规则 1:

```
/*仔细区别 pointers 和 reference*/
/*
 *  没有所谓的 null reference  一个 reference 必须代表一个对象
 * pointer 可以设为 null
 * /

//下面的代码是有害的  可能导致不可预期的行为
  char *pc=0;
  char& rc=*pc;

//c++要求  reference 必须有初值
string& rs;//错误
string s("xyzxyz");
string& rs=s;//ok

//pointer 可以没有初值
string* ps;

//reference 比 pointer 更富效率  使用前不需要测试有效性
void printDouble(const double& rd)
   {
     cout<<rd;
     }
  void printDouble(const double* rd)
   {
     if(rd)
     {
     cout<<*rd;
     }
  }

  //pointer 可以被重新赋值，而 reference 却总是指向他或初值的对象

  string s1("nacy");
  string s2("clancy");

  string& rs=s1;
  string *ps=&s1;
  rs=s2;

  ps=&s2;

  /*
```

*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
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;
```

规则 2:

```
/*最好使用 c++转型操作符*/
/*
 *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)
 *
 */

/*
 *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。
 */
   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;

/*
 *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 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.
 */
   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。

/*
 *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 M24），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。

       /*
        *这四个类型转换符中的最后一个是 reinterpret_cast。
```

```
*使用这个操作符的类型转换，其的转换结果几乎都是执行期定义
*（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
*/

typedef void (*FuncPtr)(); // FuncPtr is  一个指向函数
                           //  的指针，该函数没有参数
                          //  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
int doSomething();

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] = reinterpret_cast<FuncPtr>(&doSomething);// this compiles
```

规则 3:

/*绝不要以多态方式处理数组*/

```
class BST { ... };
class BalancedBST: public BST { ... };

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 i*sizeof(BST)。
 */

//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。
//以下是一种你可能采用的但不正确的做法。

//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 =new BalancedBST[50];
                        // 建立一个 BalancedBST 对象数组
...
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确运行，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。
```

规则 4:

```
/*非必要不要提供 default constructor*/
/*
 * 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
 */

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

    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

//解决方法
    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)
    };


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

```
//如果你为数组分配 raw memory，你就可以避免浪费内存。
// 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 M8
// 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 M8)
for (int i = 0; i < 10; ++i)
        new (&bestPieces[i]) EquipmentPiece( ID Number );

//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，然后调用操作符 delete[]来释放 raw memory

// 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而使用了普通的数组删除方法，
//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。
delete [] bestPieces; // 没有定义! bestPieces 不是用 new 操作符分配的。


//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）的容器类里使用。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()
}
//不提供缺省构造函数的虚基类，很难与其进行合作。
//因为几乎所有的派生类在实例化时都必须给虚基类
//构造函数提供参数。这就要求所有由没有缺省构造
//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的含义。
//派生类的作者是不会企盼和喜欢这种规定的。


//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 对象进行了有意义的初始化。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，
//它们将必须指出怎么犯的错误。不过通常不明确应该怎么去做，
//很多代码的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的做法是否能提高软件的总体质量。
class EquipmentPiece {
    public:
        EquipmentPiece( int IDNumber = UNSPECIFIED);
        ...
    private:
        static const int UNSPECIFIED; //  其值代表 ID 值不确定。
    };
    //这允许这样建立 EquipmentPiece 对象
    EquipmentPiece e; //这样合法
```

规则 5:

/*对定制的类型转换函数保持警觉*/

```
/*
 * 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。
 */

 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。
 */

 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

/*
 * 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，但是在现在这种情况下，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。
    */
   Rational r(1, 2);
   cout << r; // 应该打印出"1/2"

//解决方法是用不使用语法关键字的等同的函数来替代转换运算符
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

        /*
         * 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。
         */
        template<class T>
            class Array {
                public:
                    Array(int lowBound, int highBound);
                    Array(int size);
                    T& operator[](int index);
                    ...
            };
        //调用以下代码
        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]"
            do something for when
            a[i] and b[i] are equal;
        }
        else {
```

```
            do something for when they are not;
}
        //会执行一下代码
        for (int i = 0; i < 10; ++i)
    if (a == static_cast< Array<int> >(b[i])) ...

    //解决办法 使用关键词 explicit
    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

        //另外的解决办法
        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
                ...
        };
    /*
     * 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。
```

```
*/

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。
```

规则 6:

```
/*M6：自增(increment)、自减(decrement)操作符前缀形式与后缀形式的区别*/
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);

        // 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
        }

    /*
    * 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。
    */
```

规则 7:

```
/*ItemM7：不要重载“&&”，“||”，或“,”*/
//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
/*
 * 以上调用可以导致的结果:
 * 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，没有采用短路计算法。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。完全可能与具有从左参数到
 * 右参数计算顺序的短路计算法相反。
 */

/*
 * 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。
 * 如果你重载逗号表达式将无法提供这样的效果
 */

/*
 * 你不能重载下面的操作符：
 *    .      .*      ::      ?:
 *   new   delete   sizeof    typeid
 *   static_cast dynamic_cast const_cast reinterpret_cast
 */

/*
 * 你能重载下面的操作符：
*   operator new operator delete
*   operator new[] operator delete[]
*   + - * / % ^ & | ~
*   ! = < > += -= *= /= %=
```

```
*    ^= &= |= << >> >>= <<= == !=
*    <= >= && || ++ -- , ->* ->
*    () []
*/
```

规则 8:

```
/*理解各种不同含义的 new 和 delete*/
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。

//函数 operator new  通常这样声明：
void * operator new(size_t size);
//返回值类型是 void*，因为这个函数返回未经处理（raw）的指针，未初始化的内存。

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。

string *ps = new string("Memory Management");
//以下代码的功能和上面的一样
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 =static_cast<string*>(memory); //  是 ps 指针指向新的对象

//placement new  的用法  #include<new>
class Widget {
      public:
            Widget(int widgetSize);
            ...
};
Widget * constructWidgetInBuffer(void *buffer,int widgetSize)
{
      return new (buffer) Widget(widgetSize);
}


//删除与内存释放
string *ps;
...
delete ps; //  使用 delete  操作符
```

```
void *buffer = operator new(50*sizeof(char));// 分配足够的 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

//placement new 释放内存的方法
/*
 * 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

        //数组资源的释放
        string *ps=new string[10];

        delete [] ps;
```

规则:9

```
/*利用 destructors 避免资源泄漏*/

//
class ALA{
    public:
        virtual void processAdoption()=0;

};
class Puppy:public:ALA{

    public:
        virtual void processAdoption();

};
class Kitten:public:ALA{

    public:
        virtual void processAdoption();

};
ALA *read(istream& s);//从 s 读取信息，返回一个指针，只想一个新分配的对象
void processAdoptions(istream& dataSource)
{

    while(dataSource){//如果还有数据
        ALA *pa=readALA(dataSource)//取出下一只动物
        pa->processAdoption();//处理收养事宜  此处很容易抛出异常
        delete pa;//删除 readALA 返回的对象
    }
}

//解决方法 2
void processAdoptions(istream& dataSource)
{

    while(dataSource){//如果还有数据
        auto_ptr<ALA> pa=readALA(dataSource)//取出下一只动物

        pa->processAdoption();//处理收养事宜
        //auto_ptr 会自动释放资源
    }
}
//解决方法 1(不是很好  使程序晦涩  建议使用解决方法 2)
```

```
void processAdoptions(istream& dataSource)
{

    while(dataSource){//如果还有数据
        ALA *pa=readALA(dataSource)//取出下一只动物
        try{
        pa->processAdoption();//处理收养事宜
    }
    catch(...){
        delete pa;
        throw;
    }
        delete pa;
    }
}
```

