CPP-Style-规范

2022年11月14日　　13:55

==========

==========

https://www.cnblogs.com/isLinXu/p/14598270.html

它的原文是　http://github.com/zh-google-styleguide/zh-google-styleguide
不过 里面的网址 不太好访问。。。

C++ 有很多强大的特性,但这种强大不可避免的导致它走向复杂，使代码更容易产生 bug,难以阅读和维护.

本指南的目的是通过详细阐述 C++ 注意事项来驾驭其复杂性.
这些规则在保证代码易于管理的同时, 也能高效使用 C++ 的语言特性.

使代码易于管理的方法之一是加强代码一致性.
让任何程序员都可以快速读懂你的代码这点非常重要.
保持统一编程风格并遵守约定意味着可以很容易根据 "模式匹配"规则来推断各种标识符的含义.

C++是一门包含大量高级特性的庞大语言. 某些情况下,
我们会限制甚至禁止使用某些特性. 这么做是为了保持代码清爽,
避免这些特性可能导致的各种问题. 指南中列举了这类特性,
并解释为什么这些特性被限制使用.

Google 主导的开源项目均符合本指南的规定.

1. 头文件

通常每个 .cc 文件都有一个 对应的 .h 文件， 也有一些常见例外， 如 单元测试代码 和只包含 main() 函数的 .cc 文件

正确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.

下面的规则 将引导你 规避 使用头文件时的 各种陷阱

1.1 self-contained 头文件

头文件应该能够 自给自足 (self-contained，也就是 可以作为 第一个头文件 被引入)， 以 .h 结尾。

至于 用来插入 文本的 文件，说到底 它们并不是 头文件，所以应该 以 .inc 结尾。

不允许 分离出 -inl.h 头文件做法。

所有头文件要能够自给自足。即，用户和 重构工具 不需要 为特别场合 而 包含 额外的 头文件。
详细来说，一个头文件 要有 define-guard， 统统包含 它所需要的 其他头文件，也不要求定义 任何 特别 symbol。

不过有一个例外，即一个文件 并不是 self-contained 的，而是作为 文本 插入到 代码某处。 或者，文件内容 实际上 是其他 头文件的 特定平台 (platform-specific) 扩展部分。这些文件 要用 .inc 扩展名。

如果 .h 文件 声明了 一个 模板 或 内联函数，同时 也在 该文件 加以定义。
凡是有用到这些的 .cc 文件，就得 统统包含 该头文件，否则 程序 可能会在 构建中 链接失败。

不要把这些定义 放到 分离的 -inl.h 文件中。 (该规范 过去曾 提倡 把 定义放到 -inl.h)

有个例外，如果某函数模板行为 所有相关模板参数 显示 实例化，或本身 就是 某类的 一个私有成员，那么 它就只能定义在 实例化 该模板的 .cc 文件里。

1.2 #define 保护

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：<PROJECT>_<PATH>_<FILE>_H_

为了保证唯一性，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 中头文件 foo/src/bar/baz.h 可以按如下方式 保护

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

1.3 前置声明

尽可能避免使用前置声明，使用 #include 包含需要的 头文件即可

定义：

所谓[前置声明] (forward declaration) 是类，函数 和模板的 纯粹声明，没伴随着其定义

优点

前置声明可以节省编译时间，多余的 #include 会 迫使 编译器展开 更多的文件，处理更多的输入。

前置声明 能够节省不必要的 重新编译的时间。#include 使代码 因为头文件中无关的改动 而被重新编译多次。

缺点

前置声明 隐藏了 依赖关系，头文件改动时，用户的代码 会跳过必要的 重新编译过程。

前置声明可能会被 库的 后续 更改 所破坏

前置声明函数 或模板 有时会妨碍头文件 开发者 变动其 API。

例如扩大 形参类型，加个 自带默认参数的 模板形参等

前置声明来自 命名空间 std 的 symbol 时， 其行为 未定义。

很难判断 什么时候该用前置声明，什么时候该用 #include。

极端情况下，用前置声明代替 #include 甚至 会 暗暗地 改变 代码的含义

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

// good_user.cc:
#include "b.h"
void f(B*);
void f(void*);
void test(D* x) { f(x); }  // calls f(B*)
```

如果 #include 被 B 和 D 的前置声明替代，test() 就会调用 f(void*) 。

前置声明了 不少来自 头文件 的symbol 时， 就会比 单单一行的 include 冗长。

仅仅为了能前置声明 而重构代码（比如 用 指针成员 代替 对象成员）会使得 代码 变成 更慢更复杂。

结论：

尽量避免 前置声明 那些定义在 其它项目中的 实体

函数： 总是 使用 #include

类模板： 优先使用 #include

1.4 内联函数

只有当 函数 行数 小于等于10行 时 才 定义为 内联函数

只有函数被声明为 内联函数之后，编译器才会 将其内联展开，而不是 按通常的 函数调用机制 进行 调用。

优点：

只要内联的 函数体 较小，内联该函数 可以 令目标 代码更加高效

对于 存取函数 及 其他函数体比较短，性能关键的 函数，鼓励使用 inline

缺点

滥用 内联 会导致 程序 更慢，内联可能使 目标代码量 或增 或减，这取决于 内联函数

的 大小，内联非常短小的 存取函数 通常 会减少 代码大小，但内联一个 相当大的函数 会 增加代码大小， 现代处理器由于 更好地利用了 指令缓存， 小巧的代码 往往执行更快。

结论：

一个较为合理的 经验准则是， 不要内联 超过10行的 函数， 谨慎对待 析构函数，析构函数 往往 比其 表面 看起来 要更长，因为有 隐含的 成员 和 基类 析构函数 被调用。

另一个 经验准则是： 内联那些 包含循环 或 switch 的 函数 往往 得不偿失 (除非 在大多数情况下，这些循环 或switch 从不执行)

有些函数即使声明为inline 也不一定会被 编译器 inline。 比如 虚函数，递归函数 通常 就 不会被 内联。

通常 递归函数 就不应该声明为 inline (注：递归调用 堆栈的 展开 并不像 循环那样简单，比如递归层数 在编译时 可能是未知的，大多数编译器都不支持 内联递归函数)

虚函数内联的 主要原因是 想把它的 函数体放在类定义内，亦或是 当做文档描述其行为，如 存取函数

1.5 #include 的路径及顺序

使用标准的头文件包含顺序 可以增强可读性，避免隐藏依赖： 相关头文件，C库，C++库，其他库的.h，本项目的.h

项目内 头文件应该按照 项目源代码目录树结构排列，避免使用unix 特殊的 快捷目录 . 或 .. 。

例如，google-awesome-project/src/base/logging.h 应该 按如下方式包含

```
#include "base/logging.h"
```

又如， dir/foo.cc 或 dir/foo_test.cc 的 主要作用 是 实现或 测试 dir2/foo2.h 的功能， foo.cc 中 头文件 顺序如下

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

这种顺序保证 当dir2/foo2.h 遗漏 某些必要的库时， dir/foo.cc 或 dir/foo_test.cc 的 构建会立刻中止。 因此 这一条规则 保证维护 这些文件的 人 首先看到 构建中止的消息 而 不是维护其他包的人。

dir/foo.cc 和 dir/foo2.h 通常位于 同一目录下

你所依赖的 符号(symbol) 被哪些头文件 所定义，你就 应该 include 哪些头文件， 前置声明 情况除外。 比如你要用到 bar.h 中的某个符号，哪怕你所包含的 foo.h 已经包含了 bar.h， 也照样得包含 bar.h， 除非 foo.h 明确说明 它会自动向你 提供 bar.h 中的 symbol。 不过凡事 cc文件 所对应的 相关头文件 已经包含的，就不用再重复 含进其 cc 文件中了， 就像 foo.cc 只包含 foo.h 文件就足够了，不用再管 后者所包含的其他内容。

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"
#include "base/commandlineflags.h"
#include "foo/public/bar.h"
```

例外

有时，平台特定 代码需要条件编译，这些代码可以放到 其他 include 之后。当然，你的平台特定代码 也要足够 简练 且独立：

```
#include "foo/public/fooserver.h"

#include "base/port.h"  // For LANG_CXX11.

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

2.1 命名空间

鼓励在 .cc 文件内使用 匿名命名空间 或 static 声明

使用 具名的命名空间时，其名称 可基于项目名 或 相对路径。

禁止 using 指示 (using-directive)。

禁止使用 内联命名空间(inline namespace)

定义

命名空间将全局作用域细分为 独立的，具名的作用域， 可以有效防止全局作用域的 命名冲突。

优点

虽然类已经提供了 (可嵌套的) 命名轴线 (注：将命名分割在不同类的 作用域内)，命名空间在这基础上又封装了一层。

距离来说，2个不同项目 的全局作用域 都有一个类 Foo，这样在 编译 或运行时 造成冲突，如果每个项目 将代码置于不同的命名空间中， project1::Foo, prohect2::Foo 作为不同符号自然不会冲突。

内联命名空间会自动把 内部标识符 放到 外层作用域：

```
namespace X {
    inline namespace Y {
        void foo();
    }  // namespace Y
}  // namespace X
```

X::Y::foo() 和 X::foo() 是等价的

内联命名空间主要用于 保持跨版本 的 ABI 兼容性

缺点

命名空间具有迷惑性，因为它们 使得 区分2个相同命名所指代的 定义更加困难。
内联命名空间 很容易令人迷惑，毕竟其 内部的成员 不再受其 声明所在 的命名空间的限制。内联命名空间 只在 大型版本控制中 有用。
有时候，不得不多次引用 某个定义在许多 嵌套命名空间中的 实体，使用完整的命名空间会导致代码的冗长。
在头文件中使用 匿名空间 导致违背了 C++ 的唯一定义原则 (One Definition Rule (ODR))

结论

根据下文要提到的 策略合理使用 命名空间
遵循 命名空间命名 中的规则
像之前的几个例子中一样，在命名空间的 最后注释出命名空间的名字
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，以区别于其他的命名空间：

```
// .h 文件
namespace mynamespace {

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

} // namespace mynamespace


// .cc 文件
namespace mynamespace {

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

} // namespace mynamespace
```

更复杂的 .cc 文件包含更多，更复杂的细节，比如 gflags 或 using 声明

```
#include "a.h"

DEFINE_FLAG(bool, someflag, false, "dummy flag");

namespace a {

...code for a...                // 左对齐
```

```
} // namespace a
```

不要在头文件中使用 命名空间别名， 除非显式标记内部命名空间使用。因为任何在头文件中引入的 命名空间 会成为 公开API 的一部分。

```
// 在 .cc 中使用别名缩短常用的命名空间
namespace baz = ::foo::bar::baz;

// 在 .h 中使用别名缩短常用的命名空间
namespace librarian {
namespace impl {  // 仅限内部使用
namespace sidetable = ::pipeline_diagnostics::sidetable;
}  // namespace impl

inline void my_inline_function() {
  // 限制在一个函数中的命名空间别名
  namespace baz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

禁止 内联命名空间

2.2 匿名命名空间和静态变量

在 .cc 文件中定义一个不需要 被外部引用的变量时， 可以将它们放在 匿名命名空间 或 声明为 static。 但 不 要 在 .h 中 这样 做

定义

所有 至于 匿名命名空间 的声明 都 具有 内部链接性， 函数 和 变量 可以经 由 声明为 static 拥有内部链接性， 这意味着 你在这个 文件中 声明的 这些 标识符 都不能在 另一个文件中 被访问。 即使 2个文件 声明了 完全一样 名字的 标识符， 它们所指向的 实体 实际上 是完全不同的。

结论：

推荐，鼓励， 在 .cc 中 对于 不需要再 其他地方引用 的 标识符 使用内部链接性声明， 但 不要在 .h 中使用。

匿名命名空间 的声明 和 具名 的格式相同，在最后注释上 namespace

```
namespace {
...
}  // namespace
```

2.3 非成员函数，静态成员函数，全局函数

使用静态成员函数 或 命名空间 内的 非成员函数，尽量 不要用 裸的全局函数

将一系列函数 直接置于 命名空间中，不要用类的 静态方法 模拟出 命名空间的 效果，类的 静态方法 应当和 类的实例 或 静态数据 紧密相关。

优点

某些情况下，非成员函数 和 静态成员函数 是非常有用的。
将非成员函数放在 命名空间 中 可以 避免污染 全局作用域

缺点

将 非成员函数 和 静态成员函数 作为 新类的成员 或许更有意义
当它们需要访问 外部资源 或 具有 重要的 依赖关系时 更是如此。

结论

有时，把函数的定义 同 类的实例 脱钩 是有益的，甚至是 必要的。这样的函数 可以被定义为 静态成员 或 非成员函数。 非成员函数 不应该依赖于 外部 变量，应该尽量至于某个 命名空间中。 相比 单纯为了封装 若干 不共享 任何 静态数据的 静态成员函数 而 创建类，不如使用 命名空间。
例如，对于头文件 myproject/foo_bar.h， 应该使用：

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

而非

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

定义在同一编译单元 的函数，被其他编译单元 直接调用 可能会引入 不必要的 耦合 和 链接时依赖。
静态成员函数 对此 尤其敏感，可以考虑提取到 新类中， 或者将 函数 置于独立库 的 命名空间中。
如果你必须定义 非成员函数，又 只是在 .cc 文件中 使用它，可以使用 匿名 命名空间 或 static 链接关键字 (如 static int foo() { .. }) 限定其作用域

2.4 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 的任何地方 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，应该使用 初始化的 方式 替代 声明+赋值：

```
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
```

属于 if， while ，for 的 变量 应该在 这些语句中 正常滴声明，这样 这些变量的作用域 就被 限制在这些语句中了， 如

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

警告

有一个例外，如果变量是 一个对象，每次进入作用域 都要调用 构造器，出作用域 要调用 析构器，这会导致低效

```
// 低效的实现
for (int i = 0; i < 1000000; ++i) {
    Foo f;                  // 构造函数和析构函数分别调用 1000000 次!
    f.DoSomething(i);
}

Foo f;                      // 构造函数和析构函数只调用 1 次
for (int i = 0; i < 1000000; ++i) {
    f.DoSomething(i);
}
```

2.5 静态 和 全局变量

禁止定义 静态 存储周期 非 POD 变量，禁止使用 含有 副作用的 函数 初始化 POD 全局变量，因为 多编译单元中的 静态变量 执行时的 构造 和 析构 顺序 是 未明确的，这将导致 代码 的不可移植

POD：plain oid data，即 int char float

禁止使用 类的 静态储存周期 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，它们不涉及 动态初始化 或 析构。

静态生存周期的 对象，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)：即int char float，以及 POD 类型的指针，数组 和 结构体。

静态变量的 构造器，析构器 和初始化顺序 在 C++ 中 只有部分是明确的， 甚至随着 构建变化 而变化， 导致 难以发现的 bug。

所以除了禁用 类 类型的 全局变量，我们也不允许 用 函数返回值 来 初始化 OID 变量， 除非 该函数 (比如 getenv()， getpid()) 不涉及 任何 全局变量。

函数作用域 里的 静态变量除外，因为 它们的 初始化顺序 是有明确定义的， 而且 只会在指令 执行到 它的声明那里才会发生。

注：

同一个编译单元内 是明确的， 静态初始化 先于 动态初始化，初始化顺序 按照 声明顺序进行， 销毁则 逆序。 不同的编译单元之间 初始化 和 销毁顺序 属于 未明确行为。

同理，全局 和 静态变量 在 程序中断时 会被 析构，无论所谓中断 是从 main() 返回 还是

对 exit() 的 调用。
析构顺序 正好和 构造器 调用顺序相反。 但既然 构造顺序未定义，那么 析构顺序自然也是不定的。 比如，在程序结束时 某个静态变量 已经被析构，但 代码还在跑 -- 比如其他线程 -- 并试图访问它 且失败。 在比如，一个静态string 变量 也许会在 一个 引用了 前者的 变量 析构之前 被 析构掉。

改善以上 析构问题的 办法之一是 用 quick_exit() 来 代替 exit() 并中断程序。它们的 不同之处 是 前者 不会 执行任何 析构， 也不会执行 atexit() 所绑定的 任何 handlers， 如果你想在 执行 quick_exit() 来中断时 执行某个 handler(比如刷新log)， 你可以把它 绑定到 _at_quick_exit()。 如果你想在 exit() 和 quick_exit() 都用上该 handler， 就都绑上去。

综上所述，我们只 允许 POD 类型的 静态变量，即 完全禁止 vector(使用C数组代替) 和 string (使用 const char [])

如果你确实 需要一个 class 类型的 静态 或全局变量， 可以考虑在 main() 函数 或 pthread_once() 内初始化 一个 指针 且永不回收。 注意只能用 raw 指针，别用智能指针，毕竟 后者的 析构函数 涉及到上文 指出的 不定顺序 问题。

> 注
> 上文提及的 静态变量 泛指 静态生存周期的 对象，包括 全局变量，静态变量，静态类成员变量， 函数静态变量。

类 是 C++ 中 代码的 基本单元。 下面列举 在写一个类 时 主要注意事项
3.1 构造函数的职责
总述

> 不要在 构造函数中 调用 虚函数，也不要在 无法报出错误时 进行 可能失败的 初始化。

定义

> 构造器中可以进行 各种初始化操作

优点

> 无需考虑 类是否被初始化
> 经过 构造器 完全初始化后的 对象 可以 为 const 类型，也能更方便地 被 标准容器或算法 使用。

缺点

> 如果在 构造器中 调用了 自身的 虚函数，这类调用 是不会 重定向 到子类的 虚函数实现，即使 当前 没有子类化实现，将来 仍是隐患。
>
> 在没有使 程序崩溃 (因为并不是 一个 始终合适的方法) 或者 使用 异常 (因为已经禁用了 <exceptions>) 等方法的条件下，构造函数 难以上报错误。
>
> 如果执行失败，会得到一个 初始化失败的 对象，这个对象可能进入 不正常 的状态，必须使用 bool isValid() 或 类似的机制 才能检查出来， 然而 这是一个 容易被忽略的方法。

构造函数的 地址 是无法被取得的， 因此，举例来说，由 构造函数 完成的 工作 是无法 以简单的方式 交给 其他线程的。

结论

构造器 不允许 调用 虚函数。 如果代码允许， 直接终止程序 是一个 合适的 处理错误的方式。苟泽，考虑用 Init() 方法 或 工程函数。

构造器 不得调用 虚函数，或尝试 报告 一个 非致命错误。
如果对象 需要进行 有意义的(non-trivial) 初始化， 考虑使用 明确的 init() 或 工厂模式。 Avoid Init() methods on objects with no other states that affect which public methods may be called (此类形式的半构造对象有时无法正确工作).

