# Bigo CG C++ 开发规范
# 之
# 命名约定、代码风格 、工程实践

自律者得自由

# 版本历史

| 版本号 | 修改人 | 日期 |
| --- | --- | --- |
| V1.0 | **陈伟** | **2018.8.20** |
| V1.1 | **陈伟** | **2019.3.19** |
| V1.2 | **陈伟** | **2019.8.16** |
| | | |
| | | |
| | | |
| | | |

# 目录

# 概述

本文档所描述 C++ 代码规范为开发 Bigo CG 的开发人员的代码编写提供参考依据和统一标准。

本文档主要以 Google C++代码规范 (google coding style) 为基础，并针对 Bigo CG 的具体特点和开发人员的偏好做了一些相应的修改。此外，本文档还大量参考了如《Effective C++》系列、《Exceptional C++》系列、《代码大全》、《C++编程规范(101)》、《代码整洁之道》等工程实践方面的书籍，并添加了一些其他如效率、工程实践方面的要求。

本文档在主题划分上一定程度参考了 Google 代码规范，但并未按照 Google 代码规范的结构进行编排。本文档主要划分为三个主题：命名约定、代码风格、工程实践。

对于有经验并具有良好代码风格的开发人员，遵循一致的命名约定便可以开始编写高质量程序，所以本文档特意将“命名约定”编排为首个主题。关于“代码风格”和“工程实践”，“代码风格”本身实质上就可认作是一种“工程实践”，但本文档将“代码风格”着重于讨论关乎代码微观(局部)结构及风格的部分，阐述重要但对代码整体结构和性能影响较小的、关于代码细节实现的一些重要事项。此外，“代码风格”部分还讨论了关于项目源码结构的组织，这可认为是一种“项目风格”，实际上关乎的是项目架构，本文档并未深入讨论。在“工程实践”主题，本文档主要阐述一些重要的关乎代码整体结构的注意事项，这些规定至少关乎类层面的设计。此外，此主题还列举了关于设计的一些启发式原则，以及对 git 使用的一些最佳实践。

本文档仍然处于比较初步的阶段，需进一步完善。

本文档没有说明的地方，可以参照 Google 的 C++代码规范原文，或经进一步讨论后补充进本文档。若有冲突，请以本文档为准。

# 1 命名约定

命名是程序员最为纠结的一件事。

但在开发过程中，最重要的一致性规则就是命名管理。

命名风格的统一能够快速让开发者获知名字代表是什么含义：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，一致性更重要，所以不管你怎么想，规则总归是规则。

## 1.1 通用命名规则

### 1.1.1 命名原则

名字得能描述其要做什么，为什么存在，以及是如何工作的。所以应该尽可能选择能够表达意图、具有准确描述性的名称。

简洁固然重要，但名字应该更注重无歧义、清晰、明确。不要节约行空间，让开发者（包括自己）能第一时间理解代码更为重要。

开发者应该多试几种不同的名字，直至足以描述其含义。千万不要害怕在这上面花时间，以后阅读代码的人（包括自己）将会因此而受益。此外，一个描述性的名称甚至还能有助于你在心中理清模块的设计。良好的命名的确需要花费时间，但是从长远来看，利大于弊。

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

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

### 1.1.2 缩写

函数命名，变量命名， 文件命名应具备准确描述性，不要过度缩写。

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 proper names with no abbreviations
int num_dns_connections;    // dns is known to everyone
int price_count_reader;     // Price Count make sense

// Bad
// Ambiguity!
int wgc_connections;
int pc_reader;

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

# 1.2 文件及文件夹命名

## 1.2.1 文件后缀

Bigo CG 对源码文件的后缀名做如下规定：

| 文件类型 | 后缀名 |
| --- | --- |
| C++ 源码文件 | .cpp |
| C++ 头文件 | .hpp |
| C 纯头文件（如接口文件） | .h |
| 头文件实现文件（如模板实现等） | .inl |

## 1.2.2 文件命名

Bigo CG 规定源文件名字使用小写+下划线格式，特殊名词可以采用全大写形式，文件名应无歧义、明确并具有准确描述性。

不要使用已经存在于编译器搜索系统头文件的路径下的文件名。

通常应尽量让文件名更加明确。http_server_logs.hpp 就比 logs.hpp 要好。定义类时文件名一般成对出现，如 sparse_matrix.hpp 和 sparse_matrix.cpp，对应于类 SparseMatrix。

```
file: sparse_matrix.hpp
```

```
// 类为 SparseMatrix，文件名为 sparse_matrix.hpp
class SparseMatrix
{
}
```

### 1.2.3 文件编码

Bigo CG 规定所有源文件都必须使用 UTF-8 编码。当前大多数编辑器都使用 UTF-8 作为默认编码，而且存储引擎一般也使用 UTF-8 编码，能有效避免乱码问题。

### 1.2.4 文件夹命名

Bigo CG 规定文件夹名字为首字母为大写的形式，多个 word 则首字母都为大写，可以采用下划线连接，也可以不采用下划线连接，但项目应保持一致。即：

```
CG_Src/Sparse_Matrix/xxxxx.hpp
//or
CGSrc/SparseMatrix/xxxxx.hpp
```

## 1.3 类型命名(Class/Struct/Enum/Typedef/Namespace)

Class 名称的每个单词首字母均大写，不包含下划线。

类型名一般为具有准确描述性的名词，而非使用动词。

所有类型命名（类，结构体，类型别名 typedef，枚举），均使用相同约定。

Bigo CG 规定所有源码的顶层命名空间为 namespace xxxx。（依具体项目而定）

```
// namespace
namespace bg{...

// classes and structs
class CGTester{...
struct CGProperties{...

// typedefs/using alias
using Vector3f = vector<float , 3>;

// enums
enum Errors {...
```

## 1.4 变量命名

### 1.4.1 普通/类/结构体成员变量命名

变量命名同类型命名，一般为具有准确描述性的名词，而非使用动词。

变量名一律小写，单词之间用下划线连接，类的成员变量以 “m_” 前缀开始。

结构体的成员变量命名同类命名。

```
string table_name;       // 普通变量

string m_table_name;     // 类成员变量

string m_num_entries;    // 结构体成员变量
```

对于有特殊设计的类，如 Vec3 包含变量 x,y,z 并且特意设计为 public，,其命名经慎重考虑可以不遵循上述规则。

```
class Vec3
{
public:
    double x, y, z;
     ...
};
```

参考资料：《代码大全》第 11 章：变量名的力量

### 1.4.2 全局/常量命名/枚举/宏

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

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

枚举和宏的命名同常量命名一致。类中的常量命名和普通常量命名一致。

```
string g_num;                //全局变量

const double PI = 3.14;      //常量
```

```
class MyClass
{
    static const int PI = 3.14; //类常量
};

enum class Type             //枚举成员
{
    FIRST_TYPE,
    SECOND_TYPE
};

#define MY_MACRO           //宏
```

## 1.5 函数命名

Bigo CG 对于函数的命名方式不同于 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。函数命名不用下划线。

