C++语言编码规范

姓名： 李依芳

班级： 信1604-1

学号： 20163755

日期： 2019年5月31日

目录

[1命名规范 2](#_Toc11422211)

[1.1 基本原则 2](#_Toc11422212)

[1.2 文件命名 2](#_Toc11422213)

[1.3 类型命名 2](#_Toc11422214)

[1.4 变量命名 2](#_Toc11422215)

[1.5 函数命名 3](#_Toc11422216)

[1.6 其他命名 3](#_Toc11422217)

[2 代码文件 3](#_Toc11422218)

[2.1头文件 3](#_Toc11422219)

[2.2 #define保护 4](#_Toc11422220)

[2.3#include 的顺序 4](#_Toc11422221)

[2.4前置声明 5](#_Toc11422222)

[2.5impl手法 6](#_Toc11422223)

[2.6#include中的头文件，尽量使用全路径，或者相对路径 6](#_Toc11422224)

[3 类 7](#_Toc11422225)

[3.1 让类的接口尽可能小 7](#_Toc11422226)

[3.2 声明顺序 8](#_Toc11422227)

[3.3 继承 8](#_Toc11422228)

[4 函数 9](#_Toc11422229)

[4.1 编写短小的函数 9](#_Toc11422230)

[4.2 函数的参数可能少，原则上不超过5个 9](#_Toc11422231)

[4.3 函数参数顺序 10](#_Toc11422232)

[4.5 不建议使用函数的缺省参数 10](#_Toc11422233)

[5 格式 11](#_Toc11422234)

[5.1 每行代码不多于 80 个字符 11](#_Toc11422235)

[5.2 使用空格(Space)，而不是制表符(Tab)来缩进，每次缩进4个字符 11](#_Toc11422236)

[5.3 指针符号\*，引用符号& 的位置，写在靠近类型的地方 11](#_Toc11422237)

[5.4 花括号位置 12](#_Toc11422238)

[6 其它 13](#_Toc11422239)

[6.1 const的使用 13](#_Toc11422240)

[6.2 不要注释代码，代码不使用就直接删掉 14](#_Toc11422241)

# 1命名规范

## 1.1 基本原则

所有文件的命名全部采用英文命名或者英文缩写，通篇文章的命名要有统一性，即使用全称就都用全称，使用缩写就都用缩写，避免出现中英文混合使用。另外命名的意义必须要清晰，要能够体现出名称的意义。

## 文件命名

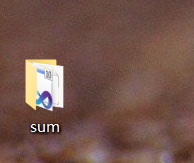
文件命名要全部使用小写，可以包括数字、字母下划线。可以接受的文件命名示例：*sum、find\_data、search\_student*。

图 1 文件命名

## 类型命名

类型的命名每个单词的首字母要大写，不包括数字下划线。可以接受的类型命名：*Student、Data、Person*。

## 变量命名

变量和数据成员的命名一律使用小写，单词之间使用下划线进行连接。可以接受的变量命名：*student\_name、number、amount*。

## 函数命名

函数的命名全部使用大写，可以使用数字下划线。

可以接受的函数命名：*SUM（）、SORT（）、FIND\_PERSON（）。*

## 其他命名

命名空间以小写字母命名. 最高级命名空间的名字取决于项目名称. 要注意避免嵌套命名空间的名字之间和常见的顶级命名空间的名字之间发生冲突.顶级命名空间的名称应当是项目名或者是该命名空间中的代码所属的团队的名字. 命名空间中的代码, 应当存放于和命名空间的名字匹配的文件夹或其子文件夹中.注意 [不使用缩写作为名称](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/naming/#general-naming-rules) 的规则同样适用于命名空间. 命名空间中的代码极少需要涉及命名空间的名称, 因此没有必要在命名空间中使用缩写。

枚举的命名应当和 [常量](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/naming/#constant-names) 或 [宏](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/naming/#macro-names) 一致。

# 代码文件

## 2.1头文件

头文件应该自给自足。也就是说用户和重构工具不需要为特别场合而包含额外的头文件。详言之，一个头文件要有#define保护。统统包含它所需要的其它头文件，也不要求定义任何特别 symbols.

在代码文件中每出现一次#include包含, 就会多一层依赖。比如，有A，B类型，各自有对应的.h文件和.cpp文件。

当A.cpp包含了A.h, A.cpp就依赖了A.h，表示为：A.cpp->A.h

这样，当A.h被修改的时候，A.cpp就需要重修编译.