规则:10

```
/*在构造函数中防止资源泄漏*/
/*
 *在构造函数中防止资源泄漏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
 */

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
    };
    //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 != "") {
                    //如果 BookEntry 抛出异常 将不会调用析构函数
```

```
            theImage = new Image(imageFileName);//有可能出发异常

        }
    if (audioClipFileName != "") {

        theAudioClip = new AudioClip(audioClipFileName);//有可能出发异常
        }
    }
    BookEntry::~BookEntry()
    {
            delete theImage;
            delete theAudioClip;
    }

    //解决方法 1

    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
        void cleanup();//共同清理释放动作
```

```
};
    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 { // 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
                                cleanup();//释放资源
                                throw; // 继续传递异常
                            }
          }


        BookEntry::~BookEntry()
        {
                cleanup();//释放资源
        }


    //另外一种情况
    //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 const 类型
                    AudioClip * const theAudioClip; //只能在初始化列表初始化
```

```
};
// 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)
  {}

//解决方法
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
      };
//以下代码就不会 出现资源泄漏
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)
{}
```

规则:11

```
/*禁止异常信息（exceptions）传递到析构函数外*/
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
 */

    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
 *是不是在堆栈退栈（stack unwinding）中被释放，terminate 函数都不会被调用。
 */
```

规则:12

/*了解抛出一个 exception 与传递一个参数或调用一个虚函数之间的差异*/

```
/*
 *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。
 */

 // 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注意到 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。
 */



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。
*/

//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 M19），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（WQ 加注，
//重要：是两个!）。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：拷贝同样是一个临时对象。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。

```
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

*句只能捕获类型真真正正为 double 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。
1.//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。
2.//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。

/*
*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一定需要被拷贝。
*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。
*/

规则:13

```
   /*以 by reference  方式捕获异常*/
//以 by pointer 方式捕获异常不是好注意  你应该避开他，以 by value  或 by reference 捕获

/*
*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
*M12）。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，而且当本准备调用它们的虚拟函数时，系统解析后调用的*
却是基类对象的函数。
    */

    /*
     *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。
     */

     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:public exception { … }; //也来自标准 C++异常类
  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  Validation_error::what()
        ...
        }
}
```

规则:14

/*明智而审慎的使用 exception specification*/

```
/*
 *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，但因为 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编译器拒绝这种调用方式.
 */
        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。
                …
        }

//使用 exception specification 的明智选择
1.避免将 exception specification 放在需要类型自变量的 template 身上
            // 一个不良的 template 设计 因为他带有 exception specification
              template<class T>
              bool operator==(const T& lhs, const T& rhs) throw()
              {
                        return &lhs == &rhs;
              }

/*
 *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，这个更一般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。
 */
2.如果 A 函数内调用了 B 函数，而 B 函数无 exception specification，那么 A 函数本身也不
  要设定 exception specification。
          // 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：
//  一个没有异常规格的回调函数
        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
```

3.是处理系统本身抛出的 exception。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 M8）。如果你在函数里使用 new 操作符（还参见条款 M8），你必须为函数可能遇到 bad_alloc 异常作好准备。

```
//解决办法、
//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。
```

```
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，但正如我们所见，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，但这是 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）。
*/

规则:15

/*了解异常处理的成本*/

1.你需要一些空间，来放置某些数据结构，必须付出一些时间，随时保持数据的正确性

2.来自的 try 语句快的成本

```
/*
 *谨慎的做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，只要可能就尽量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。
 */
```

规则:16

/*谨记 80-20 法则*/

//影响效率的代码是你程序中的那 20%的代码，80% 的代码对效率没什么影响

规则:17

/*考虑使用缓式评估*/

//1.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
    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 M29 中被提供，但是其蕴含的原
//则就是 lazy evaluation：除非你确实需要，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。

//2.区别对待读取和写入
    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。条款 30 有相信介绍
```

```
//3.Lazy Fetching（懒惰提取）
        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。

            //解决方法
            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。

//4.Lazy Expression Evaluation(懒惰表达式计算)

  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

    //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？

    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。

//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！

//为 lazy evaluation 卖命是值得的。可以换取大量的时间和空间

规则:18

```
/*分期摊还预期的计算成本*/
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。


//over-eager evaluation 的方法

  //1.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，用来缓存查找到的数据。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。
    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;
        }

//2.Prefetching(预提取)是另一种方法


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;
}

//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

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
```

规则:19

/*了解临时对象的来源*/

//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。

//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!
```

//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 M22 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 M20 的内容。
```

规则:20

/*协助完成返回值优化*/

```
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 M6,
        const Rational operator*(const Rational& lhs,const Rational& rhs);

        // 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？

        //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

        // 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;//WQ 加注 返回时，其指向的对象已经不存在了
        }

        // 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());
        }
```

```
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

// 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());
}
```

规则:21

/*以重载技术避免隐式类型转换*/

//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。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

//以下的代码不是很好 ，存在隐式转换

```
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;
```

//现在考虑下面这些语句：

```
upi3 = upi1 + 10;//ok
upi3 = 10 + upi2;//ok
```

//改进的算法

```
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。
```

规则:22

```
/*考虑用操作符的复合形式（op=）取代其单独形式（op）*/

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;
    }

    //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
    }


    //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 M19 和条款 M20，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
```

```
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 M20）。
//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。

规则:23

/*考虑使用其他程序库*/

//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。

//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。不仅很难拿出一组数据能够代表程序或程序库的
//典型用法，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
//的客户就是典型的代表。不过 benchmark 还是能在同一个问题的不同解决
//方法间的比较上提供一些信息，所以尽管完全依靠 benchmark 是愚蠢的，
//但是忽略它们也是愚蠢的。

//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 M8 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。

规则:24

/*理解虚拟函数、多继承、虚基类和 RTTI 所需的代价*/

//虚函数

```
/*
*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。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。虚函数所需的第二个代价：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。虚函数所需的第三个代价：你实际上放弃了使用内联函数。
*/
```

//多继承

```
 /*
  *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（每一个基
  *类对应一个）；除了我们已经讨论过的单独的自己的 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。
  */
```

//虚基类

```
/*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。
*/
```

//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类型信息。
   */
```

/*

性质	对象大小	class 数据量增加	inlining 几率降低
虚函数	是	是	是
多重继承	是	是	否
虚基类	往往如此	有时候	否
RTTI	否	是	否

*/

规则 25:

```
/*将构造函数和非成员函数虚拟化*/
//virtual constructor
class NLComponent { //用于 newsletter components
    public: // 的抽象基类
        ... //包含至少一个纯虚函数
    private:

    };
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:
        // 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);
        list<NLComponent*> components;
    };

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));
        }
    }
    //virtual copy constructor
    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
        ...
};
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 M35.
    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());
        }
}
/*
* 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，它使得有可能声明象虚拟构造函数这样的函数.
 */

//将 non-member functions 的行为虚化
//下面的代码会带来麻烦
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
//解决办法
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);
}
```

规则 26:

```
/*限制某个类所能产生的对象数量*/
//禁止产生对象
class CantBeInstantiated {
    private:
        CantBeInstantiated();
        CantBeInstantiated(const CantBeInstantiated&);
        ...
};

//只能产生一个对象
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。
 */

 class PrintJob {
        public:
            PrintJob(const string& whatToPrint);
            ...
    };
    string buffer;
```

```
            ... //填充 buffer
        thePrinter().reset();
        thePrinter().submitJob(buffer);

        //另一种产生一个对象的方法
        class Printer {
            public:
                static Printer& thePrinter();
                ...
            private:
                Printer();
                Printer(const Printer& rhs);
                ...
        };
        Printer& Printer::thePrinter()
        {
            static Printer p;
            return p;
        }
        //用户使用 printer 时有些繁琐：
        Printer::thePrinter().reset();
        Printer::thePrinter().submitJob(buffer);

        //加入 namespace 中
        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
        {//不能使用 inline  后果见教材
            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

//产生一个对象的另一种写法
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;
    }

    //有问题的代码
```

```
        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;
        //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

