XINJE

C++ 编程规范

函数名大驼峰、小驼峰、下划线风格

说明:

本编程规范主要由风格和规则组成。

风格：命名约定、注释、格式构成，更注重代码的外观。

规则：良好的实践建议（由常见易错场景总结归纳所得），更注重代码的内在。

良好一致的风格有助于代码阅读维护，也方便后续开展代码审查。
严格遵守代码规则能很好的降低代码出错的风险和BUG率，间接提高开
发效率。

一、 命名约定

命名的风格可能有多种方式，但其目的主要是统一风格方便阅读，目前该规范主要约定如下两种命名风格。

unix内核风格	单词全小写，用下划线分割
驼峰风格	大小写字母混用

以上两种风格结合项目需要选择其一，确定下来要保证风格一致即可。(下文主要对驼峰风格进行说明)

1. 文件命名

1.1 文件名全小写构成，中间可由下划线（_）进行连接。

可接受的文件命名：

```
my_useful_class.cpp
myusefulclass.cpp
```

1.2 文件名不要和系统头文件或者C++标准库头文件相同，文件名应尽可能明确表述内容功能。

例如：

```
ui_control_factory.cpp
```

1.3 文件名后缀，如无特别原因默认使用如下风格：

.cpp 源文件

.h 头文件

2. 命名空间

命名空间名称由全小写构成，中间可由下划线（_）进行连接，其命名尽量基于项目名称和目录结构：例如：xinje：:xxxprj：:ui_control。

3. 函数命名

函数采用小驼峰/大驼峰命名方式。如：

```
//小驼峰
void readBook();
int addTableEntry()
int deleteUrl()

//大驼峰
void ReadBook();
int AddTableEntry()
int DeleteUrl()
```

4. 类型命名

类型采用大驼峰命名方式：

MyExcitingClass、MyExcitingEnum。

所有类型命名：类、结构体、类型定义（typedef）、枚举 使用相同约定，

例如：

```
//类和结构体
 class UrlTableTester
struct UrlTableProperties

// 类型定义
 typedef hash_map<UrlTableProperties *, string> PropertiesMap;

//枚举
 enum UrlTableErrors
```

5. 变量命名

变量名采用小驼峰/下划线风格，私有成员变量采用小驼峰/下划线风格后加下划线方式。

5.1 普通变量命名：

举例：

```
 string tableName;   //小驼峰
 string table_name;  //下划线
```

5.2 数据类成员：

```
struct UrlTableProperties
{
    string name;
   //数据成员变量
    int numEntries;//小驼峰
    int num_entries;//下划线
}

class Student
{
 private:
     int age_;// 类的成员变量
     //类的成员变量
     int skinColor_;//小驼峰
      int skin_color_;//下划线
}
```

5.3 全局变量：

全局变量以g_标识前缀，后面采用**小驼峰/下划线**命名 。例如： g_totalDevices或g_total_devices；

6. 常量命名

采用全部大写，单词间以下划线相连

```
const int DAYS_IN_WEEK = 7;
```

7. 枚举命名

采用全部大写，单词间以下划线相连：ERROR_OUT_OF_MEMORY。

例如：

```
enum UrlTableErrors
{
    OK = 0,
    ERROR_OUT_OF_MEMORY,
    ERROR_MALFORMED_INPU
};
```

8. 宏命名

采用全部大写，中间可由下划线（_）进行连接，宏内容如非必要需要进行包裹，用括号或者do while(0)等：

```
#define ROUND(x)  do{}while(0)
#define_PI_ROUNDED__(3.0)___
```

但在C++中，尽量以const，enum，inline替换#define

二、 注释

一般的，尽量通过清晰的架构逻辑，好的符号命名来提高代码可读性；需要的时候， 才辅以注释说明。 注释是为了帮助阅读者快速读懂代码，所以要从读者的角度出发，按需注释。

注释内容要简洁、明了、无二义性，信息全面且不冗余。

注释跟代码一样重要。 写注释时要换位思考，用注释去表达此时读者真正需要的信息。在代码的功能、意图层次上进行注释，即注释解释代码难以表达的意图，不要重复代码信息。 修改代码时，也要保证其相关注释的一致性。只改代码，不改注释是一种不文明行为，破坏了代码与注释的一致性，让阅读者迷惑、费解，甚至误解

