**C++编码规则**

**头文件**

**函数参数顺序**

　　C/C++函数参数分为输入参数和输出参数两种，有时输入参数也会输出（注：值被修改时）。输入参数一般传值或常数引用（const references），输出参数戒输入/输出参数为非常数指针（non-const pointers）。对参数排序时，将所有输入参数置于输出参数之前。不要仅仅因为是新添加的参数，就将其置于最后，而应该依然置于输出参数之前。这一点并不是必须遵循的规则，输入/输出两用参数（通常是类/结构体变量）混在其中，会使得规则难以遵守。

　　个人感受：这条规则相当重要，自己写代码的时候可能没有太大感觉，但是在阅读别人代码的时候感觉特别明显。如果代码按照这种规范来写，从某种角度来说，这段代码具有“自注释”的功能，那么在看代码的时候就会比较轻松。Doom3的代码规范中提到，“Use ‘const’ as much as possible”，也是同样的意义。当然，const除了阅读方便以外，还有个很重要的就是防止编码错误，一旦在程序中修改const变量，编译器就会报错，这样就减少了人工出错了可能性，这点尤为重要！

**包含文件的名称及次序**

　　将包含次序标准化可增强可读性、避免隐藏依赖（hidden dependencies，注：隐藏依赖主要是指包含的文件编译），次序如下：C 库、C++库、其他库的.h、项目内的.h。

　　项目内头文件应按照项目源代码目录树结构排列，并且避免使用UNIX文件路径.（当前目录）和..（父目录）。

　　举例来说，google-awesome-project/src/foo/internal/fooserver.cc 的包含次序如下：

1. #include "foo/public/fooserver.h" *// 优先位置*
2. #include <sys/types.h>
3. #include <unistd.h>
4. #include <hash\_map>
5. #include <vector>
6. #include "base/basictypes.h"
7. #include "base/commandlineflags.h"
8. #include "foo/public/bar.h"

　　注意，对应的头文件一定要先包含，这样避免隐藏依赖，隐藏依赖的问题不懂的可以去Google，网上有很多资料。另外，《C++编程思想》中提到的包含次序正好相反，从特殊到一般，但是有一点和Google代码规范是一样的，那就是对应的头文件是第一个包含。对于隐藏依赖的问题，以前只是习惯性的把对应的头文件放第一个，没有想过为什么，现在学习了……

**作用域**

**全局变量**

　　class 类型的全局变量是被禁止的，内建类型的全局变量是允许的，当然多线程代码中非常数全局变量也是被禁止的。永远不要使用函数返回值初始化全局变量。

　　不幸的是，全局变量的构造函数、析构函数以及初始化操作的调用顺序只是被部分规定，每次生成有可能会有变化，从而导致难以发现bug。因此，禁止使用class类型的全局变量（包括STL的string,vector等），因为它们的初始化顺序可能会导致出现问题。内建类型和由内建类型构成的没有构造函数的结构体可以使用，如果你一定要使用class类型的全局变量，请使用单件模式。

**C++类**

**构造函数的职责**

　　构造函数中只进行那些没有实际意义的初始化，可能的话，使用Init()方法集中初始化为有意义(non-trivial)的数据。

　　个人感受：这种做法可以从一开始就避免一些bug的出现，或更容易解决一些bug。构造函数+Init()函数初始化的方式与只用构造函数的方法相比，对计算机来说他们是没有区别的，但是人是会犯错的，这一条代码规范在某种程度上避免了一些人为错误，这个在开发中特别重要。

**拷贝构造函数**

　　仅在代码中需要拷贝一个类的对象的时候使用拷贝构造函数，不需要拷贝时使用DISALLOW\_COPY\_AND\_ASSIGN这个宏（关于这个宏的内容，可以在网上搜到，我这里就不写了）。C++中对象的隐式拷贝是导致很多性能问题和bugs的根源。拷贝构造函数降低了代码可读性，相比按引用传递，跟踪按值传递的对象更加困难，对象修改的地方变得难以捉摸。