    //你想让一个类产生多个对象 但是不希望被继承

    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); }

        //为了防止内存泄漏 可以如下调用
        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。


        //产生特定多个对象 最多 10 个
```

```
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); }
//maxObject 也可以这样定义
class Printer {
    private:
        enum { maxObjects = 10 }; // 在类中,
        ... // maxObjects 为常量 10
};


//一个用来计算对象个数的 Base Class
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> // 定义 numObjects
    int Counted<BeingCounted>::numObjects; // 自动把它初始化为 0

    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;
        }

//Printer 的写法
    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);
};

//对于 maxObject 我们选择让用户定义它
//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;
```

规则 27:

/*：要求或禁止在堆中产生对象*/

```
/*
 * 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。
 */
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）
  * 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*/
    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;
};

// 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
}

//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*/

//禁止用户直接实例化对象很简单，因为总是调用 new 来建立这种对象，
//你能够禁止用户调用 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 M8），
//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，如果你想不想让用户在堆中建立 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);
            ...
};
//现在用户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 M8)
//也声明为 private。


//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 operator new 和 operator delete 是
//自动继承的，如果 operator new 和 operator delete 没有在派生类中被声明
//为 public（进行改写，overwrite），它们就会继承基类中 private 的版本，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 =new NonNegativeUPNumber; // 错误！试图调用

// private operator new

//UPNumber 的 operator new 是 private 这一点，不会对包含 UPNumber 成员对象的
//对象的分配产生任何影响：
    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
```

规则 28:

```
/*智能指针*/

//大部分智能指针模版的样子
class SmartPtr {
    public:
        SmartPtr(T* realPtr = 0); // 建立一个灵巧指针
            // 指向 dumb pointer 所指的
            // 对象。未初始化的指针
            // 缺省值为 0(null)
        SmartPtr(const SmartPtr& rhs); // 拷贝一个灵巧指针
        ~SmartPtr(); // 释放灵巧指针
        // make an assignment to a smart ptr
        SmartPtr& operator=(const SmartPtr& rhs);
        T* operator->() const; // dereference 一个灵巧指针
        // 以访问所指对象的成员
        T& operator*() const; // dereference 灵巧指针
    private:
        T *pointee; // 灵巧指针所指的对象
};


//智能指针的构造、赋值和析构

template<class T>
    class auto_ptr {
        public:
            ...
            auto_ptr(auto_ptr<T>& rhs); // 拷贝构造函数
            auto_ptr<T>& // 赋值
            operator=(auto_ptr<T>& rhs); // 操作符
            ...
    };
template<class T>
auto_ptr<T>::auto_ptr(auto_ptr<T>& rhs)
{
    pointee = rhs.pointee; // 把*pointee 的所有权
                           // 传递到 *this
    rhs.pointee = 0; // rhs 不再拥有
} // 任何东西
template<class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
    if (this == &rhs) // 如果这个对象自我赋值
```

```
        return *this; // 什么也不要做
    delete pointee; // 删除现在拥有的对象
    pointee = rhs.pointee; // 把*pointee 的所有权
    rhs.pointee = 0; // 从 rhs 传递到 *this
    return *this;
}

//因为当调用 auto_ptr 的拷贝构造函数时，对象的所有权被传递出去，
//所以通过传值方式传递 auto_ptr 对象是一个很糟糕的方法。因为：
// 这个函数通常会导致灾难发生
    void printTreeNode(ostream& s, auto_ptr<TreeNode> p)
        { s << *p; }
    int main()
    {
        auto_ptr<TreeNode> ptn(new TreeNode);
        ...
        printTreeNode(cout, ptn); //通过传值方式传递 auto_ptr
        ...
    }

    //这不是说你不能把 auto_ptr 做为参数传递，这只意味着不能使用传值的方法。
    //通过 const 引用传递（Pass-by-reference-to-const）的方法是这样的：
    // 这个函数的行为更直观一些
    void printTreeNode(ostream& s,const auto_ptr<TreeNode>& p)
        { s << *p; }
   //智能指针的析构函数通常是这样的：
    template<class T>
    SmartPtr<T>::~SmartPtr()
    {
        if (*this owns *pointee) {
        delete pointee;
        }
    }

    /*实现解引用操作符*/
    template<class T>
    T& SmartPtr<T>::operator*() const
    {
        perform "smart pointer" processing;
        return *pointee;
    }

    template<class T>
    T* SmartPtr<T>::operator->() const
```

```
{
        perform "smart pointer" processing;
        return pointee;
}

  /*测试智能指针是否为 NULL*/
//目前为止我们讨论的函数能让我们建立、释放、拷贝、赋值、dereference 灵巧指针。
//但是有一件我们做不到的事情是“发现灵巧指针为 NULL”：
  SmartPtr<TreeNode> ptn;
  ...
  if (ptn == 0) ... // error!
  if (ptn) ... // error!
  if (!ptn) ... // error!
  //这是一个严重的限制。

  //如果这样写下面的代码来解决问题  会带来一定的问题
  template<class T>
  class SmartPtr {
        public:
              ...
              operator void*(); //  如果灵巧指针为 null，
              ... //  返回 0，   否则返回
  }; //  非 0。
  SmartPtr<TreeNode> ptn;
  ...
  if (ptn == 0) ... //  现在正确
  if (ptn) ... //  也正确
  if (!ptn) ... //  正确
 //这与 iostream 类中提供的类型转换相同，所以可以这样编写代码：
  ifstream inputFile("datafile.dat");
  if (inputFile) ... //  测试 inputFile 是否已经被
  //  成功地打开。
  //象所有的类型转换函数一样，它有一个缺点：在一些情况下虽然大
  //多数程序员希望它调用失败，但是函数确实能够成功地被调用（参见条款 M5）。
  //特别是它允许灵巧指针与完全不同的类型之间进行比较：
        SmartPtr<Apple> pa;
        SmartPtr<Orange> po;
        ...
        if (pa == po) ... //  这能够被成功编译!

  //一种差强人意的解决办法
  template<class T>
  class SmartPtr {
        public:
```

```
        ...
        bool operator!() const; // 当且仅当灵巧指针是
        ... // 空值，返回 true。
};
//用户程序如下所示：
SmartPtr<TreeNode> ptn;
    ...
if (!ptn) { // 正确
... // ptn 是空值
}
else {
... // ptn 不是空值
}
//但是这样就不正确了：
if (ptn == 0) ... // 仍然错误
if (ptn) ... // 也是错误的
//仅在这种情况下会存在不同类型之间进行比较：
SmartPtr<Apple> pa;
SmartPtr<Orange> po;
...
if (!pa == !po) ... // 能够编译

/*把智能指针转变成 dumb 指针*/
class Tuple { ... }; // 同上
void normalize(Tuple *pt); // 把*pt 放入
// 范式中；注意使用的
// 是 dumb 指针
//考虑一下，如果你试图用指向 Tuple 的灵巧指针作参数调用 normalize，
//会出现什么情况：
DBPtr<Tuple> pt;
...
normalize(pt); // 错误!
//除非有一个让人非常信服的原因去这样做 否则不要提供对 dumb pointer 的隐式
类型转换操作符。

/*smart pointer 和 "与继承有关的"类型转换*/

class MusicProduct {
    public:
        MusicProduct(const string& title);
        virtual void play() const = 0;
        virtual void displayTitle() const = 0;
        ...
};
```

```
class Cassette: public MusicProduct {
    public:
        Cassette(const string& title);
        virtual void play() const;
        virtual void displayTitle() const;
        ...
};
class CD: public MusicProduct {
    public:
        CD(const string& title);
        virtual void play() const;
        virtual void displayTitle() const;
          ...
};
//再接着假设，我们有一个函数，给它一个 MusicProduct 对象，
//它能显示产品名，并播放它：
void displayAndPlay(const MusicProduct* pmp, int numTimes)
{
    for (int i = 1; i <= numTimes; ++i)
        {
            pmp->displayTitle();
            pmp->play();
         }
}
//这个函数能够这样使用：
Cassette *funMusic = new Cassette("Alapalooza");
CD *nightmareMusic = new CD("Disco Hits of the 70s");
displayAndPlay(funMusic, 10);
displayAndPlay(nightmareMusic, 0);
//这并没有什么值得惊讶的东西，但是当我们用灵巧指针替代 dumb 指针，
//会发生什么呢：
 void displayAndPlay(const SmartPtr<MusicProduct>& pmp, int numTimes);
 SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
 SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));
 displayAndPlay(funMusic, 10); //  错误!
 displayAndPlay(nightmareMusic, 0); //  错误!

