

运营支持部 C&C++代码规范

Taomee 淘米

目 录

运营支持部 C&C++代码规范.....	1
一、关于本文档.....	1
1.1 修改历史.....	1
1.2 参考文档.....	1
1.3 编写目的.....	1
二、规范内容.....	1
2.1 文件结构.....	1
2.2 代码风格规范.....	3
2.3 注释.....	7
2.4 标志符命名规则.....	12
2.5 其它规则.....	15
三、附录.....	16
3.1 doxygen 简介.....	16
3.2 vim 和 doxygen.....	16

一、关于本文档

1.1 修改历史

版本	作者	修改日期	修改概要	备注
V1.0.0	Aceway	2010-01-14	整理、创建	针对部分需要，细化、明确规范；

1.2 参考文档

1、《C&C++编码规范.doc》公司早期后台开发组编制的规范。

2、《代码大全》第二版。

1.3 编写目的

公司已有一份《C&C++编码规范.doc》作为后台开发的指导，但其部分细节未明确，针对性不强，对于我们部门内部的开发情况指导有些不到位，因此针对部门内部需求编制补充规范，在两份规范不一致的地方，**请以本规范为准**。

目的在于：

强化、统一编码习惯，提高团队的整体编码质量。

给新同事明确的要求，从编码方面加强团队建设。

二、规范内容

本部分在包含公司《C&C++编码规范.doc》内容的基础上补充、修订一些条例。

2.1 文件结构

每个 C++/C 程序通常需要分为两个文件。一个文件用于编写程序的声明（declaration），称为头文件。另一个文件用于保存程序的实现（implementation），称为定义文件。程序的头

文件以“.h”为后缀；C 程序的定义文件以“.c”为后缀，C++程序的定义文件以“.cpp”为后缀。

2.1.1 文件头注释

头文件和定义文件的开头**必须**有注释说明该文件的用途。成对的头文件和定义文件（比如 a.c 和 a.h）中，**必须**至少有一个文件的开头有注释说明文件的用途，最好两个都有注释说明，参见 [2.3.2 推荐的注释风格](#)。

2.1.2 头文件

1、为了防止头文件被重复包含，头文件的开头和结尾**必须**用#ifndef/#define/#endif 包围起来。下面会有例子说明。

2、用#include <headername>格式来包含标准头文件、系统头文件及第三方库的头文件；用#include “headername.h”格式来包含自定义的头文件。

3、包含头文件的顺序在不影响编译时为：标准头文件、系统头文件、第三方库头文件，最后才是自定义头文件。

```
/**
 *=====
 *
 * @file      foo.h
 *
 * @brief     编码规范举例说明
 *
 *
 * compiler   gcc4.1.2
 *
 * platform   比如 Linux
 *
 *
 * copyright:  TaoMee, Inc. ShangHai CN. All rights reserved.
 *

```

```

*=====

*/

#ifndef FOO_H_

#define FOO_H_


#include <math.h>

#include <unistd.h>

#include <openssl/md5.h>


#include "myheader.h"

#endif

```

4、如果是 C++ 的话，**禁止**在头文件中使用“using namespace xxx;”的语句，这样会污染包含或者间接包含了该头文件的文件里的名字空间。

2.2 代码风格规范

C 代码风格主要遵循为大多数人所熟知的 K&R 风格；C++ 代码风格遵循 C++ 之父 B.S 的风格。每个人的代码风格稍有差异是可以接受的，但是至少必须保证同一个源文件中所有的代码都是统一的风格！多人同时编写同一个源文件时，所有人都**必须**遵循统一的风格。最好能做到同一个模块的代码风格完全一致。

2.2.1 空行

- 1、在结构体、枚举类型、联合类型、类、函数定义结束之后**必须**加至少一个空行。
- 2、逻辑上密切相关的语句之间不加空行，否则应当加空行分隔。

typedef struct foo {	while (condition) {
----------------------	---------------------

<pre> ... } foo_t; 空行 void function1() { ... } 空行 void function2() { ... } </pre>	<pre> statement1; 空行 if (condition) { statement2; } else { statement3; } 空行 statement4; } </pre>
--	---

2.2.2 代码行

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。

2、if、for、while、do while、switch 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**必须**用大括号{}括起来。

3、**尽可能**在定义变量的同时初始化该变量。只定义变量而不初始化的话，容易导致最后使用了未被初始化的变量，从而导致程序崩溃。但是，初始化变量必须使用有意义的值。如果纯粹为了初始化而不考虑初始化的值是否有意义，那还不如不初始化。