1. 注释风格

在C++ 代码中，使用/* */和//都是可以的。 按注释的目的和位置，注释可分为不同的类型，如文件头注释、函数头注释、代码注释等等； 同一类型的注释应该保持统一的风格。

建议：

单行注释使用 //进行注释

多行注释使用/**/注释

如果需要生成接口文档等特殊原因，可按照第三方工具注释风格 。例如doxygen、ghostdoc等。

2. 文件注释

在每一个文件开头加入版权公告，然后是文件内容描述。

```
/*************************************************************************
 * Copyright (C) 2020-, Thinget Electronic Co., Ltd.
 *
 * File Name: // 文件名 （注释对齐）
 * Description: // 简要描述本文件的内容，完成的主要功能
```

```
 * Others: // 其它内容的说明
 * Version: // 输入当前版本
 * Author: // 输入作者名字及单位
 * Date: // 输入完成日期，例：2021-12-12
 *
 * History 1: // 修改历史记录，包括修改日期、修改者及修改内容
 * Date:
 * Version:
 * Author:
 * Modification:
 * History 2: …
 ***********************************************************************/
```

建议：每次进行代码修改需要填写修改记录，记录较多可组织该文件开发人员和所有修改人员进行评审确认，删除合并修改记录。

3. 类注释

每个类的定义要附着描述类的功能和用法的注释，除非类的功能非常清晰明了。在类的注释描述中重点描述功能、使用场景、以及使用注意事项和风险点。

```
/*******************************小驼峰**********************************
*  Iterates over the contents of a GargantuanTable. Sample usage:
*   GargantuanTable_Iterator* iter = table->newIterator();
*   for (iter->seek("foo"); !iter->done(); iter->next()) {
*    process(iter->key(), iter->value());
*   }
*   delete iter;
***********************************************************************/
class GargantuanTable_Iterator
{
    ...
};

/*******************************大驼峰**********************************
*  Iterates over the contents of a GargantuanTable. Sample usage:
*   GargantuanTable_Iterator* iter = table->NewIterator();
*   for (iter->Seek("foo"); !iter->Done(); iter->Next()) {
*   process(iter->key(), iter->value());
*   }
*   delete iter;
***********************************************************************/
class GargantuanTable_Iterator
{
    ...
};

/*******************************下划线**********************************
*  Iterates over the contents of a GargantuanTable. Sample usage:
*   GargantuanTable_Iterator* iter = table->new_iterator();
*   for (iter->seek("foo"); !iter->done(); iter->next()) {
*   process(iter->key(), iter->value());
*   }
```

```
*   delete iter;
*************************************************************************/
class GargantuanTable_Iterator
{
    ...
};
```

4. 函数注释

对外接口函数注释分定义和实现：

定义处的函数：注释主要偏函数功能，使用方法和可能异常等内容

实现处的函数：注释主要偏函数的实现和细节

4.1 函数声明：

注释于声明之前，描述函数功能及用法

```
/*************************************************************************
* @brief   : 两数相减 a-b
* @param   : a 被减数
* @param   : b 减数
* @return  : a 与 b 的差
* @note    : 当 a,b 为不同符号时，此函数结果可能会出现溢出现象
*************************************************************************/
int sub(int a,int b);//小驼峰
int Sub(int a,int b);//大驼峰
```

4.2 函数定义实现：

```
/*************************************************************************
* 描述实现细节，比如使用了 xxx 方法进行加速等
*************************************************************************/
//小驼峰
int sub(int a,int b)
{
}
//大驼峰
int Sub(int a,int b)
{
}
```

对非对外接口类函数注释可以结合必要性进行说明，对于过于简单清晰的可以不加注释，需要添加注释的可以简化，重点描述功能和实现即可

5. 变量注释

5.1 类数据成员：

每个类数据成员应尽量用友好的命名来进行自说明含义用途，如果单个变量内容富含特殊逻辑或者名称不足以表达，需要加以说明 如：

```
private:
  /* 用于指向用户自定义格式数据内容，其数据内存的申请和释放
   * 由用户自己管理 */
  void* privateData_;//小驼峰
  void* private_data_;//下划线
```