　　个人感受：和上一项的目的类似，为了避免人为错误！拷贝构造函数本来是为了方便程序员编程了，但是却有可能成为一个坑，为了避免这类问题，不需要拷贝时使用DISALLOW\_COPY\_AND\_ASSIGN，这样在需要调用拷贝构造函数的时候就会报错，减少了人为出错的可能性。C#和Java在这方面就做得比较好，虽然性能上不如C++，但是人为出错的概率减少了很多。当然，使用一定的代码规范，可以在一定程度上减少C++的坑。

**继承**

　　虽然C++的继承很好用，但是在实际开发中，尽量多用组合少用继承，不懂的去看GoF的《Design Patterns》。

　　但重定义派生的虚函数时，在派生类中明确声明其为virtual。这一条是为了为了阅读方便，虽然从语法的角度来说，在基类中声明了virtual，子类可以不用再声明该函数为virtual，但这样一来阅读代码的人需要检索类的所有祖先以确定该函数是否为虚函数o(╯□╰)o。

**多重继承**

　　虽然允许，但是只能一个基类有实现，其他基类是接口，这样一来和JAVA一样了。这些东西在C#和JAVA中都进行了改进，直接从语法上解决问题。C++的灵活性过高，也是个麻烦的问题，只能通过代码规范填坑。

**接口**

　　虚基类必须以Interface为后缀，方便阅读。阅读方便。

**重载操作符**

　　除少数特定情况外，不要重载操作符！！！“==”和“=”的操作Euqals和CopyFrom函数代替，这样更直观，也不容易出错。

　　个人感受：看到这一条，我有点惊讶，在学习C++的时候，说重载操作符有神马神马好处，为什么现在又说不要重载操作符呢？仔细看了他的文档，确实说的有道理，导致可能出现的bug见其具体文档。在实际应用中，由于C++的坑实在太多了，不得不把这种“好用”的东西干掉，因为出了bug又找不到，是一件很O疼的事情。

**声明次序**

　　1）typedefs和enums；  
　　2）常量；  
　　3）构造函数；  
　　4）析构函数；  
　　5）成员函数，含静态成员函数；  
　　6）数据成员，含静态数据成员。  
　　宏 DISALLOW\_COPY\_AND\_ASSIGN 置于private:块之后，作为类的最后部分。

**其他 C++ 特性**

**引用参数**

　　函数形参表中，所有的引用必须的const！

　　个人感受：这么做是为了防止引用引起的误解，因为引用在语法上是值，却有指针的意义。虽然引用比较好用，但是牺牲其某些方面的特性，换来软件管理方面的便利，还是很值得了。

**缺省参数**

　　禁止使用函数缺省参数！

　　个人感受：看到这一点的时候觉得有点因噎废食了，其实缺省参数感觉还是蛮好用的。当然从另外一个角度来说，要使用C++就不要怕这种小麻烦，如果因为使用这些特性造成了找不到的bug，那会损失更多时间。

**异常**

　　不要使用 C++异常。

　　这一点我没有看懂，也许是因为它的异常机制没有C#和Java那么完善吧……毕竟在C#和Java里面异常还是很好用的东东。

**流**

除了记录日志，不要使用流，使用printf之类的代替。

　　这一条其实是有一些争议的，当然大多数人认为代码一致性比较重要，所以选择printf，具体的可以看原文文档。

**const 的使用**

　　在任何可以的情况下都要使用const。

　　这条规则赞一个，Doom3的代码规范里也提到了这一条。这么做有两个好处，一个是防止程序出错，因为修改了const类型的变量会报错；另一个就是方便阅读，使代码“自注释”。虽然这么做也有坏处，当然，总体来说利大于弊。

**命名约定**

　　1、总体规则：不要随意缩写，如果说 ChangeLocalValue 写作ChgLocVal还有情可原的话，把ModifyPlayerName写作MdfPlyNm就太过分了，除函数名可适当为动词外，其他命名尽量使用清晰易懂的名词；