3.2 隐式类型转换

总述

不要定义隐式类型转换。对于转换运算符 和 单参数构造器，请使用 explicit

定义

隐式类型转换 允许 一个 某种类型 的对象 被用于 需要 另一种类型 的位置。例如 将一个 int 类型 的实参 传递给 double 类型的 函数。

除了语言定义的 隐式类型转换，用户还可以通过 在 类定义中 添加 合适的 成员 定义自己需要的 转换。
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 (例如 operator bool())。在目的类型 中 定义 隐式类型转换，则通过 以 源类型作为 唯一参数 (或 唯一 无默认值的参数) 的构造器实现。

explicit 可以用于构造器 或 (C++11引入)类型转换运算符，以保证只有当 目的类型 在调用点 被显式写明时 才能进行类型转换，例如 使用 cast。 这不仅用于 隐式类型转换，还能作用于 C++11 的列表初始化语法：

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

此时，下面的代码 是 非法的

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

这一代码 从技术上 说 并非 隐式类型转换，但是 语言标准认为 这是 explicit 应当限制的行为。

优点

有时 目的类型名 是一目了然的， 通过 避免 显式写出类型名，隐式类型转换 可以让一个类型的 可用性 和 表达性 更强。
隐式类型转换 可以简单地 取代 函数重载。
在初始化对象时， 列表初始化语法 是一种 简洁明了的写法

缺点

隐式类型转换 会隐藏 类型不匹配的错误， 有时，目的类型 并不符合 用户预期，甚至用户 根本没有意识到 发生了 类型转换

隐式类型转换会让 代码难以阅读，尤其是在 有函数重载的时候，因为 这时很难判断 到底是 哪个函数 被调用。

单参数构造器 可能被 无意中 用来 隐式转换

如果 单参数构造器 没有加上 explicit，读者 无法判断 这一 函数 究竟是 作为 隐式类型转换，还是 作者忘记加上 explicit。

没有明确的方法用来判断 哪个类 应该提供类型转换，这会使得 代码变得含糊不清。

如果目的类型是 隐式指定的，那么 列表初始化 会出现 和 隐式类型 转换 一样的问题，尤其是在 列表中 只有一个元素的 时候。

结论

在类型定义中， 类型转换运算符 和 单参数构造器 都应该用 explicit 标记。 一个例外是，拷贝 和 移动构造函数 不应该 被标记为 explicit，因为 它们并不执行类型转换。

对于设计目的 就是用于 对 其他类型 进行 透明包装的 类来说，隐式类型转换 有时 是必要 且 合适的。这时应当联系 项目组长 并说明 特殊情况。

不能以 一个参数进行 调用 的 构造器 不应该 加 explicit，接受一个 std::initializer_list 作为参数的 构造器 也 应该 省略 explicit，以便支持 拷贝初始化 (如 MyType m = {1,2};)

3.3 可拷贝类型 和 可移动类型

总述

如果你的类型需要，就让它们 支持 拷贝、移动，否则，就把 隐式产生的 拷贝 和 移动函数 禁用。

定义

可拷贝类型 允许 对象 在初始化时 得到 来自 相同类型的 另一个对象 的值，或者在赋值时 被赋予相同类型的 另一对象的值，同时不改变 源对象的 值。 对于 用户定义的类型，拷贝操作 一般通过 拷贝构造器 和 拷贝赋值操作符 定义。 string 类型 就是一个可拷贝类型的 例子

可移动类型 允许 对象 在初始化时 得到 来自 相同类型的 临时对象的 值，或在 赋值时 被赋予 相同类型的 临时对象的 值 (因此，所有 可拷贝对象 也是 可移动的)，std::unique_ptr<int> 就是一个 可移动 但不可复制 的例子。

对于用户定义的类型，移动操作 一般是通过 移动构造器 和 移动赋值操作符 实现的。

拷贝、移动构造函数 在 某些情况下 会被 编译器 隐式调用。例如，通过 传值的方式传递对象。

优点

可移动 及 可拷贝类型的 对象可以通过 传值的方式 进行传递 或返回，这使得API 更简单，更安全 更通用。
和传指针 和 引用 不同，这样的 传递 不会造成 所有权，生命周期，可变性 等方面的混乱，也就 没有必要 在 协议中 予以明确。
这同时也防止了 客户端 与 实现 在 非作用域内的 交互， 使得它们 更容易 理解和维护。
这样的对象可以和 需要传值操作的 通用API 一起使用，例如大多数容器。

拷贝、移动构造函数 与 赋值操作 一般来说 要比 它们的 各种替代方案，比如 Clone(),CopyFrom(),Swap() 更容易定义，因为它们能通过编译器产生， 无论是 隐式 还是 通过 = default。
这种方式 很简洁，也爆炸 所有数据成员 都会被 复制。
拷贝 和 移动构造函数 一般 也 更高效。
因为 它们不需要 堆的分配 或者 单独的 初始化 和 赋值步骤，同时，对于 类似 省略不必要的拷贝(https://en.cppreference.com/w/cpp/language/copy_elision) 这样的优化，它们也更合适。

移动操作 允许隐式且高效地 将 源数据 转移出 右值对象。这有时 能让代码风格更加清晰。

缺点

许多类型都不需要拷贝，为它们提供拷贝操作 会让人困惑，也显得 荒谬且不合理。
单例(Registerer)，特定作用域相关的类型(Cleanup)，与其他对象实体紧密耦合的类型(Mutex) 从逻辑上来说 都不应该提供 拷贝操作。

为基类提供 拷贝/赋值 操作 是有害的， 因为在使用 它们时 会造成 对象切割。

默认的 或 随意的 拷贝操作实现 可能是不正确的，这往往导致 令人困惑 且 难以诊断的错误。

拷贝构造函数 是隐式调用的，也就是说，这些调用 很容易 被忽略，这会让人 困惑。同时，这从一定程度上说 会鼓励过度拷贝，从而 导致 性能上的问题。

结论

如果需要 就让 你的类型 可拷贝/可移动，作为一个经验法则，如果 对于你的用户来说 这个 拷贝操作 不是一眼就能看出来的，那么就不要 把类型设置为 可拷贝的。

如果让类型可拷贝，一定要同时给出 拷贝构造函数 和 赋值操作的 定义，反之亦然。

如果 让类型 可拷贝，同时 移动操作的 效率 高于 拷贝操作，那么就把 移动的 2个操作 (移动构造函数 和 赋值操作) 也给出定义。

如果类型不可拷贝，但是 移动操作的 正确性 对 用户 显然可见，那么把 这个类型设置为 只可移动 并定义 移动的 2个操作。

如果定义了 拷贝、移动操作，则要保证 这些操作的 默认实现是正确的。 记得时刻检查默认 操作的 正确性，并且 在文档中 说明 类是 可拷贝 且/或 可移动的。

```
class Foo {
 public:
  Foo(Foo&& other) : field_(other.field) {}
  // 差，只定义了移动构造函数，而没有定义对应的赋值运算符.

 private:
  Field field_;
};
```

由于存在对象切割的风险，不要为 任何有可能 有 派生类 的 对象提供 赋值操作 或者 拷贝/移动 构造函数 (当然，也不要 继承 有这样的 成员函数的 类)。
如果你的基类 需要 可复制属性，请提供一个 public virtual Clone() 和 一个 protected 的 拷贝构造函数 以供 派生类实现

如果你的类 不需要 拷贝、移动操作，请 显式地 通过 在 public 域中 使用 = delete 或 其他手段 禁用。

```
// MyClass is neither copyable nor movable.
MyClass(const MyClass&) = delete;
MyClass& operator=(const MyClass&) = delete;
```

3.4 结构体 vs 类

总述

仅当 只有数据成员时 使用 struct，其他 一律使用 class。

说明

在C++中 struct 和 class 几乎等价。
我们为这2个 关键字 添加 自己的 语义理解，以便 为 定义数据类型 时 选择 合适的关键字。

struct 用来定义 包含数据的 被动式对象，也可以包含 相关的常量，但除了 存取数据成员之外，没有别的 函数功能。并且 存取功能 是通过 直接访问 位域，而非 函数调用。 除了 构造器，析构器， Initialize(), Reset(), Validate() 等类似的 用于设定 数据成员的 函数外， 不能提供其他功能的函数。

如果需要更多的 函数功能，class 更合适。

如果拿不准，就用 class。

为了和 stl 一致，对于 仿函数 等特性可以不用 class，而是使用 struct。

注意，类 和 结构体 的成员变量 使用 不同的 命名规则。

3.5 继承

总述

使用组合 常常 比使用 继承更合理。如果 使用继承的话，定义为 public 继承。

定义

当子类继承 基类时，子类包含了 父基类 所有数据 及 操作的 定义。
C++ 实践中，继承 主要用于 两种场合：
实现继承，子类继承父类的实现代码；
接口继承，子类 仅 继承 父类的 方法名称。

优点

实现继承 通过 原封不动的 复用基类代码 减少了 代码量。
由于 继承是 在编译时声明， 程序员 和 编译器 都可以 理解相应 操作 并发现错误。
从编程角度而言，接口继承 是用来 强制类输出 特定的 API，在类没有实现 API 中 某个必须的 方法时，编译器同样会发现 并报告 错误。

缺点

对于实现继承，由于 子类的 实现代码 散布于 父类 和 子类 中，要理解 其实现 变得更加困难。
子类 不能重写 父类的 非虚函数，当然也就 不能修改 其实现。
基类 也可能定义了 一些数据成员，因此 还必须区分 基类的 实际布局。

结论

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 实现继承。 组合常常更合适一些，尽量做到 只在 is-a 的情况下 使用继承，其他情况(has-a) 使用 组合。

必要的话，析构函数声明为 virtual。 如果你的类 有虚函数，则析构函数 也应该是 虚函数。

对于 可能被子类访问的 成员函数，不要过度使用 protected 关键字。 注意 数据成员都必须是 私有的。

对于重载的 虚函数 或 虚析构函数， 使用 override， 或 (较不常用的) final 关键字显式 标记。 较早(早于C++11) 的代码可能会使用 virtual 作为不得已的 选项。因此，在声明重载时， 请使用 override，final，virtual 之一 进行标记。 标记为 override 或 final 的 析构器 如果不是 对 基类 虚函数的 重载的话，编译会报错。这些标记 起到了 文档的作用，如果省略，读者 将 不得不检查 所有父类，以判断 该函数是否是 虚函数。

3.6 多重继承

总述

真正需要用到 多重继承实现 的 情况 非常少，只有 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，其他基类 都是 以 Interface 为后缀的 纯接口类。

定义

多重继承 允许子类拥有 多个基类。要将 作为 纯接口的 基类 和 具有实现 的 基类 区分开来。

优点

相比单继承，多重继承 可以 复用更多的 代码。

缺点

真正需要 多重继承的 情况 非常非常少。
有时 多重继承 看起来是一个 不错的 方案，但是 你通常也可以找到一个 更明确的，更清晰的 不用 解决方案。

结论

只有当所有父类 除第一个外 都是 纯接口类 时， 才允许使用 多重继承。为了确保 它们是 纯接口，这些类 必须以 Interface 为后缀。

注意

对于该规则，window 有个 特例。

。。没说window特例是什么。。。。 而且 能不能 多继承的 类 设置为 final 类。 这样的话 ，继承树 就很简单，就 不会造成 混乱。

3.7 接口

总述

接口 是指 满足 特定条件的 类，这些类 以 Interface 为后缀 (不强制)

定义

当一个类满足 以下要求时，称之为 纯接口

只有 纯虚函数 (= 0) 和 静态函数 (除了下文提到的 析构函数)
没有 非静态数据成员
没有定义 任何 构造器，如果有，也不能带有参数， 且必须是 protected。
如果 它是一个 子类，也只能满足 上述条件 并以 Interface 为后缀的类 继承。

接口类 不能被直接实例化，因为 它声明了 纯虚函数。
为了确保 接口类的 所有实现 可以被正确销毁，必须为之声明 虚析构函数

优点

以 Interface 为后缀 可以提醒其他人 不要 为该接口 增加函数实现 或 非静态数据成员。 这一点 对于 多重继承 尤其重要。

缺点

Interface 后缀增加了 类名长度，为 阅读 和理解 带来不便。同时，接口属性 作为 实现细节 不应该暴露给 用户。

结论

只有在满足上述条件时, 类才以 Interface 结尾, 但反过来,满足上述需要的类未必一定以 Interface 结尾.

3.8 运算符重载

总述

除了少数 特定环境外，不要重载运算符，也不要创建 用户定义字面量。

定义

C++ 允许用户 通过 使用 operator 关键字 对内建运算符 进行重载定义，只要其中一个参数 是用户定义的 类型。

operator 还允许 用户 使用 operator"" 定义新的 字面运算符，并且 定义类型转换函数，例如， operator bool()

优点

重载运算符 可以让 代码更简洁易懂，也使得用户定义的类型 和 内建类型 拥有 相似的行为。

重载运算符 对于某些运算来说 是符合 语言习惯的 (== < = <<)，遵循这些 语言约定 可以让 用户定义的类型 更易读， 也能更好地和 需要这些 重载运算符的 函数库 进行交互。

对于创建用户定义的类型的 对象来说，用户定义 字面量是 一种非常简洁的标记。

缺点

要提供 正确，一致，无异常行为 的 操作符 需要 花费不少精力，而且 如果 达不到这些要求的话，会出现 令人困惑的bug。

过度使用运算符 会带来 难以理解的 代码，尤其是 在重载的 操作符的 语义 和 通常的约定不符合时。

运算符重载 有着 函数重载的 所有弊端。

运算符重载 会混淆视听，让你 误以为 一些耗时的 操作 和 操作内建类型一样轻巧。

对 重载运算符 的调用点 的查找 需要的 可就不仅仅想 grep 那样的 程序了， 这时需要能够理解 C++ 语法的搜索 工具

如果重载运算符的 参数写错，此时得到的可能是一个 完全不同的 重载 而非 编译错误。

重载某些运算符 本身就是有害的。例如，重载 一元运算符 & 会导致 相同的代码 有完全不同的含义，这 取决于 重载的声明 对某段代码 而言是否 可见的。 重载 诸如 &&, || 和 , 会导致 运算符 顺序 和 内建运算的 顺序不一致。

。。。？ 逗号也能重载？ 难道 ; 也可以重载？？？？

运算符重载 通常 定义在 类的外部，所以 对于 同一个运算， 可能出现 不同的文件 引入 不同的 定义 。 如果2种 定义 都链接到 同一 二进制文件，则会出现 未定义行为，导致难以发现的 运行时错误。

用户定义 字面量 所创建的 语义形式 对于 某些有经验的 C++ 程序员来说 都是 很陌生的。

结论

只有 在 意义明显，不会出现 奇怪的行为 并且 与对应的 内建 运算符的 行为 一致时 才 定义 重载运算符。

只有对 用户自己定义的 类型 重载运算符。更准确地说，将它们 和 它们所操作的 类型 定义在 同一个 头文件中，.cc 中， 和 命名空间中。 这样做 无论类型 在哪里 都能使用 定义的 运算符，并且 最大程度上 避免了 多重定义的 风险。如果可能的话，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。

如果你定义了 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 保证语义一致。 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， 并保证，对于 同一组参数， < 和 > 不会 同时返回 true。

