1. 软件编码规范

良好的编码规范是IT从业人员必备的素质，C++(包括C)和python语言是当前比较主流的编程语言。请参照google等编码规范并结合自己的编码习惯，完成C++或python或C#或其它语言的编码规范，以供今后使用。具体要求：

(1) 提交一种编程语言的编码规范文档；

(2) 按照自己的规范提交对应的实例代码，代码行必须50行以上，必须保证规范和实例的统一。

（1）c++程序编码规范

**一、文件排版方面**

1. 包含头文件   
   1.1 先系统头文件，后用户头文件。   
   1.2 系统头文件，稳定的目录结构，应采用包含子路径方式。   
   1.3 自定义头文件，不稳定目录结构，应在dsp中指定包含路径。   
   1.4 系统头文件应用：#include <xxx.h>   
   1.5 自定义同文件应用：#include "xxx.h"   
   1.6 只引用需要的头文件。
2. h和cpp文件   
   2.1 头文件命名为\*.h，内联文件命名为\*.inl；C++文件命名为\*.cpp   
   2.2 文件名用大小写混合，或者小写混合。例如DiyMainView.cpp，infoview.cpp。不要用无意义的名称：例如XImage.cpp；SView.cpp；xlog.cpp；   
   2.3 头文件除了特殊情况，应使用#ifdef控制块。   
   2.4 头文件#endif应采用行尾注释。   
   2.5 头文件，首先是包含代码块，其次是宏定义代码块，然后是全局变量，全局常量，类型定义，类定义，内联部分。   
   2.6 CPP文件，包含指令，宏定义，全局变量，函数定义。
3. 文件结构   
   3.1 文件应包含文件头注释和内容。   
   3.2 函数体类体之间原则上用2个空行，特殊情况下可用一个或者不需要空行。
4. 空行   
   4.1 文件头、控制块，#include部分、宏定义部分、class部分、全局常量部分、全局变量部分、函数和函数之间，用两个空行。

**二、注释方面**

1. 文件头注释   
   1.1 作者，文件名称，文件说明，生成日期(可选)
2. 函数注释   
   2.1 关键函数必须写上注释，说明函数的用途。   
   2.2 特别函数参数，需要说明参数的目的，由谁负责释放等等。   
   2.3 除了特别情况，注释写在代码之前，不要放到代码行之后。   
   2.4 对每个#else或#endif给出行末注释。   
   2.5 关键代码注释，包括但不限于：赋值，函数调用，表达式，分支等等。   
   2.6 善未实现完整的代码，或者需要进一步优化的代码，应加上 // TODO ...   
   2.7 调试的代码，加上注释 // only for DEBUG   
   2.8 需要引起关注的代码，加上注释 // NOTE ...   
   2.9 对于较大的代码块结尾，如for,while,do等，可加上 // end for|while|do

