C++编码规范

# 一、类型命名规约（类、接口、结构体、枚举、联合、命名空间）

* 类名以t\_为前缀，如class t\_person;
* 接口名以i\_为前缀，如class i\_changable;
* 结构体名以t\_为前缀，如struct t\_itemData
* 类模板的前缀与类前缀一样，模板参数以T开头，如果不引起混淆则只用T
* 枚举类型名以e\_开头，变量用e为类型前缀，枚举常量用ec开头，常量使用枚举类型时建议同一类型的成员应使用同一个接头词。

如enum e\_btnState { ecNormal，ecDown，ecHighlight，ecDisable }；

e\_btnState m\_eBtnState;

* 联合类型以u\_为前缀
* 命名空间以n\_为前缀
* 函数指针类型 以fn\_ 为前缀

# 二、变量命名规约

## 1、准则：“表意清晰，见词达意”

通过变量名应该能够清晰的得到如下信息：

* 变量的属性（全局变量、静态变量、普通成员变量、静态成员变量、局部变量、函数参数）

通过属性前缀展示（全小写）。

* 变量的类型（int、char、bool、结构体、枚举类型、指针、类对象、函数指针等）

通过类型前缀展示（全小写）。

* 变量的用意

通过变量名称展示（驼峰命名法）。

通常情况下，合格的变量名应该是如下形式：**属性前缀\_类型前缀+变量名称**。

举例：

HANDLE g\_hInstance = NULL； // 进程实例，句柄，全局变量

static DWORD ms\_dwSkinNum； // 皮肤数量，DWORD，静态成员变量

bool m\_bVisible； // 是否可见，bool，成员变量

CButton\* pBtnClose; // 关闭按钮，按钮指针，局部变量

注意：为了团队统一性，同一工程内宜采用统一的编码规范。变量名禁止以下划线开头和结尾；变量名内部除了属性前缀与类型前缀之间，其他地方应该使用驼峰命名法，尽量不用下划线分割单词

## 2、属性前缀

变量名的属性前缀用于指示变量的作用域（全局变量、静态变量、普通成员变量、静态成员变量、局部变量、函数参数）。

**注：变量的属性前缀都是全小写。**

|  |  |
| --- | --- |
| 前缀 | 意义 |
| g\_ | 全局变量 |
| S\_ | 静态变量 |
| m\_ | 普通成员变量(类成员、结构体成员) |
| ms\_ | 静态成员变量 |
| p\_ | 函数参数 |
| 无 | 局部变量 |

## 3、类型前缀

|  |  |  |
| --- | --- | --- |
| 基本类型 | 意义 | 举例 |
| b | 布尔型(bool, BOOL) | BOOL bSex; |
| ch/wch | 字符变量（wchar\_t, char, TCHAR） | TCHAR chName; 尽量使用TCHAR  如果必须使用wchar\_t或char字符，建议在名字中有所提示 |
| a | 数组 | char achName[32]; 结尾不一定有0 |
| sz/wsz | NULL结尾的字符串数组(char, TCHAR) | TCHAR \* szAddress;  TCHAR szName[32]; 字符串结尾必须有0 |
| str | CString、t\_str、std::string等字符串对象 | Cstring strName. |
| I, n | int | int iCount; |
| ui, un | unsigned int, UINT | UINT uiCameraID |
| l | long（Win下不建议使用该类型） | long lFrameHeight |
| ul | unsigned long（win下不建议使用该类型） | unsigned long ulFrameWeight |
| s | short | short sValueSize |
| us | unsigned short(建议直接使用WORD) | unsigned short usValueSize |
| w | WORD | WORD wValueSize |
| dw | DWORD | DWORD dwValueSize |
| f | float | float fAccount; |
| d | double | double dMoney; |
| p | 指针变量 | t\_sysDict\* pSysDict; |
| pfn | 函数指针 | fn\_CodeSetFilter pfnCodeSetFilter = xxx; |
| lstr | 长度开头的串，（长度占用两个字节，后面每个元素占用两字节） | BYTE \* lstrPys |
| lbuf | 长度开头的缓冲区，长度占用两字节 | BYTE \* lbufPys |
| ap | 指针数组 (array of pointers) | t\_candEntry \*\* apEntrys |
| lp | 长指针类型 （不推荐使用） | LPVOID lpContex |
| h | 句柄 | HANDLE hEvent； |
| t （或 首单词全小写，不推荐） | 结构体 | t\_skinInfo m\_tSkinInfo; |
| o （或 首单词全小写，不推荐） | 对象 | t\_globalVariable& p\_oGlobalVariable |
| e | enum变量 | e\_btnState m\_eBtnState; |
| ec | enum常量，第一个元素必须显式声明初值 | enum e\_btnState  {  ecNormal = 0,  ecPressed  } |
| by | BYTE,unsigned char | unsigned char byData |
| c | 常量变量名 | const int g\_ciMaxPageSize = 99; |
|  |  |  |

