# Physika C++ 代码规范

杨升

2014 年 3 月 31 日

## 目录

# 概述

此代码规范为开发 **Physika** 的开发人员的代码规范编写提供参考依据和统一标准，本文 C++代码规范以 Google 的 **C++代码规范**为基础，结合 Physika 的具体特点做了一些相应的修改，并且添加了一些其他特性如效率之间的要求。本文档没有说明的地方，请参照 Google 的 **C++代码规范原文**。若有冲突，以本文档为准

# 1 头文件

通常每一个.cc/.cpp 文件都有一个对应的.h 文件，也有一些常见例外，如单元测试代和只包含 main()函数的.cc/.cpp 文件。

正常使用头文件可令代码在可读性、文件大小和性能上大为改观。

## 1.1 #define 保护

所有文件都应该使用#define 防止头文件呗多重包含，命名格式当时：<PROJECT>_<PATH>_<FILE>_H_为保证唯一性，头文件的命名应该依据所在项目源代码树的全路径。例如，我们的项目源文件全部在 Physika_Src 下，那么我们为 **Physika_Src/Physika_Core/Utilities/global_config.h** 做的头文件保护应该如下面一样书写：

```
#ifndef    PHYSIKA_CORE_UTILITIES_GLOBAL_CONFIG_H_
#define    PHYSIKA_CORE_UTILITIES_GLOBAL_CONFIG_H_
...
#endif  // PHYSIKA_CORE_UTILITIES_GLOBAL_CONFIG_H_
```

## 1.2 内联函数

一般内联函数声明允许编译器把函数直接展开而不是一般的函数调用，一般只在代码很小而且经常被调用的时候声明，10 行或者更少。尽管函数可能被声明为内联，但实际上有时候他比你更不是这样工作的，比如虚函数和递归函数，所以函数声明为内联函数的时候要谨慎，对那些展开代码很少，并且不包含循环或者判断条件的代码，才声明其为内联函数。

## 1.3 函数参数的顺序

定义函数式，参数顺序依次为：输入参数 输出参数

C/C++函数参数分别为输入参数，输出参数，和输入/输出参数三种，输入参数一般传值或传 const 引用，输出参数活输入/输出参数则是非-const 指针。对参数排序时，将只输入的

参数放在所有输出参数之前。尤其是不要仅仅因为是新加入的额参数，就把它放在最后；及时是新加的只输入参数也要放在输出参数之前。

这条规则并不需要严格遵守，输入/输出两用参数把事情变得辅助，为保持相关函数的一致性，有时候可以做出一定的变通。

## 1.4 #include 的路径及顺序

使用标准的头文件包含顺序可增强可读性，避免隐藏依赖：C 库，C++库，其他库的.h，本项目内的.h

项目内头文件应按照项目源代码目录树结构排列，避免使用 UNIX 特殊的快捷目录. (当前目录) 或.. (上级目录). **例如，Physika_Src/Physika_Core/Utilities/global_config.h 应该按如下方式包含:**

```
#include "Physika_Core/Utilities/global_config.h"
```

又如, dir/foo.cc 的主要作用是实现或测试 dir2/foo2.h 的功能， foo.cc 中包含头文件的次序如下：

1. dir2/foo2.h (优先位置，详情如下)
2. C 系统文件
3. C++ 系统文件
4. 其他库的.h 文件
5. 本项目内.h 文件

这种排序方式可有效减少隐藏依赖.。我们希望每一个头文件都是可被独立编译的 (yospaly 译注：即该头文件本身已包含所有必要的显式依赖),，最简单的方法是将其作为第一个.h 文件 #included 进对应的.cc/.cpp. dir/foo.cpp 和 dir2/foo2.h 通常位于同一目录下 (如 base/basictypes_unittest.cc 和 base/basictypes.h), 但也可以放在不同目录下。

按字母顺序对头文件包含进行二次排序是不错的主意 (yospaly 译注:：之前已经按头文件类别排过序了。举例来说, google-awesome-project/src/foo/internal/fooserver.cc 的包含次序如下：

```
#include "foo/public/fooserver.h"  // 优先位置.
#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>
#include "base/basictypes.h"        // 其他位置按头字母排序
#include "base/commandlineflags.h"
```