 //既然灵巧指针这么聪明，为什么不能编译这些代码呢？
 //  不能进行编译的原因是不能把 SmartPtr<CD>或 SmartPtr<Cassette>
 //转换成 SmartPtr<MusicProduct>。从编译器的观点来看，这些类之间
 //没有任何关系。为什么编译器的会这样认为呢？毕竟 SmartPtr<CD>  或
 // SmartPtr<Cassette>不是从 SmartPtr<MusicProduct>继承过来的，
 //这些类之间没有继承关系，我们不可能要求编译器把一种对象转换成
 //（完全不同的）另一种类型的对象。
```

```
//解决办法是
template<class T> // 模板类，指向 T 的
class SmartPtr { // 灵巧指针
      public:
            SmartPtr(T* realPtr = 0);
            T* operator->() const; T& operator*() const;
            template<class newType> // 模板成员函数
            operator SmartPtr<newType>() // 为了实现隐式类型转换.
            {
                  return SmartPtr<newType>(pointee);
                  }
            ...
};
void displayAndPlay(const SmartPtr<MusicProduct>& pmp, int numTimes);
SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));
displayAndPlay(funMusic, 10); // 正确!
displayAndPlay(nightmareMusic, 0); // 正确!

//下面代码调用的过程
displayAndPlay(funMusic, 10);

/*
 *funMusic 对象的类型是 SmartPtr<Cassette>。函数 displayAndPlay 期望的参数是
 * SmartPtr<MusicProduct>地对象。编译器侦测到类型不匹配，于是寻找把 funMusic
 * 转换成 SmartPtr<MusicProduct>对象的方法。它在 SmartPtr<MusicProduct>类里寻
 * 找带有 SmartPtr<Cassette>类型参数的单参数构造函数（参见条款 M5），但是没有找
 *到.然后它们又寻找成员函数模板，以实例化产生这样的函数。它们在
 *SmartPtr<Cassette> 发现了模板，把 newType 绑定到 MusicProduct 上，生成了所需的
 *函数.
 */

 //这种技术能给我们几乎所有想要的行为特性。假设我们用一个新类 CasSingle 来扩充
 //MusicProduct 类层次，用来表示 cassette singles。
 template<class T> // 同上，包括作为类型
 class SmartPtr { ... }; // 转换操作符的成员模板
 void displayAndPlay(const SmartPtr<MusicProduct>& pmp, int howMany);
 void displayAndPlay(const SmartPtr<Cassette>& pc, int howMany);
 SmartPtr<CasSingle> dumbMusic(new CasSingle("Achy Breaky Heart"));
 displayAndPlay(dumbMusic, 1); // 错误!
 //在这个例子里，displayAndPlay 被重载，一个函数带有 SmartPtr<Cassette> 对象参数，
 //其它函数的参数为 SmartPtr<CasSingle>，我们期望调用 SmartPtr<Cassette>，
 //因为 CasSingle 是直接从 Cassette 上继承下来的，而间接继承自 MusicProduct。
```

```
//当然这是 dumb 指针时的工作方法。我们的灵巧指针不会这么灵巧，它们的转换操
//作符是成员函数，对 C++编译器而言，所有类型转换操作符是同等地位的。
//因此 displayAndPlay 的调用具有二义性，因为从 SmartPtr<CasSingle> 到
//SmartPtr<Cassette>的类型转换并不比到 SmartPtr<MusicProduct>的类型转换优先。

/*smart pointer 与 const*/

CD goodCD("Flood");
    const CD *p;// p 是一个 non-const 指针
                                //指向 const CD 对象
    CD * const p = &goodCD; // p 是一个 const 指针
                                        // 指向 non-const CD 对象
                                        // 因为 p 是 const，它
                                        // 必须被初始化
    const CD * const p = &goodCD; // p 是一个 const
                                                    // 指向一个 const CD 对
    //我们自然想要让灵巧指针具有同样的灵活性。
    //不幸的是只能在一个地方放置 const，
    //并只能对指针本身起作用，而不能针对于所指对象：
     const SmartPtr<CD> p = &goodCD;// p 是一个 const 灵巧指针
                                            // 指向 non-const CD 对象

    //考虑下面四种情况
    SmartPtr<CD> p; // non-const 对象 // non-const 指针
    SmartPtr<const CD> p; // const 对象, // non-const 指针
    const SmartPtr<CD> p = &goodCD; // non-const 对象 // const 指针
    const SmartPtr<const CD> p = &goodCD; // const 对象 // const 指针

    CD *pCD = new CD("Famous Movie Themes");
    const CD * pConstCD = pCD; // 正确
    //但是如果我们试图把这种方法用在灵巧指针上，情况会怎么样呢？
    SmartPtr<CD> pCD = new CD("Famous Movie Themes");
    SmartPtr<const CD> pConstCD = pCD; // 正确么?不 SmartPtr<CD> 与 SmartPtr<const
                                            //CD>是完全不同的类型

    //上述问题的解决方法
    template<class T> // 指向 const 对象的
    class SmartPtrToConst { // 灵巧指针
        ... // 灵巧指针通常的
                // 成员函数
        protected:
            union { const T* constPointee; // 让 SmartPtrToConst 访问
                T* pointee; // 让 SmartPtr 访问
                };
```

```
};
 template<class T> // 指向 non-const 对象
 class SmartPtr: // 的灵巧指针
      public SmartPtrToConst<T> {
            ... // 没有数据成员
};

//利用这种新设计，我们能够获得所要的行为特性：
SmartPtr<CD> pCD = new CD("Famous Movie Themes");
SmartPtrToConst<CD> pConstCD = pCD; // 正确
```

规则 29:

/*Reference counting 引用计数*/

```
//实现引用计数
template<class T> // template class for smart
class RCPtr { // pointers-to-T objects; T
        public: // must inherit from RCObject
                RCPtr(T* realPtr = 0);
                RCPtr(const RCPtr& rhs);
                ~RCPtr();
                RCPtr& operator=(const RCPtr& rhs);
                T* operator->() const;
                T& operator*() const;
        private:
                T *pointee;
        void init();
};
class RCObject { // base class for reference-
        public: // counted objects
                void addReference();
                void removeReference();
                void markUnshareable();
                bool isShareable() const;
                bool isShared() const;
                protected:
                RCObject();
                RCObject(const RCObject& rhs);
                RCObject& operator=(const RCObject& rhs);
                virtual ~RCObject() = 0;
private:
                int refCount;
                bool shareable;
};
class String { // class to be used by
        public: // application developers
                String(const char *value = "");
                const char& operator[](int index) const;
                char& operator[](int index);
        private:
                // class representing string values
                struct StringValue: public RCObject {
                char *data;
                StringValue(const char *initValue);
```

```
            StringValue(const StringValue& rhs);
            void init(const char *initValue);
            ~StringValue();
        };
            RCPtr<StringValue> value;
    };

