Google C++ Style Guide

Background

- Style, also known as readability, is what we call the conventions that govern our C++ code.

命名

General Naming Rules

- 命名应该描述使用这个对象的目的
- 少用缩写（如果这个缩写在Wikipedia中有那么可以使用用）
- 一些习惯性命名是可以接受的，比如迭代变量i,模版参数T
- 不要吝啬水平空间，这意味着你可以使用长一些的名称和空格使得代码更容易阅读和理解。

File Names

- 文件名应该全部小写
- 可以使用下划线_或者短横线-，如果没有特别要求，倾向于使用下划线_
- 不要使用已经存在于 /usr/include下的头文件名称，比如netdb.h
- 尽量使名称更加特殊化，比如使用http_server_logs.h而不是logs.h
- 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

```
my_useful_class.cc
my-useful-class.cc
myusefulclass.cc
```

Type Names

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

```
// classes and structs
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// typedefs
```

```
  typedef hash_map<UrlTableProperties *, std::string> PropertiesMap;

  // using aliases
  using PropertiesMap = hash_map<UrlTableProperties *, std::string>;

  // enums
  enum class UrlTableError { ...
```

Variable Names

- 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, 但结构体的就不用

```
// class
class TableInfo { ...
 private:
  string table_name_; // 可 - 尾后加下划线。 string tablename_;
  static Pool<TableInfo>* pool_; // 可。
};
// struct
struct UrlTableProperties {
  string name;
  int num_entries;
};
```

Constant Names

- 在全局或类里的常量名称前加 k ,且除去开头的 k 之外每个单词开头字母均大写。
- 在不能用大写字母分隔的特殊情况下，允许使用下划线

```
const int kDaysInAWeek = 7;
const int kAndroid8_0_0 = 24;  // Android 8.0.0
```

Function Names

- 常规函数使用大写字母开头的驼峰命名, 取值和设值函数则要求与变量名匹配， 例如
  - MyExcitingFunction()
  - MyExcitingMethod()
- Accessors and mutators (get and set functions) 使用小写单词加下划线连接

    - my_exciting_member_variable()
    - set_my_exciting_member_variable()
- 取值和设值函数要与存取的变量名匹配,但不是强制要求
- 单词首字母缩写形成的单词，倾向于将它看作普通单词， `StartRpc()` 而不是 `StartRPC()` .
- 为了遵循惯例，文件作用域内部的函数cuda kernel可以使用小写+下划线形式，对外导出的函数采用大驼峰形式，参阅tensorflow

https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/rnn/lstm_ops_gpu.cu.cc#L85

tensorflow/lstm_ops_gpu.cu.cc at master · tensorflow/tensorflow

An Open Source Machine Learning Framework for Everyone - tensorflow/lstm_ops_gpu.cu.cc at master · tensorflow/tensorflow

```
class MyClass {
 public:
  ...
  int num_entries() const { return num_entries_; }
  void set_num_entries(int num_entries) { num_entries_ = num_entries; }
  void StartRpc() { ... }
 private:
  int num_entries_;
};

__global__ void rga2bgr(...) {
  ......
}
```

Namespace Names

- 命名空间的名称应该全小写，单词用下划线分隔
- 顶层的命名空间名称应该是project的名称，或者团队的名称

```
  hipp::cuda::resize
```

Enumerator Names

- 枚举命名应该像常量一样命名
- 枚举类型名还是使用类的命名风格

```
enum class UrlTableError {
```

```
  kOk = 0,
  kOutOfMemory,
  kMalformedInput,
};
```

Macro Names

- 全部大写
- 下划线分隔

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

Exceptions to Naming Rules

- bigopen(): 函数名, 参照 open() 的形式
- sparse_hash_map: STL 风格的实体; 参照 STL 命名约定
- LONGLONG_MAX:常量，类似于 INT_MAX

头文件

Self-contained

- 头文件需要满足自给自足，也就是说，它可以作为第一个头文件被引入，而不需要在引入它之前为它包含额外的头文件

define guards

- 所有头文件都应该使用 #define 来防止头文件被多重包含, 命名格式当是: <PROJECT>_<PATH>_<FILE>_H_
- 为保证唯一性, 头文件的命名应该基于所在项目源代码树的全路径。例如项目 foo 中的头文件 foo/src/bar/baz.h

