More Effective C++读书笔记

**【条款1】仔细区别pointers和references**

pointers和references的区别：

相同点：它们都使得你可以**间接参考其他对象**

不同点：

**1、有null pointer，没有null reference。**

null pointer是值为NULL的指针，即空指针，它不指向任何对象，因此不能对null pointer进行解引用，在对指针进行操作前需要判断该指针是否是空指针。

没有null reference，因此reference在定义时就必须引用到一个已存在的对象（即reference在定义时就必须进行初始化）。

**2、可以改变pointer的值，不能让reference去指向另一个对象**

改变pointer的值即是让pointer去指向另一个对象。

reference总是指向（代表）它最初获得的那个对象，一旦代表了某对象，就不能再改变。

什么时候使用pointers或references？

**当你需要指向（代表）某个已存在的对象，而且绝不会改变指向其他对象，或是你实现一个操作而其语法需求无法由pointers达成（比如operator[]），你就应该选用references。其他任何时候，请采用pointer。**

**【条款2】最好使用C++转型操作符**

C++引入了4个新的转型操作符：

(1)**static\_cast**

static\_cast是C++中适用范围最广的转型操作符，其行为类似于C旧式转型，以及相同的限制。

(2)**const\_cast**

只能用于移除对象的常量性。

(3)**dynamic\_cast**

在继承体系中执行安全的向下转型，即**把指向基类对象的指针或引用转型为指向派生类对象的指针或引用。如果转型对象是指针，当转型失败时会返回NULL；如果转型对象是引用，当转型失败时会抛出异常。**

注意，**dynamic\_cast只能用于有虚函数的类型身上**。

(4)**reinterpret\_cast**

如何选择各种新转型操作符？

如果你想为一个不涉及继承机制的类型执行转型动作，请用static\_cast

如果你仅仅是想移除一个对象的常量性，请用const\_cast

**【条款4】非必要不提供default constructor**

constructor用来将对象初始化，default constructor就是不需要任何参数就可以调用的构造函数。

如果一个class缺乏一个default constructor，使用这个类便会有某些限制：

**1、不能产生由缺乏default constructor的类对象构成的数组。**

例：TestClass arr[10];或TestClass\* pArr = new TestClass[10];

要产生由TestClass对象构造的数组，需要调用TestClass的default constructor（TestClass::TestClass()）产生数组中的每个对象。

**2、classes如果缺乏default constructor，它们将不适用于许多template-based container classe**s**。**

**3、一个缺乏default constructor的vitual base class，要求其所有的derived classes—不论距离多么遥远，都必须提供virtual base class的constructor的参数。**

有4种情况，C++编译器会为未声明constructor的类生成default constructor：

**1、带有default constructor的member class object。**

**2、带有default constructor的base class。**

**3、带有virtual function的class**

**4、带有一个virtual base class的class**

在以上4种情况下，如果某类没有声明default constructor，编译器会暗自生成一个default constructor来完成必要的操作；如果某类已经声明了default constructor，则编译器会在该类所有的constructor中插入所需的代码。（细节请参看《深入探索C++对象模型》）

**【条款5】对定制的类型转换函数保持警觉**

一个implicit类型转换操作符的声明：

class Rational  
{

public:

…

operator double() const; //将Rational对象转换为double

}

**以功能对等的另一个函数取代类型转换操作符**，如：

class Rational

{

public:

…

double asDouble() const; //将Rational对象转换为double

}

**只要将constructor声明为explicit，编译器便不能因隐式类型转换的需要而调用它们，不过显式类型转换是允许的。**

**【条款6】区分increment/decrement操作符的前置（prefix)和后置形式（postfix）**

后置式的increment/decrement有一个int参数，并且在它被调用时，编译器默默的为此int参数指定一个**0**值。

处理用户定制类型时，应该尽可能使用前置式increment/decrement。

后置式操作符应该以前置式操作符为实现基础。

**【条款7】使用析构函数防止资源泄漏**

**用一个对象存储需要被自动释放的资源，然后依靠对象的析构函数来释放资源**。

实现方式请参考auto\_ptr<T>:

（示例：一个简单的auto\_ptr实现）

template <typename T>

class auto\_ptr

{

public:

auto\_ptr(T\* p = NULL): ptr(p) { }

~auto\_ptr()

{

delete ptr; //释放资源

}

private:

T\* ptr; //原始资源指针，指向需要被自动释放的资源对象

}

用auto\_ptr封装资源对象，资源对象的生命周期和auto\_ptr对象的生命周期相一致，当auto\_ptr对象被销毁时，会调用auto\_ptr的析构函数，从而释放其持有的资源。