一般而言，函数名通常是指令性的，首个单词一般为动词，如设值函数为 set+变量名，同样参照大小写。取值函数则直接用变量名。

函数名应该准确描述函数所完成的功能，切忌挂羊头、卖狗肉！

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

// 类内函数均采用一致命名法
class Test{
public:
    int  numEntries() const;  // 取值函数直接用变量名
    void setNumEntries(int num_entries) ;
    void info();

private:
    int m_num_entries;
};
```

# 2 代码风格

## 2.1 源码结构

### 2.1.1 文件夹

项目源码文件的组织应该具有合理层次结构，方便开发者理解项目框架，以及对源码进行快速定位。

Bigo CG 规定，功能内聚的源码实现文件，应该位于相同文件夹。顶层模块源码和底层模块源码应该位于不同的文件夹层次。

### 2.1.2 文件

Bigo CG 推荐无论是头文件还是源码文件， 应该尽量只包含一个类的声明、定义和实现（即采用类似于 JAVA 的方式），嵌套类和 POD 则可以除外。

## 2.2 头文件

### 2.2.1 头文件保护

方案 1：

所有的头文件都应该使用 #define 防止头文件被多重包含。为保证唯一性，头文件保护的命名格式应该依据所在项目源代码树的全路径： <PROJECT>_<PATH>_<FILE>_H_。例如，“Bigo_CG_Src/Core/Utilities/global_config.h” 的头文件保护应该如下面一样书写：

```
#ifndef    BIGO_CG_SRC_CORE_UTILITIES_GLOBAL_CONFIG_H_
#define    BIGO_CG_SRC_CORE_UTILITIES_GLOBAL_CONFIG_H_
...
#endif  // BIGO_CG_SRC_CORE_UTILITIES_GLOBAL_CONFIG_H_
```

方案 2：使用 #pragma once。相同的效果，好处是不需要考虑宏定义的唯一性，但可能存在编译性能问题。

```
#pragma once

...
```

## 2.2.2 #include 的顺序及路径

使用标准的头文件包含顺序可增强可读性：

① 与 .cpp 对应的.hpp（如 a.cpp 对应 a.hpp）
② 空行
③ C 库
④ C++库
⑤ 空行
⑥ 其他库的 .hpp/.h
⑦ 本项目内其他的 .hpp/.h

也可以把与 .cpp 对应的.hpp 放在最后，但请至少在同一模块内保持一致。

Bigo CG 严禁底层头文件 #inlucde 顶层模块的头文件，否则容易产生循环依赖。

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)。最好不使用同文件夹下的相对目录。 包含相同文件夹下的头文件最好不需要从 include 根目录写起。

```
//file: a.cpp
#include "a.hpp" //注意 include 和 "a.hpp" 之间的空格

#include <vector>
#include <list>

#include "other_lib/xxx.h"
#include "Core/Utilities/global_config.h" //注意使用绝对路径，不用 . / ..
```

## 2.2.3 前置声明

头文件应该自给自足，确保所编写的每个头文件都能够独自进行编译。

Bigo CG 建议只 #include 那些必要的头文件，尽量使用前置声明。

假设头文件 a.hpp 定义类 A，b.hpp 定义类 B，并且类 A 依赖于类 B，则在下述情形应使用 #include：

```
//file: a.hpp
```

```
#include <b.hpp>
/*
如果
① B 是 A 的父类。
② A 包含一个 B 对象（非指针）。
*/
class A: B {};
class A{B b;};
```

否则，应该使用前置声明。

```
//file: a.hpp
class B;
/*
如果
① A 的成员函数声明使用 A 的指针/引用/对象。
② A 包含一个 B 的一个指针，包括智能指针 shared_ptr<B>, unique_ptr<A>。
*/
class A
{
    B * func(B * b_ptr, B b_obj);
    B * b1;
    unique_ptr<B> * b2;
};
```

最后，如果 A 和 B 不相关，或者仅仅将 B 声明为 A 的友元，则既不需要 #include，也不需要前置声明。而对于 std 以及其他库中的头文件，直接使用 #include，或 #include<iosfwd>。包含其他库的头文件不会产生循环依赖问题。

注意，google 代码规范给出的建议是避免尽可能使用前置声明，需要时就 #include。其意见倾向与本文档稍微相左，但本质上不存在原则性矛盾，开发者应慎重考虑。

参考资料：http://www.cplusplus.com/forum/articles/10627/

http://google.github.io/styleguide/cppguide.html#Namespaces

《C++编程规范（101）》条款 23：头文件应该自给自足

## 2.3 作用域

### 2.3.1 命名空间

Bigo CG 的所有源码的顶层命名空间为 namespace bg。(具体项目具体而定)

命名空间使用方式如下：用命名空间把除文件包含，以及 bg 外部类的前置声明以外的整个源文件封装起来，以区别于其他命名空间。

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。

```
// .hpp
#include <xxx.hpp>

namespace bg{ // 大括号位于命名空间后面，不单起一行

// 所有 bg 内部类的声明都置于命名空间中

// 不要使用缩进
class Foo {} ;

} // namespace bg
```

```
// . cpp
namespace bg {

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

} // namespace bg
```

在 bg 中使用其他命名空间中的内容时，不要在头文件使用 using 关键字直接引入整个命名空间，即在头文件不要使用 using 指示（如 using namespace std)。但为代码简洁性和可读性，可以适度使用 using 声明。在实现文件（.cpp）中，可以以谨慎的态度使用 using 指示。对于 std::swap()，如果类已经自定义 swap，则应使用 using std::swap，避免显式调用 std::swap() 而忽略自定义版本。

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，如有必要请 #include <iosfwd>。 在 std 名字空间声明实体为未定义行为。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。

禁止内联命名空间，其会污染整个当前的名字空间。

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

// 显式指定命名空间
std::vector<int> data;

// 允许在 cpp 文件中
// 允许在 hpp 文件的函数、方法或者类中使用 using 声明
using std::vector;
using std::swap;

// 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 = ::bg::Matrix;
```

参考资料：

《Effective C++》条款 25：考虑写出一个不抛异常的 swap 函数

《C++编程规范(101)》条款 59:不要在头文件或者#include 之前编写名字空间 using

## 2.3.2 局部变量

将函数变量尽可能置于最小作用域内（最短声明周期），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;
doOtherThing();
...
j = f();  // 更坏 - 早早声明，却在很晚使用。

