C++编程规范

# 总体原则

* 代码规范确保两件事情：减少代码错误，降低代码维护难度。因此写代码时思考两件事：
  + **这段代码可能的隐患在哪里？**
    - 先思后行有助于规避逻辑错误。
    - 隐患是自测的重要用例来源。
  + **其他人（包括以后的自己）会怎么读这段代码？**
    - 保证代码易读性是维护代码的前提。
* **写任何代码都确保自测**
  + 代码分支必须全覆盖
  + 关键的算法更要重点测试、自动测试。‘
* **修改代码时应当尽量充分理解上下文、知晓历史问题**
  + 理解原来的设计意图才能避免改错，引发新的问题。
  + 知晓历史问题才能顾全兼容问题，不会重复犯错。
  + 如果遇到代码历史不可追溯、无人可以解答的情况，必须保证修改后的逻辑完全兼容之前的逻辑。充分评估修改影响范围，并告知测试与其他相关人员。
* **易读性是代码质量的重要标准**
  + 正式代码、临时代码、测试代码，只要未来可能有人（包括自己）会读就要保证易读性。
  + 除非代码执行效率与易读性冲突 且 执行效率极为关键，否则易读性的重要性更高。
* **设计和修改代码尽量遵循开闭原则**
  + 若原代码没有BUG，拓展业务首先考虑继承子类而非修改原有类型代码。
  + 不要无限延伸此概念，如果使用组合来代替继承更好、原有架构不足以支持新业务等情况，需积极重构。
  + 设计时应当合理考虑可能的拓展方向，使用合理的设计模式，不要太过局限，也不要太过发散。

# 注释

## 规则：所有public的类、函数、变量、属性必须加注释。

说明：

所有公开的接口会被人调用的代码应该带有注释，让调用者猜测用法是危险且不负责任的。注释应该满足一定的格式（见2.1.3），以便调用者在不接触源码的情况下，能够通过文档或IDE提示获知调用方式。析构函数除外。

## 规则：维护代码时一定要维护注释。

确保注释与代码的一致性，不再有用的注释要删除。

## 规则：注释的格式需满足注释格式规范

## 建议：注释应当说明为什么，不是干了什么。

## 建议：应当首先考虑提升代码的自解释性，如果逻辑实在复杂再注释说明。

优秀的代码可以自我注释，不通过注释不要为了注释而注释。

## 建议：废弃代码直接删除，不要注释。

不要将无用的代码留在注释中，随时可以从代码版本库中找回代码。即使只是想暂时排除代码，也要留个标注，不然可能会忘记处理它。建议使用 //TODO: 标记待处理的代码段。

## 建议：注释语言应当统一，建议使用中文。

# 命名

## 规则：没有最好的命名规则，但项目组内应当统一。

说明：项目组内部自行规定好命名风格。团队统一命名规范unix like风格或者windows风格，**不建议使用匈牙利命名法**，因为难以阅读。

C++命名规范参考各项目组或能力组的规范。

## 规则：除了循环下标以及有意义的名称，禁止使用单字母命名。

说明：变量名应该有意义。

## 规则：不要使用自创的缩写。

# 排版格式

## 规则：正确使用缩进。

说明：

* 每一级缩进4个空格，默认一个Tab就是4个空格

## 规则：正确使用空格。

说明：

* 操作符及逻辑判断符的两端添加空格，例如：

I = I + 1;

if (a == b)

* 左括号的右侧，右括号的左侧不要添加空格，例如：

CallProc(A);

List.Items[3]

* 变量声明关键字的右侧加空格，例如：

int a;

* 逗号左侧无空格，逗号的右侧有空格，例如：

void Proc(int aParam1, int aParam2);

Proc(A, B);