5.2 全局变量（常量）：

和数据成员相似，所有全局变量（常量）在名称不能自说明含义用途的情况下，应增加注释说明如：

```
1.  风格一：
2.  //当前连接的 iot 从设备数量
2.  const int g_numSlaves = 6;//小驼峰
3.  const int g_num_slaves = 6;//下划线
4.  风格二：
5.  // 当前连接的 iot 设备数量，包括不兼容的设备
6.  const int g_numSlaveDevice = 6;//小驼峰
7.  const int g_num_slave_device = 6;//下划线
```

以上两种注释风格都可，主要根据注释内容长短来选择。较长的选择第二种，较短的可以选择第一种。

6. 实现注释

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释。

如：

```
1.  int count = 0;
2.  //以下 while 循环逻辑是获取 n 的二进制格式中 1 的个数
3.  while (n)
4.  {
5.      n = n & (n - 1);
6.      count++;
7.  }
```

7. TODO 注释

对那些临时的、短期的解决方案，或已经够好但并不完美的代码使用TODO 注释。

这样的注释要使用全大写的字符串TODO，后面括号里加上你的姓名、邮件地址等，再加上冒号： 目的是可以根据统一的TODO 格式进行查找：

```
1.  // TODO(zhangsan@xinje.com): 图形锯齿可以使用第三方库优化
2.  // TODO(lisi) : 此处 CPU 资源占用略高，需优化
```

三、 格式

1. 行长度

每一行代码字符数不超过 100左右。

例外：

如果一行注释包含了超过100左右的命令或URL，出于复制粘贴的方便可以超过100

2. 空格还是制表位

使用空格，每次缩进 4 个空格，MSVC中可以使用tab缩进

3. 函数声明与定义

返回类型和函数名在同一行。合适的话，参数也放在同一行。函数看上去像这样：

```
//小驼峰
ReturnType ClassName::functionName(Type parName1, Type parName2)
{
    doSomething();
    ...
}
//下划线
ReturnType ClassName::functionName(Type par_name1, Type par_name2)
{
    doSomething();
    ...
} _
//大驼峰
ReturnType ClassName::FunctionName(Type parName1, Type parName2)
{
    DoSomething();
    ...
}
```

如果同一行文本较多，容不下所有参数：

```
ReturnType ClassName::ReallyLongFunctionName(Type parName1,
                                Type parName2,
                                Type parName2)//大驼峰
 ReturnType ClassName::reallyLongFunctionName(Type par_name1,
                                Type par_name2,
                                Type par_name3)//下划线
ReturnType ClassName::reallyLongFunctionName(Type parName1,
                                Type parName2,
                                Type parName3)//小驼峰
 {
    DoSomething();//大驼峰
    doSomething();//小驼峰/下划线
    ...
}
```

甚至连第一个参数都放不下：

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type  ParName1,
    Type  ParName2,
    Type  ParName3)//大驼峰
ReturnType LongClassName::reallyReallyReallyLongFunctionName(
    Type  par_name1,
    Type  par_name2,
    Type  par_name3) //下划线
ReturnType LongClassName::reallyReallyReallyLongFunctionName(
    Type  parName1,
    Type  parName2,
    Type  parName3) //小驼峰
{
    doSomething();//小驼峰
    DoSomething();//大驼峰
}
```

如果函数为const 的，关键字const 应与最后一个参数位于同一行。

```
ReturnType FunctionName(Type par) const

ReturnType functionName(Type par) const

ReturnType ReallyLongFunctionName(Type par1,
                                  Type par2) const

ReturnType reallyLongFunctionName(Type par1,
                                  Type par2) const
```

4. 函数调用

函数调用参数尽量放在同一行。

函数调用遵循如下形式：

```
bool retval = doSomething(argument1, argument2, argument3); // 逗号后留一空格
bool retval = DoSomething(argument1, argument2, argument3); // 逗号后留一空格
```

如果同一行放不下，可断为多行，后面每一行都和第一个实参对齐，左圆括号后和右圆括号前不要留空格：

```
bool retval = doSomething(averyveryveryverylongargument1,
                          argument2, argument3);
bool retval = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

如果函数参数比较多，可以出于可读性的考虑每行只放一个参数：

```
bool retval = doSomething(argument1,
argument2,
argument3,
argument4);
bool retval = DoSomething(argument1,
argument2,
argument3,
argument4);
```

