思想 array-of-pointers 允许你在没有缺省构造函数的情况下建立一个对象数组 它没有绕过对构造函数参数的需求 实际上也做不到 如果能做到的话 就不能保证对象被正确初始化
使用 placement new 的缺点除了是大多数程序员对它不熟悉外 能使用它就更难了 还有就是当你不想让它继续存在使用时 必须手动调用数组对象的析构函数 调用操作符 delete[]来释放 raw memory 请再参见条款 8

```
// 以与构造 bestPieces 对象相反的顺序
// 解构它
for (int i = 9; i >= 0; --i)
    bestPieces[i].~EquipmentPiece();

// deallocate the raw memory
operator delete[](rawMemory);
```

如果你忘记了这个要求或没有用这个数组删除方法 那么你程序的运行将是不可预测的 这是因为直接删除一个不是用 new 操作符来分配的内存指针 其结果没有被定义的

```
delete [] bestPieces;                    // 没有定义! bestPieces
                                         //不是用 new 操作符分配的
```

有关 new placement new 和它们如何与构造函数 析构函数一起使用的更多信息请见条款 8
对于类里没有定义缺省构造函数所造成的第二个问题是它们无法在许多基于模板 template-based 容器类里使用 因为实例化一个模板时 模板的类型参数应该提供一个缺省构造函数 这是一个常见的要求 这个要求总是来自于模板内部被建立的模板参数类型数组里 例如一个数组模板类

```
template<class T>
class Array {
public:
    Array(int size);
    ...

private:
    T *data;
};

template<class T>
Array<T>::Array(int size)
{
    data = new T[size];                          // 为每个数组元素
    ...                                          //依次调用 T::T()
}
```

在多数情况下 通过仔细设计模板可以杜绝对缺省构造函数的需求 例如标准的 vector 模板 生成一个类似于可扩展数组的类 对它的类型参数没有必须有缺省构造函数的要求 不幸的是 很多模板类没有以仔细的态度去设计 这样没有缺省构造函数的类就不能与许多模板兼容 当 C++程序员深入领会了模板设计以后 这样的问题应该不再那么突出了 这会花多长时间 完全在于个人的造化

最后讲一下在设计虚基类时所面临的是要提供缺省构造函数还是不提供缺省构造函数的两难决策 不提供缺省构造函数的虚基类很难与其进行工作 因为几乎所有的派生类在实例化时斗必须给虚基类构造函数提供参数 这就要求所有从没有缺省构造函数的虚基类继承下来的派生类(无论有多远)都必须知道并理解提供给虚基类构造函数的参数含义 派生类的作者是不会企盼和喜欢这种规定的

因为这些强加于没有缺省构造函数的类上的种种限制 一些人认为所有的类都应该有缺省构造函数 即使缺省构造函数没有足够的数据来初始化一个对象 比如这个原则的拥护者会这样修改 EquipmentPiece 类

```
class EquipmentPiece {
public:
    EquipmentPiece(    int IDNumber = UNSPECIFIED);
    ...
private:
    static const int      UNSPECIFIED;              // ID 值不确定
};
```

这允许这样建立 EquipmentPiece 对象

```
EquipmentPiece e;                                    //这样合法
```

这样的修改使得其他成员函数变得复杂　因为不再能确保 EquipmentPiece 对象进行有意义的初始化 假设它建立一个因没有 ID 而没有意义的 EquipmentPiece 对象 那么大多数成员函数必须检测 ID 是否存在　如果不存在 ID　它们将必须知道怎么犯的错误　不过这经常是不明晰的　很多代码实现什么也没有提供　只是抛出一个异常或调用一个函数终止程序　当这种情形发生时　很难说提供缺省构造函数而放弃了一种保证机制这种做法是否能提高软件的总体质量

提供无意义的缺省构造函数也会影响类的工作效率 如果成员函数必须测试所有的部分是否都被正确地初始化　那么这些函数的调用者就得为此付出更多的时间　而且还得付出更多的代码　因为这使得可执行文件或库变得更大　它们也得在测试失败的地方放置代码来处理错误 如果一个类的构造函数能够确保所有的部分被正确初始化　所有这些弊病都能够避免　缺省构造函数一般不会提供这种保证　所以在它们能使类变得没有意义时　尽量去避免使用它们　使用这种类的确有一些限制　但是当你使用它时　它也给你提供了一种保证　你能相信这个类被正确地建立和高效地实现

条款 5　谨慎定义类型转换函数

C　　编译器能够在两种数据类型之间进行隐式转换　implicit conversions　它继承了 C 语言的转换方法　例如允许把 char 隐式转换为 int 和从 short 隐式转换为 double　因此当你把一个 short 值传递给准备接受 double 参数值的函数时　依然可以成功运行　C 中许多这种可怕的转换可能会导致数据的丢失　它们在 C++中依然存在　包括 int 到 short 的转换和 double 到 char 的转换

你对这些类型转换是无能为力的　因为它们是语言本身的特性　不过当你增加自己的类型时　你就可以有更多的控制力　因为你能选择是否提供函数让编译器进行隐式类型转换

有两种函数允许编译器进行这些的转换　单参数构造函数　single-argument constructors　和隐式类型转换运算符　单参数构造函数是指只用一个参数即可以调用的构造函数　该函数可以是只定义了一个参数　也可以是虽定义了多个参数但第一个参数以后的所有参数都有缺省值　以下有两个例子

```
class Name {                                          // for names of things
public:
   Name(const string& s);                             //  转换  string  到
                                                          // Name
   ...

};
```

```
class Rational {                                    // 有理数类
public:
   Rational(int numerator = 0,                      // 转换 int 到
            int denominator = 1);                   // 有理数类
   ...

};
```

隐式类型转换运算符只是一个样子奇怪的成员函数　operator 关键字　其后跟一个类型符号　你不用定义函数的返回类型　因为返回类型就是这个函数的名字　例如为了允许 Rational(有理数)类隐式地转换为 double 类型　在用有理数进行混合类型运算时　可能有用　　你可以如此声明 Rational 类

```
class Rational {
public:
   ...
   operator double() const;                         // 转换 Rational 类成
};                                                  // double 类型
```

在下面这种情况下　这个函数会被自动调用

```
Rational r(1, 2);                                   // r 的值是 1/2

double d = 0.5 * r;                                 // 转换 r 到 double,
                                                    // 然后做乘法
```

以上这些说明只是一个复习　我真正想说的是为什么你不需要定义各中类型转换函数

根本问题是当你在不需要使用转换函数时　这些的函数缺却能被调用运行　结果这些不正确的程序会做出一些令人恼火的事情　而你又很难判断出原因

让我们首先分析一下隐式类型转换运算符　它们是最容易处理的　假设你有一个如上所述的 Rational 类　你想让该类拥有打印有理数对象的功能　就好像它是一个内置类型　因此　你可能会这么写

```
Rational r(1, 2);

cout << r;                                          // 应该打印出"1/2"
```

再假设你忘了为 Rational 对象定义 operator<<　你可能想打印操作将失败　因为没有合适的的 operator<<被调用　但是你错了　当编译器调用 operator<<时　会发现没有这样的函数存在　但是它会试图找到一个合适的隐式类型转换顺序以使得函数调用正常运行　类型转换顺序的规则定义是复杂的　但是在这种情况下编译器会发现它们能调用 Rational::operator double 函数　来把 r 转换为 double 类型　所以

上述代码打印的结果是一个浮点数　而不是一个有理数　这简直是一个灾难　但是它表明了隐式类型转换的缺点　它们的存在将导致错误的发生

解决方法是用等同的函数来替代转换运算符　而不用语法关键字　例如为了把 Rational 对象转换为 double　用 asDouble 函数代替 operator double 函数

```
class Rational {
public:
   ...
   double asDouble() const;                          //转变 Rational
};                                                          // 成 double
```

这个成员函数能被显式调用

```
Rational r(1, 2);

cout << r;                                          // 错误! Rationa 对象没有
                                                        // operator<<

cout << r.asDouble();                                    // 正确，用 double 类型
//打印 r
```

在多数情况下　这种显式转换函数的使用虽然不方便　但是函数被悄悄调用的情况不再会发生　这点损失是值得的　一般来说　越有经验的 C++程序员就越喜欢避开类型转换运算符　例如在 C++标准库　参见条款 49 和 35　委员会工作的人员是在此领域最有经验的　他们加在库函数中的 string 类型没有包括隐式地从 string 转换成 C 风格的 char*的功能　而是定义了一个成员函数 c_str 用来完成这个转换　这是巧合么　我看不是

通过单参数构造函数进行隐式类型转换更难消除 而且在很多情况下这些函数所导致的问题要甚于隐式类型转换运算符

举一个例子　一个 array 类模板　这些数组需要调用者确定边界的上限与下限

```
template<class T>
class Array {
public:
   Array(int lowBound, int highBound);
   Array(int size);

   T& operator[](int index);

   ...
```

```
};
```

第一个构造函数允许调用者确定数组索引的范围　例如从 10 到 20　它是一个两参数构造函数　所以不能做为类型转换函数　第二个构造函数让调用者仅仅定义数组元素的个数　使用方法与内置数组的使用相似　　不过不同的是它能做为类型转换函数使用　能导致无穷的痛苦

例如比较 Array<int>对象　部分代码如下

```
bool operator==( const Array<int>& lhs,
                 const Array<int>& rhs);

Array<int> a(10);
Array<int> b(10);

...

for (int i = 0; i < 10; ++i)
   if (a == b[i]) {                    // 哎呦! "a"  应该是  "a[i]"
      do something for when
      a[i] and b[i] are equal;
   }
   else {
      do something for when they're not;
   }
```

我们想用 a 的每个元素与 b 的每个元素相比较　但是当录入 a 时　我们偶然忘记了数组下标　当然我们希望编译器能报出各种各样的警告信息　但是它根本没有　因为它把这个调用看成用 Array<int>参数(对于 a)和 int (对于 b[i])参数调用 operator==函数　　然而没有 operator==函数是这些的参数类型　我们的编译器注意到它能通过调用 Array<int>构造函数能转换 int 类型到 Array<int>类型　这个构造函数只有一个 int 类型的参数　然后编译器如此去编译　生成的代码就象这样

```
for (int i = 0; i < 10; ++i)
   if (a == static_cast< Array<int> >(b[i]))      ...
```

每一次循环都把 a 的内容与一个大小为 b[i]的临时数组　内容是未定义的　比较　这不仅不可能以正确的方法运行　而且还是效率低下的　因为每一次循环我们都必须建立和释放 Array<int>对象　见条款 19

通过不声明运算符　operator　的方法,可以克服隐式类型转换运算符的缺点　但是单参数构造函数没有那么简单　毕竟　你确实想给调用者提供一个单参数构造函数　同时你也希望防止编译器不加鉴别地调用这个构造函数　幸运的是　有一个方法可以让你鱼肉与熊掌兼得　事实上是两个方法　一是容易的方法　二是当你的编译器不支持容易的方法时所必须使用的方法

容易的方法是利用一个最新编译器的特性　explicit 关键字　为了解决隐式类型转换而特别引入的这个特性　它的使用方法很好理解　构造函数用 explicit 声明　如果这样做　编译器会拒绝为了隐式类型转换而调用构造函数　显式类型转换依然合法

```
template<class T>
class Array {
public:
    ...
    explicit Array(int size);                    //  注意使用"explicit"
    ...
};

Array<int> a(10);                                //  正确, explicit  构造函数
                                                 //  在建立对象时能正常使用

Array<int> b(10);                                //  也正确

if (a == b[i]) ...                               //  错误!  没有办法
                                                 //  隐式转换
                                                 // int  到  Array<int>

if (a == Array<int>(b[i])) ...                   //  正确,显式从 int 到
                                                 // Array<int>转换
                                                 //     但是代码的逻辑
                                                 //  不合理

if (a == static_cast< Array<int> >(b[i])) ...
                                                 //  同样正确　同样
                                                 //  不合理

if (a == (Array<int>)b[i]) ...                   //C 风格的转换也正确
                                                 //  但是逻辑
                                                 //  依旧不合理
```

在例子里使用了 static_cast　参见条款 2　　两个　>　字符间的空格不能漏掉　如果这样写语句

```
if (a == static_cast<Array<int>>(b[i])) ...
```

这是一个不同的含义的语句　因为 C++编译器把　>>　做为一个符号来解释　在两个　>　间没有空格　语句会产生语法错误

如果你的编译器不支持 explicit　你不得不回到不使用成为隐式类型转换函数的单参数构造函数

我前面说过复杂的规则决定哪一个隐式类型转换是合法的　哪一个是不合法的　这些规则中没有一个转换能够包含用户自定义类型　调用单参数构造函数或隐式类型转换运算符　　你能利用这个规则来正确构造你的类　使得对象能够正常构造　同时去掉你不想要的隐式类型转换

再来想一下数组模板　你需要用整形变量做为构造函数参数来确定数组大小　但是同时又必须防止从整数类型到临时数组对象的隐式类型转换　你要达到这个目的　先要建立一个新类 ArraySize　这个对象只有一个目的就是表示将要建立数组的大小　你必须修改 Array 的单参数构造函数　用一个 ArraySize 对象来代替 int　代码如下

```
template<class T>
class Array {
public:

   class ArraySize {                              // 这个类是新的
   public:
      ArraySize(int numElements): theSize(numElements) {}
      int size() const { return theSize; }

private:
      int theSize;
   };

Array(int lowBound, int highBound);
   Array(ArraySize size);                         // 注意新的声明

...

};
```

这里把 ArraySize 嵌套入 Array 中　为了强调它总是与 Array 一起使用　你也必须声明 ArraySize 为公有　为了让任何人都能使用它

想一下　当通过单参数构造函数定义 Array 对象　会发生什么样的事情

```
Array<int> a(10);
```

你的编译器要求用 int 参数调用 Array<int>里的构造函数　但是没有这样的构造函

数　编译器意识到它能从 int 参数转换成一个临时 ArraySize 对象　ArraySize 对象只是 Array<int>构造函数所需要的　这样编译器进行了转换　函数调用　及其后的对象建立　也就成功了

事实上你仍旧能够安心地构造 Array 对象　不过这样做能够使你避免类型转换　考虑一下以下代码

```
bool operator==( const Array<int>& lhs,
                 const Array<int>& rhs);
Array<int> a(10);
Array<int> b(10);
...
for (int i = 0; i < 10; ++i)
   if (a == b[i]) ...                    // 哎呦! "a"  应该是  "a[i]";
                                         // 现在是一个错误
```

为了调用 operator　　函数　编译器要求 Array<int>对象在　==　右侧,但是不存在一个参数为 int 的单参数构造函数 而且编译器无法把 int 转换成一个临时 ArraySize 对象然后通过这个临时对象建立必须的 Array<int>对象 因为这将调用两个用户定义 user-defined 的类型转换 一个从 int 到 ArraySize 一个从 ArraySize 到 Array<int> 这种转换顺序被禁止的　所以当试图进行比较时编译器肯定会产生错误

ArraySize 类的使用有些象一个有目的的帮手　这是一个更通用技术的应用实例　类似于 ArraySize 的类经常被称为 proxy classes　因为这样类的每一个对象都为了支持其他对象的工作　ArraySize 对象实际是一个整数类型的替代者　用来在建立 Array 对象时确定数组大小　Proxy 对象能帮你更好地控制软件的在某些方面的行为,否则你就不能控制这些行为　比如在上面的情况里　这种行为是指隐式类型转换　所以它值得你去学习和使用　你可能会问你如何去学习它呢　一种方法是转向条款 33　它专门讨论 proxy classes

在你跳到条款 33 之前　再仔细考虑一下本条款的内容　让编译器进行隐式类型转换所造成的弊端要大于它所带来的好处　所以除非你确实需要　不要定义类型转换函数

条款6　自增(increment)　自减(decrement)操作符前缀形式与后缀形式的区别

很久以前　八十年代　　没有办法区分++和--操作符的前缀与后缀调用　这个问题遭到程序员的报怨　于是C++语言得到了扩展　允许重载increment 和 decrement 操作符的两种形式

然而有一个句法上的问题　重载函数间的区别决定于它们的参数类型上的差异　但是不论是increment或decrement的前缀还是后缀都只有一个参数　为了解决这个

语言问题　C++规定后缀形式有一个int类型参数　当函数被调用时　编译器传递一个0做为int参数的值给该函数

```
class UPInt {                                   // "unlimited precision int"
public:
    UPInt& operator++();                        // ++ 前缀
    const UPInt operator++(int);                // ++ 后缀

    UPInt& operator--();                        // -- 前缀
    const UPInt operator--(int);                // -- 后缀

    UPInt& operator+=(int);                     // += 操作符　UPInts
                                                //  与ints 相运算
    ...
};
UPInt i;
++i;                                            //  调用 i.operator++();
i++;                                            //  调用 i.operator++(0);
--i;                                            //  调用 i.operator--();
i--;                                            //  调用 i.operator--(0);
```

这个规范有一些古怪　不过你会习惯的　而尤其要注意的是这些操作符前缀与后缀形式返回值类型是不同的　前缀形式返回一个引用　后缀形式返回一个const类型　下面我们将讨论++操作符的前缀与后缀形式　这些说明也同样使用与--操作符

从你开始做C程序员那天开始　你就记住increment的前缀形式有时叫做“增加然后取回”　后缀形式叫做“取回然后增加”　这两句话非常重要　因为它们是increment前缀与后缀的形式上的规范

```
//  前缀形式　增加然后取回值
UPInt& UPInt::operator++()
{
    *this += 1;                                 //  增加
    return *this;                               //  取回值
}
// postfix form: fetch and increment
const UPInt UPInt::operator++(int)
{
```

```
    UPInt oldValue = *this;                          // 取回值
    ++(*this);              // 增加
return oldValue;                                     // 返回被取回的值
}
```

后缀操作符函数没有使用它的参数　它的参数只是用来区分前缀与后缀函数调用　如果你没有在函数里使用参数　许多编译器会显示警告信息　很令人讨厌　为了避免这些警告信息　一种经常使用的方法时省略掉你不想使用的参数名称　如上所示

很明显一个后缀increment必须返回一个对象　它返回的是增加前的值　　但是为什么是const对象呢　假设不是const对象　下面的代码就是正确的

```
UPInt i;
i++++;                                                // 两次increment后缀
                                                      // 运算
```

这组代码与下面的代码相同

```
i.operator++(0).operator++(0);
```

很明显　第一个调用的operator++函数返回的对象调用了第二个operator++函数

有两个理由导致我们应该厌恶上述这种做法　第一是与内置类型行为不一致　当设计一个类遇到问题时　一个好的准则是使该类的行为与int类型一致　而int类型不允许连续进行两次后缀increment

```
int i;
i++++;                                                // 错误!
```

第二个原因是使用两次后缀increment所产生的结果与调用者期望的不一致　如上所示　第二次调用operator++改变的值是第一次调用返回对象的值　而不是原始对象的值　因此如果

```
i++++;
```

是合法的　i将仅仅增加了一次　这与人的直觉相违背　使人迷惑　对于int类型和UPInt都是一样　,所以最好禁止这么做

C++禁止int类型这么做　同时你也必须禁止你自己写的类有这样的行为　最容易的方法是让后缀increment 返回const对象　当编译器遇到这样的代码

```
i++++;                                          // same as
i.operator++(0).operator++(0);
```

它发现从第一个operator++函数返回的const对象又调用operator++函数 然而这个函数是一个non　const成员函数　所以const对象不能调用这个函数　如果你原来想过让一个函数返回const对象没有任何意义　现在你就知道有时还是有用的　后缀increment和decrement就是例子　　更多的例子参见Effective C++ 条款21

如果你很关心效率问题　当你第一次看到后缀increment函数时，你可能觉得有些问题　这个函数必须建立一个临时对象以做为它的返回值　　参见条款19　　上述实现代码建立了一个显示的临时对象　oldValue　　这个临时对象必须被构造并在最后被结构　前缀increment函数没有这样的临时对象　由此得出一个令人惊讶的结论　如果仅为了提高代码效率　UPInt的调用者应该尽量使用前缀increment少用后缀increment　除非确实需要使用后缀increment　让我们明确一下　当处理用户定义的类型时　尽可能地使用前缀increment　因为它的效率较高

我们再观察一下后缀与前缀increment 操作符　它们除了返回值不同外　所完成的功能是一样的　即值加一　简而言之　它们被认为功能一样　那么你如何确保后缀increment和前缀increment的行为一致呢　当不同的程序员去维护和升级代码时　有什么能保证它们不会产生差异　除非你遵守上述代码里的原则　这才能得到确保　这个原则是后缀increment和decrement应该根据它们的前缀形式来实现　你仅仅需要维护前缀版本　因为后缀形式自动与前缀形式的行为一致

正如你所看到的　掌握前缀和后缀increment和decrement是容易的　一旦了解了他们正确的返回值类型以及后缀操作符应该以前缀操作符为基础来实现的规则 就足够了

条款7　不要重载overload &&, ||, or ,.

与C一样　C++使用布尔表达式简化求值法(short-circuit evaluation)　这表示一旦确定了布尔表达式的真假值　即使还有部分表达式没有被测试　布尔表达式也停止运算　例如

```
char *p;
...
if ((p != 0) && (strlen(p) > 10)) ...
```

这里不用担心当p为空时strlen无法正确运行　因为如果p不等于0的测试失败　strlen不会被调用　同样

```
int rangeCheck(int index)
{
if ((index < lowerBound) || (index > upperBound)) ...
...
}
```

如果index小于lowerBound　它不会与upperBound进行比较

很早以前上述行为特性就被反复灌输给C和C++的程序员　所以他们都知道该特性　而且他们也依赖于简短求值法来写程序　例如在上述第一个代码中　当p为空指针时确保strlen不会被调用是很重要的　因为C++标准说(正如C标准所说)用空指针调用strlen　结果不确定

C++允许根据用户定义的类型　来定制&&和||操作符　方法是重载函数operator&&和operator||　你能在全局重载或每个类里重载　然而如果你想使用这种方法　你必须知道你正在极大地改变游戏规则　因为你以函数调用法替代了简短计算法　也就是说如果你重载了操作符&&　对于你来说代码是这样的

```
if (expression1 && expression2) ...
```

对于编译器来说　等同于下面代码之一

```
if (expression1.operator&&(expression2)) ...
                                        // when operator&& is a
                                        // member function
if (operator&&(expression1, expression2)) ...
                                        // when operator&& is a
                                        // global function
```

这好像没有什么不同　但是函数调用法与简短求值法是绝对不同的　首先当函数被调用时　需要运算其所有参数　所以调用函数functions operator&& 和 operator||时　两个参数都需要计算　换言之　没有采用简短计算法　第二是C++语言规范没有定义函数参数的计算顺序　所以没有办法知道表达式1与表达式2哪一个先计算　完全与具有从左参数到右参数计算顺序的简短计算法相反

因此如果你重载&&或||　就没有办法提供给程序员他们所期望和使用的行为特性

所以不要重载&&和||

同样的理由也适用于括号操作符 但是在我们深入研究它之前 我还是暂停一下让你不要太惊讶 “逗号操作符 哪有逗号操作符 ”确实存在

逗号操作符用于组成表达式 你经常在for循环的更新部分 update part 里遇见它例如下面来源于Kernighan's and Ritchie's 经典书籍The C Programming Language 第二版(Prentice-Hall, 1988)的函数

```
// reverse string s in place
void reverse(char s[])
{
for (int i = 0, j = strlen(s)-1;
        i < j;
        ++i, --j)                  //  啊!  逗号操作符!
{
   int c = s[i];
   s[i] = s[j];
   s[j] = c;
}
}
```

在for循环的最后一个部分里 i被增加同时j被减少 在这里使用逗号很方便 因为在最后一个部分里只能使用一个表达式 分开表达式来改变i和j的值是不合法的

对于内建类型&&和|| C++有一些规则来定义它们如何运算 与此相同 也有规则来定义逗号操作符的计算方法 一个包含逗号的表达式首先计算逗号左边的表达式 然后计算逗号右边的表达式 整个表达式的结果是逗号右边表达式的值 所以在上述循环的最后部分里 编译器首先计算++i 然后是—j 逗号表达式的结果是--j

也许你想为什么你需要知道这些内容呢 因为你需要模仿这个行为特性 如果你想大胆地写自己的逗号操作符函数 不幸的是你无法模仿

如果你写一个非成员函数operator 你不能保证左边的表达式先于右边的表达式计算 因为函数 operator 调用时两个表达式做为参数被传递出去 但是你不能控制函数参数的计算顺序 所以非成员函数的方法绝对不行

剩下的只有写成员函数operator的可能性了 即使这里你也不能依靠于逗号左边表达式先被计算的行为特性 因为编译器不一定必须按此方法去计算 因此你不能重载逗号操作符 保证它的行为特性与其被料想的一样 重载它是完全轻率的行为

你可能正在想这个重载恶梦究竟有没有完 毕竟如果你能重载逗号操作符 你还有什么不能重载的呢 正如显示的 存在一些限制 你不能重载下面的操作符

. .* :: ?:

new delete sizeof typeid
static_cast dynamic_cast const_cast reinterpret_cast
你能重载
operator new operator delete
operator new[] operator delete[]
+ - * / % ^ & | ~
! = < > += -= *= /= %=
^= &= |= << >> >>= <<= == !=
<= >= && || ++ -- , ->* ->
() []

有关new和delete还有operator new, operator delete, operator new[], and operator delete[]的信息参见条款8

当然能重载这些操作符不是去重载的理由 操作符重载的目的是使程序更容易阅读 书写和理解 而不是用你的知识去迷惑其他人 如果你没有一个好理由重载操作符 就不要重载 在遇到&&, ||, 和 ,时 找到一个好理由是困难的 因为无论你怎么努力 也不能让它们的行为特性与所期望的一样

条款8　理解各种不同含义的new和delete

人们有时好像喜欢故意使C++语言的术语难以理解　比如说new操作符　new operator　和operator new的区别
当你写这样的代码

```
string *ps = new string("Memory Management");
```

你使用的new是new操作符　这个操作符就象sizeof一样是语言内置的　你不能改变它的含义　它的功能总是一样的　它要完成的功能分成两部分　第一部分是分配足够的内存以便容纳所需类型的对象　第二部分是它调用构造函数初始化内存中的对象　new操作符总是做这两件事情　你不能以任何方式改变它的行为
你所能改变的是如何为对象分配内存　new操作符调用一个函数来完成必需的内存分配　你能够重写或重载这个函数来改变它的行为　new操作符为分配内存所调用函数的名字是operator new
函数operator new 通常这样声明

```
void * operator new(size_t size);
```

返回值类型是void*　因为这个函数返回一个未经处理　raw　的指针　未初始化的内存　　如果你喜欢　你能写一种operator new函数　在返回一个指针之前能够初始化内存以存储一些数值　但是一般不这么做　　参数size_t确定分配多少内存　你能增加额外的参数重载函数operator new　但是第一个参数类型必须是size_t　有关operator new更多的信息参见Effective C++　条款8至条款10
你一般不会直接调用operator new　但是一旦这么做　你可以象调用其它函数一样调用它

```
void *rawMemory = operator new(sizeof(string));
```

操作符operator new将返回一个指针　指向一块足够容纳一个string类型对象的内存
就象malloc一样　operator new的职责只是分配内存　它对构造函数一无所知　operator new所了解的是内存分配　把operator new　返回的未经处理的指针传递给一个对象是new操作符的工作　当你的编译器遇见这样的语句

```
string *ps = new string("Memory Management");
```

它生成的代码或多或少与下面的代码相似　更多的细节见Effective C++条款8和条款10　还有我的文章Counting object里的注释

```
void *memory =                                    // 得到未经处理的内存
   operator new(sizeof(string));                  // 为String对象
call string::string("Memory Management")          //初始化
on *memory;                                       // 内存中
                                                  // 的对象
string *ps =                                      // 是ps指针指向
   static_cast<string*>(memory);                  // 新的对象
```

注意第二步包含了构造函数的调用　你做为一个程序员被禁止这样去做　你的编译器则没有这个约束　它可以做它想做的一切　因此如果你想建立一个堆对象就必须用new操作符　不能直接调用构造函数来初始化对象

Placement new

有时你确实想直接调用构造函数　在一个已存在的对象上调用构造函数是没有意义的　因为构造函数用来初始化对象　而一个对象仅仅能在给它初值时被初始化一次　但是有时你有一些已经被分配但是尚未处理的的(raw)内存　你需要在这些内存中构造一个对象　你可以使用一个特殊的operator new　它被称为placement new