# 2 作用域

## 2.1 名字空间

鼓励在.cpp 文件内使用匿名名字空间。食欲具名的名字空间是，其名称可基于项目名称或者相对路径。不要使用 using 关键字。

具名的名字空间使用方式如下：用名字空间把文件包含，以及类的前置声明意外的整个源文件封装起来，以区别于其他名字空间，**目前我们拥有的命名空间只有 Physika(建议改为 physika)**

```
// . h ´
namespace Physika {

// 所有声明都置于命名空间中
// 不要使用缩进
class Vector {
public :
. . .
Vector ( ) ;
 } ;

} // namespace Physika
```

```
// . cc ´
namespace Physika {

// 函数定义都置于命名空间中
void Physika :: Foo ( ) {
. . .
}

} // namespace Physika
```

下面是一个比较全面的展示，前向声明 以及多个命名空间的写法：

```
#include "Physika_Core/Utilities/global_conflg.h"

DEFINE_bool(somefloag, false, "dummy flag");  //宏定义

class C;                        //全局名字空间中类 C 的前置声明

namespace a{class A;}           //a:A 的前置声明
```

```
namespace b{

code for b                          //b 中的代码

} // namspace b
```

不要在名字空间 std 内声明任何东西，包括标准库的类前置声明。 在 std 名字空间声明实体会导致不确定的问题。 比如不可移植。声明标准库下的实体， 需要包含对应的头文件。最好不要使用 “using” 关键字， 以保证名字空间下的所有名称都可以正常使用。

```
//禁止使用 污染名字空间
using namespace Physika;

//允许在 cpp 文件中
//允许在 h 文件的函数、方法或者类中使用 using 关键字
using namespace Physika::Matrix;

//允许在 cpp 文件
//允许在 h 文件的函数、方法或者类中使用名字空间别名
namespace tmp = ::Physika::Matrix;
```

## 2.2 局部变量

将函数变量尽可能置于最小作用域内， 并在变量声明时进行初始化。C++ 允许在函数的任何位置声明变量. 我们提倡在尽可能小的作用域中声明变量,离第一次使用越近越好. 这使得代码浏览者更容易定位变量声明的位置，了解变量的类型和初始值. 特别是，应使用初始化的方式替代声明再赋值，比如：

```
int i;
i = f();  // 坏 - 初始化和声明分离
int j = g(); // 好
```

注意, GCC 可正确实现了 for (int i = 0; i < 10; ++i) (i 的作用域仅限 for 循环内) 。 所以其他 for 循环中可以重新使用 i。 在 if 和 while 等语句中的作用域声明也是正确的。
Warning：如果变量是一个对象，每次进入作用域都要调用其构造函数，每次退出作用域都要调用其析构函数。

```
//低效
for (int i = 0; i < 1000000; ++i)
{
    Foo f;                  //构造析构分别调用 1000000 次
    f.DoSth(i);
}
```

```
Foo f;                          // 正确的用法
for (int i = 0; i < 1000000; ++i)
{
    f.DoSth(i);
}
```

## 2.3 静态和全局变量

禁止使用 class 类型的静态或者全局变量：它们会导致很难发现的 bug 和不确定的构造和析构函数调用顺序。

# 3 类

类是 C++ 中代码的基本单元。显然， 它们被广泛使用。 本节列举了在写一个类时的主要注意事项。

## 3.1 构造函数的职责

构造函数中只进行那些没什么意义的 (trivial, YuleFox 注: 简单初始化对于程序执行没有实际的逻辑意义, 因为成员变量 “有意义” 的值大多不在构造函数中确定) 初始化, 可能的话, 使用 Init() 方法集中初始化有意义的 (non-trivial) 数据

## 3.2 结构体 VS 类

仅当只有数据时使用 struct, 其它一概使用 class。

## 3.3 多重继承

真正需要用到多重实现继承的情况少之又少. 只在以下情况我们才允许多重继承: 最多只有一个基类是非抽象类; 其它基类都是以 Interface 为后缀的纯接口类。

## 3.4 存取控制

将所有数据成员声明为 private, 并根据需要提供相应的存取函数. 例如, 某个名为 foo_ 的变量, 其取值函数是 foo(). 还可能需要一个赋值函数 set_foo().一般在头文件中把存取函数定义成内联函数。