## 规则：正确使用空行。

说明：

* 每一个段落前后都空一行，包括函数开始、结束、类型声明开始、结束
* 加在一个类的实现前面的注释前后
* REGION前后

## 规则：正确大小写。

说明：

* 所有的关键字、保留字，一律小写，例如if, while, for等
* 文件名、函数名、变量名等参考【驼峰规则】

## 规则：正确使用换行。

说明：

* 一行代码的长度不要超过N个字符（不要超过右竖线）。

N建议为120。因为现代显示器通常为宽屏分辨率，原本80字符已经过时，120比较合适。此长度包括了代码加注释。

* 换行点的选择以不影响单词和语句的完整性为原则，一般选择在逗号分号之后，and，or之后。
* 转行后需要缩进两格。
* 对于在一行代码之后添加的注释导致超过了右竖线的情况，将注释作为单独一行，添加在需要注释的语句的上一行。
* 不允许把多个短语句写在一行中，即一行只写一条语句
* If…then/else独占一行（else if除外）

好的例子：

Inc(A);

Dec(B);

坏的例子：

Inc(A);Dec(B);

# 声明

## 规则：内部使用的类、函数、变量应声明成内部的。

说明：

* 可以放到implementation下的，就不要放到interface下。
* 可以放到private下的，不要放到public下。record 默认可见性是public。 class 默认为published，不需要published的放到public中， 声明类的时候尽量虽然是默认可见性尽量把published关键字添上。
* 考虑调用者的感受，给调用者开放最简洁的接口。

## 规则：引用其他文件时，如果存在重名，需要通过命名空间前缀来访问。

## 建议：引用其他文件时，按照依赖关系从底层开始排序，避免多余的单元、包依赖。

## 建议：建议避免交叉引用，即便是通过interface和implementation相互引用。

## 建议：类、函数的声明应放入正确合理的模块中，不要为了方便而编写“功能齐全”的巨大文件。

## 建议：常量、变量、函数合理的排列顺序。

# 函数

## 规则：尽可能减少重复代码，提取成函数。

## 规则：如果函数返回错误码或有明确的报错，调用者必须处理。

说明：API执行出现异常时，上层必须做出对应处理

好的例子：

void test()

{

string str = “123.45”;

double value;

//必须判断返回值，否则Value将是未定义的

  if(!TryStrToFloat(str, value))

{

value := 0.0;

}

  ShowMessage(FloatToStr(value));

}

案例：

1. 从加工轨迹获取指定位置的坐标，获取失败时的处理
   1. 历史版本API通过返回False告知上层
   2. 某一版API改为直接返回默认值（TPoint2D.Zero）
   3. 后果：执行沿轨迹后退时，机床将失败后的返回值作为空移目标位置，直接空移到原点位置（平面机台横梁直接快速划过整个幅面，幸亏当时没有站人）
2. 空移前状态检测未通过时的处理
   1. 主程序空移检测未通过，判断无法执行时，通过返回False告知上层，但上层未做处理。
   2. 后果：代码鲁棒性降低。其它模块的逻辑错误可能导致加工动作异常，进而切错甚至撞机（本该执行的空移未执行，却继续执行后续动作，位置全部错误）

## 规则：内部函数尽量少依赖外层变量

说明：

* [函数依赖最少，尽可能独立](https://docs.fscut.com/x/CZOLCg)

## 规则：子函数避免嵌套太深（不超过3层）。

## 建议：函数参数的合法性检查应统一规定。

说明：

* 应统一规定参数的合法性检查，由调用者还是被调用函数检查。缺省是由函数调用者负责。
* 避免合法性检查缺失，也避免重复检查浪费效率。
* 按引用/指针传参时，除非设计需要，不要改变指针中的内容。

## 建议：函数职责应单一。

说明：

* 明确函数功能，精确（而不是近似）地实现函数设计
* 一个函数仅完成一件功能
* 为简单功能编写函数
* 不要设计多用途面面俱到的函数
* 函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出

好的例子：

int max(int aLeft,int aRight)

{

if (aLeft > aRight)

{

return left;

}

else

{

return right;

}

}

void foo1()

{

int val = max(2,3);

}

void foo2()

{

int val = max(4,5);