建议不要将 不进行修改的 二元运算符 定义为 成员函数。 如果一个 二元运算符 被定义为 类成员，这时隐式转换 会 作用于 右侧参数 但不会作用于 左侧。这时 会出现 a < b 能够编译，但是 b < a 不能编译的情况。

不要为了避免 重载操作符 而走极端。 比如说，应当定义 ==，=，<<， 而不是 Equals(),CopyFrom(),PrintTo()。 反过来说，不要只是为了满足 函数库 需要 而去定义 运算符重载。 比如说，如果你的类型 没有自然顺序，而你要将 它们存入 std::set 中，最好 还是 定义 一个 自定义的 比较运算符 而不是 重载 < 。

不要重载 && || ，或一元运算符 & ，不要重载 operator""

3.9 存取控制

总述

将 所有 数据成员 声明为 private， 除非是 static const 类型成员 。 出于技术上的原因，在使用 Google Test 时 我们允许测试固件类 的数据成员 为 protected。

3.10 声明顺序

总述

将相似的声明放在一起，将 public 部分放在 最前面

说明

类定义 一般应以 public: 开始，后跟 protected: 最后是 private:

在各个部分中，建议 将类似的 声明放在一起，并且建议以如下的顺序： 类型(包括 typedef，using，嵌套的结构体与类)，常量，工厂函数，构造器，赋值运算符，析构器，其他函数，数据成员。

不要将 大段的 函数定义 内联在 类定义中。通常，只有那些 普通的，或者 性能关键且 短小的 函数 可以内联在 类定义中。

译者(YuleFox)笔记

不要在 构造器中 做太多 逻辑相关的 初始化

编译器提供的默认构造器 不会对 变量进行初始化，如果定义了其他构造器，编译器不再提供默认构造器

为了避免隐式转换，需要将 单参数构造器 声明为 explicit

为了避免 拷贝构造函数，赋值操作的 滥用 和 编译器 自动生成，可以将其声明为 private 且 无需实现

仅在 作为 数据集合时 使用 struct

组合 > 实现继承 > 接口继承 > 私有继承， 子类重载的 虚函数 也要声明 virtual，虽然编译器无强制要求。

避免使用多重继承，使用时，除了 一个基类 含有实现外，其他基类 均为 纯接口。

接口类 类名以 Interface 结尾，除了 提供 带实现的 虚析构函数，静态成员函数外，其他 均为 纯虚函数， 不定义 非静态 数据成员，不提供 构造器，提供的话，声明为 protected。

为降低复杂性，尽量不重载 操作符，模板，标准类中 使用时 提供文档说明

存取函数 一般 内联在 头文件中

声明次序 public -> protected -> private

函数体 尽量 短小，紧凑，功能单一

4. 函数

4.1 参数顺序

总述

函数的 参数顺序为： 输入参数在先，后跟输出参数

说明

C/C++ 中函数参数 或者是 函数的输入，或者是 函数的输出，或 兼而有之。
输入参数 通常是 值参 或 const 引用， 输出参数 或 输入/输出 参数 一般是 非const 指针。
在排列参数时，将所有的 输入参数 置于 输出参数 之前。
特别是，在加入 新参数时，不要以为它们 是新参数 就置于参数列表最后，而是仍然要按照 上述的规范。

4.2 编写简短函数

总述

我们倾向于 编写简短，凝练的函数

说明

我们承认 长函数 有时是合理的，所以 并不会 硬性限制 函数的长度。如果函数 超过 40行，可以 考虑 能否在 不影响程序结构的前提下 对其进行 分割。

即使一个 长函数 现在工作的很好， 一旦有人进行修改，有可能出现 新的问题，甚至导致 难以发现的 bug， 使函数尽量简短，以便于 他人阅读 和 修改代码。

在处理代码时， 你可能会发现 复杂的 长函数。 不要害怕 修改现有代码： 如果 证实这些 代码 使用/调试 起来很困难，或者 你只需要 使用 其中的 一小段代码，考虑将其分割 为 更加简短 并 易于管理的 若干函数

4.3 引用参数

总述

所有 按引用传递的 参数 必须加上 const。

定义

在C中，如果函数 需要修改变量的值，参数 必须是 指针，如 int foo(int *pVal) 在C++中，函数还可以声明为 引用参数 int foo(int &val)

优点

定义引用参数 可以防止 出现 (*pVal)++ 这样 丑陋的代码。
引用参数 对于 拷贝构造函数 和类似的 也是必须的。
同时也更明确地不接受空指针

缺点

容易引起无解，因为 引用 在 语法上 是值变量 但是 有 指针的 语义。

结论

函数参数列表中，所有的 引用参数 都必须是 const：

```
void Foo(const string &in, string *out);
```

事实上，这在 Google Code 是一个 硬性约定： 输入参数 是 值参 或 const 引用， 输出参数 是指针。 输入参数可以是 const 指针，但绝不能是 非const的 引用参数，除非特殊要求，比如 swap()。

有时，在输入形参中 用 const T* 比 const T& 更明智。比如：

可能会传递 空指针
函数要把 指针 或 对地址的 引用 赋值给 输入形参

总而言之，大多时候，输入形参 往往是 const T&，如果使用 const T* 则说明输入另有处理。 所以如果要 使用 const T*，则要给出相应的理由，否则 读者会困惑。

4.4 函数重载

总述

如果要使用 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 构造器

定义

你可以编写 一个 参数类型 为 const string& 的函数，然后 用另一个参数 类型 为 const char* 的函数 对其进行重载。

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

```
};
```

优点

通过重载 参数不同的 同名函数，可以令代码更加直观。模板化代码 需要重载，这同时也能为 使用者 带来便利。

缺点

如果函数 单 靠 不同的参数类型 而重载 (注：参数数量不变)， 读者就得 十分熟悉 C++ 五花八门的 匹配规则，以了解 匹配过程 具体到底如何。 另外，如果派生类 只重载了 某个函数的 部分变体，继承语义 就容易 令人困惑。

结论

如果打算重载 一个函数，可以 试试 在 函数名中 增加 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 可以用初始化列表<braced-initializer-list> 来指定参数。

4.5 缺省参数

总述

只允许在 非虚函数 中 使用缺省参数，并且 必须保证 缺省参数的值 始终一致。
缺省参数 和 函数重载 遵循 相同的 规则。
一般情况下 建议使用 函数重载，尤其是 在 缺省函数 带来的 可读性 提升 不能弥补下文中 所提到的 缺点的 情况下。

优点

有些函数 一般情况下 使用 默认参数，但有时需要 又使用 非默认的参数。 缺省参数为这样的 情况 提供了便利，使程序员 不需要为了 极少的特例 编写大量的函数。
。。但是 直接一个 最通用的 就可以了，不需要 编写 函数吧， 只是调用的时候，
和函数重载相比，缺省参数的 语法更简洁明了，减少了 大量的 样板代码，也更好地区别了 必要参数 和 可选参数

缺点

缺省参数 实际上 是 函数重载语义 的另一种实现方式，因此 所有 不应使用 函数重载的 理由 也适用于 缺省参数。

虚函数调用的 缺省参数 取决于 目标对象的 静态类型，此时 无法保证 给定函数的 所有 重载声明的 都是 同样的 缺省参数。 。。 应该是指 缺省值 ？

缺省参数是在 每个调用点 都要进行 重新求值的，这会造成 生成的 代码迅速膨胀。作为读者，一般来说，也更希望 缺省的 参数 在声明时 就已经被固定了，而不是在 每次调用时 都可能会 有不同的取值。

缺省参数 会 干扰 函数指针，导致 函数前面 和 调用点的 签名 不一致。而 函数重载不会导致这样的问题。

结论

对于虚函数，不允许 使用 缺省参数，因为 在虚函数中 缺省参数 不一定 能正常工作。

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，不要写： void f(int n = counter++); 这样的代码)

其他情况下，如果 缺省参数 对 可读性的提升 远远 超过了 以上提及的 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。

4.6 函数返回类型后置语法

总述

只有在 常规写法（返回类型前置）不便于 书写 或 不便于阅读的时候 使用 类型后置语法。

定义

C++ 现在允许 2种 不同的 函数声明方式

以往的写法 是 将 返回类型 置于 函数名之前：

```
int foo(int x);
```

C++ 11引入了新的 形式，可以在 函数名 前面使用 auto，在参数列表后 后置 返回类型

```
auto foo(int x) -> int;
```

后置返回类型 为 函数作用域。 对于 int 这样的简单类型，2种写法 没有区别。 但是对于 复杂情况，例如 类域中的 类型声明 或 以函数参数 的形式 书写的 类型，写法的不同会 造成 区别。

优点

后置返回类型 是 显式地 指定 lambda 表达式 的 返回值的 唯一方式。 某些情况下，编译器可以 推导出 lambda 的返回类型，但有时不行。 即使 编译器 可以自动推导，显式指定返回类型 也让 读者 更明了。

有时 后置返回类型 能让 书写更简单，更易读，尤其是在 返回类型 依赖于 模板参数时：

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

对比下面的例子

```
template <class T, class U> decltype(declval<T&>() + declval<U&>()) add(T t, U u);
```

。。。？没见过。。

。。

```
int& foo(int& i);
float foo(float& f);
template <class T> auto transparent_forwarder(T& t) -> decltype(foo(t))
{ return foo(t);}
```

。。

缺点

后置返回类型 相对来说 是 非常新的语法，而且 在 C 和Java 中 都没有相似的 写法，因此对于读者可能比较 陌生。

在已有的 代码中 有 大量的 函数声明，你不可能 把 它们 都用 新的 语法 重写一遍。因此 实际的 做法 只能是 使用 旧语法 或 新旧混用。 在这种情况下， 只使用 一种 版本 是相对来说 更规整的 形式。

结论

在大部分情况下，应当继续使用 以往的 函数声明写法， 即 返回类型前置。
只有在 必要的时候 (如 Lambda) 或者 使用 后置语法 能够 简化 书写 并 提到 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 很少见，而且 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 复杂的 模板代码。

5 来自google 的奇技

Google 用了 很多自己 实现的 技巧/工具 使得 C++ 代码 更加健壮，我们使用 C++ 的方式可能和 你在 其他地方见到的 有所不同。

5.1 所有权与智能指针

总述

动态分配出的 对象 最好 有单一且 固定的 所有主，并通过 智能指针传递所有权。

定义

所有权 是一种 登记/管理 动态内存 和 其他资源的 技术。
动态分配对象 的 所有主 是一个 对象或函数，后者负责 确保 当 前者 无用时 就自动销毁前者。
所有权 有时可以共享，此时就由最后一个 所有主 来负责销毁它。甚至也可以不用共享，在代码中 直接把 所有权 传递给其他对象。

智能指针 是一个 通过 重载 * 和 -> 运算符 以表现得 如指针一样的类。智能指针类型被用来 自动化 所有权的 登记工作，来确保 执行销毁义务 到位。

std::unique_ptr

是C++11 新推出的 一种 智能指针类型，用来标识 动态分配出 的对象 的独一无二的 所有权，当 std::unqiue_ptr 离开作用域时，对象就会 被销毁。std::unique_ptr 不能被复制，但是 可以把它 move 给 新的所有主。

std::shared_ptr

同样表示动态分配 对象的 所有权，但是 可以被 共享，也可以被复制；对象的所有权由 所有复制者 共同拥有，最后一个 复制者 被销毁时，对象也会随着 被销毁。

优点

如果没有清晰，逻辑条理 的 所有权安排，不可能管理好 动态分配的 内存。

传递对象 的所有权，开销比 复制来的小，如果可以复制的话。

传递所有权 也比 "借用" 指针 或引用 来得 简单，毕竟它 大大省去了 2个 用户 一起协调对象生命周期的 工作。

如果所有权 逻辑条理，有文档 且不紊乱的话， 可读性有 很大提升。

可以不用 手动 完成 所有权的登记工作，大大简化了 代码，也免去了 一大波错误之烦恼。

对于 const 对象来说， 智能指针 简单易用，也比 深度复制高效。

缺点

不得不用 指针 (不管是 智能的 还是 原生的) 来表示 和 传递 所有权。 指针语义 可要比 值语义 复杂的多， 特别是在 API中：这里不光要 操心 所有权，还要 顾忌别名，生命周期，可变性 以及 其他大大小小的 问题。

其实 值语义的 开销 经常 被 高估， 所以 所有权 传递 带来的性能提升 不一定 能弥补可读性 和 复杂度的 损失。

如果 API 依赖 所有权的 传递， 就使得 客户端 不得不 用 单一的 内存管理模型。

如果 使用 智能指针，那么 资源释放 发生的 位置 就会 变得 不那么明显。

std::unique_ptr 的所有权传递 原理 是 C++11 的move 语法，容易迷惑程序员。

如果原本的 所有权 设计 已经足够完善了，那么 如果要引入 所有权 共享机制，可能不得不 重构整个系统。

所有权 共享机制 的 登记工作 在 运行时进行，开销可能很大。

某些极端情况下 (如循环引用)，所有权 被共享的 对象 永远不会被 销毁。

智能指针 无法 完全 代替 原生 指针

结论

如果必须使用 动态分配，那么 更倾向于 将 所有权 保持在 分配者 手中。

如果其他地方 要使用 这个对象，最好 传递 它的 拷贝，或者 传递一个 不用 改变 所有权的 指针 或 引用， 倾向于 使用 std::unique_ptr 来明确 所有权 传递：

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

如果没有很好的理由，则不要使用 共享 所有权。 这里的理由 可以 是 为了 避免 开销昂贵的 拷贝操作， 但是 只有 当 性能提升 非常明显，并且 操作的 对象 是不可变的 (比如说， std::shared_ptr<const Foo>) 时，才能这么做。 如果 确实要 使用 共享所有权，建议使用 std::shared_ptr。

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

译者笔记

3: scoped_ptr 和 auto_ptr 已经过时，现在是 shared_ptr 和 unique_ptr 的天下。

5.2 Cpplint

总述

使用 cpplint.py 检查风格错误。

6 其他C++特性

6.1 引用参数

所有 按引用传递的 参数 必须加上 const。

定义

在C中，如果函数需要 修改 参数的值，参数 必须是 指针， C++中，还可以声明为 引用参数

优点

定义引用参数 防止 出现 (*pval)++ 这样 丑陋的代码。

像拷贝构造函数 这样的 应用也是 必须的。而且 更明确，不接受 null 指针。

缺点

容易引起无解，因为 引用 在语法上 是值变量 却 拥有 指针的语义。

结论

函数参数列表中， 所有 引用参数 都必须是 const

```
void Foo(const string &in, string *out);
```

事实上，这在 google code 是一个 硬性约定：输入参数 是值 或 const 引用， 输出参数 为 指针。 输入参数 可以是 const 指针，但决不能 是 非 const 的引用参数，除非 用于交换，比如，swap()，有时，在输入形参中 用 const T* 指针 比 const T& 更明智。比如：

你会传 null 指针

函数 要把 指针 或 对地址的 引用 赋值给 输入形参。

总之 大多数时候 输入参数 往往是 const T&。 如果使用 const T* 说明 输入 另有处理。 所以 如果 你要使用 const T* (。。？不是 T&)， 则应该 有理有据，否则会使得 读者 误解。

6.2 右值引用

只有 定义 移动构造函数 和 移动赋值操作时 使用 右值引用，不要使用 std::forward。

定义

右值引用 是一种 只能绑定到 临时对象的 引用的一种，其语法 和 传统的 引用语法 相似，例如， void f(string&& s); 声明了 一个 参数是字符串的 右值引用 的 函数。

优点

用于定义移动构造函数 (使用类的 右值 引用 进行构造的函数) 使得 移动 一个值 而不是 拷贝 成为可能。 例如，如果 v1 是一个 vector<string>， 则 auto

v2(std::move(v1)) 将很可能 不再 进行 大量的 数据复制 而是 简单地 进行 指针操作。这种情况下，带来 大幅度的 性能提升。

右值引用 使得 编写 通用的 函数 封装来转发 其参数 到 另外一个 函数 称为可能，无论 其参数 是否是 临时对象 都能正常工作。 右值引用 能实现 可移动 但不可拷贝的类型，这一特性 对那些 在 拷贝方面 没有 实际需求，但有时又 需要 将它们 作为 函数 参数 传递 或 塞入 容器的 类型 很有用。

要高效地使用 某些标准类型，例如 std::unique_ptr， std::move 是必须的

缺点

右值引用 是一个 相对比较新的 特性 (由 C++11 引入)，它尚未被广泛理解。。。写的时候 是新的。。

类似 引用奔溃，移动构造函数 的 自动推导 这样的 规则 是 很复杂的。

结论

只在 定义 移动构造函数 和 移动赋值操作时 使用 右值引用，不要 使用 std::forward 功能函数，你可能会 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。

6.3 函数重载

如果要用好 函数重载，最好能让 读者 一看 调用点 (call site) 就 胸有成竹，不用花心思 猜测 调用的 重载 函数 到底 是哪一种。 该规则 适用于 构造器。