下面的例子是placement new如何使用　考虑一下

```
class Widget {
public:
   Widget(int widgetSize);
   ...
};
Widget * constructWidgetInBuffer(void *buffer,
                                 int widgetSize)
{
   return new (buffer) Widget(widgetSize);
}
```

这个函数返回一个指针　指向一个Widget对象　对象在转递给函数的buffer里分配　当程序使用共享内存或memory-mapped I/O时这个函数可能有用　因为在这样程序里对象必须被放置在一个确定地址上或一块被例程分配的内存里　参见条款4,一个如何使用placement new的一个不同例子

在constructWidgetInBuffer里面　返回的表达式是

```
new (buffer) Widget(widgetSize)
```

这初看上去有些陌生　但是它是new操作符的一个用法　需要使用一个额外的变量　buffer　当new操作符隐含调用operator new函数时　把这个变量传递给它　被调用的operator new函数除了待有强制的参数size_t外　还必须接受void*指针参数　指向构造对象占用的内存空间　这个operator new就是placement new　它看上去象这样

```
void * operator new(size_t, void *location)
{
   return location;
}
```

这可能比你期望的要简单　但是这就是placement new需要做的事情　毕竟operator new的目的是为对象分配内存然后返回指向该内存的指针　在使用placement new的情况下　调用者已经获得了指向内存的指针　因为调用者知道对象应该放在哪

里 placement new必须做的就是返回转递给它的指针 没有用的 但是强制的参数size_t没有名字 以防止编译器发出警告说它没有被使用 见条款6
placement new是标准C++库的一部分 见Effective C++ 条款49 为了使用placement new 你必须使用语句#include <new> 或者如果你的编译器还不支持这新风格的头文件名(再参见Effective C++ 条款49) <new.h>
让我们从placement new回来片刻 看看new操作符 new operator 与operator new的关系 你想在堆上建立一个对象 应该用new操作符 它既分配内存又为对象调用构造函数 如果你仅仅想分配内存 就应该调用operator new函数 它不会调用构造函数 如果你想定制自己的在堆对象被建立时的内存分配过程 你应该写你自己的operator new函数 然后使用new操作符 new操作符会调用你定制的operator new 如果你想在一块已经获得指针的内存里建立一个对象 应该用placement new
有关更多的不同的new与delete的观点参见Effective C++ 条款7和我的文章
Counting objects
Deletion and Memory Deallocation
为了避免内存泄漏 每个动态内存分配必须与一个等同相反的deallocation对应函数operator delete与delete操作符的关系与operator new与new操作符的关系一样 当你看到这些代码

```
string *ps;
...
delete ps;                                  // 使用delete 操作符
```

你的编译器会生成代码来析构对象并释放对象占有的内存
Operator delete用来释放内存 它被这样声明

```
void operator delete(void *memoryToBeDeallocated);
```

因此

```
delete ps;
```

导致编译器生成类似于这样的代码

```
ps->~string();                              // call the object's dtor
operator delete(ps);                        // deallocate the memory
                                            // the object occupied
```

这有一个隐含的意思是如果你只想处理未被初始化的内存 你应该绕过new和delete操作符 而调用operator new 获得内存和operator delete释放内存给系统

```
void *buffer =                              // 分配足够的
   operator new(50*sizeof(char));       // 内存以容纳50个char
                                            //没有调用构造函数
...
operator delete(buffer);                    // 释放内存
                                            // 没有调用析构函数
```

这与在C中调用malloc和free等同

如果你用placement new在内存中建立对象　你应该避免在该内存中用delete操作符　因为delete操作符调用operator delete来释放内存　但是包含对象的内存最初不是被operator new分配的　placement new只是返回转递给它的指针　谁知道这个指针来自何方　而你应该显式调用对象的析构函数来解除构造函数的影响

```
// 在共享内存中分配和释放内存的函数
void * mallocShared(size_t size);
void freeShared(void *memory);
void *sharedMemory = mallocShared(sizeof(Widget));
Widget *pw =                                          // 如上所示,
   constructWidgetInBuffer(sharedMemory, 10);     // 使用
                                                      // placement new
...
delete pw;                    // 结果不确定! 共享内存来自
                              // mallocShared, 而不是operator new
pw->~Widget();                // 正确　析构 pw指向的Widget
                              // 但是没有释放
                              //包含Widget的内存
freeShared(pw);               // 正确　释放pw指向的共享内存
                              // 但是没有调用析构函数
```

如上例所示　如果传递给placement new的raw内存是自己动态分配的　通过一些不常用的方法　如果你希望避免内存泄漏 你必须释放它　参见我的文章Counting objects里面关于placement delete的注释

Arrays

到目前为止一切顺利　但是还得接着走　到目前为止我们所测试的都是一次建立一个对象　怎样分配数组　会发生什么

```
string *ps = new string[10];                  // allocate an array of
                                              // objects
```

被使用的new仍然是new操作符　但是建立数组时new操作符的行为与单个对象建立有少许不同 第一是内存不再用operator new分配 代替以等同的数组分配函数叫做operator new[]　经常被称为array new　　它与operator new一样能被重载　这就允许你控制数组的内存分配 就象你能控制单个对象内存分配一样　但是有一些限制性说明　参见Effective C++ 条款8

operator new[]对于C++来说是一个比较新的东西　所以你的编译器可能不支持它　如果它不支持　无论在数组中的对象类型是什么　全局operator new将被用来给每个数组分配内存 在这样的编译器下定制数组内存分配是困难的 因为它需要重写全局operator new　这可不是一个能轻易接受的任务　缺省情况下　全局

operator new处理程序中所有的动态内存分配 所以它行为的任何改变都将有深入和普遍的影响 而且全局operator new有一个正常的签名 normal signature (也就单一的参数size_t 参见Effective C++条款9) 所以如果你 决定用自己的方法声明它 你立刻使你的程序与其它库不兼容 参见条款27 基于这些考虑 在缺乏operator new[]支持的编译器里为数组定制内存管理不是一个合理的设计
第二个不同是new操作符调用构造函数的数量 对于数组 在数组里的每一个对象的构造函数都必须被调用

```
string *ps =                    // 调用operator new[]为10个
   new string[10];              // string对象分配内存,
                                // 然后对每个数组元素调用
                                // string对象的缺省构造函数
```

同样当delete操作符用于数组时 它为每个数组元素调用析构函数 然后调用operator delete来释放内存
就象你能替换或重载operator delete一样 你也替换或重载operator delete[] 在它们重载的方法上有一些限制 请参考优秀的C++教材 有关优秀的C++教材的信息参见本书285页的推荐
new和delete操作符是内置的 其行为不受你的控制 凡是它们调用的内存分配和释放函数则可以控制 当你想定制new和delete操作符的行为时 请记住你不能真的做到这一点 你只能改变它们为完成它们的功能所采取的方法 而它们所完成的功能则被语言固定下来 不能改变 You can modify how they do what they do, but what they do is fixed by the language

异常

C++新增的异常 exception 机制改变了某些事情 这种改变是深刻的 彻底的可能是令人不舒服的 例如使用未经处理的或原始的指针变得很危险 资源泄漏的可能性增加了 写出具有你希望的行为的构造函数与析构函数变得更加困难特别小心防止程序执行时突然崩溃 执行程序和库程序尺寸增加了同时运行速度减少了
这就使我们所知道的事情 很多使用C++的人都不知道在程序中使用异常 大多数人不知道如何正确使用它 在异常被抛出后 使软件的行为具有可预测性和可靠性 在众多方法中至今也没有一个一致的方法能做到这点 为了深刻了解这个问题 参见Tom Cargill写的Exception Handling: A False Sense of Security 有关这些问题的进展情况的信息 参见Jack Reeves 写的Coping with Exceptions和Herb Sutter写的Exception-Safe Generic Containers
我们知道 程序能够在存在异常的情况下正常运行是因为它们按照要求进行了设计 而不是因为巧合 异常安全 Exception-safe 的程序不是偶然建立的 一个

没有按照要求进行设计的程序在存在异常的情况下运行正常的概率与一个没有按照多线程要求进行设计的程序在多线程的环境下运行正常的概率相同 概率为0

为什么使用异常呢 自从C语言被发明初来 C程序员就满足于使用错误代码 Error code 所以为什么还要弄来异常呢 特别是如果异常如我上面所说的那样存在着问题 答案是简单的 异常不能被忽略 如果一个函数通过设置一个状态变量或返回错误代码来表示一个异常状态 没有办法保证函数调用者将一定检测变量或测试错误代码 结果程序会从它遇到的异常状态继续运行 异常没有被捕获 程序立即会终止执行

C程序员能够仅通过setjmp和longjmp来完成与异常处理相似的功能 但是当longjmp在C++中使用时 它存在一些缺陷 当它调整堆栈时不能对局部对象调用析构函数 而大多数C程序员依赖于这些析构函数的调用 所以setjmp和longjmp不能够替换异常处理 如果你需要一个方法 能够通知异常状态 又不能忽略这个通知 并且搜索栈空间 searching the stack 以便找到异常处理代码时 你还得确保局部对象的析构函数必须被调用 这时你就需要使用C++的异常处理

因为我们已经对使用异常处理的程序设计有了很多了解 下面这些条款仅是一个对于写出异常安全 Exception-safe 软件的不完整的指导 然而它们给任何在C++中使用异常处理的人介绍了一些重要思想 通过留意下面这些指导 你能够提高自己软件的正确性 强壮性和高效性 并且你将回避开许多在使用异常处理时经常遇到的问题

条款9 使用析构函数防止资源泄漏

对指针说再见 必须得承认你永远都不会喜欢使用指针

Ok 你不用对所有的指针说再见 但是你需要对用来操纵局部资源 local resources 的指针说再见 假设 你正在为一个小动物收容所编写软件 小动物收容所是一个帮助小狗小猫寻找主人的组织 每天收容所建立一个文件 包含当天它所管理的收容动物的资料信息 你的工作是写一个程序读出这些文件然后对每个收容动物进行适当的处理 appropriate processing

完成这个程序一个合理的方法是定义一个抽象类 ALA "Adorable Little Animal") 然后为小狗和小猫建立派生类 一个虚拟函数processAdoption分别对各个种类的动物进行处理

```
 class ALA {
public:
   virtual void processAdoption() = 0;
   ...
};
class Puppy: public ALA {
public:
   virtual void processAdoption();
```

```
    ...
};
class Kitten: public ALA {
public:
    virtual void processAdoption();
    ...
};
```

你需要一个函数从文件中读去信息　然后根据文件中的信息产生一个puppy　小狗 对象或者kitten(小猫)对象 这个工作非常适合于虚拟构造器 virtual constructor 在条款25详细描述了这种函数　为了完成我们的目标　我们这样声明函数

```
// 从s中读去动物信息, 然后返回一个指针
// 指向新建立的某种类型对象
ALA * readALA(istream& s);
```

你的程序的关键部分就是这个函数　如下所示

```
void processAdoptions(istream& dataSource)
{
    while (dataSource) {                        // 还有数据时,继续循环
        ALA *pa = readALA(dataSource);          //得到下一个动物
        pa->processAdoption();                  //处理收容动物
        delete pa;                                  //删除readALA返回的对象
    }
}
```

这个函数循环遍历dataSource内的信息　处理它所遇到的每个项目　唯一要记住的一点是在每次循环结尾处删除ps 这是必须的 因为每次调用readALA都建立一个堆对象　如果不删除对象　循环将产生资源泄漏

现在考虑一下　如果pa->processAdoption抛出了一个异常　将会发生什么 processAdoptions没有捕获异常　所以异常将传递给processAdoptions的调用者　转递中　processAdoptions函数中的调用pa->processAdoption语句后的所有语句都被跳过 这就是说pa没有被删除　结果　任何时候pa->processAdoption抛出一个异常都会导致processAdoptions内存泄漏

堵塞泄漏很容易

```
void processAdoptions(istream& dataSource)
{
    while (dataSource) {
        ALA *pa = readALA(dataSource);
    try {
            pa->processAdoption();
```

```
    }
    catch (...) {                    // 捕获所有异常
        delete pa;                   // 避免内存泄漏
                                     // 当异常抛出时
        throw;                       // 传送异常给调用者
    }
    delete pa;                       // 避免资源泄漏
}                                    // 当没有异常抛出时
}
```

但是你必须用try和catch对你的代码进行小改动 更重要的是你必须写双份清除代码 一个为正常的运行准备 一个为异常发生时准备 在这种情况下 必须写两个delete代码 象其它重复代码一样 这种代码写起来令人心烦又难于维护 而且它看上去好像存在着问题 不论我们是让processAdoptions正常返回还是抛出异常 我们都需要删除pa 所以为什么我们必须要在多个地方编写删除代码呢

我们可以把总被执行的清除代码放入processAdoptions函数内的局部对象的析构函数里 这样可以避免重复书写清除代码 因为当函数返回时局部对象总是被释放 无论函数是如何退出的 仅有一种例外就是当你调用longjmp时 Longjmp的这个缺点是C++率先支持异常处理的主要原因

具体方法是用一个对象代替指针pa 这个对象的行为与指针相似 当pointer-like 类指针 对象被释放时 我们能让它的析构函数调用delete 替代指针的对象被称为smart pointers 灵巧指针 参见条款28的解释 你能使得pointer-like对象非常灵巧 在这里 我们用不着这么聪明的指针 我们只需要一个pointer-lik对象当它离开生存空间时知道删除它指向的对象

写出这样一个类并不困难 但是我们不需要自己去写 标准C++库函数包含一个类模板 叫做auto_ptr 这正是我们想要的 每一个auto_ptr类的构造函数里 让一个指针指向一个堆对象 heap object 并且在它的析构函数里删除这个对象

下面所示的是auto_ptr类的一些重要的部分

```
template<class T>
class auto_ptr {
public:
    auto_ptr(T *p = 0): ptr(p) {}            // 保存ptr 指向对象
    ~auto_ptr() { delete ptr; }              // 删除ptr指向的对象
private:
    T *ptr;                                  // raw ptr to object
};
```

auto_ptr类的完整代码是非常有趣的 上述简化的代码实现不能在实际中应用 我们至少必须加上拷贝构造函数 赋值operator和将在条款28讲述的pointer-emulating函数 但是它背后所蕴含的原理应该是清楚的 用auto_ptr对象代替raw指针

你将不再为堆对象不能被删除而担心 即使在抛出异常时 对象也能被及时删除(因为auto_ptr的析构函数使用的是单对象形式的delete 所以auto_ptr不能用于指向对象数组的指针 如果想让auto_ptr类似于一个数组模板 你必须自己写一个 在这种情况下 用vector代替array可能更好)

使用auto_ptr对象代替raw指针 processAdoptions如下所示

```
void processAdoptions(istream& dataSource)
{
   while (dataSource) {
      auto_ptr<ALA> pa(readALA(dataSource));
      pa->processAdoption();
   }
}
```

这个版本的processAdoptions在两个方面区别于原来的processAdoptions函数 第一 pa被声明为一个auto_ptr<ALA>对象 而不是一个raw ALA*指针 第二 在循环的结尾没有delete语句 其余部分都一样 因为除了析构的方式 auto_ptr对象的行为就象一个普通的指针 是不是很容易

隐藏在auto_ptr后的思想是 用一个对象存储需要被自动释放的资源 然后依靠对象的析构函数来释放资源 这种思想不只是可以运用在指针上 还能用在其它资源的分配和释放上 想一下这样一个在GUI程序中的函数 它需要建立一个window来显式一些信息

```
// 这个函数会发生资源泄漏 如果一个异常抛出
void displayInfo(const Information& info)
{
   WINDOW_HANDLE w(createWindow());

   在w对应的window中显式信息

   destroyWindow(w);
}
```

很多window系统有C like接口 使用象like createWindow 和 destroyWindow函数来获取和释放window资源 如果在w对应的window中显示信息时 一个异常被抛出 w所对应的window将被丢失 就象其它动态分配的资源一样

解决方法与前面所述的一样 建立一个类 让它的构造函数与析构函数来获取和释放资源

```
//一个类 获取和释放一个window 句柄
class WindowHandle {
public:
    WindowHandle(WINDOW_HANDLE handle): w(handle) {}
```

```
    ~WindowHandle() { destroyWindow(w); }
      operator WINDOW_HANDLE() { return w; }                // see below
private:
    WINDOW_HANDLE w;
    // 下面的函数被声明为私有  防止建立多个WINDOW_HANDLE拷贝
    //有关一个更灵活的方法的讨论请参见条款28
    WindowHandle(const WindowHandle&);
    WindowHandle& operator=(const WindowHandle&);
};
```

这看上去有些象auto_ptr 只是赋值操作与拷贝构造被显式地禁止 参见条款27

有一个隐含的转换操作能把WindowHandle转换为WINDOW_HANDLE 这个能力对于使用WindowHandle对象非常重要 因为这意味着你能在任何地方象使用raw WINDOW_HANDLE一样来使用WindowHandle 参见条款5 了解为什么你应该谨慎使用隐式类型转换操作

通过给出的WindowHandle类 我们能够重写displayInfo函数 如下所示

```
// 如果一个异常被抛出  这个函数能避免资源泄漏
void displayInfo(const Information& info)
{
    WindowHandle w(createWindow());

    在w对应的window中显式信息;

}
```

即使一个异常在displayInfo内被抛出 被createWindow 建立的window也能被释放

资源应该被封装在一个对象里 遵循这个规则 你通常就能避免在存在异常环境里发生资源泄漏 但是如果你正在分配资源时一个异常被抛出 会发生什么情况呢 例如当你正处于resource-acquiring类的构造函数中 还有如果这样的资源正在被释放时 一个异常被抛出 又会发生什么情况呢 构造函数和析构函数需要特殊的技术 你能在条款10和条款11中获取有关的知识

条款10　在构造函数中防止资源泄漏　上

如果你正在开发一个具有多媒体功能的通讯录程序 这个通讯录除了能存储通常的文字信息如姓名　地址　电话号码外　还能存储照片和声音　可以给出他们名字的正确发音
为了实现这个通信录　你可以这样设计

```
class Image {                                   // 用于图像数据
public:
    Image(const string& imageDataFileName);
    ...
};

class AudioClip {                               // 用于声音数据
public:
    AudioClip(const string& audioDataFileName);
    ...
};

class PhoneNumber {                 ... };      // 用于存储电话号码


class BookEntry {                               // 通讯录中的条目
public:

BookEntry(const string& name,
                const string& address = "",
                const string& imageFileName = "",
                const string& audioClipFileName = "");
~BookEntry();

// 通过这个函数加入电话号码
void addPhoneNumber(const PhoneNumber& number);
...

private:
    string theName;                             // 人的姓名
    string theAddress;                          // 他们的地址
    list<PhoneNumber> thePhones;                // 他的电话号码
```

```
    Image *theImage;                        // 他们的图像
    AudioClip *theAudioClip;                // 他们的一段声音片段
};
```

通讯录的每个条目都有姓名数据 所以你需要带有参数的构造函数 参见条款3 不过其它内容 地址 图像和声音的文件名 都是可选的 注意应该使用链表类 list 存储电话号码 这个类是标准C++类库 STL 中的一个容器类 container classes 参见Effective C++条款49 和本书条款35

编写BookEntry 构造函数和析构函数 有一个简单的方法是

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     Const string& audioClipFileName)
: theName(name), theAddress(address),
    theImage(0), theAudioClip(0)
{
    if (imageFileName != "") {
        theImage = new Image(imageFileName);
    }

if (audioClipFileName != "") {
theAudioClip = new AudioClip(audioClipFileName);
    }
}

BookEntry::~BookEntry()
{
    delete theImage;
    delete theAudioClip;
}
```

构造函数把指针theImage和theAudioClip初始化为空 然后如果其对应的构造函数参数不是空 就让这些指针指向真实的对象 析构函数负责删除这些指针 确保BookEntry对象不会发生资源泄漏 因为C++确保删除空指针是安全的 所以BookEntry的析构函数在删除指针前不需要检测这些指针是否指向了某些对象

看上去好像一切良好 在正常情况下确实不错 但是在非正常情况下 例如在有异常发生的情况下 它们恐怕就不会良好了

请想一下如果BookEntry的构造函数正在执行中 一个异常被抛出 会发生什么情况呢

```
if (audioClipFileName != "") {
```

```
    theAudioClip = new AudioClip(audioClipFileName);
}
```
一个异常被抛出　可以是因为operator new　参见条款8　不能给AudioClip分配足够的内存　也可以因为AudioClip的构造函数自己抛出一个异常　不论什么原因　如果在BookEntry构造函数内抛出异常　这个异常将传递到建立BookEntry对象的地方　在构造函数体的外面　　译者注

现在假设建立theAudioClip对象建立时　一个异常被抛出　而且传递程序控制权到BookEntry构造函数的外面　　那么谁来负责删除theImage已经指向的对象呢　答案显然应该是由BookEntry来做　但是这个想当然的答案是错的　BookEntry根本不会被调用　永远不会

C++仅仅能删除被完全构造的对象　fully contructed objects　，只有一个对象的构造函数完全运行完毕　这个对象才能被完全地构造　所以如果一个BookEntry对象b做为局部对象建立　如下

```
void testBookEntryClass()
{
    BookEntry b("Addison-Wesley Publishing Company",
                "One Jacob Way, Reading, MA 01867");

...

}
```
并且在构造b的过程中　一个异常被抛出　b的析构函数不会被调用　而且如果你试图采取主动手段处理异常情况　即当异常发生时调用delete　如下所示

```
void testBookEntryClass()
{
    BookEntry *pb = 0;

    try {
        pb = new BookEntry("Addison-Wesley Publishing Company",
                           "One Jacob Way, Reading, MA 01867");
        ...
    }
    catch (...) {                    // 捕获所有异常

        delete pb;                   // 删除pb,当抛出异常时

        throw;                       // 传递异常给调用者
    }
```

```
    delete pb;                          // 正常删除pb
}
```

你会发现在BookEntry构造函数里为Image分配的内存仍旧被丢失了 这是因为如果new操作没有成功完成 程序不会对pb进行赋值操作 如果BookEntry的构造函数抛出一个异常 pb将是一个空值 所以在catch块中删除它除了让你自己感觉良好以外没有任何作用 用灵巧指针 smart pointer 类auto_ptr<BookEntry> 参见条款9代替raw BookEntry*也不会也什么作用 因为new操作成功完成前 也没有对pb进行赋值操作

C++拒绝为没有完成构造操作的对象调用析构函数是有一些原因的 而不是故意为你制造困难 原因是 在很多情况下这么做是没有意义的 甚至是有害的 如果为没有完成构造操作的对象调用析构函数 析构函数如何去做呢 仅有的办法是在每个对象里加入一些字节来指示构造函数执行了多少步 然后让析构函数检测这些字节并判断该执行哪些操作 这样的记录会减慢析构函数的运行速度并使得对象的尺寸变大 C++避免了这种开销 但是代价是不能自动地删除被部分构造的对象 类似这种在程序行为与效率这间进行折衷处理的例子还可以参见Effective C++条款13

因为当对象在构造中抛出异常后C++不负责清除对象 所以你必须重新设计你的构造函数以让它们自己清除 经常用的方法是捕获所有的异常 然后执行一些清除代码 最后再重新抛出异常让它继续转递 如下所示 在BookEntry构造函数中使用这个方法

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  try {                                 // 这try block是新加入的
    if (imageFileName != "") {
      theImage = new Image(imageFileName);
    }

  if (audioClipFileName != "") {
      theAudioClip = new AudioClip(audioClipFileName);
    }
  }
  catch (...) {                         // 捕获所有异常
```

```
        delete theImage;                    // 完成必要的清除代码
        delete theAudioClip;


        throw;                              // 继续传递异常
    }
}
```

不用为BookEntry中的非指针数据成员操心　在类的构造函数被调用之前数据成员就被自动地初始化　所以如果BookEntry构造函数体开始执行　对象的theName, theAddress 和 thePhones数据成员已经被完全构造好了 这些数据可以被看做是完全构造的对象　所以它们将被自动释放　不用你介入操作　当然如果这些对象的构造函数调用可能会抛出异常的函数　那么哪些构造函数必须去考虑捕获异常在允许它们继续传递之前完成必需的清除操作

你可能已经注意到BookEntry构造函数的catch块中的语句与在BookEntry的析构函数的语句几乎一样　这里的代码重复是绝对不可容忍的　所以最好的方法是把通用代码移入一个私有helper function中　让构造函数与析构函数都调用它

```
class BookEntry {
public:
    ...                                 // 同上

private:
    ...
    void cleanup();                 // 通用清除代码
};

void BookEntry::cleanup()
{
    delete theImage;
    delete theAudioClip;
}

BookEntry::BookEntry(const string& name,
                     const string& address,
                      const string& imageFileName,
```

```
                                    const string& audioClipFileName)
: theName(name), theAddress(address),
   theImage(0), theAudioClip(0)
{
   try {
      ...                              // 同上
   }
   catch (...)     {
      cleanup();                    // 释放资源
      throw;                         // 传递异常
   }
}

BookEntry::~BookEntry()
{
   cleanup();
}
```

这就行了　但是它没有考虑到下面这种情况　假设我们略微改动一下设计　让theImage 和theAudioClip是常量　constant　指针类型

```
class BookEntry {
public:
   ...                                                  // 同上

private:
   ...
   Image * const theImage;                      // 指针现在是
   AudioClip * const theAudioClip;          // const类型
};
```

必须通过BookEntry构造函数的成员初始化表来初始化这样的指针　因为再也没有其它地方可以给const指针赋值　参见Effective C++条款12　　通常会这样初始化theImage和theAudioClip

```
// 一个可能在异常抛出时导致资源泄漏的实现方法
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
```

```
                                const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(imageFileName != ""
           ? new Image(imageFileName)
           : 0),
  theAudioClip(audioClipFileName != ""
               ? new AudioClip(audioClipFileName)
               : 0)
{}
```

这样做导致我们原先一直想避免的问题重新出现　如果theAudioClip初始化时一个异常被抛出　theImage所指的对象不会被释放　而且我们不能通过在构造函数中增加try和catch 语句来解决问题　因为try和catch是语句　而成员初始化表仅允许有表达式　这就是为什么我们必须在 theImage 和 theAudioClip的初始化中使用?:以代替if-then-else的原因

无论如何　在异常传递之前完成清除工作的唯一的方法就是捕获这些异常　所以如果我们不能在成员初始化表中放入try和catch语句　我们把它们移到其它地方　一种可能是在私有成员函数中　用这些函数返回指针　指向初始化过的theImage和 theAudioClip对象

```
class BookEntry {
public:
  ...                           // 同上

private:
  ...                           // 数据成员同上

Image * initImage(const string& imageFileName);
  AudioClip * initAudioClip(const string&
                                audioClipFileName);
};

BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(initImage(imageFileName)),
```

```
    theAudioClip(initAudioClip(audioClipFileName))
{}

// theImage 被首先初始化,所以即使这个初始化失败也
// 不用担心资源泄漏 这个函数不用进行异常处理
Image * BookEntry::initImage(const string& imageFileName)
{
    if (imageFileName != "") return new Image(imageFileName);
    else return 0;
}

// theAudioClip被第二个初始化, 所以如果在theAudioClip
// 初始化过程中抛出异常 它必须确保theImage的资源被释放
// 因此这个函数使用try...catch
AudioClip * BookEntry::initAudioClip(const string&
                                                audioClipFileName)
{
    try {
        if (audioClipFileName != "") {
            return new AudioClip(audioClipFileName);
        }
        else return 0;
    }
    catch (...)     {
        delete theImage;
        throw;
    }
}
```

上面的程序的确不错 也解决了令我们头疼不已的问题 不过也有缺点 在原则上应该属于构造函数的代码却分散在几个函数里 这令我们很难维护

更好的解决方法是采用条款9的建议 把theImage 和 theAudioClip指向的对象做为一个资源 被一些局部对象管理 这个解决方法建立在这样一个事实基础上 theImage 和theAudioClip是两个指针 指向动态分配的对象 因此当指针消失的时候 这些对象应该被删除 auto_ptr类就是基于这个目的而设计的 参见条款9 因此我们把theImage 和 theAudioClip raw指针类型改成对应的auto_ptr类型

```
class BookEntry {
```

```
public:
  ...                                                    // 同上

private:
  ...
  const auto_ptr<Image> theImage;                  // 它们现在是
  const auto_ptr<AudioClip> theAudioClip;    // auto_ptr对象
};
```

这样做使得BookEntry的构造函数即使在存在异常的情况下也能做到不泄漏资源 而且让我们能够使用成员初始化表来初始化theImage 和 theAudioClip 如下所示