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
…
#endif // FOO_BAR_BAZ_H_

```

Include What You Use

- 不要使用传递包含
  - foo.cc should include bar.h if it uses a symbol from it even if foo.h includes bar.h.

Forward Declarations

- 前置声明就是声明一个不带具体定义的实体

```
// In a C++ source file:
class B;
void FuncInB();
extern int variable_in_b;
ABSL_DECLARE_FLAG(flag_in_b);
```

- 尽量不要使用前置声明
  - 前置声明隐藏了依赖关系
  - 前置声明可能会被库的后续更改所破坏。
  - 前置声明来自命名空间 `std::` 的 symbol 时，其行为未定义。
  - 前置声明可能改变代码的含义

```
// b.h
struct B {};
struct D : B {};

// main.cc
#include<iostream>
// #include "b.h"
struct B;
struct D;
void f(B*) {
  std::cout << " void f(B*) " << std::endl;
}

void f(void*) {
  std::cout << " void f(void*) " << std::endl;
}

void test(D* x) { f(x); }  // which f will be called?
int main(){
  D* d;
  test(d);  // void f(void*)
  return 0;
}
```

Inline Functions

- 定义一个函数为内联函数，一般来说要求这个函数的行数在10行以内，行数过多的内联请求可能会被编译器忽略
- 含有loops和switch语句的函数一般不要内联，除非这些语句从来未被执行过
- 虚函数和递归函数一般不内联

Names and Order of Includes

- include名称
  - 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, google-awesome-project/src/base/logging.h应该这样被包含

```
1  #include "base/logging.h"
2
3  // grpc/src/core/lib/http/parser.h
4  #include "src/core/lib/iomgr/error.h"
```

- 顺序
  - 直接相关头的文件，如cpp文件对应的h文件，或者测试类的头文件
  - C标准库头文件
  - C++标准库头文件
  - 第三方库头文件
  - 本工程头文件
- 例子（注意每个非空的头文件组需要用空行分开）

```
1  #include "foo/server/fooserver.h"
2
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  #include <string>
7  #include <vector>
8
9  #include "base/basictypes.h"
10 #include "foo/server/bar.h"
11 #include "third_party/absl/flags/flag.h"
```

- 例外：有时候必须用到条件包含，这时候可以按照逻辑顺序而不是规定的顺序

注释

Comment Style

- 尽管注释很重要，但是好的代码是self-documenting的
- 可以使用 // 或者/**/，但是需要保持一致性

https://google.github.io/styleguide/cppguide.html#Comment_Style

Google C++ Style Guide

Google C++ Style Guide Background C++ is one of the main development languages used by many of Google's open-source projects. As every C++ programmer knows, the language has many powerful features, bu

File Comments

- 在每一个文件开头加入版权公告license.为项目选择合适的许可证版本.(比如, Apache 2.0, BSD, LGPL, GPL)
- 文件注释描述文件的内容。如果一个文件只声明, 或实现, 或测试了一个对象, 并且这个对象已经在它的声明处进行了详细的注释, 那么就没必要再加上文件注释. 除此之外的其他文件都需要文件注释.

Class Comments

- 如果这个类不是特别简单(non-obvious),每个类的声明都要附带一份注释, 描述类的功能和用法,
- 类注释应当为读者理解如何使用与何时使用类提供足够的信息, 同时应当提醒读者在正确使用此类时应当考虑的因素，要特别注意文档说明多线程环境下相关的规则
- 用一小段代码演示这个类的基本用法或通常用法, 放在类注释里也非常合适
- 描述类用法的注释应当和接口定义放在一起, 描述类的操作和实现的注释应当和实现放在一起.

Function Comments

- 如果这个函数不是特别简单(non-obvious)，函数声明处的注释描述函数功能; 定义处的注释描述函数实现.
- 在函数声明处做如下注释
  - inputs and outputs 含义
  - 指针参数是否可以为空
  - 对于输入和输出参数，是否有状态上的改变
  - 该函数使用时是否有性能上的后果
- 如果函数中使用了一些tricks，在函数定义处应该给予注释
- *不要* 从 `.h` 文件或其他地方的函数声明处直接复制注释到函数定义处，可以简短复述，重点是说明函数怎样实现

Variable Comments

- 一般来说，变量名称应该有足够的描述信息，但是有时候需要更多的注释
- 每个类数据成员都应该用注释说明用途，还有一些特殊值, 数据成员之间的关系, 生命周期等不能够用类型与变量名明确表达, 则应当加上注释；变量名称足够描述清楚的不需要添加注释
- 全局变量都应该添加注释
  - what they are
  - what they are used for
  - why they need to be global

Don'ts

- 不要给很明显的代码注释