int k = f(); // 好
```

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，用户需要衡量效率与可读性而做出选择。

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

参考资料：《Effective C++》条款 26：尽可能延后变量定义式出现的时间

《代码大全》第 10 章：使用变量的一般事项

## 2.3.3 静态和全局变量

Bigo CG 不推荐使用 class 类型的 non-local 静态及全局变量。此处，local 是指位于函数作用域，其他作用域如全局作用域、file 作用域、类作用域、namespace 作用域都为 non-local。

不同编译单元的 non-local static 对象的初始化顺序是不确定的。如果一个 non-local static 对象的初始化依赖于另一个 non-local static 对象，则会导致很难发现的 bug。

全局变量几乎不具有封装性，虽然其能够提供一个全局的访问点，在代码实现时获取此变量较为便利。但实际上会带来紧耦合问题，程序的可读性也会较差。推荐使用单例模式或者工厂模式管理或替换全局变量。

（备注：多单例模式，如果析构时仍有依赖，仍然可能存在顺序问题。）

参考资料：《Effective C++》条款 04：确定对象被使用前已先被初始化

## 2.4 类

### 2.4.1 类声明与成员顺序

所有基类名应在 80 列限制下尽量与子类名放在同一行。

在类中按照存取控制权限声明顺序，即

- public
- protected
- private

三个控制关键字不进行缩进，除第一个关键词（一般为 public）外，其他关键词前要空一行。非特殊要求，这些关键词后不要保留空行。

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

为区别成员函数和数据成员，关键字（如 private）可以多次使用。

构造及控制复制成员 和 其他成员函数应该留有空行。其他成员函数按照功能划分应该适当添加空行以增加可读性。

在声明函数成员和数据成员时，Bigo CG 强调需要精心组织成员之间的顺序。一般而言，功能内聚的成员函数和数据成员应该同样聚集，其逻辑是收敛而非发散的。类定义向用户提供接口，清晰的结构有有利于代码阅读者理解，增加可读性。不同功能块的函数和数据，应以空行区分。如果数据成员和函数过多，则应该考虑分割。

“.cpp”文件中函数的定义应尽可能和声明顺序一致。

```
class MyClass : public OtherClass
{               // 大括号单起一行
public:         // 关键字不要缩进
    MyClass();
    MyClass(int var);
```

```
    ~MyClass() {}
    //空行
    void someFunction();
    void someFunctionThatDoesNothing() {}
    void set_some_var(int var) { m_some_var = var; }
    int some_var() const { return m_some_var; }
    //空行
private:
    bool someInternalFunction();
    //空行
private: //多次使用 private, 区别成员函数和成员变量
    int m_some_var;
    int m_some_other_var;
};
```

## 2.4.2 构造函数

当构造函数仅有一个参数时，除非特意设计，务必将“单参量构造函数”声明为 explicit，禁用隐式类型转换。拷贝和移动构造函数不应当被标记为 explicit，因为它们并不执行类型转换。

禁止在构造函数中调用虚函数，因为这类调用是不会重定向到子类的虚函数实现。即对象在构造期间不具有多态特性。

参考资料：《More Effective C++》条款 05：对定制的“类型转换函数”保持警觉
《Effective C++》条款 09：绝不在构造和析构过程中调用 virtual 函数

## 2.4.3 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) : m_some_var(var), m_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)
```

```
        : m_some_var(var)               // 8 space indent
        , m_some_other_var(var + 1)     // lined up
{
    ...
    DoSomething();
    ...
}
```

如果类数据成员非常多，并且存在多个构造函数。此时可以考虑提取出 init() 方法进行集中初始化，以供各个构造函数调用。也可以考虑使用 C++11 提供的委托构造函数功能。

## 2.4.4 delete 编译器自动生成函数

在不同情形下，编译器总共能够自动生成 6 类函数：

- 默认构造函数
- 析构函数
- 拷贝构造函数
- 移动构造函数
- 赋值操作符
- 移动赋值操作符

开发者在编写类时应尤其注意这 6 个函数，若要禁止编译器自动生成，请使用 C++ 11 提供的 delete 函数特性，而非将这些函数声明为 private，然后不提供定义。

```
class MyClass
{
public:
    MyClass(const & rhs) = delete; //禁止编译器自动生成拷贝构造函数
};
```

参考资料：

《Effective C++》条款 05：了解 C++默默编写并调用哪些函数

《Effective C++》条款 06：若不想使用编译器自动生成的函数，就该明确拒绝

《Effective Modern C++》条款 11：优先选用删除函数，而非 private 未定义函数

《Effective Modern C++》条款 17：理解特种成员函数的生成机制

## 2.4.5 虚函数

覆写基类的虚函数，请显式添加 override 关键字进行声明。

```
class BaseClass
{
public:
    BaseClass();
    virtual ~BaseClass ();

    virtual void virtualFunc();
};

class DerivedClass: public BaseClass
{
public:
    void virtualFunc() override;
};
```

参考资料：

《Effective Modern C++》条款 12：为意在改写的虚函数添加 override 声明

## 2.4.6 访问权限控制

将所有数据成员声明为 private/protected，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 setFoo()。一般在头文件中把存取函数定义成内联函数。

参考资料：《Effective C++》条款 22：将成员变量声明为 private

## 2.4.7 隐式类型转换

除普通的数值类型转换外（如 float → double），C++还存在两种形式的类型转换：单参量构造函数和隐式类型转换运算符，如 operator double () const。除非有意设计，并且经过慎重思考，否则应该将“单参量构造函数”声明为 explicit，且不提供“隐式类型转换运算符”。否则发生诸多隐式类型转换往往并非你预期的，并且难以调试。

可以考虑提供类似 asDouble() 的成员函数，以显式的进行调用。

参考资料：《More Effective C++》 条款 5：谨慎定义类型转换函数

### 2.4.8 友元

不同于 Google 代码规范，Bigo CG 不建议使用友元。

友元是一种比继承更为耦合的关系，朋友带来的麻烦往往多于好处。

### 2.4.9 类 VS 结构体

仅当只有数据时使用 struct，即 POD (plain of data)，其它一概使用 class。

### 2.4.10 枚举类

当定义枚举类型时，请使用 enum class 而不是 enum。

```
// Bad
enum Type
{
    FIRST_TYPE,
    SECOND_TYPE
};

// Good
enum class Type
{
    FIRST_TYPE,
    SECOND_TYPE
};
```

参考资料：

《Effective Modern C++》条款 10：优先选用限定作用域的枚举型别，而非不限作用域的枚举型别

## 2.5 函数

### 2.5.1 函数编写原则

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。

为提高可读性，函数中的语句及其所调用的其他函数都应该处于同一逻辑层次。

参考资料：

《代码整洁之道》第三章：函数（短小、只做一件事、每个函数一个抽象层级）

## 2.5.2 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
- 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
- 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。

```
// 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);
```

### 2.5.3 函数参数顺序

定义函数时，函数参数顺序依次为：

① 输入参数

② 输入/输出参数

③ 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，输入参数传值一般使用 const 引用，输出参数或输入/输出参数则是非-const 指针或引用。可以对函数参数恰当命名以增加可读性。如 output_para。

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参数，就把它放在最后。即使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。

```
void func(const vector<int> &input_data, vector<double> &output_data)
```

### 2.5.4 引用实参传递

输入参数除 C++ 内置类型（数值类型+指针类型+枚举类型）外，所有自定义类型对象如果不需要修改，应该按 const 引用/指针传递，而不应该按值传递。

参考资料：

《Effective C++》条款 20：宁以 pass-by-reference-to-const 替换 pass-by-value

## 2.5.5 函数重载

若要使用函数重载，则必须能让读者一看调用点就胸有成竹，而不用花心思猜测调用的重载函数到底是哪一种。这一规则也适用于构造函数。

如果打算重载一个函数，可以试试改在函数名里加上参数信息。 例如，用 appendString() 和 appendInt() 等，而不是一口气重载多个 append()。 如果重载函数的目的是为了支持不同数量的同一类型参数，则优先考虑使用 std::vector 以便使用者可以用 列表初始化 指定参数。