## 3.5 声明顺序

在类中使用特定的声明顺序: public: 在 private: 之前, 成员函数在数据成员 (变量) 前;类的访问控制区段的声明顺序依次为: public:, protected:, private:. 如果某区段没内容, 可以不声明.每个区段内的声明通常按以下顺序:

- typedefs 和枚举
- 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

宏 DISALLOW_COPY_AND_ASSIGN 的调用放在 private: 区段的末尾. 它通常是类的最后部分. 参考拷贝构造函数..cc 文件中函数的定义应尽可能和声明顺序一致.不要在类定义中内联大型函数. 通常, 只有那些没有特别意义或性能要求高, 并且是比较短小的函数才能被定义为内联函数. 更多细节参考内联函数。

## 3.6 编写简单函数

倾向编写简短, 凝练的函数.如果函数超过 40 行, 可以思索一下能不能在不影响程序结构的前提下对其进行分割。

# 4 其他 C++特性

## 4.1 引用参数

所有按引用传递的参数必须加上 const.

## 4.2 函数重载

仅在输入参数类型不同, 功能相同时使用重载函数 (含构造函数). 不要用函数重载模拟缺省函数参数.

## 4.3 缺省参数

我们强烈建议不使用缺省函数参数.

• 优点: 多数情况下, 你写的函数可能会用到很多的缺省值, 但偶尔你也会修改这些缺省值. 无须为了这些偶尔情况定义很多的函数, 用缺省参数就能很轻松的做到这点.

• 缺点: 大家通常都是通过查看别人的代码来推断如何使用 API. 用了缺省参数的代码更难维护, 从老代码复制粘贴而来的新代码可能只包含部分参数. 当缺省参数不适用于新代码时可能会导致重大问题.

• 结论: 我们规定所有参数必须明确指定, 迫使程序员理解 API 和各参数值的意义, 避免默默使用他们可能都还没意识到的缺省参数.

## 4.4 类型转换

使用 C++ 的类型转换, 如 static_cast<>(). 不要使用 int y = (int)x 或 int y = int(x) 等转换方式;

## 4.5 前置自增和自减

对于迭代器和其他模板对象使用前缀形式 (++i) 的自增, 自减运算符.

## 4.6 const 的使用

我们强烈建议你在任何可能的情况下都要使用 const.

## 4.7 预处理宏

使用宏时要非常谨慎, 尽量以内联函数, 枚举和常量代替之.

宏意味着你和编译器看到的代码是不同的. 这可能会导致异常行为, 尤其因为宏具有全局作用域.

## 4.8 NULL 和 0

整数用 0, 实数用 0.0, 指针用 NULL, 字符 (串) 用'\0'.

整数用 0, 实数用 0.0, 这一点是毫无争议的.

对于指针 (地址值), 到底是用 0 还是 NULL, Bjarne Stroustrup 建议使用最原始的 0. 我们建议使用看上去像是指针的 NULL, 事实上一些 C++ 编译器 (如 gcc 4.1.0) 对 NULL 进行了特殊的定义, 可以给出有用的警告信息, 尤其是 sizeof (NULL) 和 sizeof (0) 不相等的情况.

字符 (串) 用'\0', 不仅类型正确而且可读性好.

## 4.9 sizeof

尽可能用 sizeof (varname) 代替 sizeof (type).

# 5 命名约定

最重要的一致性规则是命名管理. 命名风格快速获知名字代表是什么东东: 类型? 变量? 函数? 常量? 宏...? 甚至不需要去查找类型声明. 我们大脑中的模式匹配引擎可以非常可靠的处理这些命名规则.

命名规则具有一定随意性, 但相比按个人喜好命名, 一致性更重, 所以不管你怎么想, 规则总归是规则。

## 5.1 通用命名规则

函数命名, 变量命名, 文件命名应具备描述性; 不要过度缩写. 类型和变量应该是名词, 函数名可以用 “命令性” 动词

### 5.1.1 如何命名

尽可能给出描述性的名称，不要节约行空间，让别人很快理解你的代码更重，好的命名风格：

```
int num_errors;                    // good
int num_completed_connections;     // good

int n;                            //Bad - Meaningless
int nerr;                         //Bad - ambiguous abbreviation
int n_comp_conns;                 //Bad - ambiguous abbreviation
```

类型和变量名一般为名词：如 FileOpener, num_errors.

