物联网实验室 C++ 代码风格指南

项目结构规范

项目名称

项目名称应该使用大驼峰命名法命名，且尽量不要使用单词缩写。

项目结构

项目结构应该按照如下所示的结构组织：

```
<PROJECT>
├── build                       // 编译输出目录
├── include                     // 头文件(.h)
│   └── <Abbr-of-Project>       // 项目名称缩写
│       └── ...
├── src                         // 源文件(.cpp)
│   └── ...
├── CMakeLists.txt              // CMake 配置文件
└── README.md                   // 项目自述文件
```

注意，include里嵌套的文件夹名称应该为项目名称缩写，且需与项目的顶级命名空间名称一致。

自述文件规范

项目自述文件并无强制要求，但是应该尽可能详细地罗列项目的依赖库（名称、版本、下载地址）。

命名空间规范

命名空间名称

命名空间名称应该全部小写，不应该包含下划线。

项目的所有代码必须放在一个顶级命名空间中，这个顶级命名空间的名称应该为项目名称或者项目名称的缩写。

例如，项目 EmergencyTeleoperatedRobotSystem（应急遥操作机器人系统）的顶级命名空间应该为项目名称缩写 `etrs`。

命名空间嵌套

命名空间嵌套应以如下所示的一行代码的形式，不要使用多行代码的形式。

```
namespace etrs::utility {
    ...
}
```

命名空间别名

命名空间别名应该使用 `namespace` 关键字，不要使用 `typedef`。

举例：

```
namespace util = etrs::utility;      // 好
typedef etrs::utility util;          // 差
```

有关命名空间的格式规范请参考命名空间格式。

命名规范

文件夹名规范

文件夹名称应该全部小写，用下划线(`_`)连接两个单词。

正确的命名：

- `utility`
- `object_detection`

错误的命名：

- `PythonInvoker`
- `pythonInvoker`
- `pythoninvoker`

文件名规范

头文件(`.h`)和源文件(`.cpp`)名称以大驼峰命名法命名，不包含下划线。

正确的命名：

- `Log.h`
- `MathUtils.h`

错误的命名：

- `log.h`
- `math_utils.h`
- `mathUtils.h`

普通变量名规范

普通变量名应该全部小写，用下划线(`_`)连接两个单词。

不要以不明所以的缩写命名变量，除非该缩写是众所周知的，例如`num` 表示 `number`、`img` 表示 `image`、`i` 表示 `index`。

不要以下划线(`_`)开头命名变量，因为以下划线开头的变量名是给编译器，给系统库，给未来语言升级保留的。

不要以过长且不用下划线隔开的单词命名变量，例如 `superlongvariablename`。

正确的命名：

- `table_name`
- `pcd`
- `age`

错误的命名：

- `tableName_`
- `TableName`
- `legacyazurekinectpointcloud`
- `a`

只有在变量名的声明周期很短的情况下，可以考虑使用缩写命名变量。

类成员变量名规范

对于类成员变量，同样遵循普通变量名规范。

举例：

```
class ObjectDetector {
private:
    int time;

protected:
    int count;

public:
    int num;
};
```

普通函数名规范

普通函数名应该以小驼峰命名法命名。且以动词开头，动词应该使用现在时或者过去时。

对于特定的全大写缩写的单词，应该仅将首字母大写，其他字母小写。例如 `MAC` 应该命名为 `Mac`， `URL` 应该命名为 `Url`。

正确的命名：

- `computeDistance()`
- `getObjcetName()`
- `getMacAddress()`

错误的命名：

- `compute_distance()`
- `ComputeDistance()`
- `getMACAddress()`

如果是返回布尔值的函数，应该以 `is` 或者 `has` 开头。

举例：

```
bool isRunning() {
    ...
}

bool hasObject() {
    ...
}
```

类成员函数名规范

对于类成员函数，静态(`static`)成员函数名应该以大驼峰命名法命名，普通成员函数则遵循普通函数名规范。

举例：

```
namespace bt {
    void writeLog();         // 普通函数 - 小驼峰命名法

    class BluetoothDevice {
    public:
        void connectDevice();           // 成员函数 - 小驼峰命名法
        static void GetMacAddress();    // 静态成员函数 - 大驼峰命名法
    };
}
```