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(imageFileName != ""
           ? new Image(imageFileName)
           : 0),
  theAudioClip(audioClipFileName != ""
               ? new AudioClip(audioClipFileName)
               : 0)
{}
```

在这里 如果在初始化theAudioClip时抛出异常 theImage已经是一个被完全构造的对象 所以它能被自动删除掉 就象theName, theAddress和thePhones一样 而且因为theImage 和 theAudioClip现在是包含在BookEntry中的对象 当BookEntry被删除时它们能被自动地删除 因此不需要手工删除它们所指向的对象 可以这样简化BookEntry的析构函数

```
BookEntry::~BookEntry()
{}                                                       // nothing to do!
```

这表示你能完全去掉BookEntry的析构函数

综上所述 如果你用对应的auto_ptr对象替代指针成员变量 就可以防止构造函数在存在异常时发生资源泄漏 你也不用手工在析构函数中释放资源 并且你还能象以前使用非const指针一样使用const指针 给其赋值

在对象构造中 处理各种抛出异常的可能 是一个棘手的问题 但是auto_ptr(或

者类似于auto_ptr的类)能化繁为简　它不仅把令人不好理解的代码隐藏起来　而且使得程序在面对异常的情况下也能保持正常运行

条款11　禁止异常信息　exceptions　传递到析构函数外

在有两种情况下会调用析构函数　第一种是在正常情况下删除一个对象　例如对象超出了作用域或被显式地delete　第二种是异常传递的堆栈辗转开解
　stack-unwinding　过程中,由异常处理系统删除一个对象
在上述两种情况下　调用析构函数时异常可能处于激活状态也可能没有处于激活状态　遗憾的是没有办法在析构函数内部区分出这两种情况　因此在写析构函数时你必须保守地假设有异常被激活　因为如果在一个异常被激活的同时　析构函数也抛出异常　并导致程序控制权转移到析构函数外　C++将调用terminate函数　这个函数的作用正如其名字所表示的　它终止你程序的运行　而且是立即终止　甚至连局部对象都没有被释放
下面举一个例子　一个Session类用来跟踪在线计算机的sessions　session就是运行在从你一登录计算机开始一直到注销出系统为止的这段期间的某种东西　每个Session对象关注的是它建立与释放的日期与时间

```
class Session {
public:
   Session();
   ~Session();
   ...

private:
   static void logCreation(Session *objAddr);
   static void logDestruction(Session *objAddr);
};
```

函数logCreation 和 logDestruction被分别用于记录对象的建立与释放　我们因此可以这样编写Session的析构函数

```
Session::~Session()
{
   logDestruction(this);
}
```

一切看上去很好　但是如果logDestruction抛出一个异常　会发生什么事呢　异常没有被Session的析构函数捕获住　所以它被传递到析构函数的调用者那里　但是如果析构函数本身的调用就是源自于某些其它异常的抛出　那么terminate函数将被自动调用　彻底终止你的程序　这不是你所希望发生的事情　程序没有记录下释放对象的信息　这是不幸的　甚至是一个大麻烦　那么事态果真严重到了必须终止程序运行的地步了么　如果没有　你必须防止在logDestruction内抛出的异常传递到Session析构函数的外面　唯一的方法是用try和catch blocks　一种很自然的做法会这样编写函数

```
Session::~Session()
{
   try {
      logDestruction(this);
   }
   catch (...) {
      cerr   << "Unable to log destruction of Session object "
             << "at address "
             << this
             << ".\n";
   }
}
```

但是这样做并不比你原来的代码安全 如果在catch中调用operator<<时导致一个异常被抛出 我们就又遇到了老问题 一个异常被转递到Session析构函数的外面 我们可以在catch中放入try 但是这总得有一个限度 否则会陷入循环 因此我们在释放Session时必须忽略掉所有它抛出的异常

```
Session::~Session()
{
   try {
      logDestruction(this);
   }
   catch (...) {   }
}
```

catch表面上好像没有做任何事情 这是一个假象 实际上它阻止了任何从logDestruction抛出的异常被传递到session析构函数的外面 我们现在能高枕无忧了 无论session对象是不是在堆栈辗转开解 stack unwinding 中被释放 terminate函数都不会被调用

不允许异常传递到析构函数外面还有第二个原因 如果一个异常被析构函数抛出而没有在函数内部捕获住 那么析构函数就不会完全运行 它会停在抛出异常的那个地方上 如果析构函数不完全运行 它就无法完成希望它做的所有事情 例如 我们对session类做一个修改 在建立session时启动一个数据库事务 database transaction ,终止session时结束这个事务

```
Session::Session()                  // 为了简单起见 ,
{                                       // 这个构造函数没有
                                        // 处理异常
   logCreation(this);
   startTransaction();             // 启动 database transaction
}
```

```
Session::~Session()
{
    logDestruction(this);
    endTransaction();               // 结束database transaction
}
```

如果在这里logDestruction抛出一个异常 在session构造函数内启动的transaction就没有被终止 我们也许能够通过重新调整session析构函数内的函数调用顺序来消除问题 但是如果endTransaction也抛出一个异常 我们除了回到使用try和catch外别无选择

综上所述 我们知道禁止异常传递到析构函数外有两个原因 第一能够在异常转递的堆栈辗转开解 stack-unwinding 的过程中 防止terminate被调用 第二它能帮助确保析构函数总能完成我们希望它做的所有事情 如果你仍旧不很信服我所说的理由 可以去看Herb Sutter的文章Exception-Safe Generic Containers 特别是“Destructors That Throw and Why They’re Evil”这段

条款12　理解“抛出一个异常”与“传递一个参数”或“调用一个虚函数”间的差异

从语法上看　在函数里声明参数与在catch子句中声明参数几乎没有什么差别

```
class Widget { ... };                    //一个类　具体是什么类
                                                  //  在这里并不重要
void f1(Widget w);                         //  一些函数　其参数分别为
void f2(Widget& w);                        // Widget, Widget&,或
void f3(const Widget& w);                  // Widget*  类型
void f4(Widget *pw);
void f5(const Widget *pw);
catch (Widget w) ...                       //一些catch  子句　用来
catch (Widget& w)      ...                 //捕获异常　异常的类型为
catch (const Widget& w) ...                // Widget, Widget&,  或
catch (Widget *pw) ...                     // Widget*
catch (const Widget *pw) ...
```

你因此可能会认为用throw抛出一个异常到catch子句中与通过函数调用传递一个参数两者基本相同　这里面确有一些相同点　但是他们也存在着巨大的差异
让我们先从相同点谈起　你传递函数参数与异常的途径可以是传值　传递引用或传递指针　这是相同的　但是当你传递参数和异常时　系统所要完成的操作过程则是完全不同的　产生这个差异的原因是　你调用函数时　程序的控制权最终还会返回到函数的调用处　但是当你抛出一个异常时　控制权永远不会回到抛出异常的地方
有这样一个函数　参数类型是Widget　并抛出一个Widget类型的异常

```
//  一个函数　从流中读值到Widget中
istream operator>>(istream& s, Widget& w);
void passAndThrowWidget()
{
   Widget localWidget;
   cin >> localWidget;                 //传递localWidget到  operator>>
   throw localWidget;                   //  抛出localWidget异常
}
```

当传递localWidget到函数operator>>里　不用进行拷贝操作　而是把operator>>内的引用类型变量w指向localWidget　任何对w的操作实际上都施加到localWidget上　这与抛出localWidget异常有很大不同　不论通过传值捕获异常还是通过引用捕获　不能通过指针捕获这个异常　因为类型不匹配　都将进行lcalWidget的拷贝操作　也就说传递到catch子句中的是localWidget的拷贝　必须这么做　因为当localWidget离开了生存空间后　其析构函数将被调用　如果把localWidget本身　而不是它的拷

贝　传递给catch子句　这个子句接收到的只是一个被析构了的Widget　一个Widget的“尸体”这是无法使用的 因此C++规范要求被做为异常抛出的对象必须被复制 即使被抛出的对象不会被释放　也会进行拷贝操作　例如如果passAndThrowWidget函数声明localWidget为静态变量　static

```
void passAndThrowWidget()
{
   static Widget localWidget;              // 现在是静态变量　static　;
                                                    //一直存在至程序结束

   cin >> localWidget;                        // 象以前那样运行
   throw localWidget;                          // 仍将对localWidget
}                                                    //进行拷贝操作
```

当抛出异常时仍将复制出localWidget的一个拷贝　这表示即使通过引用来捕获异常　也不能在catch块中修改localWidget　仅仅能修改localWidget的拷贝　对异常对象进行强制复制拷贝 这个限制有助于我们理解参数传递与抛出异常的第二个差异　抛出异常运行速度比参数传递要慢

当异常对象被拷贝时　拷贝操作是由对象的拷贝构造函数完成的　该拷贝构造函数是对象的静态类型　static type　所对应类的拷贝构造函数　而不是对象的动态类型　dynamic type　对应类的拷贝构造函数　比如以下这经过少许修改的passAndThrowWidget

```
class Widget { ... };
class SpecialWidget: public Widget { ... };
void passAndThrowWidget()
{
   SpecialWidget localSpecialWidget;
   ...
   Widget& rw = localSpecialWidget;          // rw 引用SpecialWidget
   throw rw;                                        //它抛出一个类型为Widget
                                                       // 的异常
}
```

这里抛出的异常对象是Widget　即使rw引用的是一个SpecialWidget　因为rw的静态类型　static type　是Widget　而不是SpecialWidget　你的编译器根本没有主要到rw引用的是一个SpecialWidget　编译器所注意的是rw的静态类型　static type　　这种行为可能与你所期待的不一样　但是这与在其他情况下C++中拷贝构造函数的行为是一致的　　不过有一种技术可以让你根据对象的动态类型dynamic type进行拷贝　参见条款25

异常是其它对象的拷贝　这个事实影响到你如何在catch块中再抛出一个异常　比如下面这两个catch块　乍一看好像一样

```
catch (Widget& w)                    // 捕获Widget异常
{
   ...                               // 处理异常
   throw;                            // 重新抛出异常  让它
}                                    // 继续传递
catch (Widget& w)                    // 捕获Widget异常
{
   ...                               // 处理异常
   throw w;                          // 传递被捕获异常的
}                                    // 拷贝
```

这两个catch块的差别在于第一个catch块中重新抛出的是当前捕获的异常 而第二个catch块中重新抛出的是当前捕获异常的一个新的拷贝 如果忽略生成额外拷贝的系统开销 这两种方法还有差异么

当然有 第一个块中重新抛出的是当前异常 current exception ,无论它是什么类型 特别是如果这个异常开始就是做为SpecialWidget类型抛出的 那么第一个块中传递出去的还是SpecialWidget异常 即使w的静态类型 static type 是Widget 这是因为重新抛出异常时没有进行拷贝操作 第二个catch块重新抛出的是新异常类型总是Widget 因为w的静态类型 static type 是Widget 一般来说 你应该用throw

来重新抛出当前的异常 因为这样不会改变被传递出去的异常类型 而且更有效率 因为不用生成一个新拷贝

顺便说一句 异常生成的拷贝是一个临时对象 正如条款19解释的 临时对象能让编译器优化它的生存期 optimize it out of existence 不过我想你的编译器很难这么做 因为程序中很少发生异常 所以编译器厂商不会在这方面花大量的精力

让我们测试一下下面这三种用来捕获Widget异常的catch子句 异常是做为passAndThrowWidgetp抛出的

```
catch (Widget w) ...                 // 通过传值捕获异常
catch (Widget& w) ...                // 通过传递引用捕获
                                     // 异常
catch (const Widget& w) ...          //通过传递指向const的引用
                                     //捕获异常
```

我们立刻注意到了传递参数与传递异常的另一个差异 一个被异常抛出的对象

刚才解释过 总是一个临时对象 可以通过普通的引用捕获 它不需要通过指向const对象的引用 reference-to-const 捕获 在函数调用中不允许转递一个临时对象到一个非const引用类型的参数里 参见条款19 但是在异常中却被允许

让我们先不管这个差异 回到异常对象拷贝的测试上来 我们知道当用传值的方

式传递函数的参数　我们制造了被传递对象的一个拷贝　参见Effective C++ 条款22　并把这个拷贝存储到函数的参数里　同样我们通过传值的方式传递一个异常时　也是这么做的　当我们这样声明一个catch子句时

```
catch (Widget w) ...                    // 通过传值捕获
```

会建立两个被抛出对象的拷贝　一个是所有异常都必须建立的临时对象　第二个是把临时对象拷贝进w中　同样　当我们通过引用捕获异常时

```
catch (Widget& w) ...                   // 通过引用捕获

catch (const Widget& w) ...             //也通过引用捕获
```

这仍旧会建立一个被抛出对象的拷贝　拷贝是一个临时对象　相反当我们通过引用传递函数参数时　没有进行对象拷贝　当抛出一个异常时　系统构造的　以后会析构掉　被抛出对象的拷贝数比以相同对象做为参数传递给函数时构造的拷贝数要多一个

我们还没有讨论通过指针抛出异常的情况　不过通过指针抛出异常与通过指针传递参数是相同的　不论哪种方法都是一个指针的拷贝被传递　你不能认为抛出的指针是一个指向局部对象的指针　因为当异常离开局部变量的生存空间时　该局部变量已经被释放　Catch子句将获得一个指向已经不存在的对象的指针　这种行为在设计时应该予以避免

对象从函数的调用处传递到函数参数里与从异常抛出点传递到catch子句里所采用的方法不同　这只是参数传递与异常传递的区别的一个方面　第二个差异是在函数调用者或抛出异常者与被调用者或异常捕获者之间的类型匹配的过程不同　比如在标准数学库　the standard math library　中sqrt函数

```
double sqrt(double);                    // from <cmath> or <math.h>
```

我们能这样计算一个整数的平方根　如下所示

```
int i;

double sqrtOfi = sqrt(i);
```

毫无疑问　C++允许进行从int到double的隐式类型转换　所以在sqrt的调用中　i 被悄悄地转变为double类型　并且其返回值也是double　有关隐式类型转换的详细讨论参见条款5　一般来说　catch子句匹配异常类型时不会进行这样的转换　见下面的代码

```
void f(int value)
{
   try {
      if (someFunction()) {          // 如果 someFunction()返回
         throw value;                //真　抛出一个整形值
      ...
      }
```

```
    }
    catch (double d) {                  //  只处理double类型的异常
        ...
    }

    ...

}
```

在try块中抛出的int异常不会被处理double异常的catch子句捕获　该子句只能捕获真真正正为double类型的异常　不进行类型转换　因此如果要想捕获int异常　必须使用带有int或int&参数的catch子句

不过在catch子句中进行异常匹配时可以进行两种类型转换　第一种是继承类与基类间的转换　一个用来捕获基类的catch子句也可以处理派生类类型的异常　例如在标准C++库　STL　定义的异常类层次中的诊断部分　diagnostics portion　(参见 Effective C++　条款49)

捕获runtime_errors异常的Catch子句可以捕获range_error类型和overflow_error类型的异常　可以接收根类exception异常的catch子句能捕获其任意派生类异常

这种派生类与基类　inheritance_based　间的异常类型转换可以作用于数值　引用以及指针上

```
catch (runtime_error) ...                   // can catch errors of type
catch (runtime_error&) ...                  // runtime_error,
catch (const runtime_error&) ...            // range_error, or
                                            // overflow_error
catch (runtime_error*) ...                  // can catch errors of type
catch (const runtime_error*) ...            // runtime_error*,
                                            // range_error*, or
                                            // overflow_error*
```

第二种是允许从一个类型化指针　typed pointer　转变成无类型指针　untyped pointer　　所以带有const void*　指针的catch子句能捕获任何类型的指针类型异常

```
catch (const void*) ...                     //捕获任何指针类型异常
```

传递参数和传递异常间最后一点差别是catch子句匹配顺序总是取决于它们在程序中出现的顺序　因此一个派生类异常可能被处理其基类异常的catch子句捕获　即使同时存在有能处理该派生类异常的catch子句　与相同的try块相对应　例如

```
try {
    ...
}
```

```
catch (logic_error& ex) {                    // 这个catch块 将捕获
    ...                                              // 所有的logic_error
}                                                    // 异常, 包括它的派生类

catch (invalid_argument& ex) {               // 这个块永远不会被执行
    ...                                              //因为所有的
}                                                    // invalid_argument
                                                     // 异常 都被上面的
                                                     // catch子句捕获
```

与上面这种行为相反 当你调用一个虚拟函数时 被调用的函数位于与发出函数调用的对象的动态类型 dynamic type 最相近的类里 你可以这样说虚拟函数采用最优适合法 而异常处理采用的是最先适合法 如果一个处理派生类异常的catch子句位于处理基类异常的catch子句前面 编译器会发出警告 因为这样的代码在C++里通常是不合法的 不过你最好做好预先防范 不要把处理基类异常的catch子句放在处理派生类异常的catch子句的前面 象上面那个例子 应该这样去写

```
try {
    ...
}
catch (invalid_argument& ex) {               // 处理 invalid_argument
    ...                                              //异常
}
catch (logic_error& ex) {                    // 处理所有其它的
    ...                                              // logic_errors异常
}
```

综上所述 把一个对象传递给函数或一个对象调用虚拟函数与把一个对象做为异常抛出 这之间有三个主要区别 第一 异常对象在传递时总被进行拷贝 当通过传值方式捕获时 异常对象被拷贝了两次 对象做为参数传递给函数时不需要被拷贝 第二 对象做为异常被抛出与做为参数传递给函数相比 前者类型转换比后者要少 前者只有两种转换形式 最后一点 catch子句进行异常类型匹配的顺序是它们在源代码中出现的顺序 第一个类型匹配成功的catch将被用来执行 当一个对象调用一个虚拟函数时 被选择的函数位于与对象类型匹配最佳的类里 即使该类不是在源代码的最前头

条款13　通过引用　reference　捕获异常

当你写一个catch子句时　必须确定让异常通过何种方式传递到catch子句里　你可以有三个选择　与你给函数传递参数一样　通过指针　by pointer　　通过传值　by value　或通过引用　by reference
我们首先讨论通过指针方式捕获异常　catch by pointer　　从throw处传递一个异常到catch子句是一个缓慢的过程　在理论上这种方法的实现对于这个过程来说是效率最高的　因为在传递异常信息时　只有采用通过指针抛出异常的方法才能够做到不拷贝对象　参见条款12　　例如

```
class exception { ... };                  // 来自标准C++库　STL
                                                  // 中的异常类层次
                                                  //　参见条款12

void someFunction()
{
   static exception ex;                   // 异常对象

...


   throw &ex;                                    // 抛出一个指针　指向ex

   ...

}

void doSomething()
{
   try {
      someFunction();                        // 抛出一个 exception*
   }
   catch (exception *ex) {                // 捕获 exception*;
      ...                                          // 没有对象被拷贝
   }
}
```

这看上去很不错　但是实际情况却不是这样　为了能让程序正常运行　程序员定义异常对象时必须确保当程序控制权离开抛出指针的函数后　对象还能够继续生存　全局与静态对象都能够做到这一点　但是程序员很容易忘记这个约束　如果

真是如此的话　他们会这样写代码

```
void someFunction()
{
   exception ex;                          // 局部异常对象;
                                          // 当退出函数的生存空间时
                                          // 这个对象将被释放
   ...


   throw &ex;                             // 抛出一个指针　指向
   ...                                    // 已被释放的对象
}
```

这简直糟糕透了　因为处理这个异常的catch子句接受到的指针　其指向的对象已经不再存在

另一种抛出指针的方法是在建立一个堆对象　new heap object

```
void someFunction()
{
   ...
   throw new exception;                   // 抛出一个指针　指向一个在堆中
   ...                                    // 建立的对象(希望
}                                         // 操作符new — 参见条款8—
                                          // 自己不要再抛出一个
                                          // 异常!)
```

这避免了捕获一个指向已被释放对象的指针的问题　但是catch子句的作者又面临一个令人头疼的问题　他们是否应该删除他们接受的指针　如果是在堆中建立的异常对象　那他们必须删除它　否则会造成资源泄漏　如果不是在堆中建立的异常对象　他们绝对不能删除它　否则程序的行为将不可预测　该如何做呢

这是不可能知道的　一些clients可能会传递全局或静态对象的地址　另一些可能转递堆中建立的异常对象的地址　通过指针捕获异常　将遇到一个哈姆雷特式的难题　是删除还是不删除　这是一个难以回答的问题　所以你最好避开它

而且　通过指针捕获异常也不符合C++语言本身的规范　四个标准的异常－－bad_alloc(当operator new(参见条款8)不能分配足够的内存时　被抛出)　bad_cast　当dynamic_cast针对一个引用　reference　操作失败时　被抛出　,bad_typeid　当dynamic_cast对空指针进行操作时　被抛出　和bad_exception　用于unexpected异常　参见条款14　－－都不是指向对象的指针　所以你必须通过值或引用来捕获它们

通过值捕获异常　catch-by-value　可以解决上述的问题　例如异常对象删除的问题和使用标准异常类型的问题　但是当它们被抛出时系统将对异常对象拷贝两次　参见条款12　　而且它会产生slicing problem　即派生类的异常对象被做为基类

异常对象捕获时　那它的派生类行为就被切掉了　sliced off　　这样的sliced对象实际上是一个基类对象　它们没有派生类的数据成员　而且当调用它们的虚拟函数时　系统解析后调用的是基类对象的函数　　当一个对象通过传值方式传递给函数　也会发生一样的情况－－参见Effective C++ 条款22　　例如下面这个程序采用了扩展自标准异常类的异常类层次体系

```
class exception {                          //  如上　这是
public:                                      //  一个标准异常类

virtual const char * what() throw();
                                               //  返回异常的简短描述.
   ...                                        //     在函数声明的结尾处
                                               //  的"throw()"
};                                           //有关它的信息
                                               //  参见条款14)

class runtime_error:                       //也来自标准C++异常类
   public exception { ... };

class Validation_error:                    //  客户自己加入个类
   public runtime_error {
public:
   virtual const char * what() throw();
                                               //  重新定义在异常类中
   ...                                        //虚拟函数
};                                           //

void someFunction()                        //  抛出一个  validation
{                                              //  异常
   ...

   if (a validation  测试失败) {
      throw Validation_error();
   }

...

}
```

```
void doSomething()
{
   try {
      someFunction();                    //  抛出  validation
   }                                        //异常


   catch (exception ex) {                //捕获所有标准异常类
                                            //  或它的派生类




      cerr << ex.what();                 //  调用  exception::what(),
      ...                                   //  而不是Validation_error::what()
}
}
```

调用的是基类的what函数　即使被抛出的异常对象是Validation_error和Validation_error类型　它们已经重新定义的虚拟函数　这种slicing行为绝不是你所期望的

最后剩下方法就是通过引用捕获异常　catch-by-reference　　通过引用捕获异常能使你避开上述所有问题　不象通过指针捕获异常　这种方法不会有对象删除的问题而且也能捕获标准异常类型　也不象通过值捕获异常　这种方法没有slicing problem　而且异常对象只被拷贝一次

我们采用通过引用捕获异常的方法重写最后那个例子　如下所示

```
void someFunction()                     //这个函数没有改变
{
   ...

   if (a validation  测试失败) {
      throw Validation_error();
   }

   ...

}
```

```
void doSomething()
{
   try {
      someFunction();                          // 没有改变
   }
   catch (exception& ex) {                    // 这里   我们通过引用捕获异常
                                                 // 以替代原来的通过值捕获


      cerr << ex.what();                       // 现在调用的是
                                                    // Validation_error::what(),
      ...                                          // 而不是 exception::what()
   }
}
```

这里没有对throw进行任何改变　仅仅改变了catch子句　给它加了一个&符号　然而这个微小的改变能造成了巨大的变化 因为catch块中的虚拟函数能够如我们所愿那样工作了　调用的Validation_erro函数是我们重新定义过的函数

如果你通过引用捕获异常　catch by reference　　你就能避开上述所有问题　不会为是否删除异常对象而烦恼　能够避开slicing异常对象　能够捕获标准异常类型　减少异常对象需要被拷贝的数目　所以你还在等什么　通过引用捕获异常吧

Catch exceptions by reference

条款14　审慎使用异常规格(exception specifications)

毫无疑问　异常规格是一个引人注目的特性　它使得代码更容易理解　因为它明确地描述了一个函数可以抛出什么样的异常　但是它不只是一个有趣的注释　编译器在编译时有时能够检测到异常规格的不一致　而且如果一个函数抛出一个不在异常规格范围里的异常　系统在运行时能够检测出这个错误　然后一个特殊函数unexpected将被自动地调用　异常规格既可以做为一个指导性文档同时也是异常使用的强制约束机制　它好像有着很诱人的外表
不过在通常情况下　美貌只是一层皮　外表的美丽并不代表其内在的素质　函数unexpected缺省的行为是调用函数terminate　而terminate缺省的行为是调用函数abort　所以一个违反异常规格的程序其缺省的行为就是halt　停止运行　　在激活的stack frame中的局部变量没有被释放 因为abort在关闭程序时不进行这样的清除操作　对异常规格的触犯变成了一场并不应该发生的灾难
不幸的是　我们很容易就能够编写出导致发生这种灾难的函数　编译器仅仅部分地检测异常的使用是否与异常规格保持一致　一个函数调用了另一个函数　并且后者可能抛出一个违反前者异常规格的异常　　A函数调用B函数　因为B函数可能抛出一个不在A函数异常规格之内的异常 所以这个函数调用就违反了A函数的异常规格　译者注　编译器不对此种情况进行检测　并且语言标准也禁止它们拒绝这种调用方式　尽管可以显示警告信息
例如函数f1没有声明异常规格　这样的函数就可以抛出任意种类的异常

```
extern void f1();                    // 可以抛出任意的异常
```

假设有一个函数f2通过它的异常规格来声明其只能抛出int类型的异常

```
void f2() throw(int);
```

f2调用f1是非常合法的　即使f1可能抛出一个违反f2异常规格的异常

```
void f2() throw(int)
{
  ...
  f1();                          // 即使f1可能抛出不是int类型的
                                 //异常　这也是合法的
  ...
}
```

当带有异常规格的新代码与没有异常规格的老代码整合在一起工作时 这种灵活性就显得很重要
因为你的编译器允许你调用一个函数其抛出的异常与发出调用的函数的异常规格不一致　并且这样的调用可能导致你的程序执行被终止　所以在编写软件时采取措施把这种不一致减小到最少　一种好方法是避免在带有类型参数的模板内使用异常规格　例如下面这种模板　它好像不能抛出任何异常

```
// a poorly designed template wrt exception specifications
```

```
template<class T>
bool operator==(const T& lhs, const T& rhs) throw()
{
   return &lhs == &rhs;
}
```

这个模板为所有类型定义了一个操作符函数operator== 对于任意一对类型相同的对象 如果对象有一样的地址 该函数返回true 否则返回false

这个模板包含的异常规格表示模板生成的函数不能抛出异常 但是事实可能不会这样 因为opertor&(地址操作符,参见Effective C++ 条款45)能被一些类型对象重载 如果被重载的话 当调用从operator==函数内部调用opertor&时 opertor&可能会抛出一个异常 这样就违反了我们的异常规格 使得程序控制跳转到unexpected

上述的例子是一种更一般问题的特例 这个问题也就是没有办法知道某种模板类型参数抛出什么样的异常 我们几乎不可能为一个模板提供一个有意义的异常规格 因为模板总是采用不同的方法使用类型参数 解决方法只能是模板和异常规格不要混合使用

能够避免调用unexpected函数的第二个方法是如果在一个函数内调用其它没有异常规格的函数时应该去除这个函数的异常规格 这很容易理解 但是实际中容易被忽略 比如允许用户注册一个回调函数

```
//  一个window系统回调函数指针
//当一个window系统事件发生时
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack);
//window系统类 含有回调函数指针
//该回调函数能被window系统客户注册
class CallBack {
public:
   CallBack(CallBackPtr fPtr, void *dataToPassBack)
   : func(fPtr), data(dataToPassBack) {}
   void makeCallBack(int eventXLocation,
                     int eventYLocation) const throw();
private:
   CallBackPtr func;                  // function to call when
                                      // callback is made
    void *data;                       // data to pass to callback
};                                    // function
//  为了实现回调函数 我们调用注册函数
//事件的作标与注册数据做为函数参数
```

```
void CallBack::makeCallBack(int eventXLocation,
                            int eventYLocation) const throw()
{
   func(eventXLocation, eventYLocation, data);
}
```

这里在makeCallBack内调用func　要冒违反异常规格的风险　因为无法知道func会抛出什么类型的异常

通过在程序在CallBackPtr typedef中采用更严格的异常规格来解决问题