如果有以下关系：

*B.cpp->B.h*

*B.h->A.h*

这表示，B.cpp 包含了B.h, B.h包含了A.h, 这个时候。B.cpp虽然没有直接包含A.h, 但也间接依赖于A.h。当A.h修改了，B.cpp也需要重修编译。

当在头文件中，出现不必要的包含，就会生成不必要的依赖，引起连锁反应，使得编译时间大大被拉长。

使用前置声明，而不是直接#include，可以显著地减少依赖数量。

## #define保护

所有的头文件，都应该使用#define来防止头文件被重复包含。命名的格式为

*\_<模块>\_<文件名>\_H\_*

更多时候模块名字都跟命名空间对应。比如:

*#ifndef \_\_GEO\_POINT\_H\_\_*

*#define \_\_GEO\_POINT\_H\_\_*

*namespace geo*

*{*

*class Point*

*{*

*.....*

*};*

*}*

*#endif*

并且，#define宏，的名字全部都为大写。不要出现大小写混杂的形式。

## 2.3#include 的顺序

C++代码使用#include来引入其它的模块的头文件。尽可能，按照模块的稳定性顺序来排列#include的顺序。按照稳定性从高到低排列。

比如:

*#include <map>*

*#include <vector>*

*#include <boost/noncopyable.hpp>*

*#include "cocos2d.h"*

*#include "json.h"*

*#include "FlaSDK.h"*

*#include "support/TimeUtils.h"*

*#include "Test.h"*

上面例子中。#include的顺序，分别是C++标准库，boost库，第三方库，我们自己写的跟工程无关的库，工程中比较基础的库，应用层面的文件。

但有一个例外，就是 .cpp中，对应的.h文件放在第一位。比如geo模块中的, Point.h 跟 Point.cpp文件，Point.cpp中的包含。

*#include "geo/Point.h"*

*#include <cmath>*

这里，将 #include "geo/Point.h"，放到第一位，之后按照上述原则来排列#include顺序。理由下一条规范来描述。

## 2.4前置声明

首先，只在头文件中使用引用或者指针，而不是使用值的，可以前置声明。而不是直接包含它的头文件。 比如：

*class Test : public Base*

*{*

*public:*

*void funA(const A& a);*

*void funB(const B\* b);*

*void funC(const space::C& c);*

*private:*

*D \_d;*

*};*

这里，我牵涉到几个其它类，Base, A, B, space::C（C 在命名空间space里面）, D。Base和D需要知道值，A, B, space::C只是引用和指针。所以Base, C的头文件需要包含。A, B，space::C只需要前置声明。

*#include "Base.h"*

*#include "D.h"*

*namespace space*

*{*

*class C;*

*}*

*class A;*

*class B;*

*class Test : public Base*

*{*

*public:*

*void funA(const A& a);*

*void funB(const B\* b);*

*void funC(const space::C& c);*

*private:*

*D \_d;*

*};*

注意命名空间里面的写法。

## 2.5impl手法

就是类里面包含实现类的指针。在cpp里面实现。

## 2.6#include中的头文件，尽量使用全路径，或者相对路径

路径的起始点，为工程文件代码文件的根目录。

比如：

*#include "ui/home/HomeLayer.h"*

*#include "ui/home/HomeCell.h"*

*#include "support/MathUtils.h"*

不要直接包含

*#include "HomeLayer.h"*

*#include "HomeCell.h"*

*#include "MathUtils.h"*

这样可以防止头文件重名，比如一个第三方库文件有可能就叫 MathUtils.h。

并且移植到其它平台，配置起来会更容易。这样做，还有个好处。就是只用一个简单脚本，或者一些简单工具。就可以分析出头文件的包含关系图，然后就很容易看出循环依赖。

# 3 类

作用域，表示某段代码或者数据的生效范围。作用域越大，修改代码时候影响区域也就越大，原则上，作用域越小越好。

## 3.1 让类的接口尽可能小

设计类的接口时，不要想着接口以后可能有用就先加上，而应该想着接口现在没有必要，就直接去掉。这里的接口，你可以当成类的成员函数。添加接口是很容易的，但是修改，去掉接口会会影响较大。

接口小，不单指成员函数的数量少，也指函数的作用域尽可能小。

比如：

*class Test*

*{*

*public:*

*void funA();*

*void funB();*

*void funC();*

*void funD();*

*};*

假如，funD 其实是可以使用 funA, funB, funC 来实现的。这个时候，funD，就不应该放到Test里面。可以将funD抽取出来。funD 只是一个封装函数，而不是最核心的。