//实现
//这是 RCObject 的实现：
    RCObject::RCObject(): refCount(0), shareable(true) {}
    RCObject::RCObject(const RCObject&): refCount(0), shareable(true) {}
    RCObject& RCObject::operator=(const RCObject&){ return *this; }
    RCObject::~RCObject() {}
    void RCObject::addReference() { ++refCount; }
    void RCObject::removeReference()
        { if (--refCount == 0) delete this; }
    void RCObject::markUnshareable()
            { shareable = false; }
    bool RCObject::isShareable() const
            { return shareable; }
    bool RCObject::isShared() const
            { return refCount > 1; }
    //这是 RCPtr 的实现：
    template<class T>
    void RCPtr<T>::init()
        {
            if (pointee == 0) return;
            if (pointee->isShareable() == false) {
            pointee = new T(*pointee);
            }
            pointee->addReference();
        }
    template<class T>
    RCPtr<T>::RCPtr(T* realPtr): pointee(realPtr)
        { init(); }
    template<class T>
    RCPtr<T>::RCPtr(const RCPtr& rhs): pointee(rhs.pointee)
        { init(); }
    template<class T>
    RCPtr<T>::~RCPtr()
        { if (pointee)pointee->removeReference(); }
    template<class T>
    RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
        {
```

```
            if (pointee != rhs.pointee) {
            if (pointee) pointee->removeReference();
            pointee = rhs.pointee;
            init();
            }
            return *this;
        }
    template<class T>
    T* RCPtr<T>::operator->() const { return pointee; }
    template<class T>
    T& RCPtr<T>::operator*() const { return *pointee; }
//这是 String::StringValue 的实现：
    void String::StringValue::init(const char *initValue)
    {
            data = new char[strlen(initValue) + 1];
            strcpy(data, initValue);
    }
    String::StringValue::StringValue(const char *initValue)
            { init(initValue); }
    String::StringValue::StringValue(const StringValue& rhs)
            { init(rhs.data); }
    String::StringValue::~StringValue()
            { delete [] data; }
//最后，归结到 String，它的实现是：
    String::String(const char *initValue): value(new StringValue(initValue)) {}
    const char& String::operator[](int index) const
            { return value->data[index]; }
    char& String::operator[](int index)
        {
            if (value->isShared()) {
            value = new StringValue(value->data);
            }
            value->markUnshareable();
            return value->data[index];
        }


        //将 reference counting 加到既有的 classes 身上
        template<class T>
        class RCIPtr {
            public:
                RCIPtr(T* realPtr = 0);
                RCIPtr(const RCIPtr& rhs);
                ~RCIPtr();
```

```
        RCIPtr& operator=(const RCIPtr& rhs);
        const T* operator->() const; // see below for an
        T* operator->(); // explanation of why
        const T& operator*() const; // these functions are
        T& operator*(); // declared this way
    private:
        struct CountHolder: public RCObject {
            ~CountHolder() { delete pointee; }
            T *pointee;
        };
        CountHolder *counter;
        void init();
        void makeCopy(); // see below
};
template<class T>
void RCIPtr<T>::init()
{
    if (counter->isShareable() == false) {
    T *oldValue = counter->pointee;
    counter = new CountHolder;
    counter->pointee = new T(*oldValue);
    }
    counter->addReference();
}
template<class T>
RCIPtr<T>::RCIPtr(T* realPtr): counter(new CountHolder)
{
    counter->pointee = realPtr;
    init();
}
template<class T>
RCIPtr<T>::RCIPtr(const RCIPtr& rhs): counter(rhs.counter)
    { init(); }
template<class T>
RCIPtr<T>::~RCIPtr()
    { counter->removeReference(); }
template<class T>
RCIPtr<T>& RCIPtr<T>::operator=(const RCIPtr& rhs)
    {
        if (counter != rhs.counter) {
        counter->removeReference();
        counter = rhs.counter;
        init();
        }
```

```
            return *this;
        }
template<class T> // implement the copy
void RCIPtr<T>::makeCopy() // part of copy-on-
{ // write (COW)
        if (counter->isShared()) {
            T *oldValue = counter->pointee;
            counter->removeReference();
            counter = new CountHolder;
            counter->pointee = new T(*oldValue);
            counter->addReference();
            }
}
template<class T> // const access;
const T* RCIPtr<T>::operator->() const // no COW needed
        { return counter->pointee; }
template<class T> // non-const
T* RCIPtr<T>::operator->() // access; COW
        { makeCopy(); return counter->pointee; } // needed
template<class T> // const access;
const T& RCIPtr<T>::operator*() const // no COW needed
        { return *(counter->pointee); }
template<class T> // non-const
T& RCIPtr<T>::operator*() // access; do the
        { makeCopy(); return *(counter->pointee); } // COW thing
class Widget {
        public:
            Widget(int size);
            Widget(const Widget& rhs);
            ~Widget();
            Widget& operator=(const Widget& rhs);
            void doThis();
            int showThat() const;
};
//那么 RCWidget 将被定义为这样：
class RCWidget {
        public:
            RCWidget(int size): value(new Widget(size)) {}
            void doThis() { value->doThis(); }
            int showThat() const { return value->showThat(); }
        private:
            RCIPtr<Widget> value;
};
```

规则 30:

```
/*proxy classes*/
//实现二维数组
class Array2D {
    public:
        class Array1D {
            public:
                T& operator[](int index);
                const T& operator[](int index) const;
                ...
    };
        Array1D operator[](int index);
        const Array1D operator[](int index) const;
        ...
};
//现在，它合法了：
    Array2D<float> data(10, 20);
    ...
    cout << data[3][6]; // fine

//区分 operator  []的读写动作
class String { // reference-counted strings;
    public: // see Item 29 for details
        class CharProxy { // proxies for string chars
            public:
                CharProxy(String& str, int index); // creation
                CharProxy& operator=(const CharProxy& rhs); // lvalue
                CharProxy& operator=(char c); // uses
                operator char() const; // rvalue
                // use
            private:
                String& theString; // string this proxy pertains to
                int charIndex; // char within that string
                // this proxy stands for
        };
            // continuation of String class
        const CharProxy
        operator[](int index) const; // for const Strings
        CharProxy operator[](int index); // for non-const Strings
        ...
        friend class CharProxy;
    private:
        RCPtr<StringValue> value;
};
```

```
String s1, s2; // reference-counted strings
// using proxies
...
cout << s1[5]; // still legal, still works
//表达式 s1[5]返回的是一 CharProxy 对象。没有为这样的对象定义输出流操作，
//所以编译器努力地寻找一个隐式的类型转换以使得 operator<<调用成功（见 Item M5）。
//它们找到一个：在 CahrProxy 类内部申明了一个隐式转换到 char 的操作。于是
//自动调用这个转换操作，结果就是 CharProxy 类扮演的字符被打印输出了。这个
//CharProxy 到 char 的转换是所有代理对象作右值使用时发生的典型行为。
s2[5] = 'x'; // also legal, also works
//和前面一样，表达式 s2[5]返回的是一个 CharProxy 对象，但这次它是赋值操作的目标。
//由于赋值的目标是 CharProxy 类，所以调用的是 CharProxy 类中的赋值操作。
//这至关重要，因为在 CharProxy 的赋值操作中，我们知道被赋值的 CharProxy 对象是作
//左值使用的。因此我们知道 proxy 类扮演的字符是作左值使用的，必须执行一些必要
//的操作以实现字符的左值操作。
s1[3] = s2[8]; // of course it's legal,of course it works
//调用作用于两个 CharProxy 对象间的赋值操作，在此操作内部，
//我们知道左边一个是作左值，右边一个作右值。

const String::CharProxy String::operator[](int index) const
    {
        return CharProxy(const_cast<String&>(*this), index);
    }
String::CharProxy String::operator[](int index)
    {
        return CharProxy(*this, index);
    }
String::CharProxy::CharProxy(String& str, int index)
    : theString(str), charIndex(index) {}
String::CharProxy::operator char() const
    {
        return theString.value->data[charIndex];
    }

String::CharProxy&
String::CharProxy::operator=(const CharProxy& rhs)
{
    // if the string is sharing a value with other String objects,
    // break off a separate copy of the value for this string only
    if (theString.value->isShared()) {
        theString.value = new StringValue(theString.value->data);
    }
    // now make the assignment: assign the value of the char
```