}

## 建议：函数参数数量不宜过多（不超过5个）。

当传递参数较多时，建议封装成一个结构体，将结构体作为参数整体传递。

## 建议：函数内逻辑层级（缩进层级）不超过3层。

示例：<https://docs.fscut.com/x/FZKLCg> 。

例外：特殊算法可重载此规则。

## 建议：除非作为返回值，否则不要改动参数本身，如果算法需要改动则需先克隆参数数据。

说明：建议使用const修饰传参。

## 建议：越底层的模块越应减少对全局变量的依赖。

说明：

底层模块通常会被较多模块依赖或被应用在不同程序间，如果底层依赖了一些全局变量，会降低底层代码可维护性。

示例：生成工艺时应当由外部传入参数，而不是自己从全局设置CADConfig中获取参数。

示例：创建NcDevice时应当由外部传入初始化数值，而不是自己从全局变量MachineConfig中获取参数。

## 建议：减少函数本身或函数间的递归调用。

说明：

* 减少函数本身或函数间的递归调用。
* 递归次数应当可以被正确评估。

## 建议：避免过度封装。

说明：如一个函数能搞定的事情，不必封装了成一个类。避免强行套设计模式。

## 建议：函数高扇入，合理扇出（<7）。

说明：

扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。

扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数；而扇出过小，如总是1，表明函数的调用层次可能过多，这样不利程序阅读和函数结构的分析，并且程序运行时会对系统资源如堆栈空间等造成压力。函数较合理的扇出（调度函数除外）通常是3-5。扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函数进一步分解多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的功能，也不能违背函数间的独立性。

扇入越大，表明使用此函数的上级函数越多，这样的函数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

较良好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

## 建议：避免函数重载滥用。

# 变量

## 规则：不用或少用全局变量、全局函数。

说明：

* 涉及到多个任务/线程读写全局变量时，请谨慎。一般不允许有多个写入者。
* 可以写成类的成员变量，就不要写成全局变量。这样变量就有了所有者。
* 可重入函数中强烈不建议使用全局变量。若使用，则应通过关中断、信号量（即P 、V 操作）等手段对其加以保护。
* 避免使用接口类型的全局变量。

## 规则：避免使用魔数（Magic Number）。

说明：

避免使用魔数，除非具有公认的含义，应使用常量/常量表达式代替。不得不使用时必须注释说明。

示例：表示有限种状态时优先使用枚举以代替整型，见<https://docs.fscut.com/x/L5WLCg>。

## 规则：严禁使用未经初始化的变量作为右值。

说明：

* 全局变量、类的成员变量应当在声明处或构造函数中显式地初始化为一个确定的值。Delphi环境下，初始化在堆中的变量默认为0，初始化在栈中的变量可能不为0。因此，Delphi环境下，全部变量、类的成员变量，如果初值为0，可以不用显式地初始化。如果不嫌麻烦，建议初始化一下比较好。C/C++环境，所有变量必须初始化(包括初值为0/nil的)。
* 初始化位置应当统一，不要散落在代码各处。
* 明确全局变量的初始化顺序，避免跨模块的初始化依赖，避免造成低级而又灾难性的结果
* 变量初始化的位置离使用位置越近越好，也要避免无意义和不正确的初始化

示例：结构体应使用Default初始化，避免不初始化就作为右值使用，也应避免使用 FillChar填充0。（<https://docs.fscut.com/x/uwnmAQ>）

## 规则：减少共享数据。

说明：

在访问变量时，减少变量的数据共享，避免无意的数据修改。如函数传参时，对于不希望发生变化的数据，应使用const修饰或传递拷贝，避免直接传递引用。

## 建议：避免使用强制类型转换

说明：

* 尽量减少没有必要的数据类型默认转换与强制转换

# 类/结构/枚举

## 建议：本类的成员函数访问本类的成员变量时（同时存在setVal和getVal），应直接访问成员变量，而非setVal和getVal。

## 建议：遵循最小可见性原则，不需要暴露的接口不要放在public域中。

## 建议：可以将功能相关/对称的字段函数声明在相近的位置，也可以用public等标记分隔不同功能区块。

说明：不建议按照字母顺序排列，阅读代码跳跃太大，影响效率。

## 建议：类/结构功能应当单一，字段和方法间逻辑强相关。

## 建议：模块内如果不需要解耦，不要使用接口，直接使用类。

# 内存管理

## 规则：避免空指针访问。

说明：

* 使用指针/对象的地方，如果不能保证其一定存在，则需要判断是否为空。

**示例：**

**如果构造函数执行完，还有为NULL的成员变量指针（有的指针会在执行中根据条件创建对象），则在使用该指针变量时，必须增加空指针判断保护，因为依赖**的资源不一定能保证存在。

示例：

if (obj != nullptr)