函数名通常是指令性的，如 OpenFile(),set_num_errors().取值函数是个特例，函数名和他要取之的变量同名。

### 5.1.2 缩写

除非该缩写在其它地方都非常普遍, 否则不要使用. 例如

```
//Good
//These show prper names with no abbreviations
int num_dns_connections;    //dns 大家都知道是啥
int price_count_reader;     //Price Count 有意义

//Bad 有歧义
int wgc_connections;
```

```
int pc_reader;

int error_count;          // Good
int error_cnt;            //  Bad
```

## 5.2 文件命名

**文件名全部小写，可以包含下划线，Physika 所有文件名统一为下划线模式。C++文件已.cpp 结束，头文件以.h 结尾，特殊的文件格式如 CUDA 按照 CUDAC++格式命名,头文件以.cuh 结束，实现文件已.cu 结束。**

不要使用已经存在于 /usr/include 下的文件名 (yospaly 注: 即编译器搜索系统头文件的路径), 如 db.h.

通常应尽量让文件名更加明确. http_server_logs.h 就比 logs.h 要好. 定义类时文件名一般成对出现, 如 foo_bar.h 和 foo_bar.cc, 对应于类 FooBar.内联函数必须放在.h 文件中. 如果内联函数比较短, 就直接放在.h 中. 如果代码比较长, 可以放到以 -inl.h 结尾的文件中. 对于包含大量内联代码的类, 可以使用三个文件

```
global_config.h
global_config.cpp
global_config.cuh
global_config.cu
global_config-inl.h //内联代码文件
```

## 5.3 类型命名

类型名称的每个单词首字母均大写, 不包含下划线: MyExcitingClass, MyExcitingEnum. 所有类型命名——类, 结构体, 类型定义 (typedef), 枚举——均使用相同约定. 例如:

```
//classes and structs
class Physika{...
class PhysikaTester{...
struct PhysikaProperties{...

//typedefs
typdef Vector<float , 3> Vector3f

//enums
enum PhysikaErrors {...
```

## 5.4 变量命名/名字空间

变量名一律小写，单词之间用下划线连接，类的成员变量以下划线结尾，命名空间也是，现有工程里要做相应修改。

### 5.4.1 普通/类/结构体成员变量命名

Physika 所用变量命名为单词下划线模式，结构体命名同类命名，如：

```
string table_name;      //普通变量

string table_name_;     //类成员变量

string num_entries_;    //结构体成员变量

namespace physika
```

### 5.4.2 全局/常量命名/枚举/宏

**Physika 中用 g_作为前缀，区分局部变量，如 g_num，尽量少用。**

**Physika 中常量字母全部大写，单词用下划线连接，结尾不用_，如 PI_RECIPROCAL π 的倒数。**

**枚举和宏的命名同常量命名一致。**

## 5.5 函数命名

**Physika 命名方式： 不同于 Google C++风格， 常规函数和取值设置函数统一风格，均采用首个单词小写，后面单词首字母大写，若只有一个单词，小写即可；函数不用下划线。取值直接用变量名，也参照大小写，设值用 set+变量名：**

```
//普通函数
addTableEntry()
deleteUrl()

//类内函数 均采用一致命名法
class Test{
    public:
        int numEntries() const { return num_entriese_; }
        void setNumEntries(int num_entries) { num_entries_ num_entries; }
        void info();
    private:
```

```
        int num_entries_;

};
```

# 6 注释

注释虽然写起来很痛苦，但对保证代码可读性至关重要．下面的规则描述了如何注释以及在哪儿注释．当然也要记住：注释固然很重要，但最好的代码本身应该是自文档化．有意义的类型名和变量名，要远胜过要用注释解释的含糊不清的名字。