```
    // represented by rhs to the char represented by *this
    theString.value->data[charIndex] =
    rhs.theString.value->data[rhs.charIndex];
    return *this;
    }
    string::CharProxy& String::CharProxy::operator=(char c)
    {
        if (theString.value->isShared()) {
            theString.value = new StringValue(theString.value->data);
            }
        theString.value->data[charIndex] = c;
        return *this;
    }



    //使用代理类的限制
    //如果 String::operator[]返回一个 CharProxy 而不是 char &，下面的代码将不能编译：
    String s1 = "Hello";
    char *p = &s1[1]; // error!
    //表达式 s1[1]返回一个 CharProxy，于是“=”的右边是一个 CharProxy *。
    //没有从 CharProxy *到 char *的转换函数，所以 p 的初始化过程编译失败了。
    //通常，取 proxy 对象地址的操作与取实际对象地址的操作得到的指针，其类型是
    //不同的。

    //要消除这个不同，你需要重载 CharProxy 类的取地址运算：
    class String {
        public:
            class CharProxy {
                public:
                    ...
                    char * operator&();
                    const char * operator&() const;
                   ...
            };
        ...
  };

  const char * String::CharProxy::operator&() const
{
        return &(theString.value->data[charIndex]);
}
char * String::CharProxy::operator&()
{
```

```
    // make sure the character to which this function returns
    // a pointer isn't shared by any other String objects
    if (theString.value->isShared()) {
        theString.value = new StringValue(theString.value->data);
    }
    // we don't know how long the pointer this function
    // returns will be kept by clients, so the StringValue
    // object can never be shared
    theString.value->markUnshareable();
    return &(theString.value->data[charIndex]);
}


template<class T> // reference-counted array
class Array { // using proxies
    public:
        class Proxy {
            public:
                Proxy(Array<T>& array, int index);
                Proxy& operator=(const T& rhs);
                operator T() const;
                ...
        };
    const Proxy operator[](int index) const;
    Proxy operator[](int index);
...
};
//看一下这个数组可能被怎样使用：
    Array<int> intArray;
    ...
    intArray[5] = 22; // fine
    intArray[5] += 5; // error!
    ++intArray[5]; // error!
    //解决办法就是为代理类手动添加+= ++操作符

//我们将定义一个 Rational 类，然后使用前面看到的 Array 模板：
class Rational {
    public:
        Rational(int numerator = 0, int denominator = 1);
        int numerator() const;
        int denominator() const;
        ...
};
```

```
Array<Rational> array;
//这是我们所期望的使用方式，但我们很失望：
cout << array[4].numerator(); // error!
int denom = array[22].denominator(); // error!

//另一个 proxy 对象替代实际对象失败的情况是作为非 const 的引用传给函数：
void swap(char& a, char& b); // swaps the value of a and b
string s = "+C+"; // oops, should be "C++"
swap(s[0], s[1]); // this should fix the
// problem, but it won't
// compile


class TVStation {
      public:
            TVStation(int channel);
            ...
};
      void watchTV(const TVStation& station, float hoursToWatch);
      //借助于 int 到 TVStation 的隐式类型转换（见 Item M5），我们可以这么做：
      watchTV(10, 2.5); // watch channel 10 for
      // 2.5 hours
     //然而，当使用那个用 proxy 类区分 operator[]作左右值的带引用计数的数组模板时，
     //我们就不能这么做了：
      Array<int> intArray;
      intArray[4] = 10;
      watchTV(intArray[4], 2.5); // error! no conversion
      // from Proxy<int> to
      // TVStation
```

规则:32

/*在未来的时态下发展程序*/

```
/*
 *这么做的一种方法是：用 C++语言自己来表达设计上的约束条件，而不是用注释或文档。
 *例如，如果一个类被设计得不会被继承，不要只是在其头文件中加个注释，用 C++的方法
 *来阻止继承；Item M26 显示了这个技巧。如果一个类需要其实例全部创建在堆中,不要只
 *是对用户说了这么一句，用 Item M27 的方法来强迫这一点.如果拷贝构造和赋值对一个类
 *是没有意义的,通过申明它们为私有来阻止这些操作（见 Item E27）.C++提供了强大的功能、
 *灵活度和表达力。用语言提供的这些特性来强迫程序符合设计。
 */
 /*
  *因为万物都会变化，要写能承受软件发展过程中的混乱攻击的类。避免“demand-paged”
  *WQ：“用户要求型”之类的意思吧）的虚函数，凭什么你本没有写虚函数而直到有人来
  *要求后你就更改为虚函数?应该判断一个函数的含意,以及它被派生类重定义的话是否有
  *意义。如果是有意义的，申明它为虚，即使没有人立即重定义它。如果不是的话，申明
  *它为非虚，并且不要在以后为了便于某人而更改；确保更改是对整个类的运行环境和类
  *所表示的抽象是有意义的（见 Item E36）。
  */
 /*
  *处理每个类的赋值和拷贝构造函数，即使“从没人这样做过”。他们现在没有这么做并不
  *意味着他们以后不这么做（见 Item E18）。如果这些函数是难以实现的，那么申明它们为
  *私有。这样，不会有人误调编译器提供的默认版本而做错事（这在默认赋值和拷贝构造
  *函数上经常发生，见 Item E11）。
  *要承认：只要是能被人做的，就有人这么做努力于可移植的代码。写可移植的代码并不
  *比不可移植的代码难太多,只有在性能极其重要时采用不可移植的结构才是可取的
  *将你的代码设计得当需要变化时，影响是局部的。尽可能地封装；将实现细节申明为私
  *有（例子见 Item E20）。只要可能，使用无名的命名空间和文件内的静态对象或函数（见
  *Item E31）。避免导致虚基类的设计，因为这种类需要每个派生类都直接初始化它——即
  *使是那些间接派生类（见 Item M4 和 Item E43）。避免需要 RTTI 的设计，它需要 if...then...else
  *型的瀑布结构（再次参见 Item M31，然后看 Item E39 上的好方法）。每次，类的继承层
  *次变了，每组 if...then...else 语句都需要更新，如果你忘掉了一个，你不会从编译器得到
  *任何告警。
  */
    //下列的忠告是不正确的
    1.//你需要虚析构函数，只要有人 delete 一个实际值向 D 的 B *。
        //这里，B 是基类，D 是其派生类。换句话说，这位作者暗示，
        //如果你的程序看起来是这样时，并不需要 B 有虚析构函数：
        class B { ... }; // no virtual dtor needed
        class D: public B { ... };
        B *pb = new D;
        //然而，当你加入这么一句时，情况就变了：
        delete pb; // NOW you need the virtual
        // destructor in B
        //这意味着，用户代码中的一个小变化——增加了一个 delete 语句——实际上能导
```

//致需要修改 B 的定义。
//如果这发生了的话，所有 B 的用户都必须重编译。
//采纳了这个作者的建议的话，一条语句的增加将导致
//大量代码的重编译和重链接。这绝不是一个高效的设计。

2.//如果一个公有基类没有虚析构函数，所有的派生类基其成员函数都不应该有析构函
//数。也就是说，这是没问题的：

```
class string { // from the standard C++ library
public:
~string();
};
class B { ... }; // no data members with dtors,
// no virtual dtor needed
```

//但从 B 继承一个新类后，事情就变了：

```
class D: public B {
string name; // NOW ~B needs to be virtual
};
```

//再一次，一个关于 B 的使用的小小的变化（这里是增加了一个包含有析构函数
//的成员对象的派生类）可能需要大量代码的重编译和重链接。但在系统中，小的
//变化应该只有小的影响。这个设计在这个测试上失败了。

3.//如果多重继承体系有许多析构函数，每个基类都应该有应该虚析构函数。
4.//我们没有使用虚析构函数，因为我们不想这个 string 类有 vtbl。
//我们甚至不期望想有一个 string *,所以这不成为问题。
//我们很清楚这么做将造成的困难

//由于 string 没有 virtual destructor，当他派出新类，是一种高度冒险的行为。
//这个生产商应该提供明确的文档以指出他的 string 类没有被设计得可被继承的，
//但如果程序员没注意到这个警告或未能读到这个文档/时会发生什么？
//一个可选方法是用 C++自己来阻止继承。Item M26 描述了怎么限制对象只生
//成于堆中，以及用 auto_ptr 对象
//来操作堆中的对象。构造 string 对象的接口将不符合传统也不方便，需要这样：