{

obj.Do();

}

## ****规则：内存对象的生命周期应正确管理。****

说明：

* **对于类成员变量中的对象，创建对象尽量放在构造函数中，释放放在析构中，成对出现；**
* 一块内存的创建和释放，最好有一个所有者（Owner）。Owner负责对这块内存的创建和释放。
* 多个任务同时使用一块内存时，需要注意谁是所有者，什么时候所有权会变更。

## 规则：尽量保证哪里创建、哪里释放。

说明：

* 否则容易出现内存泄漏或重复释放问题。
* 接口/服务的提供者、跨模块调用的实现对象需要严格管理生命周期，并建立相应的约定。
* 若有特殊情况需注释说明。

## 规则：避免内存泄露。

说明：

* 创建和释放成对出现
* 不停地创建新的内存而不释放，可能导致软件运行占用的内存越来越大。系统会报错Out of Memory。

坏的例子：

while True do

begin

  TObject.Create;

end;

## 规则：避免内存越界。

说明：

* 访问数组、vector类型时，注意访问的序号不要超出范围。尤其是差1错误
* 注意避免把“<=”误写成“<”或“>=”误写成“>”。
* 使用指针操作，修改指针地址，尤其要谨慎。

## 规则：避免使用野指针/对象。

说明：

* 内存释放回收后，不得再被使用。
* 如果可以，在释放内存时，把相关指针赋值为nullptr。
* 经典的用法是对象在构造函数中创建，在析构函数中回收。

## 建议：尽量避免直接操纵内存。

直接内存操（Move、MemCopy）时必须严格检查逻辑并处理可能的异常，避免访问越界造成数据损坏。

## 建议：合理定义结构体。

说明：

* 仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，遵守对齐原则。
* 结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地。如：RTOS的共享内存结构体。
* 注意涉及到跨平台或者通讯时，大端和小端模式。

# 多线程

多线程编写如不注意一些细节，往往会带来较多难以排查的问题（典型的如概率性触发问题、系统性能严重下降）。

## 规则：多线程读写同一数据需要加保护。

说明：

当多个线程需要使用同一块内存时，尤其是写操作，可能会导致内存本身完整性出现问题，或者代码执行步骤混乱导致的不确定性后果。因此应当避免多线程读写同一个数据，避免不了的时候，就加锁、临界区进行保护；或者通过标志量避免同时访问。对于标志量的操作应当是原子的（如 InterlockedIncrement）。

## 规则：锁和临界区控制范围应尽可能减小。

说明：锁和临界区控制的范围尽可能小，最好只包括需要同时访问的变量，这样可以减小被锁的时间和概率。

## 建议：给线程命名。

说明：在排查多线程问题时，知道线程名能够提高排查效率。

## 规则：注意让出CPU。

说明：

* 在多线程的while循环中增加Sleep，避免while循环消耗过多CPU资源。
* 创建线程时，需要考虑线程的优先级问题，核心线程需要提升优先级，显示相关线程需要降低优先级。如插补线程需要使用最高优先级。