## 6.1 注释风格

使用 // 或 /* */，统一就好．

// 或 /* */ 都可以；但 // 更常用．要在如何注释及注释风格上确保统一，在 **Physika 中，文件注释使用/**/ 其他所有注释使用//**

## 6.2 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，**Physika 中的 MatrixMxN_test。cpp 文件注释模板如下，每次编辑时编辑相应位置，在 h 文件中简要描述类功能，在 cpp 文件中有必要描述实现细节和算法。现有项目有必要修改从 .h 直接复制到 .cpp 文件的注释。**

```
/*
 * @file matrixMxN_test.cpp
 * @brief Test the MatrixMxN class.
 * @author Fei Zhu
 *
 * This file is part of Physika, a versatile physics simulation library.
 * Copyright (C) 2013 Physika Group.
 *
 * This Source Code Form is subject to the terms of the GNU General Public
License v2.0.
 * If a copy of the GPL was not distributed with this file, you can obtain
one at:
 * http://www.gnu.org/licenses/gpl-2.0.html
 *
 */
```

## 6.3 类注释

每个类的定义都要附带一份注释，**描述类的功能和用法(目前执行并不好，需要修改)** 如果你觉得已经在文件顶部详细描述了该类，想直接简单的来上一句 “完整描述见文件顶部” 也不打紧，但务必确保有这类注释.如果类有任何同步前提，文档说明之. 如果该类的实例可被多线程访问，要特别注意文档说明多线程环境下相关的规则和常量使用.

```
//Iterates over the contents of a Gargantuantable. Sample usage:
// GargantuantableIterator * iter = table->NewIterator();
// for (iter->Seek("foo"); !iter->done()_; iter->Next()){
//  process(iter->key(), iter->value());
//}
//delete iter;
class GargantuanTableIterator{
    ...
}
```

## 6.4 函数注释

函数声明处注释描述函数功能；定义处描述函数实现．注释位于声明之前，对函数功能及用法进行描述．通常，注释不会描述函数如何工作．

那是函数定义部分的事情．

函数声明处注释的内容：

- 函数的输入输出．
- 对类成员函数而言：函数调用期间对象是否需要保持引用参数，是否会释放这些参数．
- 如果函数分配了空间，需要由调用者释放．
- 参数是否可以为 NULL．
- 是否存在函数使用上的性能隐患．
- 如果函数是可重入的，其同步前提是什么？

举例如下：

```
// Returns an iterator for this table.  It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
```

```
//    return iter;
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

每个函数定义时要用注释说明函数功能和实现要点．比如说说你用的编程技巧，实现的大致步骤，或解释如此实现的理由，为什么前半部分要加锁而后半部分不需要．不要从.h 文件或其他地方的函数声明处直接复制注释．简要重述函数功能是可以的，但注释重点要放在如何实现上．

## 6.5 变量注释

通常变量名足以很好说明变量用途。某些情况下，也需要额外的注释说明，这包括类数据成员/全局变量。

每个类数据成员 (也叫实例变量或成员变量) 都应该用注释说明用途．如果变量可以接受 NULL 或 -1 等警戒值，须加以说明．比如:

```
private:
// Keeps track of the total number of entries in the table.
// Used to ensure we do not go over the limit. -1 means
// that we don't yet know how many entries the table has.
int num_total_entries_;
```

## 6.6 实现注释

对于代码中巧妙、晦涩、有趣、重要的地方嫁衣注释

### 6.6.1 代码前注释

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++) {
x = (x << 8) + (*result)[i];
(*result)[i] = x >> 1;
```

## 6.6.2 行注释

比较隐晦的地方要在行尾加入注释. 在行尾空两格进行注释. 比如:

```
DoSomething();                  // Comment here so the comments line up.
DoSomethingElseThatIsLonger();  // Comment here so there are two spaces
between
// the code and the comment.
{ // One space before comment when opening a new scope is allowed,
// thus the comment lines up with the following comments and code.
DoSomethingElse();  // Two spaces before line comments normally.
}
DoSomething(); // For trailing block comments, one space is fine.
```

## 6.6.3 NULL,true/false1,2,3…;

向函数传入 NULL, 布尔值或整数时, 要注释说明含义, 或使用常量让代码望文知意. 例如, 对比:

```
//错误的实例
bool success = CalculateSomething(interesting_value,
10,
false,
NULL);  // What are these arguments??

//正确的做法
bool success = CalculateSomething(interesting_value,
10,     // Default base value.
false,  // Not the first time we're calling this.
NULL);  // No callback.

//替代做法
const int kDefaultBaseValue = 10;
const bool kFirstTimeCalling = false;
Callback *null_callback = NULL;
bool success = CalculateSomething(interesting_value,
kDefaultBaseValue,
kFirstTimeCalling,
null_callback);
```

