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

避免在构造函数中调用虚函数[1]，也不要在无法报出错误时候进行初始化。

定义：

- 在构造函数的函数体中可以进行任意的初始化【arbitrary】。
- 不用担心类是否已经被初始化
- 已经完全初始化的对象可以为 const,并且可以容易的被标准的容器和算法使用。

结论：

构造函数不能调用虚函数，在适当的情况下，终止程序是对错误比较合适的回应。否则需要考虑工厂函数，或者 Init()函数，在 TotW #42 有详细的讨论。Avoid Init()methods on objects with no other states that affect which public methods [3].

[1] 为什么不能在构造函数中调用虚函数?

答：在下面的例子中，创建子类对象的时候需要先调用基类的构造函数，然后调用子类的构造函数，所以当调用基类构造函数时，子类对象还没有完全的创建，所以只能将其当作是基类对象而不是子类对象，因此调用的是基类的 func 函数。

```
class Base {
 public:
    Base(){ func(); }
    virtual void func() { cout << "Base" << endl; }
};
class SubClass : public Base{
 public:
    SubClass(int a):Base(),a_(a){}
    virtual void func() { cout << "Case" << endl; }
 private:
     int a_;
};
int main() {
     SubClass* b = new SubClass(4);
     delete b;
     return 0;
}
```

C++之所以这样设计，是因为防止引入 bug，如果调用的是子类的 func 函数，假设子类的 func 函数使用了还未初始化的子类成员，那将会出现未知的后果。

[2] 什么叫工厂函数？

[3] 怎样翻译和理解这句话？

4.2 隐式转换

不要定义隐式转换【implicit conversions】，对于转换运算符或者单参数的构造函数，请使用 explicit 关键字。

定义：

隐式转换允许一个类型的对象（源对象）被作用为不同的类型（目标类型），如，当传入 int 参数给接受 double 参数的函数。

这段实际可以不看：除了【in addition to】语言定义的隐士转换之外，用户也可以自己定义，在源类型中定义隐式类型转换，可以通过目的类型名的类型转换运算符实现，在目的类型中定义隐士类型转换，则通过以源类型作为其唯一参数的构造函数实现[1]。

Explicit 关键字可以用于构造函数，或者转换操作符（从 c++11 后）如 cast，保证在目标类型在调用的时候是显式的写明的。Explicit 不仅用于隐式转换，而且可用于 C++11 的列表初始化语法上：

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

```
Fun({42, 3.14}); // error,这种调用是错误的，如果不加 explicit 可以正常调用。
```

从技术上讲上面不是隐式转换，但是语法上应该使用 explicit 对上面行为进行限制。

结论：

当调用只有一个参数的类型转换操作和构造函数，需要在类定义的时候标记为 explicit。特例是：拷贝和移动构造函数不应该用 explicit 修饰，因为他们不会执行类型转换。

不是一个参数的构造函数应该忽略 explicit。构造函数如果参数为一个 std::initializer_list，也应该忽略 explicit,用于支持拷贝-初始化（如：MyType m = {1,2}）.

[1] 怎么样实现隐式类型转换？

源类型-->目标类型

将 string 类型隐士转化为 dog 类型，只需要通过一个单参数的构造函数。

```
class dog {
public:
    dog(string name) {m_name = name;}
    string getName() {return m_name;}
private:
    string m_name;
};
int main() {
   string dogname = "dog";
   dog d = dogname;   // 隐式将 string 转化为 dog
   cout << "my name is " << d.getName() << endl;
   return 0;
};
```

目标类型-->源类型

只需要将 dog 类中加入，就可以使用（string）d 将 dog 转化为 string。

```
operator string () const { return m_name; }
```

4.3 可拷贝和可移动类型

类的程序接口必须说明白【make clear】类是否可以拷贝【copyable】,只能移动【movable】,或者两者不能。如果这些操作是有意义的，则可以支持拷贝或者移动。

定义：

可移动类型可以从临时变量进行初始化【initialized】或者赋值【assigned】.