## 规则：禁止在非主线程中操作VCL控件。

说明：

* VCL不是多线程安全的，禁止在子线程中直接访问。

## 建议：仔细判断每一个变量是否存在访问冲突的可能，合理加锁。

## 建议：仔细判断每一把锁的出入口，避免死锁问题。

## 建议：注意使用多线程要有节制，避免线程过多。

说明：

* 只有在必要的情况下才使用多线程。
* 确保多线程不会引发严重问题。
* 避免频繁的创建大量线程，避免线程过多软件卡死。在必要的时候使用线程池。

坏的例子：

软件中为了支持50个脚本，在开软件的时候创建了50个线程，导致开软件耗时很长。

## 建议：使用多线程时确保进行了充分测试。

# 数值计算

## 原则：注意数值计算的精度要求。

说明：在对参数有精度要求的地方，如果涉及到跨平台的数据交互（上位机和RTOS传递参数），并且该数据交互的通道为整形传递时，需要进行缩放，此时需要考虑缩放倍数；另外在迭代算法中，出口处需要考虑所使用的数据类型是否能够达到出口精度。

## 规则：除零错误检查和保护。

说明：所有除法都要进行除0检查和保护，防止异常发生。

## 规则：避免数值计算溢出。

说明：

* 避免符号错误，常见比如：无符号的0减1导致溢出。
* 分母极小时会产生很大的计算误差，甚至溢出、异常错误等。除数为变量时，应当分析分母极小的可能性，做出相应处理。

## 规则：避免数值类型溢出。

说明：

* 变量是有范围的，极端条件下可能会溢出。如：记录编码器反馈的变量类型，不能使用Integer，要使用INT64。
* 常用数学函数使用时，要注意传入值域，防止轻微越界。如：Power传参不能为负。反三角函数传参不能大于1。
* 尽量避免数值类型转换。

坏的例子：

void setPower(unsigned int aValue)

{

laserPower := aValue;

}

void fol()

{

int power := -1;

  setPower(power); //laserPower 将会是一个巨大的整数$FFFFFF，造成风险

}

## 规则：浮点数比较应充分考虑引入合理容差。

说明：通常情况下，浮点数不能用等号比较大小，也应注意不能引入容差的情况（比如排序）。

## 建议：设计函数调用关系时应仔细计算容差的影响，上层容差不应小于底层容差。

## 建议：尽量用乘法或其它方法代替除法，特别是浮点运算中的除法。

## 整型类型使用建议

1. 如果表示地址则使用intptr\_t，它表示与程序位数相对应的有符号整形，因此可以兼容32、64位。  
   (2) 如果需要明确区分“是&否”两种情况，则使用bool。  
   (3) 如果需要用有限个数字来表示多个状态，则推荐使用枚举（class enum）。  
   (4) 如果数字可能溢出int的范围，则使用long long。  
   (5) 其余情况推荐使用int  
       对于范围更小的整型，可以节省内存，但是  
       ① 目前在大多数情况下并不是空间占用的瓶颈  
       ② 在数值计算中会转为int进行计算，然后再裁断，降低效率  
       ③ 对于使用者而言可能需要做类型转换，容易出错  
       对于无符号整型，可以表示更大的正数范围，但是  
       ① 目前在大多数情况下int无法满足的值域，unsigned int也无法满足  
       ② 经常会和有符号类型交互或转换而导致错误（例如下标不会小于0而导致死循环）

## 浮点数类型使用建议

推荐使用double  
① float相对于double可以节省内存，但精度不足，且在现代计算机中速度并不如double  
② long double在不同编译器和平台中的情况不一致，不轻易使用

# 性能

## 原则：在保证软件系统正确性、简洁、可维护性及可测性的前提下，提高代码效率。

说明：

* 局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。
* 不应花过多的时间拼命地提高调用不很频繁的函数代码效率。
* 保护性代码可能会降低效率，提高稳定性。需要综合分析。