类名规范

类名应该以大驼峰命名法命名。

正确的命名：

- `TypeConverter`
- `Record`

错误的命名：

- `typeConverter`
- `Type_Converter`
- `RECORD`

枚举名规范

枚举名应该以大驼峰命名法命名。枚举值应该全部大写，用下划线(`_`)连接两个单词。

举例：

```
enum LogLevel {      // 普通枚举
    DEBUG,
    INFO,
    WARNING,
    ERROR
};

enum class MessageType {     // C++11 引入的枚举类
    POINT_CLOUD,
    IMAGE,
    TEXT
};
```

常量名规范

常量名遵循普通变量名规范，且以 `k_` 开头。

举例：

```
const double k_pi = 3.1415926;
const int k_max = 100;
```

宏名规范

在 C++ 中不推荐使用宏，宏展开可以使用内联函数代替，宏常量可以使用 `const` 或者 `constexpr` 代替。

举例：

```
#define PI 3.1415926     // 不推荐
const double k_pi = 3.1415926;     // 推荐

#define MAX(a, b) ((a) > (b) ? (a) : (b))     // 不推荐
inline int max(int a, int b) { return a > b ? a : b; }     // 推荐
```

若非要使用，则遵循以下规范，宏名应该全部大写，用下划线(`_`)连接两个单词。

举例：

```
#define PI 3.1415926
#define DEFAULT_CHARSET "utf-8"
```

宏的注意事项：

- 当不再使用宏时，必须使用 `#undef` 将其取消定义，否则可能造成命名冲突。
- 不应该在头文件中定义宏。

模板变量名规范

模板变量名推荐使用单个大写字母 `T`、`U`、`V` 等表示。

可变参数模板变量名使用大驼峰命名 `Args` 或 `Params` 表示。

举例：

```
template <typename T, typename... Args>
static void CoutSuccess(T t, Args... args) {
    ...
}
```

类规范

成员声明顺序

类的成员声明顺序应该先声明成员变量，再声明成员函数，且成员的访问权限按照 `private` -> `protected` -> `public` 的顺序。

不推荐使用非私有成员变量，建议将所有成员变量都声明为私有的，通过公有成员函数来访问。

即按照如下顺序排列：

1. 私有成员变量
2. 保护成员变量(不推荐)
3. 公有成员变量(不推荐)
4. 私有成员函数
5. 保护成员函数
6. 公有成员函数

即使类中默认的访问权限是私有的，但也要显式写出 `private` 关键字。

静态成员(`static`)放在普通成员之后。

举例：

```
class Line3D {
private:
    double x;
    double y;
    double z;

protected:
    double length;

public:
    int size;
    static int count;

private:
    void print() {
```

```
        std::cout << "Line3D" << std::endl;
    }

public:
    Line3D(): x(0), y(0), z(0) {
        ++count;
    }
}
```

类成员变量规范参考普通成员变量规范

构造函数

为了保证代码的可读性与一致性，有参构造函数必须用 `explicit` 关键字修饰，禁止使用隐式转换。

举例：

```
class Config {
public:
    explicit Config(std::string path) {
        ...
    }
};
```

析构函数

析构函数应该是虚函数(`virtual`)，除非该类不会被继承。

拷贝构造函数与拷贝赋值运算符

当类需要定义为可拷贝时，应该显式地定义拷贝构造函数。

虽然不显式定义拷贝构造函数，C++ 也会提供一个默认的拷贝构造函数，进行成员变量之间的拷贝。
但是不显式定义拷贝构造函数读者会不知道该类是否支持拷贝，因此应该显式定义拷贝构造函数。提高代码的可读性。