推荐风格	不推荐或严禁的风格
<code>int width = 20; /** 宽度 */</code>	<code>/** 宽度高度深度 */</code>

<pre>int height = 10; /** 高度 */ int depth = 5; /** 深度 */</pre>	<pre>int width = 20, height = 10, depth = 5;</pre> <p>推荐使用左边的风格，但这种风格也可以接受。</p>
<pre>x = a + b; y = c + d; z = e + f;</pre>	<pre>x = a + b; y = c + d; z = e + f;</pre> <p>严禁这种风格</p>
<pre>if (width < height) { do_some (); } else { do_some_other_thing(); }</pre>	<pre>if (width < height) dosomething(); else do_some_other_thing();</pre> <p>严禁这种么写！</p>
<pre>for (initialization; condition; update) { dosomething(); } other();</pre>	<pre>for (initialization; condition; update) dosomething(); other();</pre> <p>严禁这种么写！</p>

4、长行拆分。长表达式要在适当的位置拆分成多行，运算符**必须**统一放在行首或行尾（**禁止**在同一个拆分中有的行首有的行尾）。拆分出的新行要进行适当的缩进，尽量是每行长度一致，使代码优美可读。例如：

<pre>if ((very_longer_variable1 >= very_longer_variable12) && (very_longer_variable3 <= very_longer_variable14) && (very_longer_variable5 <= very_longer_variable16)) {</pre>

<pre>dosomething(); }</pre>
<pre>template <typename rule_t, typename policy_t> inline int classname<rule_t, policy_t>::do_sth() { ... }</pre>
<pre>for (very_longer_initialization; very_longer_condition; very_longer_update) { dosomething(); }</pre>

2.2.3 代码行内的空格

- 1、函数名之后**不要**留空格，紧跟左括号‘(’，以明确区分开 if、for 等关键字。
- 2、逗号‘,’和分号‘;’**不要**和前面的标识符用空格隔开。如：func(x,y)。
- 3、逗号‘,’后**要**留空格，如 func(x, y, z)。如果分号‘;’不是一行的结束符号，其后**要**留空格，如 for(initialization; condition; update)。
- 4、为了让代码更加美观，标识符和多目运算符**必须**用空格隔开。比如：x = y + z、x = (x ? x : y);。单目运算符和标识符**必须**不要用空格隔开。比如：*p、p->member、++i、arr[10]。

2.2.4 对齐

1、代码按照一次 4 个字符宽度的方式缩进，允许使用 tab 键，但是**必须**将 tab 键设置为 4 个空格。

2、**建议**将结构体、枚举、联合体类型的定义的开大括号 ‘{’ 放在第一行末尾且前留一空格，收大括号 ‘}’ 则新起一行对齐到相应关键字的第一个字母。

3、对于类、函数定义，**必须**新起一行来写 ‘{’，最后也新起一行来写 ‘}’。大括号对齐到文件的行首。

2.2.5 if 语句

1、**不要**在条件判断语句中进行赋值操作。

2、建议将 if (var == 1) 写成 if (1 == var)，即将常量写在 == 前。

3、if 或 else 后即使只有一条语句，仍然**必须**按照多条的方式写：前后加花括弧。

4、if 语句如果有 else if 分支，则**必须**在最后加上 else 分支。

2.2.6 switch 语句

1、switch 语句中的 case 标签必须和 switch 对齐。

2、每个 case 占用一行，case 后的语句用新行，并相对 case 行缩进。

3、无论是否需要 default，都**必须**把它写上，该 break 的地方别忘了 break。

2.2.7 结构体、联合体、枚举、类

具体请参见章节 [2.4.3](#)、[2.4.4](#)。

2.3 注释

C&C++有两种注释方式：多行注释 “/* ... */” 和单行注释 “// ...”。对于文件注释、函数、类、块注释，建议使用多行注释。

2.3.1 聪明地写注释

- 1、注释应当准确、易懂，不得有二义性。
- 2、注释应该说明代码为什么要这么写，而不是单纯地写代码在做什么。比如：

```
i += 3; /**< 把 i 的值增加 3 */
```

这种注释是完全没有意义的，应该这么写：

```
i += 3; /**< 因为 xxx，所以 xxx，故而这里要把 i 的值增加 3 */
```

- 3、行末的块形式注释使用 “/**< 注释 */” 形式，方便 doxygen 生成文档。
- 4、修改代码同时修改相应的注释，以保证注释与代码的一致性，不再有用的注释一定要删除。
- 5、对于注释掉的代码，建议直接删除掉。
- 6、注释**必须**放在代码的上方或右方，不可放在下方。
- 7、鉴于汉字编码的多样性，以及有些编辑器不能显示中文，建议使用英文编写注释。如果需要使用中文注释，则**必须**统一使用 utf-8 编码。
- 8、对注释的风格**必须**符合 doxygen 支持的注释风格，doxygen 本身支持多种注释风格，**建议**使用本文档的风格；同时注意注释的排版。