## 2.5.6 缺省(默认)参数

方案 1：

禁止在虚函数中使用缺省参数。

对于非虚函数，如果缺省参数对可读性的提升远远超过了以下提及的缺点的话，可以使用缺省参数。 如果仍有疑惑，就使用函数重载。

方案 2：

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。

参考资料：《Effective C++》条款 37：绝不重新定义继承而来的缺省参数值

## 2.5.7 函数返回类型后置语法

只有在常规写法（返回类型前置）不便于书写或不便于阅读时使用返回类型后置语法。

在大部分情况下，应当继续使用以往的函数声明写法，即将返回类型置于函数名前。只有在必需的时候（如 Lambda 表达式）或者使用后置语法能够简化书写并且提高易读性的

时候才使用新的返回类型后置语法。但是后一种情况一般来说是很少见的，大部分时候都出现在相当复杂的模板代码中，而多数情况下不鼓励写这样复杂的模板代码。

## 2.5.8 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。

内联函数不一定能带来性能提升，反而可能降低性能，也不大可能成为性能瓶颈。尽管函数可能被声明为内联，但实际上只是对编译器的建议，虚函数和递归函数只能在部分情况下内联或不能内联。所以函数声明为内联函数的时候要谨慎，只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。

为保证类定义的整洁性，不应将内联函数实现在类定义的内部，而应该实现于同一头文件的类定义的外部。即

```
//bad
class B
{
    //note: 在类内定义函数，默认内联，可以不加 inline 关键字。
    inline void func(){ doSomething(); }
};

//good
class B
{
    inline void func();
};

inline void B::func()
{
    doSomething();
}
```

参考资料：《Effective C++》条款 30：透彻了解 inlining 的里里外外

《More Effective C++》条款 16：谨记 80-20 法则

《C++编程规范（101）》条款 09：不要进行不成熟的优化

## 2.6 语句

### 2.6.1 条件语句

圆括号内不使用空格， 关键字 else 另起一行，大括号都另起并且独占一行。

条件语句如果某个分支使用了大括号，所有分支都需要使用大括号。

对于短小的单行条件语句，最好不使用大括号。如果条件判断非常长，需写成多行，为避免“头重脚轻”，推荐使用大括号。如果单行语句特别长，需要折断写成多行，则也应该使用大括号。

编写条件语句时，请将正常情况的处理放在 if 后面而不要放在 else 后面，保证正常情况的执行路径在代码中是清晰的。这对可读性和代码性能来说非常重要。

```
// 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)
{
   normalSituation();
}
else
{
    annormalSituation();
}
```

参考资料：《代码大全》第 15 章：使用条件语句

## 2.6.2 开关选择语句

switch 语句可以使用大括号分段。

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 的分支如果不是没有语句的 case，都必须包含 break。

switch 语句必须有 default 分支，并且在不可能到达的分支使用 assert，或者进行适当的错误处理。

switch 语句存在维护困难，在新添代码切勿忽略对代码进行相应修改。

一般而言，使用 switch 说明设计上存在缺陷，可以考虑以某种多态机制进行替换。

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

## 2.6.3 空循环语句

空循环体应使用 {} 或 continue。

```
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.
```

### 2.6.4 避免深层嵌套

无论是对于条件语句还是循环语句，避免使用超过 3 到 4 层的嵌套，否则将会严重降低代码的可理解性。一般可以对代码重构，简化代码结果，或者将深层嵌套提取为函数。

```
// bad
if(condition_1)
{
      if(condition_2)
      {
           if(condition_3){}
      }
}

// good
if(condition_1)
{
      if(!condition_2) break; // 用 break 剪枝
      if(!condition_3) break;
}
```

参考资料：《代码大全》第 19.4 节：驯服危险的深层嵌套

## 2.7 变量&表达式

### 2.7.1 auto 变量定义

现阶段 Bigo CG 不对 auto 的使用做强制规定。

我们推荐在循环中使用 auto 来声明迭代器变量，以避免代码过于冗长。

当前，C++领域专家推荐使用 auto 来统一进行所有变量的声明与定义，即所谓的 AAA style（almost away auto）。但考虑到 auto 一定程度上影响代码的可读性，并且当前开发环境对 auto 的实时解析能力有限，请尽量在不影响代码可读性情形下使用 auto 自动类型推导。

禁止使用 auto () {} 来定义变量，必须使用等号。如下

```
// bad
auto count(3);
auto count{3};

// good
```

```
Auto count = 3;
```

参考资料：

《Effective Modern C++》条款 02：理解 auto 型别推导

《Effective Modern C++》条款 05：条款 5：优先选用 auto，而非显式型别声明

https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

## 2.7.2 整型变量定义

Bigo CG 强烈推荐对于用于运算的整型变量，使用 int8_t, int16_t, int32_t, int64_t 等定宽类型，而非 char, short, int, long, long long 等类型。因为 C++ 标准只规定这些类型的最小位数，不保证内置类型一定具有 16/32/64 位。

特别的，C++ 未规定 char 为有符号类型或无符号类型。一般而言，在 PC 端 char 为有符号类型，但在 android 端则为无符号类型。在进行数值运算或者类型转换，容易发生溢出，并且往往出乎开发者所预期。因此考虑平台不一致性问题，不允许使用 char 类型用于数值运算。

参考资料：《C++ Primer》P30-P32

## 2.7.3 无符号类型与有符号类型

在定义函数参数和定义变量时，如果参数不为负数，则定义为无符号类型，如果参数可以为负数，则定义为有符号类型。从语义逻辑上看，这是一种貌似合理的做法。

我们经常使用无符号类型来表示不为负数的参数（如 width，height，size 等），其能够带来两点好处：第一，相对于有符号类型，其更不容易溢出；第二，对于函数参数，将范围信息编码到类型，能够给接口调用者更好的提示。

但是从代码行为上看，使用无符号类型时，容易产生下溢问题，往往较为隐晦难查，而且和有符号类型混合运算时，需要进行类型提升，其规则更为晦涩难懂，而且会和平台相关。如下代码：

```
for(uint32_t n = 10; n >= 0; --n) //bad, loops infinitely

std::vector<int> data; //empty vector
for(uint32_t n = data.size() - 1 ; n >= 0; --n)
// access violation and infinite loop
```

```
unsigned short us = 10;
int i = -5;
if(i > us) printf("whoops!\n");
// if sizeof(unsigned short) == sizeof(int), i is promoted to
// unsigned short

uint32_t x = 2;
int32_t radius = 3;
uint32_t down_x = std::max<uint32_t>(x - radius, 0);
//down_x not equal 0, but a large number
```