　　2、宏、枚举等使用全部大写+下划线；

　　3、变量（含类、结构体成员变量）、文件、命名空间、存取函数等使用全部小写+下划线，类成员变量以下划线结尾，全局变量以g\_开头；

　　4、普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线；

　　使用这套命名约定，可以使代码具有一定程度的“自注释”功能，方便他人阅读，也方便自己以后修改。当然3、4两点也可以使用其他的命名约定，只要团队统一即可。

**格式**

　　1、行宽原则上不超过80列，把22寸的显示屏都占完，怎么也说不过去；

　　2、尽量不使用非ASCII字符，如果使用的话，参考 UTF-8 格式（尤其是 UNIX/Linux 下，Windows 下可以考虑宽字符），尽量不将字符串常量耦合到代码中，比如独立出资源文件，返不仅仅是风格问题了；

　　3、UNIX/Linux下无条件使用空格，MSVC的话使用 Tab 也无可厚非； （我没用过Linux，不懂为什么在Linux下无条件使用空格）

　　4、函数参数、逻辑条件、初始化列表：要么所有参数和函数名放在同一行，要么所有参数并排分行；

　　5、除函数定义的左大括号可以置于行首外，包括函数/类/结极体/枚举声明、各种语句的左大括号置于行尾，所有右大括号独立成行；

　　6、./->操作符前后丌留空格，\*/&不要前后都留，一个就可，靠左靠右依各人喜好；

　　7、预处理指令/命名空间不使用额外缩进，类/结构体/枚举/函数/语句使用缩进；

　　8、初始化用=还是()依个人喜好，统一就好；

　　9、return不要加()；

　　10、水平/垂直留白不要滥用，怎么易读怎么来。

**写在最后**

　　总的来说，这套代码规范还是相当不错的，既有防止错误使用C++的某些特性而导致bugs的规范，又有代码书写的相关规范使其便于阅读，建议搞C++的童鞋都看一看。当然，具体的团队应该会有具体的代码规范，代码风格方面大家可能会有一些区别；不使用C++某些特性（比如不使用C++异常，禁止使用函数缺省参数）方面，应该按照具体情况进行折中处理，而不应该生搬硬套代码规范；但是“不将字符串常量耦合到代码中”这种规范，是大家必须遵守的。

**三种编程命名规范（匈牙利命名法、驼峰式命名法、帕斯卡命名法）**

三种流行的命名法则

目前业界共有四种命名法则：**驼峰命名法**、**匈牙利命名法**、**帕斯卡命名法**和 **下划线命名法**，其中前三种是较为流行的命名法。

**1 . 匈牙利命名：**

**开头字母用变量类型的缩写，其余部分用变量的英文或英文的缩写，且每个单词的第一个字母都大写。**

示例：

1. int iMyAge;  *// i 是 int 类型的缩写。*
2. char cMyName[10]; *// c 是 char 类型的缩写。*
3. float fManHeight; *// f 是 float 类型的缩写。*

匈牙利命名广泛应用于象 Microsoft Windows 这样的环境中， Windows 编程中用到的变量（还包括宏）的命名规则都是匈牙利命名法，这种命名技术是由一位能干的 Microsoft 程序员查尔斯- 西蒙尼(Charles Simonyi) 提出的

匈牙利命名法 通过在变量名前面加上相应的小写字母的符号标识作为前缀，标识出变量的作用域，类型等这些符号可以多个同时使用，顺序是先 m\_（成员变量）， 再指针，再简单数据类型，再其它 。

