亿绪组织编程建议格式标准（C++）

宣言：

本格式标准结合当前亿绪组织已经开发的项目所获得的经验进行编写。仅供亿绪组织内部使用。本标准不一定与其他标准一致或近似，可能有较大差别。

1. 命名

(1) 无论如何，命名中不得出现任何汉语拼音首字母缩写。除了类的 **private** 部分、孤立的简单函数、临时变量等并不涉及到太大使用范围的命名可以使用汉语拼音全拼命名之外，其他命名必须使用英文进行命名。不得将中英文混用。不要轻易将一个英文单词做跳字母缩写，例如把 **advertise** 缩写成 **adi**。若需要做跳字母缩写，则必须是首字母+后续各音节中取出第一个辅音字母（但不包括进音 **r**、边音 **l**、半元音 **j**、**w**）作为缩写，例如 **advertise** 缩写成 **advts**，**Threshold** 缩写成 **trshd**。

(2) 规定几种命名称谓：

① 小驼峰：以若干英文单词或汉语拼音音节连贯为一个名称，但第一词首字母小写，后续词首字母大写，如 **setStyleSheet**。

② 大驼峰：以若干英文单词或汉语拼音音节连贯为一个名称，所有首字母均大写，如 **ObjectName**。

③ 全小写，全大写：以若干英文单词或汉语拼音音节连贯为一个名称，所有字母均为小写或大写。如 **xinjian**，**GUANBI**。一般情况下不使用，除非是一个英文单词作为名称，或者 **API** 要求全大写。

④ 下划分割：以若干英文单词或汉语拼音音节连贯为一个名称，词间用下划线分割。如 **EMIT_CURRENT_STATUS**。分割法不对大小写做出要求。

⑤ 简单命名：对于循环判据用的变量或循环体中间临时变量，可以在不引起歧义的情况下使用少于三个字母的简单名称进行命名。

(3) 若一个单词\音节的末字母与下一起始字母一致，则除非是 **API** 要求的全大写情况之外，强制使用驼峰法或分割法进行命名。例如，**addnewwidget** 应该被换为 **addNewWidget** 或者 **add_new_widget**。

(4) 对于循环判据用的变量或循环体中间临时变量，可以在不引起歧义的情况下使用少于三个字母的简单名称进行命名。例如：

```
QStringList EgList;
...
for ( int i = 0 ; i < EgList.length() [1]; i++){
    qDebug() << EgList[i];
}
```

此时 **i** 是循环判据，不必要使用正式命名。

(5) 针对单个字母的简单命名约定：

① 数字，主要使用 **i**、**j**、**k**、**m**、**n** 进行命名。

② 时间，主要使用 **x,t** 进行命名.

③ 文字，主要使用 **a**、**b**、**c**、**d** 进行命名

④ 文件\路径，主要使用 **f**、**p** 进行命名

[1] 在追求性能的情况下，此处应当考虑使用迭代器而非下标值自增。

⑤ 无论如何不允许单独出现字母 l、o

(6) 针对 API 进行二次扩展而出现的命名需求。
扩展时产生的扩展内部命名按本节要求进行命名。产生的扩展 API 命名按照原 API 命名风格命名。例如，针对 Qt 的 QWidget 进行扩展的类的名称可以按大驼峰命名法被命名为 YWidget，YExpandWidget，但不能命名成 yWidget，ywidget，y_widget 等。

(7) 函数主要使用小驼峰命名法。

(8) 变量主要使用大驼峰命名法。对于可能引起歧义的两个作用域不同的变量，应对作用域最大的变量使用以 g 开头的小驼峰命名，如 gFileName；

(9) 类、结构、命名空间、枚举类均允许使用大驼峰或小驼峰。

(10) 新 API 命名。对于暴露给用户的命名，对于类、结构、命名空间、枚举类，均使用全大写或者带下划线的全大写命名。例如 EXPANDIMAGE，EXPAND_IMAGE，对于函数、变量等其他内容，使用小驼峰命名，如 loadImage。

(11) 对于类内的对象，建议命名为对象类型+用途或用途+类型：

① 当某一用途有多个对象的时候，采用用途+类型的方式，例如某一线程数选择列表和其对应的说明标签，可以命名为：

```
QLabel* ThreadLabel;
QComboBox* ThreadBox;
```