```
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack) throw();
```

这样定义typedef后　如果注册一个可能会抛出异常的callback函数将是非法的

```
// 一个没有异常给各的回调函数
void callBackFcn1(int eventXLocation, int eventYLocation,
                  void *dataToPassBack);
void *callBackData;
...
CallBack c1(callBackFcn1, callBackData);
                                    //错误　callBackFcn1可能
                                    // 抛出异常
//带有异常规格的回调函数
void callBackFcn2(int eventXLocation,
                  int eventYLocation,
                  void *dataToPassBack) throw();
CallBack c2(callBackFcn2, callBackData);
                                    // 正确　callBackFcn2
                                    // 没有异常规格
```

传递函数指针时进行这种异常规格的检查　是语言的较新的特性　所以有可能你的编译器不支持这个特性　如果它们不支持　那就依靠你自己来确保不能犯这种错误

避免调用unexpected的第三个方法是处理系统本身抛出的异常　这些异常中最常见的是bad_alloc　当内存分配失败时它被operator new　和operator new[]抛出　参见条款8　　如果你在函数里使用new操作符　还参见条款8　　你必须为函数可能遇到bad_alloc异常作好准备

现在常说预防胜于治疗　即做任何事都要未雨绸缪　译者注　　但是有时却是预防困难而治疗容易　也就是说有时直接处理unexpected异常比防止它们被抛出要简单　例如你正在编写一个软件　精确地使用了异常规格　但是你必须从没有使用异常规格的程序库中调用函数　要防止抛出unexpected异常是不现实的　因为

这需要改变程序库中的代码
虽然防止抛出unexpected异常是不现实的　但是C++允许你用其它不同的异常类型替换unexpected异常　你能够利用这个特性　例如你希望所有的unexpected异常都被替换为UnexpectedException对象　你能这样编写代码

```
class UnexpectedException {};                  // 所有的unexpected异常对象被
                                                //替换为这种类型对象


void convertUnexpected()                        // 如果一个unexpected异常被
{                                               // 抛出　这个函数被调用
   throw UnexpectedException();
}
```

通过用convertUnexpected函数替换缺省的unexpected函数　来使上述代码开始运行

```
set_unexpected(convertUnexpected);
```

当你这么做了以后　一个unexpected异常将触发调用convertUnexpected函数　Unexpected异常被一种UnexpectedException新异常类型替换　如果被违反的异常规格包含UnexpectedException异常　那么异常传递将继续下去　好像异常规格总是得到满足　　如果异常规格没有包含UnexpectedException　terminate将被调用　就好像你没有替换unexpected一样
另一种把unexpected异常转变成知名类型的方法是替换unexpected函数　让其重新抛出当前异常　这样异常将被替换为bad_exception　你可以这样编写

```
void convertUnexpected()                // 如果一个unexpected异常被
{                                       //抛出　这个函数被调用
   throw;                               // 它只是重新抛出当前
}                                       // 异常

set_unexpected(convertUnexpected);
                                        // 安装 convertUnexpected
                                        // 做为unexpected
                                        // 的替代品
```

如果这么做　你应该在所有的异常规格里包含bad_exception　或它的基类　标准类exception　　你将不必再担心如果遇到unexpected异常会导致程序运行终止　任何不听话的异常都将被替换为bad_exception　这个异常代替原来的异常继续传递　到现在你应该理解异常规格能导致大量的麻烦　编译器仅仅能部分地检测它们的使用是否一致　在模板中使用它们会有问题　一不注意它们就很容易被违反　并且在缺省的情况下它们被违反时会导致程序终止运行　异常规格还有一个缺点就是它们能导致unexpected被触发即使一个high-level调用者准备处理被抛出的异常

比如下面这个几乎一字不差地来自从条款11例子

```
class Session {                          // for modeling online
public:                                  // sessions
   ~Session();
   ...

private:
   static void logDestruction(Session *objAddr) throw();
};

Session::~Session()
{
   try {
      logDestruction(this);
   }
   catch (...) {    }
}
```

session的析构函数调用logDestruction记录有关session对象被释放的信息　它明确地要捕获从logDestruction抛出的所有异常　但是logDestruction的异常规格表示其不抛出任何异常　现在假设被logDestruction调用的函数抛出了一个异常　而logDestruction没有捕获　我们不会期望发生这样的事情　凡是正如我们所见　很容易就会写出违反异常规格的代码　当这个异常通过logDestruction传递出来　unexpected将被调用　缺省情况下将导致程序终止执行　这是一个正确的行为　这是session析构函数的作者所希望的行为么　作者想处理所有可能的异常　所以好像不应该不给session析构函数里的catch块执行的机会就终止程序　如果logDestruction没有异常规格　这种事情就不会发生　一种防止的方法是如上所描述的那样替换unexpected

以全面的角度去看待异常规格是非常重要的　它们提供了优秀的文档来说明一个函数抛出异常的种类　并且在违反它的情况下　会有可怕的结果　程序被立即终止　在缺省时它们会这么做　同时编译器只会部分地检测它们的一致性　所以他们很容易被不经意地违反　而且他们会阻止high-level异常处理器来处理unexpected异常　即使这些异常处理器知道如何去做　综上所述　异常规格是一个应被审慎使用的公族　在把它们加入到你的函数之前　应考虑它们所带来的行为是否就是你所希望的行为

条款15　了解异常处理的系统开销

为了在运行时处理异常　程序要记录大量的信息　无论执行到什么地方　程序都必须能够识别出如果在此处抛出异常的话　将要被释放哪一个对象　程序必须知道每一个入口点　以便从try块中退出　对于每一个try块　他们都必须跟踪与其相关的catch子句以及这些catch子句能够捕获的异常类型　这种信息的记录不是没有代价的　确保程序满足异常规格不需要运行时的比较　runtime comparisons　而且当异常被抛出时也不用额外的开销来释放相关的对象和匹配正确的catch字句　但是异常处理确是有代价的　即使你没有使用try　throw或catch关键字　你同样得付出一些代价

让我们先从你不使用任何异常处理特性也要付出的代价谈起　你需要空间建立数据结构来跟踪对象是否被完全构造　constructed　(参加条款10)　你也需要系统时间保持这些数据结构不断更新　这些开销一般不是很大　但是当采用不支持异常的方法编译的程序一般比支持异常的程序运行速度更快所占空间也更小

在理论上　你不能对此进行选择　C++编译器必须支持异常　也就是说　当你不用异常处理时你不能让编译器生产商消除这方面的开销　因为程序一般由多个独立生成的目标文件　object files　组成　只有一个目标文件不进行异常处理并不能代表其他目标文件不进行异常处理　而且即使组成可执行文件的目标文件都不进行异常处理　那么还有它们所连接的程序库呢　如果程序的任何部分使用了异常　其它部分必须也支持异常　否则在运行时程序就不可能提供正确的异常处理　不过这只是理论　实际上大部分支持异常的编译器生产商都允许你自由控制是否在生成的代码里包含进支持异常的内容　如果你知道你程序的任何部分都不使用try　throw或catch　并且你也知道所连接的程序库也没有使用try　throw或catch　你就可以采用不支持异常处理的方法进行编译　这可以缩小程序的尺寸和提高速度　否则你就得为一个不需要的特性而付出代价　随着时间的推移　使用异处理的程序库开始变得普遍了　上面这种方法将逐渐不能使用　但是根据目前的软件开发情况来看　如果你已经决定不使用任何的异常特性　那么采用不支持异常的方法编译程序是一个性能优化的合理方法　同样这对于想避开异常的程序库来说也是一个性能优化的好方法　这能保证异常不会从客户端程序传递进程序库里　不过同时这样做也会妨碍客户端程序重定义程序库中声明的虚拟函数　并不允许有在客户端定义的回调函数

使用异常处理的第二个开销来自于try块　无论何时使用它　也就是无论何时你想能够捕获异常　那你都得为此付出代价　不同的编译器实现try块的方法不同　所以编译器与编译器间的开销也不一样　粗略地估计　如果你使用try块　代码的尺寸将增加5　　10　并且运行速度也同比例减慢　这还是假设程序没有抛出异常　我这里讨论的只是在程序里使用try块的开销　为了减少开销　你应该避免使用无用的try块

编译器为异常规格生成的代码与它们为try块生成的代码一样多　所以一个异常规

格一般花掉与tyr块一样多的系统开销 什么 你说你认为异常规格只是一个规格而已 你认为它们不会产生代码 那么好 现在你应该对此有新的认识了

现在我们来到了问题的核心部分 看看抛出异常的开销 事实上我们不用太关心这个问题 因为异常是很少见的 这种事件的发生往往被描述为exceptional 异常的 罕见的 80 20规则 参见条款16 告诉我们这样的事件不会对整个程序的性能造成太大的影响 但是我知道你仍旧好奇地想知道如果抛出一个异常到底会有多大的开销 答案是这可能会比较大 与一个正常的函数返回相比 通过抛出异常从函数里返回可能会慢三个数量级 这个开销很大 但是仅仅当你抛出异常时才会有这个开销 一般不会发生 但是如果你用异常表示一个比较普遍的状况 例如完成对数据结构的遍历或结束一个循环 那你必须重新予以考虑

不过请等一下 你问我是怎么知道这些事情的呢 如果说支持异常对于大多数编译器来说是一个较新的特性 如果说不同的编译器异常方法也不同 那么我如何能说程序的尺寸将增大5%-10% 它的速度也同比例减慢 而且如果有大量的异常被抛出 程序运行速度会呈数量级的减慢呢 答案是令人惊恐的 一些传闻和一些基准测试 benchmarks (参见条款23) 事实是大部分人包括编译器生产商在异常处理方面几乎没有什么经验 所以尽管我们知道异常确实会带来开销 却很难预测出开销的准确数量

谨慎的方法是对本条款所叙述的开销有了解 但是不深究具体的数量 即定性不定量 译者注 不论异常处理的开销有多大我们都得坚持只有必须付出时才付出的原则 为了使你的异常开销最小化 只要可能尽量就采用不支持异常的方法编译程序 把使用try块和异常规格限制在你确实需要它们的地方 并且只有在确为异常的情况下 exceptional 才抛出异常 如果你在性能上仍旧有问题 总体评估一下你的软件以决定异常支持是否是一个起作用的因素 如果是 那就考虑选择其它的编译器 能在C++异常处理方面具有更高实现效率的编译器

效率

我怀疑一些人在C++软件开发人员身上进行秘密的巴甫洛夫试验　否则为什么当提到"效率"这个词时　许多程序员都会流口水　　Scott Meyers真幽默　译者注　事实上　效率可不是一个开玩笑的事情　一个太大或太慢的程序它们的优点无论多么引人注目都不会为人们所接受　本来就应该这样　软件是用来帮助我们更好地工作　说运行速度慢才是更好的　说需要32MB内存的程序比仅仅需要16MB内存的程序好　说占用100MB磁盘空间的程序比仅仅占用50MB磁盘空间的程序好　这简直是无稽之谈　而且尽管有一些程序确是为了进行更复杂的运算才占用更多的时间和空间　但是对于许多程序来说只能归咎于其糟糕的设计和马虎的编程　在用C++写出高效地程序之前　必须认识到C++本身绝对与你所遇到的任何性能上的问题无关　如果想写出一个高效的C++程序　你必须首先能写出一个高效的程序　太多的开发人员都忽视了这个简单的道理　是的　循环能够被手工展开　移位操作　shift operation　能够替换乘法　但是如果你所使用的高层算法其内在效率很低　这些微调就不会有任何作用　当线性算法可用时你是否还用二次方程式算法　你是否一遍又一遍地计算重复的数值　如果是的话　可以毫不夸张地把你的程序比喻成一个二流的观光胜地　即如果你有额外的时间　才值得去看一看　本章的内容从两个角度阐述效率的问题　第一是从语言独立的角度　关注那些你能在任何语言里都能使用的东西　C++为它们提供了特别吸引人的实现途径　因为它对封装的支持非常好　从而能够用更好的算法与数据结构来替代低效的类实现　同时接口可以保持不变

第二是关注C++语言本身　高性能的算法与数据结构虽然非常好　但如果实际编程中代码实现得很粗糙　效率也会降低得相当多　潜在危害性最大的错误是既容易犯又不容易察觉的错误　濒繁地构造和释放大量的对象就是一种这样的错误　过多的对象构造和对象释放对于你的程序性能来说就象是在大出血　在每次建立和释放不需要的对象的过程中　宝贵的时间就这么流走了　这个问题在C++程序中很普遍　我将用四个条款来说明这些对象从哪里来的　在不影响程序代码正确性的基础上如何消除它们

建立大量的对象不会使程序变大而只会使其运行速度变慢　还有其它一些影响性能提高的因素　包括程序库的选择和语言特性的实现　implementations of language features　　在下面的条款中我也将涉及

在学习了本章内容以后　你将熟悉能够提高程序性能的几个原则　这些原则可以适用于你所写的任何程序　你将知道如何准确地防止在你的软件里出现不需要的对象　并且对编译器生成可执行代码的行为有着敏锐的感觉

俗话说有备无患　forewarned is forearmed　　所以把下面的内容想成是战斗前的准备

条款16　牢记80　20准则　80　20 rule

80　20准则说的是大约20　的代码使用了80　的程序资源　大约20%的代码耗用了大约80　的运行时间　大约20　的代码使用了80　的内存　大约20　的代码执行80　的磁盘访问　80　的维护投入于大约20　的代码上　通过无数台机器　操作系统和应用程序上的实验这条准则已经被再三地验证过　80　20准则不只是一条好记的惯用语　它更是一条有关系统性能的指导方针　它有着广泛的适用性和坚实的实验基础

当想到80　20准则时　不要在具体数字上纠缠不清　一些人喜欢更严格的90　10准则　而且也有一些试验证据支持它　不管准确地数字是多少　基本的观点是一样的　软件整体的性能取决于代码组成中的一小部分

当程序员力争最大化提升软件的性能时　80　20准则既简化了你的工作又使你的工作变得复杂　一方面80　20准则表示大多数时间你能够编写性能一般的代码　因为80　的时间里这些代码的效率不会影响到整个系统的性能　这会减少一些你的工作压力　而另一方面这条准则也表示如果你的软件出现了性能问题　你将面临一个困难的工作　因为你不仅必须找到导致问题的那一小块代码的位置　还必须寻找方法提高它们的性能　这些任务中最困难的一般是找到系统瓶颈　基本上有两个不同的方法用来寻找　大多数人用的方法和正确的方法

大多数人寻找瓶颈的方法就是猜　通过经验　直觉　算命纸牌　显灵板　传闻或者其它更荒唐的东西　一个又一个程序员一本正经地宣称程序的性能问题已被找到　因为网络的延迟　不正确的内存分配　编译器没有进行足够的优化或者一些笨蛋主管拒绝在关键的循环里使用汇编语句　这些评估总是以一种带有嘲笑的盛气凌人的架式发布出来　通常这些嘲笑者和他们的预言都是错误的

大多数程序员在他们程序性能特征上的直觉都是错误的　因为程序性能特征往往不能靠直觉来确定　结果为提高程序各部分的效率而倾注了大量的精力　但是对程序的整体行为没有显著的影响　例如在程序里加入能够最小化计算量的奇特算法和数据结构　但是如果程序的性能限制主要在I/O上　I/O-bound　那么就丝毫起不到作用　采用I/O性能强劲的程序库代替编译器本身附加的程序库　参见条款23　　如果程序的性能瓶颈主要在CPU上　CPU-bound　　这种方法也不会起什么作用

在这种情况下　面对运行速度缓慢或占用过多内存的程序　你该如何做呢　80　20准则的含义是胡乱地提高一部分程序的效率不可能有很大帮助　程序性能特征往往不能靠直觉确定　这个事实意味着试图猜出性能瓶颈不可能比胡乱地提高一部分程序的效率这种方法好到哪里去　那么会后什么结果呢

结果是用经验识别程序20　的部分只会导致你心痛　正确的方法是用profiler程序识别出令人讨厌的程序的20　部分　不是所有的工作都让profiler去做　你想让它去直接地测量你感兴趣的资源　例如如果程序太缓慢　你想让profiler告诉你程序的各个部分都耗费了多少时间　然后你关注那些局部效率能够被极大提高的地方　这也将会很大地提高整体的效率

profiler告诉你每条语句执行了多少次或各函数被调用了多少次,这是一个作用有

限的工具　从提高性能的观点来看　你不用关心一条语句或一个函数被调用了多少次　毕竟很少遇到用户或程序库的调用者抱怨执行了太多的语句或调用了太多的函数　如果软件足够快　没有人关心有多少语句被执行　如果程序运行过慢　不会有人关心语句有多么的少　他们所关心的是他们厌恶等待　如果你的程序让他们等待　他们也会厌恶你

不过知道语句执行或函数调用的频繁程度　有时能帮助你洞察软件内部的行为　例如如果你建立了100个某种类型的对象　会发现你调用该类的构造函数有上千次　这个信息无疑是有价值的　而且语句和函数的调用次数能间接地帮助你理解不能直接测量的软件行为　例如如果你不能直接测量动态内存的使用　知道内存分配函数和内存释函数的调用频率也是有帮助的　　也就是　operators new, new[], delete, and delete[]—参见条款8

当然即使最好的profiler也是受其处理的数据所影响　如果用缺乏代表性的数据　profile你的程序　你就不能抱怨profiler会导致你优化程序的那80　的部分　从而不会对程序通常的性能有什么影响　记住profiler仅能够告诉你在某一次运行　或某几次运行　时一个程序运行情况　所以如果你用不具有代表性的输入数据profile一个程序　那你所进行的profile也没有代表型　相反这样做很可能导致你去优化不常用的软件行为　而在软件的常用领域　则对软件整体的效率起相反作用　即效率下降

防止这种不正确的结果　最好的方法是用尽可能多的数据profile你的软件　此外你必须确保每组数据在客户　或至少是最重要的客户　如何使用软件的方面能有代表性　通常获取有代表性的数据是很容易的　因为许多客户都愿意让你用他们的数据进行profile　毕竟你是为了他们需求而优化软件

条款17　考虑使用lazy evaluation　懒惰计算法

从效率的观点来看　最佳的计算就是根本不计算　那好　不过如果你根本就不用进行计算的话　为什么还在程序开始处加入代码进行计算呢?并且如果你不需要进行计算　那么如何必须执行这些代码呢
关键是要懒惰
还记得么　当你还是一个孩子时　你的父母叫你整理房间　你如果象我一样　就会说“好的“　然后继续做你自己的事情　你不会去整理自己的房间　在你心里整理房间被排在了最后的位置 实际上直到你听见父母下到门厅来查看你的房间是否已被整理时　你才会猛跑进自己的房间里并用最快的速度开始整理　如果你走运　你父母可能不会来检查你的房间　那样的话你就能根本不用整理房间了
同样的延迟策略也适用于具有五年工龄的C++程序员的工作上 在计算机科学中我们尊称这样的延迟为lazy evaluation　懒惰计算法　　当你使用了lazy evaluation后　采用此种方法的类将推迟计算工作直到系统需要这些计算的结果　如果不需要结果　将不用进行计算　软件的客户和你的父母一样　不会那么聪明
也许你想知道我说的这些到底是什么意思 也许举一个例子可以帮助你理解 lazy evaluation广泛适用于各种应用领域　所以我将分四个部分讲述

引用计数

```
class String { ... };                    //  一个string  类  (the standard
                                                 // string type may be implemented
                                                 // as described below, but it
                                                 // doesn't have to be)

String s1 = "Hello";

String s2 = s1;                                  /  调用string拷贝构造函数
```

通常string拷贝构造函数让s2被s1初始化后　s1和s2都有自己的”Hello”拷贝　这种拷贝构造函数会引起较大的开销　因为要制作s1值的拷贝　并把值赋给s2　这通常需要用new操作符分配堆内存(参见条款8)　需要调用strcpy函数拷贝s1内的数据到s2　这是一个eager evaluation　热情计算　　只因为到string拷贝构造函数　就要制作s1值的拷贝并把它赋给s2　然而这时的s2并不需要这个值的拷贝　因为s2没有被使用
懒惰能就是少工作　不应该赋给s2一个s1的拷贝　而是让s2与s1共享一个值　我们只须做一些记录以便知道谁在共享什么　就能够省掉调用new和拷贝字符的开销　事实上s1和s2共享一个数据结构　这对于client来说是透明的　对于下面的例子来说　这没有什么差别　因为它们只是读数据

```
cout << s1;                                    // 读s1的值

cout << s1 + s2;                               // 读s1和s2的值
```

仅仅当这个或那个string的值被修改时 共享同一个值的方法才会造成差异 仅仅修改一个string的值 而不是两个都被修改 这一点是极为重要的 例如这条语句

```
s2.convertToUpperCase();
```

这是至关紧要的 仅仅修改s2的值 而不是连s1的值一块修改

为了这样执行语句 string的convertToUpperCase函数应该制作s2值的一个拷贝 在修改前把这个私有的值赋给s2 在convertToUpperCase内部 我们不能再懒惰了 必须为s2 共享的 值制作拷贝以让s2自己使用 另一方面 如果不修改s2 我们就不用制作它自己值的拷贝 继续保持共享值直到程序退出 如果我们很幸运 s2不会被修改 这种情况下我们永远也不会为赋给它独立的值耗费精力

这种共享值方法的实现细节 包括所有的代码 在条款29中被提供 但是其蕴含的原则就是lazy evaluation 除非你却是需要 不去为任何东西制作拷贝 我们应该是懒惰的 只要可能就共享使用其它值 在一些应用领域 你经常可以这么做

区别对待读取和写入

继续讨论上面的reference-counting string对象 来看看使用lazy evaluation的第二种方法 考虑这样的代码

```
String s = "Homer's Iliad";                    // 假设是一个
                                               // reference-counted string
...

cout << s[3];                                  // 调用 operator[] 读取s[3]
s[3] = 'x';                                    // 调用 operator[] 写入 s[3]
```

首先调用operator[]用来读取string的部分值 但是第二次调用该函数是为了完成写操作 我们应能够区别对待读调用和写调用 因为读取reference-counted string是很容易的 而写入这个string则需要在写入前对该string值制作一个新拷贝

我们陷入了困难之中 为了能够这样做 需要在operator[]里采取不同的措施 根据是为了完成读取操作而调用该函数还是为了完成写入操作而调用该函数 我们如果判断调用operator[]的context是读取操作还是写入操作呢 残酷的事实是我们不可能判断出来 通过使用lazy evaluation和条款30中讲述的proxy class 我们可以推迟做出是读操作还是写操作的决定 直到我们能判断出正确的答案

Lazy Fetching 懒惰提取

第三个lazy evaluation的例子 假设你的程序使用了一些包含许多字段的大型对象

这些对象的生存期超越了程序运行期　所以它们必须被存储在数据库里　每一个对都有一个唯一的对象标识符　用来从数据库中重新获得对象

```
class LargeObject {                                    //  大型持久对象
public:
   LargeObject(ObjectID id);                           //  从磁盘中恢复对象

   const string& field1() const;                       // field 1的值
   int field2() const;                                 // field 2的值
   double field3() const;                              // ...
   const string& field4() const;
   const string& field5() const;
   ...

};
```

现在考虑一下从磁盘中恢复LargeObject的开销

```
void restoreAndProcessObject(ObjectID id)
{
   LargeObject object(id);                             //  恢复对象

   ...

}
```

因为LargeObject对象实例很大　为这样的对象获取所有的数据　数据库的操作的开销将非常大　特别是如果从远程数据库中获取数据和通过网络发送数据时　而在这种情况下　不需要读去所有数据　例如　考虑这样一个程序

```
void restoreAndProcessObject(ObjectID id)
{
   LargeObject object(id);

   if (object.field2() == 0) {
      cout << "Object " << id << ": null field2.\n";
   }
}
```

这里仅仅需要filed2的值　所以为获取其它字段而付出的努力都是浪费

当LargeObject对象被建立时　不从磁盘上读取所有的数据　这样懒惰法解决了这个问题　不过这时建立的仅是一个对象“壳“　当需要某个数据时　这个数据才被从数据库中取回　这种” demand-paged”对象初始化的实现方法是:

```
class LargeObject {
```

```
public:
    LargeObject(ObjectID id);

    const string& field1() const;
    int field2() const;
    double field3() const;
    const string& field4() const;
    ...

private:
    ObjectID oid;

    mutable string *field1Value;                    //参见下面有关
    mutable int *field2Value;                        // "mutable"的讨论
    mutable double *field3Value;
    mutable string *field4Value;
    ...

};

LargeObject::LargeObject(ObjectID id)
: oid(id), field1Value(0), field2Value(0), field3Value(0), ...
{}

const string& LargeObject::field1() const
{
    if (field1Value == 0) {
        从数据库中为filed 1读取数据    使
        field1Value  指向这个值;
    }

    return *field1Value;
}
```

对象中每个字段都用一个指向数据的指针来表示　LargeObject构造函数把每个指针初始化为空　这些空指针表示字段还没有从数据库中读取数值　每个LargeObject成员函数在访问字段指针所指向的数据之前必须字段指针检查的状态　如果指针为空　在对数据进行操作之前必须从数据库中读取对应的数据

实现Lazy Fetching时　你面临着一个问题　在任何成员函数里都有可能需要初始化

空指针使其指向真实的数据　包括在const成员函数里　例如field1　然而当你试图在const成员函数里修改数据时　编译器会出现问题　最好的方法是声明字段指针为mutable　这表示在任何函数里它们都能被修改　甚至在const成员函数里　参见Effective C++条款21　　这就是为什么在LargeObject里把字段声明为mutable
关键字mutalbe是一个比较新的C++ 特性　所以你用的编译器可能不支持它　如果是这样　你需要找到另一种方法让编译器允许你在const成员函数里修改数据成员　一种方法叫做"fake this"　伪造this指针　,你建立一个指向non-const指针　指向的对象与this指针一样　当你想修改数据成员时　你通过"fake this"访问它

```
class LargeObject {
public:
   const string& field1() const;                    // 没有变化
   ...

private:
   string *field1Value;                             // 不声明为 mutable
   ...                                                    // 因为老的编译器不
};                                                        // 支持它

const string& LargeObject::field1() const
{
   // 声明指针, fakeThis, 其与this指向同样的对象
   // 但是已经去掉了对象的常量属性
   LargeObject * const fakeThis =
      const_cast<LargeObject* const>(this);

   if (field1Value == 0) {
      fakeThis->field1Value =                       // 这赋值是正确的,
         the appropriate data                       // 因为fakeThis指向的
         from the database;                         //对象不是const
   }

   return *field1Value;
}
```

这个函数使用了const_cast(参见条款2)　去除了*this的const属性　如果你的编译器不支持cosnt_cast　你可以使用老式C风格的cast

```
// 使用老式的cast　来模仿mutable
const string& LargeObject::field1() const
{
```

```
    LargeObject * const fakeThis = (LargeObject* const)this;

    ...                                                    // as above

}
```

再来看LargeObject里的指针　必须把这些指针都初始化为空　然后每次使用它们时都必须进行测试　这是令人厌烦的而且容易导致错误发生　幸运的是使用smart(灵巧)指针可以自动地完成这种苦差使　具体内容可以参见条款28　如果在LargeObject里使用smart指针　你也将发现不再需要用mutalbe声明指针　这只是暂时的　因为当你实现smart指针类时你最终会碰到mutalbe

Lazy Expression Evaluation(懒惰表达式计算)

有关lazy evaluation的最后一个例子来自于数字程序　考虑这样的代码

```
template<class T>
class Matrix { ... };                                  // for homogeneous matrices

Matrix<int> m1(1000, 1000);                            //  一个  1000       1000  的矩阵
Matrix<int> m2(1000, 1000);                            //  同上

...

