编码规范

1 缩进

是用Tab键好，还是2、4、8个空格？

结论：4个空格，在VS2005和其他的一些编辑工具中都可以定义Tab键扩展成为几个空格键。不用 Tab键的理由是Tab键在不同的情况下会显示不同的长度。4个空格的距离从可读性来说正好。

2 行宽

行宽必须限制，但是以前有些文档规定的80字符行宽太小了（以前的计算机/打字机显示行宽为80字符），现在时代不同了，可为100字符。

3 括号

在复杂的条件表达式中，用括号清楚地表示逻辑优先级。

4 断行与空白的{ }行

程序的结构是什么风格？下面有几种格式，我们一一讨论。

最精简的格式A：

if (condition) DoSomething();

else DoSomethingElse();

有人喜欢这样，因为可以节省几行，但是不同的语句（Statement）放在一行中，会使程序调试（DeBug）非常不方便，如果要一步一步观察condition（condition有可能是包含函数调用的复杂表达式）中各个变量的变化情况，单步执行就很难了。

因此，我们还是要有断行，这样可以得到如下的结构——格式B：

if (condition)

DoSomething();

else

DoSomethingElse();

这样的结构，由于没有明确的“{”和“}”来判断程序的结构，在有多层控制嵌套的时候，就不容易看清结构和对应关系。下面的改进（格式C）虽好，但还是不够清晰：

if ( condition) {

DoSomething();

} else {

DoSomethingElse();

}

于是我们最后做了这个选择，每个“{”和“}”都独占一行。就是格式D：

if ( condition)

{

DoSomething();

}

else

{

DoSomethingElse();

}

5 分行

不要把多行语句放在一行上。

a = 1; b = 2; // bogus

if (fFoo) Bar(); // bogus

更严格地说，不要把不同的变量定义在一行上。

Foo foo1, foo2; // bogus

6 命名

程序中的实体、变量是程序员昼思夜想的对象，要起一个好的名字才行。大家都知道用单个字母给有复杂语义的实体命名是不好的，目前最通用的，也是经过了实践检验的方法叫“匈牙利命名法”。例如：

fFileExist，表明是一个bool值，表示文件是否存在；

szPath，表明是一个以0结束的字符串，表示一个路径。

如此命名的目的是让程序员一眼就能看出变量的类型，避免在使用中出错。早期的计算机语言（如BCPL）不作类型检查，在C语言中，int、byte、char、bool大概都是一回事。下面这一句话：

if (i)

从语义来说，i可以是表示真/假的一个值，也可以表示长度是否为零，

还可以表示是否到了字符串的结束位置，或者可以表示两个字符串比较的结果不相等（strcmp（）返回-1，0，1）。从程序的文字上，很难看出确切的语义。

同样是字符串类型，char \*，BSTR的有些行为是很不一样的。

HRESULT的值也可以用来表示真假，但是HR\_TRUE == 0，HR\_FALSE ==1，这和通常的true/false刚好相反。

大部分的程序，错就错在这些地方！在变量面前加上有意义的前缀，就可以让程序员一眼看出变量的类型及相应的语义。这就是“匈牙利命名法”的用处。

匈牙利命名法的一些通用规定，见本书附录B（第337页）。

还有一些地方不适合用“匈牙利命名法”，比如，在一些强类型的语言（如C#）中，不同类型的值是不能做运算的，对类型有严格的要求，例如C# 中，if()语句只能接受bool值的表达式，这样就大大地防止了以上问题的发生。在这样的语言中，前缀就不是很必要的，匈牙利命名法则不适用了。Microsoft .Net Framework就不主张用这样的法则。

