google c++ style guide

一. 背景

在大多数的 Google 开源项目中，c++是最主要的开发语言。每个 c++程序员都知道，c++语言有很强大的语言特性，但是会导致程序的复杂度提升，在编程的过程中容易出现 bug【bug-prune】，并且难以维护【maintain】。

这个文档通过详细的描述该做【dos】与不该做【don't s】来进行管理代码的复杂度。在兼顾代码库易维护【manageable】的同时，保证 c++程序员在使用 c++特性的时候可以得心应手【productively】。

风格【style】,也叫代码的可读性，通常也叫作惯例【conventions】为了去管理 C++代码，.风格这个词有些不恰当【misnomer】，因为这些惯例不仅仅包括源码文件的格式。

大多数的 Google 开源项目都会遵循【conform】文档中描述的风格。

注意：这个文档不是 c++的使用说明书，我们默认读者对 c++足够熟悉

1.1 目的

为什么我们需要提供这个文档？

该文档需要提供【serve】一些主要的目标。找到每个单一规则的最原始的出发点。通过对每一个规则适用性【in place】进行详细的讨论，并且得出相应的结论【decision】.如果读者已经理解了每一个规则所要达到的目的，那么在规则被遗弃，或者某个规则被改变的时候，读者也会理解这样做的意图。

正如你看到的这个文档的目的如下：

- **规则应起作用【pull their weight】**

  遵守规则所带来的益处需足够的大，从而可以证明让所有的工程师去记住这样的规则是正确的。规则大多数解释我们不用相应的特性，而不是我们使用相应的特性。比如说 goto 违反【contravenes】了许多下面相应的规则，但是 goto 已经被遗弃，因此该文档将不会考虑讨论它。

- **从读者的角度进行优化**

  通常期望代码库（大多数的子模块【individual components】提交到代码库中）可以使用很长时间.因此，大多数时间会去读相应的 code，而不是去写。我们明确的【explicitly】去选择优化中等水平的软件工程师的读代码，维护【maintain】，和 debug 代码的能力，而不是去优化编写所述 code 的难度。“给读者留下线索”是这些准则的另外一个特别【particularly】目的。当代码库中有一段【snippet】精妙或者不寻常【unusual】

的 code(比如说，指针 ownership 的转换)，留下相应的文字提示【textual hints】是非常重要的(std::unique_ptr 表明了 ownership 转换的无二义性【unambiguously】)。

- **与现有的 code 保持一致性**
- **在必要的情况下与 c++社区保持一致**
- **避免奇怪的【surprising】或者危险的设计**
- **避免设计出中等 engineer 难以维护的代码**
- **注意【mindful】代码库的大小**
- **在必要的情况下需要考虑优化**

性能优化【performance optimization】有些时候是必要的，尽管同时会与这个文档中一些准则会有冲突。

这个文档的意图是提供最大的指导，通过合理的约束【reasonable restriction】。通常编程常识或者效果好的习惯【good taste】将被接受并且盛行【prevail】。我们特指的是在整个 Google c++社区中的建立的惯例，而不是你的个人或者某个团队的偏好。如果在使用精妙的【clever】或者不寻常的观念【constructs】表示怀疑【skeptical】或者不情愿【reluctant】，没有禁止【prohibition】并不以为这允许【proceed】，通过自己的判断，如果还不确信，可以向 project leader 进行确认得到额外的解释。

1.2 c++版本

当前 code 主要面向于 c++17,而不是 c++2x 中的特性。随着时间的推移，这个文档面向的 c++版本将会进行修改【advance】.

不要去使用非标准的扩展【extensions】.

在使用 c++14 和 c++17 中的特性之前，应该认真考虑转换到其他环境时的便携性【portability】。

二. 头文件

通常，每一个.cc 文件都应有相关的【associated】.h 文件。当然会有一些例外【exceptions】，例如单元测试和仅仅含有 main 函数的.cc 文件。

正确的使用头文件将对代码可读性，文件大小，性能产生很大的影响【make a huge difference to】。

下面的规则将会带领你认识使用头文件所遇到的陷阱【pitfall】。

2.1 self-contained 头文件

2.2 头文件保护符【guard】

所有的头文件都应含有#define 保护，用来防止多重引入【multiple inclusion】,标识符的命名格式应为 **<PROJECT>_<PATH>_<FILE>_H_**.

为了保证唯一性【uniqueness】，标识符的命名应根据项目源文件所在的全路径。例如，文件/scr/bar/baz.h 位于项目 foo 下，则该头文件应有下面的头文件保护符。