### 6.6.4 不允许

注意永远不要用自然语言翻译代码作为注释. 要假设读代码的人 C++ 水平比你高, 即便他/她可能不知道你的用意。

## 6.7 TODO 注释

对那些临时的, 短期的解决方案, 或已经够好但仍不完美的代码使用 TODO 注释. TODO 注释要使用全大写的字符串 TODO, 在随后的圆括号里写上你的大名, 邮件地址, 或其它身份标识. 冒号是可选的. 主要目的是让添加注释的人 (也是可以请求提供更多细节的人) 可根据规范的 TODO 格式进行查找. 添加 TODO 注释并不意味着你要自己来修正.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations
```

## 译者 (YuleFox) 笔记

- 关于注释风格，很多 C++ 的 coders 更喜欢行注释, C coders 或许对块注释依然情有独钟, 或者在文件头大段大段的注释时使用块注释;
- 文件注释可以炫耀你的成就, 也是为了捅了篓子别人可以找你;
- 注释要言简意赅, 不要拖沓冗余, 复杂的东西简单化和简单的东西复杂化都是要被鄙视的;
- 对于 Chinese coders 来说, 用英文注释还是用中文注释, it is a problem, 但不管怎样, 注释是为了让别人看懂, 难道是为了炫耀编程语言之外的你的母语或外语水平吗;
- 注释不要太乱, 适当的缩进才会让人乐意看. 但也没有必要规定注释从第几列开始
- (我自己写代码的时候总喜欢这样), UNIX/LINUX 下还可以约定是使用 tab 还是 space, 个人倾向于 space;
- TODO 很不错, 有时候, 注释确实是为了标记一些未完成的或完成的不尽如人意的地方, 这样一搜索, 就知道还有哪些活要干, 日志都省了.

# 7 格式

代码风格和格式比较随意，但一个项目中所有人遵循同一个风格是非常容易的，个体未必同意下书每一处格式规则，但整个项目服从统一的变成风格很重要，只有这样灿能让所有人能很轻松的阅读和理解代码。**注意，以下都会规定所有的在 Physika 中使用的规则，请大家务必遵守。**

## 7.1 行长度

每一行代码字符数尽量不要超过 80，超过了就必须换行书写。

## 7.2 非 ASCII 字符

尽量不要使用非 ASCII 字符，使用时必须使用 UTF-8 编码。

## 7.3 使用空格而不是制表符

只使用空格，每次缩进 4 个空格，在 VS 开发时请将 tab 定位 4 个空格，设置编辑器将制表符转为空格。

## 7.4 函数声明和定义

注意以下几点:

- 返回值总是和函数名在同一行;
- 左圆括号总是和函数名在同一行;
- 函数名和左圆括号间没有空格;
- 圆括号与参数间没有空格;
- 实现时左大括号总是单独位于最后一个参数下一行;，除非只有一行代码 可在同一行。
- 右大括号总是单独位于函数最后一行;
- 函数声明和实现处的所有形参名称必须保持一致;
- 所有形参应尽可能对齐;
- 缺省缩进为 4 个空格;
- 换行后的参数保持 4 个空格的缩进;

如果函数声明成 const，关键字 const 应与最后一个参数位于同一行。

实例:

```
// Always have named parameters in interfaces.
class Shape
{
public:
    virtual void Rotate(double radians) = 0;
}
// Always have named parameters in the declaration.
class Circle : public Shape
{
public:
    virtual void Rotate(double radians);
}
```

```
// Comment out unused named parameters in definitions.
void Circle::Rotate(double /*radians*/) {}
// Bad - if someone wants to implement later, it's not clear what the
// variable means.
void Circle::Rotate(double) {}
```

**函数调用时：**

```
bool retval = DoSomething(argument1, argument2, argument3);
bool retval = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
bool retval = DoSomething(argument1,
                          argument2,
                          argument3,
                          argument4);
```

## 7.5 条件语句

**圆括号内不适用空格， 关键字 else 另起一行，大括号都另起并且独占一行，圆括号前后都需要一个空格，目前项目里需要做大量修改。条件语句单行不需要大括号，如果某个分支使用了大括号，所有分支都需要使用大括号。**