此外，如果无符号类型需要和 float 混合运算，x86 平台没有直接指令能够将无符号类型转换成 float，可能会有一定的效率损失。

事实上，回头看无符号类型所带来的两点好处。使用无符号类型减少了上溢的风险，但其实增加了下溢的风险。将函数参数定义成无符号类型，并不能完全阻止用户传入一个负数（虽然编译器会给出警告），函数调用者更应该通过参数含义明白参数的有效范围。如果将函数参数定义成无符号类型， 但在函数内部实现有大量混合无符号类型和有符号类型的二元运算，此时会进行类型提升，但很多时候并不会如编码者所预期。而编码者往往又会倾向于忽略编译器产生的警告，通过大量使用 static_cast 进行显式转换来避免这种混合运算可能又得不偿失。此时，从语义上看，无符号类型是合理的，但从代码行为（混合运算）上看，直接使用有符号类型可能是更为合适的选择。

此外注意，C++ 数组使用有符号类型作为索引，而 C++ STL 使用无符号类型(std::size_t)作为容器的索引（诸多 C++标准委员会成员认为这是一个设计缺陷）。部分编程语言，如 java， 则干脆没有无符号类型。

关于此点，《Google C++ 代码规范》亦有讨论。

总结：

注意无符号整数的下溢问题，和无符号有符号类型混合运算产生的类型提升问题。

算术运算不要混用有符号类型和无符号类型，最好使用有符号类型。

对于位运算和取模运算，使用无符号类型是合理的。

如果不涉及算术运算（如只用来遍历），使用无符号类型可能是合理的（如为适应 STL）。

但如果涉及算术运算（整型提升），直接选择使用有符号数可能更为合适。

参考资料：

https://softwareengineering.stackexchange.com/questions/97541/what-are-the-best-practices-regarding-unsigned-ints

https://stackoverflow.com/questions/22587451/c-c-use-of-int-or-unsigned-int

http://c-faq.com/expr/preservingrules.html

https://coolshell.cn/articles/11466.html

《C++ Primer》P32-P35， p142-p143 关于类型转换与整型提升

《C++ Core Guidelines》 ES.100 Don’t mix signed and unsigned arithmetic

ES.101 Use unsigned types for bit manipulation

ES.102 Use signed types for arithmetic

ES.103 Don’t overflow

ES.104 Don’t underflow

ES.106 Don’t try to avoid negative values by using unsigned

## 2.7.4 指针和引用声明

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。这一点主要是考虑一行定义多个指针或者引用变量时会出现问题。但一般而言，Bigo CG 推荐，一行只定义一个变量。

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

// These are fine, space preceding.
char *c;
const string &str;

// Bad
char* c;
const string& str;

// Not That Good
char * c;
const string & str;
```

## 2.7.5 右值引用

在定义移动构造函数与移动赋值操作时使用右值引用，不要使用 std::forward，应该使用 std::move 来表示将值从一个对象移动而不是复制到另一个对象。

如果是自己实现完美转发，则可以使用 std::forward。

参考资料:

《Effective Modern C++》条款 23：理解 std::move 和 std::forward

《Effective Modern C++》条款 24：区分万能引用和右值引用

《Effective Modern ++》条款 25：针对右值引用实施 std::move，针对万能引用实施 std::forward

## 2.7.6 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，注意对齐。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)
{
...
}
```

## 2.7.7 前置自增和自减

前置自增/自减在效率上绝不会低于后置自增/自减。

Bigo CG 规定，在不影响程序逻辑的情形下，对于所有对象（尤其是迭代器和其他自定义类型对象）使用前缀形式（++i）的自增，自减运算符。

```
// bad
for(auto iter = data.begin(); iter != data.end(); iter++)
{
    doSomething();
}


// good
for(auto iter = data.begin(); iter != data.end(); ++iter)
{
    doSomething();
}
```

Bigo CG 不建议使用以下后置自增运算形式，来缩减代码长度。

```
// bad
int pos = 0;
while(pos < 100)
    data[pos++] = 0;  //一行代码包含两个逻辑

// good
int pos = 0;
while(pos < 100)
{
    data[pos] = 0;
    ++pos;
}
```

### 2.7.8 sizeof

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

## 2.8 其他 C++ 特性

### 2.8.1 C++标准

Bigo CG 考虑当前各编译器厂商（gcc、vs、clang）对 C++标准的支持程度，规定可以使用标准为 C++ 11，暂不建议使用 C++ 14。C++ 14 补充的常用 make_unique 操作可以参考 C++ 相关 proposal，直接添加源码，作为工具引入。（可参考 BIPL 项目）

Bigo CG 暂时严禁使用 C++17/20 的特性。

### 2.8.2 （显式）强制类型转换

使用 C++ 的类型的显式强制转换，如 static_cast<>()。

不要使用 int y = (int)x 或 int y = int(x) 等转换方式。

同一个继承体系内，类的基类指针/引用到派生类指针/引用，请使用 static_cast。

跨继承体系的类型转换则使用 dynamic_cast。

关于隐式类型转换，见本文档章节 2.4.6 。

参考资料：《Effective C++》条款 27：尽量少做转型动作

《More Effective C++》 条款 5：谨慎定义类型转换函数。

### 2.8.3 类型别名

对于类型别名，Bigo CG 推荐使用 C++ 11 中的 using 别名声明。using 别名声明更为清晰，并且支持模板，可以完全替代 typedef。相反， typedef 需要确定类型，并不支持模板。

```
// bad
typedef vector<int> IntVector;

// good
using IntVector = vector<int>;
```

参考资料：《Effective Modern C++》条款 9：优先选用别名声明，而非 typedef

### 2.8.4 const/constexpr 的使用

一般而言，我们强烈建议你在任何可能的情况下都要使用 const/constexpr。

对于指针或者引用类型，使用底层 const 修饰符(const A *, const A &)表示指针/引用指向对象不可更改。

值得注意的是，对于基本类型（如 int，float）采用值的函数参数。使用或不使用 const 并不会发生函数重载。

```
//the following two function declarations are exactly the same
void func(int param)
void func(const int param)

//the following two function definitions are not the same
void func(int param) {…}
void func(cons tint param) {…}
```

《C++编程规范（101）》 认为在函数声明中，要避免将通过值传递的函数参数声明为 const，其建议函数声明不包含顶级 const，避免函数接口让用户产生疑惑。

参考资料：

《Effective C++》条款 03：尽量使用 const

《Effective Modern C++》条款 15：只要有可能使用 constexpr，就使用它

《C++编程规范（101）》条款 15：积极使用 const

## 2.8.5 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。

预处理宏不要缩进。

参考文献：《Effective C++》条款 02：尽量以 const, enum, inline 替换 #define

## 2.8.6 空指针(nullptr VS NULL VS 0)

Bigo CG 规定使用 C++11 中的 nullptr 来代表空指针，而不是使用 NULL/0 来表示。NULL 本质上是整数 0，可能会导致意外的类型转换。而 nullptr 在 C++11 中为确定的类型，从而避免此问题。