## 4、变量名称

### 4.1命名方针

驼峰命名法，单词首字母大写，清晰表达出变量的作用。在所有命名中，都应使用标准的英文单词或缩写。不得使用拼音或拼音缩写，除非该名字描述的是中文特有的内容，如半角、全角, 声母、韵母等。所有命名都应遵循此原则，即名称应含义清晰、明确。

### 4.2命名长度

长度无强制性限制，Min-Length && Max-Information原则，做到“表意明确，见词达意”。

### 4.3资源的定义

图片 IDB\_       例：IDB\_USER

窗口 IDD\_        例：IDD\_LOGON, IDD\_LOG\_SEARCH

控件  IDC\_       例：IDC\_BTN\_SAVE, IDC\_EDIT\_USER\_NAME, IDC\_STATIC\_PASSWORD

字符串  IDS\_

菜单 IDM\_        例：IDM\_FILE, IDM\_VIEW

另外：菜单项基于父菜单命名，如视图下的菜单项电视墙IDM\_VIEW\_COPY

### 4.4禁止使用

变量名内部除了属性前缀与类型前缀之间，其他地方应该使用驼峰命名法，尽量不用下划线分割变量名，不允许使用i, j, k这样的变量。

例外情况：如果循环不太长（一屏幕可以装下）可以使用i，j，k。但仅限于循环体内部。

### 4.5避免使用的名字

**只是大小写的区别，如循环变量i和I**

**预定义**  注意不要和C++语言预定义的词语冲突。

**容易读写错误的文字**   如1 - l    1 - I    . - ,    0 - O    2 - Z    ; - :    S - 5

### 4.6特别注意

（1）变量定义推荐每个单独一行，定义时**【必须】**进行初始化；

（2）变量定义“就近定义”原则，尽量贴近使用处定义。

（3）变量定义“越晚越好”原则，这样在返回前可以省去某些构造函数的调用。

（4）由于构造和析构顺序和时机都存在不确定性，务必谨慎使用非原生类型全局变量和静态变量。原则上禁止使用非原生类型的全局变量和静态变量。

# 三、函数命名及使用规约

## 1、准则：“表意清晰，见词达意”

通过函数名应该能够清晰的得到如下信息：

* 函数的属性（全局函数、局部静态函数、普通成员函数、静态成员函数）
* 函数的用处（Min-Length && Max-Information原则）

## 2、函数使用规约

调用全局函数时【务必】以::为前缀. 使用WinAPI务必以::为前缀.

静态成员函数请尽量带上类名，如CMyClass::CreateInstance\_S();

普通成员函数推荐this->MemberFun()方式调用（不强求）。

调用基类成员函数时建议在函数名前加上“基类名::” （不强求）。

## 3、函数命名规约

* 采用 Windows API 的命名方式命名函数。函数名第一个字母必须大写，各单词首字母大写。
* 函数名应该能清晰反映函数的功能、用途。推荐使用动宾结构命名函数。
* 局部函数（即普通静态函数和匿名空间函数）必须以\_L结尾；只要是非全局函数，请务必带上static关键字，或者务必放入匿名空间中，防止链接错误。
* 静态成员函数以\_S结尾；
* 必须在虚函数声明处写明virtual，不论处于继承体系中的那一层；
* 虚函数以\_V结尾；
* 专用于回调的函数以Cb开头，比如void CbLogWarning（const TCHAR \* pszWarning）; （建议而不强制）
* 注意const函数使用，及时注明const。 **Use const whenever you need！**
* 推荐使用命名空间分类定义全局函数。
* 类的private成员函数，如果因为类的public接口调用它之前已经执行了参数有效性检查，不需要它重复执行此检查，则命名时**必须加上后缀 \_Unsafe**。以此命名提示开发人员做到以下两点：1禁止外部调用private后缀为\_unsafe的成员函数，只能同类中的其他成员函数调用；2. 后缀为\_unsafe的private成员函数内部没做参数、成员变量等有效性检查，必须由调用者来保证。