举例：

```
class PointCloud{
public:
    PointCloud(const PointCloud& point_cloud) {
        ...
    }

    PointCloud& operator=(const PointCloud& point_cloud) {
        ...
    }
};
```

当你的类不需要拷贝构造函数时，应该将其声明为 `delete`，禁用拷贝构造函数。

举例：

```
class Timer{
public:
    Timer(const Timer&) = delete;
    Timer& operator=(const Timer&) = delete;
};
```

移动构造函数与移动赋值运算符

移动构造函数与移动赋值运算符的规范和拷贝构造函数与拷贝赋值运算符的规范一样，需要移动构造函数时应该显式定义，不需要时应该将其声明为 `delete`。

举例：

```
class Mesh {
public:
    Mesh(Mesh&& mesh) {
        ...
    }

    Mesh& operator=(Mesh&& mesh) {
        ...
    }
};

class Communicator {
public:
    Communicator(Communicator&&) = delete
    Communicator& operator=(Communicator&&) = delete;
};
```

类与结构体

当结构体或者类只包含公有成员变量且不包含任何函数时，应该使用结构体(`struct`)，其他情况一律使用类(`class`)。

对于结构体也需要显式地写出默认访问权限 `public` 关键词。

举例：

```
struct Point{
public:
    double x;
    double y;
    double z;
};

class PointCloud {
private:
    std::vector<Point> points;
```

```
public:
    PointCloud() {
        ...
    }
};
```

重载运算符

不要重载 `&&`、`||`、`,`、`&` 和 `""` 运算符。

只有在以下情况下，才考虑重载运算符：

- 只有在你的类有需要时
- 该运算符的含义与 C++ 原生运算符的含义相同
- 该运算符可以被读者理解

函数规范

函数复杂度

避免编写过长的函数，函数的长度应该尽可能短小，且每个函数只做一件事情。

当你的函数过长时，应该考虑将其拆分为多个函数。使代码更加清晰可读。

引用参数

所有引用传递的参数都应该声明为 `const`，除非你需要在函数内部修改该参数。`string` 也应该声明为 `const` 引用。

举例：

```
using namespace std;
void print(const string& str);
```

缺省参数

优先考虑使用缺省参数，而不是函数重载。

举例：

```
void writeLog(const string& str, int level = 0);    // 好

void writeLog(const string& str);                    // 差
void writeLog(const string& str, int level);         // 差
```

其他规范

auto 关键字

可以使用 `auto` 关键字来声明变量，但是不要滥用。众所周知，C++ 代码有时候真的是又臭又长，使用 `auto` 关键字可以减少代码量，提高代码的可读性。

`auto` 关键字可以用于以下情况：

- 迭代器
- 复杂返回值类型
- 依赖于模板参数类型的类型

举例：

```
for (auto it = vec.begin(); it != vec.end(); ++it) {    // 好
    ...
}
for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {    // 差

}

auto point_cloud_ptr = std::make_shared<PointCloud>();  // 好
std::shared_ptr<PointCloud> point_cloud_ptr = std::make_shared<PointCloud>();  // 差

template <typename T>
void print(T t) {
    auto value = t.getValue();    // 好
    ...
}
```

输入输出

推荐使用 C 标准库的 `printf` 和 `scanf`。

相比于 C++ 标准库的 `cout`，`printf` 是线程安全的，在多线程环境下，`printf` 不会出现输出错乱的情况。

模板类型

模板类型声明一律使用 `typename`，不要使用 `class`。

举例：

```
template <typename T>       // 好
T max(T a, T b) {
    return a > b ? a : b;
}

template <class T>          // 差
T max(T a, T b) {
    return a > b ? a : b;
}
```