*void Test\_funD(Test\* test);*

编写类的函数时候，一些辅助函数，优先采用 Test\_funD 这样的方式，将其放到.cpp中，使用匿名空间保护起来，外界就就不用知道此函数的存在，那些都只是实现细节。

当不能抽取独立于类的辅助函数，先将函数，变成private, 有必要再慢慢将其提出到public。 不要觉得这函数可能有用，一下子就写上一堆共有接口。

从作用域大小，来看

独立于类的函数，比类的成员函数要好

私有函数，比共有函数要好

非虚函数，比虚函数要好

## 3.2 声明顺序

类的成员函数或者成员变量，按照使用的重要程度，从高到低来排列。

比如，使用类的时候，用户更关注函数，而不是数据，所以成员函数应该放到成员变量之前。 再比如，使用类的时候，用户更关注共有函数，而不是私有函数，所以public，应该放在private前面。

具体规范

按照 public, protected, private 的顺序分块。那一块没有，就直接忽略。

每一块中，按照下面顺序排列

typedef，enum，struct，class 定义的嵌套类型

常量

构造函数

析构函数

成员函数,含静态成员函数

数据成员,含静态数据成员

.cpp 文件中，函数的实现尽可能给声明次序一致。

## 3.3 继承

优先使用组合，而不是继承。

继承主要用于两种场合：实现继承，子类继承了父类的实现代码。接口继承，子类仅仅继承父类的方法名称。

我们不提倡实现继承，实现继承的代码分散在子类跟父亲当中，理解起来变得很困难。通常实现继承都可以采用组合来替代。

规则：

继承应该都是 public

假如父类有虚函数，父类的析构函数为 virtual

假如子类覆写了父类的虚函数，应该显式写上 override

# 4 函数

## 4.1 编写短小的函数

函数尽可能的短小，凝聚，功能单一。

只要某段代码，可以用某句话来描述，尽可能将这代码抽取出来，作为独立的函数，就算那代码只有一行。最典型的就是C++中的max, 实现只有一句话。

*template <typename T>*

*inline T max(T a, T b)*

*{*

*return a > b ? a : b;*

*}*

将一段代码抽取出来，作为一个整体，一个抽象，就不用纠结在细节之中。

将一个长函数，切割成多个短小的函数。每个函数中使用的局部变量，作用域也会变小。

短小的函数，更容易复用，从一个文件搬到另一个文件也会更容易。

短小的函数，因为内存局部性，运行起来通常会更快。

短小的函数，也容易阅读，调试。

## 4.2 函数的参数可能少，原则上不超过5个

人脑短时记忆的数字是很有限的，大约可以记忆7个数字。有些人多些，有些人少些。我们这里取最少值，就是5个参数。

参数的个数，太多，就很容易混乱，记不住参数的意义。

同时参数的个数太多，很可能是因为这个函数做的事情有点多了。

可以通过很多手段来减少参数的个数。比如将函数分解，分解成多个短小的函数。或者将几个经常一起的参数，封装成一个类或者结构。

当然，每个规则都会有例外。比如设置一个矩阵的数值，二维矩阵本来就需要6个数字来表示，设置接口自然需要6个参数。

## 4.3 函数参数顺序

参数顺序，按照传入参数，传出参数，的顺序排列。不要使用可传入可传出的参数。

*bool loadFile(const std::string& filePath, ErrorCode\* code); //* 对

*bool loadFile(ErrorCode\* code, const std::string& filePath);* // 错

保持统一的顺序，使得他人容易记忆。

## 4.5 不建议使用函数的缺省参数

我们经常会通过查看现有的代码来了解如何使用函数的接口。缺省参数使得某些参数难以从调用方就完全清楚，需要去查看函数的接口，也就是完全了解某个接口，需要查看两个地方。

另外，缺省参数那个数值，其实是实现的一部分，写在头文件是不适当的。

缺省参数，其实可以通过将一个函数拆分成两个函数。实现放到.cpp中。

# 5 格式

## 5.1 每行代码不多于 80 个字符

从前的电脑终端，每行只可以显示 80 个字符。现在有更大更宽的显示屏，很多人会认为这条规则已经没有必要。但我们有充分的理由：

版本控制软件，或者编码过程中，经常需要在同一显示屏幕上，左右并排对比新旧两个文件。80 个字符的限制，使得两个文件都不会折行，对比起来更清晰。

当代码超过 3 层嵌套，代码行就很容易超过 80 个字符。这条规则防止我们嵌套太多层级，层级嵌套太深会使得代码难以读懂。

规则总会有例外。比如当你有些代码行，是 82 个字符，假如我们强制规定少于80字符，人为将一行容易读的代码拆分成两行代码，就太不人性化了。我们可以适当超过这个限制。