```
// Not allowed - curly on IF but not ELSE
if (condition)
{
    foo;
}
else
    bar;
// Not allowed - curly on ELSE but not IF
if (condition)
    foo;
else {
    bar;
}
// Curly braces around both IF and ELSE required because
// one of the clauses used braces.
if (condition)
{
    foo;
}
else {
    bar;
}
```

## 7.6 循环和开关选择语句

**switch 语句可以使用大括号分段. 空循环体应使用 {} 或 continue.**

**switch 语句中的 case 块不使用大括号，且 switch 大括号同函数括号一样，都另外独占一行。**

```
switch (var)
{
    case 0:      // 4 space indent
        ...      // 8 space indent
        break;
    case 1:
    ...
    break;
    default:
    assert(false);
    break;
}

while (condition)
{
// Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // Good - empty body.
while (condition) continue;  // Good - continue indicates no logic.
while (condition);  // Bad - looks like part of do/while loop.
```

## 7.7 指针和引用

**句点或箭头前后不要有空格. 指针/地址操作符 (*, &) 之后不能有空格。**

```
x = *p;
p = &x;
x = r.y;
x = r->y;
// These are fine, space preceding.
char *c;
const string &str;
// These are fine, space following.
char* c;    // but remember to do "char* c, *d, *e, ...;"!
const string& str;
char * c;  // Bad - spaces on both sides of *
const string & str;  // Bad - spaces on both sides of &
```

## 7.8 布尔表达式

如果一个布尔表达式超过标准行宽，断行方式要统一一下.
下例中，逻辑与（&&）操作符总位于行尾:

```
if (this_one_thing > this_other_thing &&
a_third_thing == a_fourth_thing &&
yet_another && last_one) {
...
}
```

## 7.9 函数返回值/变量及数组初始化。

返回值表达式中不适用圆括号，变量初始化使用()。

## 7.10 预处理指令

预处理指令不要缩进。

## 7.11 类格式

访问控制块的声明依次序是 public:, protected:, private:这三个控制不缩进，
所有基类名应在 80 列限制下尽量与子类名放在同一行.

- 关键词 public:, protected:, private: 不缩进
- 除第一个关键词 (一般是 public) 外，其他关键词前要空一行. 如果类比较小的话也可以不空.
- 这些关键词后不要保留空行.
- public 放在最前面，然后是 protected，最后是 private.
- 关于声明顺序的规则请参考声明顺序一节.

```
class MyClass : public OtherClass
{
public:      // Note the 1 space indent!
    MyClass();  // Regular 2 space indent.
    MyClass(int var);
    ~MyClass() {}
    void SomeFunction();
    void SomeFunctionThatDoesNothing() {}
```

```
    void set_some_var(int var) { some_var_ = var; }
    int some_var() const { return some_var_; }
private:
    bool SomeInternalFunction();
    int some_var_;
    int some_other_var_;
    DISALLOW_COPY_AND_ASSIGN(MyClass);
};
```

## 7.12 初始化列表

构造函数初始化列表放在同一行或按 8 格缩进并排几行.下面两种初始化列表方式都可以接受:

```
// When it all fits on one line:
MyClass::MyClass(int var) : some_var_(var), some_other_var_(var + 1) {}
// When it requires multiple lines, indent 8 spaces, putting the colon
on
// the first initializer line:
MyClass::MyClass(int var)
        : some_var_(var),             // 8 space indent
        some_other_var_(var + 1)
{  // lined up
    ...
    DoSomething();
    ...
}
```

## 7.13 命名空间格式

命名空间不需要添加任何缩进，且大括号位于命名空间的后面，添加一个空格隔开，其中的类以及函数都每行开始不需要缩进，现在的项目需要修改。

```
namespace physika {

void test()
{
    ...
}

}  //namespace physika
```

## 7.14 水平留白

水平留白，永远不要在行尾添加没有意义的留白。

添加冗余的留白会给其他人编辑时造成额外负担. 因此，行尾不要留空格. 如果确定一行代码已经修改完毕，将多余的空格去掉；或者在专门清理空格时去掉（确信没有其他人在处理). (yospaly 注：现在大部分代码编辑器稍加设置后，都支持自动删除行首/行尾空格，如果不支持，考虑换一款编辑器或 IDE)