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif    // FOO_BAR_BAZ_H_
```

说明：第二种方法：#pragma once

2.3 前向声明

避免使用前向声明，取而代之的是使用#include 引入需要的头文件。

2.4 内联函数

当函数为 **10 行或者低于 10 行**，可以定义该函数为内联函数。

定义：可以声明函数为内联函数，从而允许编译器通过内联扩展，而不是像通常函数那样进行调用。

优点：内联函数可以生成更高效的目标代码，只要被内联的函数足够的小巧。放心的去内联 accessors,mutators，以及其他小巧的重要的性能函数【通常为 set 和 get 方法】。

缺点: 过度【overuse】的使用内联可能会导致程序变慢。根据函数大小的不同，内联可能会导致代码增加或者较少，内联 accessor 函数通常将会减少代码的大小，如果内联一个很大的函数，将显著【dramatically】增加代码的大小。因为更好的使用指令缓存【instruction cache】,现代的处理器在运行更少的代码将会更快。

结论：

经验 1：一个比较好【decent】的经验是低于 10 行的函数可以将其内联。特别注意析构函数，因为隐式的成员和调用基类析构函数，所以析构函数实际上要比看起来要长。

经验 2：通常【typically】不要内联带有 loop 或者是 switch 语句，除非大多数情况下，它们不会被执行【excuted】。

虽然将一个函数定义为内联函数，但是这些函数通常不会被内联。例如，虚函数和递归函数通常不会被内联。通常**递归函数不应该【should not】**被内联.最主要的原因将虚函数放到类的定义中，是为了方便【convenience】或者为了记录他们的行为，如 accessors 和 mutators.

注：在使用虚函数内联需要注意以下几点

1. 虚函数可以是内联函数，内联可以修饰虚函数的，但是当虚函数表现多态性的时候，不能进行内联。

2. inline virtual 唯一可以内联的时候是：编译器知道所调用的对象属于哪个类，如Base::who()，这只有在编译器具有实际对象而不是对象的指针或者引用的时才会发生。

代码示例

2.5 include 顺序

按照如下的包含头文件顺序：相关的头文件，C 系统头文件，C++标准库头文件，其他库头文件，该项目头文件。

所有的项目头文件应按照项目的目录树进行排列，并且避免使用 UNIX 目录别名，如（.当前目录），（..父目录）。例如：google-awesome-project/src/base/logging.h 应该包含：

```
#include “base/logging.h”
```

文件 dir/foo.cc 或者 dir/foo_test.cc，两者的目的是实现或者测试 dir2/foo2.h 中的函数，包含的头文件应为：

1. dir2/foo2.h
2. **空行**
3. C 系统头文件（更确切的说，使用<>包含以.h 结尾的文件）如：<unistd.h>,<stdlib.h>
4. **空行**
5. C++标准库头文件（不需要扩展名）如：<algorithm>,<cstddef>
6. **空行**
7. 别的库.h 文件
8. 自己本身库.h 文件

使用空行将每一个非空的组分开。

使用上述顺序，如果相关的头文件 dir2/foo2.h 遗漏【omits】任何必要的包含文件，则文件 dir/foo.cc 或者 dir/foo_test.cc 的 build 过程会崩溃【break】.因此这个规则保证了，如果 build 过程崩溃一定是目前的工作文件出现了问题，而不是别的包。

dir/foo.cc 和 dir2/foo2.h 通常在同一个目录下（如，base/basictypes_test.cc 和 base/basictypes.h），但是通常也可能不在同一个目录下。

注意 C 头文件（如，stddef.h）基本上【essentially】都是可以与 C++系统头文件（cstdef）交换的【interchangeable】。两种风格都是可以接受的，但是更倾向于和当前的代码库相一致【consistency】.

包含的头文件中，每一组都应该按照字母表顺序【alphabetically】进行排列，注意如果以往的代码可能没有遵循这个规则，如果在方便的时候可以将其修改。

应该包含所有的依赖的头文件，前向声明除外（但是该特性不太建议使用），如果依赖 bar.h 中的标识【symbol】，如果还依赖于 foo.h,不要因为 bar.h 中已经依赖了 foo.h 就不去依赖他。除非 foo.h 明确的【explicitly】显示 bar.h 提供相应的标识。

举个栗子，google-awesome-project/src/foo.internal/fooserver.cc 所包含的文件如下

```
#include “foo/server/fooserver.h”

#include <sys/types.h>
#include <unistd.h>

#include <string>
#include <vector>

#include “base/basictypes.h”
#include “base/commandlineflags.h”
#include “foo/server/bar.h”
```

例外：在特定系统【system-specific】的代码中需要条件包含【conditional includes】，条件包含的代码应该放在其他包含文件后面，当然特定平台的代码需要尽量的小巧，独立。

```
#include “foo/public/fooserver.h”

#include “base/port.h”

#ifdef  LANG_CXX11
#include <initializer_list>
#endif  // LANG_CXX11
```

四. 类

4.1 构造函数的职责

4.2 隐式转换

4.3 可拷贝和可移动类型

4.4 结构和类

4.5 结构和 pairs 和 tuples

4.6 继承

4.7 操作符重载

4.8 存取控制

4.9 声明顺序

五. 函数

5.1 输出参数

在 C++中，函数的输出通常【naturally】通过返回值或者通过输出参数【output parameter】。

与输出参数相比更倾向于使用返回值，它可以增加可读性，经常会调高相应性能。如果仅仅提供输出参数，那么输出参数应出现在输出参数的后面。

参数要么是函数的输入，函数的输出或者两者都含有。输入参数通常是值【values】或者是 const 类型的引用【references】。输出参数或者输入/输出参数则是指向 non-const。

当对函数参数进行排序【order】的时候，将所有的输入参数放到任何输出参数的前面，不要将新的参数放到函数参数的最后，放置新的输入参数在输出参数前面。

这不是硬性规定【hard-and-fast rule】，输入/输出参数（通常为结构体/类）会将程序搞的很混乱【muddy the water】。但是通常与相关的函数保持一致性可能需要你变通【bend the rule】。

5.2 短函数

5.3 引用参数

5.4 函数重载

5.5 默认参数

六. google 的奇巧

6.1 ownership 和智能指针

6.2 c++格式检查工具