## 4、参数规约

* 返回类型和函数名在同一行, 参数也尽量放在同一行.
* 参数名的书写要完整，不要贪图省事只写参数的类型而省略参数名字。
* 函数参数以“p\_类型“作为前缀，跟局部变量以示区别。
* 形参定义顺序应该采用先输入参数，后输出参数的原则。

特殊情况：StringCopy（）、MyMemCopy（）等应将目的参数放在前面，源参数放在后面。

* 非原生类型的形参，如非必要，必须采用const &方式传递参数（输出参数不用const），如非必要，少用指针传递（减少指针有效性检查的开销）；
* 如有可能，定义IN、OUT、INOUT宏，在模块导出函数形参中注明（不强求），特别是提供给他人用的函数
* 建议函数的参数不要过多。如果参数太多，在使用时容易将参数类型或顺序搞错。

## 5、返回值的规则

* 不要将正常值和错误标志混在一起返回。正常值用输出参数获得，而错误标志用return语句返回。
* 如果函数的返回值是一个对象，注意“引用传递”与“值传递”的区别；

## 6、函数内部实现的规则

* 在函数体的“入口处”，对参数的有效性进行检查。并正确使用“断言”（assert）来给出提示。
* 在函数体的“出口处”，对return语句的正确性和效率进行检查。

（1）return语句不可返回指向“栈内存”的“指针”或者“引用”。

（2）要搞清楚返回的究竟是“值”、“指针”还是“引用”。

（3）如果函数返回值是一个对象，要考虑return语句的效率。

（4）单个函数内有多个return出口时，推荐使用goto语句设置一个出口或者使用智能指针。（goto允许被使用的仅有两个场合之一，另一个场合是跳出多重循环）

## 7、其它建议

* 函数的功能尽量单一。功能越多越难理解，使用也越易出错。
* 尽量避免函数带有“记忆”功能。相同的输入应当产生相同的输出。建议尽量少用static局部变量，除非必需。
* 要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

# 四、const使用规约

**注意：Use const whenever you need！**

## 1 const提高函数的健壮性

* 输入参数采用“指针传递”，必须使用const修饰可以防止意外地改动该指针，起到保护作用。
* 输入参数故意采用“值传递”，该输入参数只是实参的一个复制，所以不要加const修饰。
* 非内部数据类型的参数而言，务必使用const &方式传递参数。（为提高效率）
* 原生数据类型的参数不需要const修饰。

## 2 const修饰函数的返回值

* 函数返回值根据需要尽量添加const修饰。该返回值只能被赋给加const修饰的类型变量。

## 3 const成员函数

* 任何不会修改数据成员的函数都应该声明为const类型。如果在编写const成员函数时，不慎修改了数据成员，或者调用了其它非const成员函数，编译器将指出错误，这无疑会提高程序的健壮性。

# 五、goto使用规约

## 1、禁止使用goto语句。仅仅以下两种情况例外。

**例外1**：函数内部有多个return返回时。强烈推荐使用goto设置唯一的一个return出口；

好处：（1）在该出口处进行所有资源的释放和回收，防止回收有遗漏；

（2）减少编译时局部展开带来的开销

**例外2**：跳出多重循环时。

## 2、尽量使用智能析构来自动释放资源

对于局部临时分配的资源（内存、内核对象等），建议使用智能析构进行管理。

# 六、assert使用规约

断言assert是仅在Debug版本起作用的宏，它用于检查“不应该”发生的情况。

* 鼓励使用断言捕捉并报警不应该发生的非法情况。但不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的。
* 在函数的入口处，使用断言检查参数的有效性（合法性）。

# 七、版式与对齐

版式虽然不会影响程序的功能和性能，但会影响可读性、可维护性。程序的版式追求清晰、美观，是程序风格的重要构成因素。

## 1长行拆分

* 代码行过长（超过一屏宽度）时应该进行换行拆分；
* 长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

|  |  |
| --- | --- |
| if( (LongExpress1 >= LongExpress 2)  && (LongExpress3 <= LongExpress4)  && (LongExpress5 <= LongExpress6) )  {  dosomething();  } | for( LongExpress1;  LongExpress2;  LongExpress3 )  {  dosomething();  } |
| virtual CMatrix CMultiplyMatrix (CMatrix leftMatrix,  CMatrix rightMatrix); | |