注意模板变量名规范。

模板函数显式实例化

当模板函数或者模板类只有少数几种类型的模板参数时，应该在 `.cpp` 文件中显式实例化模板，将模板函数的声明与实现分开，而不是将模板定义放在头文件中。

且模板函数显式实例化时，应该省略函数参数名。

举例：

```
// .h 文件

template <typename T>
int findElement(const std::vector<T>& vec, T value);         //模板函数声明

// .cpp 文件

template <typename T>
int findElement(const std::vector<T>& vec, T value) {        //模板函数定义
    ...
}

template int findElement(const std::vector<int>&, int);      //省略函数参数名
```

元组

不推荐使用元组(`tuple`)去表示较长的数据结构，应该使用结构体(`struct`)或者类(`class`)。

只有在元组的元素数量较少，且元组生命周期较短时，可以考虑使用元组。

#include 规范

尖括号与双引号

C/C++ 中`#include`使用尖括号(`<>`)和双引号(`""`)包含头文件的区别在于：

- 尖括号(`<>`)会先在系统目录中查找头文件。
- 双引号(`""`)会先在当前工作目录中查找头文件。

因此，除了系统或者 C/C++ 标准库提供的头文件应使用尖括号(`<>`)包含外，其他头文件都应该使用双引号(`""`)包含。

#include 顺序

`#include` 应该按照如下顺序排列：

1. 本文件的头文件（仅对于 `.cpp` 文件）
2. C 标准库头文件
3. C++ 标准库头文件
4. 依赖库头文件
5. 本项目内头文件

对于第一项，如果是 `.cpp` 文件，则应该最优先包含本文件的头文件，如果是 `.h` 文件，则忽略第一项。

用一个空行隔开不同类型的头文件。

举例：

```
#include "etrs/geometry/PointCloud.h"

#include <sys/types.h>

#include <iostream>
#include <vector>

#include "k4a/k4a.h"
#include "open3d/Open3D.h"

#include "etrs/utility/Log.h"
```

头文件规范

头文件后缀

头文件的后缀一律使用 `.h`，不要使用 `.hpp`。

#define 保护

所有头文件都应该有 `#define` 保护来防止头文件被多重包含，命名格式当是: `<PROJECT>_<PATH>_<FILE>_H_`.

例如，项目 `ETRS` 中的头文件 `ETRS/include/utility/Log.h` 可按如下方式保护:

```
#ifndef ETRS_UTILITY_LOG_H_
#define ETRS_UTILITY_LOG_H_
...
#endif // ETRS_UTILITY_LOG_H_
```

注释规范

注释语言

使用母语！母语！母语！说中文！

注释风格

行注释使用 `//`， 块注释使用 `/* ... */`。

`//` 和注释内容之间应该有一个空格。

块注释以 `/*` 开头，以 `*/` 结尾，且开头和结尾都应该单独占一行，每行注释前面应该有一个 `*`，后面跟一个空格。

举例：

```
// 这是一个正确的行注释

/*
 * 这是一个正确的块注释
 */

/* 这个
块注释是错误的，
难看！ */
```

函数注释

函数注释应该放在函数声明上方，用块注释，但是以 `/**` 开头。

注意，函数注释应该写在函数声明上方，而不是函数定义上方，即函数注释应该写在 `.h` 文件中，而不是 `.cpp` 文件中。

函数注释需包含以下内容：

- @brief 函数功能描述
- @param 参数描述
- @return 返回值描述

还可以包含以下内容：

- @author 函数作者（们）
- @date 函数编写日期
- @bug 已知的 bug

如果函数的参数或者返回值是模板类型，还需要加上 `@tparam` 注释。

举例：

```
/**
 * @brief 移除变换矩阵指定轴的旋转变换量
 * @tparam T 变换矩阵类型
 * @param transformation 变换矩阵
 * @param axis 指定轴
 * @return 移除旋转变换量后的变换矩阵
 */
template <typename T>
static T RemoveRotation(const T &transformation, Axis axis);
```