Matrix<int> m3 = m1 + m2;                                  // m1    m2
```

通常operator的实现使用eagar evaluation　在这种情况下　它会计算和返回m1与m2的和　这个计算量相当大　1000000次加法运算　　当然系统也会分配内存来存储这些值

lazy evaluation方法说这样做工作太多　所以还是不要去做　而是应该建立一个数据结构来表示m3的值是m1与m2的和　在用一个enum表示它们间是加法操作　很明显　建立这个数据结构比m1与m2相加要快许多　也能够节省大量的内存

考虑程序后面这部分内容　在使用m3之前　代码执行如下

```
Matrix<int> m4(1000, 1000);

...                                                    //  赋给m4一些值

m3 = m4 * m1;
```

现在我们可以忘掉m3是m1与m2的和　因此节省了计算的开销　　在这里我们应该记住m3是m4与m1运算的结果　不必说　我们不用进行乘法运算　因为我们是懒惰的　还记得么

这个例子看上去有些做作　因为一个好的程序员不会这样写程序　计算两个矩阵的和而不去用它们　但是它实际上又不象看上去的那么做作　虽然好程序员不会进行不需要的计算　但是在维护中程序员修改了程序的路径　使得以前有用的计算变得没有了作用　这种情况是常见的　通过定义使用前才进行计算的对象可以减少这种情况发生的可能性　参见Effective C++条款32　　不过这个问题偶尔仍然会出现

但是如果这就是使用lazy evaluation唯一的时机　那就太不值得了　一个更常见的应用领域是当我们仅仅需要计算结果的一部分时　例如假设我们初始化m3的值为m1和m2的和　然后象这样使用m3

```
cout << m3[4];                                          // 打印m3的第四行
```

很明显　我们不能再懒惰了　应该计算m3的第四行值　但是我们也不能雄心过大　我们没有理由计算m3第四行以外的结果　m3其余的部分仍旧保持未计算的状态直到确实需要它们的值　很走运　我们一直不需要

我们怎么可能这么走运呢　矩阵计算领域的经验显示这种可能性很大　实际上lazy evaluation就存在于APL语言中　APL是在1960年代发展起来语言　能够进行基于矩阵的交互式的运算 那时侯运行它的计算机的运算能力还没有现在微波炉里的芯片高　APL表面上能够进行进行矩阵的加　乘　甚至能够快速地与大矩阵相除 它的技巧就是lazy evaluation 这个技巧通常是有效的 因为一般APL的用户加乘或除以矩阵不是因为他们需要整个矩阵的值　而是仅仅需要其一小部分的值 APL使用lazy evaluation 来拖延它们的计算直到确切地知道需要矩阵哪一部分的结果　然后仅仅计算这一部分　实际上　这能允许用户在一台根本不能完成eager evaluation的计算机上交互式地完成大量的计算　现在计算机速度很快　但是数据集也更大　用户也更缺乏耐心　所以很多现在的矩阵库程序仍旧使用lazy evaluation

公正地讲　懒惰有时也会失败　如果这样使用m3

```
cout << m3;                                             // 打印m3所有的值
```

一切都完了　我们必须计算m3的全部数值　同样如果修改m3所依赖的任一个矩阵　我们也必须立即计算

```
m3 = m1 + m2;                                           // 记住m3是m1与m2的和
                                                        //

m1 = m4;                                                // 现在m3是m2与m1的旧值之和
                                                        //
```

这里我们我们必须采取措施确保赋值给m1以后不会改变m3　在Matrix<int>赋值操作符里　我们能够在改变m1之前捕获m3的值　或者我们可以给m　　1的旧值制作一个拷贝让m3依赖于这个拷贝计算　我们必须采取措施确保m1被赋值以后m3的值保持不变　其它可能会修改矩阵的函数都必须用同样的方式处理

因为需要存储两个值之间的依赖关系　维护存储值　依赖关系或上述两者　重载操作符例如赋值符　拷贝操作和加法操作　所以lazy evaluation在数字领域应用得很多　另一方面运行程序时它经常节省大量的时间和空间

总结

以上这四个例子展示了lazy evaluation在各个领域都是有用的　能避免不需要的对象拷贝　通过使用operator[]区分出读操作　避免不需要的数据库读取操作　避免不需要的数字操作　但是它并不总是有用　就好象如果你的父母总是来检查你的房间　那么拖延整理房间将不会减少你的工作量　实际上　如果你的计算都是重要的　lazy evaluation可能会减慢速度并增加内存的使用　因为除了进行所有的计算以外　你还必须维护数据结构让lazy evaluation尽可能地在第一时间运行　在某些情况下要求软件进行原来可以避免的计算　这时lazy evaluation才是有用的

lazy evaluation对于C++来说没有什么特殊的东西　这个技术能被运用于各种语言里　几种语言例如著名的APL　dialects of Lisp　事实上所有的数据流语言　都把这种思想做为语言的一个基本部分　然而主流程序设计语言采用的是eager evaluation　C++是主流语言　不过C++特别适合用户实现lazy evaluation　因为它对封装的支持使得能在类里加入lazy evaluation　而根本不用让类的client知道

再看一下上述例子中的代码片段　你就能知道采用eager还是lazy evaluation　在类的interface并没有半点差别　这就是说我们可以直接用eager evaluation方法来实现一个类　但是如果你用通过profiler调查　参见条款16　显示出类实现有一个性能瓶颈　就可以用使用lazy evaluation的类实现来替代它　参见Effective C++条款34

对于client来说所改变的仅是性能的提高　重新编译和链接后　　这是client喜欢的软件升级方式　它使你完全可以为懒惰而骄傲

条款18　分期摊还期望的计算

在条款17中　我极力称赞懒惰的优点　尽可能地拖延时间　并且我解释说懒惰如何提高程序的运行效率　在这个条款里我将采用一种不同的态度　这里将不存在懒惰　我鼓励你让程序做的事情比被要求的还要多　通过这种方式来提高软件的性能　这个条款的核心就是over-eager evaluation　过度热情计算法　　在要求你做某些事情以前就完成它们　例如下面这个模板类　用来表示放有大量数字型数据的一个集合

```
template<class NumericalType>
class DataCollection {
public:
   NumericalType min() const;
   NumericalType max() const;
   NumericalType avg() const;
   ...
};
```

假设min,max和avg函数分别返回现在这个集合的最小值　最大值和平均值　有三种方法实现这三种函数　使用eager evaluation(热情计算法)　当min,max和avg函数被调用时　我们检测集合内所有的数值　然后返回一个合适的值　使用lazy evaluation　懒惰计算法　,只有确实需要函数的返回值时我们才要求函数返回能用来确定准确数值的数据结构　使用 over-eager evaluation　过度热情计算法　　我们随时跟踪目前集合的最小值　最大值和平均值　这样当min,max或avg被调用时　我们可以不用计算就立刻返回正确的数值　如果频繁调用min,max和avg　我们把跟踪集合最小值　最大值和平均值的开销分摊到所有这些函数的调用上　每次函数调用所分摊的开销比eager evaluation或lazy evaluation要小

隐藏在over-eager evaluation后面的思想是如果你认为一个计算需要频繁进行　你就可以设计一个数据结构高效地处理这些计算需求　这样可以降低每次计算需求的开销

采用over-eager最简单的方法就是caching(缓存)那些已经被计算出来而以后还有可能需要的值　例如你编写了一个程序　用来提供有关雇员的信息　这些信息中的经常被需要的部分是雇员的办公隔间号码　而假设雇员信息存储在数据库里　但是对于大多数应用程序来说　雇员隔间号都是不相关的　所以数据库不对查抄它们进行优化　为了避免你的程序给数据库造成沉重的负担　可以编写一个函数findCubicleNumber　用来cache查找的数据　以后需要已经被获取的隔间号时　可以在cache里找到　而不用向数据库查询

以下是实现findCubicleNumber的一种方法　它使用了标准模板库　STL　里的map对象　有关STL参见条款35

```
int findCubicleNumber(const string& employeeName)
```

```
{
   // 定义静态map  存储 (employee name, cubicle number)
   // pairs. 这个 map 是local cache
   typedef map<string, int> CubicleMap;
   static CubicleMap cubes;

   // try to find an entry for employeeName in the cache;
   // the STL iterator "it" will then point to the found
   // entry, if there is one (see Item 35 for details)
   CubicleMap::iterator it = cubes.find(employeeName);

   // "it""s value will be cubes.end() if no entry was
   // found (this is standard STL behavior). If this is
   // the case, consult the database for the cubicle
   // number, then add it to the cache
   if (it == cubes.end()) {
      int cubicle =
         the result of looking up employeeName's cubicle
         number in the database;

      cubes[employeeName] = cubicle;                    // add the pair
                                                        // (employeeName, cubicle)
                                                        // to the cache
      return cubicle;
   }
   else {
      // "it" points to the correct cache entry, which is a
      // (employee name, cubicle number) pair. We want only
      // the second component of this pair, and the member
      // "second" will give it to us
      return (*it).second;
   }
}
```

不要陷入STL代码的实现细节里　你读完条款35以后　你会比较清楚　　应该把注意力放在这个函数蕴含的方法上　这个方法是使用local cache　用开销相对不大的内存中查询来替代开销较大的数据库查询　假如隔间号被不止一次地频繁需要　在findCubicleNumber内使用cache会减少返回隔间号的平均开销

　上述代码里有一个细节需要解释一下　最后一个语句返回的是(*it).second　而

不是常用的it->second 为什么 答案是这是为了遵守STL的规则 简单地说 iterator是一个对象 不是指针 所以不能保证"->"被正确应用到它上面 不过STL要求"."和"*"在iterator上是合法的 所以(*it).second在语法上虽然比较繁琐 但是保证能运行

catching是一种分摊期望的计算开销的方法 Prefetching(预提取)是另一种方法 你可以把prefech想象成购买大批商品而获得的折扣 例如磁盘控制器从磁盘读取数据时 它们会读取一整块或整个扇区的数据 即使程序仅需要一小块数据 这是因为一次读取一大块数据比在不同时间读取两个或三个小块数据要快 而且经验显示如果需要一个地方的数据 则很可能也需要它旁边的数据 这是位置相关现象 正因为这种现象 系统设计者才有理由为指令和数据使用磁盘cache和内存cache 还有使用指令prefetch

你说你不关心象磁盘控制器或CPU cache这样低级的东西 没有问题 prefetch在高端应用里也有优点 例如你为dynamic数组实现一个模板 dynamic就是开始时具有一定的尺寸 以后可以自动扩展的数组 所以所有非负的索引都是合法的

```
template<class T>                                    // dynamic数组
class DynArray { ... };                              // 模板

DynArray<double> a;                                  // 在这时, 只有 a[0]
                                                     // 是合法的数组元素


a[22] = 3.5;                                         // a 自动扩展
                                                     //: 现在索引|0  22
                                                     // 是合法的

a[32] = 0;                                           // 有自行扩展;
                                                     // 现在 a[0]-a[32]是合法的
```

一个DynArray对象如何在需要时自行扩展呢 一种直接的方法是分配所需的额外的内存 就象这样

```
template<class T>
T& DynArray<T>::operator[](int index)
{
   if (index < 0) {
      throw an exception;                            // 负数索引仍不
   }                                                 // 合法

   if (index >当前最大的索引值) {
      调用new分配足够的额外内存 以使得
```

```
    索引合法;
  }

  返回index位置上的数组元素;
}
```

每次需要增加数组长度时　这种方法都要调用new　但是调用new会触发operator new(参见条款8)　operator new (和operator delete)的调用通常开销很大　因为它们将导致底层操作系统的调用　系统调用的速度一般比进程内函数调用的速度慢　因此我们应该尽量少使用系统调用

使用Over-eager evaluation方法　其原因我们现在必须增加数组的尺寸以容纳索引i 那么根据位置相关性原则我们可能还会增加数组尺寸以在未来容纳比i 大的其它索引　为了避免为扩展而进行第二次　预料中的　内存分配　我们现在增加DynArray的尺寸比能使i 合法的尺寸要大 我们希望未来的扩展将被包含在我们提供的范围内　例如我们可以这样编写DynArray::operator[]

```
template<class T>
T& DynArray<T>::operator[](int index)
{
  if (index < 0) throw an exception;

  if (index >  当前最大的索引值) {
    int diff = index –  当前最大的索引值;

    调用new分配足够的额外内存　使得
    index+diff合法;
  }

  返回index位置上的数组元素;
}
```

这个函数每次分配的内存是数组扩展所需内存的两倍 如果我们再来看一下前面遇到的那种情况　就会注意到DynArray只分配了一次额外内存　即使它的逻辑尺寸被扩展了两次

```
DynArray<double> a;                          //  仅仅a[0]是合法的

a[22] = 3.5;                                 //  调用new扩展
                                                  // a的存储空间到索引44
                                                  // a的逻辑尺寸
                                                  //  变为23
```

```
a[32] = 0;                                          // a的逻辑尺寸
                                                    // 被改变　允许使用a[32],
                                                    // 但是没有调用new
```

如果再次需要扩展a　只要提供的新索引不大于44　扩展的开销就不大

贯穿本条款的是一个常见的主题　更快的速度经常会消耗更多的内存　跟踪运行时的最小值　最大值和平均值　这需要额外的空间　但是能节省时间　Cache运算结果需要更多的内存　但是一旦需要被cache的结果时就能减少需要重新生成的时间　Prefetch需要空间放置被prefetch的东西　但是它减少了访问它们所需的时间　自从有了计算机就有这样的描述　你能以空间换时间　　然而不总是这样　使用大型对象意味着不适合虚拟内存或cache 页　在一些罕见的情况下　建立大对象会降低软件的性能　因为分页操作的增加　详见操作系统中内存管理　译者注　　cache命中率降低　或者两者都同时发生　如何发现你正遭遇这样的问题呢　你必须profile, profile, profile(参见条款16)

在本条款中我提出的建议　即通过over-eager方法分摊预期计算的开销　例如caching和prefething　这并不与我在条款17中提出的有关lazy evaluation的建议相矛盾　当你必须支持某些操作而不总需要其结果时　可以使用lazy evaluation用以提高程序运行效率 当你必须支持某些操作而其结果几乎总是被需要或被不止一次地需要时　可以使用over-eager用以提高程序运行效率　它们对性能的巨大提高证明在这方面花些精力是值得的

条款19　理解临时对象的来源

当程序员之间进行交谈时　他们经常把仅仅需要一小段时间的变量称为临时变量　例如在下面这段swap(交换)例程里

```
template<class T>
void swap(T& object1, T& object2)
{
    T temp = object1;
    object1 = object2;
    object2 = temp;
}
```

通常把temp叫做临时变量　不过就C++而言　temp跟本不是临时变量　它只是一个函数的局部对象

在C++中真正的临时对象是看不见的　它们不出现在你的源代码中　建立一个没有命名的非堆　non-heap　对象会产生临时对象　这种未命名的对象通常在两种条件下产生　为了使函数成功调用而进行隐式类型转换和函数返回对象时　理解如何和为什么建立这些临时对象是很重要的 因为构造和释放它们的开销对于程序的性能来说有着不可忽视的影响

首先考虑为使函数成功调用而建立临时对象这种情况 当传送给函数的对象类型与参数类型不匹配时会产生这种情况　例如一个函数　它用来计算一个字符在字符串中出现的次数

```
// 返回ch在str中出现的次数
size_t countChar(const string& str, char ch);

char buffer[MAX_STRING_LEN];
char c;

// 读入到一个字符和字符串中　用setw
// 避免缓存溢出　当读取一个字符串时
cin >> c >> setw(MAX_STRING_LEN) >> buffer;

cout << "There are " << countChar(buffer, c)
        << " occurrences of the character " << c
        << " in " << buffer << endl;
```

看一下countChar的调用　第一个被传送的参数是字符数组　但是对应函数的正被绑定的参数的类型是const string&　仅当消除类型不匹配后　才能成功进行这个调用　你的编译器很乐意替你消除它　方法是建立一个string类型的临时对象　通过以buffer做为参数调用string的构造函数来初始化这个临时对象　countChar的参数str

被绑定在这个临时的string对象上　当countChar返回时　临时对象自动释放　这样的类型转换很方便　尽管很危险　参见条款5　　但是从效率的观点来看　临时string对象的构造和释放是不必要的开销　通常有两个方法可以消除它　一种是重新设计你的代码　不让发生这种类型转换　这种方法在条款5中被研究和分析　另一种方法是通过修改软件而不再需要类型转换　条款21讲述了如何去做　仅当通过传值　by value　方式传递对象或传递常量引用　reference-to-const　参数时　才会发生这些类型转换　当传递一个非常量引用　reference-to-non-const　参数对象　就不会发生　考虑一下这个函数

```
void uppercasify(string& str);                    // 把str中所有的字符
                                                  // 改变成大写
```

在字符计数的例子里　能够成功传递char数组到countChar中　但是在这里试图用char数组调用upeercasify函数　则不会成功

```
char subtleBookPlug[] = "Effective C++";

uppercasify(subtleBookPlug);                      // 错误!
```

没有为使调用成功而建立临时对象　为什么呢

假设建立一个临时对象　那么临时对象将被传递到upeercasify中　其会修改这个临时对象　把它的字符改成大写　但是对subtleBookPlug函数调用的真正参数没有任何影响　仅仅改变了临时从subtleBookPlug生成的string对象　无疑这不是程序员所希望的　程序员传递subtleBookPlug参数到uppercasify函数中　期望修改subtleBookPlug的值　当程序员期望修改非临时对象时　对非常量引用　references-to-non-const　进行的隐式类型转换却修改临时对象　这就是为什么C++语言禁止为非常量引用　reference-to-non-const　产生临时对象　这样非常量引用　reference-to-non-const　参数就不会遇到这种问题

建立临时对象的第二种环境是函数返回对象时　例如operator+必须返回一个对象以表示它的两个操作数的和　参见Effective C++ 条款23　　例如给定一个类型Number　这种类型的operator+被这样声明

```
const Number operator+(const Number& lhs,
                       const Number& rhs);
```

这个函数的返回值是临时的　因为它没有被命名　它只是函数的返回值　你必须为每次调用operator+构造和释放这个对象而付出代价　有关为什么返回值是const的详细解释　参见Effective C++条款21

通常你不想付出这样的开销　对于这种函数　你可以切换到operator=　而避免开销　条款22告诉我们进行这种转换的方法　不过对于大多数返回对象的函数来说　无法切换到不同的函数　从而没有办法避免构造和释放返回值　至少在概念上没有办法避免它　然而概念和现实之间又一个黑暗地带　叫做优化　有时你能以某种方法编写返回对象的函数　以允许你的编译器优化临时对象　这些优化中　最常见和最有效的是返回值优化　这是条款20的内容

综上所述　临时对象是有开销的　所以你应该尽可能地去除它们　然而更重要的是训练自己寻找可能建立临时对象的地方　在任何时候只要见到常量引用　reference-to-const　参数　就存在建立临时对象而绑定在参数上的可能性　在任何时候只要见到函数返回对象　就会有一个临时对象被建立　以后被释放　学会寻找这些对象构造　你就能显著地增强透过编译器表面动作而看到其背后开销的能力

条款20　协助完成返回值优化

一个返回对象的函数很难有较高的效率 因为传值返回会导致调用对象内的构造和析构函数(参见条款19)　这种调用是不能避免的　问题很简单　一个函数要么为了保证正确的行为而返回对象要么就不这么做　如果它返回了对象　就没有办法摆脱被返回的对象　就说到这

考虑rational(有理数)类的成员函数operator*

```
class Rational {
public:
    Rational(int numerator = 0, int denominator = 1);
    ...
    int numerator() const;
    int denominator() const;
};

// 有关为什么返回值是const的解释　参见条款6,
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

甚至不用看operator*的代码　我们就知道它肯定要返回一个对象　因为它返回的是两个任意数字的计算结果　这些结果是任意的数字　operator*如何能避免建立新对象来容纳它们的计算结果呢　这是不可能的 所以它必须得建立新对象并返回它　不过C++程序员仍然花费大量的精力寻找传说中的方法　能够去除传值返回的对象　参见Effective C++ 条款23和条款31

有时人们会返回指针　从而导致这种滑稽的句法

```
// 一种不合理的避免返回对象的方法
const Rational * operator*(const Rational& lhs,
                           const Rational& rhs);

Rational a = 10;
Rational b(1, 2);

Rational c = *(a * b);                    //你觉得这样很“正常”么
```

它也引发出一个问题 调用者应该删除函数返回对象的指针么　答案通常是肯定的　并且通常会导致资源泄漏

其它一些开发人员会返回引用　这种方法能产生可接受的句法

```
//一种危险的(和不正确的)方法　用来避免返回对象
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs);
```

```
Rational a = 10;
Rational b(1, 2);

Rational c = a * b;                                    // 看上去很合理
```

但是函数不能被正确地实现　一种尝试的方法是这样的:

```
// 另一种危险的方法 (和不正确的)方法　用来
// 避免返回对象
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs)
{
   Rational result(lhs.numerator() * rhs.numerator(),
                   lhs.denominator() * rhs.denominator());
   return result;
}
```

这个函数返回的引用　其指向的对象已经存在了　它返回的是一个指向局部对象result的引用　当operator*　退出时result被自动释放　返回指向已被释放的对象的引用　这样的引用绝对不能使用

相信我　一些函数　operator*也在其中　必须要返回对象　这就是它们的运行方法　不要与其对抗　你不会赢的

你消除传值返回的对象的努力不会获得胜利　这是一场错误的战争　从效率的观点来看　你不应该关心函数返回的对象　你仅仅应该关心对象的开销　你所应该关心的是把你的努力引导到寻找减少返回对象的开销上来　而不是去消除对象本身　我们现在认识到这种寻求是无用的　　如果没有与这些对象相关的开销　谁还会关心有多少对象被建立呢

以某种方法返回对象　能让编译器消除临时对象的开销　这样编写函数通常是很普遍的　这种技巧是返回constructor argument而不是直接返回对象　你可以这样做

```
// 一种高效和正确的方法　用来实现
// 返回对象的函数
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
{
   return Rational(lhs.numerator() * rhs.numerator(),
                   lhs.denominator() * rhs.denominator());
}
```

仔细观察被返回的表达式　它看上去好象正在调用Rational的构造函数　实际上确是这样　你通过这个表达式建立一个临时的Rational对象

```
Rational(lhs.numerator() * rhs.numerator(),
```

```
                lhs.denominator() * rhs.denominator());
```

并且这是一个临时对象　函数把它拷贝给函数的返回值

返回constructor argument而不出现局部对象　这种方法还会给你带来很多开销　因为你仍旧必须为在函数内临时对象的构造和释放而付出代价　你仍旧必须为函数返回对象的构造和释放而付出代价　但是你已经获得了好处　C++规则允许编译器优化不出现的临时对象　temporary objects out of existence　　因此如果你在如下的环境里调用operator*

```
Rational a = 10;
Rational b(1, 2);

Rational c = a * b;                                    // 在这里调用operator*
```

编译器就会被允许消除在operator*内的临时变量和operator*返回的临时变量　它们能在为目标c分配的内存里构造return表达式定义的对象　如果你的编译器这样去做　调用operator*的临时对象的开销就是零　没有建立临时对象　你的代价就是调用一个构造函数－－建立c时调用的构造函数　而且你不能比这做得更好了　因为c是命名对象　命名对象不能被消除　参见条款22　　不过你还可以通过把函数声明为inline来消除operator*的调用开销　不过首先参见Effective C++　条款33

```
// the most efficient way to write a function returning
// an object
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
   return Rational(lhs.numerator() * rhs.numerator(),
                   lhs.denominator() * rhs.denominator());
}
```

“好　不错”　你嘀咕地说　“优化　谁关心编译器能做什么　我想知道它们确实做了什么　Does any of this nonsense work with real compilers?” It does　这种特殊的优化－－通过使用函数的return location　或者用一个在函数调用位置的对象来替代　　来消除局部临时对象－－是众所周知的和被普遍实现的　它甚至还有一个名字　返回值优化　return value optimization　　实际上这种优化有自己的名字本身就可以解释为什么它被广泛地使用　寻找C++编译器的程序员会问销售商编译器是否有返回值优化功能　如果一个销售商说有而另一个问“那是什么东西　”　第一个销售商就会有明显的竞争优势　啊　资本主义　有时你实在应该去爱它　　谨代表作者观点　译者坚决拥护四项基本原则　　译者注　:-　)

附录

文中最后一段黑体部分如何翻译　我有些拿不准　请高手告知　为了容易理解　我在此附上此文最后一段的英文原文

"Yeah, yeah," you mutter, "optimization, schmoptimization. Who cares what compilers can do? I want to know what they do do. Does any of this nonsense work with real compilers?" It does. This particular optimization — eliminating a local temporary by using a function's return location (and possibly replacing that with an object at the function's call site) — is both well-known and commonly implemented. It even has a name: the return value optimization. In fact, the existence of a name for this optimization may explain why it's so widely available. Programmers looking for a C++ compiler can ask vendors whether the return value optimization is implemented. If one vendor says yes and another says "The what?," the first vendor has a notable competitive advantage. Ah, capitalism. Sometimes you just gotta love it.

to zhc,

“Does any of this nonsense work with real compilers?” It does.是两句话

第一句　“Does any of this nonsense work with real compilers?”

this nonsense指的是前面提到的optimization和schmoptimization　也许还泛指这类技术　以表示问话人的一种浮躁的语气　从这里也可以看出作者Scott Meyers一贯诙谐幽默的笔风

work with　动词词组,这里表示“发挥作用”之类的意思

这个句子的陈述句应该是 one(or each, or every one) of this nonsense works with real compilers.

第二句　It does. 作者又自己回答了这个问题　表示这种技术已经在编译器上实现了

然后文章就接着说明了编译器实现这个技术的具体情况

另外　其实我自己也不懂什么是schmoptimization　所以干脆连前面那个optimization也不翻译　让两个词形成并列的效果　因为他们本来也就是并列的关系,如果一

个翻译了　另一个不翻译　读者就会觉得两个词不太对称　原句所要表达的并列关系就会被削弱　　待到以后搞清楚了schmoptimization的意思　就可以两个词一起翻译过来　仍然保持并列关系　我想这大凡是新手翻译文章的基本技巧——至少我就经常这么干

zhc (2001-10-23 16:34:06)

这个礼拜　到昌平培训　没有时间翻译　礼拜五回来后　会继续翻译
TO kingofark
我觉得你翻译得不错　我根据上下文推断出来的意思与你的翻译差不多　但是我没有见过Does any of this nonsense work with real compilers?” It does　这种句式　你能帮我分一下这个句子的结构么　在这里非常感谢你的帮助

kingofark (2001-10-22 10:37:13)

to zhc,

在看过你翻译的这个条款后　我翻译了你留下的那段原文　希望对你有所帮助　当然我的理解也许不正

确　也请你指正　谢谢

[原文]

"Yeah, yeah," you mutter, "optimization, schmoptimization. Who cares what compilers can

do? I want to know what they do do. Does any of this nonsense work with real compilers?"

It does. This particular optimization — eliminating a local temporary by using a

function's return location (and possibly replacing that with an object at the function's

call site) — is both well-known and commonly implemented. It even has a name: the return

value optimization. In fact, the existence of a name for this optimization may explain why

it's so widely available. Programmers looking for a C++ compiler can ask vendors whether the return value optimization is implemented. If one vendor says yes and another says "The what?," the first vendor has a notable competitive advantage. Ah, capitalism. Sometimes you just gotta love it.