可拷贝类型可以从其他同类型的比那辆进行初始化和赋值，并且约定【stipulation】源对象的值不会改变。std::unique_ptr<int> 可以移动但是不能拷贝类型（因为 std::unique_ptr<int>的源对象一定被改变）。int 和 std::string 都可以移动和拷贝。（对于 int 拷贝和移动是一样的[1]，std::string 移动操作比拷贝更高效）

对于用户定义类型，拷贝行为是由拷贝构造和拷贝赋值定义，移动行为由移动构造和移动赋值定义。移动的行为是由移动构造和移动赋值定义的，如果移动构造或者移动赋值不存在，由拷贝构造和拷贝赋值替代[2]。

拷贝构造可以由编译器隐式支持【invoked】,当通过值传递对象的时。

结论：

每一个类的公共接口必须明确类是否支持拷贝或者移动操作。在声明中的 public 区域应该显式的声明或者删除相应的操作。

具体的说，一个可拷贝的类应该显式声明拷贝操作，一个仅仅可移动的类应该显式声明移动操作。如果提供拷贝或者移动赋值操作，那么必须提供相应的构造函数。

```
Class Copyable {
 Public:
   Copyable (const Copyable& other) = default;
   Copyable& operator= (const Copyable& other) = default;
   // move 操作被上面的声明隐式支持【suppressed】
}

Class MoveOnly {
 Public:
   MoveOnly (MoveOnly&& other);
   MoveOnly& operator= (MoveOnly&& other);
   // 拷贝操作隐式的被删除，当然也可以显式的进行说明
   MoveOnly (const MoveOnly&) = delete;
   MoveOnly& operator= (const MoveOnly&) = delete;
}

Class NotCopyableOrMovable {
 Public:
   NotCopyableOrMovable (const NotCopyableOrMovable & other) = delete;
   NotCopyableOrMovable & operator= (NotCopyableOrMovable & other) = delete;
   // 移动操作隐士删除，但是可以显式的说明
   NotCopyableOrMovable (NotCopyableOrMovable &&) = delete;
   NotCopyableOrMovable & operator= (NotCopyableOrMovable &) = delete;
}
```

[1] 为什么 int 拷贝和移动是一样的？

[2] 移动行为如果没有移动构造或者移动赋值定义，则会由拷贝构造替代，也就是如果向使用移动构造，但是没有定义，那只能调用拷贝构造或者拷贝赋值。

4.4 结构和类

当被动对象【passive object】只携带数据时，可以使用 struct，其他使用 class.

struct 和 class 关键字在 c++中几乎是一样的，因此我们为每个关键字添加了自己的语义理解。以便在定义数据类型的时候选择合适的关键字。

Struct 应该只携带数据，或者相关的常量，除了存取函数外没有其他的函数。所有的成员必须是 public，并且直接存取而不是通过 getter/setter 方法。Struct 中不能包含属性之间的不变性，因为用户的直接存取会破坏这种性质。函数不能提供行为上的变化，只能去设定数据成员，如：构造函数，析构函数，Initialize(), Reset().

如果需要更多的函数功能和不变性，class 比较合适，如果不确定，那就是用 class.

与 STL 保持一致【consistency】,对于仿函数[1]可以使用 struct 替代 class。

注意成员变量在类和结构体中有不同的命名规则[2]。

[1] 仿函数使用 struct 类型？

[2] 类和结构体成员变量不同的命名规则？

类和结构体成员的成员变量命名，都应该使用小写，并且使用_连接不同的单词。但是 class 的成员变量需要在最后加下划线_。而 struct 不需要。

A_struct_data_number，a_class_data_number_

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

提倡短并且明确【focused】的函数。

我们发现长的函数有时是合适的【appropriate】,因此没有对函数的长度进行严格的限制。如果一个函数超过【exceed】了 40 行，那么需要考虑是否可以将其分开在不会影响【harm】到项目的情况下。

5.3 引用参数

所有的左值引用参数都必须使用 const 进行修饰。

定义：在 C 语言中，如果一个函数想去修改【modify】一个变量,那么参数必须使用指针，如：int foo(int* pval)。在 C++语言中，函数也可以申明一个引用参数：int foo(int& val)

优点：定义一个引用参数防止使用不太美观的代码如：（*pval）++,并且在 copy 构造的时候是必须的，更加明确的是，不会接收一个空的指针。（why）

缺点：引用是令人迷惑的，因为它有值变量，但是有指针的语义【semantics】。

结论：函数参数列表中，引用参数必须为 const:

```
Void Foo(const std::string& in, std::string* out)
```

在 Google 的代码库中，输入参数通常为值【values】或者是 const 修饰的引用变量，输出变量往往为指针【pointer】。输入参数可能为 const 的指针，但是不允许非 const 的引用参数除非是因为惯例，比如 swap()函数。