**三、命名方面**

1. 原则   
   1.1 同一性：在编写一个子模块或派生类的时候，要遵循其基类或整体模块的命名风格，保持命名风格在整个模块中的同一性。   
   1.2 标识符组成：标识符采用英文单词或其组合，应当直观且可以拼读，可望文知意，用词应当准确，避免用拼音命名。   
   1.3 最小化长度 && 最大化信息量原则：在保持一个标识符意思明确的同时，应当尽量缩短其长度。   
   1.4 避免过于相似：不要出现仅靠大小写区分的相似的标识符，例如"i"与"I"，"function"与"Function"等等。   
   1.5 避免在不同级别的作用域中重名：程序中不要出现名字完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但容易使人误解。   
   1.6 正确命名具有互斥意义的标识符：用正确的反义词组命名具有互斥意义的标识符，如："nMinValue" 和"nMaxValue"，"GetName()" 和 "SetName()" ….   
   1.7 避免名字中出现数字编号：尽量避免名字中出现数字编号，如Value1,Value2等，除非逻辑上的确需要编号。这是为了防止程序员偷懒，不肯为命名动脑筋而导致产生无意义的名字（因为用数字编号最省事）。
2. T,C,M,R类   
   2.1 T类表示简单数据类型，不对资源拥有控制权，在析构过程中没有释放资源动作。   
   2.2 C表示从CBase继承的类。该类不能从栈上定义变量，只能从堆上创建。   
   2.3 M表示接口类。   
   2.4 R是资源类，通常是系统固有类型。除了特殊情况，不应在开发代码中出现R类型。
3. 函数名   
   3.1 M类的函数名称应采用HandleXXX命名，例如：HandleTimerEvent；不推荐采用java风格，例如handleTimerEvent；除了标准c风格代码，不推荐用下划线，例如，handle\_event。   
   3.2 Leave函数，用后缀L。   
   3.3 Leave函数，且进清除栈，用后缀LC。   
   3.4 Leave函数，且删除对象，用后缀LD。
4. 函数参数   
   4.1 函数参数用a作为前缀。   
   4.2 避免出现和匈牙利混合的命名规则如apBuffer名称。用aBuffer即可。   
   4.3 函数参数比较多时，应考虑用结构代替。   
   4.4 如果不能避免函数参数比较多，应在排版上可考虑每个参数占用一行，参数名竖向对齐。
5. 成员变量   
   5.1 成员变量用m最为前缀。   
   5.2 避免出现和匈牙利混合的命名规则如mpBuffer名称。用mBuffer即可。
6. 局部变量   
   6.1 循环变量和简单变量采用简单小写字符串即可。例如，int i;   
   6.2 指针变量用p打头，例如void\* pBuffer;
7. 全局变量   
   7.1 全局变量用g\_最为前缀。
8. 类名   
   8.1 类和对象名应是名词。   
   8.2 实现行为的类成员函数名应是动词。   
   8.3 类的存取和查询成员函数名应是名词或形容词。
9. 风格兼容性   
   9.1 对于移植的或者开源的代码，可以沿用原有风格，不用C++的命名规范。

**四、代码风格方面**

1. Tab和空格   
   1.1 每一行开始处的缩进只能用Tab，不能用空格，输入内容之后统一用空格。除了最开始的缩进控制用Tab，其他部分为了对齐，需要使用空格进行缩进。这样可以避免在不同的编辑器下显示不对齐的情况。   
   1.2 在代码行的结尾部分不能出现多余的空格。   
   1.3 不要在"::","->","."前后加空格。   
   1.4 不要在"，","；"之前加空格。
2. 类型定义和{   
   2.1 类，结构，枚举，联合：大括号另起一行
3. 函数   
   3.1 函数体的{需要新起一行，在{之前不能有缩进。   
   3.2 除了特别情况，函数体内不能出现两个空行。   
   3.3 除了特别情况，函数体内不能宏定义指令。   
   3.4 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。   
   3.5 在头文件定义的inline函数，函数之间可以不用空行，推荐用一个空行。
4. 代码块   
   4.1 "if"、"for"、"while"、"do"、"try"、"catch" 等语句自占一行，执行语句不得紧跟其后。不论执行语句有多少都要加 "{ }" 。这样可以防止书写和修改代码时出现失误。   
   4.2 "if"、"for"、"while"、"do"、"try"、"catch" 的括号和表达式，括号可紧挨关键字，这样强调的是表达式。
5. else   
   5.1 if语句如果有else语句，用 } else { 编写为一行，不推荐用 3 行代码的方式。
6. 代码行   
   6.1 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。   
   6.2 多行变量定义，为了追求代码排版美观，可将变量竖向对齐。   
   6.3 代码行最大长度宜控制在一定个字符以内，能在当前屏幕内全部可见为宜。
7. switch语句   
   7.1 case关键字应和switch对齐。   
   7.2 case子语句如果有变量，应用{}包含起来。   
   7.3 如果有并列的类似的简单case语句，可考虑将case代码块写为一行代码。   
   7.4 简单的case之间可不用空行，复杂的case之间应考虑用空行分割开。   
   7.5 case字语句的大括号另起一行，不要和case写到一行。   
   7.6 为所有switch语句提供default分支。   
   7.7 若某个case不需要break一定要加注释声明。
8. 循环   
   8.1 空循环可用 for( ;; ) 或者 while( 1 ) 或者 while( true )
9. 类   
   9.1 类继承应采用每个基类占据一行的方式。   
   9.2 单继承可将基类放在类定义的同一行。如果用多行，则应用Tab缩进。   
   9.3 多继承在基类比较多的情况下，应将基类分行，并采用Tab缩进对齐。   
   9.4 重载基类虚函数，应在该组虚函数前写注释 // implement XXX   
   9.5 友元声明放到类的末尾。
10. 宏   
    10.1 不要用分号结束宏定义。   
    10.2 函数宏的每个参数都要括起来。   
    10.3 不带参数的宏函数也要定义成函数形式。
11. goto   
    11.1 尽量不要用goto。