```
1 // Find the element in the vector.  <-- Bad: obvious!
2 if (std::find(v.begin(), v.end(), element) != v.end()) {
3   Process(element);
4 }
5 // Process "element" unless it was already processed. <-- Bad: obvious! already Se
6 if (!IsAlreadyProcessed(element)) {
7   Process(element);
8 }
```

Punctuation, Spelling, and Grammar

- 注意标点, 拼写和语法;
- 虽然被别人指出该用分号时却用了逗号多少有些尴尬, 但清晰易读的代码还是很重要的。正确的标点， 拼写和语法对此会有很大帮助。

TODO Comments

- 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名字, 邮件地址, bug ID等身份标识

```
1 // TODO(kl@gmail.com): Use a "*" here for concatenation operator.
2 // TODO(Zeke) change this to use relations.
3 // TODO(bug 12345): remove the "Last visitors" feature.
4 // TODO(Zeke) change this to use relations. Fix by November 2005.
5 // TODO(Zeke) change this to use relations. Remove this code when all clients can
```

作用域

Namespaces

- 作用：将全局作用域细分为独立的, 具名的作用域, 可有效防止全局作用域的命名冲突.
- 命名空间的名称需要遵循“**命名空间命名规则**”
- 终止命名空间时使用注释标明
- 用命名空间把文件包含, gflags 的声明/定义, 以及类的前置声明**以外**的整个源文件封装起来, 以区别于其它命名空间
- 不要使用 using-directive(e.g., `using namespace foo`).
- 不要在std中声明任何东西，包括将标准库中的类做前置声明，前置声明std命名空间内的实体是undefined behavior
- 不要在头文件中使用*Namespace aliases，除非它是一个 internal-only namespace；但允许在.cc文件中使用Namespace aliases。*
- 不要使用内联命名空间(inline namespace)。

```
// In the .h file
// Namespace aliases Forbidden --
namespace sidetable = ::pipeline_diagnostics::sidetable;
namespace mynamespace {
// Forbidden -- This pollutes the namespace.
using namespace foo;
// All declarations are within the namespace scope.
// Notice the lack of indentation.
class MyClass {
 public:
   ...
   void Foo();
};

}  // namespace mynamespace

// In the .cc file
#include "a.h"

ABSL_FLAG(bool, someflag, false, "a flag");  // you can use flags

namespace mynamespace {

// Definition of functions is within scope of the namespace.
void MyClass::Foo() {
```

```
  ...
}

namespace baz = ::foo::bar::baz;  // you can Shorten access to some commonly used
using ::foo::Bar;  // you can use using-declarations.

...code for mynamespace...    // Code goes against the left margin.

}  // namespace mynamespace
// Forbiden inline namespace
namespace X {
inline namespace Y {
  void foo();
}
}
//X::Y::foo() 与 X::foo() 彼此可代替
```

Internal Linkage

- 当在.cc文件中定义不需要被外部引用的符号时，让它们处在匿名namespace中或者使用static
- 不要在.h文件中使用 internal linkage

```
// add.cc
namespace {
int InternalFunction(int a) { return a; }
}
static int AddImpl(int a, int b) {
  return a + b;
}

int Add(int a, int b){
  return AddImpl(a, b);
}
// add.h
int Add(int a, int b);
```

Nonmember, Static Member, and Global Functions

- 基本不使用全局函数
- 有时候需要定义不绑定实例的函数，这样的函数可以是静态函数或者非成员函数，非成员函数不应该依赖外部变量，倾向于把非成员函数放在一个namespace 里
- 如果一个非成员函数只在当前.cc文件中使用，那么需要参考**Internal Linkage**
- 类的静态方法应该和和类实例或者类的静态数据密切相关

- 不要简单用一个class把静态成员聚集起来，这与只是给名称一个公共前缀没有什么不同，而且这种group通常是不必要的

```
// bad
namespace myproject {
class FooBar {
 public:
   static void Function1();
   static void Function2();
};
}  // namespace myproject