当然也有一些情况使用更推荐使用 const T*相比与 const T&，

- 可能会传递空指针

- 函数要把指针或者对引用赋值给输入形参

注：这里笔者的理解是，输入形参肯定为一个非 const 的指针，所以如果是将指针赋值给输入形参，如果传入的是 const T*就可以直接进行赋值，而将引用赋值给输入形参也是，如果传入指针就可以直接进行赋值给输入形参。

记住大多数时间输入参数将会指定为 const T&。如果使用 const T*则表明输入另有处理，所以如果选择 const T*而不是 const T&,应为为之写出具体的【concrete】理由。否则将会使读者迷惑。

5.4 函数重载

使用重载函数（包括构造函数）需要让读者看到调用就知道调用的是什么函数，而不需要花心思去了解到底哪个函数被调用。

定义：可以编写一个函数的参数为 const std::string&来进行重载 const char*，然而在这里例子中可以使用 std::string_view 进行代替。

```
class MyClass {
 public:
   void Analyze(const std::string& text);
   Void Analyze(const char* text, size_t textlen);
}
```

优点:重载可以是代码更加的直观简单【intuitive】,通过允许相同名字的函数右不同的参数。重载对于模板是必要的，并且对于访问者【Victors】非常的方便。详情请查看 TotW_148.

缺点：如果函数单纯的只靠参数的类型进行重载，那么读者只能先去理解 C++的复杂的匹配规则去理解到底发生了什么。并且如果子类只是重载了父类函数的部分变体，那么继承语义【semantics】会让人感到困惑。

结论：可能在重载一个函数的时候，各个变体之间没有任何语义【semantic】上的不同，他们可能通过类型，限定符【qualifiers: 如&，&&来限制只能左值调用或者右值调用】，或者参数的个数进行重载。如果一个读者读到一个函数重载而不必要知道哪个函数被调用，只知道有个函数被调用。如果你能在头文件中对每一个重载函数都增加一个注释，那么将是设计很好的【well-designed】重载函数集.

注：或者可以通过修改函数名增加参数信息，例如用 AppendString()和 AppendInt()等，而不是一口气重载多个 Append()。

5.5 默认参数

默认参数允许使用在非虚【non-virtual】函数上，如果定义在非虚函数上，必须保证

【guaranteed】参数值相同[1]。默认参数和函数重载有相同的限制【restrictions】,并且一般情况下更倾向于使用函数重载，如果使用默认参数所带来的可读性【readability】不会超过【outweigh】下面所提到的缺点【downsides】。

优点：

通常函数使用默认值，但是有时【occasionally】也使用非默认值。缺省值允许使用一个简单的方式，而不需要定义大量的函数去处理例外情况。

与函数重载相比，默认参数有更简洁的语法结构【syntax】，更少的样版代码[2]，并且可以区分‘必要参数’和‘可选参数’。

缺点：

默认参数是另外一种实现重载函数语义的方式,因此不要使用重载函数也同样使用与默认参数。

在虚函数中的默认参数的值是通过目标变量的静态类型决定的,因此不能保证所有的函数都声明相同的默认值。

默认参数在每次调用的时候，值都不是相同的，那么将会使生成的代码极具的膨胀【bloat】。而阅读代码希望默认参数值在声明的时候是固定的。

注：函数指针相关内容不去翻译。

结论：

默认参数不允许【banned】在虚函数中使用,因为在这情况下不能正确的运行。并且当不同的调用点默认参数都有不同的值的时候，也不允许使用。（如： 不要写 void f(int n = couter++) 这样的代码）。

在其他的情况下，如果使用默认参数可以提高可读性，并且足以克服上面提到的缺点，那么可以使用默认参数。如果还有疑问就使用函数重载。

[1] 为什么不能在虚函数上面定义默认参数呢？

[2] 为什么默认参数相比与重载函数有更少的样板代码？

六. google 的奇巧

6.1 ownership 和智能指针

对于动态申请的对象，更倾向有单个，固定的 owner。 建议使用智能指针来转移【transfer】所有权。

定义：ownership 是一种记账技术【bookkeeping】用来惯例动态分配的内存和其他的资源。动态分配的对象的 owner 在对象不在使用的时候，有责任【responsible】确保对象会被删除。ownership 有时候可以共享【shared】,通常最后的 ownership 需要删除对象。尽管如果对象只有一个 ownership，可以在代码中转移到其他的地方。