5. 条件语句

5.1 if之后如果需要else则关键字else另起一行。

```
if (condition)
{
    ... // 4 空格缩进.
}
else
{
    ...
}
```

5.2 条件判断语句，括号和括号内逻辑内容前后无空格

```
if (cond1 && cond2)   // 左括号和 cond1 之间、cond2 和右括号之间无空格
                //&& 前后有空格
{
    ... // 4 空格缩进
}
else
{
    ...
}
```

注意所有情况下if 和左圆括号间有个空格：

```
if (condition) //正确格式
if(condition)  //错误格式
```

有些条件语句写在同一行以增强可读性，只有当语句简单并且没有使用else子句时使用：

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

如果语句有else 分支，是不允许写在同一行的：

```
// 错误格式
if (x) doThis();
else doThat();
```

通常，单行语句不需要使用大括号，但建议if必须使用大括号：

```
if (condition)
    doSomething();
if (condition)
 {
    doSomething();  // 4 空格缩进. 【推荐】
 }
```

但如果语句中哪一分支使用了大括号的话，其他部分也必须使用。

6. 循环和开关选择语句

switch 语句可以使用大括号分块；空循环体应使用{}或continue。如果有不满足case 枚举条件的值，要总是包含一个default。如果default 永不会执行，可以简单的使用assert：

```
switch (var)
{
    case 0: // 4空格缩进
    {
        ... // 4空格缩进
        break;
    }
    case 1:
    {
        ...
        break;
    }
    default:
    {
        assert(false);
    }
}
```

空循环体应使用{}或 continue，而不是一个简单的分号：

```
while (condition)  continue;  // 推荐格式
while (condition);            // 不推荐
```

7. 指针和引用表达式

句点（.）或箭头（->）前后不要有空格，指针/地址操作符（*、&）后不要有空格。下面是指针和引用表达式的正确范例：

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

注意：

在访问成员时，句点或箭头前后没有空格；指针操作符*或&后没有空格。在声明指针变量或参数时，星号与类型或变量名紧挨都可以：

```
// 推荐格式
char *c;
const string &str;
//不推荐格式
char * c;
const string & str;
```

同一个文件（新建或现有）中起码要保持一致。

8. 布尔表达式

如果一个布尔表达式超过标准行宽（100 字符），可以进行断行处理。下例中，逻辑与（&&）操作符总位于行尾：

```
if (thisOneThing > thisOtherThing &&
    aThirdThing == aFourthThing &&
    yetAnother & lastOne)
{
    ...
}
```

为增强可读性，可以考虑额外插入圆括号，明确优先级。

```
if ((thisOneThing > thisOtherThing) &&
    (aThirdThing == aFourthThing) &&
    (yetAnother & lastOne))
{
    ...
}
```

9. 预处理指令

预处理指令不要缩进，从行首开始。即使预处理指令位于缩进代码块中，指令也应从行首开始。

```
1.  //if (lopsidedScore)
2.  if (lopsided_score)
3.  {
4.  #if DISASTER_PENDING    // 正确格式，不需要缩进
5.      //DropEverything();
6.      dropEverything();
7.  #endif
8.      //DackToNormal();
9.      backToNormal();
10. }
11.
12. if (lopsided_score)
13. {
14.     #if DISASTER_PENDING  // 错误格式
15.     //DropEverything();
16.     dropEverything();
17.     #endif                // 错误格式
18.     //DackToNormal();
19.     backToNormal();
20. }
```

10. 类格式

声明属性依次序是public：、protected：、private：

类声明的基本格式如下：

```
1.  class MyClass : public OtherClass
2.  {
3.  public:
4.      MyClass();  // 4空格缩进
5.      explicit MyClass(int var);
6.      ~MyClass() {}
7.
8.     //void someFunction();
9.      void SomeFunction();
10.  //void someFunctionThatDoesNothing()
11.     void SomeFunctionThatDoesNothing()
12.     {
13.     }
14.  // void setSomeVar_ (int var)
15.     void SetSomeVar_ (int var)
16.     {
17.        //some_var_ = var;
18.          someVar_ = var;
19.     }
20.  // int someVar() const
21.  int SomeVar() const
22.     {
23.         //return some_var_;
24.         return someVar_;
25.     }
26.
27. private:
28.     //bool someInternalFunction();
29.     bool SomeInternalFunction();
30.     //int some_var_;
31.     int someVar_;
32.    //int some_other_var_;
```

```
    int someOtherVar_;
};
```