[kingofark的翻译]

“好好好 ”你嘀咕着 “optimization 还有那个什么schmoptimization 但谁会关心编译器应该可

以做什么 我只想知道编译器现在能够做什么 那些该死的什么优化到底能不能在实际的编译器中工作

”回答是确实可以 这种很特别的优化方法——即通过利用函数在return处的位置来消除一些局部临

时对象 并可能就是利用函数调用处的一个对象的空间来顶替局部临时对象的空间 的方法——其实已经

是众所周知并被广泛实现的了 这个方法还有一个名称 返回值优化 return value optimization

实际上 “它有一个名称”这个事实或许就是它被广泛实现的原因 试想程序员在选择C++编译器的

时候 向销售商询问其编译器是否实现了返回值优化 return value optimization 的功能 如果一

个销售商说是的我们的产品实现了这个功能 而另一个销售商却无知的反问道“什么功能 ” 那么第

一个销售商显然就具有了明显的竞争优势 呃 资本主义 有时候你必须热爱它 译注 这里作者似

乎在以调侃的方式说明 资本主义营造的这种商业竞争状况 使得返回值优化

return value

optimization 这样一个技术很快的被编译器厂商用来作为对抗竞争对手的砝码——特别是一旦这个技

术有了个冠冕堂皇的名称的时候 这个名称马上就可以被列在宣传广告中 招揽客户

to zhc,

我觉得在“This particular optimization — eliminating a local temporary by using a

function's return location (and possibly replacing that with an object at the function's

call site) — is both well-known and commonly implemented.”这一句中

1)“eliminating a local temporary by using a function's return location”指的就是

return Rational(lhs.numerator() * rhs.numerator(),

lhs.denominator() * rhs.denominator());

即你的译文中“返回constructor argument而不出现局部对象 这种方法还会给你带来很多开销 因为

你仍旧必须为在函数内临时对象的构造和释放而付出代价 你仍旧必须为函数返回对象的构造和释放而

付出代价 但是你已经获得了好处 C++规则允许编译器优化不出现的临时对象 temporary objects

out of existence ”这个事实

2)“replacing that with an object at the function's call site”指的就是

你的译文中“编译器就会被允许消除在operator*内的临时变量和operator*返回的临时变量　它们能在

为目标c分配的内存里构造return表达式定义的对象　”这个事实

我的理解恐有错误　请你指正

条款21　通过重载避免隐式类型转换

以下是一段代码　如果没有什么不寻常的原因　实在看不出什么东西

```
class UPInt {                                          // unlimited precision
public:                                                  // integers 类
   UPInt();
   UPInt(int value);
   ...

};

//有关为什么返回值是const的解释　参见Effective C++ 条款21
const UPInt operator+(const UPInt& lhs, const UPInt& rhs);

UPInt upi1, upi2;

...

UPInt upi3 = upi1 + upi2;
```

这里还看不出什么令人惊讶的东西　upi1 和upi2都是UPInt对象　所以它们之间相加就会调用UPInts的operator函数

现在考虑下面这些语句

```
upi3 = upi1 + 10;

upi3 = 10 + upi2;
```

这些语句也能够成功运行 方法是通过建立临时对象把整形数10转换为UPInts 参见条款19

让编译器完成这种类型转换是确实是很方便 但是建立临时对象进行类型转换工作是有开销的　而我们不想承担这种开销　就象大多数人只想从政府那里受益而不想为此付出一样　大多数C++程序员希望进行没有临时对象开销的隐式类型转换　但是在计算领域里发生不了赤字现象　我们如何能这么做呢

让我们回退一步　认识到我们的目的不是真的要进行类型转换　而是用UPint和int做为参数调用operator　隐式类型转换只是用来达到目的的手段　但是我们不要混淆手段与目的　还有一种方法可以成功进行operator的混合类型调用　它将消除隐式类型转换的需要　如果我们想要把UPInt和int对象相加　通过声明如下几个函数达到这个目的　每一个函数有不同的参数类型集

```
const UPInt operator+(const UPInt& lhs,          // add UPInt
                      const UPInt& rhs);         // and UPInt
```

```
const UPInt operator+(const UPInt& lhs,          // add UPInt
                      int rhs);                  // and int

const UPInt operator+(int lhs,                   // add int and
                      const UPInt& rhs);         // UPInt

UPInt upi1, upi2;

...

UPInt upi3 = upi1 + upi2;                        // 正确,没有由upi1 或 upi2
                                                 // 生成的临时对象

upi3 = upi1 + 10;                                // 正确, 没有由upi1 or 10
                                                 // 生成的临时对象

upi3 = 10 + upi2;                                //正确, 没有由10 or upi2
                                                 //生成的临时对象
```

一旦你开始用函数重载来消除类型转换 你就有可能这样声明函数 把自己陷入危险之中

```
const UPInt operator+(int lhs, int rhs);         // 错误!
```

这个想法是合情合理的 对于UPInt和int类型 我们想要用所有可能的组合来重载operator函数 上面只给出了三种重载函数 唯一漏掉的是带有两个int参数的operator 所以我们想把它加上

有道理么 在C++中有一条规则是每一个重载的operator必须带有一个用户定义类型 user-defined type 的参数 int不是用户定义类型 所以我们不能重载operator成为仅带有此类型参数的函数 如果没有这条规则 程序员将能改变预定义的操作 这样做肯定把程序引入混乱的境地 比如企图重载上述的operator 将会改变int类型相加的含义

利用重载避免临时对象的方法不只是用在operator函数上 比如在大多数程序中你想允许在所有能使用string对象的地方 也一样可以使用char* 反之亦然 同样如果你正在使用numerical 数字 类 例如complex 参见条款35 ,你想让int和double这样的类型可以使用在numerical对象的任何地方 因此任何带有string char* complex参数的函数可以采用重载方式来消除类型转换

不过 必须谨记80 20规则 参见条款16 没有必要实现大量的重载函数 除非你有理由确信程序使用重载函数以后其整体效率会有显著的提高

条款22　考虑用运算符的赋值形式　op=　取代其单独形式　op

大多数程序员认为如果他们能这样写代码

x = x + y;　　x = x - y;

那他们也能这样写

x += y;　　x -= y;

如果x和y是用户定义的类型　user-defined type　　就不能确保这样　就C++来说　operator+　operator=和operator+=之间没有任何关系　因此如果你想让这三个operator同时存在并具有你所期望的关系　就必须自己实现它们　同理　operator -, *, /,　等等也一样

确保operator的赋值形式　assignment version　　例如operator+=　与一个operator的单独形式　stand-alone　(例如　operator+)之间存在正常的关系　一种好方法是后者　指operator+　译者注　根据前者　指operator+=　译者注　来实现　参见条款6　这很容易

```
class Rational {
public:
    ...
    Rational& operator+=(const Rational& rhs);
    Rational& operator-=(const Rational& rhs);
};

// operator+  根据operator+=实现;
//有关为什么返回值是const的解释
//参见Effective C++条款21  和  109页  的有关实现的警告
const Rational operator+(const Rational& lhs,
                         const Rational& rhs)
{
    return Rational(lhs) += rhs;
}

// operator-  根据  operator -=  来实现
const Rational operator-(const Rational& lhs,
                         const Rational& rhs)
{
    return Rational(lhs) -= rhs;
}
```

在这个例子里　从零开始实现operator+=和-=　而operator+　和operator-　则是通过调用前述的函数来提供自己的功能　使用这种设计方法　只用维护operator的赋值形

式就行了 而且如果假设operator赋值形式在类的public接口里 这就不用让operator的单独形式成为类的友元 参见Effective C++ 条款19
如果你不介意把所有的operator的单独形式放在全局域里 那就可以使用模板来替代单独形式的函数的编写

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
   return T(lhs) += rhs;                                  // 参见下面的讨论
}

template<class T>
const T operator-(const T& lhs, const T& rhs)
{
   return T(lhs) -= rhs;                                   // 参见下面的讨论
}

...
```

使用这些模板 只要为operator赋值形式定义某种类型 一旦需要 其对应的operator单独形式就会被自动生成
这样编写确实不错 但是到目前为止 我们还没有考虑效率问题 效率毕竟是本章的主题 在这里值得指出的是三个效率方面的问题 第一 总的来说operator的赋值形式比其单独形式效率更高 因为单独形式要返回一个新对象 从而在临时对象的构造和释放上有一些开销 参见条款19和条款20 还有Effective C++条款23 operator的赋值形式把结果写到左边的参数里 因此不需要生成临时对象来容纳operator的返回值
第二 提供operator的赋值形式的同时也要提供其标准形式 允许类的客户端在便利与效率上做出折衷选择 也就是说 客户端可以决定是这样编写

```
Rational a, b, c, d, result;
...
result = a + b + c + d;                    // 可能用了3个临时对象
                                                  // 每个operator+ 调用使用1个
```

还是这样编写

```
result = a;                                                  //不用临时对象
result += b;                                                 // 不用临时对象
result += c;                                                 //不用临时对象
result += d;                                                 //不用临时对象
```

前者比较容易编写 debug和维护 并且在80 的时间里它的性能是可以被接受

的　参见条款16　　后者具有更高的效率　估计这对于汇编语言程序员来说会更直观一些　通过提供两种方案　你可以让客户端开发人员用更容易阅读的单独形式的operator来开发和debug代码　同时保留用效率更高的operator赋值形式替代单独形式的权力　而且根据operator的赋值形式实现其单独形式　这样你能确保当客户端从一种形式切换到另一种形式时　操作的语义可以保持不变

最后一点　涉及到operator单独形式的实现　再看看operator+ 的实现

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{ return T(lhs) += rhs; }
```

表达式T(lhs)调用了T的拷贝构造函数　它建立一个临时对象　其值与lhs一样　这个临时对象用来与rhs一起调用operator+=　　操作的结果被从operator+.返回　这个代码好像不用写得这么隐密　这样写不是更好么

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
   T result(lhs);                                  // 拷贝lhs 到 result中
   return result += rhs;                          // rhs与它相加并返回结果
}
```

这个模板几乎与前面的程序相同　但是它们之间还是存在重要的差别　第二个模板包含一个命名对象　result　这个命名对象意味着不能在operator+ 里使用返回值优化　参见条款20　　第一种实现方法总可以使用返回值优化　所以编译器为其生成优化代码的可能就会更大

广告中的事实迫使我指出表达式

```
return T(lhs) += rhs;
```

比大多数编译器希望进行的返回值优化更复杂 上面第一个函数实现也有这样的临时对象开销　就象你为使用命名对象result而耗费的开销一样　然而未命名的对象在历史上比命名对象更容易清除 因此当我们面对在命名对象和临时对象间进行选择时　用临时对象更好一些　它使你耗费的开销不会比命名的对象还多　特别是使用老编译器时　它的耗费会更少

这里谈论的命名对象　未命名对象和编译优化是很有趣的　但是主要的一点是operator的赋值形式　operator+=　比单独形式(operator+)效率更高　做为一个库程序设计者　应该两者都提供　做为一个应用程序的开发者　在优先考虑性能时你应该考虑考虑用operator赋值形式代替单独形式

条款23　考虑变更程序库

程序库的设计就是一个折衷的过程　理想的程序库应该是短小的　快速的　强大的　灵活的　可扩展的　直观的　普遍适用的　具有良好的支持　没有使用约束　没有错误的　这也是不存在的　为尺寸和速度而进行优化的程序库一般不能被移植　具有大量功能的的程序库不会具有直观性　没有错误的程序库在使用范围上会有限制　真实的世界里　你不能拥有每一件东西　总得有付出
不同的设计者给这些条件赋予了不同的优先级　他们从而在设计中牺牲了不同的东西　因此一般两个提供相同功能的程序库却有着完全不同的性能特征
例如　考虑iostream和stdio程序库　对于C++程序员来说两者都是可以使用的　iostream程序库与C中的stdio相比有几个优点　参见Effective C++　　例如它是类型安全的　type-safe　　它是可扩展的　然而在效率方面　iostream程序库总是不如stdio　因为stdio产生的执行文件与iostream产生的执行文件相比尺寸小而且执行速度快
首先考虑执行速度的问题　要想掌握iostream和stdio之间的性能差别　一种方法就是用这两个程序库来运行benchmark程序　不过你必须记住benchmark也会撒谎　不仅很难拿出一组能够代表程序或程序库典型用法的数据　而且就算拿出来也是没用　除非有可靠的方法判断出你或你的客户的具有什么样的特征　不过在解决一个问题的不用方法的比较上　benchmark还是能够提供一些信息　所以尽管完全依靠benchmark是愚蠢的　但是忽略它们也是愚蠢的
让我们测试一个简单的benchmark程序　只测试最基本的I/O功能　这个程序从标准输入读取30000个浮点数　然后把它们以固定的格式写到标准输出里　编译时预处理符号STDIO决定是使用stdio还是iostream　如果定义了这个符号　就是用stdio　否则就使用iostream程序库

```
#ifdef STDIO
#include <stdio.h>
#else
#include <iostream>
#include <iomanip>
using namespace std;
#endif


const int VALUES = 30000;                    // # of values to read/write

int main()
{
   double d;
```

```
   for (int n = 1; n <= VALUES; ++n) {
#ifdef STDIO
        scanf("%lf", &d);
        printf("%10.5f", d);
#else
        cin >> d;
        cout    << setw(10)                                  //  设定field宽度
                << setprecision(5)                          //  设置小数位置
                << setiosflags(ios::showpoint)    // keep trailing 0s
                << setiosflags(ios::fixed)          //  使用这些设置
                << d;
#endif

        if (n % 5 == 0) {
#ifdef STDIO
            printf("\n");
#else
            cout << '\n';
#endif
        }
   }

   return 0;
}
```

当把正整数的自然对数传给这个程序　它会这样输出

```
0.00000     0.69315     1.09861     1.38629     1.60944
1.79176     1.94591     2.07944     2.19722     2.30259
2.39790     2.48491     2.56495     2.63906     2.70805
2.77259     2.83321     2.89037     2.94444     2.99573
3.04452     3.09104     3.13549     3.17805     3.21888
```

这种输出至少表明了使用iostreams也能这种也能产生fixed-format I/O　当然

```
cout    << setw(10)
          << setprecision(5)
          << setiosflags(ios::showpoint)
          << setiosflags(ios::fixed)
          << d;
```

远不如　printf("%10.5f", d); 输入方便

但是操作符<<既是类型安全 type-safe 又可以扩展 而printf则不具有这两种优点

我做了几种计算机 操作系统和编译器的不同组合 在其上运行这个程序 在每一种情况下都是使用stdio的程序运行得较快 优势它仅仅快一些 大约20 有时则快很多 接近200 但是我从来没有遇到过一种iostream的实现和与其相对应的stdio的实现运行速度一样快 另外 使用stdio的程序的尺寸比与相应的使用iostream的程序要小 有时是小得多 对于程序现实中的尺寸 这点差异就微不足道了

应该注意到stdio的高效性主要是由其代码实现决定的 所以我已经测试过的系统其将来的实现或者我没有测试过的系统的当前实现都可能表现出iostream和stdio并没有显著的差异 事实上 有理由相信会发现一种iostream的代码实现比stdio要快 因为iostream在编译时确定它们操作数的类型 而stdio的函数则是在运行时去解析格式字符串 format string iostream和stdio之间性能的对比不过是一个例子 这并不重要 重要的是具有相同功能的不同的程序库在性能上采取不同的权衡措施 所以一旦你找到软件的瓶颈 通过进行 profile 参见条款16 你应该知道是否可能通过替换程序库来消除瓶颈 比如如果你的程序有I/O瓶颈 你可以考虑用stdio替代iostream 如果程序在动态分配和释放内存上使用了大量时间 你可以想想是否有其他的operator new 和 operator delete的实现可用 参见条款8和Effective C++条款10 因为不同的程序库在效率 可扩展性 移植性 类型安全和其他一些领域上蕴含着不同的设计理念 通过变换使用给予性能更多考虑的程序库 你有时可以大幅度地提高软件的效率

条款24　理解虚拟函数　多继承　虚基类和RTTI所需的代价

此文包含一些图片　无法贴到文档区　所以我把word文档压成zip文件放在了文件交流区　请下载阅读　请下载

C++编译器们必须实现语言的每一个特性　这些实现的细节当然是由编译器来决定的　并且不同的编译器有不同的方法实现语言的特性　在多数情况下　你不用关心这些事情　然而有些特性的实现对对象大小和其成员函数执行速度有很大的影响　所以对于这些特性有一个基本的了解　知道编译器可能在背后做了些什么就显得很重要　这种特性中最重要的例子是虚拟函数

当调用一个虚拟函数时　被执行的代码必须与调用函数的对象的动态类型相一致　指向对象的指针或引用的类型是不重要的　编译器如何能够高效地提供这种行为呢　大多数编译器是使用virtual table和virtual table pointers　virtual table和virtual table pointers通常被分别地称为vtbl和vptr

一个vtbl通常是一个函数指针数组　　一些编译器使用链表来代替数组　但是基本方法是一样的　在程序中的每个类只要声明了虚函数或继承了虚函数　它就有自己的vtbl　并且类中vtbl的项目是指向虚函数实现体的指针　例如　如下这个类定义

```
class C1 {
public:
   C1();

   virtual ~C1();
   virtual void f1();
   virtual int f2(char c) const;
   virtual void f3(const string& s);

   void f4() const;

   ...
};
```

C1的virtual table数组看起来如下图所示

注意非虚函数f4不在表中　而且C1的构造函数也不在　非虚函数　包括构造函数它也被定义为非虚函数　就象普通的C函数那样被实现　所以有关它们的使用在性能上没有特殊的考虑

如果有一个C2类继承自C1　重新定义了它继承的一些虚函数　并加入了它自己的一些虚函数

```
class C2: public C1 {
public:
```

```
    C2();                                   // 非虚函数
    virtual ~C2();                          // 重定义函数
    virtual void f1();                      // 重定义函数
    virtual void f5(char *str);             // 新的虚函数
    ...
};
```

它的virtual table项目指向与对象相适合的函数　这些项目包括指向没有被C2重定义的C1虚函数的指针

这个论述引出了虚函数所需的第一个代价　你必须为每个包含虚函数的类的virtual talbe留出空间　类的vtbl的大小与类中声明的虚函数的数量成正比　包括从基类继承的虚函数　　每个类应该只有一个virtual table　所以virtual table所需的空间不会太大　但是如果你有大量的类或者在每个类中有大量的虚函数　你会发现vtbl会占用大量的地址空间

因为在程序里每个类只需要一个vtbl拷贝　所以编译器肯定会遇到一个棘手的问题　把它放在哪里　大多数程序和程序库由多个object　目标　文件连接而成　但是每个object文件之间是独立的　哪个object文件应该包含给定类的vtbl呢　你可能会认为放在包含main函数的object文件里　但是程序库没有main　而且无论如何包含main的源文件不会涉及很多需要vtbl的类 编译器如何知道它们被要求建立那一个vtbl呢

必须采取一种不同的方法　编译器厂商为此分成两个阵营　对于提供集成开发环境　包含编译程序和连接程序　的厂商　一种干脆的方法是为每一个可能需要vtbl的object文件生成一个vtbl拷贝　连接程序然后去除重复的拷贝　在最后的可执行文件或程序库里就为每个vtbl保留一个实例

更普通的设计方法是采用启发式算法来决定哪一个object文件应该包含类的vtbl　通常启发式算法是这样的　要在一个object文件中生成一个类的vtbl　要求该object文件包含该类的第一个非内联　非纯虚拟函数　non-inline non-pure virual function　定义　也就是类的实现体　　因此上述C1类的vtbl将被放置到包含C1::~C1定义的object文件里　不是内联的函数　　C2类的vtbl被放置到包含C1::~C2定义的object文件里　不是内联函数

实际当中　这种启发式算法效果很好　但是如果你过分喜欢声明虚函数为内联函数　参见Effective C++条款33　　如果在类中的所有虚函数都内声明为内联函数　启发式算法就会失败　大多数基于启发式算法的编译器会在每个使用它的object文件中生成一个类的vtbl　在大型系统里　这会导致程序包含同一个类的成百上千个vtbl拷贝　大多数遵循这种启发式算法的编译器会给你一些方法来人工控制vtbl的生成　但是一种更好的解决此问题的方法是避免把虚函数声明为内联函数　下面我们将看到　有一些原因导致现在的编译器一般总是忽略虚函数的的inline指令

Virtual table只实现了虚拟函数的一半机制　如果只有这些是没有用的　只有用某

种方法指出每个对象对应的vtbl时　它们才能使用　这是virtual table pointer的工作　它来建立这种联系

每个声明了虚函数的对象都带有它　它是一个看不见的数据成员　指向对应类的virtual table　这个看不见的数据成员也称为vptr　被编译器加在对象里　位置只有才编译器知道 从理论上讲 我们可以认为包含有虚函数的对象的布局是这样的　这幅图片表示vptr位于对象的底部　但是不要被它欺骗　不同的编译器放置它的位置也不同　存在继承的情况下　一个对象的vptr经常被数据成员所包围　如果存在多继承(Multiple inheritance)　这幅图片会变得更复杂　等会儿我们将讨论它　现在只需简单地记住虚函数所需的第二个代价是 在每个包含虚函数的类的对象里　你必须为额外的指针付出代价

如果对象很小　这是一个很大的代价　比如如果你的对象平均只有4比特的成员数据　那么额外的vptr会使成员数据大小增加一倍　假设vptr大小为4比特　　在内存受到限制的系统里　这意味着你必须减少建立对象的数量　即使在内存没有限制的系统里　你也会发现这会降低软件的性能　因为较大的对象有可能不适合放在缓存　cache　或虚拟内存页中(virtual memory page)　这就可能使得系统换页操作增多

假如我们有一个程序　包含几个C1和C2对象　对象　vptr和刚才我们讲到的vtbl之间的关系　在程序里我们可以这样去想象

考虑这段这段程序代码

```
void makeACall(C1 *pC1)
{
   pC1->f1();
}
```

通过指针pC1调用虚拟函数f1　仅仅看这段代码　你不会知道它调用的是那一个f1函数－－C1::f1或C2::f1　因为pC1可以指向C1对象也可以指向C2对象　尽管如此编译器仍然得为在makeACall的f1函数的调用生成代码　它必须确保无论pC1指向什么对象　函数的调用必须正确　编译器生成的代码会做如下这些事情

1　　通过对象的vptr找到类的vtbl　这是一个简单的操作　因为编译器知道在对象内哪里能找到vptr　毕竟是由编译器放置的它们　　因此这个代价只是一个偏移调整　以得到vptr　和一个指针的间接寻址　以得到vtbl

2　　找到对应vtbl内的指向被调用函数的指针　在上例中是f1　这也是很简单的　因为编译器为每个虚函数在vtbl内分配了一个唯一的索引　这步的代价只是在vtbl数组内的一个偏移

3　　调用第二步找到的的指针所指向的函数

如果我们假设每个对象有一个隐藏的数据叫做vptr　而且f1在vtbl中的索引为i　此语句

```
pC1->f1();
```

生成的代码就是这样的

```
(*pC1->vptr[i])(pC1);                    //调用被vtbl中第i个单元指
                                         // 向的函数  而pC1->vptr
                                         //指向的是vtbl  pC1被做为
                                         // this指针传递给函数
```

这几乎与调用非虚函数效率一样 在大多数计算机上它多执行了很少的一些指令 调用虚函数所需的代价基本上与通过函数指针调用函数一样 虚函数本身通常不是性能的瓶颈

在实际运行中 虚函数所需的代价与内联函数有关 实际上虚函数不能是内联的 这是因为"内联"是指"在编译期间用被调用的函数体本身来代替函数调用的指令 " 但是虚函数的"虚"是指"直到运行时才能知道要调用的是哪一个函数 "如果编译器在某个函数的调用点不知道具体是哪个函数被调用 你就能知道为什么它不会内联该函数的调用 这是虚函数所需的第三个代价 你实际上放弃了使用内联函数

当通过对象调用的虚函数时 它可以被内联 但是大多数虚函数是通过对象的指针或引用被调用的 这种调用不能被内联 因为这种调用是标准的调用方式 所以虚函数实际上不能被内联

到现在为止我们讨论的东西适用于单继承和多继承 但是多继承的引入 事情就会变得更加复杂 参见Effective C++条款43 这里详细论述其细节 但是在多继承里 在对象里为寻找vptr而进行的偏移量计算会变得更复杂 在单个对象里有多个vptr 一个基类对应一个 除了我们已经讨论过的单独的vtbl以外 还得为基类生成特殊的vtbl 因此增加了每个类和每个对象中的虚函数额外占用的空间 而且运行时调用所需的代价也增加了一些

多继承经常导致对虚基类的需求 没有虚基类 如果一个派生类有一个以上从基类的继承路径 基类的数据成员被复制到每一个继承类对象里 继承类与基类间的每条路径都有一个拷贝 程序员一般不会希望发生这种复制 而把基类定义为虚基类则可以消除这种复制 然而虚基类本身会引起它们自己的代价 因为虚基类的实现经常使用指向虚基类的指针做为避免复制的手段 一个或者更多的指针被存储在对象里

例如考虑下面这幅图 我经常称它为"恐怖的多继承菱形" the dreaded multiple inheritance diamond

这里A是一个虚基类 因为B和C虚拟继承了它 使用一些编译器 特别是比较老的编译器 D对象会产生这样布局

把基类的数据成员放在对象的最底端 这显得有些奇怪 但是它经常这么做 当然如何实现是编译器的自由 它们想怎么做都可以 这幅图只是虚基类如何导致对象需要额外指针的概念性描述 所以你不应该在此范围以外还使用这幅图 一些编译器可能加入更少的指针 还有一些编译器会使用某种方法而根本不加入额外的指针 这种编译器让vptr和vtbl负担双重责任

如果我们把这幅图与前面展示如何把virtual table pointer加入到对象里的图片合并起来 我们就会认识到如果在上述继承体系里的基类A有任何虚函数 对象D的

内存布局就是这样的
这里对象中被编译器加入的部分　我已经做了阴影处理　这幅图可能会有误导　因为阴影部分与非阴影部分之间的面积比例由类中数据量决定　对于小类　额外的代价就大　对于包含更多数据的类　相对来说额外的代价就不大　尽管也是值得注意的
还有一点奇怪的是虽然存在四个类　但是上述图表只有三个vptr　只要编译器喜欢　当然可以生成四个vptr　但是三个已经足够了　它发现B和D能够共享一个vptr　大多数编译器会利用这个机会来减少编译器生成的额外负担
我们现在已经看到虚函数能使对象变得更大　而且不能使用内联　我们已经测试过多继承和虚基类也会增加对象的大小　让我们转向最后一个话题　运行时类型识别　RTTI
RTTI能让我们在运行时找到对象和类的有关信息　所以肯定有某个地方存储了这些信息　让我们查询　这些信息被存储在类型为type_info的对象里　你能通过使用typeid操作符访问一个类的type_info对象
在每个类中仅仅需要一个RTTI的拷贝　但是必须有办法得到任何对象的信息　实际上这叙述得不是很准确　语言规范上这样描述　我们保证可以获得一个对象动态类型信息　如果该类型有至少一个虚函数　这使得RTTI数据似乎有些象virtual function talbe(虚函数表)　每个类我们只需要信息的一个拷贝　我们需要一种方法从任何包含虚函数的对象里获得合适的信息　这种RTTI和virtual function table之间的相似点并不是巧合　RTTI被设计为在类的vtbl基础上实现
例如　vtbl数组的索引0处可以包含一个type_info对象的指针　这个对象属于该vtbl相对应的类　上述C1类的vtbl看上去象这样
使用这种实现方法 RTTI耗费的空间是在每个类的vtbl中的占用的额外单元再加上存储type_info对象的空间　就象在多数程序里virtual table所占的内存空间并不值得注意一样　你也不太可能因为type_info对象大小而遇到问题
下面这个表各是对虚函数　多继承　虚基类以及RTTI所需主要代价的总结
FeatureIncreases
Size of ObjectsIncreases
Per-Class DataReduces
Inlining
Virtual FunctionsYesYesYes
Multiple InheritanceYesYesNo
Virtual Base ClassesOftenSometimesNo
RTTINoYesNo

一些人看到这个表格以后　会很吃惊　他们宣布“我还是应该使用C”　很好　但是请记住如果没有这些特性所提供的功能　你必须手工编码来实现　在多数情况下　你的人工模拟可能比编译器生成的代码效率更低　稳定性更差　例如使用嵌

套的switch语句或层叠的if　then　else语句模拟虚函数的调用　其产生的代码比虚函数的调用还要多　而且代码运行速度也更慢　再有　你必须自己人工跟踪对象类型　这意味着对象会携带它们自己的类型标签　type tag　　因此你不会得到更小的对象

理解虚函数　多继承　虚基类　RTTI所需的代价是重要的　但是如果你需要这些功能　不管采取什么样的方法你都得为此付出代价　理解这点也同样重要　有时你确实有一些合理的原因要绕过编译器生成的服务　例如隐藏的vptr和指向虚基类的指针会使得在数据库中存储C++对象或跨进程移动它们变得困难　所以你可能希望用某种方法模拟这些特性　能更加容易地完成这些任务　不过从效率的观点来看　你自己编写代码不可能做得比编译器生成的代码更好

技巧

本书涉及的大多数内容都是编程的指导准则 这些准则虽是重要的 但是程序员不能单靠准则生活 有一个很早以前的卡通片叫做“菲利猫” Felix the Cat ，菲利猫无论何时遇到困难 它都会拿它的trick包 如果一个卡通角色都有一个trick包那么C++程序员就更应该有了 把这一章想成你的trick包的启动器
当设计C++软件时 总会再三地受到一些问题的困扰 你如何让构造函数和非成员函数具有虚拟函数的特点 你如何限制一个类的实例的数量 你如何防止在堆中建立对象呢 你如何又能确保把对象建立在堆中呢 其它一些类的成员函数无论何时被调用 你如何能建立一个对象并让它自动地完成一些操作 你如何能让不同的对象共享数据结构 而让每个使用者以为它们每一个都拥有自己的拷贝 你如何区分operator[]的读操作和写操作 你如何建立一个虚函数 其行为特性依赖于不同对象的动态类型
所有这些问题 还有更多 都在本章得到解答 在本章里我叙述的都是C++程序员普遍遇到的问题 且解决方法也是已被证实了的 我把这些解决方法叫做技巧 不过当它们以程式化的风格 stylized fashion 被归档时 也被做为idiom和pattern 不管你把它称做什么 在你日复一日地从事软件开发工作时 下面这些信息都将使你受益 它会使你相信无论你做什么 总可以用C++来完成它

条款25 将构造函数和非成员函数虚拟化
从字面来看 谈论“虚拟构造函数”没有意义 当你有一个指针或引用 但是不知道其指向对象的真实类型是什么时 你可以调用虚拟函数来完成特定类型 type-specific 对象的行为 仅当你还没拥有一个对象但是你确切地知道想要对象的类型时 你才会调用构造函数 那么虚拟构造函数又从何谈起呢
很简单 尽管虚拟构造函数看起来好像没有意义 其实它们有非常大的用处 如果你认为没有意义的想法就没有用处 那么你怎么解释现代物理学的成就呢
因为现代物理学的主要成就是狭义 广义相对论 量子力学 这些理论看起来都好象很荒谬 不好理解 译者注 例如假设你编写一个程序 用来进行新闻报道的工作 一条新闻报道由文字或图片组成 你可以这样管理它们

```
class NLComponent {                    //用于 newsletter components
public:                                 // 的抽象基类

  ...                                   //包含只少一个纯虚函数
};

class TextBlock: public NLComponent {
public:
  ...                                   // 不包含纯虚函数
```

```
};

class Graphic: public NLComponent {
public:
    ...                                          // 不包含纯虚函数
};

class NewsLetter {                          // 一个 newsletter 对象
public:                                        // 由NLComponent 对象
    ...                                          // 的链表组成

private:
    list<NLComponent*> components;
};
```

类之间的关系如下图所示

在NewsLetter中使用的list类是一个标准模板类 STL STL是标准C++类库的一部分 参见Effective C++条款49和条款35 list类型对象的行为特性有些象双向链表尽管它没有以这种方法来实现

对象NewLetter不运行时就会存储在磁盘上 为了能够通过位于磁盘的替代物来建立Newsletter对象 让NewLetter的构造函数带有istream参数是一种很方便的方法 当构造函数需要一些核心的数据结构时 它就从流中读取信息

```
class NewsLetter {
public:
    NewsLetter(istream& str);
    ...
};
```

此构造函数的伪代码是这样的

```
NewsLetter::NewsLetter(istream& str)
{
    while (str) {
        从str读取下一个component对象;

        把对象加入到newsletter的 components
      对象的链表中去;
    }
}
```

或者 把这种技巧用于另一个独立出来的函数叫做readComponent 如下所示

```
class NewsLetter {
```

```
public:
    ...

private:
    // 为建立下一个NLComponent对象从str读取数据,
    // 建立component 并返回一个指针
    static NLComponent * readComponent(istream& str);
    ...
};

NewsLetter::NewsLetter(istream& str)
{
    while (str) {
        // 把readComponent返回的指针添加到components链表的最后
        // "push_back" 一个链表的成员函数 用来在链表最后进行插入操作
        components.push_back(readComponent(str));
    }
}
```

考虑一下readComponent所做的工作 它根据所读取的数据建立了一个新对象 或是TextBlock或是Graphic 因为它能建立新对象 它的行为与构造函数相似 而且因为它能建立不同类型的对象 我们称它为虚拟构造函数 虚拟构造函数是指能够根据输入给它的数据的不同而建立不同类型的对象 虚拟构造函数在很多场合下都有用处 从磁盘 或者通过网络连接 或者从磁带机上 读取对象信息只是其中的一个应用

还有一种特殊种类的虚拟构造函数－－虚拟拷贝构造函数－－也有着广泛的用途 虚拟拷贝构造函数能返回一个指针 指向调用该函数的对象的新拷贝 因为这种行为特性 虚拟拷贝构造函数的名字一般都是copySelf cloneSelf或者是象下面这样就叫做clone 很少会有函数能以这么直接的方式实现它

```
class NLComponent {
public:
    // declaration of virtual copy constructor
    virtual NLComponent * clone() const = 0;
    ...