```
auto_ptr<String> ps(String::makeString("Future tense C++"));
... // treat ps as a pointer to
// a String object, but don't
// worry about deleting it
```

//来代替这个：

```
String s("Future tense C++");
```

//但，多半，为了减少不正确的继承行为是值得换用不方便的语法的。
//（对 string 类，这未必很合算，但对其它类，这样的交换是完全值得的。）

//未来时态的考虑只是简单地增加了一些额外约束：

1.//提供完备的类（见 Item E18），即使某些部分现在还没有被使用。

//如果有了新的需求，你不用回过头去改它们。
2.//将你的接口设计得便于常见操作并防止常见错误（见 Item E46）。
//使得类容易正确使用而不易用错。例如,阻止拷贝构造和赋值操作，
//如果它们对这个类没有意义的话（见 Item E27）。防止部分赋值（见 Item M33）。
3.//如果没有限制你不能通用化你的代码，那么通用化它。
//例如，如果在写树的遍历算法，考虑将它通用得可以处理任何有向不循环图。

规则:33

/*将非尾端类设计为抽象类*/

```
        Animal
        /    \
  Lizard     Chicken
```

```
class Animal {
    public:
        Animal& operator=(const Animal& rhs);
        ...
    };
class Lizard: public Animal {
    public:
        Lizard& operator=(const Lizard& rhs);
        ...
    };
class Chicken: public Animal {
    public:
        Chicken& operator=(const Chicken& rhs);
        ...
    };
    //这里只写出了赋值运算函数，但已经够我们忙乎一阵了。看这样的代码：
    Lizard liz1;
    Lizard liz2;
    Animal *pAnimal1 = &liz1;
    Animal *pAnimal2 = &liz2;
    ...
    *pAnimal1 = *pAnimal2;
    //这里有两个问题。
    /*
    第一,最后一行的赋值运算调用的是 Animal 类的，虽然相关对象的类型是 Lizard。
        结果，只有 liz1 的 Animal 部分被修改。这是部分赋值。在赋值后，liz1 的
        Animal 成员有了来自于 liz2 的值，但其 Lizard 成员部分没被改变。
    第二,问题是真的有程序员把代码写成这样。用指针来给对象赋值并不少见，特别
        是那些对 C 有丰富经验而转移到 C++的程序员。所以，我们应该将赋值设计
        得更合理的。如 Item M32 指出的，我们的类应该容易被正确适用而不容易被
        用错，而上面这个类层次是容易被用错。
 */

 //针对上述问题的解决办法是将赋值运算申明为虚函数。如果 Animal::operator=是
 //虚函数，那句赋值语句将调用 Lizard 的赋值操作（应该被调用的版本）
 class Animal {
        public:
```

```
        virtual Animal& operator=(const Animal& rhs);
        ...
};
class Lizard: public Animal {
    public:
        virtual Lizard& operator=(const Animal& rhs);
        ...
};
class Chicken: public Animal {
    public:
        virtual Chicken& operator=(const Animal& rhs);
        ...
};
Lizard liz;
Chicken chick;
Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;
...
*pAnimal1 = *pAnimal2; // assign a chicken to
// a lizard!

//上面的设计的带来的问题
//这是一个混合类型赋值：左边是一个 Lizard，右边是一个 Chicken。混合类型赋值在
//C++中通常不是问题，因为 C++的强类型原则将评定它们非法。然而,通过将 Animal 的
//赋值操作设为虚函数，我们打开了混合类型操作的门。

//这使得我们处境艰难。我们应该允许通过指针进行同类型赋值,而禁止通过同样的指针
//进行混合类型赋值。换句话说，我们想允许这样：
        Animal *pAnimal1 = &liz1;
        Animal *pAnimal2 = &liz2;
        ...
        *pAnimal1 = *pAnimal2; // assign a lizard to a lizard
        //而想禁止这样：
        Animal *pAnimal1 = &liz;
        Animal *pAnimal2 = &chick;
        ...
        *pAnimal1 = *pAnimal2; // assign a chicken to a lizard

        //对于上述修改后带来的问题的解决办法
        //我们可以使用 dynamic_cast（见 Item M2）来实现。
        //下面是怎么实现 Lizard 的赋值操作：
        Lizard& Lizard::operator=(const Animal& rhs)
        {
        // make sure rhs is really a lizard
```

```
    const Lizard& rhs_liz = dynamic_cast<const Lizard&>(rhs);
    proceed with a normal assignment of rhs_liz to *this;
    }
```

//这个函数只在 rhs 确实是 Lizard 类型时将它赋给*this。如果 rhs 不是 Lizard 类型，
//函数传递出 dynamic_cast 转换失败时抛的 bad_cast 类型的异常。

```
    Lizard liz1, liz2;
    ...
    liz1 = liz2; // no need to perform a
    // dynamic_cast: this
    // assignment must be valid
```

//我们可以处理上述情况而无需增加复杂度或花费 dynamic_cast，
//只要在 Lizard 中增加一个通常形式的赋值操作：

```
class Lizard: public Animal {
public:
virtual Lizard& operator=(const Animal& rhs);
Lizard& operator=(const Lizard& rhs); // add this
...
};
Lizard liz1, liz2;
...
liz1 = liz2; // calls operator= taking
// a const Lizard&
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;
...
*pAnimal1 = *pAnimal2; // calls operator= taking
// a const Animal&
```

//实际上，给出了后面那个的 operator=，也就简化了前者的实现：

```
Lizard& Lizard::operator=(const Animal& rhs)
{
return operator=（dynamic_cast<const Lizard>(rhs));
}
```

//现在这个函数试图将 rhs 转换为一个 Lizard。如果转换成功，通常的赋值操作被调用；
//否则，一个 bad_cast 异常被抛出。

//解决上述问题我们还可以找到另一个解法
//最容易的方法是在 Animal 中将 operator=置为 private。于是，Lizard 对象可以赋值给
//Lizard 对象，Chicken 对象可以赋值给 Chicken 对象，但部分或混合类型赋值被禁止：

```
    class Animal {
    private:
    Animal& operator=(const Animal& rhs); // this is now
```

```
... // private
};
class Lizard: public Animal {
public:
Lizard& operator=(const Lizard& rhs);
...
};
class Chicken: public Animal {
      public:
Chicken& operator=(const Chicken& rhs);
...
};
Lizard liz1, liz2;
...
liz1 = liz2; // fine
Chicken chick1, chick2;
...
chick1 = chick2; // also fine
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &chick1;
...
*pAnimal1 = *pAnimal2; // error! attempt to call
// private Animal::operator=
//不幸的是，Animal 也是实体类，这个方法同时将
//Animal 对象间的赋值评定为非法了：
Animal animal1, animal2;
...
animal1 = animal2; // error! attempt to call
// private Animal::operator=
//而且，它也使得不可能正确实现 Lizard 和 Chicken 类的赋值操作，
//因为派生类的赋值操
//作函数有责任调用其基类的赋值操作函数：
Lizard& Lizard::operator=(const Lizard& rhs)
{
if (this == &rhs) return *this;
Animal::operator=(rhs); // error! attempt to call
                        // private function. But
                        // Lizard::operator= must
                        // call this function to
... // assign the Animal parts
} // of *this!
```

//上述代码依然存在问题。但“允许 Animal 对象间的赋值而阻止 Lizard 和 Chicken
//对象通过 Animal 的指针进行部分赋值”的两难问题仍然存在。程序该怎么办？

//最终的解决办法