② 当某类型被用在多处时，采用类型+用途的方式，例如上一帧和下一帧按钮：

```
QPushButton* ButtonNextFrame;
QPushButton* ButtonBackFrame;
```

③ 当某类型被用在对于同一对象的多种操作时，例如对于“文件”菜单中常见的按钮，也可以采用一种接近自然语言的命名方式（但不推荐）：

```
QAction* NewFile;
QAction* OpenFile;
QAction* SaveFile;
QAction* SaveFileAs;
QAction* CloseFile;
QAction* ExitProgram;
```

(12)

2. 缩进、花括号、换行

(1) 代码块要比起始行的位置再向后四个空格或一个缩进符号。右花括号则与起始行重新对齐。
例如：

```
void eg(void){
    if (...){
        ...
        ...
        if (...){
            ...
            ...
        }
    }
```

```
    else{
        ...
    }
}
```

对于 **if-else** 结构，有的编译器会在键入 **else** 和其左花括号后，自动把 **else** 和其左花括号挪到下一行并与上面的右花括号对齐（例如上）。这样也符合本规范的要求，但实际上会增加 **if-else** 的割裂感。建议将 **else** 和其左花括号挪回，例如：

```
if (...){
    ...
}else{
    ...
}
```

(2) 函数体、**if**、**for**、**while**、**switch** 的左花括号与参数的右圆括号处于同一行。

例如：

```
int main(void)[2]{
    return 0;
}
```

再例如：

```
A(int a, int b):
    x(a),y(b){
    int c = x + y;
}
```

对于类的构造函数这种快速初始化变量的写法，应当在冒号后换行填写快速初始化内容，并在该行结尾键入左括号。

(3) **do**、类、结构、枚举类、命名空间的左花括号在其下一行。

例如：

```
class A :public B
{
    ...
};
```

(4) 对于函数参数超级长，长到一般人显示器和字号不足以完整显示这一行的情况下，在参数到达屏幕边缘之前换行。

例如：

```
void imageAnalysis(QString ImageName, QString ImageRawDir,
       cv::Mat InputImage, cv::Mat OutputImage,
       int Threshold, QString* AnalysisInfo,
       QString OutputFolder);
```

如上例，换行之后的参数要和起始行的左圆括号位置基本对齐或至少自函数名称开头位置起再空出四格，不得完全顶头书写。

*更不要在换行时把类型和参数名分立在两行。~~否则拉去做一篇完形填空，文章印在正面，选项印在反面。~~

[2] 非成员函数无参时请填写 **void**，建议不要省略 **void**。此建议是考虑到对于 C 语言而言，空括号代表参数不定，而不是无参数。

(5) 对于文本超级长，或文本内嵌入其他脚本的情况，依据文本内容或脚本语法进行换行。[3]

例如：

```
ManageButton->setStyleSheet("\
    QPushButton{\
        border:2px solid #1473E6;\
        border-radius:10px;\
        background-color:#555555;\
        font-size:"+
            QString::number((int)(height() * 0.015)) + "px;\
        color:#FFFFFF;\
        }");
```

忠告：除非不维护，否则不要怕麻烦。

3. 建议语法：

(1) 不得用 **goto**。只有一种情况允许使用 **goto**，那就是在程序异常后的崩溃前兜底（例如 **catch**）中为了快速跳出各级代码块而使用，并且使用目的仅限于回到位于 **main** 的 **catch** 或 **main** 下最多两级的异常信息收集处理块内。若使用了 **goto** 做这种异常后跳转，则程序必须在处理异常信息后崩溃，不得尝试继续运行。
简而言之，确保 **goto** 只被用于快速结束正在崩溃的程序这一个需求上。

(2) 当函数声明其参数为不确定个数的某类型的参数时，应当声明参数为该类型的某容器，例如 `string joinString(list<string>)`，不要使用`...`，例如 `string joinString(...)`。

(3) 不要把类的非静态成员（函数）原型和定义分立在`.h` 和`.cpp` 两个文件内编写。除非对于 **inline** 有特殊需求。

(4) 所有 C++标准语法中允许省略花括号的情况均不建议使用。

(5) 不建议在 **for**、**while**、**if** 等的条件区域键入操作性语句。

(6) 只允许使用 `using namespace std`，对于其他 **namespace**，无论名称有多长，使用次数有多繁复，除非该文件内除了 **std** 之外只有该 **namespace**，否则均不使用 **using**。