**五、类型**

1. 定义指针和引用时\*和&紧跟类型。
2. 尽量避免使用浮点数，除非必须。
3. 用typedef简化程序中的复杂语法。
4. 避免定义无名称的类型。例如：typedef enum { EIdle, EActive } TState;
5. 少用union，如果一定要用，则采用简单数据类型成员。
6. 用enum取代(一组相关的)常量。
7. 不要使用魔鬼数字。
8. 尽量用引用取代指针。
9. 定义变量完成后立即初始化，勿等到使用时才进行。
10. 如果有更优雅的解决方案，不要使用强制类型转换。

**六、表达式**

1. 避免在表达式中用赋值语句。
2. 避免对浮点类型做等于或不等于判断。
3. 不能将枚举类型进行运算后再赋给枚举变量。
4. 在循环过程中不要修改循环计数器。
5. 检测空指针，用 if( p )
6. 检测非空指针，用 if( ! p )

**七、函数**

1. 引用   
   1.1 引用类型作为返回值：函数必须返回一个存在的对象。   
   1.2 引用类型作为参数：调用者必须传递一个存在的对象。
2. 常量成员函数   
   2.1 表示该函数只读取对象的内容，不会对对象进行修改。
3. 返回值   
   3.1 除开void函数，构造函数，析构函数，其它函数必须要有返回值。   
   3.2 当函数返回引用或指针时，用文字描述其有效期。
4. 内联函数   
   4.1 内联函数应将函数体放到类体外。   
   4.2 只有简单的函数才有必要设计为内联函数，复杂业务逻辑的函数不要这么做。   
   4.3 虚函数不要设计为内联函数。
5. 函数参数   
   5.1 只读取该参数的内容，不对其内容做修改，用常量引用。   
   5.2 修改参数内容，或需要通过参数返回，用非常量应用。   
   5.3 简单数据类型用传值方式。   
   5.4 复杂数据类型用引用或指针方式。

**八、类**

1. 构造函数   
   1.1 构造函数的初始化列表，应和类的顺序一致。   
   1.2 初始化列表中的每个项，应独占一行。   
   1.3 避免出现用一个成员初始化另一个成员。   
   1.4 构造函数应初始化所有成员，尤其是指针。   
   1.5 不要在构造函数和析构函数中抛出异常。
2. 纯虚函数   
   2.1 M类的虚函数应设计为纯虚函数。
3. 构造和析构函数   
   3.1 如果类可以继承，则应将类析构函数设计为虚函数。   
   3.2 如果类不允许继承，则应将类析构函数设计为非虚函数。   
   3.3 如果类不能被复制，则应将拷贝构造函数和赋值运算符设计为私有的。   
   3.4 如果为类设计了构造函数，则应有析构函数。
4. 成员变量   
   4.1 尽量避免使用mutable和volatile。   
   4.2 尽量避免使用公有成员变量。
5. 成员函数   
   5.1 努力使类的接口少而完备。   
   5.2 尽量使用常成员函数代替非常成员函数，const函数   
   5.3 除非特别理由，绝不要重新定义(继承来的)非虚函数。（这样是覆盖，基类的某些属性无初始化）
6. 继承   
   6.1 继承必须满足IS-A的关系，HAS-A应采用包含。   
   6.2 虚函数不要采用默认参数。   
   6.3 除非特别需要，应避免设计大而全的虚函数，虚函数功能要单一。   
   6.4 除非特别需要，避免将基类强制转换成派生类。
7. 友元   
   7.1 尽量避免使用友元函数和友元类。