例如：m\_lpszStr, 表示指向一个以0字符结尾的字符串的长指针成员变量

匈牙利命名法关键是：标识符的名字以一个或者多个小写字母开头作为前缀；前缀之后的是首字母大写的一个单词或多个单词组合，该单词要指明变量的用途

匈牙利命名法中常用的小写字母的前缀：

属性 + 类型 + 描述。属性一般是 **小写字母** **+** \_ :

g\_ : 全局变量  
m\_ : 类成员变量  
s\_ : 静态变量  
c\_ : 常量

|  |  |
| --- | --- |
| 类 型 前 缀 | 类　　型 |
| a | 数组 (Array) |
| b | 布尔值 (Boolean) |
| by | 字节 (Byte) |
| c | 有符号字符 (Char)，用 c 开头 cCount |
| cb | 无符号字符 (Char Byte，没有多少人用) |
| cr | 颜色参考值 (ColorRef) |
| d | double 用d开头 dDeta |
| f | float 用f开头 fAvg |
| cx,cy | 坐标差（长度 ShortInt） |
| w | Word，unsigned int(WORD) 用w开头 wCount |
| dw | Double Word，unsigned long int(DWORD) 用dw开头 dwBroad |
| fn | 函数 |
| h | Handle（句柄） |
| i | 整型 int ，用 i 开头 iCount |
| n | short int  短整型  用 n 开头 nStepCount |
| l | Long Int   长整型  用 l 开头 lSum |
| lp | Long Pointer |
| m\_ | 类的成员 |
| np | Near Pointer |
| p | Pointer |
| s | 字符串类型，用s开头 sFileName |
| sz | 以null做结尾的字符串型 (String with Zero End)，    用\0结尾的字符串 用sz开头 szFileName |

**2 . 驼峰式命名法（小驼峰式 命名法）：**

小驼峰法（camel方法）：第一个单词 以小写字母开始；第二个单词的首字母大写，或从第二个单词开始，后面的每一个单词的首字母都采用大写字母，**即 小驼峰式命名法： 第一个单词首字母小写，后面其他单词首字母大写。**

**变量 一般用 小驼峰法标识。**

ex:   
int myAge;   
char myName[10];   
float manHeight;

小驼峰式 命名规则（**第一个单词首字母小写，后面所有单词的首字母都大写**）：firstName, camelCase

大驼峰式 命名规则（**所有单词的首字母都大写**）：FirstName, CamelCase

下面是分别用 **骆驼式命名法** 和 **下划线法命名** 的同一个函数：

printEmployeePaychecks()；    骆驼式命名法 ----  函数名中的每一个逻辑断点都有一个大写字母来标记  
print\_employee\_paychecks()；下划线法         ----  函数名中的每一个逻辑断点都有一个下划线来标记。

**3 . 帕斯卡命名法（大驼峰式 命名法）：**

帕斯卡命名法（pascal方法）**又叫 大驼峰式命名法。**相比小驼峰法，大驼峰法把第一个单词的首字母也大写了，**每个单词的第一个字母都大写。即 骆驼命名法是首字母小写，而帕斯卡命名法是首字母大写**

**（pascal方法）常用于 类名，函数名，属性，命名空间。**

大驼峰法（Upper Camel Case） ex:   
int MyAge;   
char MyName[10];   
float ManHeight ;  
public class DataBaseUser

**4. 下划线 命名规则**

下划线法是随着 C语言 的出现流行起来的，在 UNIX/LIUNX 这样的环境，以及 GNU 代码中使用非常普遍



**5. 命名规则 小结：**

MyData 就是一个帕斯卡命名的示例 。  
而 myData 是一个骆驼命名法，它第一个单词的第一个字母小写，后面的单词首字母大写，看起来像一个骆驼 。  
而 iMyData 是一个匈牙利命名法，它的小写的i说明了它的型态，后面的和帕斯卡命名相同,指示了该变量的用途。

通常每种语言都有自己的 Coding Style，比如 C/C++ 和 python 是下划线，java 和 go 是驼峰。

所以，对于要使用哪种命名法可以根据个人的代码编写风格，也是可使用不同的命名规范混合使用。

如：骆驼 + 下划线 （int temperature\_Sensor;）