智能指针是一个像指针的类，通过重载运算符*和->。并且智能指针用来自动【automate】管理记账技术，从而确保在对象不在使用时进行删除。C++11 推出的智能指针 std::unique_ptr 类型表示对动态分配的内存有独占的【exclusive】作用。当 std::unique_ptr 离开了作用域，则对象将被删除。std:::unique_ptr 不能进行复制，但是可以通过 move 运算进行转移 ownership[1]。std::shared_ptrs 可以进行复制，ownership 在所有的备份中都可以共享，当最后一个 shared_ptrs 销毁的时候，对象将被删除。

优点：

- 如果没有管理 ownership 的相应逻辑，将动态内存管理好几乎【virtually】是不可能的。
- 转移 ownership 的代价要比复制一份要小的多（如果可以复制的话）。
- 转移 ownership 要比“借用”指针或者引用要简单的多，因为它大大减少了在两个用户时间协调【coordinate】对象的声明周期。
- 如果智能指针可以有清晰的逻辑，自己进行注释，并且没有二义性，那么可以大大增加代码的可读性。
- 智能指针可以进行手动消除记账功能，简化代码并且可以排除【ruling out】大量的错误。
- 对于 const 对象，共享所有权简单并且高效相对于深拷贝。

缺点：

- **ownership**
- **A**
- **B**
- 如果使用智能指针，那么不容易察觉资源何时释放。
- Std::unique_ptr 的所用权转移是使用 c++11 中的移动语义，这个概念相对来说比较新，可能会迷惑很多编程者。
- **C**

- **E**
- 在许多情况下（例如，循环引用）对象，那么对象将不会被删除[0]。
- 智能指针并不能完美的替代原始的指针

结论：

如果动态申请内存非常有必要，倾向于将所有权保持在分配者手中。如果其他的地方要访问这个对象，可以考虑传递拷贝，引用或者指针，这样可以避免所有权的转移[2]。建议使用 std::unique_ptr 显式的【explicit】转移所有权。例如：

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

如果没有一个特别的理由,不要共享所有权，即不要使用 shared_ptr。这样说的一个原因是避免拷贝操作[3]，如果性能提升特别明显，并且操作的对象是不可变的[4]（例如，std::shared_ptr<const Foo>）. 如果真的需要使用共享所有权的话，建议使用 std::shared_ptr.

不要使用 std::auto_ptr,用 std::unique_ptr 代替。

[0] 为啥循环引用后对象就不会被删除？

[1] 为什么 std::unique_ptr 不能进行复制？这样去转移相应的 ownership?

因为 **unique_ptr 拥有它所指向的对象**，在某一时刻，只能有一个 unique_ptr 指向特定的对象，因此不能进行拷贝和赋值。并且初始化只能使用直接初始化的方式。

```
Unique_ptr<int> up1(new int()); // ok, 直接初始化
Unique_Ptr<int> up2 = new int();  //错误，构造函数是 explicit 的
Unique_ptr<int> up3(up1);  //错误，不允许拷贝。
```

例外：虽然不可以进行拷贝，但是可以拷贝或者赋值一个将要被销毁的 unique_ptr

```
//返回局部对象的拷贝。
unique_ptr func(int a){
   unique_ptr<int> up(new int(a));
   return up;
}
```

[2] 怎样使用 unique_ptr 防止所有权的转移？

有两种方式可以防止所有权的转移：1. **传递引用** 2. **暂时移交所有权**

```
void   func1(unique_ptr<int>& up){
   cout<<*up<<endl;
}
```

```
unique_ptr<int> func2(unique_ptr<int> up){
   cout<<*up<<endl;
   return up;
}

unique_ptr<int> up(new int(10));

//传递引用，不拷贝，不涉及所有权的转移
func1(up)
//  暂时转移所有权，  函数结束时返回拷贝，重新收回所有权
up = func2(unique_ptr<int> (up.release));
```

[3] Shared_ptr 有拷贝操作吗？它不是使用引用计数的吗？为什么需要拷贝？

[4] 为什么需要操作的对象是不可变的呢？可能讨论这个也不是太有意义，但是还是需要讨论一下？

七． 其他 c++特性

7.1 右值引用

7.2 使用 const

在应用程序接口中（API）,只要不犯错就尽量使用 const,并且 constexpr 在有些时候会是更好的选择，

定义：在变量或者是参数前声明 const 表明变量不会被改变（如，const int foo）.类函数可以有 const 修饰词表明函数不会改变类成员变量的状态（如，class Foo { int Bar(char c) const; }）.

