Pointcloud Recommeneded Standard 001 – Coding styles

Author: Chen Tianze (crazycz.cc@ckx-ily.cn)

---

0. 注释

0.1 建议将注释分为两种类别：

- 用于简要说明代码的行为（如函数的功能，类的设计意图）
- 用于详细说明代码的工作原理

[例子

```
-- 用于说明代码行为的注释
// 解析字符串中的整数，不考虑负数的情形。如果字符串不能解析为整数，
// 在 Debug 模式下，程序通过调用 abort 退出；
// 在 Release 模式下，程序行为未有定义
int parseInt(const char *str) {
  int ret = 0;
 -- 用于说明代码原理的注释
  // 取出每一位，将已有结果乘以 10，加上新的一位
  while (*str != '\0') {
    // assert 仅仅对 debug 模式有效
    assert(isdigit(*str));
    ret *= 10;
    ret += *str - '\0';
    ++str;
  }
  return ret;
}
```

— 例子结束]

0.2 用于说明代码行为的注释应被置于头文件中，和函数声明/类声明放在一起；而用于说明代码原理的注释则应被置于源文件中，和代码实现放在一起

0.3 切勿将注释当作除臭剂使用，也不要编写无意义的注释。如果你发现你需要大量的注释才能说明函数或者类的功能，或者需要大量的注释才能说明代码的工作原理，那么就有限考虑不使用注释。采取更好的命名，更好地分割职责往往有助于消除无意义的注释，且让代码更容易阅读

[例子

```
int n; // n 为学生总数        -- 原本完全不必要的注释
int studentCount;            -- 改变命名
```

— 例子结束]

0.4 如有必要请编写文档来描述程序的总体结构。注释是对于一部分程序的详细说明，而文档有益于对于程序整体结构的理解。

0.5 更新代码时务必更新代码和注释。过期的文档和注释起不到任何作用，只会让程序更难阅读。

0.6 不要编写无意义的注释，不要逐字逐句解释程序，而是帮助读者从一个更高的层面上了解程序。

0.7 不建议将 changelog 信息写入代码注释中，对代码的更改应该由版本控制系统记录，而非手动记录。

1. 基本风格

1.1 命名风格

推荐使用Java 风格的命名；也可以使用其他风格，但是在一个项目中，所有命名应该保持统一。

[例子

```
class MessageHandler {
public:
  MessageHandler(MessageDecoder const& decoder);
  void parseRawMessage(int *rawMessage, size_t length);
};
```

-- 例子结束]

1.2 缩进

推荐使用两个空格进行代码缩进，并在编辑器设置中选择将 Tab 拆分为空格；也可以使用其他缩进风格，但是在一个项目中，所有缩进风格应该保持一致。

1.3 留白

推荐在二元运算符左右两侧留出一个空格，在逗号后面留出一个空格

[例子

```
int value=a+b+fun(a,b);                    // bad
int value = a + b + fun(a, b);      // good
```

-- 例子结束]

1.4 折行

建议一行不超过 80 列：如果代码一行超过 80 列，会对分屏阅读产生不利影响。如果一行超过了 80 列，应该在适当的位置采取折行

[例子

```
total = someArray[someIndex]
         + someArray[someIndex * 3]
         + someArray[someIndex * 4];
```

-- 例子结束]

1.5 左大括号

世界上有两种程序员：

```
void fun() {
  ... // do stuff
}
```

```
void fun()
{
  ... // do stuff
}
```

推荐左大括号不换行的方式。实际程序中可以自行选择，但是在同一个项目中要保持一致。

1.6 代码格式化工具

建议使用如 clang-format 等代码格式化工具在提交代码之前进行自动格式化。控制格式化工具行为的配置文件如.clang-format 应被放置于代码仓库中。

2. 数据操作

2.1 建议永远使用语言提供的标准 bool, true 和 false 符号

- C 语言引用 stdbool.h 来获得 bool, true, false 的定义
- C++内建 bool 类型

于 windows.h 中定义的 BOOL 是一个 typedef，它是不可移植的，建议不要使用。也不要自己定义 bool, true 和 false。

2.2 建议不使用 int, long, long long，这些数据类型的长度在不同平台上存在差异，有时会产生显著问题。建议使用于 stdint.h 中定义的 intxx_t 和 uintxx_t。

2.3 不推荐使用普通整数作为循环下标，应使用 size_t。引用 stddef.h 来获得它。

2.4 不推荐将指针转换为普通的整数，应使用 intptr_t：它保证能容纳一个指针所需要的数据。

2.5 不推荐将 char 作 byte 使用，应至少使用 unsigned char；最好的办法是使用 uint8_t。char 类型默认既非 signed 亦非 unsigned，而 signed char 在进行整数提升时可能会出现潜在问题（该问题已在 serialport 部分出现）。