};

class TextBlock: public NLComponent {
public:
```

```
    virtual TextBlock * clone() const               // virtual copy
    { return new TextBlock(*this); }                // constructor
    ...

};

class Graphic: public NLComponent {
public:
    virtual Graphic * clone() const                 // virtual copy
    { return new Graphic(*this); }                  // constructor
    ...

};
```

正如我们看到的 类的虚拟拷贝构造函数只是调用它们真正的拷贝构造函数 因此"拷贝"的含义与真正的拷贝构造函数相同 如果真正的拷贝构造函数只做了简单的拷贝 那么虚拟拷贝构造函数也做简单的拷贝 如果真正的拷贝构造函数做了全面的拷贝 那么虚拟拷贝构造函数也做全面的拷贝 如果真正的拷贝构造函数做一些奇特的事情 象引用计数或copy-on-write 参见条款29 那么虚拟构造函数也这么做 完全一致 太棒了

注意上述代码的实现利用了最近才被采纳的较宽松的虚拟函数返回值类型规则 被派生类重定义的虚拟函数不用必须与基类的虚拟函数具有一样的返回类型 如果函数的返回类型是一个指向基类的指针 或一个引用 那么派生类的函数可以返回一个指向基类的派生类的指针 或引用 这不是C++的类型检查上的漏洞 它使得又可能声明象虚拟构造函数这样的函数 这就是为什么TextBlock的clone函数能够返回TextBlock*和Graphic的clone能够返回Graphic*的原因 即使NLComponent的 clone返回值类型为NLComponent*

在NLComponent中的虚拟拷贝构造函数能让实现NewLetter的(正常的)拷贝构造函数变得很容易

```
class NewsLetter {
public:
    NewsLetter(const NewsLetter& rhs);
    ...

private:
    list<NLComponent*> components;
};

NewsLetter::NewsLetter(const NewsLetter& rhs)
```

```
{
  // 遍历整个rhs链表  使用每个元素的虚拟拷贝构造函数
  // 把元素拷贝进这个对象的component链表
  // 有关下面代码如何运行的详细情况  请参见条款35.
  for (list<NLComponent*>::const_iterator it =
            rhs.components.begin();
       it != rhs.components.end();
       ++it) {

    // "it"  指向rhs.components的当前元素  调用元素的clone函数
// 得到该元素的一个拷贝  并把该拷贝放到
//这个对象的component链表的尾端
    components.push_back((*it)->clone());
  }
}
```

如果你对标准模板库 STL 不熟悉 这段代码可能有些令人费解 不过原理很简单 遍历被拷贝的NewsLetter对象中的整个component链表 调用链表内每个元素对象的虚拟构造函数 我们在这里需要一个虚拟构造函数 因为链表中包含指向NLComponent对象的指针 但是我们知道其实每一个指针不是指向TextBlock对象就是指向Graphic对象 无论它指向谁 我们都想进行正确的拷贝操作 虚拟构造函数能够为我们做到这点

虚拟化非成员函数

就象构造函数不能真的成为虚拟函数一样 非成员函数也不能成为真正的虚拟函数 参加Effective C++ 条款19 然而 既然一个函数能够构造出不同类型的新对象是可以理解的 那么同样也存在这样的非成员函数 可以根据参数的不同动态类型而其行为特性也不同 例如 假设你想为TextBlock和Graphic对象实现一个输出操作符 显而易见的方法是虚拟化这个输出操作符 但是输出操作符是operator<< 函数把ostream&做为它的左参数 left-hand argument 即把它放在函数参数列表的左边 译者注 这就不可能使该函数成为TextBlock 或 Graphic成员函数

这样做也可以 不过看一看会发生什么

```
class NLComponent {
public:
  // 对输出操作符的不寻常的声明
  virtual ostream& operator<<(ostream& str) const = 0;
  ...
};
```

```
class TextBlock: public NLComponent {
public:
   // 虚拟输出操作符(同样不寻常)
   virtual ostream& operator<<(ostream& str) const;
};

class Graphic: public NLComponent {
public:
   // 虚拟输出操作符 (让就不寻常)
   virtual ostream& operator<<(ostream& str) const;
};

TextBlock t;
Graphic g;

...

t << cout;                                          // 通过virtual operator<<
                                                        //把t打印到cout中
                                                        // 不寻常的语法

g << cout;                                          //通过virtual operator<<
                                                        //把g打印到cout中
//不寻常的语法
```

类的使用者得把stream对象放到<<符号的右边 这与输出操作符一般的用发相反
为了能够回到正常的语法上来 我们必须把operator<<移出TextBlock 和 Graphic类
但是如果我们这样做 就不能再把它声明为虚拟了
另一种方法是为打印操作声明一个虚拟函数 例如print 把它定义在TextBlock 和 Graphic类里 但是如果这样 打印TextBlock 和 Graphic对象的语法就与使用operator<<做为输出操作符的其它类型的对象不一致了
这些解决方法都不很令人满意 我们想要的是一个称为operator<<的非成员函数 其具有象print虚拟函数的行为特性 有关我们想要什么的描述实际上已经很接近 如何得到它的描述 我们定义operator<< 和print函数 让前者调用后者

```
class NLComponent {
public:
   virtual ostream& print(ostream& s) const = 0;
   ...
```

```
};

class TextBlock: public NLComponent {
public:
    virtual ostream& print(ostream& s) const;
    ...

};

class Graphic: public NLComponent {
public:
    virtual ostream& print(ostream& s) const;
    ...

};

inline
ostream& operator<<(ostream& s, const NLComponent& c)
{
    return c.print(s);
}
```

具有虚拟行为的非成员函数很简单　你编写一个虚拟函数来完成工作　然后再写一个非虚拟函数　它什么也不做只是调用这个虚拟函数　为了避免这个句法花招引起函数调用开销　你当然可以内联这个非虚拟函数　参见Effective C++ 条款33　现在你知道如何根据它们的一个参数让非成员函数虚拟化　你可能想知道是否可能让它们根据一个以上的参数虚拟化呢　可以　但是不是很容易　有多困难呢　参见条款31　它将专门论述这个问题

条款26　限制某个类所能产生的对象数量

你很痴迷于对象　但是有时你又想束缚住你的疯狂　例如你在系统中只有一台打印机　所以你想用某种方式把打印机对象数目限定为一个　或者你仅仅取得16个可分发出去的文件描述符　所以应该确保文件描述符对象存在的数目不能超过16个　你如何能够做到这些呢　如何去限制对象的数量呢

如果这是一个用数学归纳法进行的证明　你会从n=1开始证明　然后从此出发推导出其它证明　幸运的是这既不是一个证明也不是一个归纳　而从n=0开始更具有启发性　所以我们就从这里开始　你如何能够彻底阻止对象实例化　instantiate呢

允许建立零个或一个对象

每次实例化一个对象时　我们很确切地知道一件事情　“将调用一个构造函数　”事实确实这样　阻止建立某个类的对象　最容易的方法就是把该类的构造函数声明在类的private域

```
class CantBeInstantiated {

private:

    CantBeInstantiated();

    CantBeInstantiated(const CantBeInstantiated&);



...



;
```

这样做以后　每个人都没有权力建立对象　我们能够有选择性地放松这个限制　例如如果想为打印机建立类　但是要遵守我们只有一个对象可用的约束　我们应把打印机对象封装在一个函数内　以便让每个人都能访问打印机　但是只有一个打印机对象被建立

```
class PrintJob;                                    // forward 声明
                                                   //  参见Effective C++条款34

class Printer {
public:
    void submitJob(const PrintJob& job);
    void reset();
    void performSelfTest();

...

friend Printer& thePrinter();

private:
    Printer();
    Printer(const Printer& rhs);

    ...
```

```
};


Printer& thePrinter()

{

    static Printer p;                                    // 单个打印机对象

    return p;

}
```

这个设计由三个部分组成　第一　Printer类的构造函数是private　这样品能阻止建立对象　第二　全局函数thePrinter被声明为类的友元　让thePrinter避免私有构造含函数引起的限制　最后thePrinter包含一个静态Printer对象　这意味着只有一个对象被建立

客户端代码无论何时要与系统的打印机进行交互访问 它都要使用thePrinter函数

```
class PrintJob {

public:

    PrintJob(const string& whatToPrint);

    ...



};


string buffer;
```

```
...                                                    //填充buffer


thePrinter().reset();

thePrinter().submitJob(buffer);
```

当然你感到thePrinter使用全局命名空间完全是多余的 “是的” 你会说 “全局函数看起来象全局变量 但是全局变量是gauche(不知如何翻译 译者注) 我想把所有与打印有关的功能都放到Printer类里 ”好的 我绝不敢与使用象gauche这样的词的人争论 有些不明白这里的意思译者注 这很简单 只要在Prihter类中声明thePrinter为静态函数 然后把它放在你想放的位置 就不再需要友元声明了 使用静态函数 如下所示

```
class Printer {

public:

    static Printer& thePrinter();

    ...


private:

    Printer();

    Printer(const Printer& rhs);

    ...


};
```

```
Printer& Printer::thePrinter()

{

    static Printer p;

    return p;

}
```

客户端使用printer时有些繁琐

```
Printer::thePrinter().reset();

Printer::thePrinter().submitJob(buffer);
```

另一种方法是把thePrinter移出全局域　放入namespace　命名空间　　参见Effective C++条款28　　命名空间是C + +一个较新的特性　任何能在全局域声明东西也能在命名空间里声明　包括类　结构　函数　变量　对象　typedef等等　把它们放入命名空间并不影响它们的行为特性　不过能够防止在不同命名空间里的实体发生命名冲突　把Printer类和thePrinter函数放入一个命名空间　我们就不用担心别人也会使用Printer和thePrinter名字　命名空间能够防止命名冲突

命名空间从句法上来看有些象类　但是它没有public　protected或private域　所有都是public　如下所示　我们把Printer　thePrinter放入叫做PrintingStuff的命名空间里

```
namespace PrintingStuff {

    class Printer {                                    //  在命名空间

    public:                                              // PrintingStuff中的类



        void submitJob(const PrintJob& job);
```

```
        void reset();
        void performSelfTest();
        ...

        friend Printer& thePrinter();

    private:
        Printer();
        Printer(const Printer& rhs);
        ...

};

Printer& thePrinter()                          // 这个函数也在命名空间里
{
        static Printer p;
        return p;
}
}
```

```
                                                    // 命名空间到此结束
```

使用这个命名空间后 客户端可以通过使用fully-qualified name 完全限制符名 即包括命名空间的名字

```
PrintingStuff::thePrinter().reset();

PrintingStuff::thePrinter().submitJob(buffer);
```

但是也可以使用using声明 以简化键盘输入

```
using PrintingStuff::thePrinter;        // 从命名空间"PrintingStuff"

                                                    //引入名字"thePrinter"

                                                     // 使其成为当前域

thePrinter().reset();                              // 现在可以象使用局部命名

thePrinter().submitJob(buffer);         // 一样 使用thePrinter
```

在thePrinter的实现上有两个微妙的不引人注目的地方 值得我们看一看 第一 单独的Pritner是位于函数里的静态成员而不是在类中的静态成员 这样做是非常重要的 在类中的一个静态对象实际上总是被构造 和释放 即使不使用该对象 与此相反 只有第一次执行函数时 才会建立函数中的静态对象 所以如果没有调用函数 就不会建立对象 不过你得为此付出代价 每次调用函数时都得检查是否需要建立对象 建立C++一个理论支柱是你不需为你不用的东西而付出 在函数里 把类似于Printer这样的对象定义为静态成员就是坚持这样的理论 你应该尽可能坚持这种理论

与一个函数的静态成员相比 把Printer声明为类中的静态成员还有一个缺点 它的初始化时间不确定 我们能够准确地知道函数的静态成员什么时候被初始化 “在第一次执行定义静态成员的函数时” 而没有定义一个类的静态成员被初始化的时间 C++为一个translation unit 也就是生成一个object文件的源代码的集合 内的静态成员的初始化顺序提供某种保证 但是对于在不同translation unit中的静态

成员的初始化顺序则没有这种保证　参见Effective C++条款47　　在实际使用中　这会给我们带来许多麻烦　当函数的静态成员能够满足我们的需要时　我们就能避免这些麻烦　在这里的例子里　既然它能够满足需要　我们为什么不用它呢

第二个细微之处是内联与函数内静态对象的关系　再看一下thePrinter的非成员函数形式

```
Printer& thePrinter()

{

    static Printer p;

    return p;

}
```

除了第一次执行这个函数时　也就是构造p时　　其它时候这就是一个一行函数——它由"return p;"一条语句组成　这个函数最适合做为内联函数使用　然而它不能被声明为内联　为什么呢　请想一想　为什么你要把对象声明为静态呢　通常是因为你只想要该对象的一个拷贝　现在再考虑“内联”意味着什么呢　从概念上讲　它意味着编译器用函数体替代该对函数的每一个调用　不过非成员函数除外　非成员函数还有其它的含义　它还意味着internal linkage　内部链接

通常情况下　你不需要理解这种语言上令人迷惑的东西　你只需记住一件事　“带有内部链接的函数可能在程序内被复制　也就是说程序的目标　object　代码可能包含一个以上的内部链接函数的代码　　这种复制也包括函数内的静态对象　”　结果如何　如果建立一个包含局部静态对象的非成员函数　你可能会使程序的静态对象的拷贝超过一个　所以不要建立包含局部静态数据的非成员函数

但是你可能认为建立函数来返回一个隐藏对象的引用　这种限制对象的数量的方法是错误的　也许你认为只需简单地计算对象的数目　一旦需要太多的对象　就抛出异常　这样做也许会更好　如下所示　这样建立printer对象

```
class Printer {

public:
```

```
    class TooManyObjects{};                         // 当需要的对象过多时
                                                    // 就使用这个异常类
    Printer();
    ~Printer();

    ...

private:
    static size_t numObjects;

    Printer(const Printer& rhs);                    // 这里只能有一个printer
                                                    // 所以不允许拷贝
};                                                  //    参见Effective C++ 条款27
```

此法的核心思想就是使用numObjects跟踪Pritner对象存在的数量　当构造类时　它的值就增加　释放类时　它的值就减少　如果试图构造过多的Printer对象　就会抛出一个TooManyObjects类型的异常

```
// Obligatory definition of the class static
size_t Printer::numObjects = 0;
```

```
Printer::Printer()
{
    if (numObjects >= 1) {
        throw TooManyObjects();
    }

    继续运行正常的构造函数;

    ++numObjects;
}

Printer::~Printer()
{
    进行正常的析构函数处理;

    --numObjects;
}
```

这种限制建立对象数目的方法有两个较吸引人的优点　一个是它是直观的　每个人都能理解它的用途　另一个是很容易推广它的用途　可以允许建立对象最多的数量不是一　而是其它大于一的数字

建立对象的环境

这种方法也有一个问题　假设我们一个特殊的打印机　是彩色打印机　这种打印机类有许多地方与普通的打印机类相同　所以我们从普通打印类继承下来

```
class ColorPrinter: public Printer {

    ...



};
```

现在假设我们系统有一个普通打印机和一个彩色打印机

```
Printer p;

ColorPrinter cp;
```

这两个定义会产生多少Pritner对象　答案是两个　一个是p　一个是cp　在运行时　当构造cp的基类部分时　会抛出TooManyObjects异常　对于许多程序员来说　这可不是他们所期望的事情　　设计时避免从其它的concrete类继承concrete类　就不会遇到这种问题　这种设计思想详见条款33

当其它对象包含Printer对象时　会发生同样的问题

```
class CPFMachine {                                  // 一种机器　可以复印　打印

private:                                            // 发传真


   Printer p;                                       // 有打印能力

   FaxMachine f;                                    // 有传真能力

   CopyMachine c;                                   // 有复印能力
```

```
    ...

};

CPFMachine m1;                          // 运行正常

CPFMachine m2;                          // 抛出 TooManyObjects异常
```

问题是Printer对象能存在于三种不同的环境中　只有它们本身　作为其它派生类的基类　被嵌入在更大的对象里　存在这些不同环境极大地混淆了跟踪“存在对象的数目”的含义　因为你心目中的“对象的存在”的含义与编译器不一致

通常你仅会对允许对象本身存在的情况感兴趣　你希望限制这种实例 instantiation　的数量　如果你使用最初的Printer类示例的方法　就很容易进行这种限制　因为Printer构造函数是private　　不存在friend声明　带有private构造函数的类不能作为基类使用　也不能嵌入到其它对象中

你不能从带有private构造函数的类派生出新类　这个事实导致产生了一种阻止派生类的通用方法　这种方法不需要和限制对象实例数量的方法一起使用　例如你有一个类FSA　表示一个finite state automata(有限态自动机)　　这种机器能用于很多环境下　比如用户界面设计　　并假设你允许建立任意数量的对象　但是你想禁止从FSA派生出新类　　这样做的一个原因是表明在FSA中存在非虚析构函数　Effective C++ 条款14解释了为什么基类通常需要虚拟析构函数　本书条款24解释了为什么没有虚函数的类比同等的具有虚函数的类要小　　如下所示　这样设计FSA可以满足你的这两点需求

```
class FSA {

public:
```

```
    // 伪构造函数
    static FSA * makeFSA();
    static FSA * makeFSA(const FSA& rhs);
    ...

private:
    FSA();
    FSA(const FSA& rhs);
    ...
};

FSA * FSA::makeFSA()
{ return new FSA(); }

FSA * FSA::makeFSA(const FSA& rhs)
{ return new FSA(rhs); }
```

不象thePrinter函数总是返回一个对象的引用　引用的对象是固定的　　每个makeFSA的伪构造函数则是返回一个指向对象的指针　指向的对象都是惟一的不相同的　　也就是说允许建立的FSA对象数量没有限制

那好　不过每个伪构造函数都调用new这个事实暗示调用者必须记住调用delete　否则就会发生资源泄漏　如果调用者希望退出生存空间时delete会被自动调用　他可以把makeFSA返回的指针存储在auto_ptr中　参见条款9　　当它们自己退出生存空间时　这种对象能自动地删除它们所指向的对象

```
// 间接调用缺省FSA构造函数

auto_ptr<FSA> pfsa1(FSA::makeFSA());



// indirectly call FSA copy constructor

auto_ptr<FSA> pfsa2(FSA::makeFSA(*pfsa1));



...                                    // 象通常的指针一样使用pfsa1和pfsa2,

                                       //不过不用操心删除它们
```

允许对象来去自由
我们知道如何设计只允许建立一个实例的类　我们知道跟踪特定类的对象数量的工作是复杂的　因为在三种不同的环境中都可能调用对象的构造函数　我们知道消除对象计数中混乱现象的方法是把构造函数声明为private　还有最后一点值得我们注意　使用thePrinter函数封装对单个对象的访问　以便把Printer对象的数量限制为一个　这样做的同时也会让我们在每一次运行程序时只能使用一个Printer对象　导致我们不能这样编写代码

```
建立 Printer 对象 p1;

使用 p1;

释放 p1;

建立Printer对象p2;

使用 p2;
```

释放 p2;

...
这种设计在同一时间里没有实例化多个Printer对象　而是在程序的不同部分使用了不同的Printer对象　不允许这样编写有些不合理　毕竟我们没有违反只能存在一个printer的约束　就没有办法使它合法化么
当然有　我们必须把先前使用的对象计数的代码与刚才看到的伪构造函数代码合并在一起

```
class Printer {
public:
    class TooManyObjects{};

    //  伪构造函数
    static Printer * makePrinter();

~Printer();

    void submitJob(const PrintJob& job);
    void reset();
    void performSelfTest();
    ...

private:
    static size_t numObjects;

    Printer();

    Printer(const Printer& rhs);              //我们不定义这个函数
};                                                    //因为不允许
                                                        //进行拷贝
                                                      // (参见Effective C++条款27)

// Obligatory definition of class static
size_t Printer::numObjects = 0;

Printer::Printer()
{
```

```
    if (numObjects >= 1) {
        throw TooManyObjects();
    }

    继续运行正常的构造函数;

    ++numObjects;
}

Printer * Printer::makePrinter()
{ return new Printer; }
```

当需要的对象过多时　会抛出异常　如果你认为这种方式给你的感觉是unreasonably harsh　你可以让伪构造函数返回一个空指针　当然客户端在使用之前应该进行检测

除了客户端必须调用伪构造函数　而不是真正的构造函数之外　它们使用Printer类就象使用其他类一样

```
Printer p1;                                    // 错误! 缺省构造函数是
                                               // private

Printer *p2 =
    Printer::makePrinter();                    // 正确, 间接调用
                                               // 缺省构造函数

Printer p3 = *p2;                              // 错误! 拷贝构造函数是
                                               // private

p2->performSelfTest();                         // 所有其它的函数都可以
p2->reset();                                   // 正常调用

...

delete p2;                                     // 避免内存泄漏　如果
                                               // p2 是一个 auto_ptr
                                               // 就不需要这步
```

这种技术很容易推广到限制对象为任何数量上　我们只需把hard-wired常量值1改为根据某个类而确定的数量　然后消除拷贝对象的约束　例如　下面这个经过修改的Printer类的代码实现　最多允许10个Printer对象存在

```
class Printer {
```

```
public:
    class TooManyObjects{};

    // 伪构造函数
    static Printer * makePrinter();
    static Printer * makePrinter(const Printer& rhs);

    ...

private:
    static size_t numObjects;
    static const size_t maxObjects = 10;            // 见下面解释

    Printer();
    Printer(const Printer& rhs);
};

// Obligatory definitions of class statics
size_t Printer::numObjects = 0;
const size_t Printer::maxObjects;

Printer::Printer()
{
    if (numObjects >= maxObjects) {
        throw TooManyObjects();
    }

    ...

}

Printer::Printer(const Printer& rhs)
{
    if (numObjects >= maxObjects) {
        throw TooManyObjects();
    }

    ...
```

```
}

Printer * Printer::makePrinter()
{ return new Printer; }

Printer * Printer::makePrinter(const Printer& rhs)
{ return new Printer(rhs); }
```

如果你的编译器不能编译上述类中Printer::maxObjects的声明　这丝毫也不奇怪　特别是应该做好准备　编译器不能编译把10做为初值赋给这个变量这条语句　给static const成员　例如int, char, enum等等　确定初值的功能是最近才加入到C++中的　所以一些编译器还不允许这样编写　如果没有及时更新你的编译器　可以把maxObjects声明为在一个private内匿名枚举类型里的枚举元素

```
class Printer {
private:
   enum { maxObjects = 10 };                    // 在类中,
   ...                                                  // maxObjects为常量10
};
```

或者象non-const static成员一样初始化static常量

```
class Printer {
private:
   static const size_t maxObjects;              // 没有赋给初值

   ...

};

// 放在一个代码实现的文件中
const size_t Printer::maxObjects = 10;
```

后面这种方法与原来的方法有一样的效果　但是显示地确定初值能让其他程序员更容易理解　当你的编译器支持在类定义中给const static成员赋初值的功能时　你应该尽可能地利用这个功能

一个具有对象计数功能的基类

把初始化静态成员撇在一边不说　上述的方法使用起来就像咒语一样灵验　但是另一方面它也有些繁琐　如果我们有大量像Printer需要限制实例数量的类　就必须一遍又一遍地编写一样的代码　每个类编写一次　这将会使大脑变得麻木　应

该有一种方法能够自动处理这些事情 难道没有方法把实例计数的思想封装在一个类里吗

我们很容易地能够编写一个具有实例计数功能的基类 然后让像Printer这样的类从该基类继承 而且我们能做得更好 我们使用一种方法封装全部的计数功能 不但封装维护实例计数器的函数 而且也封装实例计数器本身 当我们在条款29中测试引用计数时 将看到需要同样的技术 有关这种设计的测试细节 参见我的文章counting objects

Printer类的计数器是静态变量numObjects 我们应该把变量放入实例计数类中 然而也需要确保每个进行实例计数的类都有一个相互隔离的计数器 使用计数类模板可以自动生成适当数量的计数器 因为我们能让计数器成为从模板中生成的类的静态成员

```
template<class BeingCounted>
class Counted {
public:
  class TooManyObjects{};                          // 用来抛出异常

  static int objectCount() { return numObjects; }

protected:
  Counted();
  Counted(const Counted& rhs);

  ~Counted() { --numObjects; }

private:
  static int numObjects;
  static const size_t maxObjects;

  void init();                                     // 避免构造函数的
};                                                 // 代码重复

template<class BeingCounted>
Counted<BeingCounted>::Counted()
{ init(); }

template<class BeingCounted>
Counted<BeingCounted>::Counted(const Counted<BeingCounted>&)
{ init(); }
```

```
template<class BeingCounted>
void Counted<BeingCounted>::init()
{
   if (numObjects >= maxObjects) throw TooManyObjects();
   ++numObjects;
}
```

从这个模板生成的类仅仅能被做为基类使用 因此构造函数和析构函数被声明为protected 注意private成员函数init用来避免两个Counted构造函数的语句重复

现在我们能修改Printer类 这样使用Counted模板

```
class Printer: private Counted<Printer> {
public:
   // 伪构造函数
   static Printer * makePrinter();
   static Printer * makePrinter(const Printer& rhs);

   ~Printer();

   void submitJob(const PrintJob& job);
   void reset();
   void performSelfTest();
   ...

   using Counted<Printer>::objectCount;        // 参见下面解释
   using Counted<Printer>::TooManyObjects;   // 参见下面解释

private:
   Printer();
   Printer(const Printer& rhs);
};
```

Printer使用了Counter模板来跟踪存在多少Printer对象 坦率地说 除了Printer的编写者 没有人关心这个事实 它的实现细节最好是private 这就是为什么这里使用private继承的原因 参见Effective C++条款42 另一种方法是在Printer和counted<Printer>之间使用public继承 但是我们必须给Counted类一个虚拟析构函数 否则如果有人通过Counted<Printer>*指针删除一个Printer对象 我们就有导致对象行为不正确的风险——参见Effective C++条款14 条款24已经说得很明白了 在Counted中存在虚函数 几乎肯定影响从Counted继承下来的对象的大小和布局 我们不想引入这些额外的负担 所以使用private继承来避免这些负担

Counted所做的大部分工作对于Printer的客户端来说都是隐藏的　但是这些客户端可能很想知道有当前多少Printer对象存在　Counted模板提供了objectCount函数　用来提供这种信息　但是因为我们使用private继承　这个函数在Printer类中成为了private　为了恢复该函数的public访问权　我们使用using声明

```
class Printer: private Counted<Printer> {
public:
   ...
   using Counted<Printer>::objectCount; //  让这个函数对于Printer
                                                //是public
   ...
};
```

这样做是合乎语法规则的　但是如果你的编译器不支持命名空间　编译器就不允许这样做　如果这样的话　你应使用老式的访问权声明语法

```
class Printer: private Counted<Printer> {
public:
   ...
   Counted<Printer>::objectCount;          //  让objectCount
                                                //  在Printer中是public
   ...