注意：除第一个关键词（一般是public）外，其他关键词前空一行，如果类比较小的话也可以不空；

1）这些关键词后不要空行；

2）public 放在最前面，然后是protected 和private；

11. 初始化列表

构造函数初始化列表放在同一行或按四格缩进并排几行。两种可以接受的初始化列表格式：

```
//MyClass::MyClass(int var) : someVar_(var), some_other_var_(var + 1)
MyClass::MyClass(int var) : someVar_(var), someOtherVar_(var + 1)
{
}
```

或

```
/* MyClass::MyClass(int var)
        : some_var_(var),
        some_other_var_(var + 1)   */
MyClass::MyClass(intvar)
    : someVar_(var),        // 4 空格缩进
    someOtherVar_(var + 1)   // 向上对齐
{
    ...
    //DoSomething();
    doSomething();
    ...
}
```

12. 命名空间格式化

命名空间不添加额外缩进层次，例：

```
namespace xinje
{
namespace xxprj
{
class FooClass
{
public:
private:
}
}
}
```

例外：

在MSVC或者其他特定平台，可根据IDE 默认格式进行调整

13. 水平留白

水平留白的使用因地制宜。不要在行尾添加无谓的留白。

普通：

```
//void Foo(bool b)
void f(bool b)
{
    ...
    //以下 2 种风格都可，但要做到上下统一
    int x[] = { 0 }; // 初始化列表前后增加空格
    int x[] = {0};   // 初始化列表前后无空格
 }

// 继承列表和初始化列表中的冒号前后有空格
class Foo : public Bar
{
public:
    Foo(int b) : Bar(), baz_(b) {}
    //void Reset() { baz_ = 0; }
    void reset() { baz_ = 0; }
    ...
}
```

循环和条件语句：

```
while (test)
{
}
switch (i)
{
}
for (int i = 0; i < 5; ++i)
{
}
```

```
if (test)
{
}
```

while,switch,for,if 后面和（之间有一个空格

操作符：

```
x = 0;          // =左右各有一个空格
x = -1;             // -单目运算符，不留空格

if (x && !y)        // 逻辑运算符左右各加一个空格

v = w * x + y / z;  // 双目运算符，通常前后各一个空格
v = w*x + y/z;  // 但可以更清晰表达内容时，去掉前后空格也是可以的.
v = w * (x + z);    // 括号和内容之间不留空格.
```

模板和转换：

```
vector<string> x; // 尖括号内不需要空格
vector<char *> x; // 类型和指针符之间可以加空格
```

14. 垂直留白

14.1 垂直留白越少越好。

14.2 函数定义之间空行不要超过 2 行，函数体头、尾不要有空行，函数体中也不要随意添加空行

```
 //void Function()
void function()
{
    // 前后不需要空行
}

//代码块头、尾不要有空行
while (condition)
{
    // 代码逻辑，后面不需要有空行
}

if (condition)
{
    // 代码逻辑，前面不需要有空行
}
```

四、 规则（以下是DEMO，具体内容可以各个部门进行补充）

1. 禁止在构造函数和析构函数中调用虚函数 （XX部门提供）

说明：在构造函数和析构函数中调用当前对象的虚函数，会导致未实现多态的行为。在 C++ 中，一个基类一次只构造一个完整的对象。

示例：类 Base 是基类，Sub 是派生类

```
class Base
{
public:
    Base();
    //virtual void Log() = 0;
    virtual void log() = 0; // 不同的派生类调用不同的日志文件
};

Base::Base()
{
    log(); // 调用虚函数 Log
    //Log();
}

class Sub : public Base
{
public:
    virtual void log();
    //virtual void Log();
};
```

当执行如下语句： Sub sub；会先执行 Sub 的构造函数，但首先调用 Base 的构造函数，由于 Base 的构造函数调用虚函数 log/Log，此时 log/Log 还是基类的版本，只有基类构造完成后，才会完成派生类的构造，从而导致未实现多态的行为。 同样的道理也适用于析构函数。