2.3.2 推荐的注释风格

- 1、文件头注释。

```
/**
 *=====
 *
 * @file      文件名
 *
 * @brief     文件用途说明
 *
 *
 * compiler   编译器名称及其版本（比如 gcc4.1.2）
```

```

* platform    平台名称（比如 Linux）

*

* copyright:  TaoMee, Inc. ShangHai CN. All rights reserved.

*

*=====

*/

```

以上为文件头注释**必须**有的信息。当然，也可以在此基础上添加其它信息，比如文件创建者名字、联系方式、创建日期等。

2、 函数定义注释。

<pre> /** * @fn function_name * @brief * * * @param * * @return * * @note * * @see */ </pre>	<p>@brief, @param, @return 后必须有说明，如果有多个参数，分多个@param说明。</p> <p>无参数，或返回值的相应写 void 或 none.</p> <p>@note, @see 可选。</p> <p>@note, 对调用者需要特别注意的说名。</p> <p>@see, 同类或相关函数的列表说明。</p>
<pre> /** * @fn void c_tetris::init(sprite_t* player) * @brief 提供给上层调用的三个接口函数之一. 以初始化游戏数据 * @param sprite_t* p, 玩家结构体指针 </pre>	

```

* @return void

* @note player 的个体信息数据上层已经初始化过，该用户已经加入游戏。

*/

void c_tetris::init(sprite_t* player)

{

    .....

}

```

3、类定义注释。

<pre> /** * @class class_name * @brief * * @note * @see */ </pre>	<p>@class, @brief,后必须有说明。对于接口类将@class 换成@interface interface_name</p> <p>@note, @see 可选。</p> <p>@note, 对调用者需要特别注意的说名。</p> <p>@see, 同类或相关函数的说明列表。</p>
<pre> /** * @class c_tetris * @brief 实现俄罗斯方块服务端模块。玩家加入游戏，开始游戏等逻辑由上层管理, * 本模块只处理俄罗斯方块游戏内部逻辑。 * * @note 重载实现父类的三个纯虚函数；类内部可以添加计时器和相应的超时处理函数； </pre>	

```

    * @see mpog, c_tugofwar(拔河游戏类), c_pop_t(泡泡龙游戏类)

    */

class c_tetris : public mpog

{

    .....

}

```

4、结构体、枚举、联合体定义的注释

注释的结构形式同上“类定义注释”，但将@class 相应换成：@ struct 、@ enum、@ union。

5、宏定义的注释

<pre> /** * @def MACRO_NAME * @brief * * * @note * * @see * */ </pre>	<p>@def, @brief,后必须有说明。</p> <p>@note, @see 可选。</p> <p>@note, 对调用者需要特别注意的说名。</p> <p>@see, 同类或相关函数的说明列表。</p>
<pre> /** * @def MAX(x,y) * Computes the maximum of x and y. */ #define MAX(x_, y_) (((x_) > (y_)) ? (x_) : (y_)) </pre>	

6、函数，类，结构体等代码内部的注释

<pre>/** * Comment description. * */</pre>	块式注释。
<pre>/** Comment description.*/</pre>	单行注释 或 代码行尾注释，类成员变量注释建议用此格式注释。

2.4 标志符命名规则

所谓标识符，是指我们为变量（variable）、宏（macro），或者函数（function）等取的名字。标识符的命名符合 Linux 风格，使用小写字母和下划线结合的方式。例如：this_is_a_variable。

2.4.1 必须遵守的命名规则

- 1、**限制**使用单个字符为变量名，建议长度 3 至 10 个字符。
- 2、宏**必须**全部使用大写字母，而且**必须**使用有意义的词语。
- 3、除了宏、枚举、const 常量以外的标识符，**禁止**全部使用大写来命名。
- 4、全局变量**必须**加前缀 g_，建议少使用全局变量，除非它带来明显的好处。
- 5、命名类型时，必须以“_t”结尾。例如：typedef uint32_t userid_t。
- 6、程序中**禁止**出现仅靠大小写区分的标识符。比如：int x, X。
- 7、**禁止**标识符出现大写字母和下划线混用的情况，**禁止**使用单词首字母大写划分单词：TimeType。例如：Time_Type 必须改成：time_type。
- 8、**禁止**使用以下划线‘_’打头的标识符名称以及包含两个连续下划线‘__’的标识符名称。因为 C&C++标准明确规定这些是保留标识符。使用保留标识符来命名标识符不是语法错误，可以通过编译，但是，因为这些标识符已经被 C 语言使用或者保留了，所以这样做可能会引起意想不到的问题。
- 9、**限制**使用 0、o、O、1、及 I 等特别容易引起混淆的字符。