参考文献：《Effective Modern C++》条款 08：优先选用 nullptr，而非 0 或 NULL

## 2.8.7 智能指针

C++ 11 提供了三种智能指针：std::unique_ptr, std::shared_ptr, std::weak_ptr 以供使用。

Bigo CG 推荐使用 C++ 11 提供的智能指针来自动管理内存和资源，实施所谓的 RAII （资源获取就是初始化）和异常安全。但使用者对这三类智能指针的特性应该有足够的了解，否则误用反而会导致难以调试的 Bug。

一般而言，std::unique_ptr 表示“所有权”语义，而 std::shared_ptr 表示“共享”语义，应该考虑适当的语义，以使用不同类型的智能指针。特别地，对于“工厂模式”或者 PIMPL 手法，除非经仔细思考确实会发生“共享”，一般使用 std::unique_ptr 足以，否则请勿滥用 std::shared_ptr，因为 std::shared_ptr 会有额外的拷贝开销，并且资源释放的位置也会变得不明显。

对于不涉及智能指针本身对于资源生命周期进行管理的函数，请通过 get() 直接传递裸指针参数即可。

参考文献：《Effective C++》条款 13：以对象管理资源

《Effective C++》条款 14：在资源管理类中小心 copying 行为

《Effective C++》条款 15：在资源管理类中提供对原始资源的访问

《Effective C++》条款 17：以独立语句将 newed 对象置入智能指针

《Effective Modern C++》条款 18：使用 std::unique_ptr 管理具有专属所有权的资源

《Effective Modern C++》条款 19：使用 std::shared _ptr 管理具有共享所有权的资源

《Effective Modern C++》条款 20：对于类似 std::shared _ptr 但有可能空悬的指针使用 std::weak_ptr

《Effective Modern C++》条款 21：优先选用 std::make_unique 和 std::make_shared，而非直接使用 new

《Effective Modern C++》条款 22：使用 Pimpl 习惯用法时，将特殊成员函数定义到实现文件中。

https://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/

https://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/

## 2.8.8 Lambda 表达式

适当使用 lambda 表达式。别用默认 lambda 捕获，所有捕获都要显式写出来。

不要使用 std::bind，在 C++ 14 标准下，lambda 表达式完全可以替代 std::bind。

```
// bad
int count = 100;
[&](){doSomething(count);}
[=](){doSomething(count);}


// good
int count = 100;
[&count](){doSomething(count);}
[count](){doSomething(count);}
```

参考资料：《Effective Modern C++》条款 31：避免默认捕获模式

《Effective Modern C++》条款 34：优先选用 lambda 式，而非 std::bind

## 2.8.9 std::function VS 函数指针

Bigo CG 推荐使用 std::function 来传递函数（可调用对象），而非裸函数指针。

## 2.8.10 assert

请尽量考虑代码的鲁棒性，对函数参数及用户行为不要做过多的假设，必要时使用 assert 语句，便于调试。这也是防御式编程的重要手段。请使用 Bigo CG 自定义的 assert 语句/宏，不要使用 c++的 assert 语句。Bigo CG 的 assert 语句头文件目录为："xxxx/xxxx/xxxx"。(实际依项目而定)

## 2.8.11 Warning 相关

Bigo CG 强调所编写代码应该以无警告、干净的方式通过编译。对于编译期所报告警告，应仔细对待。尤其对于数值类型出现隐性转换之处，应显式使用 static_cast<>。

参考资料：《Effective C++》条款 53：不要轻忽编译期的警告。

## 2.8.12 异常

当前工业界对于异常的使用仍然存在一定分歧。

C++ 引入"异常"固然有其合理性，利用"异常"能够更方便地进行错误处理，减少大量的错误处理代码，从而让核心逻辑更加清晰。然而要写出"异常安全"的代码是非常费力的（如采用常见的 copy & swap 手法），并且含有异常处理的代码可能会让阅读者对控制流产生不清晰的认识。此外，如果忘记对"异常"进行 catch，则会导致整个程序的崩溃。

（备注：对于崩溃的时机也存在两种认知，第一种是尽量让程序尽早崩溃从而暴露问题，及时修复。第二种是，不能让当前模块的错误，导致整个程序的崩溃，严重影响用户体验，所以尽量不崩溃。）

但是，考虑到 C++标准库本身在使用"异常"，所以要完全禁止异常也是不太可能的。另外，new 操作符同样会抛出"异常",虽然作为替代可以使用 new std::no_throw。

Bigo CG 考虑当前 C++项目大多为底层算法实现，如果上层模块禁止异常，可能会导致兼容性问题。所以规定：自有代码尽量避免使用异常，通过使用状态码+检查机制来进行错误处理。

参考资料：《Effective C++》条款 29：为“异常安全”而努力是值得的。

## 2.9 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。

### 2.9.1 注释风格

使用 // 或 /* */ 都可以， 但 // 更常用。

考虑到现阶段中文编码问题，Bigo CG 推荐使用英文注释。本文档所用中文注释仅为更明晰地进行阐述，请不要模仿。

开发者对于每一行注释都应该仔细考虑，注释应该简洁，并能准确描述所应说明内容。

### 2.9.2 文件注释

Bigo CG 规定，无论是在头文件还是在实现文件中，都使用统一的文件头：

注意：下面代码包含透明字符“_”，复制时请手动去除。

```
/**
 * @author      : Chen Wei
 * @date        : 2019-2-18
 * @description : data type enum class
 * @last_update : 2019-3-18 (optional)
 * @version     : 1.0
 */
```

### 2.9.3 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。

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{
    ...
}
```

### 2.9.4 函数注释

函数注释描述函数功能和接口定义。

函数注释位于声明之前，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?

Bigo CG 规定使用以下形式来编写函数注释:

```
/**
 * implementation of glLookAt which gives the view matrix.
 *
 * @param[in] origin   camera position specified in the world coordinate
 * @param[in] target   the viewing target position specified in the …
 * @param[in] up       the up direction of the camera specified in the
 *
 * @return the view matrix(world to camera transformation)
 */
/**
```

```
 * implementation of doSometing
 *
 * @param[in]     in     parameter to read only
 * @param[in,out] in_out parameter to read and write
 * @param[out]    out    parameter to write only
 */
void doSomething(const int &in, int &in_out, itn &out);
```

### 2.9.5 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。

成员变量的注释直接在后面使用 //!< 来实现。

```
// 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;

class Myclass
{
private:
    int m_num_total_entries; //!< total entries num
};
```

### 2.9.6 实现注释

对于代码中巧妙、晦涩、有趣、重要的地方要加注释。

在修改作者为他人的源文件时，如果改动较大，请在代码修改处添加说明。

#### 2.9.6.1 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;
}
```

### 2.9.6.2 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.
```

### 2.9.6.3 字面值函数实参

向函数传入字面常量值时（如 NULL、rue/false、123）时，要注释说明含义，或使用常量让代码望文知意。

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

// 正确的做法
bool success = func(interesting_value,
                    10,    // Default base value.
                    false, // Not the first time to call 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);
```

### 2.9.6.7 TODO 和 NOTE 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
```

对于需要代码阅读者（包括自己）尤其注意的地方，可以使用 NOTE 注释，同样需要在圆括号里面写上你的大名。

```
// NOTE(Chen Wei): disable warning 4068 for MSVC
```

### 2.9.6.8 避免无意义注释

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。

## 2.10 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，只有这样才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 Bigo CG 中使用的规则，请大家务必遵守。

## 2.10.1 行长度

每一行代码字符数尽量不要超过 120，超过了考虑换行书写，并注意对齐。

## 2.10.2 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。本要求在文件编码章节 1.2.3 中亦有提及。

## 2.10.3 缩进

使用空格而不是制表符来进行缩进。

不同的编辑器对于 tab 的显示宽度存在差异（2 个空格或 4 个空格）。

为避免显示差异及对齐问题。Bigo CG 规定只使用空格，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。

## 2.10.4 水平留白（空格）

水平留白，永远不要在行尾添加没有意义的留白。

添加的空格的主要目的为增加代码的可读性。

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）。

```
//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
//注意：操作算子留白能提高代码的可读性，但经常被 Bigo CG 团队成员忽略，在此强调。
```

```
x = 0;       // Assignment operators always have spaces around them.