```
class AbstractAnimal {
        protected:
            AbstractAnimal& operator=(const AbstractAnimal& rhs);
        public:
            virtual ~AbstractAnimal() = 0; // see below
            ...
 };
 class Animal: public AbstractAnimal {
        public:
            Animal& operator=(const Animal& rhs);
            ...
  };
 class Lizard: public AbstractAnimal {
       public:
           Lizard& operator=(const Lizard& rhs);
           ...
  };
 class Chicken: public AbstractAnimal {
        public:
            Chicken& operator=(const Chicken& rhs);
            ...
  };
```

//这个设计给你所以你需要的东西。同类型间的赋值被允许，
//部分赋值或不同类型间的赋值被禁止；
//派生类的赋值操作函数可以调用基类的赋值操作函数。
//此外，所有涉及 Aniaml、Lizard 或 Chicken
//类的代码都不需要修改，因为这些类仍然操作，
//其行为与引入 AbstractAnimal 前保持了一致。
//肯定，这些代码需要重新编译，但这是为获得
//“确保了编译通过的赋值语句的行为是正确的而行为
//可能不正确的赋值语句不能编译通过”所付出的很小的代价。

//如果你有两个实体类 C1 和 C2 并且你喜欢 C2 公有继承自 C1，
//你应该将两个类的继承层次改为三个类的继

//承层次，通过创造一个新的抽象类 A 并将 C1 和 C2 都从它继承：

规则 34:

```
/*如何在同一个程序中结合 c++和 c*/

/*
 * 确认兼容后，还有四个要考虑的问题：
 * 1.名变换
 * 2.静态初始化
 * 3.内存动态分配
 * 4.数据结构兼容
 */

//1.名称重整
 /*
  *编译时，c++由于存在重载。所以函数会被编译器给予另外一个名字，
  *而 c 不存在这些情况，你给定的函数是什么名字就是什么名字，编译器
  *不会改变函数的名字。所以要使 c 和 c++有兼容性，必须抑制 c++对函数
  *名字改变。压抑名字重整的做法：
  */
  // declare a function called drawLine; don't mangle
    // its name
    extern "C"
    void drawLine(int x1, int y1, int x2, int y2);

    //压抑一组不需要重整的名字
    extern "C" { // disable name mangling for
    // all the following functions
    void drawLine(int x1, int y1, int x2, int y2);
    void twiddleBits(unsigned char bits);
    void simulate(int iterations);
    ...
    }
    //在 c++中压抑名字重整，在 c 中不需要
    //当用 C++编译时，你应该加 extern 'C'，但用 C 编译时，不应该这样。
    //通过只在 C++编译器下定义的宏__cplusplus，你可以将头文件组织得这样：
    #ifdef __cplusplus
    extern "C" {
    #endif
    void drawLine(int x1, int y1, int x2, int y2);
    void twiddleBits(unsigned char bits);
    void simulate(int iterations);
    ...
    #ifdef __cplusplus
    }
    #endif
```

//2.statics 的初始化

```
/*
 *在掌握了名变换后，你需要面对一个 C++中事实：在 main 执行前和执行
 *后都有大量代码被执行。尤其是，静态的类对象和定义在全局的、命名
 *空间中的或文件体中的类对象的构造函数通常在 main 被执行前就被调用。
 *这个过程称为静态初始化（参见 Item E47）。这和我们对 C++和 C 程序的
 *通常认识相反，我们一直把 main 当作程序的入口。同样，通过静态初始
 *化产生的对象也要在静态析构过程中调用其析构函数；这个过程通常发
 *生在 main 结束运行之后。
 */

//为了解决 main()应该首先被调用，而对象又需要在 main()执行前被构造
//的两难问题，许多编译器在 main()的最开始处插入了一个特别的函数，
//由它来负责静态初始化。同样地，编译器在 main()结束处插入了一个函
//数来析构静态对象。产生的代码通常看起来象这样：
    int main(int argc, char *argv[])
    {
    performStaticInitialization(); // generated by the
                                   // implementation
    the statements you put in main go here;
    performStaticDestruction(); // generated by the
                                   // implementation
    }

    //有时看起来用 C 写 main()更有意义——比如程序的大部分是 C 的，C++部分
    //只是一个支持库。然而，这个 C++库很可能含有静态对象（即使现在没有，
    //以后可能会有——参见 Item M32），所以用 C++写 main()仍然是个好主意。
    //这并不意味着你需要重写你的 C 代码。只要将 C 写的 main()改名为 realMain()，
    //然后用 C++版本的 main()调用 realMain()：
    extern "C" // implement this
    int realMain(int argc, char *argv[]); // function in C
    int main(int argc, char *argv[]) // write this in C++
    {
        return realMain(argc, argv);
    }
```

//3.动态内存分配

```
    /*
     * C++部分使用 new 和 delete（参见 Item M8），C 部分使用 malloc（或其变形）
     *和 free。只要 new 分配的内存使用 delete 释放，malloc 分配的内存用 free
     *释放，那么就没问题。用 free 释放 new 分配的内存或用 delete 释放 malloc
```

```
        *分配的内存，其行为没有定义。那么，唯一要记住的就是：将你的 new 和
        *delete 与 mallco 和 free 进行严格的隔离。
        */
  //4.数据结构兼容性
     /*
    *因为 C++中的 struct 的规则兼容了 C 中的规则，假设“在两类编译器下定义的同一
    *结构将按同样的方式进行处理”是安全的。这样的结构可以在 C++和 C 见安全地来
    *回传递。如果你在 C++版本中增加了非虚函数，其内存结构没有改变，所以，只有
    *非虚函数的结构（或类）的对象兼容于它们在 C 中的孪生版本（其定义只是去掉了
    *这些成员函数的申明）。增加虚函数将结束游戏，因为其对象将使用一个不同的内存
    *结构（参见 Item M24）。从其它结构（或类）进行继承的结构，通常也改变其内存
    *结构，所以有基类的结构也不能与 C 函数交互。
    */

//总结
    // 如果想在同一程序下混合 C++与 C 编程，记住下面的指导原则：
        1.确保 C++和 C 编译器产生兼容的 obj 文件。
        2.将在两种语言下都使用的函数申明为 extern 'C'。
        3.只要可能，用 C++写 main()。
        4.总用 delete 释放 new 分配的内存；总用 free 释放 malloc 分配的内存。
        5.将在两种语言间传递的东西限制在用 C 编译的数据结构的范围内；这些结构
          的 C++版本可以包含非虚成员函数。
```

规则:35

/*让自己习惯使用标准 C++语言*/

//ISO/ANSI C++标准是厂商实现编译器时将要考虑的，是作者们准备出书时将要分析的，
//是程序员们在对 C++发生疑问时用来寻找权威答案的。在《ARM》之后发生的最主要
//变化是以下内容：

1.增加了新的特性：RTTI、命名空间、bool，关键字 mutable 和 explicit，
对枚举的重载操作，已及在类的定义中初始化 const static 成员变量。

2.模板被扩展了：现在允许了成员模板，增加了强迫模板实例化的语法，
模板函数允许无类型参数，模板类可以将它们自己作为模板参数。

3.异常处理被细化了：异常规格申明在编译期被进行更严格的检查，
unexpected()函数现在可以抛一个 bad_exception 对象了。

4.内存分配函数被改良了：增加了 operator new[]和 operator delete[]函数，
operator new/new[]在内存分配失败时将抛出一个异常，并有一个返回为 0（不抛
异常）的版本供选择。（见 Effective C++ Item 7）

5.增加了新的类型转换形式：static_cast、dynamic_cast、const_cast，和
reinterpret_cast。

6.语言规则进行了重定义：重定义一个虚函数时，其返回值不需要完全的匹配了
（如果原来返回基类对象或指针或引用，派生类可以返回派生类的对象、指针或
引用），临时对象的生存期进行了精确地定义。

//STL

1.Container

2.Iterator

3.Algorithm