结论：只有是正确的【accurate】并且有意义【meaningful】，我们强烈建议使用 const 在应用程序接口中（如，函数参数，方法，非本地变量）。const 提供了一种编译器验证过的说明，哪些变量是可以进行修改的。并且有一致可靠的方式去区别读和写，对于写一个线程安全的代码是有用的，并且在其他场景中可以非常重要的【critical】.特别是：

- 如果函数可以保证不会改变传入的引用或者是指针，相应的函数参数应该是 const 引用(const T&)，或者是 const 指针(const T*)。
- 如果通过值进行传入参数，那么 const 对于调用是不影响的，所以**不建议**在声明中加 const(TotW #109)
- 声明函数为 const，除非他们会改变对象的状态（或者是允许使用者改变其状态，如，返回一个非 const 的引用），或者在并发中是不安全的[1]。

在局部变量中是用 const 是不支持的，也是不支持的。所有的类的 cosnt 操作应该是并发安全的，如果不可行，必须提示这是线程不安全的。

哪里需要加 const:

有些人将 const 放到后面如 int const *foo,而不是 const int* foo,不强制要求，但是需要保持代码一致性【consistent】.

[1] 为什么并发是不安全的？对线程不了解，但是可以了解一下？

[2] 那么传入一个 double 类型或者是 int 类型的值，需不需要加 const?

答： 按照上面的描述，应该不需要加 const，因为 double 前面加入 const 是没有意义的，因为传入的是实参的拷贝，函数里面改不改变该值是没有影响的，即是没有意义的【meanful】。

7.3 类型转换

使用 C++转换形式如 static_cast<float>(double_value),或者是算数类型【arithmetic】的大括号初始化如 int64 y = int64{1} << 42,不要使用类似这样的转换形式 int y = (int)x, 或者 int y = int(x),但是后者如果是类类型的构造函数是可以的。

定义： C++介绍了有别于 C 的转换语法，并且区分了不同的转换类型。

结论：不要使用 C 类型的转换，取而代之的是，当显式的转换需要使用 C++类型转换。

- 使用大括号【brace initialization】初始化去转换算数类型【arithmetic type】,如 int64{x},这是一种安全的方法，因为如果转换会导致精度上的损失，编译将不会通过，并且语法非常简明。
- 类型转换需要使用 static_cast 作为 C 类型转换的替代品【equivalent】，当上行转换【up-cast】一个指针，从一个类转换到父类。或者显式的【explicitly】将指针从父类转换到子类。但是必须保证对象就是一个子类的实例[1]。
- 可以使用 const_cast 将 const 修饰符去除
- 使用 reinterpret_cast 做不安全的转换，可以将指针类型，转换为另一种指针类型，或者指针和整数相互转换。仅当你对所作的一切了然于心时使用。

参见 RTTI section 了解 dynamic_cast

[1] static_cast 与 dynamic_cast 类型转换？

static 可以对基本数据类型进行转换，如将 int 转换为 char,把 char 转换为 int。但是这种转换需要开发者自己保证（这里表示需要保证数据的精度）。

static_cast 还可以用于类层次结构中父类和子类之间**指针和引用**的转换。这种转换可以分为上行转换和下行转换，上行转换是安全的，下行转换是不安全的。因为 static_cast 的转换是粗暴的，对于下行转换，将父类转换为子类，子类有独有的属性和方法，会导致程序崩溃。

Dynamic_cast 用法和 static_cast 一样，但是 dynamic_cast 将父类转换为子类，具有运行时类型检查，因此可以保证下行转换的安全性，何为安全性，即转换成功就返回转换后的正确类型指针，如果转换失败就返回 NULL,之所以说 static_cast 是不安全的是因为，即使转换失败也不会返回 NULL.

```
class Base{};
```

```
class Derived : public Base{};
int main() {
	Base* p= new Derived();
	Derived* pd1 = static_cast<Derived*>(p);   // 不会报错，所以不安全
	Derived* pd2 = dynamic_cast<Derived*>(p);   // 会报错，不能从父类转换到子类。
	return 0;
}
```

八． 命名规范

最重要的一致性规则是命名管理。在没有查找实体【entity】声明的前提下，名字的风格可以立即提醒我们实体是什么：类型，变量，函数，常量，宏定义，等等。大脑中的模式匹配引擎很大程度依赖于命名规则【rely a great deal on】。