// good
namespace myproject {
namespace foo_bar {
void Function1();
void Function2();
}  // namespace foo_bar
}  // namespace myproject

```

Local Variables

- 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。
- 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。
- while, for中需要的局部变量应该在while/for语句内部声明，这些局部变量作用域将限定在while/for块内
- 注意如果一个局部变量是object，那么它的构造函数将会在每一次进入特定scope时调用一次，析构函数在每次出这个作用域都会调用一次

```
int i;
i = f(); // 坏—初始化和声明分离
int j = g(); // 好—初始化时声明
vector<int> v;
v.push_back(1); // 用花括号初始化更好 v.push_back(2);
vector<int> v = {1, 2}; // 好—v 一开始就初始化
while (const char* p = strchr(str, '/')) str = p + 1;
// Inefficient implementation:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // My ctor and dtor get called 1000000 times each.
  f.DoSomething(i);
```

```
}
Foo f;  // My ctor and dtor get called once each.
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

Static and Global Variables

- 静态存储周期变量，即包括了全局变量、静态变量、静态类成员变量和函数静态变量，都必须是原生数据类型（POD：Plain Old Data）：即int、char和float，以及POD类型的指针、数组和结构体。
- 禁止使用类的静态存储周期变量。由于构造和析构函数调用顺序的不确定性，它们会导致难以发现的bug。不过constexpr变量除外，因为它不涉及动态初始化和析构。

类

Doing Work in Constructors

- 避免在构造函数中调用虚函数。子类的构造函数初始化时先进入父类的构造函数，当前对象的虚函数表地址为父类的虚函数表，并非子类的虚函数表
- 如果你不能发出错误信号，避免可能发生错误的初始化

Implicit Conversions

- 不要定义隐式转换，使用explicit关键字修饰单参数构造函数
  - 代码更难阅读；
  - 接收单个参数的构造方法可能会意外地被用做隐式类型转换；

```
class Foo {
  explicit Foo(int x, double y);
  ...
};

void Func(Foo f);
Func({42, 3.14});  // Error
```

- 特例
  - 拷贝构造和移动函数可以不声明成 explicit，因为它们没有进行类型转换
  - 参数是 std::initializer_list 的构造函数可以是非 explicit

Copyable and Movable Types

- copyable class 应该显式声明拷贝操作，如果需要可以声明高效的移动操作；move-only class应该显式声明移动操作并且应该显式delete拷贝操作
- 如果提供了copy/move的assignment operator, 那么也需要提供对应的constructor

```
class Copyable {
 public:
  Copyable(const Copyable& other) = default;
  Copyable& operator=(const Copyable& other) = default;

  // The implicit move operations are suppressed by the declarations above.
  // You may explicitly declare move operations to support efficient moves.
};

class MoveOnly {
 public:
  MoveOnly(MoveOnly&& other) = default;
  MoveOnly& operator=(MoveOnly&& other) = default;

  // The copy operations are implicitly deleted, but you can
  // spell that out explicitly if you want:
  MoveOnly(const MoveOnly&) = delete;
  MoveOnly& operator=(const MoveOnly&) = delete;
};

class NotCopyableOrMovable {
 public:
  // Not copyable or movable
  NotCopyableOrMovable(const NotCopyableOrMovable&) = delete;
  NotCopyableOrMovable& operator=(const NotCopyableOrMovable&)
      = delete;

  // The move operations are implicitly disabled, but you can
  // spell that out explicitly if you want:
  NotCopyableOrMovable(NotCopyableOrMovable&&) = delete;
  NotCopyableOrMovable& operator=(NotCopyableOrMovable&&)
      = delete;
};
```

Structs vs. Classes

- 仅当只有数据时使用struct,其它一概使用class.
- 和 STL 保持一致, 对于仿函数和 trait 特性可以不用 class 而是使用 struct.
- struct和class成员变量的命名规则略有不同(下划线)

Structs vs. Pairs and Tuples

- 当数据成员有明确的意义的时候，使用struct，其他的可以使用pair 或者 tuple

Inheritance

- 所有继承必须是 public 的. 如果你想使用私有继承, 你应该替换成把基类的实例作为成员对象的方式.
- 不要过度使用继承，组合常常更合适。主要是需要区分is-a和has-a，如果要使用继承，需要明确这是is-a的关系
- 数据成员应该是private的，除非它是常量
- 虚函数需要显式地标注override或者final关键字
- 多重继承是允许的，但是强烈不鼓励使用多重*implementation* inheritance
  - "Interface inheritance" is inheritance from a pure abstract base class (one with no state or defined methods);
  - all other inheritance is "implementation inheritance"