10、目录的命名**必须**使用中横线分割单词，单词必须全部使用小写字母。

2.4.2 建议的命名规则

1、函数的命名建议采用“动词+名词”结构。

2、类、结构体、联合体、枚举、变量等的名字应当使用“名词”或者“形容词+名词”。

3、**不要**使用传说中的“匈牙利命名法”。对现代的编辑器而言，在标识符中嵌入类型信息已经再无任何意义。因为任何一个好的编辑器都会告诉你这个标识符是什么类型的。如果你非得靠在命名中嵌入类型信息来告诉你该标识符的类型，那么你应该赶快换一个编辑器了。而且，对于面向对象编程和泛型编程而言，不可能在名字中嵌入类型。

4、标识符**应当**直观且可以拼读，可望文生义。程序中的英文单词一般不会太复杂，用词应当准确。例如不要把 `current_value` 写成 `now_value`。

5、标识符的长度**应当**符合“min-length && max-information”原则。“`max_value_of_xxx`”并不见得就胜于“`xxx_max_val`”。

6、命名规则**尽量**与所采用的操作系统或开发工具的风格保持一致。Windows 应用程序的标识符通常采用“大小写”混排的方式，如 `AddChild`。而 Unix 应用程序的标识符通常采用“小写加下划线”的方式，如 `add_child`。**禁止**把这两类风格混在一起用，第三方库引入的情况除外。

7、作用域嵌套时，不同层次作用域的变量名字**禁止**完全相同。尽管两者的作用域不同不会发生语法错误，但会使人误解。

8、尽量避免名字中出现数字编号，如 `val1`、`val2` 等，除非逻辑上的确需要编号。

2.4.3 结构体、联合体、枚举的命名规则

结构体、联合体和枚举类型的名称建议都 typedef 成以“`_t`”结尾。建议这些类型的定义里每行只定义一个标识符。例如：

```
typedef struct foo {  
  
    int bar;  
  
    int foobar;  
}
```

<pre>} foo_t;</pre>
<pre>union foo { int bar; int foobar; }; typedef union foo foo_t;</pre>
<pre>typedef enum foo { bar = 1, foobar = 2 } foo_t;</pre>

2.4.4 类（class）的命名规则

- 1、普通类名**必须**使用前缀 `c_`，接口类名**必须**使用前缀 `i_`，类成员变量**必须**加前缀 `m_`。
- 2、结构体、枚举、联合体的命名规则遵循一般命名规则，无需加前缀。
- 3、**禁止** `typedef` 类名。
- 4、`public`、`protected`、`private` 标签**必须**对齐到 `class` 首字母，成员函数、变量的声明**必须**缩进 4 个空格，函数和变量的声明**禁止**混合，**必须**归类排列。这三个标签在类里面的顺序建议和本条建议里它们的顺序一样。
- 5、类的成员**必须**统一使用小写字母和下划线命名。

<pre>class c_bigint</pre>

```
{

public:

    void fun();

    ...

    int  m_data;

    ...

protected:

    int to_int();

    ...

    int  m_goody;

    ...

private:

    float to_float();

    ...

    float m_buggy;

    ...

};
```

2.5 其它规则

1、一个函数的有效代码(去掉空行，括号行，注释行)行数建议不要超过 200 行(大概 4 屏)，否则的话建议调整代码，拆分成函数调用。

2、当 if 嵌套层数达到 4 层时，请考虑程序逻辑是否可以优化，或者是否需要拆分代

码为函数调用。

3、函数的参数，特别是接口函数，尽量使用 `const` 修饰。

4、一个项目中使用的第三方库、代码，建议放在一个单独的文件夹下。

5、对于多个模块需要互相共用的代码，建议放在独立文件中共用，不要以复制粘贴代码的方式共用。

6、代码中的 SQL 语句，对于 SQL 的关键字**必须**大写。

三、附录

3.1 doxygen 简介

通常我们在写程序时，或多或少都会写上注释。所以，如果能依据程序本身的结构，结合注释整理成一个程序的参考手册，那么对于后面将接触、使用该程序代码的人而言会减少许多负担。Doxygen 正是这种工作的强有力的助手，只要你按照一定的规则写代码注释，doxygen 就可以帮助你完成整理、生成参考手册的工作。

本规范推荐的代码注释是 doxygen 能够处理的注释风格之一，在 linux 下和 windows 下都可以使用 doxygen 工具，linux 下使用方式请使用 `man doxygen` 查看说明，windows 下的使用请参看其手册。

3.2 vim 和 doxygen

要符合 doxygen 的注释风格还是得输入不少格式性的字符，在 vim 中可以使用不少插件工具自动完成这样的重复性工作，而让你只专注于写注释内容。