8.1 通用的命名规则

使用命名进行优化可读性对于不同组的人是很清楚的。不要担心每一行的长度，因为使代码让新的读者立即明白是更重要的。

- **尽量少的使用缩写（特别是首字母缩写）。不要通过删除单词中的字母进行缩写**
- 按照经验，如果维基百科中有相应的解释，可以使用。
- 通常命名的描述需要和名字的作用域成正比。例如，n 对于 5 行的函数是可以的，但是对于一个类，n 的含义是模糊不清的【vague】。

```
Class MyClass {
 Public:
  Int CountFooErrors(const std::vector<Foo>& foos){
     int n = 0;   // 在这个函数中 n 的意义很明确
     for(const auto&foo : foos){
        ...
        ++n
     }
     return n;
  }
  Void DoSomethingImportant(){
       Std::string fqdn = ...;   //fqdn 在维基百科中有相应的解释
  }
Private:
 Const int kMaxAllowedConnections = ...;
}
```

反例：

```
Class MyClass {
 Public:
  Int CountFooErrors(const std::vector<Foo>& foos){
    Int total_number_of_foo_errors = 0 // 解释的太多有些多余
    for (int foo_index = 0; foo_index < foos.size(); ++foo_index){
      ++total_number_of_foo_errors;
    }
  }
  void DoSomethingImportant(){
    Int cstmr_id = ...;   // 这是不对的因为删除了单词的字母
  }
  Private:
   Const int kNum = ...; //作用域很大，但是解释不明确
}
```

注意广为所知的缩写是 OK 的，比如 i 用于迭代变量，T 作为模板参数。

模板参数应该根据类别遵循不同的命名风格，如类型模板参数应该遵循 type_name,非类型模板参数应该遵循变量命名[1]。

[1] 类型模板参数命名如下：类型模板参数为 template 关键字后面的类型。即使用驼峰规则，单词首字母进行大写。

```
template<typename AnyType>
```

非类型模板参数是为了模板定义一些常量，如 template<typename T, usigned size>,size 为非类型模板参数，则命名规则和变量命名一样。

8.2 文件命名

文件名应该全部为小写并且可以包含下划线（_）或者横线（-）。需要和项目使用的惯例保持一致。如果项目中没有统一的格式，**建议使用下划线（_）。**

下面是正确的命名:

- my_useful_class.cc
- my-useful-class.cc

- myusefullclass.cc
- Myusefulclass_test.cc　// _unittest 和_regtest 已经被放弃使用【deprecated】

C++文件应该以.cc 结尾，相应的头文件应该以.h 结尾

不要使用在/usr/include 已经存在的文件名，如 db.h.

通常【In general】,使文件名尽可能的明确，例如，使用 http_server_logs.h 而不是 logs.h。更一般的情况，文件应该成对出现，如 foo_bar.h 和 foo_bar.cc，用来定义类 FooBar。

8.3 类型命名

类型命名开头为大写字母，并且每个单词开头都应该大写，没有下划线。

所有类型有：类，struct，类型别名，枚举，和类型模板参数,都应该按照上面的命名方式。

```
// 类和 struct
Class UrlTable{ ...
Class UrlTableTester {
Struct UrlTableProperties { ...

// 类型定义
typedef    hash_map<UrlTableProperties*, std::string> PropertiesMap;
//使用别名
using PropertiesMap = hash_map<UrlTableProperties*, std::string>;
// 枚举
Enum UrlTableErrors { ....
```

8.4 变量命名

变量的名字（包括函数参数）和数据成员都应该为小写，单词之间使用下划线进行连接。类的数据成员（不是 struct）应该还有末尾的下划线。如： a_local_variable, a_struct_data_number,a_class_data_number_。

8.4.1 普通变量名

例如：

```
std::string table_name; //小写使用下划线连接
```

反例：

```
std::strng tableName; //错误
```

8.4.2 类数据成员

类的数据成员不管是 static 和 non_static,和普通的非成员变量命名一样，但是需要在后面加上下划线【underscore】。

```
class TableInfo {
  private:
    std::string table_name_; //正确
    Static Pool<TableInfo>* pool_;
}
```

8.4.3 struct 数据成员

struct 的数据成员，不管是 static 和 non_static，命名和普通非成员变量一样，但是在最后不需要加下划线。

```
Struct UrlTableProperties {
    std::string_name;
    int num_entries;
    static Pool<UrlTableProperties>* pool;   //不需要加下划线。
}
```

8.5 类数据成员命名

8.6 Struct 成员命名

8.7 常量

8.8 函数命名

8.9 命名空间【Namespace】命名

8.10 枚举类型命名

8.11 宏指令【Macro】命名

九. 注释

十. 格式