Operator Overloading

- 谨慎使用操作符重载
- 只定义含义明显的，与内置操作符一致的操作符
- 只对用户自己定义的类型重载运算符
- 如果定义了一个运算符, 请将其相关且有意义的运算符都进行定义, 并且保证这些定义的语义是一致的. 例如, 如果你重载了 `<`, 那么请将所有的比较运算符都进行重载并且保证对于同一组参数, `<` 和 `>` 不会同时返回 `true`
- 尽量避免将运算符定义为模板, 因为此时它们必须对任何模板参数都能够作用
- 不要单纯避免定义运算符重载，比如说倾向于使用==,=,<<，而不是 `Equals()`, `CopyFrom()`, `PrintTo()`；但有时候也不能过度使用运算符，比如该类型实际上没有大小顺序之分，但是为了将其存在std::set里面却重载了<
- 注意：如果一个类定义了二元运算符，隐式类型转换只会发生在右侧，所以有时候a<b可以通过编译但是b<a不能
- 不要重载 `&&`, `||`, `,` 或一元运算符 `&`，不要重载 `operator""`, 也就是说, 不要引入 user-defined literals.

Access Control

- 将类的数据成员定义成private，除非他们是常量
- 当使用Google Test的时候，允许test fixture类的数据成员声明为protected

Declaration Order

- 按照public protected private顺序，空的section可以省略
- 在每个section中，把相似的类型或者声明聚集在一起
- section内部的声明顺序
  a. Types and type aliases (`typedef` , `using` , `enum` , nested structs and classes, and `friend` types)
  b. Static constants
  c. Factory functions
  d. Constructors and assignment operators
  e. Destructor
  f. All other functions (`static` and non- `static` member functions, and `friend` functions)
  g. Data members (static and non-static)
- 不要把大的方法定义内联在类定义中。通常，只有不重要的或性能关键的、非常短的方法可以内联定义。

函数

Inputs and Outputs

- 倾向于使用return形式的返回值而不是通过参数形式，可读性更高
- 返回值的类型倾向于值或者引用，避免返回一个指针
- 在排序函数参数时， 将所有输入参数放在所有输出参数之前.
- .....

Write Short Functions

- 倾向于短小的，专一的函数
- 长函数有时候是合适的，所以没有对函数长度的一个硬性规定。但是超过40行的函数，可以思考能不能在不破坏结构性的前提下将其拆分。

Function Overloading

- 若要使用函数重载, 则必须能让读者一看调用点就很清楚发生了什么，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。
- 如果函数单靠不同的参数类型而重载，那么C++复杂的匹配规则可能让读者感到困惑。

Default Arguments

- 虚函数禁止使用缺省参数，否则最终派生类执行的函数是一个定义在派生类，但使用了基类的缺省参数值的虚函数。

```
class Base {
 public:
  virtual void Display1() {
    cout << "Base: Display1" << endl;
  }

  virtual void Display2(const int x = 1) {
    cout << "Base: Display2: " << x << endl;
  }
};

class Derived : public base {
public:
    virtual void Display1() {
        cout << "Derived: Display1" << endl;
    }

    virtual void Display2(const int x = 2) {
        cout << "Derived: Display2: " << x << endl;
    }
};

Base *ptr_base = new Derived();
ptr_base->Display1();  // Derived: Display1
ptr_base->Display2();  // Derived: Display2: 1
```

- 如果在每个调用点缺省参数的值都有可能不同, 在这种情况下缺省函数也不允许使用，例如：
  - `void f(int n = counter++);`

格式

Line Length

- 建议每一行代码字符数不超过80
- 特例
  - 如果一行注释包含了超过 80 字符的命令或 URL, 出于复制粘贴的方便允许该行超过 80 字符.
  - 包含长路径的 #include 语句或者using声明可以超出 80 列
  - 头文件保护可以无视该原则.