定义

你可以编写一个 参数类型 为 const string& 的函数， 然后 用另一个 const char* 的函数 重载它

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

优点

通过重载 参数不同 的 同名函数，令代码更加直观，模板化代码 需要重载，同时 为 使用者 带来 便利

缺点

如果函数 单单 靠 不同的 参数类型 而重载 (这意味着 参数数量不变)，读者就 得 十分熟悉 C++ 五花八门 的匹配规则，以了解 匹配 过程 具体 到底如何。另外，当 派生类 只重载了 某个函数的 部分变体， 继承语义 容易令人 困惑。

结论

如果你打算 重载一个 函数，可以试试 在 函数名上 添加 参数 信息。 例如 使用 AppendString(),AppendInt() 而不是 一口气 重载多个 Append()

6.4 缺省参数

我们不允许 使用缺省参数， 少数极端情况下，尽可能 改用 函数重载。

。。这个标题 和 4.5 重复了。内容 也部分重复。

优点

当你有 依赖 缺省参数 的函数时， 你也许 偶尔 会修改 这些 缺省参数。 通过 缺省参数，不用 再 为 个别情况 而 特意定义 一大堆函数了。 与 函数重载相比，缺省参数 语法更为清晰，代码少，也很好地 区分了 必选参数 和 可选参数

缺点

缺省参数 会 干扰 函数指针，害得 后者的 函数签名（function signature）往往 对不上 所 实际要调用的 函数签名。 即在 一个 现有函数中 添加 缺省参数，就会改变它的类型，那么 调用 其地址的 代码可能会出错，不过 函数重载 就没有这个问题。此外， 缺省参数 会造成 臃肿的 代码，毕竟 它们在 每个 调用点（call site）都 重复 。函数重载正好相反，毕竟 它们所谓的 缺省参数 只会出现在 函数定义里。

结论

由于 缺点不是 很严重， 有些人 依然 偏爱 缺省函数。 所以 除了以下情况，我们要求 必须 显式提供 所有参数：

1. 位于 .cc 文件中的 静态函数 或 匿名空间函数， 毕竟 都只能 在 局部文件中 调用 该函数了。
2. 可以在 构造器中 用 缺省参数，毕竟 不可能取得 它们的 地址。
3. 可以用来模拟 变长数组。

6.5 变长数组 和 alloca()

我们不允许 使用 变长数组 和 alloc()

优点

变长数组 具有 浑然天成 的语法。 变成数组 和 alloc() 也都很高效

缺点

变长数组 和 alloca() 不是 标准C++ 的组成。

更重要的是，它们根据数据大小 动态分配 堆栈内存，会引起 难以发现的 内存越界 bug： 在我的机器上运行的好好的，发布后却莫名其妙的挂掉了

结论

改用 更安全的 分配器 (allocator)，就像 std::vector 或 std::unique_ptr<T[]>。

6.6 友元

我们允许合理的使用 友元类 及 友元函数

通常友元应该定义在 同一文件内，避免 读者 跑到 其他文件 查找 该 私有成员的类。

经常用到 友元的 一个地方是 将 FooBuilder 声明为 Foo 的友元，以便 FooBuilder 正确构造 Foo 的内部状态， 而无需将 该 状态 暴露出来。

某些情况下，将 一个 单元测试类 声明为 待测试类 的 友元 会很方便。

友元扩大了(但没有打破) 类的 封装边界。 某些情况下，相对于 将类 成员 声明为 public，使用 友元 是更好的 选择， 尤其是 如果 你只 允许 另一个类 访问 该类的 私有成员时。 当然，大多数类 都只应该 通过 其提供的 公有成员 进行 相互操作。

6.7 异常

我们不使用 C++ 异常

优点

异常允许 应用高层 决定 如何 处理 在 底层 嵌套函数中 [不可能发生] 的失败，不用管 那些 含糊 且 容易出错的 错误代码

很多现在语言 都用 异常。 引入异常 更通用。

有些 第三方 C++ 库 依赖异常， 禁用异常 就不好用了。

异常是 处理 构造器 失败的 唯一 途径。 虽然可以 用 工厂函数 或 Init() 方法 代替异常，但 前者要求 在 堆栈分配内存， 后者 会导致 刚创建的 实例 处于 "无效" 状态

在 测试框架中 很好用。

缺点

在现有函数中 添加 throw 语句时， 你必须检查所有 调用点。要么让 所有调用点 统统具备 最低限度的 异常安全保证， 要么眼睁睁地看着 异常 一路 往上跑，最终 中断整个程序。

还有更常见的， 异常 会彻底扰乱 程序的 执行流程 并难以判断，函数 也许会 在 你意想不到的 地方返回。你或许 会加一大堆 何时何处 处理异常的 规定 来 降低风险，然而 开发者的 记忆负担 更重了。

异常安全 需要 RAII 和 不同的 编码实践。 要轻松编写出 正确的 异常安全 代码 需要大量的 支持机制。更进一步地说，为了避免读者 理解 整个调用表，异常安全 必须 隔绝从 持续状态 写到 "提交" 状态的逻辑。这一点 有利有弊 (因为你也许 不得不 为了隔离提交 而 混淆代码)。 如果允许使用异常，我们就不得不 时刻关注 这样的 弊端。即使有时 它们 并不值得。

应用异常 会增加 二进制文件数据，延长 编译时间 (或许影响小)， 还可能加大 地址空间的 压力。

滥用异常 会变相 鼓励 开发者 去 捕获 不合时宜，或 本来 已经 无法恢复的 [伪异常]。比如，用户的输入 不符合 格式要求的，也用不着 抛异常。如此之类的 伪异常 列都列不完。

结论

从表面上看来，使用 异常利大于弊，尤其是在 新项目中，但是 对于 现有代码，引入异

常 会牵连到 所有 相关代码。
如果 新项目 允许 异常 向外 扩散，在跟 以前 未使用 异常的 代码 整合时 也将是个麻烦。
因为 google 现有的 大多数 C++ 代码 都没有 异常处理。 引入 带有 异常处理的 新代码 相当困难。 鉴于 Google 现有代码 不接受异常，在 现有代码 中使用 异常 比 在新项目中使用的 代价 要大一些。 迁移过程比较慢，也容易出错。

我们不相信异常的使用 有效替代方案，如错误代码，断言 等 会造成 严重负担。

我们并不是基于哲学 或 道德层面 反对 使用 异常，而是在 实践 的基础上。
我们希望 google 使用 我们自己的 开源项目， 但 项目中 使用 异常 会为此 带来不便，因此 我们也建议 不要在 google 的 开源项目中 使用异常。

对于window 有个特例 。。 没有把代码贴上来。。

。。。
https://zhuanlan.zhihu.com/p/315789294
google禁用是因为 历史包袱。代码库 中有很多 旧风格的 C++代码，它们对 异常 不友好，根本没有考虑 异常，做不到 异常安全。

LLVM禁用，是因为：异常让最终的二进制文件大小增加了不少——异常产生的位置决定了需要如何做栈展开（stack unwinding），这些数据需要存储在表里——这就是异常导致二进制较大的主要原因。

也有一些支持C++异常的，如 Bjarne Stroustrup（C++ 之父） 和 Herb Sutter
。。。

6.8. 运行时类型识别
我们禁用 RTTI

定义
RTTI 允许程序员 在运行时 识别 C++类对象的 类型，它通过 使用 typeid 或 dynamic_cast 完成。

优点
RTTI 的标准替代（下面将描述）需要 对 有问题的 类层级 进行 修改 或重构。
有时 这样的修改 并不是 我们想要的，甚至是不可取的，尤其是在 一个 已经 广泛使用的 或 成熟的 代码中。
RTTI 在 某些单元测试中 非常有用。比如 进行工程类测试时，用来验证 一个 新建对象是否为 期望的 动态类型。
RTTI 对于管理 对象 和 派生对象的 关系 也很有用。
在考虑多个抽象对象时 RTTI 也很好用，例如

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast<Derived*>(other);
  if (that == NULL)
    return false;
  ...
```

```
}
```

缺点

在运行时判断 类型 通常意味着 设计问题。 如果你需要 在运行期间 确定 一个对象的类型，这通常说明 你需要 考虑 重新设计 类。
随意地使用 RTTI 会导致 你的代码难以维护，它使得 基于类型的 判断树 或者 switch 语句 散落在 代码各处，以后 需要进行修改，你必须 检查它们。

结论

RTTI 有合理的 用途 但是 容易被 滥用，因此在 使用时 务必注意。
在单元测试中，可以使用 RTTI，但是 其他代码中 请尽量避免
尤其在 新代码中，使用 RTTI 前 务必三思。
如果你的代码 需要根据不同的 对象类型 执行 不同的 行为的话，请考虑下面的 2种方案之一：

1. 虚函数 可以根据 子类类型 的不同 而执行不同的代码，这是把工作 交给了 对象本身 去处理。
2. 如果 这一工作 需要在 对象之外 完成，可以考虑 使用 双重分发的方案。例如 访问者 设计模式，这就能够在 对象之外 进行 类型判断。

如果程序能够保证 给定的基类 实例 实际上 都是 某个 派生类的 实例，那么 就可以自由滴 使用 dynamic_cast。

基于类型的判断数 是一个 很强的暗示， 说明你的代码 已经偏离正轨，不要像下面这样：

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

一旦在类层级中 加入新的 子类，像这样的 代码 往往会崩溃。而且 一旦某个子类的 属性改变了，你很难找到 并修改 所有 受影响的 代码块。

不要手工实现 一个 类似 RTTI 的方案。 反对 RTTI 的理由 同样适用于 这些方案，比如带类型标签 的 类继承 体系。 而且，这些方案 会掩盖 你的 真实意图。

6.9 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

定义

C++采用了 有别于 C 的类型转换机制，对转换操作 进行归类。

优点

C语言的类型转换 问题在于 模棱两可的操作，有时是在 做强制转换 (如 (int)3.5)，有时是在做 类型转换 (如 (int)"hi")， 另外，C++的类型转换 在查找时 更醒目。

缺点

语法

结论

不要使用 C 风格类型转换，而应该使用 C++ 风格

使用 static_cast 替代 C 风格的 值转换，或 某个类指针 需要 明确地向上转换为 父类 指针时。

用 const_cast 去掉 const 限定符

用 reinterpret_cast 指针类型 和 整型 或 其它指针 之间 进行不安全的 相互转换，仅在 你 对所做的 一切 了然于心时 使用。

至于 dynamic_cast， 参见 RTTI。

6.10 流

只在 记录日志时 使用流

定义

流 用来替代 printf() 和 scanf()

。。。 cin cout 都不能用了？？？

优点

有了流，在打印时 不需要关系 对象的类型。不用担心 格式化字符串 与 参数列表 不匹配 (虽然在 gcc 中使用 printf 也不存在这个问题)。

流的 构造 和析构 会自动打开 和 关闭 对应的文件。

缺点

流 使得 pread() 等功能函数 很难 执行。如果 不使用 printf 风格的 格式化字符串，某些 格式化 操作 (尤其是常用的 格式化字符串 %.*s) 用 流处理 性能是很低的，流 不支持 字符串 操作符 重排序 (%1s), 而这一点 对于 软件 国际化 很有用。

结论

不要使用 流，除非是 日志接口需要， 使用 printf 之类的 代替。

使用流 还有很多 利弊， 但 代码 一致性 胜过一切。 不要在 代码中 使用流。

拓展讨论

深层次原因，回想一下 唯一性原则： 我们希望 在任何 时候 都只使用 一种 确定的 IO 类型，使代码 在所有 IO 处 都保持一致， 因此， 我们不希望 用户来决定 是 使用流 还是 printf + read/write。 相反，我们应该决定 到底使用哪一种方式。

把日志作为特例 是因为日志是一个 非常独特的 应用，还有 一些是 历史原因。

流的支持者 主张 流是不二之选，但观点 不是 那么 清晰有力。他们指出的 流的每个 优势 也都是其劣势。

流最大的优势 是 在输出时 ，不需要关系 打印对象的 类型，这是一个亮点。同时 也是一个 不足： 你很容易 用错 类型，而编译器不会 报警。使用流 很容易造成 的错误：

```
cout << this;   // 输出地址
cout << *this;  // 输出值
```

由于 << 被重载，编译器不会报错，就因为 这一点 我们反对 使用 操作符重载。

有人说 printf 的格式化丑陋，可读性差，但流也好不到哪里去：

```
cerr << "Error connecting to '" << foo->bar()->hostname.first
     << ":" << foo->bar()->hostname.second << ": " << strerror(errno);

fprintf(stderr, "Error connecting to '%s:%u: %s",
        foo->bar()->hostname.first, foo->bar()->hostname.second,
        strerror(errno));
```

你可能会说，把流封装一下就会比较好了，这里是可以，但是其他地方呢？ 而且不要忘了，我们的目标是 使语言更紧凑， 而不是 增加 一些别人 需要学习的 地方。

每种方式 都各有利弊，"没有最好，只有更适合"。 简单性原则 告诫我们 必须选择 其一，最后 大多数决定 使用 printf + read/write

6.11 前置自增和自减

对于迭代器 或 其他模板对象 使用 前缀 自增 自减

定义

对于变量 在 自增 或自减 后 表达式的值 没有被用到的 情况下，需要确定 使用 前置 还是 后置 的 自增 自减

优点

不考虑 返回值的话，前置自增 通常 要比 后置自增 效率更高。 因为后置 自增 需要对 表达式的 值 进行 一次拷贝。 如果 是 迭代器 或其他 非数值类型， 拷贝的 代价 是比较大的。

缺点

在C开发中，当表达式的 值 未被使用时， 传统的 做法是 后置自增， 特比是在 for 循环中。

有些人 觉得 后置 更易懂， 因为 这很像 自然语言， 主语i在谓语动词++ 前面。

结论

对于 简单数值，两种 都无所谓， 对于 迭代器 和 模板类型， 使用前置 自增自减。

。。。我怎么记得 map的 ++it 不行？

6.12 const 用法

我们强烈 建议 你在 任何 可能的 情况下 都要使用 const。 此外有时 改用 C++ 11 推出的 constexpr 更好。

定义

在声明的 变量 或参数 前面 加上 关键字 const 用于 指明 变量值 不可被篡改 (如 const int foo)。 为 类中的 函数 加上 const 限定符 表明 该函数 不会 修改 类成员 变量的 状态 (如 class Foo { int Bar(char c) const; };)

优点

大家更容易理解 如何使用 变量。

编译器 可以更好地 进行 类型检测，相应地，也能生成更好的代码。

人们对编写 正确的 代码 更加自信，因为 他们知道 所调用的 函数 被限定了 能或 不能 修改 变量值。 即使再 无锁的 多线程编程中， 人们也知道 什么样的 函数 是安全的。

缺点

const 是 侵入性的：如果你想 一个 函数 传入 const 变量，函数原型 声明 中 也必须对应 const 参数 (否则变量 需要 const_cast 类型转换)， 在调用库函数时 尤其麻烦。

结论

const 变量，数据成员，函数，参数 为 编译时类型 检测 增加了 一层保障；便于尽早发现错误。因此 我们强烈建议 在任何可能的情况下 使用 const：

如果函数不会 修改你传入的 引用或 指针类型参数，该参数应该声明为const

尽可能将函数 声明为const， 访问函数应该总是 const。 其他 不会修改 任何数据成员，没有调用 非const函数， 不会返回数据成员 非const指针 或 引用 的函数也应该声明为 const

如果数据成员 在 对象构造之后 不再发生变化， 可以定义为 const。

然而，也不要发疯似地 使用 const。 想 const int * const * const x; 就有点过了，虽然 它非常精确地描述了 常量 x。 关注真正有 帮助意义的信息：前面的例子写成 const int** x 就够了。 关键字 mutable 可以使用，但是 在 多线程中 是不安全的，使用时 首先考虑 线程安全。

。。mutable，在声明时 使用的 修饰符， 可以在 const 对象 和/或 const 方法 中 修改 这个 值。

const 的位置

有些人喜欢 int const *foo, 不喜欢 const int* foo， 他们认为 前者 更一致 因此 可读性更好： 遵循了 const 总位于 其描述的 对象 之后的 原则， 但是 一致性原则 不适用于此，"不要过度使用" 的声明 可以 取消 大部分 你原本 想保持的 一致性。 将 const 放在前面 才更 易读。 因为 自然语言中 形容词 const 是在 名词int 之前。 我们提倡 但不强制 const 前置， 但是要保证 代码的一致性 (就是 要么全后置，要么全前置)

6.13 constexpr 用法

C++ 11 中 使用 constexpr 来定义 真正的 常量，或实现 常量 初始化。

定义

变量可以被声明为 constexpr 以表示 它是真正意义上的 常量，即在 编译时 和 运行时都不变。 函数 或构造函数 也可以被声明为 constexpr，以用来 定义 constexpr 变量。

优点

如今 constexpr 就可以 定义浮点式 的真常量，不用在 依赖字面值
也可以定义 用户自定义类型上的常量
甚至也可以定义 函数调用所返回的 常量

缺点

如果过早把 变量优化成 constexpr 变量，将来又要把它改为常规变量时，挺麻烦的，当前对 constexpr 函数 和 构造器 中 允许的 限制 可能会 导致 这些 定义中 解决的方法模糊。

结论

依靠 constexpr 特性，才实现了 C++ 在接口上 打造真正 常量机制的 可能。
好好用 constexpr 来定义 真 常量 以及 支持 常量的 函数。 避免复杂的 函数定义，以使其能够 与 constexpr 一起使用。
千万别 痴心妄想地 想靠 constexpr 来 强制 代码 内联。

6.14 整型

C++ 内建模型中，仅使用 int。如果程序中 需要 不同大小的 变量，可以使用 <stdint.h> 中 长度 精确的 整型，如 int16_t 或 int64_t 。

定义

C++没有指定整型的 大小，通常 人们假定 short 是 16， int 是 32， long 是32，long long 是 64.

优点

保持声明统一

缺点

C++中 整型大小 因编译器 和 体系结构不同 而不同。

结论

<stdint.h> 中定义了 int16_t, uint32_t, int64_t 等整型， 在需要 确保 整型大小时可以使用 它们来代替 short, unsigned long long 等。
在 C 整型中，只使用 int。在合适的情况下，推荐使用 标准类型如 size_t，ptrdiff_t。

对于大整数，使用 int64_t。 不要使用 uint32_t 等无符号整型，除非你是在 表示 一个 位组 而不是 一个 数值，或者 你需要 定义 二进制补码溢出。
尤其不要 为了 指出 数值永不会为 负，而使用 无符号整型。相反，你应该 使用 断言来 保护数据。
如果你的 代码 涉及到容器返回的 大小(size)， 确保其类型 足以应付 容器 各种可能的 用法。拿不准时，类型越大越好。 小心 整型类型 转换 和 整型提升 (int 和 unsigned int 一起时，int 被提升为 unsigned int 而有可能溢出)。

关于无符号整数

有些人，包括一些 教科书作者， 推荐使用 无符号类型 表示 非负数，这种做法 试图达到 自我文档化。 但是 C中，这一 有点 被 其导致的 bug 淹没：