7 下划线问题

下划线用来分隔变量名字中的作用域标注和变量的语义，如：一个类型的成员变量通常用m\_来表示。移山公司规定下划线一般不用在其他方面。

8 大小写问题

由多个单词组成的变量名，如果全部都是小写，很不易读，一个简单的解决方案就是用大小写区分它们。

Pascal——所有单词的第一个字母都大写；

Camel——第一个单词全部小写，随后单词随Pascal格式，这种方式也叫lowerCamel。

一个通用的做法是：所有的类型/类/函数名都用Pascal形式，所有的变量都用Camel形式。

类/类型/变量：名词或组合名词，如Member、ProductInfo等。

函数则用动词或动宾组合词来表示，如get/set; RenderPage()。

9 注释

复杂的注释应该放在函数头，很多函数头的注释都是解释参数的类型等的，如果程序正文已经能够说明参数的类型in/out等，就不要重复！

注释也要随着程序的修改而不断更新，一个误导的（Misleading）注释往往比没有注释更糟糕。

另外，注释（包括所有源代码）应只用ASCII字符，不要用中文或其他特殊字符，它们会极大地影响程序的可移植性。

在现代编程环境中，程序编辑器可以设置各种好看的字体，我们可以使用不同的显示风格来表示程序的不同部分。

10 函数

现代程序设计语言中的绝大部分功能，都在程序的函数（Function, Method）中实现，关于函数最重要的原则是：只做一件事，但是要做好。

11 goto

函数最好有单一的出口，为了达到这一目的，可以使用goto。只要有助于程序逻辑的清晰体现，什么方法都可以使用，包括goto。

12 错误处理

1．参数处理

在DeBug版本中，所有的参数都要验证其正确性。在正式版本中，从外部（用户或别的模块）传递过来的参数要验证其正确性。

2．断言

如何验证正确性？那就要用Assert（断言）。断言和错误处理是什么关系？

当你觉得某事肯定如何，你可以用断言。

Assert (p != NULL);

然后可以直接使用变量p;

如果你认为某事可能会发生，这时就要用错误处理。

如：

……

p = AllocateNewSpace(); // could fail

if (p == NULL)

{ // error handling.

}

else

{ // use p to do something

}

13 如何处理C++中的类

注意，除了关于异常（Exception）的部分，大部分其他原则对C#也适用。

1．类

（1）使用类来封装面向对象的概念和多态（Polymorphism）。

（2）避免传递类型实体的值，应该用指针传递。换句话说，对于简单的数据类型，没有必要用类来实现。

（3）对于有显式的构造和析构函数，不要建立全局的实体，因为你不知道它们在何时创建和消除。

（4）只有在必要的时候，才使用“类”。

2．Class vs. Struct

如果只是数据的封装，用Struct即可。

3．公共/保护/私有成员Public、Private和Protected

按照这样的次序来说明类中的成员：public、protected、private

4．数据成员

（1）数据类型的成员用m\_name说明。

（2）不要使用公共的数据成员，要用inline访问函数，这样可同时兼顾封装和效率。

5．虚函数Virtual Functions

（1）使用虚函数来实现多态（Polymorphism）。

（2）只有在非常必要的时候，才使用虚函数。

（3）如果一个类型要实现多态，在基类（Base Class）中的析构函数应该是虚函数。

6．构造函数Constructors

（1）不要在构造函数中做复杂的操作，简单初始化所有数据成员即可。

（2）构造函数不应该返回错误（事实上也无法返回）。把可能出错的操作放到HrInit()或FInit()中。

7．析构函数

（1）把所有的清理工作都放在析构函数中。如果有些资源在析构函数之前就释放了，记住要重置这些成员为0或NULL。

（2）析构函数也不应该出错。

8．New和Delete

（1）如果可能，实现自己的New/Delete，这样可以方便地加上自己的跟踪和管理机制。自己的New/Delete可以包装系统提供的New/Delete。

（2）检查New的返回值。New不一定都成功。

（3）释放指针时不用检查NULL。

9．运算符（Operators）

（1）在理想状态下，我们定义的类不需要自定义操作符。只有当操作符的确需要时。

（2）运算符不要做标准语义之外的任何动作。例如，“==”的判断不能改变被比较实体的状态。

（3）运算符的实现必须非常有效率，如果有复杂的操作，应定义一个单独的函数。

（4）当你拿不定主意的时候，用成员函数，不要用运算符。

10．异常（Exceptions）

（1）异常是在“异乎寻常”的情况下出现的，它的设置和处理都要花费“异乎寻常”的开销，所以不要用异常作为逻辑控制来处理程序的主要流程。

（2）了解异常及处理异常的花销，在C++语言中，这是不可忽视的开销。

（3）当使用异常时，要注意在什么地方清理数据。

（4）异常不能跨过DLL或进程的边界来传递信息，所以异常不是万能的。

11．类型继承（Class Inheritance）

（1）当有必要的时候，才使用类型继承。

（2）用Const标注只读的参数（参数指向的数据是只读的，而不是参数本身）。

（3）用Const标注不改变数据的函数。