**九、错误处理**

1. 申请内存用new操作符。
2. 释放内存用delete操作符。
3. new和delete，new[]和delete[]成对使用。
4. 申请内存完成之后，要检测指针是否申请成功，处理申请失败的情况。
5. 谁申请谁释放。优先级：函数层面，类层面，模块层面。
6. 释放内存完成后将指针赋空，避免出现野指针。
7. 使用指针前进行判断合法性，应考虑到为空的情况的处理。
8. 使用数组时，应先判断索引的有效性，处理无效的索引的情况。
9. 代码不能出现编译警告。
10. 使用错误传递的错误处理思想。
11. 卫句风格：先处理所有可能发生错误的情况，再处理正常情况。
12. 嵌套do-while(0)宏：目的是将一组语句变成一个语句，避免被其他if等中断。

**十、性能**

1. 使用前向声明代替#include指令。Class M;
2. 尽量用++i代替i++。即用前缀代替后缀运算。
3. 尽量在for循环之前，先写计算估值表达式。
4. 尽量避免在循环体内部定义对象。
5. 避免对象拷贝，尤其是代价很高的对象拷贝。
6. 避免生成临时对象，尤其是大的临时对象。
7. 注意大尺寸对象数组。
8. 80-20原则。

**十一、兼容性**

1. 遵守ANSI C和ISO C++国际标准。
2. 确保类型转换不会丢失信息。
3. 注意双字节字符的兼容性。
4. 注意运算溢出问题。
5. 不要假设类型的存储尺寸。
6. 不要假设表达式的运算顺序。
7. 不要假设函数参数的计算顺序。
8. 不要假设不同源文件中静态或全局变量的初始化顺序。
9. 不要依赖编译器基于实现、未明确或未定义的功能。
10. 将所有#include的文件名视为大小写敏感。
11. 避免使用全局变量、静态变量、函数静态变量、类静态变量。在使用静态库，动态库，多线程环境时，会导致兼容性问题。
12. 不要重新实现标准库函数,如STL已经存在的。
13. 按照自己的规范提交对应的实例代码，代码行必须50行以上，必须保证规范和实例的统一。

#include <iostream.h>  
  
const float PI=3.1416;   //声明常量(只读变量)PI3.1416  
  
float fCir\_L(float);     //声明自定义函数fCir\_L()的原型  
  
float fCir\_S(float);     //声明自定义函数fCir\_S()的原型   
  
//以下是main()函数  
  
main()  
  
{  
  
float r,l,s;       //声明3个变量  
  
cout<<"R=";        //显示字符串  
  
cin>>r;          //键盘输入  
  
l=fCir\_L(r);       //计算圆的周长，赋值给变量l  
  
s=fCir\_S(r);       //计算圆的面积，赋值给变量s  
  
cout<<"l="<<l;      //显示计算结果  
  
cout<<"\ns="<<s;                 
  
}   
  
//定义计算圆的周长的函数fCir\_L()  
  
float fCir\_L(float x)  
  
{  
  
float z=-1.0;       //声明局部变量  
  
if (x>=0.0)        //如果参数大于0，则计算圆的周长  
  
z=2\*PI\*x;  
  
return(z);        //返回函数值  
  
}  
  
//定义计算圆的面积的函数fCir\_S()  
  
float fCir\_S(float x)  
  
{  
  
float z=-1.0;        //声明局部变量  
  
if (x>=0.0)         //如果参数大于0，则计算圆的面积  
  
z=PI\*x\*x;  
  
return(z);          //返回函数值  
  
}

#include <iostream.h>    //包含iostream.h头文件  
  
void main()  
  
{  
  
//输出字符常量、变量和字符串  
  
char c1='A';  
  
cout<<'W';  
  
cout<<c1<<endl;  
  
cout<<"This is a test."<<endl;  
  
cout<<"------------------"<<endl;  
//输出整型常量、变量和表达式  
  
int n=100;  
  
cout<<10;  
  
cout<<n;  
  
cout<<2\*n<<endl;  //输出整型表达式  
  
cout<<"------------------"<<endl;  
  
//输出浮点型常量、变量和表达式  
  
double pi=3.1415926,r=10.0,s=pi\*r\*r;  
  
cout<<pi<<endl;  
  
cout<<r;  
  
cout<<s;  
  
cout<<2\*r\*pi<<endl;      //输出浮点型表达式  
  
cout<<"------------------"<<endl;  
  
//一个cout可以输出多项数据  
  
cout<<'W'<<" "<<c1<<endl;  
  
cout<<"This is a test."<<endl;  
  
cout<<"pi="<<pi<<" r="<<r<<" s="<<s<<endl;  
  
}