2.6 不推荐一些常见的手动“优化”操作，例如：

```
a * 2          ⟹    a << 1;
a / 2          ⟹    a >> 1;
a % 2 ≠ 0      ⟹    a & 2;
```

编译器比普通人更懂得优化，懂得如何利用 cpu 的流水线/专用乘法器/向量化操作。手动优化往往多此一举，并且使代码变得不够清晰。

2.7 请勿对浮点数使用==运算符：浮点数天生具有缺陷，使用==进行比较会造成严重的问题。

2.8 C++中请积极使用 nullptr 来代表空指针，不推荐使用 0 或者 NULL。

2.9 请积极使用标准库解决问题。在开启优化的前提下，标准库往往比手写代码更快；标准库的效果和稳定性也更好。

2.10 推荐减少对原始数组的使用，多使用容器：它们提供了更多功能，且和标准库算法配合更好。

2.11 不推荐使用全局变量，至少采用单例取代之，或者将“共享变量”作为函数参数或者类成员。不可变的全局常量仍然可以随意使用。

3. 控制流

3.1 尽量缩减局部变量的作用域，一个不被使用却阴魂不散的局部变量往往会带来问题。

[例子
```
// bad
int i;
for (i = 0; i < 10; i++) {
  .. // stuff
}

// good
for (int i = 0; i < 10; i++) {
  .. // stuff
}
```
-- 例子结束]

3.2 建议优先使用标准库算法和 range-based for，减少手写计数循环。非常遗憾的是 C++尚未引入 range 操作，导致很多算法还没有办法很好地用 range+算法来表达。

3.3 需要 goto 时请尽管使用。对于 goto 语句的错误认知源自教科书的错误描述，它们让人相信 goto 是邪恶的，但是事实上并不是（不然美国国防部十年磨一剑的 Ada 语言也不会保留它）：它们在进行错误处理和跳出多层循环时非常有用。

[例子
```
while (someCondition) {
  while (anotherCondition) {
    if (somethingHappens) {
      goto end_while;
    }
  }
}
end_while: ...
```
-- 例子结束]

3.4 需要递归时请尽管使用，只要递归深度不至于爆栈。递归并不会引起显著性能下降，更何况在 profiling 之前考虑性能问题本身就是很邪恶的。

3.5 在使用 C++编程时，不推荐使用“类型识别码”来标记继承谱系中的类，也不推荐使用 RTTI 和向下转型。这些元素会破坏面向对象设计，并且让修改程序变得困难。

4. 函数

4.1 建议使用枚举（最好是受限制的枚举）表达某些取值范围有限的概念，例如波特率、数据校验方法等。不建议在此种场合使用字符、字符串或者数值类型。

[例子

```
-- bad
-- 实现者需要考虑当解析字符失败时的处理方式：让程序崩溃、按默认方式运行还是返回失败
-- 调用者则被迫阅读文档来确定应该传入什么字符
void setParityCheckMode(int fd, char mode);

-- good
enum class ParityCheckMode { None, Odd, Even };
void setParityCheckMode(int fd, ParityCheckMode mode);
-- 例子结束]
```

4.2 如果要给函数增加参数，建议将新增的参数加在参数列右侧（我是C++隐式转换规则的受害者）。

4.3 尽量多地使用 const 标记可以避免很多问题，也可以减少编译器警告，并且让代码具有更高的可读性。

[例子

```
-- bad
int openFileCaseInsensitive(char *path);
-- good
int openFileCaseInsensitive(const char *path);
-- 例子结束]
```

4.4 函数的长度不宜超过 50 行代码：可以考虑将函数拆分为若干个小函数。通过编译器的各种优化，进程内函数调用的开销可以降到非常低，请不要担心函数调用开销。

5. 杂项

5.1 建议使用不带 BOM 的 UTF-8 编码存储文件。UTF-8 支持多种语言字符，便于扩展，且能在多数操作系统上正常编辑。

5.2 建议使用 Unix LF 行尾而非 Windows 的 CRLF 行尾。

5.3 建议适当使用 namespace。如果不使用 namespace，建议添加特定前缀来标识属于特定模块的代码。

[例子
```
namespace UART {
  int openPort(const char *deviceFile);
  void closePort(int port);
};

// use prefix instead
int UARTOpenPort(const char *deviceFile);
void UARTClosePort(int port);
```
-- 例子结束]

5.4 不建议在头文件中 using namespace

5.5 不建议在头文件中引用用不到的头文件，建议将这部分引用工作换到相应的源文件中进行。