```
for (unsigned int i = foo.Length()-1; i >= 0; --i) ...
```

上面的循环 永远不会退出，有时 gcc 会发现该bug 并报警，但大部分情况下都不会。类似的 bug 还会出现在 比较 有符号变量 和 无符号变量时， 主要是 C的 类型提示机制会导致 无符号类型的 行为 出乎你的意料。
因此，使用 断言 来指出 变量 为 非负数，而不是 使用 无符号型

6.15 64位下的可移植性

代码应该对 64位 和 32位 系统友好。处理打印，比较，结构体对齐 时 切记：

1. 对于某些类型，printf()的指示符 在32位和 64位 系统上 可移植性不是很好。C99 标准定义了 一些 可移植的 格式化指示符，但是 MSVC 7.1 并非全部支持，而 标准中也有所遗漏，所以 有时 我们不得不自定义一个 丑陋的 版本 (头文件 inttype.h 仿 标准风格)

```
// printf macros for size_t, in the style of inttypes.h
#ifdef _LP64
#define __PRIS_PREFIX "z"
#else
#define __PRIS_PREFIX
#endif

// Use these macros after a % in a printf format string
// to get correct 32/64 bit behavior, like this:
// size_t size = records.size();
// printf("%"PRIuS"\n", size);
#define PRIdS __PRIS_PREFIX "d"
#define PRIxS __PRIS_PREFIX "x"
#define PRIuS __PRIS_PREFIX "u"
#define PRIXS __PRIS_PREFIX "X"
#define PRIoS __PRIS_PREFIX "o"
```

类型不要使用 备注
void * %lx %p(或其他指针类型)
int64_t %qd, %lld %"PRId64"
uint64_t %qu, %llu, %llx %"PRIu64", %"PRIx64"
size_t %u %"PRIuS", %"PRIxS" C99 规定 %zu
ptrdiff_t %d %"PRIdS" C99 规定 %zd

注意 PRI* 宏会被编译器扩展为独立字符串.
因此如果使用非常量的格式化字符串,
需要将宏的值而不是宏名插入格式中. 使用 PRI* 宏同样可以在 %
后包含长度指示符. 例如, printf("x = %30"PRIuS"\n", x) 在 32 位
Linux 上将被展开为 printf("x = %30" "u" "\n", x), 编译器当成
printf("x = %30u\n", x) 处理

2. 记住 sizeof(void *) != sizeof(int)。 如果需要一个 指针大小的 整数 要用 intptr_t。

3. 你要非常小心 对待 结构器对齐，尤其是要持久化到 磁盘上的 结构体 (将数据按字节流顺序 保存到 磁盘 或数据库中)。 在64位体统中，任何含有 int64_t/uint64_t 成员的 类/结构体，确实都是以 8字节 在结尾对齐。

如果 32位 和 64位 代码 要共用 持久化的 结构体，需要确保 2种体系结构下 的结构体 对齐 一致。 大多数编译器 都允许 调整 结构体对齐， gcc 中使用 __attribute__((packed)). MSVC 则提供了 #pragma pack() 和 __declspec(align())

4. 创建64位常量时 使用 LL 或 ULL 后缀：
   int64_t my_value = 0x123456789LL;
   uint64_t my_mask = 3ULL << 48;
5. 如果你确实要 32位和 64位系统 具有 不同的代码， 可以使用 #ifdef_LP64 来区分 32/64 位代码 (尽量不要这么做，如果非用不可，尽量使 修改局部化)

6.16 预处理宏

使用 宏 时 要非常 谨慎，尽量以 内联函数，枚举，常量 代替它。

宏意味着 你和 编译器 看到的 代码是不同的， 这可能导致 异常行为，尤其是因为 宏具有全局作用域。

值得庆幸的是，C++中，宏不像 在 C 中那么 必不可少。
以往 通过 宏展开 性能关键的 代码，现在 可以用 内联函数 替代。 用宏表示 常量 可以被 const 变量 代替。 用 宏 缩短 长变量名 可以 被 ref 替代。 用宏 进行 条件编译 这个 千万别这么做，会令 测试更加痛苦 (#define 防止头文件重复包含 是特例)

宏可以做到 一些 其他技术 无法实现的事情，在一些代码库中(尤其是 底层库中) 可以看到宏的 某些特性 (如 用 # 字符串化， 用 ## 连接 等等)。 但在 使用前，仔细考虑下 能不能不使用宏 达到相同的 目的。

下面的 用法模式 可以避免 使用宏带来的 问题， 如果你要 使用宏，尽可能遵守
不要在 .h 文件中 定义宏
在马上要使用时 才进行 #define， 使用后 立即 #undef
不要只是对 已经存在的 宏 使用 #undef， 选择一个 不会冲突的 名称
不要试图 使用 展开后 会导致 C++ 构造不稳定的 宏， 不然也至少 要附上 文档说明行为
不要使用 ## 处理 函数，类，变量的 名字。

6.17 nullptr 和 NULL 和 0

整数用 0， 实数用 0.0， 指针用 nullptr 或 NULL， 字符(串)用 '\0'。
整数用0， 实数用0.0 是毫无争议的。

对于 指针(地址值),到底是 0，NULL 还是 nullptr。 C++11 项目用 nullptr， C++03 项目 用 NULL，毕竟它看起来 像指针。
实际上，一些 C++ 编译器 对 NULL 的定义 比较特殊，可以 输出 有用的 警告，特别是 sizeof(NULL) 就和 sizeof(0) 不一样。

字符(串) 使用 '\0'， 不仅类型正确 且 可读性好

6.18 sizeof

尽可能使用 sizeof(varname) 代替 sizeof(type)。

使用 sizeof(varname) 是因为 当 代码中 变量类型 改变时 会自动更新。

你可能会使用 sizeof(type) 来处理 不涉及任何变量的 代码。

```
Struct data;
Struct data; memset(&data, 0, sizeof(data));

if (raw_size < sizeof(int)) {
    LOG(ERROR) << "compressed record not big enough for count: "
<< raw_size;
    return false;
}
```

6.19 auto

使用 auto 绕过 繁琐的 类型名， 只要可读性好 就继续用，别用在 局部变量之外的 地方。

定义

C++11中，如果 变量被声明为 auto，那么它的类型 就会 被 自动匹配成 初始化表达式的类型。 你可以用 auto 来 复制 初始化 或 绑定引用。

```
vector<string> v;
...
auto s1 = v[0];  // 创建一份 v[0] 的拷贝。
const auto& s2 = v[0];  // s2 是 v[0] 的一个引用。
```

优点

C++类型名 有时 又长又臭，特别是 涉及模板 或 命名空间 的时候，就像：

```
sparse_hash_map<string, int>::iterator iter = m.find(val);
```

返回类型 难读，代码目的 也无法 一目了然，重构成

```
auto iter = m.find(val);
```

就好多了。

没有auto 的话，我们不得不 在 同一个表达式中 写同一个类型 2次 ：

```
diagnostics::ErrorStatus* status = new diagnostics::ErrorStatus("xyz");
```

缺点

类型够明显时，特别是 初始化变量时，代码才 一目了然，但下面 就不行

```
auto i = x.Lookup(key);
```

看不出类型是什么，x 的声明恐怕在 几百行外。

程序员必须 区分 auto 和 const auto& 的不同之处，否则会 复制错 东西。

auto 和 C++11 列表初始化 的 合体 令人摸不着头脑：

```
auto x(3);  // 圆括号。
auto y{3};  // 大括号。
```

上面是 不同的， x 是int， y 是 initializer_list<int> 。

其他一般不可见的 代理类型 也有大同小异的 陷阱。

如果在 接口中使用 auto，比如声明头文件中的 一个常量，那么只要 仅仅因为 程序员一时修改其值 而导致 类型变化的话 -- API 就要翻天覆地了。

结论

auto只能用在 局部变量中。
别用在 文件作用域 变量，命名空间作用域变量 和 类数据成员里。
永远别 列表初始化 auto 变量。

auto 还可以和 C++11 特性 [尾置返回类型(trailing return type)] 一起用，不过后者 只能用在 lambda 表达式中。

6.20 列表初始化

你可以使用 列表初始化。
早在 C++03 中，聚合类型 (aggregate type) 就已经可以 被列表初始化了，比如 数组和 不自带函数的 结构体

```
struct Point { int x; int y; };
Point p = {1, 2};
```

C++11 中，该特性得到了推广，任何对象类型 都可以被 列表初始化：

```
// Vector 接收了一个初始化列表。
vector<string> v{"foo", "bar"};

// 不考虑细节上的微妙差别，大致上相同。
// 您可以任选其一。
vector<string> v = {"foo", "bar"};

// 可以配合 new 一起用。
auto p = new vector<string>{"foo", "bar"};

// map 接收了一些 pair, 列表初始化大显神威。
map<int, string> m = {{1, "one"}, {2, "2"}};

// 初始化列表也可以用在返回类型上的隐式转换。
vector<int> test_function() { return {1, 2, 3}; }

// 初始化列表可迭代。
for (int i : {-1, -2, -3}) {}

// 在函数调用里用列表初始化。
void TestFunction2(vector<int> v) {}
TestFunction2({1, 2, 3});
```

用户自定义类型也可以 定义 接受 std::initializer_list<T> 的 构造器 和 赋值运算符，以自动列表 初始化

```
class MyType {
 public:
  // std::initializer_list 专门接收 init 列表。
```

```
    // 得以值传递。
    MyType(std::initializer_list<int> init_list) {
      for (int i : init_list) append(i);
    }
    MyType& operator=(std::initializer_list<int> init_list) {
      clear();
      for (int i : init_list) append(i);
    }
};
MyType m{2, 3, 5, 7};
```

最后，列表初始化 也适用于 常规数据类型的 构造，哪怕没有接受 std::initializer_list<T> 的构造器

```
double d{1.23};
// MyOtherType 没有 std::initializer_list 构造函数，
 // 直接上接收常规类型的构造函数。
class MyOtherType {
 public:
  explicit MyOtherType(string);
  MyOtherType(int, string);
};
MyOtherType m = {1, "b"};
// 不过如果构造函数是显式的（explict），您就不能用 `= {}` 了。
MyOtherType m{"b"};
```

千万不要 直接列表初始化 auto 变量。下面估计没有人看得懂

```
auto d = {1.23}; // d 即是 std::initializer_list<double>
```

6.21 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。

定义

Lambda 表达式 是创建 匿名函数 对象的 一种 简单途径，常用于 把 函数 当参数 传，例如

```
std::sort(v.begin(), v.end(), [](int x, int y) {
    return Weight(x) < Weight(y);
});
```

C++11首次提出 lambda，还提供了 一系列 处理函数 对象的 工具，比如 多态包装器 std::function

优点

传 函数对象 给 stl 算法，lambda 最简单，可读性也好。

lambda， std::functions, std::bind 可以 搭配成 通用 回调机制

写 接收 有界函数 为参数的 函数 也很容易。

缺点

lambda 的变量 捕获 略显 旁门左道，可能会 造成 悬空指针

lambda 可能 失控， 层层嵌套的 匿名函数 难以阅读

结论

按 format 小用 lambda 表达式 怡情。

禁用默认捕获，捕获都要 显式写出来，比如，不写 [=](int x){return x+n;} 而是写 [n](int x){return x+n;}，这样读者 也好 一眼看出 n 是被捕获的值。

匿名函数始终要 简短，如果函数体 超过了 五行，那么还不如 起名， 或改用 函数。

如果 可读性更好，就显式 写出 lambda 的 尾置 返回类型，就像 auto。

6.22 模板编程

不要使用 复杂的 模板编程

定义

模板编程 值得是 利用 C++ 模板实例化 机制 是 图灵完备性，可以被用来 实现 编程时 的 类型判断 的 一系列 编程 技巧。

优点

模板编程 能够实现 非常灵活的 类型安全的 接口 和 极好的性能， 一些常用的 工具，如 google test，std::tuple，std::functions，boost spirit， 如果没有模板，是实现不了的。

缺点

模板编程 所使用的 技巧对于 使用 C++ 不是很熟练的 人 是比较晦涩，难懂的。在复杂的地方使用 模板的代码 让人 更不容易读懂，并且 debug 和 维护起来 都很麻烦

模板编程 经常 会导致 编译出错的 信息 非常不友好：在代码出错的时候，即使这个接口 非常的 简单，模板内部 复杂的 实现细节 也会 在 出错 信息显示。导致这个 编译出错信息 看起来 非常 难以理解。

大量的 使用 模板编程 接口 会让 重构工具 (Visual Assist X, Refactor for C++ 等) 更难 发挥作用， 首先模板的代码 会在很多 上下文 中扩展出来，所以 很难确认 重构对 所有的 这些 展开的代码有用， 其次，有些重构工具 只对 已经做过 模板类型 替换的代码 的 AST 有用。因此 重构工具 对这些 模板实现的 原始代码 并不有效，很难找出 哪些需要重构。

结论

模板编程有时能够实现 更简洁 更易用的 接口，但是 更多的时候 却 适得其反。 因此 模板编程最好只用在 少量的 基础组件，基础数据结构上，因为模板带来的 额外的 维护成本 会被 大量 的使用 给分担掉。

在使用模板编程 或 其他复杂的 模板技巧的时候， 你一定要 再三考虑一下，考虑一下你们团队成员的 平均水平 是否 能够读懂 并 维护 你写的模板代码， 或者 一个 非C++程序员 和 一些 只是在 出错的时候 偶尔 看下 代码的 人 能够读懂 这些 错误信息 或能 跟踪 函数的 调用流程。 如果你使用 递归的 模板实例化，或者 类型列表，或者 元函数，又或者 表达式模板，或者 依赖 SFINAE， 或者 sizeof 的 trick 手段 来检查

函数是否 重载，那么 说明 你的模板 用的太多了，这些模板 太复杂，我们不推荐使用。

如果你使用 模板编程，你必须考虑 尽可能的 把复杂度 最小化，并且 尽量不要让 模板对外暴露。你最好只在 实现 里面 使用 模板，然后 给用户 暴露的 接口里面 并不使用模板，这样能提高 接口 可读性。并且 你应该 在这些 使用模板的 代码上 写 尽可能详细 的注释。 你的注释应该 包含 代码如何使用，模板生成的代码大约是什么样子。 还需要额外注意 在用户错误 使用 你的 模板代码的时候 需要 输出 更人性化的 出错信息。因为这些 出错信息 也是 你的接口的 一部分，所以你的代码 必须 调整到这些 错误信息 在用户看来 应该是 非常 容易理解， 并且 用户 很容易 知道 如何修改这些错误。

6.23 Boost 库

只使用 Boost 中被认可的库

定义

Boost 库 是一个 广受欢迎，经过同行鉴定，免费开源的 C++ 库集

优点

Boost 代码质量 普遍较高， 可移植性好， 填补了 C++ 标准库的 很多空白，如型别的特性，更完善的绑定器，更好的智能指针。

缺点

某些Boost 库 提倡的 编程实践 可读性差，比如 元编程 和 其他 高级模板技术，以及过度 函数化 的 编程风格

结论

为了向 阅读，维护代码 人员 提供 更好的可读性，我们只允许使用 Boost 一部分 经过认可 的特性子集， 目前允许的 使用下列库

Call Traits: boost/call_traits.hpp

Compressed Pair: boost/compressed_pair.hpp

boost/graph, except serialization (adj_list_serialize.hpp) and parallel/distributed algorithms and data structures(boost/graph/parallel/* and boost/graph/distributed/*)

Property Map : boost/property_map.hpp

The part of Iterator that deals with defining iterators: boost/iterator/iterator_adaptor.hpp,boost/iterator/iterator_facade.hpp, and boost/function_output_iterator.hpp

The part of Polygon that deals with Voronoi diagram construction and doesn't depend on the rest of Polygon: boost/polygon/voronoi_builder.hpp, boost/polygon/voronoi_diagram.hpp, and boost/polygon/voronoi_geometry_type.hpp

Bimap : boost/bimap

Statistical Distributions and Functions : boost/math/distributions

Multi-index : boost/multi_index

Heap : boost/heap

The flat containers from Container: boost/container/flat_map, and boost/container/flat_set

我们正在积极考虑增加 其他 Boost特性，所以 列表中的 规则将 不断变化，以下库可以用，但是由于 如今 已经被 C++11 标准库取代，所以不再鼓励

Pointer Container : boost/ptr_container, 改用 std::unique_ptr
Array : boost/array.hpp, 改用 std::array

6.24 C++11

适当用 C++11 的库 和语言扩展，在 贵项目使用 C++11 特性前 三思可移植性

。。。这个文章 太老了。。。

7.1 通用命名规则

总述

函数命名，变量命名，文件命名 要有描述性；少用缩写

说明

尽可能使用描述性的命名，别心疼空间，毕竟 相比之下 让代码 易于 让 新读者 理解更重要。