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

```
//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.
```

## 2.10.5 垂直留白（空行）

考虑程序可读性适当添加空行。

函数定义之间添加空行，1 行为好。

过于紧密的代码块将会给阅读者带来心理压力，并且降低清晰性。当然，过于稀松的代码同样难看，这取决你的判断。

一般而言，在代码实现时应该注重逻辑的收敛，完成同一功能的代码应该是高度内聚的，并且位于相同的代码块。在不同的代码块之间可以添加空行以进行区分，让代码结构更为清晰，并且可以在代码块之前适当添加注释概述其主要功能。

## 2.10.6 大括号

BigoCG 推荐用来表示代码块的大括号都应该新启并独占一行。此要求暂不强制，左大括号亦可不另起一行，但需要在整个项目或者模块内保持一致。

```
//Good
if (xxx)
{
    doSomething();
}
```

```
else
{
    doSomething();
}

void func()
{
    doSomething();
}
//Not That Good
void func(){
    doSomething();
}
```

## 2.10.7 对齐

好的对齐风格对于代码的可读性和清晰度都尤为重要，无论是注释或者代码，都应该有良好的对齐风格。精致的代码应该看起来整齐而错落有致，而不是犬牙交错。

BIGO CG 强调编码者应该注重对齐。

注意在 Visual Studio 中复制代码经常会出现缩进自动调整的情形，请按 CTRL + Z 防止 Visual Studio 自己调整。

### 2.10.6.1 注释对齐

请确保注释也是良好对齐的。包括“*”字符的对齐，以及注释文本的对齐。

```
//好的注释对齐
/**
 * @author      : Chen Wei
 * @date        : 2019-2-18
 * @description : data type enum class
 * @last_update : 2019-3-18 (optional)
 * @version     : 1.0
 */

//不好的注释对齐。
/**
* @author      : Chen Wei
* @date        : 2019-2-18
* @description : data type enum class
* @last_update : 2019-3-18 (optional)
```

```
* @version      : 1.0
*/

//好的注释对齐
/** class Strides describe the strides information of tensor.
 *
 *  NOTE(Chen Wei): For a 2D image, its meaningful strides is of 3 dimensions.
 *                  The first  dimension represents the bytes of per element.
 *                  The second dimension represents the bytes of per row.
 *                  The third  dimension represents the bytes of per image
 *                  Currently, we do not support col-major mode for tensor.
 */

//不好的注释对齐
/** class Strides describe the strides information of tensor.
 *
 *  NOTE(Chen Wei): for a 2D image, its meaningful strides is of 3 dimensions.
 *    The first dimension represents the bytes of per element. The second dimension
 *  represents the bytes of per row. The third  dimension represents the bytes of
 *      per image. Currently, we do not support col-major mode for tensor.
 */
```

## 2.10.6.2 变量声明对齐

无论类的数据成员、enum 成员，还是普通变量，其声明和注释都应该尽量做到对齐。

```
//好的变量声明对齐
unsigned int   m_image_width  = 0;                          //!<image width
unsigned int   m_image_height = 0;                          //!<image height
ImageFormat    m_image_format = ImageFormat::UNKNOWN;       //!<image format

//不太好的变量声明对齐
unsigned int m_image_width = 0; //!<image width
unsigned int m_image_height = 0; //!<image height
ImageFormat m_image_format = ImageFormat::UNKNOWN; //!<image format

//好的枚举成员声明对齐
/** enum class data type */
enum class DataType
{
    UNKONWN,       //!< unknown data type
    CHAR,          //!< char
```

```
    UCHAR,         //!< unsigned char
    INT16,         //!< INT16
    //...
};

//不太好的枚举成员声明对齐
/** enum class data type */
enum class DataType
{
    UNKONWN, //!< unknown data type
    CHAR, //!< char
    UCHAR, //!< unsigned char
    INT16, //!< INT16
    //...
};
```

### 2.10.6.3 函数参数对齐

函数参数是接口的重要描述，应该着重对齐。

```
//好的函数参数对齐
void createPipeline(const std::string           &name,
                    const std::vector<uint32_t> &specializaions,
                    uint32_t                     binding_count,
                    uint32_t                     constant_count);

//不好的函数参数对齐（缩进和变量对齐都不好）
//很多 IDE 默认是这种风格，但是为保持清晰性，请多敲空格对齐。
void createPipeline(const std::string &name,
    const std::vector<uint32_t> &specializaions,
    uint32_t binding_count,
    uint32_t constant_count);

//好的函数调用对齐
memcpyBasedOnBufferInfo(dst_ptr,
                        src_ptr,
                        m_buffer_info,
                        buffer->bufferInfo(),
                        m_buffer_info.tensorShape().dimNum() - 1);

//不好的函数调用对齐
memcpyBasedOnBufferInfo(dst_ptr,
    src_ptr,
```

```
    m_buffer_info,
    buffer->bufferInfo(),
    m_buffer_info.tensorShape().dimNum() - 1);
```

### 2.10.6.4 实现对齐

对于具体的代码实现，也应该做到尽量工整和对齐。

```
//好的实现对齐
cl::NDRange CLKernelBase::getGlobalWorkSizeFromWindow(const Window &window)
{
    //if 条件语句对齐
    if(window[0].end() - window[0].start() == 0 ||
       window[1].end() - window[1].start() == 0 ||
       window[2].end() - window[2].start() == 0)
    {
        return cl::NullRange;
    }

    //参数对齐
    return cl::NDRange((window[0].end() - window[0].start()) / window[0].step(),
                       (window[1].end() - window[1].start()) / window[1].step(),
                       (window[2].end() - window[2].start()) / window[2].step());
}

//不好的实现对齐
cl::NDRange CLKernelBase::getGlobalWorkSizeFromWindow(const Window &window)
{
    //if 条件语句未对齐
    if(window[0].end() - window[0].start() == 0
       || window[1].end() - window[1].start() == 0
       || window[2].end() - window[2].start() == 0)
    {
        return cl::NullRange;
    }

    //参数未对齐
    return cl::NDRange((window[0].end() - window[0].start()) / window[0].step(),
       (window[1].end() - window[1].start()) / window[1].step(),
       (window[2].end() - window[2].start()) / window[2].step());
}
```

```
//好的实现对齐
BIPL_ASSERT(global_work_size[0] >= global_work_offset[0]);
BIPL_ASSERT(global_work_size[1] >= global_work_offset[1]);
BIPL_ASSERT(global_work_size[2] >= global_work_offset[2]);

//好的实现对齐
unsigned int new_width  = alignUp(old_width);
unsigned int new_height = alignUp(old_height);
unsigned int new_depth  = alignUp(old_depth);
```