Spaces vs. Tabs

- 只使用空格，一次缩进使用两个空格

Function Declarations and Definitions

- 返回类型和函数名在同一行,参数也尽量放在同一行，如果放不下就对形参分行。
- 一些细节
  - 左圆括号总是和函数名在同一行，并且他们之间没有空格
  - 左圆括号和第一个参数之间没有空格
  - 左大括号总在最后一个参数同一行的末尾处，而不是下一行
  - 右圆括号和左大括号之间有一个空格
  - 右大括号要么自己在最后一行，要么就是和左大括号在同一行
  - 默认缩进是2个空格
  - 换行后的参数保持4个空格的缩进;
  - 所有参数应该尽可能地对齐，这意味着换行参数最后对齐时可能通过空格来补齐
  - 如果有些参数没有用到, 在函数定义处将参数名注释起来

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething();
  ...
}
ReturnType ClassName::count() { return count_; }

ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}

// 声明中形参恒有命名。
class Circle : public Shape {
public:
virtual void Rotate(double radians);
}
// 定义中注释掉无用变量。
void Circle::Rotate(double /*radians*/) {}
```

Function Calls

- 要么一行写完函数调用，要么在圆括号里对参数分行，要么参数另起一行且缩进四格。

- 如果没有其它顾虑的话，尽可能精简行数，比如把多个参数适当地放在同一行里。
- 有时候参数有一定结构，为了提高可读性可以按照其结构决定参数格式

```
bool retval = DoSomething(argument1, argument2, argument3);
bool result = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
bool result = DoSomething(
    argument1, argument2,  // 4 space indent
    argument3, argument4);

```

```
// Transform the widget by a 3x3 matrix.
my_widget.Transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

Conditionals

- if和左圆括号之间有一个空格
- 左右圆括号与条件之间不能有空格
- 如果条件中有多条语句，那么这些语句之间要用一个空格隔开
- 如果条件语句没有else和else if，那么可以省略大括号，甚至可以在一行内解决

```
if (condition) {                    // no spaces inside parentheses, space before
  DoOneThing();                     // two space indent
  DoAnotherThing();
} else if (int a = f(); a != 3) {  // closing brace on new line, else on same lin
  DoAThirdThing(a);
} else {
  DoNothing();
}

if (x == kFoo) return new Foo();

if (x == kBar)
  return new Bar(arg1, arg2, arg3);

if (x == kQuz) { return new Quz(1, 2, 3); }
```
- 一些错误的样例
```c++
```

```
if(condition) {              // Bad - space missing after IF
if ( condition ) {           // Bad - space between the parentheses and the condi
if (condition){              // Bad - space missing before {
if(condition){               // Doubly bad

if (int a = f();a == 10) {   // Bad - space missing after the semicolon
// Bad - IF statement with ELSE is missing braces
if (x) DoThis();
else DoThat();

// Bad - IF statement with ELSE does not have braces everywhere
if (condition)
  foo;
else {
  bar;
}

// Bad - IF statement is too long to omit braces
if (condition)
  // Comment
  DoSomething();

// Bad - IF statement is too long to omit braces
if (condition1 &&
    condition2)
  DoSomething();
```

Loops and Switch Statements

- 在单语句循环里，花括号可用可不用:
- 空循环体应使用 {} 或 continue.

```
for (int i = 0; i < kSomeNumber; ++i)
  printf("I love you\n");
while (condition) {
  // Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // Good - one newline is also OK.
while (condition) continue;  // Good - continue indicates no logic.
```

- switch 语句中的 case 块可以使用大括号也可以不用, 取决于你的个人喜好
- 如果有不满足 case 条件的枚举值, switch 应该总是包含一个 default 匹配，如果default应该不会被执行，那么应该将其视作一个error

```
switch (var) {
  case 0: {  // 2 space indent
    ...      // 4 space indent
    break;
  }
  case 1: {
    ...
    break;
  }
  default: {
    assert(false);
  }
}
```

Pointer and Reference Expressions

- .和->左右都不需要空格
- int* a 和 int *a都允许,但是必须要保持一致不能二者混用

```
// These are fine, space preceding.
char *c;
const std::string &str;
int *GetPointer();
std::vector<char *>

// These are fine, space following (or elided).
char* c;
const std::string& str;
int* GetPointer();
std::vector<char*>  // Note no space between '*' and '>'
```

- 允许在同一个声明中声明多个变量，但如果其中任何一个具有指针或引用修饰，则不允许，这样的声明很容易被误解。

```
int x, y;  // it's OK.
int x, *y;  // Disallowed - no & or * in multiple declaration
int* x, *y;  // Disallowed - no & or * in multiple declaration; inconsistent spaci
char * c;  // Bad - spaces on both sides of *
const std::string & str;  // Bad - spaces on both sides of &
```