};
```

这种更传统的语法与uning声明具有相同的含义　但是我们不赞成这样做
TooManyObjects类应该也应用同样的方式来处理　因为Printer的客户端如果要捕获这种异常类型　它们必须有能力访问TooManyObjects　.
当Printer继承Counted<Printer>时　它可以忘记有关对象计数的事情　编写Printer类时根本不用考虑对象计数　就好像有其他人会为它计数一样　Printer的构造函数可以是这样的

```
Printer::Printer()
{
   进行正常的构造函数运行
}
```

这里有趣的不是你所见到的东西　而是你看不到的东西　不检测对象的数量就好像限制将被超过　执行完构造函数后也不增加存在对象的数目　所有这些现在都由Counted<Printer>的构造函数来处理　因为Counted<Printer>是Printer的基类　我们知道Counted<Printer>的构造函数总在Printer的前面被调用　如果建立过多的对象Counted<Printer>的构造函数就会抛出异常　甚至都没有调用Printer的构造函数
最后还有一点需要注意　必须定义Counted内的静态成员　对于numObjects来说这很容易——我们只需要在Counted的实现文件里定义它即可

```
template<class BeingCounted>                    // 定义numObjects
int Counted<BeingCounted>::numObjects;          // 自动把它初始化为0
```

对于maxObjects来说　则有一些技巧　我们应该把它初始化为什么值呢　如果你想允许建立10个printer对象　我们应该初始化Counted<Printer>::maxObjects为10　另一方面如果我们向允许建立16个文件描述符对象　我们应该初始化Counted<Printer>::maxObjects为16　到底应该怎么做呢

简单的方法就是什么也不做　我们不对maxObject进行初始化　而是让此类的客户端提供合适的初始化　Printer的作者必须把这条语句加入到一个实现文件里

```
const size_t Counted<Printer>::maxObjects = 10;
```

同样FileDescriptor的作者也得加入这条语句

```
const size_t Counted<FileDescriptor>::maxObjects = 16;
```

如果这些作者忘了对maxObjects进行初始化　会发生什么情况呢　很简单　连接时会发生错误 因为maxObjects没有被定义 如果我们提供了充分的文档对Counted客户端说明了需求　他们会回去加上这个必须的初始化

译者注

translation unit - a source file presented to a complier with an object file produced as a result.

linkage - refers to whether a name is visible only inside or also outside its translation unit.

条款 27　要求或禁止在堆中产生对象

有时你想这样管理某些对象　要让某种类型的对象能够自我销毁　也就是能够　delete this.　很明显这种管理方式需要此类型对象要被分配在堆中　而其它一些时候你想获得一种保障　不在堆中分配对象　从而保证某种类型的类不会发生内存泄漏　如果你在嵌入式系统上工作　就有可能遇到这种情况　发生在嵌入式系统上的内存泄漏是极其严重的　其堆空间是非常珍贵的　有没有可能编写出代码来要求或禁止在堆中产生对象　heap-based object　呢　通常是可以的　不过这种代码也会把　on the heap"的概念搞得比你脑海中所想的要模糊

要求在堆中建立对象

让我们先从必须在堆中建立对象开始说起　为了执行这种限制　你必须找到一种方法禁止以调用"new"以外的其它手段建立对象　这很容易做到　非堆对象　non-heap object　在定义它的地方被自动构造　在生存时间结束时自动被释放　所以只要禁止使用隐式的构造函数和析构函数　就可以实现这种限制

把这些调用变得不合法的一种最直接的方法是把构造函数和析构函数声明为 private　这样做副作用太大　没有理由让这两个函数都是 private　最好让析构函数成为 private　让构造函数成为 public　处理过程与条款 26 相似　你可以引进一个专用的伪析构函数　用来访问真正的析构函数　客户端调用伪析构函数释放他们建立的对象

例如　如果我们想仅仅在堆中建立代表 unlimited precision numbers　无限精确度数字　的对象　可以这样做

```
class UPNumber {
public:
   UPNumber();
   UPNumber(int initValue);
   UPNumber(double initValue);
   UPNumber(const UPNumber& rhs);

   // 伪析构函数 (一个 const 成员函数　因为
   // 即使是 const 对象也能被释放　)
   void destroy() const { delete this; }

   ...

private:
   ~UPNumber();
};
```

然后客户端这样进行程序设计

```
UPNumber n;                          // 错误! (在这里合法　但是
                                     // 当它的析构函数被隐式地
                                     // 调用时　就不合法了)

UPNumber *p = new UPNumber;          //正确
```

```
...

delete p;                                   // 错误! 试图调用
                                            // private 析构函数

p->destroy();                               // 正确
```

另一种方法是把全部的构造函数都声明为 private　这种方法的缺点是一个类经常有许多构造函数　类的作者必须记住把它们都声明为 private　否则如果这些函数就会由编译器生成　构造函数包括拷贝构造函数　也包括缺省构造函数　编译器生成的函数总是 public　参见 Effecitve C++ 条款 45　因此仅仅声明析构函数为 private 是很简单的　因为每个类只有一个析构函数

通过限制访问一个类的析构函数或它的构造函数来阻止建立非堆对象　但是在条款 26 已经说过　这种方法也禁止了继承和包容　containment

```
class UPNumber { ... };                     // 声明析构函数或构造函数
                                            // 为 private

class NonNegativeUPNumber:
  public UPNumber { ... };                  // 错误! 析构函数或
                                            //构造函数不能编译

class Asset {
private:
  UPNumber value;
  ...                                       // 错误! 析构函数或
                                            //构造函数不能编译
};
```

这些困难不是不能克服的　通过把 UPNumber 的析构函数声明为 protected　同时它的构造函数还保持 public　就可以解决继承的问题　需要包含 UPNumber 对象的类可以修改为包含指向 UPNumber 的指针

```
class UPNumber { ... };                     // 声明析构函数为 protected

class NonNegativeUPNumber:
  public UPNumber { ... };                  // 现在正确了; 派生类
                                            // 能够访问
                                            // protected 成员

class Asset {
public:
  Asset(int initValue);
  ~Asset();
  ...

private:
```

```
    UPNumber *value;
};

Asset::Asset(int initValue)
: value(new UPNumber(initValue))          // 正确
{ ... }

Asset::~Asset()
{ value->destroy(); }                     // 也正确
```

判断一个对象是否在堆中

如果我们采取这种方法　我们必须重新审视一下　在堆中　这句话的含义　上述粗略的类定义表明一个非堆的 NonNegativeUPNumber 对象是合法的

```
NonNegativeUPNumber n;                    // 正确
```

那么现在 NonNegativeUPNumber 对象 n 中的 UPNumber 部分也不在堆中　这样说对么　答案要依据类的设计和实现的细节而定　但是让我们假设这样说是不对的　所有 UPNumber 对象　即使是做为其它派生类的基类　也必须在堆中　我们如何能强制执行这种约束呢

没有简单的办法　UPNumber 的构造函数不可能判断出它是否做为堆对象的基类而被调用　也就是说对于 UPNumber 的构造函数来说没有办法侦测到下面两种环境的区别

```
NonNegativeUPNumber *n1 =
    new NonNegativeUPNumber;              // 在堆中

NonNegativeUPNumber n2;                   //不再堆中
```

不过你可能不相信我　也许你想你能够在 new 操作符　operator new 和 new　操作符调用的构造函数的相互作用中玩些小把戏　参见条款 8　可能你认为你比他们都聪明　可以这样修改 UPNumber　如下所示

```
class UPNumber {
public:
    // 如果建立一个非堆对象　抛出一个异常
    class HeapConstraintViolation {};

    static void * operator new(size_t size);

    UPNumber();
    ...

private:
    static bool onTheHeap;                //在构造函数内　指示
                                          // 对象是否被构造在
    ...                                   // 堆上

};
```

```
// obligatory definition of class static
bool UPNumber::onTheHeap = false;

void *UPNumber::operator new(size_t size)
{
  onTheHeap = true;
  return ::operator new(size);
}

UPNumber::UPNumber()
{
  if (!onTheHeap) {
    throw HeapConstraintViolation();
  }

  proceed with normal construction here;

  onTheHeap = false;                          // 为下一个对象清除标记
}
```

如果不再深入研究下去　就不会发现什么错误　这种方法利用了这样一个事实　当在堆上分配对象时　会调用 operator new 来分配 raw memory　operator new 设置 onTheHeap 为 true　每个构造函数都会检测 onTheHeap　看对象的 raw memory 是否被 operator new 所分配　如果没有　一个类型为 HeapConstraintViolation 的异常将被抛出　否则构造函数如通常那样继续运行　当构造函数结束时　onTheHeap 被设置为 false　然后为构造下一个对象而重置到缺省值

这是一个非常好的方法　但是不能运行　请考虑一下这种可能的客户端代码

```
UPNumber *numberArray = new UPNumber[100];
```

第一个问题是为数组分配内存的是 operator new[]　而不是 operator new　不过倘若你的编译器支持它　你能象编写 operator new 一样容易地编写 operator new[]函数　更大的问题是 numberArray 有 100 个元素　所以会调用 100 次构造函数　但是只有一次分配内存的调用　所以 100 个构造函数中只有第一次调用构造函数前把 onTheHeap 设置为 true　当调用第二个构造函数时　会抛出一个异常　你真倒霉

即使不用数组　bit-setting 操作也会失败　考虑这条语句

```
UPNumber *pn = new UPNumber(*new UPNumber);
```

这里我们在堆中建立两个 UPNumber　让 pn 指向其中一个对象　这个对象用另一个对象的值进行初始化　这个代码有一个内存泄漏　我们先忽略这个泄漏这有利于下面对这条表达式的测试　执行它时会发生什么事情

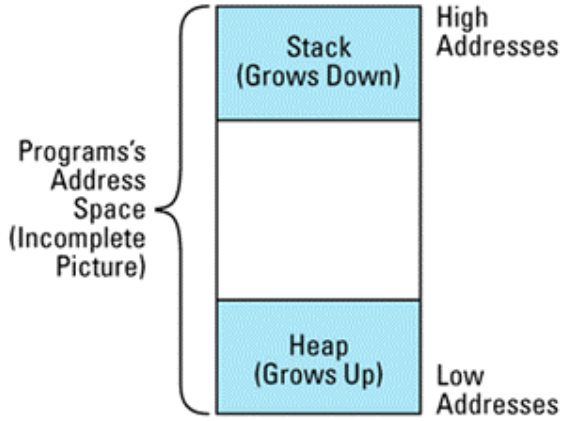
```
new UPNumber(*new UPNumber)
```

它包含 new 操作符的两次调用　因此要调用两次 operator new 和调用两次 UPNumber 构造函数　参见条款 8　程序员一般期望这些函数以如下顺序执行

调用第一个对象的 operator new

调用第一个对象的构造函数

调用第二个对象的 operator new
调用第二个对象的构造函数
但是 C++语言没有保证这就是它调用的顺序　一些编译器以如下这种顺序生成函数调用
调用第一个对象的 operator new
调用第二个对象的 operator new
调用第一个对象的构造函数
调用第二个对象的构造函数
编译器生成这种代码丝毫没有错　但是在 operator new 中 set-a-bit 的技巧无法与这种编译器一起使用　因为在第一步和第二步设置的 bit　第三步中被清除　那么在第四步调用对象的构造函数时　就会认为对象不再堆中　即使它确实在
这些困难没有否定让每个构造函数检测*this 指针是否在堆中这个方法的核心思想　它们只是表明检测在 operator new(或 operator new[])里的 bit set 不是一个可靠的判断方法　我们需要更好的方法进行判断
如果你陷入了极度绝望当中　你可能会沦落进不可移植的领域里　例如你决定利用一个在很多系统上存在的事实　程序的地址空间被做为线性地址管理　程序的栈从地址空间的顶部向下扩展　堆则从底部向上扩展



在以这种方法管理程序内存的系统里　很多系统都是　但是也有很多不是这样　你可能会想能够使用下面这个函数来判断某个特定的地址是否在堆中

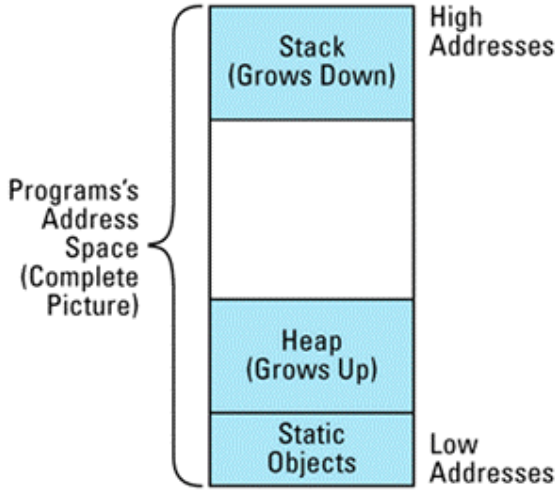
```
// 不正确的尝试　来判断一个地址是否在堆中
bool onHeap(const void *address)
{
   char onTheStack;                          // 局部栈变量

   return address < &onTheStack;
}
```

这个函数背后的思想很有趣　在 onHeap 函数中 onTheSatck 是一个局部变量

因此它在堆栈上 当调用 onHeap 时 它的栈框架 stack frame (也就是它的 activation record)被放在程序栈的顶端 因为栈在结构上是向下扩展的 趋向低地址 onTheStack 的地址肯定比任何栈中的变量或对象的地址小 如果参数 address 的地址小于 onTheStack 的地址 它就不会在栈上 而是肯定在堆上

到目前为止 这种逻辑很正确 但是不够深入 最根本的问题是对象可以被分配在三个地方 而不是两个 是的 栈和堆能够容纳对象 但是我们忘了静态对象 静态对象是那些在程序运行时仅能初始化一次的对象 静态对象不仅仅包括显示地声明为 static 的对象 也包括在全局和命名空间里的对象 参见条款 47 这些对象肯定位于某些地方 而这些地方既不是栈也不是堆

它们的位置是依据系统而定的 但是在很多栈和堆相向扩展的系统里 它们位于堆的底端 先前内存管理的图片到讲述的是事实 不过是很多系统都具有的事实 但是没有告诉我们这些系统全部的事实 加上静态变量后 这幅图片如下所示



onHeap 不能工作的原因立刻变得很清楚了 不能辨别堆对象与静态对象的区别

```
void allocateSomeObjects()
{
  char *pc = new char;                    // 堆对象: onHeap(pc)
                                          // 将返回 true

  char c;                                 // 栈对象: onHeap(&c)
                                          // 将返回 false
```

```
    static char sc;                          // 静态对象: onHeap(&sc)
                                             // 将返回 true
    ...

}
```

现在你可能不顾一切地寻找区分堆对象与栈对象的方法　在走头无路时你想在可移植性上打主意　但是你会这么孤注一掷地进行一个不能获得正确结果的交易么　绝对不会　我知道你会拒绝使用这种虽然诱人但是不可靠的　地址比对技巧

令人伤心的是不仅没有一种可移植的方法来判断对象是否在堆上　而且连能在多数时间正常工作的　准可移植　的方法也没有　如果你实在非得必须判断一个地址是否在堆上　你必须使用完全不可移植的方法　其实现依赖于系统调用　只能这样做了　因此你最好重新设计你的软件　以便你可以不需要判断对象是否在堆中

如果你发现自己实在为对象是否在堆中这个问题所困扰　一个可能的原因是你想知道对象是否能在其上安全调用 delete　这种删除经常采用　delete this　这种声明狼籍的形式　不过知道　是否能安全删除一个指针　与　只简单地知道一个指针是否指向堆中的事物　不一样　因为不是所有在堆中的事物都能被安全地 delete　再考虑包含 UPNumber 对象的 Asset 对象

```
class Asset {
private:
    UPNumber value;
    ...

};

Asset *pa = new Asset;
```

很明显*pa　包括它的成员 value　在堆上　同样很明显在指向 pa->value 上调用 delete 是不安全的　因为该指针不是被 new 返回的

幸运的是　判断是否能够删除一个指针　比　判断一个指针指向的事物是否在堆上　要容易　因为对于前者我们只需要一个 operator new 返回的地址集合　因为我们能自己编写 operator new 函数　参见 Effective C++条款 8　条款 10　所以构建这样一个集合很容易　如下所示　我们这样解决这个问题

```
void *operator new(size_t size)
{
    void *p = getMemory(size);               //调用一些函数来分配内存
                                             //处理内存不够的情况

    把 p 加入到一个被分配地址的集合;

    return p;

}
```

```
void operator delete(void *ptr)
{
  releaseMemory(ptr);                    // return memory to
                                         // free store

  从被分配地址的集合中移去 ptr;
}

bool isSafeToDelete(const void *address)
{
  返回 address 是否在被分配地址的集合中;
}
```

这很简单　operator new 在地址分配集合里加入一个元素　operator delete 从集合中移去项目　isSafeToDelete 在集合中查找并确定某个地址是否在集合中　如果 operator new 和 operator delete 函数在全局作用域中　它就能适用于所有的类型　甚至是内建类型

在实际当中　有三种因素制约着对这种设计方式的使用　第一是我们极不愿意在全局域定义任何东西　特别是那些已经具有某种含义的函数　象 operator new 和 operator delete　正如我们所知　只有一个全局域　只有一种具有正常特征形式　也就是参数类型　的 operator new 和 operator delete　这样做会使得我们的软件与其它也实现全局版本的 operator new 和 operator delete 的软件　例如许多面向对象数据库系统　不兼容

我们考虑的第二个因素是效率　如果我们不需要这些　为什么还要为跟踪返回的地址而负担额外的开销呢

最后一点可能有些平常　但是很重要　实现 isSafeToDelete 让它总能够正常工作是不可能的　难点是多继承下来的类或继承自虚基类的类有多个地址　所以无法保证传给 isSafeToDelete 的地址与 operator new 返回的地址相同　即使对象在堆中建立　有关细节参见条款 24 和条款 31

我们希望这些函数提供这些功能时能够不污染全局命名空间　没有额外的开销　没有正确性问题　幸运的是 C++使用一种抽象 mixin 基类满足了我们的需要

抽象基类是不能被实例化的基类　也就是至少具有一个纯虚函数的基类　mixin(mix in)类提供某一特定的功能 并可以与其继承类提供的其它功能相兼容 参见 Effective C++条款 7　这种类几乎都是抽象类　因此我们能够使用抽象混合　mixin　基类给派生类提供判断指针指向的内存是否由 operator new 分配的能力　该类如下所示

```
class HeapTracked {                      // 混合类; 跟踪
public:                                  // 从 operator new 返回的 ptr

  class MissingAddress{};                // 异常类　见下面代码

  virtual ~HeapTracked() = 0;

  static void *operator new(size_t size);
```

```
    static void operator delete(void *ptr);

    bool isOnHeap() const;

private:
    typedef const void* RawAddress;
    static list<RawAddress> addresses;
};
```

这个类使用了 list 链表 数据结构跟踪从 operator new 返回的所有指针 list 标准 C++库的一部分 参见 Effective C++条款 49 和本书条款 35 operator new 函数分配内存并把地址加入到 list 中 operator delete 用来释放内存并从 list 中移去地址元素 isOnHeap 判断一个对象的地址是否在 list 中

HeapTracked 类的实作 我觉得把 implementation 翻译成 实作 更好 译者注很简单 调用全局的 operator new 和 operator delete 函数来完成内存的分配与释放 list 类里的函数进行插入操作和删除操作 并进行单语句的查找操作 以下是 HeapTracked 的全部实作

```
// mandatory definition of static class member
list<RawAddress> HeapTracked::addresses;

// HeapTracked 的析构函数是纯虚函数 使得该类变为抽象类
// (参见 Effective C++条款 14). 然而析构函数必须被定义
//所以我们做了一个空定义 .
HeapTracked::~HeapTracked() {}



void * HeapTracked::operator new(size_t size)
{
    void *memPtr = ::operator new(size);   //  获得内存

    addresses.push_front(memPtr);              //  把地址放到 list 的前端
    return memPtr;
}

void HeapTracked::operator delete(void *ptr)
{
    //得到一个 "iterator"  用来识别 list 元素包含的 ptr
    //有关细节参见条款 35
    list<RawAddress>::iterator it =
        find(addresses.begin(), addresses.end(), ptr);

    if (it != addresses.end()) {          //  如果发现一个元素
        addresses.erase(it);                    //则删除该元素
        ::operator delete(ptr);              //  释放内存
```

```
    } else {                                    // 否则
        throw MissingAddress();                 // ptr 就不是用 operator new
    }                                           // 分配的  所以抛出一个异常
}

bool HeapTracked::isOnHeap() const
{
    // 得到一个指针  指向*this 占据的内存空间的起始处
    // 有关细节参见下面的讨论
    const void *rawAddress = dynamic_cast<const void*>(this);

    // 在 operator new 返回的地址 list 中查到指针
    list<RawAddress>::iterator it =
        find(addresses.begin(), addresses.end(), rawAddress);

    return it != addresses.end();           // 返回 it 是否被找到
}
```

尽管你可能对 list 类和标准 C++库的其它部分不很熟悉 代码还是很一目了然 条款 35 将解释这里的每件东西 不过代码里的注释已经能够解释这个例子是如何运行的

只有一个地方可能让你感到困惑 就是这个语句 在 isOnHeap 函数中

```
const void *rawAddress = dynamic_cast<const void*>(this);
```

我前面说过带有多继承或虚基类的对象会有几个地址 这导致编写全局函数 isSafeToDelete 会很复杂 这个问题在 isOnHeap 中仍然会遇到 但是因为 isOnHeap 仅仅用于 HeapTracked 对象中 我们能使用 dynamic_cast 操作符的一种特殊的特性来消除这个问题 只需简单地放入 dynamic_cast 把一个指针 dynamic_cast 成 void* 类型 或 const void*或 volatile void* 生成的指针指向 原指针指向对象内存 的开始处 但是 dynamic_cast 只能用于 指向至少具有一个虚拟函数的对象的指针上 我们该死的 isSafeToDelete 函数可以用于指向任何类型的指针 所以 dynamic_cast 也不能帮助它 isOnHeap 更具有选择性 它只能测试指向 HeapTracked 对象的指针 所以能把 this 指针 dynamic_cast 成 const void* 变成一个指向当前对象起始地址的指针 如果 HeapTracked::operator new 为当前对象分配内存 这个指针就是 HeapTracked::operator new 返回的指针 如果你的编译器支持 dynamic_cast 操作符 这个技巧是完全可移植的

使用这个类 即使是最初级的程序员也可以在类中加入跟踪堆中指针的功能 他们所需要做的就是让他们的类从 HeapTracked 继承下来 例如我们想判断 Assert 对象指针指向的是否是堆对象

```
class Asset: public HeapTracked {
private:
    UPNumber value;
    ...
};
```

我们能够这样查询 Assert*指针 如下所示

```
void inventoryAsset(const Asset *ap)
```

```
{
  if (ap->isOnHeap()) {
    ap is a heap-based asset — inventory it as such;
  }
  else {
    ap is a non-heap-based asset — record it that way;
  }
}
```

象 HeapTracked 这样的混合类有一个缺点　它不能用于内建类型　因为象 int 和 char 这样的类型不能继承自其它类型　不过使用象 HeapTracked 的原因一般都是要判断是否可以调用"delete this"　你不可能在内建类型上调用它　因为内建类型没有 this 指针

禁止堆对象

判断对象是否在堆中的测试到现在就结束了　与此相反的领域是　禁止在堆中建立对象　　通常对象的建立这样三种情况　对象被直接实例化　对象做为派生类的基类被实例化　对象被嵌入到其它对象内　我们将按顺序地讨论它们

禁止客户端直接实例化对象很简单　因为总是调用 new 来建立这种对象　你能够禁止客户端调用 new 你不能影响 new 操作符的可用性 这是内嵌于语言的　但是你能够利用 new 操作符总是调用 operator new 函数这点　参见条款 8　　来达到目的　你可以自己声明这个函数　而且你可以把它声明为 private.　例如　如果你想不想让客户端在堆中建立 UPNumber 对象　你可以这样编写

```
class UPNumber {
private:
  static void *operator new(size_t size);
  static void operator delete(void *ptr);
  ...
};
```

现在客户端仅仅可以做允许它们做的事情

```
UPNumber n1;                                    // okay

static UPNumber n2;                             // also okay

UPNumber *p = new UPNumber;                     // error! attempt to call
                                                // private operator new
```

把 operator new 声明为 private 就足够了　但是把 operator new 声明为 private　而把 iperator delete 声明为 public　这样做有些怪异　所以除非有绝对需要的原因　否则不要把它们分开声明　最好在类的一个部分里声明它们　如果你也想禁止 UPNumber 堆对象数组　可以把 operator new[]和 operator delete[]　参见条款 8　也声明为 private　　operator new 和 operator delete 之间的联系比大多数人所想象的要强得多　有关它们之间关系的鲜为人知的一面　可以参见我的文章 counting objects 里的 sidebar 部分

有趣的是　把 operator new 声明为 private 经常会阻碍 UPNumber 对象做为一个位于堆中的派生类对象的基类被实例化　因为如果 operator new 和 operator delete

没有在派生类中被声明为 public　它们就会被继承下来　继承了基类 private 函数的类　如下所示

```
class UPNumber { ... };                    //  同上

class NonNegativeUPNumber:                 //假设这个类
   public UPNumber {                        //没有声明 operator new
   ...
};

NonNegativeUPNumber n1;                    //  正确

static NonNegativeUPNumber n2;             //  也正确

NonNegativeUPNumber *p =                   //  错误!  试图调用
   new NonNegativeUPNumber;                // private operator new
```

如果派生类声明它自己的 operator new　当在堆中分配派生对象时　就会调用这个函数　必须得找到一种不同的方法防止 UPNumber 基类部分缠绕在这里　同样　UPNumber 的 operator new 是 private 这一点　不会对分配包含做为成员的 UPNumber 对象的对象产生任何影响

```
class Asset {
public:
   Asset(int initValue);
   ...

private:
   UPNumber value;
};

Asset *pa = new Asset(100);                //  正确,  调用
                                           // Asset::operator new  或
                                           // ::operator new,  不是
                                           // UPNumber::operator new
```

实际上　我们又回到了这个问题上来　即　如果 UPNumber 对象没有被构造在堆中　我们想抛出一个异常　当然这次的问题是　如果对象在堆中　我们想抛出异常　正像没有可移植的方法来判断地址是否在堆中一样　也没有可移植的方法判断地址是否不在堆中　所以我们很不走运　不过这也丝毫不奇怪　毕竟如果我们能辨别出某个地址在堆上　我们也就能辨别出某个地址不在堆上　但是我们什么都不能辨别出来