```
//条件语句水平留白
if (b)
{         // Space after the keyword in conditions and loops.
} else {         // Spaces around else.
}
while (test) {}   // There is usually no space inside parentheses.
switch (i) {
for (int i = 0; i < 5; ++i) {
```

```
//操作算子留白
x = 0;             // Assignment operators always have spaces around
// them.
x = -5;            // No spaces separating unary operators and their
++x;               // arguments.
if (x && !y)
...
v = w * x + y / z;  // Binary operators usually have spaces around them,
v = w * (x + z);    // Parentheses should have no spaces inside them.
```

```
//模板定义留白
vector<string> x;          // No spaces inside the angle
y = static_cast<char*>(x);  // brackets (< and >), before
// <, or between >( in a cast.
vector<char *> x;          // Spaces between type and pointer are
// okay, but be consistent.
set<list<string>> x;       // Permitted in C++11 code.
```

## 7.15 垂直留白

垂直留白越少越好.

这不仅仅是规则而是原则问题了：不在万不得已，不要使用空行. 尤其是：两个函数定义之间的空行不要超过 2 行，函数体首尾不要留空行，函数体中也不要随意添加空行.基本原则是：同一屏可以显示的代码越多，越容易理解程序的控制流. 当然，过于密集的代码块和过于疏松的代码块同样难看，取决于你的判断. 但通常是垂直留白越少越好.

1 函数首尾不要有空行
2 代码块首位不要有空行
3 if-else 之间有空行可以接受。

译者 (YuleFox) 笔记

- 对于代码格式，因人，系统而异各有优缺点，但同一个项目中遵循同一标准还是有必要的;
- 行宽原则上不超过 80 列，把 22 寸的显示屏都占完，怎么也说不过去;
- 尽量不使用非 ASCII 字符，如果使用的话，参考 UTF-8 格式 (尤其是 UNIX/Linux 下, Windows 下可以考虑宽字符)，尽量不将字符串常量耦合到代码中，比如独立出资源文件，这不仅仅是风格问题了;
- UNIX/Linux 下无条件使用空格, MSVC 的话使用 Tab 也无可厚非;
- 函数参数，逻辑条件，初始化列表: 要么所有参数和函数名放在同一行，要么所有参数并排分行;
- 除函数定义的左大括号可以置于行首外，包括函数/类/结构体/枚举声明，各种语句的左大括号置于行尾，所有右大括号独立成行;
- ./-> 操作符前后不留空格, */& 不要前后都留，一个就可，靠左靠右依各人喜好;
- 预处理指令/命名空间不使用额外缩进，类/结构体/枚举/函数/语句使用缩进;
- 初始化用 = 还是 () 依个人喜好，统一就好;
- return 不要加 ();
- 水平/垂直留白不要滥用，怎么易读怎么来.
- 关于 UNIX/Linux 风格为什么要把左大括号置于行尾 (.cc/.cpp 文件的函数实现处，左大括号位于行首)，我的理解是代码看上去比较简约，想想行首除了函数体被一对大括号封在一起之外，只有右大括号的代码看上去确实也舒服; Windows 风格将左大括号置于行首的优点是匹配情况一目了然.

# 8 代码提交

每次提交代码时需填写相关日志. 如果是增加新的功能，一次提交必须是属于某个功能的代码，提交后的代码尽量可以通过编译. [示例]：

[BUG] 修正了...问题
BUG 现象：...
出现概率：...
BUG 来源：测试部，客户或者自测等
BUG 重现步骤：...
原因：...

[DEL] 删除了...代码。
原因：...

[CHG] 改进了...功能。

改进原因：...
(需要注明原有功能存在问题)

[TRP] 移植了...代码。
原代码路径：
原代码提供人：...
(需要注明移植代码出处和功能，原代码责任人)

[UDO] 恢复到(还原到、回退到)...r1234.

[MER] 合并目录 svn://192.0.0.140/DVR-DS9000/branches/v1.1.0 中 r8347-r8349 的代码。
(如果合并的代码比较少，建议把合并过来代码的日志也复制到下面，如下所示：)

如果是修正了历史遗留问题等重要更新，需在 [BUG] 或 [CHG] 标签之前增加【重要更新】.
【重要更新】
[BUG] 更新了(解决了).....问题。(重要更新时使用中文的中括号，以达到醒目的效果)