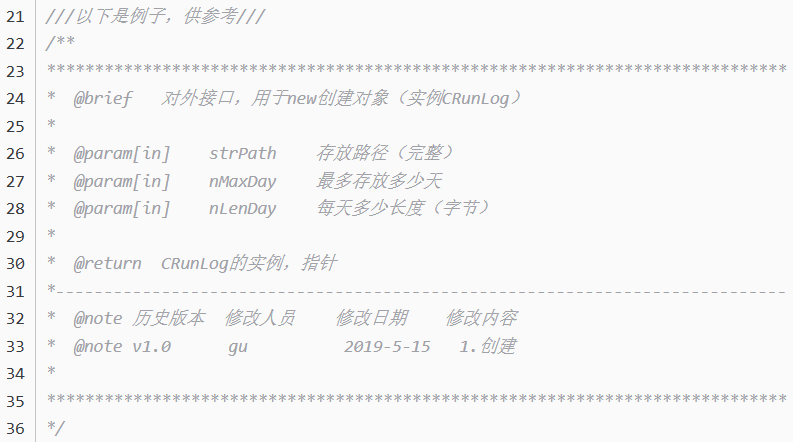
**编码指导和规范**

王小娜

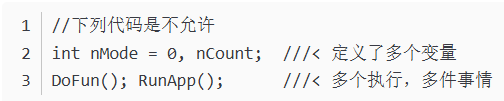
2024年六月

1. 函数
2. 声明

内容：函数功能，（入参、出参）参数描述；



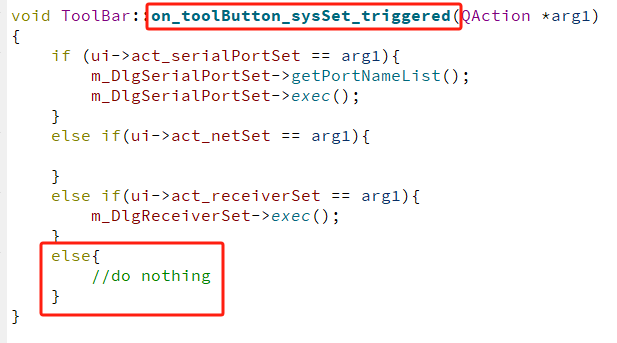
1. 编码行为规范
2. 一行代码只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便于写注释。



1. if、for、while、do 等语句自占一行，执行语句不得紧跟其后



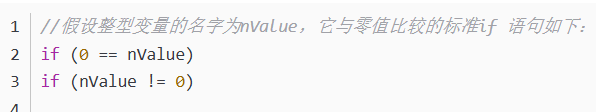
1. **if语句 必须考虑else的处理。 对于主要逻辑判断，必须写else处理**



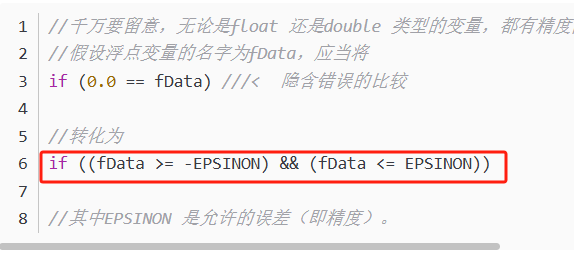


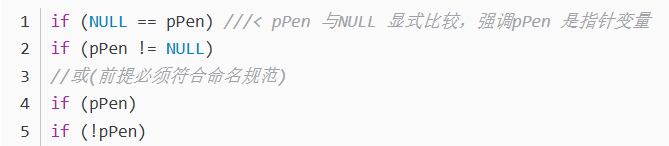
1. **不同类型和0值的比较：最好是变量在后面，常量在前面，免得写成nValue = 0；指针变量考虑情况，如qt需要比较的对象不是NULL，而是nullptr；**





**浮点型数据的零值比较：**





1. 出入口检查，安全

**在函数体的“入口处”，对参数的有效性进行检查；**

1. **每行代码不要太长：尤其是算法计算代码，变量多，非常长，理论要求的长度是视图框宽度略小一些。**
2. **Switch语句**
3. 每个case 语句的结尾不要忘了加break，否则将导致多个分支重叠（除非有意使多个分支重叠）。
4. 若某个case不需要break一定要加注释声明，便于明白是故意不加而不是忘记。
5. 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;
6. 循环语句