```
int price_count_reader;    // 无缩写
int num_errors;            // "num" 是一个常见的写法
int num_dns_connections;   // 人人都知道 "DNS" 是什么
```

```
int n;                     // 毫无意义.
int nerr;                  // 含糊不清的缩写.
int n_comp_conns;          // 含糊不清的缩写.
int wgc_connections;       // 只有贵团队知道是什么意思.
int pc_reader;             // "pc" 有太多可能的解释了.
int cstmr_id;              // 删减了若干字母.
```

一些广为人知 的 缩写是 允许的， i表示迭代， T 表示模板参数

模板参数的命名应当 遵循 对应的 分类， 类型模板参数 应该遵循 类型命名 的规则， 非类型模板 应该遵循 变量命名 的规则

7.2 文件命名

总述

文件名要全部小写，可以包含 下划线_ 或 连字符-， 按照项目的 约定， 如果没有约

定，下划线 更好

说明

可以接受的 文件 命名：

```
my_useful_class.cc
my-useful-class.cc
myusefulclass.cc
myusefulclass_test.cc // _unittest 和 _regtest 已弃用.
```

C++文件要以 .cc 结尾，头文件以 .h 结尾。 专门插入文本的 文件则以 .inc 结尾。

不要使用 已经存在于 /usr/include 下的文件名 (即 编译器搜索 系统头文件的 路径)

通常应该尽量让 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。

7.3 类型命名

总述

类型名称的 每个单词首字母均大写， 不包含下划线 MyExcitingClass,MyExcitingEnum.

说明

所有类型命名 —— 类，结构体，类型定义 (typedef)，枚举，类型模板参数 —— 均使用相同约定，即以大写字母开始，每个单词首字母均大写，不包含下划线.

7.4 变量命名

总述

变量(包括函数参数) 和数据成员 名 一律小写，单词间用 下划线 连接。
类的成员变量 以下划线结尾，但 结构体的 不用a_local_variable, a_struct_data_member, a_class_data_member_.

```
string table_name;  // 好 - 用下划线.
string tablename;   // 好 - 全小写.
string tableName;  // 差 - 混合大小写
```

。。还真是 驼峰 是差。。最新的 guide 也是， 不过 最新的里 没有 全小写 这种了。。。驼峰 有2种，这种是 小驼峰，就是 首字母小写， 大驼峰 是首字母大写，是函数的命名规则

```
class TableInfo {
  ...
 private:
  string table_name_;  // 好 - 后加下划线.
  string tablename_;   // 好.
  static Pool<TableInfo>* pool_;  // 好.
};
```

```
struct UrlTableProperties {
  string name;
  int num_entries;
  static Pool<UrlTableProperties>* pool;
};
```

7.5 常量命名

总述

声明为 constexpr 或 const 的变量，或在 程序运行期间 其值始终不变的，命名时以 k 开头，大小写混合

```
const int kDaysInAWeek = 7;
```

说明

所有具有 静态存储类型 的变量（例如 静态变量 或 全局变量）都应该以此方式命名。对于其他 存储类型的变量，如 自动变量等，这条规则是可选的。 如果不采用这条规则，就按照一般的 变量 命名规则。

7.6 函数命名

总述

常规函数 使用 大小写混合，取值 和 设置 函数 则 要求 与 变量名匹配：

```
MyExcitingFunction(), MyExcitingMethod(), my_exciting_member_variable(),
set_my_exciting_member_variable().
```

说明

一般来说，函数名的 每个单词 首字母大写，没有下划线。 对于 首字母缩写的单词，更倾向于 把它们视为 一个单词 进行 首字母大写（例如， StartRpc() 而不是 StartRPC()）

同样的命名规则同时 适用于 类作用域 与 命名空间作用域的 常量，因为 它们是 作为 API 的一部分 对外暴露的， 因此 应当让它们看起来 像是一个 函数，因为在这时， 它们实际上 是一个 对象 而不是 函数， 这一事实 对外来说 是一个 无关紧要的 实现细节。

7.7 命名空间命名

总述

命名空间 以 小写字母 命名。 最高级 命名空间 的名字取决于 项目名称。要注意 避免嵌套命名空间 的名字 之间 和 常见的 顶级命名空间的名字之间的 冲突。

顶级命名空间的名称 应当是 项目名 或者是 该命名空间中 代码 所属的 团队的 名字。命名空间中的代码，应当存放在 和命名空间 的名字 匹配的 文件夹 或 其子文件夹中。

注意 不要使用缩写作为名称。 命名空间中的 代码 极少需要 涉及 命名空间的名称，因此没有必要 在 命名空间中 使用 缩写。

要避免嵌套的 命名空间 与 常见的 顶级命名空间 发生 名称冲突。 由于 名称查找规则的 存在，命名空间 之间的 冲突 完全 有可能导致 编译失败。 尤其是，不要创建 嵌套的 std 命名空间。 建议使用 更独特的 项目标识符(websearch::index, websearch::index_util) 而非 极易发生冲突的 名称 (websearch::util)

对于 internal 命名空间，要当心 加入到 同一个 internal 命名空间的 代码之间 发生冲突，这种情况下，请使用 文件名 以 使得 内部名称 独一无二，(例如，对于 foobar.h， 使用 websearch::index::foobar_internal)

7.8 枚举命名

总述

枚举的命名 应当和 常量 或 宏 一致： kEnumName 或 ENUM_NAME

说明

单独的枚举值 应该优先 采用 常量的命名方式。 但 宏方式 的命名也接受。
枚举名 是类型，所以是 首字母大写。

```
enum UrlTableErrors {
    kOK = 0,
    kErrorOutOfMemory,
    kErrorMalformedInput,
};
enum AlternateUrlTableErrors {
    OK = 0,
    OUT_OF_MEMORY = 1,
    MALFORMED_INPUT = 2,
};
```

2009年1月之前，我们一直建议 采用 宏 的命名方式 命名枚举值。 由于 枚举值 和 宏之间的 命名冲突，直接导致了 很多问题，所以 这里 改为 优先选择 常量风格 的 命名方式。

7.9 宏命名

总述

最好不要用宏， 如果要用，像这样命名：MY_MACRO_THAT_SCARES_SMALL_CHILDREN

如果不得不用，其命名像枚举命名一样全部大写，使用下划线

7.10 命名规则的特例

如果你的命名的 实体 和 现有的 C/C++ 实体类似，可以参考现有的命名策略

```
bigopen()
    function name, follows form of open()
uint
    typedef
bigpos
```

```
    struct or class, follows form of pos
sparse_hash_map
    STL-like entity; follows STL naming conventions
LONGLONG_MAX
    a constant, as in INT_MAX
```

8.1 注释风格
使用 // 或 /* */， 统一就好
通常 // 更常用

8.2 文件注释
每个文件开头加入 版权公告
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 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 就没有必要 再加上文件注释。

如果一个 .h 文件 声明了 多个概念，则文件注释应当 对文件的 内容 做一个 大概的说明，同时指出 各个概念间的 关系。1-2行就足够。 每个概念的详细文档 就放在 每个 概念中，而不是 文件注释中。

不要在 .h 和 .cc 间 复制注释，这样的 注释 偏离了 注释的 实际意义。

8.3 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，除非它的功能相当明显。

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

类注释 需要为 读者 理解 如何使用，何时使用 类 提供足够的信息，
同时 要提醒 读者 在正确使用 此类时 需要 考虑的 因素。
如果类有 任何 同步前提，用文档说明。
如果类的 实例 可以 被 多线程 访问，要特别说明 多线程环境下 相关的规则 和 常量使用。

可以放 一小段代码 演示 基本用法。

如果类的声明 和定义 分开 (如 分别放在 .h 和 .cc 文件中)， 此时，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 实现放一起。

8.4 函数注释
函数声明处 的注释 描述 函数功能
函数定义处 的注释 描述 函数实现

函数声明
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.
函数是否分配了必须由调用者释放的空间.
参数是否可以为空指针.
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?

```
// 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;
```

注释 函数重载时，重点 是 函数中 被重载部分，而不是 简单 重复 被重载的 函数的 注释。多数情况下，函数重载 不需要额外的 文档，因此也必要 加上注释

构造器，析构器 的注释， 切记 读者知道 构造器 和 析构器的 功能，所以 "销毁这一对象" 这种注释 是 无意义的。
你应该注明 构造器 对参数 做了什么 (例如， 是否取得 指针所有权)， 析构函数 清理了什么。 如果都是无关紧要 的内容，直接省略注释。 析构器 没有注释 是很正常的。

函数定义
如果函数的 实现过程中 用到了 很巧妙的方式，那么 在 函数定义处 应该加上 解释性的 注释，例如 你使用的 编程技巧，实现的大致步骤，或解释 如此实现的 原因。 例如，你可以说明 为什么 函数 前半部分 要加锁，而 后半部分 不需要。

不要从 .h 上直接复制 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。

8.5 变量注释
通常，变量名 本身 就很好滴说明了 变量用途。

类成员变量， 全局变量 应该加上注释 说明 含义 和 用途 ，对于全局变量，还要说明 作为全局变量的原因。

8.6 实现注释
对于 代码中 巧妙的，晦涩，有趣，重要 的 地方加上注释

8.7 标点，拼写，语法

8.8 TODO注释
对于 临时的，短期的 解决方案，或 已经好 但仍不完美 的代码 使用 TODO 注释。

8.9 弃用注释
通过 DEPRECATED 来标记 某个接口 已经被 弃用。

9.1 行长度
不超过80

9.2 非ASCII字符
尽量不使用 非ASCII字符， 必须是 UTF-8 编码

9.3 空格 还是 制表符
只使用空格，每次缩进 2个 空格

9.4 函数声明和定义
返回类型 和 函数名 在同一行， 参数 也尽量 同一行

使用好的参数名.
只有在参数未被使用或者其用途非常明显时，才能省略参数名.
如果返回类型和函数名在一行放不下，分行.
如果返回类型与函数声明或定义分行了，不要缩进.
左圆括号总是和函数名在同一行.
函数名和左圆括号间永远没有空格.
圆括号与参数间没有空格.
左大括号总在最后一个参数同一行的末尾处，不另起新行.
右大括号总是单独位于函数最后一行，或者与左大括号同一行.
右圆括号和左大括号间总是有一个空格.

所有形参应尽可能对齐.
缺省缩进为 2 个空格.
换行后的参数保持 4 个空格的缩进.

未被使用的参数, 或者根据上下文很容易看出其用途的参数, 可以省略参数名:

```
class Foo {
 public:
  Foo(Foo&&);
  Foo(const Foo&);
  Foo& operator=(Foo&&);
  Foo& operator=(const Foo&);
};
```

。。? 那怎么用这个参数?

未被使用的参数如果其用途不明显的话, 在函数定义处将参数名注释起来:

```
class Shape {
 public:
  virtual void Rotate(double radians) = 0;
};

class Circle : public Shape {
 public:
  void Rotate(double radians) override;
};

void Circle::Rotate(double /*radians*/) {}
```

属性, 和展开为属性的宏, 写在函数声明或定义的最前面, 即返回类型之前

```
MUST_USE_RESULT bool IsOK();
```

9.5 lambda

Lambda 表达式对形参和函数体的格式化和其他函数一致; 捕获列表同理,表项用逗号隔开.

若用引用捕获, 在变量名和 & 之间不留空格.

9.6 函数调用

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.

9.7. 列表初始化格式

9.8. 条件语句

```
if (condition) {  // 圆括号里没有空格.
  ...  // 2 空格缩进.
} else if (...) {  // else 与 if 的右括号同一行.
  ...
} else {
  ...
}

if ( condition ) {  // 圆括号与空格紧邻 - 不常见
  ...  // 2 空格缩进.
} else {  // else 与 if 的右括号同一行.
  ...
}
```

如果语句中某个 if-else 分支使用了大括号的话, 其它分支也必须使用

9.9. 循环和开关选择语句

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 (如果有输入值没有 case 去处理, 编译器将给出 warning). 如果 default 应该永远执行不到, 简单的加条 assert

空循环体应使用 {} 或 continue, 而不是一个简单的分号.

```
while (condition) {
  // 反复循环直到条件失效.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // 可 - 空循环体.
while (condition) continue;  // 可 - contunue 表明没有逻辑.

while (condition);  // 差 - 看起来仅仅只是 while/loop 的部分之一.
```

9.10. 指针和引用表达式

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.

正确示例：

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

在访问成员时，句点或箭头前后没有空格.
指针操作符 * 或 & 后没有空格.

在声明指针变量或参数时，星号与类型或变量名紧挨都可以

```
// 好，空格前置.
char *c;
const string &str;

// 好，空格后置.
char* c;
const string& str;

int x, *y;  // 不允许 - 在多重声明中不能使用 & 或 *
char * c;  // 差 - * 两边都有空格
const string & str;  // 差 - & 两边都有空格.
```

要保持风格一致

9.11. 布尔表达式

如果一个布尔表达式超过 标准行宽 <line-length>，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) {
  ...
}
```

注意，上例的逻辑与 (&&) 操作符均位于行尾. 这个格式在 Google 里很常见，
虽然把所有操作符放在开头也可以. 可以考虑额外插入圆括号，
合理使用的话对增强可读性是很有帮助的.

直接用符号形式的操作符，比如&& 和 ~，不要用词语形式的 and 和 compl

9.12. 函数返回值

不要在 return 表达式里加上非必须的圆括号.

只有在写 x = expr 要加上括号的时候才在 return expr; 里使用括号.

9.13. 变量及数组初始化
用 =, () 和 {} 均可.

以下的例子都是正确的：

```
int x = 3;
int x(3);
int x{3};
string name("Some Name");
string name = "Some Name";
string name{"Some Name"};
```

请务必小心列表初始化 {...} 用 std::initializer_list
构造函数初始化出的类型. 非空列表初始化就会优先调用
std::initializer_list, 不过空列表初始化除外,
后者原则上会调用默认构造函数. 为了强制禁用 std::initializer_list
构造函数, 请改用括号.

```
vector<int> v(100, 1);  // 内容为 100 个 1 的向量.
vector<int> v{100, 1};  // 内容为 100 和 1 的向量.
```

列表初始化不允许整型类型的四舍五入,

```
int pi(3.14);  // 好 - pi == 3.
int pi{3.14};  // 编译错误: 缩窄转换.
```

9.14. 预处理指令
预处理指令不要缩进, 从行首开始.

即使预处理指令位于缩进代码块中, 指令也应从行首开始.

```
// 好 - 指令从行首开始
  if (lopsided_score) {
#if DISASTER_PENDING      // 正确 - 从行首开始
    DropEverything();
# if NOTIFY               // 非必要 - # 后跟空格
    NotifyClient();
# endif
#endif
    BackToNormal();
  }
```

9.15. 类格式

访问控制块的声明依次序是 public:, protected:, private:, 每个都缩进1 个空格.

类声明 (下面的代码中缺少注释, 参考 类注释 <class-comments>)
的基本格式如下:

```
class MyClass : public OtherClass {
 public:      // 注意有一个空格的缩进
  MyClass();  // 标准的两空格缩进
  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_;
};
```

。。。感觉 差好多，主要是 public 还缩进，还缩进1个空格。。

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: 要缩进 1 个空格.
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 <declaration-order> 一节.