Preprocessor Directives

- preprocessor directive应该顶格写，即使是处在缩进的body内

```
// Good - directives at beginning of line
  if (lopsided_score) {
#if DISASTER_PENDING      // Correct -- Starts at beginning of line
    DropEverything();
# if NOTIFY               // OK but not required -- Spaces after #
    NotifyClient();
# endif
#endif
    BackToNormal();
  }

// Bad - indented directives
  if (lopsided_score) {
    #if DISASTER_PENDING  // Wrong!  The "#if" should be at beginning of line
    DropEverything();
    #endif                // Wrong!  Do not indent "#endif"
    BackToNormal();
  }
```

Class Format

- 基类名称应该和子类名称在同一行（但遵循80字符限制）
- public 放在最前面, 然后是 protected, 最后是 private.
- public:, protected:, and private:关键字有1个空格缩进，且每个group需要用空行隔开（某些很小的class可以忽略这一点）
- 这些关键词后不要保留空行.
- 构造函数初始值列表放在同一行或按四格缩进并排几行.

```
class MyClass : public OtherClass {
 public:      // Note the 1 space indent!
  MyClass();  // Regular 2 space indent.
  explicit 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_;
};
// When everything fits on one line:
MyClass::MyClass(int var) : some_var_(var) {
  DoSomething();
}

// If the signature and initializer list are not all on one line,
// you must wrap before the colon and indent 4 spaces:
MyClass::MyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
  DoSomething();
}

// When the list spans multiple lines, put each member on its own line
// and align them:
MyClass::MyClass(int var)
    : some_var_(var),             // 4 space indent
      some_other_var_(var + 1) {  // lined up
  DoSomething();
}

// As with any other code block, the close curly can be on the same
// line as the open curly, if it fits.
MyClass::MyClass(int var)
    : some_var_(var) {}
```

Namespace Formatting

- 命名空间不增加额外的缩进层级
- 声明嵌套命名空间时，每命名空间都独立成行。

```
namespace name1 {
namespace name2 {

void foo() {  // Correct.  No extra indentation within namespace.
  ...
}
} // namespace name2
} // namespace name1
```

Floating-point Literals

- 有小数点，两边都有数
- 科学计数法可以使用E/e，两边都有数
- 用整型数去初始化浮点数是可以的

```
// bad
float f = 1.f;
long double ld = -.5L;
double d = 1248e6;

// good
float f = 1.0f;
float f2 = 1;   // Also OK
long double ld = -0.5L;
double d = 1248.0e6;
```

Horizontal Whitespace

```
int i = 0;  // Two spaces before end-of-line comments.

void f(bool b) {  // Open braces should always have a space before them.
...
int i = 0;  // Semicolons usually have no space before them.
// Spaces inside braces for braced-init-list are optional.  If you use them,
// put them on both sides!
int x[] = { 0 };
int x[] = {0};

// Spaces around the colon in inheritance and initializer lists.
class Foo : public Bar {
 public:
// For inline function implementations, put spaces between the braces
// and the implementation itself.
   Foo(int b) : Bar(), baz_(b) {}  // No spaces inside empty braces.
   void Reset() { baz_ = 0; }  // Spaces separating braces from implementation.
```

- 循环和条件语句
  - 主要是圆括号，花括号以及关键词之间的空格，不再赘述
- 操作符
  - 赋值等号左右总是各有一个空格

- 计算表达式操作符之间有空格，但是一个因式内部可以不需要空格分隔
- 一元操作符后不要跟空格

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

// Other binary operators usually have spaces around them, but it's
// OK to remove spaces around factors.  Parentheses should have no
// internal padding.
v = w * x + y / z;
v = w*x + y/z;
v = w * (x + z);