## 2空行

空行起着分隔程序段落（逻辑）的作用。空行得体（不过多也不过少）将使程序的布局更加清晰。空行不会浪费内存，所以不要舍不得用空行。

* 在每个类声明之后、每个函数定义结束之后都要加空行。
* 类成员变量按照功能相关性分块儿用空行分隔；
* 类成员函数按照功能相关性分块而用空行分隔；
* 在一个函数体内，逻揖上密切相关的语句之间不加空行，其它地方应加空行分隔。

## 3代码行

* 一行代码尽量只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便调试。
* if、for、while、do等语句自占一行，执行语句不能紧跟其后。不论执行语句有多少都要加{}。
* 尽可能在定义变量的同时进行初始化。

## 4空格

示例:

1. void CopyCandEntry( const t\_candEntry2 \*aEntry, t\_bufHeap& buf, int nIdx );
2. If( nLen >= m\_data.m\_arrayItems.GetLen(p\_nIdx) )

* 为提高可读性，最外层的‘（’之后，‘）’之前应该添加空格。
* 逗号‘，’之后要留空格，如Function( x, y, z )。
* 分号‘;’不是一行的结束符号，其后要留空格，如for( initialization; condition; update )
* 赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如、“+=” “>=”、“<=”、“+”、“\*”、“%”、“&&”、“||”、“<<”,“^”等二元操作符的前后应当加空格。
* 一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。
* “.”、“->”这类操作符前后不加空格。
* 特例：对于表达式比较长的for语句和if语句，为了紧凑起见可以适当地去掉一些空格，如for( i=0; i<10; i++ )和if ( (a<=b) && (c<=d) )

## 5对齐

* TAB键所占空格数采用VC集成环境中的默认设置（4个空格）；
* 程序的分界符‘{’和‘}’应独占一行并且位于同一列，同时与引用它们的语句左对齐；
* 如果出现嵌套的｛｝，则使用缩进TAB对齐；

建议：

（1）函数声明时按照功能相关性分组，用空行分隔。返回值、函数名依次对齐；

（2）成员变量声明时按照功能相关性分组，用空行分隔。类型、变量名分别对齐；

## 6类内部声明版式

* 先public，在protected，后private;
* 先函数，后数据。
* 按照功能相关性分组

# 八、关于LOG的输出和查看：

## 1 LOG的重要性

* 线上项目的调试机会非常之少，通过LOG输出可以定位一些基本问题，查找错误原因；
* 对于开发人员和测试人员来讲，可以通过LOG方便的定位问题和分析问题原因；

## 2 LOG使用方法和规范

* 尽量使用统一的log输出方式MyLogInfo/MyLogWarning/MyLogError，减少原有debug模式下

的MyTrace的使用，减少MyOutputDebugInfo的使用，从而统一log输出方式；

* 参考链接：<http://www.cnblogs.com/luguo3000/p/3543487.html>
* 参考文档：《7 Log输出和使用方法及规范.docx》

# 九、注意事项和一些有益的建议

* 注意在继承体系中基类的析构函数是否应声明为虚函数。
* 在可能引起混淆的情况下，尽量用括号确定表达式的操作顺序。
* 不可将浮点变量用“==”或“！=”与任何数字比较。
* switch每个case语句的结尾尽量加break（除非有意使多个分支重叠）。
* switch不要忘记最后那个default分支。即使程序真的不需要default处理，也应该保留语句 default : break;
* 用malloc或new申请内存之后，应该立即检查指针值是否为NULL。
* 不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。
* 避免数组或指针的下标越界，特别要当心发生“多1”或者“少1”操作。
* 动态内存的申请与释放必须配对，防止内存泄漏。
* 用free或delete释放了内存之后，立即将指针设置为NULL，防止产生“野指针”。
* **慎用继承。若在逻辑上B是A的“一种”，并且A的所有功能和属性对B而言都有意义，则允许B继承A的功能和属性。**
* 若在逻辑上A是B的“一部分”（a part of），则不允许B从A派生，而是要用A和其它东西组合出B。
* 注意循环的效率（对象声明尽量放到循环外）
* 整数用 0, 实数用 0.0, 指针用 NULL, 字符 (串) 用 '\0'.
* 全局函数都要按功能分类放在不同的命名空间里。避免命名污染。
* 尽可能用 sizeof(varname) 代替 sizeof(type).
* 变量（指针、数组）被创建之后必须及时把它们初始化，以防止把未被初始化的变量当成右值使用。
* 当心变量的初值、缺省值错误，或者精度不够。
* 当心数据类型转换发生错误。尽量使用显式的数据类型转换，避免让编译器轻悄悄地进行隐式的数据类型转换。
* 当心变量发生上溢或下溢
* 当心数组的下标越界。
* 当心文件I/O有错误。
* 必要时果断重构。
* 把编译器的选择项设置为最严格状态。
* 禁用CreateThread
* 线程自产自销
* 模块化，以达到“拿走即用”为目标。（包括项目的整体使用、单独文件和单独类的使用）
* 如非必要，禁用非POD类型的全局变量和静态变量
* 增加项目词汇表
* 枚举的声明都要在类或命名空间中，这样就不会有全局定义冲突的问题
* 变量声明时\*和&尽量靠近变量；
* 除少数特定环境外，不要重载运算符。可以定义代替的具名函数。
* 不允许使用缺省函数参数。