## 5.2 使用空格(Space)，而不是制表符(Tab)来缩进，每次缩进4个字符

代码编辑器，基本都可以设置将Tab转为空格，请打开这个设置。

制表符在每个软件中的显示，都会有所不同。有些软件中每个Tab缩进8个字符，有些软件每个Tab缩进4个字符，随着个人的设置不同而不同。只使用空格来缩进，保证团队中每个人，看同一份代码，格式不会乱掉。

## 5.3 指针符号\*，引用符号& 的位置，写在靠近类型的地方

*CCNode\* p = CCNode::create(); // (1)*

*CCNode \*p = CCNode::create(); // (2)*

也就说，上面两种写法。写成第(1)种。

我知道这个规定有很大的争议。指针符号到底靠近类型，还是靠近变量，这争论一直没有停过。其实两种写法都没有什么大问题，关键是统一。经考虑，感觉第1种写法更统一更合理。理由：

在类中连续写多个变量，通常会用 Tab 将变量对齐。( Tab 会转化成空格)。比如

*CCNode\* \_a;  
CCNode \_b;  
int \_c;*

当星号靠近类型而不是变量。\_a, \_b, \_c 等变量会很自然对齐。

而当星号靠近变量，如果不手动多按空格微调，会写成。

*CCNode \*\_a;  
CCNode \_b;  
int \_c;*

指针符号靠近类型，语法上更加统一。比如

*const char\* getTableName();  
static\_cast<CCLayer\*>(node);*

反对第一种写法的理由通常是：

假如某人连续定义多个变量，就会出错。

*int\* a, b, c;*

上面写法本身就有问题。应该每行定义一个变量, 并初始化。

*int\* a = nullptr;  
int\* b = nullptr;  
int\* c = nullptr;*

Xcode中，默认的语法提示，指针符号靠近变量，我再修改成靠近类型，比较麻烦。

这个有点道理。但我们也不能十分依赖工具。可以使用clang\_format等美化工具去辅助调整代码。

## 5.4 花括号位置

采用Allman风格，if, for, while，namespace, 命名空间等等的花括号，另起一行。例子

*for (auto i = 0; i < 100; i++)*

*{*

*printf("%d\n", i);*

*}*

这条规定，很可能又引起争议。很多人采用 K&R 风格，将上面代码写成

*for (auto i = 0; i < 100; i++) {*

*printf("%d\n", i);*

*}*

K&R风格在书籍印刷上会节省纸张。但在实际的代码中显得过于密集。Allman风格会更加清晰易读。当然，这理由带有很多主观因素。

5.5 if, for, while等语句就算只有一行，也强制使用花括号

永远不要省略花括号，不要写成：

*if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)*

*goto fail;*

需要写成：

*if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)*

*{*

*goto fail;*

*}*

省略花括号，以后修改代码，或者代码合并的时候，容易直接多写一行。如

*if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)*

*goto fail;*

*goto fail;*

就会引起错误。

# 6 其它

## 6.1 const的使用

我们建议，尽可能的多使用const。

C++中，const是个很重要的关键字，应用了const之后，就不可以随便改变变量的数值了，不小心改变了编译器会报错，就容易找到错误的地方。只要你觉得有不变的地方，就用const来修饰吧。比如:

想求圆的周长，需要用到Pi, Pi不会变的，加const，const double Pi = 3.1415926;

需要在函数中传引用，只读，不会变的，前面加const;

函数有个返回值，返回值是个引用，只读，不会变的，前面加const;

类中有个private数据，外界要以函数方式读取，不会变的，加const，这个时候const就是加在函数定义末尾。

const的位置：

*const int\* name; // 对（这样写，可读性更好）*

*int const\* name; // 错*

## 6.2 不要注释代码，代码不使用就直接删掉

有些人不习惯使用版本控制工具，某段代码不再使用了，他们会注释掉代码，而不是直接删除掉。他们的理由是，这段代码现在没有用，可能以后会有用，我注释了，以后真的再用的时候，就不用再写了。

注释掉的代码，放在源文件里面，会将正常的代码搞混乱。有个破窗理论，说假如一个窗户破了，不去管它，路人就会倾向敲烂其它的窗户。同样，假如你看到代码某个地方乱了，会觉得再搞的更乱也没有关系，就会越来越乱。