// No spaces separating unary operators and their arguments.
x = -5;
++x;
if (x && !y)
```

Other C++ Features

- 关于C++特性

https://google.github.io/styleguide/cppguide.html#Other_C++_Features

Google C++ Style Guide

Google C++ Style Guide Background C++ is one of the main development languages used by many of Google's open-source projects. As every C++ programmer knows, the language has many powerful features, bu

Google-Specific Magic

- 一些使得代码鲁棒性更强的tricks and utilities

https://google.github.io/styleguide/cppguide.html#Google-Specific_Magic

google.github.io

Exceptions to the Rules

https://google.github.io/styleguide/cppguide.html#Exceptions_to_the_Rules

Google C++ Style Guide

Google C++ Style Guide Background C++ is one of the main development languages used by many of Google's open-source projects. As every C++ programmer knows, the language has many powerful features, bu

代码格式化工具

clang-format工具

- clang-format 是一种自动格式化 C/C++/Java/JavaScript/JSON/Objective-C/Protobuf/C# 代码的工具
  - sudo apt install clang-format
- 使用内置的style如Google来格式化代码
  - clang-format -style=google -i main.cpp
- 使用.clang-format来实现自定义格式化
  - clang-format -style=可选格式名 -dump-config > .clang-format
  - 可以将.clang-format文件放在项目的根目录下；.clang-format配置文件搜索路径是相对当前文件由内向外搜索。
  - clang-format -style=file -i main.cc
- 只会调整format，不会修改任何有效代码包括注释
- .clang-format使用YAML 格式，文件各个字段的含义参阅

> https://clang.llvm.org/docs/ClangFormatStyleOptions.html
>
> **Clang-Format Style Options - Clang 15.0.0git documentation**
>
> Clang 15.0.0git documentation Clang-Format Style Options « ClangFormat :: Contents :: Clang Formatted Status » Clang-Format Style Options ¶ Clang-Format Style Options describes configurable formatting

cpplint

- cpplint.py支持的文件格式包括.cc、.h、.cpp、.cu、.cuh。

> https://raw.githubusercontent.com/google/styleguide/gh-pages/cpplint/cpplint.py
>
> **#!/usr/bin/env python # # Copyright (c) 2009 Google Inc. All rights reserved. # # Redistribution and**

- Cpplint只是一个代码风格检测工具，其并不对代码逻辑、语法错误等进行检查，并且也不保证能检查出所有的代码风格问题。
- 不会像clang-format一样自动修正风格，只是给出建议，需要自己手动修正。

```
1 $ python cpplint.py src/cvtcolor.cc
2
3 src/cvtcolor.cc:0:  No copyright message found.  You should have a line: "Copyrigh
  t [year] <Copyright Owner>"  [legal/copyright] [5]
```

```
4 src/cvtcolor.cc:110:  { should almost always be at the end of the previous line
   [whitespace/braces] [4]
5 Done processing src/cvtcolor.cc
6 Total errors found: 2
```

clang-tidy

- clang-tidy是基于AST的静态检查工具。因为它基于AST,所以要比基于正则表达式的静态检查工具更为精准，但是带来的缺点就是要比基于正则表达式的静态检查工具慢一点。
- clang-tidy诊断和修复一些编程错误，比如风格违规，接口误用，或者一些能够通过静态分析推断出的bug。
- 使用
  - `sudo apt install clang-tidy-10`
  - `clang-tidy --format-style=google -checks='-*,google-*'  --fix style.cc --`

```
// report
/mnt/workspace/hipp/style.cc:2:1: warning: do not use namespace using-directives;
using namespace std;
^
/mnt/workspace/hipp/style.cc:7:1: warning: single-argument constructors must be m
base_class(int num):number(num) {}
^
explicit
/mnt/workspace/hipp/style.cc:7:1: note: FIX-IT applied suggested code changes
clang-tidy applied 1 of 1 suggested fixes.
Suppressed 1141 warnings (1141 in non-user code).
```

- 效果

```
// 修复前
#include<iostream>
using namespace std;

class base_class{
                public:
    base_class() {}
base_class(int num):number(num) {}
virtual void funcBack()
{
    cout << "base func" << endl;
```

```
}

private:
 int number;
};

// 修复后
#include<iostream>
using namespace std;

class base_class{
                public:
    base_class() {}
    explicit base_class(int num) : number(num) {}
    virtual void funcBack() { cout << "base func" << endl; }

   private:
    int number;
};


```

https://clang.llvm.org/extra/clang-tidy/

Clang-Tidy - Extra Clang Tools 15.0.0git documentation

Extra Clang Tools 15.0.0git documentation Clang-Tidy « Extra Clang Tools 15.0.0git (In-Progress) Release Notes :: Contents :: Clang-Tidy Checks » Clang-Tidy ¶ Contents Clang-Tidy Using clang-tidy Supp