9.16. 构造函数初始值列表

构造函数初始化列表放在同一行或按四格缩进并排多行.

下面两种初始值列表方式都可以接受:

```
// 如果所有变量能放在同一行:
MyClass::MyClass(int var) : some_var_(var) {
  DoSomething();
}
```

```
// 如果不能放在同一行,
// 必须置于冒号后, 并缩进 4 个空格
MyClass::MyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
  DoSomething();
}

// 如果初始化列表需要置于多行, 将每一个成员放在单独的一行
// 并逐行对齐
MyClass::MyClass(int var)
    : some_var_(var),             // 4 space indent
      some_other_var_(var + 1) {  // lined up
  DoSomething();
}

// 右大括号 } 可以和左大括号 { 放在同一行
// 如果这样做合适的话
MyClass::MyClass(int var)
    : some_var_(var) {}
```

9.17. 命名空间格式化
命名空间内容不缩进.

```
namespace {

void foo() {  // 正确. 命名空间内没有额外的缩进.
  ...
}

}  // namespace
```

声明嵌套命名空间时, 每个命名空间都独立成行.

```
namespace foo {
namespace bar {
```

9.19. 水平留白
水平留白的使用根据在代码中的位置决定. 永远不要在行尾添加没意义的留白.

```
void f(bool b) {  // 左大括号前总是有空格.
  ...
int i = 0;  // 分号前不加空格.
```

```
// 列表初始化中大括号内的空格是可选的.
// 如果加了空格, 那么两边都要加上.
int x[] = { 0 };
int x[] = {0};

// 继承与初始化列表中的冒号前后恒有空格.
class Foo : public Bar {
 public:
  // 对于单行函数的实现, 在大括号内加上空格
  // 然后是函数实现
  Foo(int b) : Bar(), baz_(b) {}  // 大括号里面是空的话, 不加空格.
  void Reset() { baz_ = 0; }  // 用空格把大括号与实现分开.
  ...
```

循环和条件语句

```
if (b) {          // if 条件语句和循环语句关键字后均有空格.
} else {          // else 前后有空格.
}
while (test) {}   // 圆括号内部不紧邻空格.
switch (i) {
for (int i = 0; i < 5; ++i) {
switch ( i ) {    // 循环和条件语句的圆括号里可以与空格紧邻.
if ( test ) {     // 圆括号, 但这很少见. 总之要一致.
for ( int i = 0; i < 5; ++i ) {
for ( ; i < 5 ; ++i) {  // 循环里内 ; 后恒有空格, ;  前可以加个空格.
switch (i) {
  case 1:         // switch case 的冒号前无空格.
    ...
  case 2: break;  // 如果冒号有代码, 加个空格.
```

操作符

```
// 赋值运算符前后总是有空格.
x = 0;

// 其它二元操作符也前后恒有空格, 不过对于表达式的子式可以不加空格.
// 圆括号内部没有紧邻空格.
v = w * x + y / z;
v = w*x + y/z;
v = w * (x + z);

// 在参数和一元操作符之间不加空格.
x = -5;
++x;
if (x && !y)
```

...

模板和转换

```
// 尖括号(< and >) 不与空格紧邻, < 前没有空格, > 和 ( 之间也没有.
vector<string> x;
y = static_cast<char*>(x);

// 在类型与指针操作符之间留空格也可以, 但要保持一致.
vector<char *> x;
```

9.19. 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.

下面的规则可以让加入的空行更有效:

- 函数体内开头或结尾的空行可读性微乎其微.
- 在多重 if-else 块里加空行或许有点可读性.

10.1. 现有不合规范的代码

对于现有不符合既定编程风格的代码可以网开一面.

可以与代码原作者或现在的负责人员商讨. 记住, 一致性 也包括原有的一致性.

10.2. Windows 代码

Windows 程序员有自己的编程习惯, 主要源于 Windows 头文件和其它 Microsoft 代码. 我们希望任何人都可以顺利读懂你的代码, 所以针对所有平台的 C++ 编程只给出一个单独的指南.

如果你习惯使用 Windows 编码风格,
这儿有必要重申一下某些你可能会忘记的指南:

- 不要使用匈牙利命名法 (比如把整型变量命名成 iNum). 使用 Google 命名约定, 包括对源文件使用 .cc 扩展名.

Windows 定义了很多原生类型的同义词,
如 DWORD, HANDLE 等等. 在调用 Windows API
时这是完全可以接受甚至鼓励的. 即使如此, 还是尽量使用原有的 C++ 类型,
例如使用 const TCHAR * 而不是 LPCTSTR.

使用 Microsoft Visual C++ 进行编译时, 将警告级别设置为 3 或更高,
并将所有警告(warnings)当作错误(errors)处理.

不要使用 #pragma once; 而应该使用 Google 的头文件保护规则.
头文件保护的路径应该相对于项目根目录 (Yang.Y 注: 如
#ifndef SRC_DIR_BAR_H_, 参考 #define 保护 <define-guard> 一节).

除非万不得已, 不要使用任何非标准的扩展, 如 #pragma 和
__declspec. 使用 __declspec(dllimport) 和
__declspec(dllexport) 是允许的, 但必须通过宏来使用, 比如
DLLIMPORT 和 DLLEXPORT,
这样其他人在分享使用这些代码时可以很容易地禁用这些扩展.

然而, 在 Windows 上仍然有一些我们偶尔需要违反的规则:

通常我们 禁止使用多重继承 <multiple-inheritance>, 但在使用 COM 和
ATL/WTL 类时可以使用多重继承. 为了实现 COM 或 ATL/WTL 类/接口,
你可能不得不使用多重实现继承.

虽然代码中不应该使用异常, 但是在 ATL 和部分 STL（包括 Visual C++ 的
STL）中异常被广泛使用. 使用 ATL 时, 应定义 _ATL_NO_EXCEPTIONS
以禁用异常. 你需要研究一下是否能够禁用 STL 的异常, 如果无法禁用,
可以启用编译器异常. (注意这只是为了编译 STL,
自己的代码里仍然不应当包含异常处理).

通常为了利用头文件预编译, 每个每个源文件的开头都会包含一个名为
StdAfx.h 或 precompile.h 的文件. 为了使代码方便与其他项目共享,
请避免显式包含此文件 (除了在 precompile.cc 中), 使用 /FI
编译器选项以自动包含该文件.

资源头文件通常命名为 resource.h 且只包含宏,
这一文件不需要遵守本风格指南.

==========

==========

尽量不要使用using namespace std

A：

1. 要保证 尽量不要在 头文件 中 using 任何东西，尤其是 namespace，要不然 include 进来的时候 很容易 莫名其妙产生命名冲突。
2. 有条件的话，所有引入的 符号 都定义在 自己的 namespace 中。 任何情况下 都不要 using namespace std

B:
底线就一条：如果你的 头文件 (.h, .hpp) 有被外部使用，则不要 使用 任何 using 语句 引入 其它 命名空间 或 其它 命名空间中的标识符。
因为这可能会给 使用你的头文件的 人添麻烦。更何况 头文件之间都是 相互嵌套的， 如果每个人 都在 头文件中 包含 若干个 命名空间，到了第N层后发现 命名冲突，这得 回溯 多少层 才能找到 冲突啊。 而这个冲突 本来是可以避免的。

其实在 源文件(.cpp) 中 怎么使用 using 都没有关系的，因为 cpp 中的 代码 不影响 其他人。 甚至，如果你的 头文件 只是自己用，那using 也没事的。

C：
举个例子，我们原来的 代码一直没有问题，但如果要 使用 C++ 17 就报错，原因是 C++17 增加了 一种新的 类型 std::byte，而 windows 头文件中 自带一种 byte 类型。
这2个类型 本身不冲突，因为一个 是 std::byte， 另一个是 byte。

但是，如果 代码中 广泛使用了 using namespace std，再 遇到 byte 时， 编译器 就不知道它是 windows 的 byte 还是 std::byte。
如果你想省事，可以在 自己的 .cpp 文件中使用 (.cpp 不暴露 声明)。 实在想在 .h 里用(为省事)，尽量 使用 诸如 using std::vector 之类的， 用哪个 就using 哪个，而不是 一次性全部using。

D:
这就跟 python 里你写：

```
from numpy import *
from pandas import *
from tensorflow import *
```

一样，很容易有冲突。
C++ Primer Plus 第六版，中文版，人民邮电出版社，第九章，内存模型和名称空间，第 328 页："有关 using 编译命令 和 using 声明，需要记住的一点是，它们增加了 名称冲突的 可能性"
第329页："一般来说，使用 using 命令 比使用 using 编译命令 更安全，这是因为 它只导入了 指定的名称。 如果这个名称 与 局部名称 发生冲突，编译器 将发出指示。using 编译命令 导入 所有的名称，包括 可能并不需要 的名称。 如果 与 局部 名称 发生冲突，则 局部名称 将覆盖 名称空间版本，而编译器 并不会发出警告。另外，名称空间 的开放性 意味着 名称空间 的名称 可能分散在 多个地方，这使得 难以 准确知道 添加了 哪些名称。"
不要这样做：

```
using namespace std; // avoid as too indiscriminate(随意)
```

而是应该：

```
int x;
std::cin >> x ;
std::cout << x << std::endl;
```

或

```
using std::cin;
using std::cout;
using std::endl;
int x;
cin >> x;
cout << x << endl;
```

总之，头文件中不要用 using namespace。 因为 头文件内容 相当于 一段 代码的 公开部分，会在 预处理阶段 被替换进 引用者的 源文件里。

==========

==========

https://zhuanlan.zhihu.com/p/519212219
Google C++Style Guide 概要

编写此手册是为了实现下面八大编程原则：

1. style rules should pull their weight
   对语言风格加以权重
2. optimize for the reader, not the writer
   针对阅读者优化，而不是编写者，Google社区认为工程师更多时间是在 看代码。
3. be consistent with existing code

与现有代码保持一致

4. be consistent with the brorder c++ community when appropriate
   适当地与C++社区 保持一致
5. avoid surprising or dangerous constructs
   避免意外的 或 危险的构造
6. avoid constructs that our average c++ programmer would find tricky or hard to maintain
   避免那些大多数C++编程者难以理解 或 太过技巧性的构造
7. be mindful of our scale
   注意我们的规模
8. concede to optimization when necessory
   在必要的时候 对性能进行优化

C++版本应该是 17 而不是 2x

头文件
一般而言，一个 .cc 文件都必须对应一个 .h 头文件，除了单元测试 和 只包含 main 函数的文件。

头文件应该是独立的
函数和类 的声明 都必须在 .h 头文件中，模板 必须定义在 .inc 文件中。

#define 标头
#define 的格式应该是 <Project>_<PATH>_<FILE>_H
例如 foo工程 的 src/bar/baz.h头文件标头按照下面定义：

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

只引用你使用的 头文件

Forward Declaration 前向声明
尽量避免 前置声明 那些 定义在 其他项目中的 实体。

Inline Functions 内联函数
编译时，内联函数 会在 调用的 位置被展开，从而减少 时间开销，这是一种典型 的 空间换时间的方式
但过度使用 会导致 程序运行 的时间更慢。内联后的大小 与函数本身的大小有关。内联一个很小的 存取函数 通常会 减小代码体积，而内联一个 较大的 函数 会导致 代码体积增大。在现在处理器上，小的 代码通常跑的 更快，因为 它能更好地 利用 指令缓存(instruction cache)
策略：

内联函数通常用于 定义那些 小于等于 10行的 函数。注意那些 析构函数，它们比想象的 更大一些，因为 它们会调用 基类 和 成员析构函数。所以 不要对 析构函数进行 内联
通常 不建议 对 含有 循环 或选择的 函数 进行析构。
注意，即使你 声明了一个 函数 为内联的，它也不一定 被 编译器看做内联的，比如 递归函数 不会被内联。

Names and Orders of Includes 包含的名字和顺序
按照如下顺序包含头文件

1. 相对路径的头文件
2. C系统头文件
3. C++标准库头文件
4. 其他库的header
5. 本项目的header

请不要使用 UNIX 目录的 . 或 .. 。

```
//相对路径头文件
#include<dir2/foo2.h>
...

//C系统头文件
#include<unistd.h>
#include<stdlib.h>
...
//C++标准头文件
#include<vector>
#include<algorithm>
#include<memory>
...
//其它库的.h文件
#include"xxx/xxx.h"
...
//本项目的头文件
#include"XXX.h"
```

有时候，由于系统不同，引用的头文件也不同，为了实现跨平台特征，可以用宏：

```
#include "foo/public/fooserver.h"

#include "base/port.h"  // For LANG_CXX11.

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

Scoping 作用域

Namespace 命名空间
命名空间的主要作用是 避免命名冲突。例如 常见的 std:: 就是 标准的 命名空间。
在命名空间内定义的 内联的 命名空间 会被自动展开，例如下面：

```
namespace outer {
inline namespace inner {
  void foo();
}  // namespace inner
}  // namespace outer
```

使用 outer::inner::foo() 等价于 outer::foo()

策略
遵循命名空间名字所描述的规则
请全部使用小写，单词之间下划线
避免与 预定义的命名空间重名，如std
不要使用缩写
在命名空间结束的位置写上注释
在包含 gflags 定义/声明 和来自其他命名空间的 类的 前向定义，包含整个源文件
要将生成的协议消息代码 放在命名空间中，请使用 .proto文件中的 包 说明符
不要在 命名空间std 中声明任何内容，包括 标准库类 的前向声明。在 命名空间 std 中 声明实体 是未定义的行为。
要声明标准库中的实体，请包含适当的头文件。不能使用 using 指令来 使用命名空间中的 所有名称 (禁止 using namespace foo;)
除非是 只在内部使用的 清晰定义 的命名空间，否则 不要 使用 命名空间别名。因为 任何导入的 命名空间 都会成为 该文件的 API 的一部分 对外部调用者开放，这可能产生安全隐患。
不要使用内联的命名空间

Internal Linkage 内部链接
定义
当定义在 .cc 的 变量 或函数 不需要被外部引用，请把它们定义为 static，也就是 未命名的 命名空间

策略
鼓励使用内部链接，对 未命名 的命名空间 也可以 使用类似的 格式

```
namespace {
...
}  // namespace
```

Nonmember, Static Member, and Global Functions 非成员，静态成员和全局函数
策略
对非成员函数，应该放在一个命名空间内
尽可能少地 使用全局函数
不要用一个类 简单地 集中静态成员。静态成员 应该尽可能 接近类的 实例 或 类的 静态数据。
静态成员和非成员 可以作为 新类的 成员尤其是 如果 它们需要获取 外部资源 或者有很多依赖的时候。

Local Variables 局部变量
尽量把 局部变量定义在一起，并且接近第一次使用的地方。

```
int i;
i = f();      // 糟糕。初始化和定义分开。
int j = g();  // 很棒。
std::vector<int> v;
v.push_back(1);  // 更倾向于用括号初始化
v.push_back(2);
std::vector<int> v = {1, 2};  // 很棒。
```

在if, while, for中的变量应该被限制在作用域内：

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

如果变量是一个对象，那么请不要定义在循环内，下面的情况非常糟糕

```
for (int i = 0; i < 1000000; ++i) {
  Foo f;
  f.DoSomething(i);
}
```

每次循环都需要 构造 和 析构，效率低。

Static and Global Variables 静态和全局的变量
禁止 静态存储周期 的对象，除非它们是 平凡可析构 的。
关于存储周期，有4种：auto,static,thread,dynamic.

auto。对象的存储空间在封闭代码块的开头分配并在结尾处释放。所有本地对象都有这个存储持续时间，除了那些声明为 static、extern 或 thread_local 的对象。

static。对象的存储在程序开始时分配，在程序结束时释放。该对象仅存在一个实例。所有在命名空间范围内声明的对象（包括全局命名空间）都有这个存储持续时间，加上那些用 static 或 extern 声明的对象。有关具有此存储持续时间的对象初始化的详细信息，请参阅非局部变量和静态局部变量。

thread。对象的存储在线程开始时分配，在线程结束时释放。每个线程都有自己的对象实例。只有声明为 thread_local 的对象具有此存储持续时间。 thread_local 可以与 static 或 extern 一起出现以调整链接。有关具有此存储持续时间的对象初始化的详细信息，请参阅非局部变量和静态局部变量。(C++11 起)

dynamic。使用动态内存分配函数根据请求分配和释放对象的存储空间。有关具有此存储持续时间的对象初始化的详细信息，请参见 new-expression。 关于平凡可析构的 (trivially destructible) 指那些析构函数不是用户定义的，并且不是虚函数的类型。例如基本类型都是平凡可析构的。

技巧：由于动态初始化，对静态类成员 或者命名空间 作用域的 变量 使用 constexpr 修饰，也就是变为常量。
动态初始化(dynamic initialization)，比如 一个分配内存的 构造器 或者 分配当前的进程

ID。

静态初始化 总发生在 对象的 静态storage周期(初始化 对象为一个 给定的常量 或 一个全0 byte的东西)；如果有需要，则动态初始化在这个之后发生。

策略：
析构
允许的情况

