代码规范

G20小组

# 文档说明

## 文档信息

文档信息表

|  |  |
| --- | --- |
| **文档作者** | 杨际仟 |
| **创建日期** | 2019年5月27日 |
| **当前版本** | V0.1 |
| **上次版本** |  |

## 变更记录

变更记录表

|  |  |  |  |
| --- | --- | --- | --- |
| **变更日期** | **变更人** | **版本** | **备注** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[1 编程风格 4](#_Toc306288680)

[1.1 统一编程风格的意义 4](#_Toc306288681)

[1.2 变量命名的规则 4](#_Toc306288682)

[1.3 函数的命名规范 8](#_Toc306288683)

[1.4 函数参数规范 8](#_Toc306288684)

[1.5 引出函数规范 9](#_Toc306288685)

[1.6 注释规范 9](#_Toc306288686)

[1.7 语句规则 11](#_Toc306288686)

[1.8 对象规则 12](#_Toc306288686)

[2 代码组织 14](#_Toc306288687)

[3 代码优化 14](#_Toc306288688)

[3.1 代码优化的意义 14](#_Toc306288689)

[3.2代码优化的方法 14](#_Toc306288690)

[4 调试技巧 16](#_Toc306288691)

[4.1静态检查 16](#_Toc306288692)

[4.2 上机调试 16](#_Toc306288693)

# 1 编程风格

## 1.1 统一编程风格的意义

* 增加开发过程代码的强壮性、可读性、易维护性
* 减少有经验和无经验开发人员编程所需的脑力工作
* 为软件的良好维护性打下好的基础
* 在项目范围内统一代码风格
* 通过人为以及自动的方式对最终软件应用质量标准
* 使开发人员快速适应项目氛围
* 一个优秀的开发团队所必需的素质

## 1.2 变量命名的规则

①变量的命名规则要求用“匈牙利法则”。

即开头字母用变量的类型，其余部分用变量的英文意思或其英文意思的缩写,尽量避免用中文的拼音,要求单词的第一个字母应大写。 即：     变量名=变量类型+变量的英文意思（或缩写）

对非通用的变量，在定义时加入注释说明，变量定义尽量可能放在函数的开始处。

见下表：

bool(BOOL)     用b开头     bIsParent

byte(BYTE)     用by开头     byFlag

short(int)     用n开头     nStepCount

long(LONG)     用l开头     lSum

char(CHAR)     用c开头     cCount

float(FLOAT)     用f开头     fAvg

double(DOUBLE)     用d开头     dDeta

void(VOID)     用v开头     vVariant

unsigned     int（WORD）     用w开头     wCount

unsigned     long(DWORD)     用dw开头     dwBroad

HANDLE（HINSTANCE）     用h开头     hHandle

DWORD     用dw开头     dwWord

LPCSTR(LPCTSTR)     用str开头     strString

用0结尾的字符串     用sz开头     szFileName

②、指针变量命名的基本原则为：

对一重指针变量的基本原则为：

“p”+变量类型前缀+命名

如一个float\*型应该表示为pfStat

对多重指针变量的基本规则为：

二重指针：     “pp”+变量类型前缀+命名

三重指针：     “ppp”+变量类型前缀+命名

③、全局变量用g\_开头,如一个全局的长型变量定义为g\_lFailCount,即：变量名=g\_+变量类型+变量的英文意思（或缩写）

④、静态变量用s\_开头,如一个静态的指针变量定义为s\_plPerv\_Inst,即：     变量名=s\_+变量类型+变量的英文意思（或缩写）

⑤、成员变量用m\_开头,如一个长型成员变量定义为m\_lCount;即：变量名=m\_+变量类型+变量的英文意思（或缩写）

⑥、对枚举类型（enum）中的变量，要求用枚举变量或其缩写做前缀。并且要求用大写。

如：enum     cmEMDAYS

    {

EMDAYS\_MONDAY;

    EMDAYS\_TUESDAY;

    ……

    };

⑦、对struct、union、class变量的命名要求定义的类型用大写。并要加上前缀，其内部变量的命名规则与变量命名规则一致。

    结构一般用S开头

    如：struct     ScmNPoint

    {

    int     nX;//点的X位置

    int     nY;     //点的Y位置

    };

    联合体一般用U开头

    如:     union     UcmLPoint

    {

    long     lX;

    long     lY;