## 规则：循环体内工作量最小化。

说明：

* 将循环中与循环无关，不是每次循环都要做的操作，移到循环外部执行。
* 减少循环嵌套层次。
* 在多重循环中，应将最忙的循环放在最内层。

## 规则：减少内存拷贝的操作。

说明：

* 注意按值传递和按引用传递的效率。
* 注意复杂数据结构的效率，比如vector的Insert，Delete。

## 规则：使用指针而减少使用数组索引。

使用指针+偏移量去访问数据的好处是直接寻址，而使用数组时每次都要取数组地址，再寻址。

# 日志和异常

## 规则：避免循环输出日志。

说明：

日志应该用来快速定位问题，在日志中应记录有效的信息。不断的循环输出相似的日志会淹没有效信息，并造成性能损失。

## 规则：避免在析构函数中抛出异常。

说明：

* 不要在析构函数里面主动抛出异常。
* 构造函数中尽量不要抛出异常。如果必须，注意避免内存泄露。

## 规则：异常都应该被捕获并处理。

说明：

* 异常都应该被处理，不要忽略异常，不要吞异常。
* 如果捕获异常后再抛出，应该是再封装的异常类型并进行统一处理。

## 建议：制定统一的日志和异常格式。

说明：日志规范参考《[日志规范](https://docs.fscut.com/x/yAvmAQ)》。

# 可测性

## 规则：代码都应该可以进行单元测试。

应确保所有全局函数、类的公有函数可测。类的私有函数不直接测试，而是通过分支覆盖。

## 建议：内嵌测试代码。

说明：

* 复杂流程、算法，应具备预编译开关，在合适的节点通过一些办法输出测试数据，改变代码走向，从而可以对当前代码进行测试。

# 编程风格

## 规则：成对出现的操作需要确保完整执行。

说明：避免因执行流跳转导致的执行不完整。譬如对象创建（Create）后没有释放（Free），会导致内存泄露。锁获取（Lock）后没有释放（Unlock），临界区进入（Enter）后没有退出（Leave），会导致线程死锁。

## 规则：使用断言进行假设检查，不要用断言当作运行时异常处理

说明：即不要用断言检查和外部输入、条件相关的信息。断言应该只在Debug环境下使用，不应该在生产代码中生效。

## 规则：界面输入应当有所限制，如果允许不合法的输入则必须有对应的处理。

# 行为建议

## 规则：不要拷贝代码。

说明：

* 如果遇到逻辑相似的算法应提取出公共函数。
* 如果有2段以上代码很接近。或者是通过Ctrl+C,Ctrl+V出来的，就应该考虑把共同部分抽象出来了。

## 规则：不要重复造轮子。

需要某种算法、数据结构、工具时先看看有没有现成的，可能已有的轮子在你看不到的地方。通常可以通过查询docs文档、询问能力组、项目组来获得相关信息。

## 规则：重视编译器的警告。

高度重视警告：使用编译器的最高警告级别。要求构建是干净利落的（没有警告）。理解所有的警告。通过修改代码而不是降低 警告级别来排除警告。

备注1：VC开W4级别。

## 建议：解BUG要解决问题根源，不能只解决现象。

## 建议：熟知设计模式和对应原理，优先使用常见的标准的设计模式以减少代码阅读难度。

## 建议：降低耦合度。

## 建议：不要无意地编写不通用的代码

说明：编写代码时，尽可能使用最抽象的方法来完成任务。思考哪些操作对其所操作的接口造成的“能力紧张”最小。这种习惯能够使代码 更加通用，从而重用性更高、更能灵活地适应其环境的变化。

# 代码提交

## 规则：提交代码时确保没有编译error和warning

说明：

* 提交代码前必须消除代码中的error和warning。
* 使用Build编译工程来确定是否有warning，因为Compile不会重复提示。

## 规则：提交代码后必须CI编译成功。

说明：提交SVN后必须执行一次CI构建，并确保构建成功。因为模块和合作上的交叉，代码上库后应及早检查代码的可用性。

## 规则：不要提交绝对路径、和私人电脑相关的工程设置。

说明：个人的工程设置不要提交到代码库，以免他人更新代码后冲突。

# 数控系统规范

（待更新）

# 易用性

（以客户为中心）

* 显示时按界面要求进行处理，而内部处理按照标准格式进行处理（单位切换）
* 软件开发中需要合理添加日志（[日志规范](https://docs.fscut.com/pages/viewpage.action?pageId=31853512)），不应将程序搞崩溃或卡死
* 翻译支持
* 避免ANR（GUI线程堵塞问题）