# 十、跨平台

内核代码要在多种平台上编译使用，因而代码要避免使用无法通用的语法。现收集到以下注意事项：

## 枚举：

enum e\_toneString : char {…}; ×

enum e\_toneString {…}; √

Mac平台编译器是不支持enum指定底层类型的。所以上述枚举定义要去掉“:char”这部分，只能采用默认的int作为底层类型。

**数组的常量值：**

int aiNum[] = {1,2,3,}; ×

int aiNum[] = {1,2,3}; √

Mac上gcc不认识上述数组常量值最后的逗号，会报错，所以数组常量最后一个元素后面不要有逗号。

**#include预处理指令文件名大小写：**

写#include预处理指令时, 需要保证指令中的文件名与文件系统中文件的命名大小写**严格一致**。

假设存在t\_ArrayInScope.h需要include来引用，则

#include “T\_ARRAYINSCOPE.H” ×

#include “t\_arrayinscope.h” ×

#include “t\_ArrayInScope.h” √

否则会出现在Windows平台下编译通过，在Linux或者其他严格区分文件名大小写的平台下编译失败的情况。

# 十一、其他待定情况：

注释规约（待补充）

文件和目录规约（待定）

（以下待定）

使用 C++ 的类型转换, 如 static\_cast<>(). 不要使用 int y = (int)x 或 int y = int(x) 等转换方式;

不要使用 C 风格类型转换. 而应该使用 C++ 风格.

用 static\_cast 替代 C 风格的值转换, 或某个类指针需要明确的向上转换为父类指针时.

用 const\_cast 去掉 const 限定符.

用 reinterpret\_cast 指针类型和整型或其它指针之间进行不安全的相互转换. 仅在你对所做一切了然于心时使用.

dynamic\_cast 测试代码以外不要使用. 除非是单元测试, 如果你需要在运行时确定类型信息, 说明有 [设计缺陷](file:///E:\MyDocument\ProjectSample\编码规范\Google编码规范C++\designflaw).

# 禁止友元

由于友元破坏封装性，**禁止在项目中使用friend关键字，禁止使用友元。**

# 禁止使用的写法

* 单个原生类型T的对象严禁使用如下的写法进行内存分配。

T \*pObj = new T(value)

* 因为单个原生类型对象直接值拷贝就可以满足需要，使用上述写法，不仅增加了内存分配和回收的时间开销，而且容易和T\* pObjArray = new T[nElemNum]数组分配的写法发生混淆，使代码不易读、易出错，所以要直接在栈上声明并使用，需要向上层调用者传递T对象时，直接采用值拷贝的写法，如下所示。

声明： T obj = value

值传递：p\_obj = obj

# 内存操作检查

应该对内存的下列相关操作，进行必要的保护：

* 内存申请

申请的内存，【必须】判断是否为空

* 内存释放

释放之后，【必须】置空对应的指针变量，无论全局、成员、局部，杜绝野指针操作的可能性。

* 内存地址有效性

对于申请、参数传递等各种内存操作，【必须】对指针的有效性进行必要的判断，避免读写空指针造成崩溃

* 内存越界

对于源地址和目的地址，【必须】对读写的内存地址范围是否越界进行检查，避免前越界和后越界造成的崩溃、以及其他各种Undefined Behavior问题。

* 内存重叠

内存数据拷贝和移动的时候，【必须】对源地址和目的地址的重叠情况进行检查。