    }

    类一般用C开头

    如：

    class     CcmFPoint

    {

    public:

    float     fPoint;

    };

   对一般的结构应该定义为类模板，为以后的扩展性考虑

    如：

    template

    class     CcmTVector3d

    {

    public:

    TYPE     x,y,z;

    };

⑧、对常量（包括错误的编码）命名，要求常量名用大写，常量名用英文表达其意思。         如：#define     CM\_FILE\_NOT\_FOUND     CMMAKEHR(0X20B)     其中CM表示类别。

⑨、对const     的变量要求在变量的命名规则前加入c\_,即：c\_+变量命名规则；例如：        const     char\*     c\_szFileName;

## 1.3 函数的命名规范

函数的命名应该尽量用英文表达出函数完成的功能。遵循动宾结构的命名法则，函数名中动词在前,并在命名前加入函数的前缀，函数名的长度不得少于8个字母。

例如：

    long     cmGetDeviceCount(……);

## 1.4 函数参数规范

①、     参数名称的命名参照变量命名规范。

②、 为了提高程序的运行效率，减少参数占用的堆栈，传递大结构的参数，一律采用指针或引用方式传递。

③、 为了便于其他程序员识别某个指针参数是入口参数还是出口参数，同时便于编译器检查错误，应该在入口参数前加入const标志。如：

…cmCopyString(const     char     \*     c\_szSource,     char     \*     szDest)

## 

## 1.5 引出函数规范

对于从动态库引出作为二次开发函数公开的函数，为了能与其他函数以及Windows的函数区分，采用类别前缀+基本命名规则的方法命名。例如：在对动态库中引出的一个图象编辑的函数定义为     imgFunctionname(其中img为image缩写)。

    现给出三种库的命名前缀：

1. 对通用函数库，采用cm为前缀。
2. 对三维函数库，采用vr为前缀。
3. 对图象函数库，采用img为前缀。
4. 对宏定义，结果代码用同样的前缀。

## 1.6注释规范

1）、函数头的注释

    对于函数，应该从“功能”，“参数”，“返回值”、“主要思路”、“调用方法”、“日期”六个方面用如下格式注释：

    ①、     对于某些函数，其部分参数为传入值，而部分参数为传出值，所以对参数要详细说明该参数是入口参数，还是出口参数，对于某些意义不明确的参数还要做详细说明（例如：以角度作为参数时，要说明该角度参数是以弧度（PI）,还是以度为单位）,对既是入口又是出口的变量应该在入口和出口处同时标明。等等。

    ②、     函数的注释应该放置在函数的头文件中，在实现文件中的该函数的实现部分应该同时放置该注释。

    ③、     在注释中应该详细说明函数的主要实现思路、特别要注明自己的一些想法，如果有必要则应该写明对想法产生的来由。对一些模仿的函数应该注释上函数的出处。

    ④、     在注释中详细注明函数的适当调用方法，对于返回值的处理方法等。在注释中要强调调用时的危险方面，可能出错的地方。

    ⑤、     对日期的注释要求记录从开始写函数到结束函数的测试之间的日期。

    ⑥、     对函数注释开始到函数命名之间应该有一组用来标识的特殊字符串。

    如果算法比较复杂，或算法中的变量定义与位置有关，则要求对变量的定义进行图解。对难以理解的算法能图解尽量图解。

    2、变量的注释：

    对于变量的注释紧跟在变量的后面说明变量的作用。原则上对于每个变量应该注释，但对于意义非常明显的变量，如：i,j等循环变量可以不注释。

    例如：     long     lLineCount     //线的根数。

    3、其他注释：

    在函数内我们不需要注释每一行语句。但必须在各功能模块的每一主要部分之前添加块注释，注释每一组语句，在循环、流程的各分支等，尽可能多加以注释。

    其中的循环、条件、选择等位置必须注释。

对于前后顺序不能颠倒的情况，建议在注释中增加序号。

## 1.7 语句规则

简单语句的通用规则:

* 一条语句通常以分号作为结束符。

**实例:**

var values = ["Volvo", "Saab", "Fiat"];  
  
var person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue"  
};

复杂语句的通用规则:

* 将左花括号放在第一行的结尾。
* 左花括号前添加一空格。
* 将右花括号独立放在一行。
* 不要以分号结束一个复杂的声明。

**函数:**

function toCelsius(fahrenheit) {  
    return (5 / 9) \* (fahrenheit - 32);  
}

**循环:**

for (i = 0; i < 5; i++) {  
    x += i;  
}

**条件语句:**

if (time < 20) {  
    greeting = "Good day";  
}

else {  
    greeting = "Good evening";  
}

## 1.8 对象规则

对象定义的规则:

* 将左花括号与类名放在同一行。
* 冒号与属性值间有个空格。
* 字符串使用双引号，数字不需要。
* 最后一个属性-值对后面不要添加逗号。
* 将右花括号独立放在一行，并以分号作为结束符号。

**实例:**

var person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue"  
};

短的对象代码可以直接写成一行:

**实例:**

var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

# 2 代码组织

代码组织是对整个项目的代码进行整理，使之更加有序。实现类似功能的文件应该放在同一个文件夹中或者同一个项目中。尽量用树形结构组织模块，即尽量保证单级调用关系，即上级调用者不要使用下级调用的模块中的函数、变量，不要有跨级调用关系，更不要有循环调用关系，可以有自身的递归调用关系。

# 

# 3 代码优化

## 3.1 代码优化的意义

* 仅仅对符合功能说明书的要求、能正确运行的代码进行优化是有意义的
* 代码优化能减少冗余代码的数量，用更少的代码来实现同样的功能
* 提高代码的内聚程度，减少耦合程度

对代码的抽象能提高代码的重用度，对今后其他项目的进度有非常重要的意义

## 3.2代码优化的方法

1）选择一个更好的算法

应该熟悉算法语言,知道各种算法的优缺点,一般很多计算机资料文本上有介绍,应该能够看得懂算法描述。

还要选择一种合适的数据结构(记着,你的程序所干的唯一一件事就是在计算机里搬数,把一堆数从一个地方提出来,处理一下,甩到另一个地方,那么按什么方式搬数有多重要你应该知道了吧--东楼),比如你在一堆随机存放的数中使用了大量的插入和删除指令,那使用链表要快得多.如果你要做二分法查找,那提前排下序非常重要。

2）写一些清晰,可读性好并且简单的代码

3）透视你的程序

一个程序写出来,凭直觉就应该感觉出哪些地方快,哪些地方慢,一般说来,最快的运算就是分配一块内存,给指针赋值,还有就是两个整数的加法运算,别的都有点慢,最慢的就要数打开文件,打开新的进程,读写一大块内存,寻找,排序等等

4）理解你的编译程序选项

许多编译程序有几级优化选项,注意使用最优化的一项,特别注意gcc,优化选项非常多,小心使用,别弄得适得其反。

# 

# 4 调试技巧

## 4.1静态检查

先进行人工检查，即静态检查。

　　在写好一个程序以后，不要匆匆忙忙上机，而应对程序进行人工检查。这一步十分重要，它能发现程序设计人员由于疏忽而造成的多数错误。这一步往往容易被人忽视，总希望把一切都推给计算机去做，但这样会多占用机器时间，作为一个程序人员应当养成严谨的作风，每一步都要严格把关，不把问题留给后面的工序。

　　为了更有效地进行人工检查，所编的程序应力求做到以下几点：

　　①应当采用结构化程序方法编程，以增加可读性；

　　②尽可能多加注释，以帮助理解每段程序的作用；

　　③在编写复杂的程序时不要将全部语句都写在main函数中，而要多利用函数，用一个函数来实现一个单独的功能。各函数之间除用参数传递数据外，尽量少出现耦合关系，这样便于分别检查和处理。

## 4.2 上机调试

上机调试即动态调试，通过上机发现错误称为动态检查。在编译时会给出语法错误的信息，调试时可以根据提示信息具体找出程序中出错之处并改正。应当注意的是有时提示出错的地方并不是真正出错的位置，如果在提示出错的行找不到错误的话应当到上一行再找。有时提示出错的类型并非绝对准确，由于出错的情况繁多且各种错误互有关联，因此要善于分析，找出真正的错误，而不要只从字面意义上找出错信息，钻牛角尖。

　　如果系统提示的出错信息很多，应当从上到下逐一改正。有时显示出一大片出错信息往往使人感到问题严重，无从下手。其实可能只有一二个错误。例如，对使用的变量未定义，编译时就会对所有含该变量的语句发出出错信息。这时只要加上一个变量定义，就所有错误都消除了