(7) 定义模板时，使用 **typename** 而非 **class**。

(8) 类内使用该类的成员函数时，不要省略“**this->**”。

(9) 由于大部分编译器并不支持预编译指令的自动格式，故建议手动采用 **Python** 的缩进法书写预编译指令。（可以参考 **CommonEdit** 库中 **CEMacro** 和 **CE_QtMacro**）

4. 对于无副本的文件的不重复编译保证，若编译器支持`#pragma once`，且没有跨平台、跨编译器的需求，则不要用`#ifndef-#define-#endif`。对于有副本的文件的不重复编译保证仍然使用`#ifndef-#define-#endif`。

5. 源码编码仅限于 UTF-8\UTF-8(BOM)、GBK、GB2312。

6. 当使用了跨 C++与 Python 的某库时（如 Qt、OpenCV 等），以下宏被视作通用宏（已经包括在 **CommonEdit** 库中）：

```
#define self this
```

---

[3] 由于这本质上是在给字符串内平白无故增加各种空白符，所以如若字符串内多出这些空白符会引起错误，则不再要求字符串的格式。

```
#define elif else if
#define def void
#define False false
#define True true
#define match switch
#define raise throw
```

7. 当存在跨平台（跨编译器）需求时，要求定义一个申明平台（编译器）的宏，并且根据该宏定义为一个名为 DEPLOY 的宏赋值：

平台（编译器）	宏	宏 DEPLOY 的值
Windows(MSVC2022)	WINDOWS_DEPLOY	1
Android(arm64_v8a)	ANDROID_DEPLOY	2
Windows(MSVC2019)	WIN_MSVC2019	3
Windows(MinGW)	WIN_MinGW	4
Android(armv7)	AND_ARMV7	5
Linux	LINUX_DEPLOY	6
IOS	IOS_DEPLOY	7
MacOS	MAC_DEPLOY	8

对于其他未尽平台或编译器，请使用大于 100 的 DEPLOY 值。

8. 除非不方便采用枚举类（例如枚举态无法预知，或者从 Python 等无枚举[4]语言迁移时），否则对于单一目标多状态判断使用枚举和 switch 进行。

9. 除非在一行内声明多个指针，否则星号应当前置。例如：

```
QAction* NewFile;
QAction* OpenFile;
QAction* SaveFile;
QAction* SaveFileAs;
QAction* CloseFile;
QAction* ExitProgram;
```

对于此条，建议将其理解为，把 QAction*整体看做一种类型，而不是把*看做 NewFile 的一种修饰（虽然实际如此）。

10. 对于枚举类，若期望枚举值超过三个，则额外添加 Unknown = 0。

11. 仅将类型转换用于提升数字（int 换 float），指针从基类转换到派生类。不要做其他额外的类型转换。

12. 当某个函数的功能是先执行某些简短语句，再执行另一函数，或先执行另一函数，再执行某些简短语句，即该函数是另外一个函数增加少量前后置的套壳函数时，应当给套壳函数取一个与被套壳函数一样但依据单下划线告知套壳位置的名称，例如，对于类的某成员函数，若有：

```
class A {
public:
    A(void);
    void func_1(void) {
```

[4] Python 实际上早在 3.4 版本通过官方库 enum.py 实现了枚举类，但考虑到 enum 对于 Python 来说是一个库特性，而非固有特性（例如对于 C++而言 enum 是关键字），并且其对应的匹配语句 match-case 在 3.10 版本才开始加入 Python，因此目前仍然认为 Python 是无枚举的（至少未来几年之内如此）。

```
        cout << "F1" << endl;
    }
    void func_2(void) {
        func_1();
        cout << "F2" << endl;
    }
    void func_3(void) {
        cout << "F3" << endl;
        func_1();
    }
};
```

则根据本条规则，应当改为：

```
class A {
public:
    A(void);
    void func_1(void) {
        cout << "F1" << endl;
    }
    void func_1_(void) {
        func_1();
        cout << "F2" << endl;
    }
    void _func_1(void) {
        cout << "F3" << endl;
        func_1();
    }
};
```

下划线[5]清晰的标明了，该函数本质上是某函数的前、后套壳。

[5] 仅单下划线。由于在 Python 中部分较为特殊的函数、private 内容会使用双下划线，因此在 C++中也建议不要使用双下划线。