为了防止循环失去控制，尽量不在循环体内修改循环条件变量，首先是规避，然后如果无法规避，必须注释清楚。

1. 空行
2. 连续超过10行的代码必须要添加空行。
3. 不同性质的代码（或变量）之间要有空行
4. 相同性质的代码应该放在一起，超过10个的，也要添加空行
5. 性质是指 属性、含义、功能相似等条件，可以根据实际情况来区别
6. **返回值和log**

是否运行结束：true和false

不同阶段的处理说明：错误码或log输出

示例：

入参判断 ——错误码或log输出

处理过程1 ——错误码或log输出

处理过程2 ——错误码或log输出

处理过程3 ——错误码或log输出

结果判断——错误码或log输出

结束——返回值

1. 命名规范
2. 两个基本原则：
3. 含义清晰，不易混淆；
4. 不和其它模块、系统API的命名空间相冲突
5. 大驼峰和小驼峰：

大写首字母单词+大写首字母单词：AntennaScan

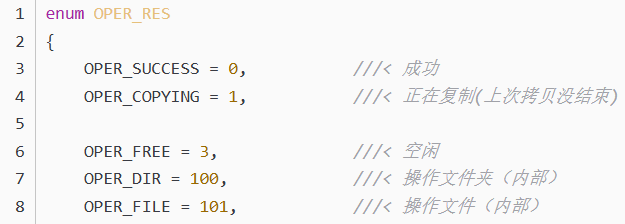
小写+ ‘\_’+小写单词分割形式：antenna\_scan.cpp, antenna\_scan.h

1. 常用命名方式：
2. 变量、参数用小写类型缩写+大写字母开头的单词组合而成。

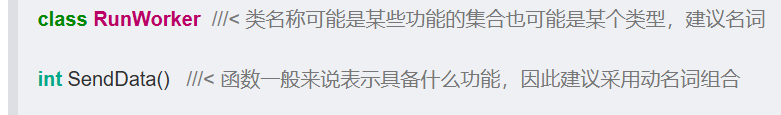


1. 禁止使用拼音首字母进行创造性缩写
2. 行业性缩写用语用大写缩写：如SNR（信噪比）等
3. 类成员变量前缀m\_（表示member）
4. 全局变量前缀g\_（表示global）
5. 静态变量加前缀s\_（表示static）
6. 常量、枚举全用大写字母，用下划线分割

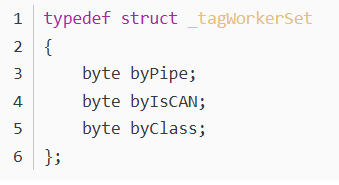




1. 类名和函数名用大写字母开头的单词组合而成



1. 自定义结构体名称要以“\_tag”为开头

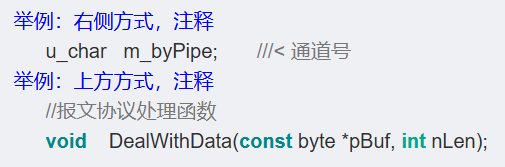


1. 函数名：一般采用动词形式，以大驼峰方式命名。例如： void GetPosition();
2. 注释
3. 代码注释形式

/\*添加方式\*/

注释采用上方以及右侧两种方式。

注释应当准确、易懂，防止注释有二义性。



1. 注释位置
2. 文件说明

如头文件.h文件、.inc文件、.def文件、编译说明文件.cfg等，在文件头进行注释

1. 函数头部
2. 重要的变量

全局变量、成员变量、静态变量、常量要有的注释

1. 复杂的处理过程说明

公式或原理参考

1. 算法的逻辑顺序
2. 版本更新的变更或者新增

作者和时间，对应版本或错误

例子：/\*增加一个延迟，分散开各个线程的运行时刻 add by wxn 20230530\*/

1. 错误排查
2. 二分法
3. Log辅助
4. 设计
5. 正常处理逻辑
6. 逻辑通顺后再编写代码；
7. 不同阶段的处理说明；
8. 所有分支：if下的else、switch下的default等；
9. 错误码或log输出；