函数注释只是对函数功能的简要描述，而不是描述函数如何工作，描述函数如何工作是属于实现注释的范畴。

类注释

类注释与函数注释相同，放在类定义上方，用块注释，且以 `/**` 开头。

类注释只需对类的功能进行简要描述即可。

举例：

```
/**
 * 与 HoloLens 通信的类，负责与 HoloLens 通信。
 */
class HoloCommunicator {
    ...
}
```

变量注释

变量注释用行注释，放在变量声明上方，用于对变量的含义进行描述。

特别地，如果变量存在特殊值，如 `0`、`-1`、`NULL` 等，应该在变量声明处注释该特殊值的含义。

举例：

```
// 计数器
int count;

// 查找范围，-1 表示全范围查找，0 表示不查找。
int find_size;
```

实现注释

实现注释用于描述代码的实现细节，用行注释，可以放在代码上方，也可以放在代码结尾。但放在代码结尾的注释必须简短。

实现注释应该描述 “为什么” 而不是 “是什么”，即实现注释应该描述代码为什么这么写，而不是描述代码是什么。

举例：

```
// 以下是差的实现注释（是什么）
// 把 i 转换为 double 类型
double size = static_cast<double>(i);

// 以下是好的实现注释（为什么）
// 为了避免整型溢出，将 i 转换为 double 类型
doubel size = static_cast<double>(i);
```

待办注释

待办注释应该以 `TODO:(your_name)` 开头，括号内为注释者的姓名，后面跟一个空格，然后是待办事项的描述。

举例：

```
// TODO:(He Zhizhou) 优化算法
```

Bug 注释

Bug 注释应该以 `FIXME:` 开头，后面跟一个空格，然后是 Bug 的描述。

举例：

```
// FIXME: 修复内存泄漏
```

格式规范

缩进格式

代码缩进使用 4 个空格，不要使用 Tab（制表符）。即 4 个空格代表一次缩进。

标准行宽

每行代码的长度不超过 80 个字符。

函数格式

函数的左大括号(`{`)应该与函数名在同一行，且左大括号(`{`)前面应该有一个空格。

函数的右大括号(`}`)应该单独占一行。

即使函数体只有一行代码，也应该遵循上述规范。

举例：

```
void goodFunction() {       // 好
    ...
}

void badFunction()         // 差
{
    ...
}
```

如果函数声明过长，且函数参数较多，可以将参数分成多行，每行一个参数，且每行参数前面有一个空格。

举例：

```
ReturnType ClassName::FunctionName(Type parameter1, Type parameter2,
                                   Type parameter3) {
    ...
}

ReturnType NameSpaceName::ClassName::FunctionName(Type parameter1,
                                                  Type parameter2,
                                                  Type parameter3) {
    ...
}
```

```
NameSpaceName::ReturnType NameSpaceName::ClassName::LongFunctionName(
    Type parameter1,
    Type parameter2,
    Type parameter3) {
    ...
}
```

条件语句格式

条件语句的左小括号(`(`)后面和右小括号(`)`)前面均不加空格。

`if` 与 `(` 之间应该有一个空格，`)` 与 `{` 之间同样应该有一个空格。

举例：

```
if (count > 10) {    // 好
    ...
}

if( count>10 ) {     // 差
    ...
}

if(count>10) {       // 差
    ...
}
```

与函数类似，条件语句的左大括号(`{`)应该与条件语句在同一行，且左大括号(`{`)与右小括号(`)`)之间应该有一个空格。

`else if` 应该与前面代码块的右大括号(`}`)在同一行，且 `else if` 与前面代码块的右大括号(`}`)之间应该有一个空格。

举例：

```
if (condition) {     // 好
    ...
} else if (condition) {
    ...
} else {
    ...
}

if (condition)       // 差
{
    ...
}
else if (condition)
{
    ...
}
```

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

条件语句必须使用大括号(`{}`)，即使条件语句只有一行代码。

举例：

```
if (count > 10) {               // 好
    return true;
}

if (count > 10) return true;    // 差
```