# 3 工程实践

在工程实践部分，本文档将关注关乎类设计的一些相关注意事项。此部分的文档撰写仍然暂时比较初步。除讨论关于类的宏观设计外，我们还顺带关注 git 的实践使用，毕竟这的的确确是一个工程问题，而非一个风格问题。

## 3.1 启发式原则

在此，本文档列出一些在进行代码设计和工程实践可以考虑的启发式原则：

面向对象设计 SOLID 原则：
①SRP 原则（单一职责原则）：类应该只有一个改变的原则。
②OCP 原则（开闭原则）：类应该对扩展开放，对修改关闭。
③LSP 原则（里氏替代原则）：确保 public 继承实现的 IS-A 关系。
④ISP 原则（接口分离原则）：客户不应该被迫依赖于他们不用的接口。
⑤DIP 原则（依赖倒置原则）：高层模块不应该依赖于底层模块，两者应该依赖于抽象。

其他原则：
①组合聚合复用原则：多用组合，少用继承。
②好莱坞原则：别找我，我来找你。
③迪米特法则、德墨忒尔原则、最少知识原则：不要和陌生人说话。
④DRY 原则：Don’t Repeat Yourself，不要重复自己。
⑤KISS 原则：Keep It Simple Stupid，最简单清晰的就是最好的。
⑤高内聚、松耦合原则。
⑥针对接口编程，而不是针对实现编程。
⑦无环依赖原则。
⑧惯例优于配置原则。
⑨You Aren’t Gonna Need it：只实现目前需要的功能，当前状态最简单。
⑩契约式设计原则。

参考资料：《Head First 设计模式》

## 3.2 类设计

### 3.2.1 析构函数

当类作为多态基类使用时，即类包含虚函数时。请务必将析构函数声明为 virtual，保证类对象能够正确析构。

```
class BaseClass
{
public:
    BaseClass();
    virtual ~BaseClass ();

    virtual void virtualFunc();
};
```

禁止在析构函数中调用虚函数，因为这类调用是不会重定向到子类的虚函数实现。即对象在析构期间不具有多态特性。

禁止在析构函数中抛出异常，将会产生未定义行为。

参考资料：《Effective C++》条款 07：为多态基类声明 virtual 析构函数。
《Effective C++》条款 09：绝不在构造和析构过程中调用 virtual 函数
《Effective C++》条款 08：别让异常逃离析构函数

## 3.3 继承设计

### 3.3.1 pubic 继承

public 继承塑造的语义是 IS-A 关系，即对于每一个接口，都可以用派生类对象来替代基类对象。Bigo CG 强调不能因为复用代码的需求，而鲁莽采用 public 继承。此时应该采用组合来完成代码复用。

相对于组合，继承是一种静态关系，其在编译期已经确定，在运行期不具有灵活性。

参考资料：《Effective C++》条款 32：确保你的 public 继承塑造 is-a 关系

### 3.3.2 private 继承

private 继承塑造的语义是“根据某物实现出”。非必要，Bigo CG 推荐使用复合来实现同一语义。

参考资料：《Effective C++》条款 39：明智而审慎地使用 private 继承。

### 3.3.3 多重继承

真正需要用到多重实现继承的情况少之又少，非绝对必要不要使用多继承。一般而言，多重继承可以用单继承+组合的方式来替代。

参考资料：《Effective C++》条款 40：明智而审慎地使用多重继承

## 3.4 Git 相关

### 3.4.1 优先使用 git pull --rebase

git pull 本质上是 git fetch + git merge 命令。如果远程分支领先本地分支，并且本地分支也有提交，则应该使用 git pull –rebase，其本质是 git fetch + git rebase，使用变基能保持线性提交记录，方便回溯。

参考资料：
https://stackoverflow.com/questions/2472254/when-should-i-use-git-pull-rebase

### 3.4.2 规范 git commit style

git commit 应遵循一定的规范，方便后续进行代码回溯和追踪。

Bigo CG 建议采用如下形式 git commit style:

```
git commit -m "type:decsription"

type 用于说明 commit 的类别，只允许使用下面 7 个标识：

feat:     新功能（feature）
```

```
fix:      修补 bug
docs:     文档（documentation）
style:    格式（不影响代码运行的变动）
refactor: 重构（即不是新增功能，也不是修改 bug 的代码变动）
test:     增加测试
chore:    构建过程或辅助工具的变动
```

参考资料：

http://www.ruanyifeng.com/blog/2016/01/commit_message_change_log.html

# 4 结束语

运用常识和判断力，并且 保持一致。

编辑代码时，花点时间看看项目中的其它代码，并熟悉其风格。如果其它代码中 if 语句使用空格，那么你也要使用。 如果其中的注释用星号（*）围成一个盒子状，那么你同样要这么做。

风格指南的重点在于提供一个通用的编程规范，这样大家可以把精力集中在实现内容而不是表现形式上。我们展示的是一个总体的风格规范，但局部风格也很重要，如果你在一个文件中新加的代码和原有代码风格相去甚远，这就破坏了文件本身的整体美观，也让打乱读者在阅读代码时的节奏，所以要尽量避免。

# 5 参考书目

除 Google 代码规范以及网上资料外，本文档还大量参考了一些著名书籍，在此进行列举。这些书籍大多出自 C++ 或其他编程语言的专家之手，具有相当的借鉴和阅读价值。

①Effective C++ 系列书籍：

《Effective C++：改善程序与设计的 55 个具体做法》

《More Effective C++：35 个改善编程与设计的有效方法》

《Effective Modern C++：42 招独家技巧助你改善 C++11 和 C++14 的高效用法》

《Effective STL：50 条有效使用 STL 的经验》

②Exceptional C++ 系列书籍：

《Exceptional C++：47 个 C++工程难题、编程问题和解决方案》

《More Exceptional C++：40 个新的工程难题、编程疑问及解决方法》

《Exceptional C++ Style：40 个新的工程难题、编程问题及解决方案》

Exceptional C++ 系列书籍的作者为 Herb Sutter，现任 C++标准委员会主席。上述书籍整理自其专栏 Guru of the Week (GotW)，其还为 C++11/14 专门更新了一些条款。

详见：https://herbsutter.com/gotw/

③《C++编程规范：101 条规则、准则与最佳实践》

④《C++ Core Guidelines》

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

⑤《代码大全》

⑥《代码整洁之道（clean code)》

⑦《深度探索 C++对象模型》

⑧《Head First 设计模式》