——排查错误和bug

——异常处理

1. 非正常处理逻辑
2. 整个逻辑流程中所有的非正常都要考虑；

如：采集卡采集失败

采集卡采集过快或者过慢

采集数据错误：类型错误、长度错误等

采集数据的处理速度和采集速度的匹配

1. 内存管理
2. 内存泄漏
3. 申请和释放：

动态内存的申请与释放必须配对（特殊应用可特殊处理，但不建议把特殊当成常规），不需要的内存应被及时释放，防止内存泄漏。

并且new、delete和new[]、delete[]要成对使用。**特别要注意中间return**。

1. 防止野指针
2. 野指针说明：

野指针指向一个已删除的对象或未申请访问受限内存区域的指针。

与空指针不同，野指针无法通过简单地判断是否为 NULL避免，而只能通过养成良好的编程习惯来尽力减少。

对野指针进行操作很容易造成程序错误。

1. 释放了内存之后，立即将指针设置为NULL，防止产生“野指针”，除非该指针变量本身将要消亡；

delete m\_pTxOri;

m\_pTxOri = NULL; ///< 及时变成NULL

delete[] m\_pWriteBuf;

m\_pWriteBuf = NULL; ///< 及时变成NULL

1. 申请内存之后，应该立即检查指针值是否为NULL。防止使用指针值为NULL的内存；
2. 指针赋值——内存释放后，呗赋值的指针最好处理= NULL：

byte \*pBuf = m\_pTxOriBuf;

//do somethings

delete[] m\_pTxOriBuf;

m\_pTxOriBuf = NULL; ///< 及时变成NULL

//.....

//...

//do somethings

if(pBuf != NULL)

{

//**此时pBuf 是野指针了，比较容易忽视这一点。**

pBuf[0] = 0;

}

1. 控制方法：

new delete 在一个函数内，最好是在一个屏幕可视范围内

new出来的对象，不要被其他对象进行引用

特别是结构体的成员或类成员是指针，要注意被引用的可能。

来一个比较隐蔽的例子：

void CallFun1(byte \*pBuf)

{

//....

delete[] pBuf;

pBuf = NULL;

}

void CallFun2(byte \*pBuf) //**关于指针传递、值传递、引用**

{

//....

pBuf[0] = 0x00;

//....

}

void main()

{

byte \*pTxBuf = new byte[250];

//do somethings

CallFun1(pTxBuf);

//.....

//...

//do somethings

if(pTxBuf != NULL)

{

//此时pBuf 是野指针了，比较容易忽视这一点。

CallFun2(pTxBuf);

}

}

1. 如果是用第三方一些API或机制，一定要注意是否有内存泄露的风险。比如在GDI中，资源如果不回收，那么就会内存泄露。

Image\* pImage = Image::FromFile(L"E:\\bac.bmp");

Graphics g(pDC->m\_hDC);

g.DrawImage(pImage,0,0);

delete pImage; ///< 必须，否则泄露。这个取决API函数的特性

g.ReleaseHDC(pDC->m\_hDC);

1. 数组越界
2. 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。
3. 容易导致越界：

数组下标是变量，当特殊情况下，变量可能变成-1或很大一个值

协议内容定义变化，原先协议是没问题，后来协议增加了，导致问题出现

const int MAX\_NMAE\_LEN = 16; ///< 原先是10，由于协议改变，现在变成16

char strName[10]; ///< 此处未及时改过来

for(int i = 0;i < MAX\_NMAE\_LEN;i++) ///< 已经越界

{

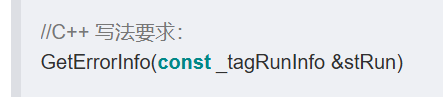
strName[i] = 0x30 + i;

}

1. 效率提升
2. 参数传递

如果输入参数以值传递的方式传递对象（类型为类、结构体），则用“const & ”方式来传递；这样可以省去临时对象的构造和析构过程，从而提高效率。

对于类型为常规类型，如char int等，建议采用直接传值的方式。



1. 其他
2. 代码行数要求

一个函数原则上要求代码行数不超过200行（包含注释、空行）

单个 类（.cpp文件）或.c文件源代码行控制在1000行以内。