循环语句格式

与条件语句类似，且必须使用大括号(`{}`)。

举例：

```
for (int i = 0; i < 10; ++i) {  // 好
    ...
}

for (int i = 0; i < 10; ++i)    // 差
    ...
```

switch 语句

`case` 语句应该有一次缩进。 `case` 语句的代码块应该用大括号(`{}`)包含，且每个 `case` 语句的代码块都必须有 `break` 语句，default 语句的代码块不需要 `break` 语句。

即使 `default` 语句的代码块为空，也应该写出 `default` 语句。

举例：

```
switch (count) {
    case 1: {
        ...
        break;
    }
    case 2: {
        ...
        break;
    }
    default: {
        ...
    }
}
```

bool 表达式格式

当 bool 表达式超过标准行宽时，应该将 bool 表达式拆分为多行，且将逻辑与(`&&`)或者逻辑或(`||`)运算符放在行尾。

举例：

```
if (first_count_number > other_count_number &&
    second_count_number > 10 ||
    is_ready) {
    ...
}
```

模板函数格式

模板函数的模板参数应该单独占一行。

举例：

```
template <typename T>
T max(T a, T b) {
    ...
}
```

类格式

类的访问权限修饰符`private`、`protected`、`public`应该单独占一行，且不加缩进。

类的数据成员都需要加一次缩进。

以下情况需要使用空行：

- 不同访问权限的成员之间（一个空行）
- 成员函数之间（一个空行）
- 不同类之间（两个空行）

举例：

```
class FaceDetector {
private:
    std::string config_path;
    int count;

public:
    FaceDetector();

    virtual ~FaceDetector();

    void inference();
};
```

```
class ObjectDetector {
    ...
}
```

其他类格式规范参考成员声明顺序。

初始化列表格式

初始化列表的冒号(`:`)的前面与后面都应该有一个空格。

当初始化列表过长时，应该将初始化列表拆分为多行，将冒号(`:`)放在行首，逗号(`,`)放在行尾。

举例：

```
Arm::Arm() : length(0.0), width(0.0),
             height(0.0) {
    ...
}

BotCar::BotCar()
    : name("bot_car"), speed(0.0), run_mode(0) {
    ...
}
```

命名空间格式

命名空间里的内容不添加缩进。

且需要在命名空间的右大括号(`}`)之后加上命名空间注释，注释格式为 `// namespace <namespace_name>`。

```
namespace etrs::network{
class Communicator {    // 好
    ...
};

    class Bluetooth{    // 差
        ...
    };
}   // namespace etrs::network
```

水平留白

以下是本风格指南中的水平留白规范：

```
int negative_count = -5;
int count = 0;

++count;
count--;
```

```
for (int i = 0; i < 10; ++i) {
    ...
}

if (count > 10 && !is_finished) {
    ...
}

switch (count) {
    case 1: {
        ...
        break;
    }
    case 2: {
        ...
        break;
    }
    default: {
        ...
    }
}

int max = std::max(1, 2);

namespace etrs::geometry {
class Point : public Geometry3D {
public:
    Point() : x(0.0), y(0.0), z(0.0) {}
};

int res = a * b + c / d;
double result = static_cast<double>(a);

vector<int> vec = {1, 2, 3, 4, 5};
vector<Point *> point_vec;
}
```

垂直留白

垂直留白是指空行，上文已经对一些情况下的空行做了规定（例如类格式），此处不再对垂直留白做要求，但不要滥用空行，也不要不使用空行。

建议在不同逻辑功能的代码之间使用空行隔开，使代码更加清晰。

VSCode 设置

Clang-Format

待完善...

写在最后

本风格指南参考了以下文章：

- Google C++ 代码风格指南
- 屎山代码书写准则

作者：He Zhizhou
Github 仓库：Cpp Style Guide

> 如果你发现了任何问题或者有任何建议，欢迎在 Github 上提出 Issue。非常欢迎你的反馈和贡献。