```
const int kNum = 10;

struct X { int n; };
const X kX[] = {{1}, {2}, {3}};

void foo() {
  static const char* const kMessages[] = {"hello", "world"};
}

constexpr std::array<int, 3> kArray = {1, 2, 3};
```

不允许的情况

```
// 糟糕: 非平凡的析构器
const std::string kFoo = "foo";

// 同样糟糕, 即使 kBar 是引用 (这条规则同样适用于延长生命周期的临时对象).
const std::string& kBar = StrCat("a", "b", "c");

void bar() {
  // 糟糕: 非平凡的析构器.
  static std::map<int, int> kData = {{1, 0}, {2, 0}, {3, 0}};
}
```

注意，引用不是对象，因此它们不受 可破坏性的约束。不过，动态初始化的约束仍然适用。特别是 `static T& t = *new T` 形式的 函数局部静态引用。

初始化
我们提出 常量初始化的 概念：初始化表达式 是常量表达式，如果对象由构造函数初始化，那么 构造函数 也必须被标位 constexpr。

```
// Some declarations used below.
time_t time(time_t*);      // Not constexpr!
int f();                   // Not constexpr!
struct Bar { Bar() {} };

// Problematic initializations.
time_t m = time(nullptr);  // Initializing expression not a constant expression.
Foo y(f());                // Ditto
Bar b;                     // Chosen constructor Bar::Bar() not constexpr.
```

下面的情况是允许的

```
int p = getpid();  // Allowed, as long as no other static variable
                   // uses p in its own initialization.
```

但是允许动态初始化本地的静态变量
对于下列模式，可以考虑：
全局字符串：考虑使用 constexpr
哈希表，集合，或其他动态容器：考虑使用 函数局部静态指针
智能指针会在 析构时被清理，因此被禁止在 全局 或 静态变量 使用。一个简单做法是 使用一般 指针，并且 动态分配，从不删除。
自定义类型的静态变量：如果需要自己定义的类型的 静态常量数据，请为 该类型提供一个 普通析构函数 和 一个 constexpr 构造函数。
如果一切都失败了，可以动态创建一个对象，并且 永远不要使用 函数局部静态指针 或 引用 将其删除 (例如 static const auto& impl = *new T(args...);)

thread_local 变量
是针对每个线程而言的 静态变量，通常用于并发编程中。

策略
函数内的 thread_local 变量 没有安全问题，因此可以不受限制地使用。请注意，你可以使用函数范围的 thread_local 来模拟类 或 命名空间范围的 thread_local，方法是 定义公开它的 函数 或静态方法

```
Foo& MyThreadLocalFoo() {
  thread_local Foo result = ComplicatedInitialization();
  return result;
}
```

类或命名空间范围类的 thread_local 变量 必须使用 真正的编译时常量 进行初始化 (即，它们不能 动态初始化)。为了强制这一点，类 或 命名空间 范围内的 thread_lcaol 变量必须使用 ABSL_CONST_INIT (或 constexpr，但这比较少见) 进行修饰

```
ABSL_CONST_INIT thread_local Foo foo = ...;
```

Classes 类

Constructors 构造器
由于基类构造函数 先于 子类执行，请不要在构造器中 调用虚函数。

在构造器内 发出错误信号 并不容易，除非 使程序崩溃 或 抛出异常。 否则考虑 init() 方法，避免在没有其他状态 影响 可以调用哪些公共方法 的对象上 使用 init() 方法 (这种形式 半构造对象特别难以正确使用)

Implicit Conversion 隐式转换
不要定义隐式转换。 对转换算符 或者 单参数构造器 使用 explicit 关键字

explicit 关键字 可以应用于 构造函数 或转换运算符，以确保 它只能在目标类型 在使用时

显式时 使用，例如，使用 强制转换。 这不仅适用于 隐式转换，还适用于 列表初始化语法：

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

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

策略
类型转换算符 或 单参数构造器 必须是 explicit 的
复制 和 移动 构造器 不可以是 explicit，因为它们没有 做转换。

Copyable and Movable Types 可复制或者可移动的类型
可移动类型指 可以通过 临时变量 初始化 或赋值 的类型；
可复制类型指可以通过 同类型的 任何其他 对象初始化 或赋值的 类型。

优点
可复制 和 可移动类型的 对象 可以按值传递 和返回，这使得API 更简单，更安全，更通用。与通过 指针 和 引用 传递对象不同，没有所有权，生命周期，可变性，和 类似问题的混淆风险，它还可以防止 客户端和 实现之间的 非本地交互，这使得 它们更容易被编译器 理解，维护，优化。此外，这些对象 可以与 需要 按值传递的 通用API 一起使用，例如 大多数容器，并且 它们允许在类型组合 等方面 具有 额外的额灵活性。

复制/移动 构造函数 和 赋值运算符 通常比 Clone()，CopyFrom()，Swap() 等 替代方法 更容易正确 定义，因为 它们可以由编译器生成，无论是 隐式生成 还是 使用=默认值。 它们简洁，并确保 复制 和 移动 构造函数 通常也 更有效，因为 它们不需要 堆分配 或 单独的 初始化 和 分配步骤，并且 它们有资格进行 复制省略 等优化。

移动操作 允许 从右值对象 中 隐式 有效地 转移资源，这在 某些情况下 允许更简单的 编码风格。

缺点
某些 类型 不必是 可复制的：单例对象(如 Register)，绑定到 特定范围的对象(如 Cleanup)，与对象标识紧密耦合的类型(如 Mutex)。 对 要多态使用的 基类类型的 复制操作是危险的， 因为使用 它们会导致 对象切片
对象切片：当一个子类的对象被复制到基类的对象，通过超类的 复制运算符，那么目标对象的某些 成员变量 会保持 原来的值 而不是 被复制。

复制构造函数 是隐式调用的，这可能会造成问题。

策略：
每个类的公共接口 都必须 明确 是否支持 复制 和 移动操作
比如按照下面的方式进行定义

```
//一个可以复制的类
class Copyable{
    public:
```

```
    Copyable(const Copyable& other) = default;
    Copyable &operator=(consr Copyable& other) = default;
    //隐式地删除移动操作
    Copyable(const Copyable&& other) = default;
    Copyable &operator=(const Copyable&& other) = default;
};

//一个只能移动的类
class MoveOnly{
  public:
    MoveOnly(MoveOnly && other) = default;
    MoveOnly& operator=(MoveOnly&& other) = default;
    //复制操作被隐式删除
    MoveOnly(const MoveOnly&) = delete;
    MoveOnly& operator=(const MoveOnly&) = delete;
};

class NotCopyableOrMovable{
  public:
    NotCopyableOrMovable(const NotCopyableOrMovable&) = delete;
    NotCopyableOrMovable& operator=(const NotCopyableOrMovable&) = delete;

    NotCopyableOrMovable(const NotCopyableOrMovable&&) = delete;
    NotCopyableOrMovable& operator=(const NotCopyableOrMovable&&) = delete;
};
```

这些操作只有在如下条件才能被忽略
如果类没有私有部分，如 结构 或 仅接口基类，则可复制性/移动性 可以由 任何公共数据成员的 可复制性/移动性确定。
如果一个基类 显然不可复制 或可移动，那么 派生类 自然也不可复制 或隐式地 保留这些操作的 纯接口基类 不足以 使具体的子类 清晰。
注意，如果你显式声明 或删除 复制的构造函数 或赋值操作，则另一个复制操作 并不明显，必须 声明 或删除。 对于 移动操作也是如此。

Structs vs. Classes 结构体vs类
只有那些 被动存储数据的 对象 才是 结构体，否则就是 对象。 结构体的 所有域 必须是 共有的。
为了与 stl 保持一致，你可以 对无状态类型 使用 struct 而不是 class，例如 特征，模板元函数 和 一些仿函数。
注意，结构 和 类 中的 成员变量 具有 不同的命名规则。
为了表达意思的 准确性，建议 用结构体 而不是 键值对(pair) 或 元组(tuple)

Inheritance 继承
继承可以减小代码的体积，但也会增加理解实现的 难度，继承分为2类：
    接口继承 指 继承 纯抽象基类

实现继承， 非 接口继承

策略
所有继承都应该是 public。 如果你确实想要 私有继承，那么必须将 基类的实例 作为 成员。你可以 使用 final 来确保 该类不会被 继承。

```
//禁用继承
class Super final{

};
//禁用重写
class Super
{
  public:
    Supe();
    virtual void SomeMethod() final;
};
```

将protected 的使用 限制 在那些可能需要从 子类访问的 成员函数。注意 数据成员应该是私有的。

使用 override 或 (不太常见的) final 中的 一个 来显式 注解 虚函数 或 虚析构函数的 重写。 声明重写时 不要使用 virtual。理由：标记为 override 或 final 的函数 或 析构函数不是 基 的覆盖说明符用作文档； 如果没有说明符，则读者 必须检查 相关类的 所有祖先以确定 函数 或 析构函数 是否为 虚拟。

允许多重继承，但强烈反对 多重实现继承。

Operator Overloading 运算符重载
C++允许用户 重载 operator 内建 运算符，也可以定义 自己的运算符。

策略
不建议用户自定义运算符

Access Control 访问控制
策略
对于数据成员一定要设置私有的

Declaration Order 类的命名顺序
类的定义经常按照 从前往后 public，protected，private 的顺序，对其中的成员而言：
- 类型和类型别名
- 静态常量
- 工厂函数
- 构造器 和赋值运算符
- 析构函数
- 所有其他函数(静态和非静态成员函数，以及友元函数)

数据成员（ 静态和非静态）

Functions 函数
Inputs and Outputs 输入输出

函数有时通过 返回值输出，有时通过 参数输出， 为了可读性，推荐 使用 返回值输出。

使用 std::optional 来表示 可选的输入。不过这是 C++17 的特性。

避免定义 需要 const 引用参数 才能超过调用的 函数，因为 const 引用参数绑定到 临时对象。 相反，找到一种 方法 来消除 生命周期要求 (如 通过 复制参数)，或 通过 const 指针 和 文档 传递它 生命周期 和 非空要求。

排序 函数参数时，将所有 输入参数 放在 输出参数 之前。特别是，不要因为 新参数 而在函数 末尾添加新参数： 将新的 仅输入参数 放在 输出参数 之前。

Write Short Functions 写短函数
如果一个函数 超过40行，就考虑拆分。
短函数在 可扩展性 和 测试 都有很大的优势

Default parameters 默认参数
只要不是 虚函数，就可以 使用 默认参数，C++也规定了 默认参数 只能在 形参列表的 最后。通过 使用 默认参数，可以减少 要定义的 析构函数，方法，及 方法重载的数量。

```
//带默认参数的函数
void func(int n, float b=1.2, char c='@'){
    ...
}

//为所有参数传值
func(10, 3.5, '#');
//为n、b传值，相当于调用func(20, 9.8, '@')
func(20, 9.8);
//只为n传值，相当于调用func(30, 1.2, '@')
func(30);
```

Trailing Return Type Syntax 尾随的返回类型
C++允许使用 auto 来定义函数，并以尾随形式 返回值，通常用于 定义 匿名函数

```
auto foo(int x) -> int;
```

大部分情况下，auto 都可以自动推导，但某些情况下， 如 递归函数 就不行，必须用 std::function 来定义

Google的魔法

Ownership and Smart Pointers 拥有权和智能指针
所谓的 拥有权 指的是 对动态分配内存的 管理，拥有者 在使用完 动态分配对象 后就必须删除它。 有时 所有权 会被 共享，或转移，即便如此，最后一个 使用它的 对象 也要 负责 回收这些 空间。

智能指针 能确保 自动管理 动态分配对象，避免复杂的 垃圾回收过程，同时 避免 对象 复制所带来的 开销。

std::unique_ptr 是独占式 指针，在离开作用域后，指向的对象就会被删除。它不能被 复制，但是可以被 move

std::shared_ptr 是共享式 指针，可以被复制，只有 所有 指针被删除后，对象才会被删除。

需要注意的是，智能指针 也不是 普通指针的 完美替代
指针语义比 值语义 更复杂，尤其在API中：不仅要注意 所有权，还要注意 别名，生命周期，可变性 等问题。
转移所有权的 API 迫使 它们的 客户 使用 单一的 内存管理模型
使用 智能指针，资源的申请 释放不太明确
在某些情况下 (例如，循环引用)，具有共享所有权的 对象可能永远不会被删除

策略
如果需要动态分配，最好保留分配它的代码的 所有权。如果其他代码 需要访问该对象，建议传递复制，或者 传递 指针或引用 而不转移所有权。 首先使用 std::unique_ptr 来明确 所有权转移。例如

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

不用使用 std::auto_ptr，而是使用 std::unique_ptr

cpplint 使用脚本检测风格问题
cpplint.py是一个检测代码风格的工具。

```
pip install cpplint
```

Other C++ Features 其它C++特性
Rvalue Reference 右值引用
右值引用可以绑定到临时对象 (也就是不可取地址的 对象)， 用 && 表示
当 && 被用到模板参数上，由于 引用折叠 的原因，会产生问题，需要进行 完美转发。

我们可以使用 std::move 来将 左值对象 转化为 右值。 std::vector<std::string>， 使用 auto v2(std::move(v1)) 就可以避免 大量的 数据拷贝。

对于那些不可复制，但可以转移 的对象，只能使用 右值进行传递。可以在 类中 定义 移动构造器 和 移动赋值运算符。

使用 std::forward 来支持 完美转发。

可以使用右值来重载函数，比如 Foo && 和 const Foo&

Friends 友元
友元通常应该定义在 同一个文件中，这样读者 就不必在 另一个文件中 查找 类的 私有成员的 用途。友元的常见用途 是让 FooBuilder 成为 Foo 的 朋友， 让 UnitTest 类 成为 它测试的类的 朋友 可能会很有用。

友元不破坏类的 封装边界。在某些情况下，当你 只想让 其他类访问成员时，这比 将成员公开 要好。

大多数类应该通过 公有方法 和 其他类 交互。

Exceptions 异常
我们不使用异常。

虽然异常可以帮助更好地 诊断代码的问题（而不是 令人抓狂的 error)， 在测试框架中 非常有用。并且 很多其他语言也使用 异常。

但它会带来很多问题

> 当你 向一个现有函数 添加 throw 语句时，你必须检查 它的 所有传递调用者。例如，如果 f() 调用 g() 调用 h()，并且 h 抛出 f 捕获的异常，则 g 必须小心，它可能无法正确处理
>
> 更一般地说，异常 使程序的控制流 难以 通过 查看代码来评估
>
> 大部分情况下，不使用异常 也是可以的，有相应的 支持机制。
>
> 打开异常 会将数据添加到 生成的 每个 二进制文件中，增加 编译时间（可能略微增加）并可能增加 地址空间压力。

策略
不要使用异常。在新的项目 使用 异常是很好的 实践，但大多数老项目 都不使用异常。

noexcept
用于明确 一个函数 是否抛出异常。如果一个 标记为 noexcept 的函数 抛出了 异常，那么程序 就会因为 std::terminate 而终止。

使用noexcept

```
void func() noexcept;
```

或者使用常量表达式

```
void func() noexcept(常量表达式)
```

策略

如果异常被完全禁用 (即大多数 Google C++环境)，则 首选 无条件 noexcept。否则，请使用具有简单条件的 条件 noexcept 说明符，以仅在 函数 可能抛出的 少数情况下 评估false 的方式。

Run-Time Type Information (RTTI) 实时类型信息

不要使用RTTI。

RTTI允许对C++对象的类型在运行时查询，通过typeid或dynamic_cast实现的。

RTTI在一些单元测试中很有用，在多抽象类中也很有用：

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast<Derived*>(other);
  if (that == nullptr)
    return false;
  ...
}
```

策略

在单元测试中可以 很自由滴使用。如果要写 和之前类 不同的代码，可以用以下方法 来替代

虚拟方法，实现 运行时多态

考虑双重调度解决方案，例如访问者设计模式。这允许对象本身之外的 设施使用内置类型系统确定 类的类型。当程序的 逻辑保证基类的 给定实例 实际上 是特定派生类 的实例时，可以在 对象上 自由使用 dynamic_cast，通常在这种情况下 可以使用 static_cast 作为替代。

基于类型的 决策树 是很不明智的 写法

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

这样的代码 通常会在 类层次结构中 添加 额外的 子类时 中断，而且 当子类的 属性 发生变化时，很难找到并修改 所有 受影响的 代码段。

Casting 强制类型转换

使用 static_cast<A>(B) 或者 A{B},不要使用C中的转换 (A)(B).

原因在于，C中的 转换具有 模糊性，有时候 是 "conersion" ， 有时候是 "cast"

策略

使用括号表达式 的转换 (比如 int_64_t{x}) 是最安全的，因为如果存在 信息损失 (精度)，编译器就会报错

对于向上类型转换，可以使用 absl::implicit_cast， 比如从 Foo* 到 SuperClassOfFoo* 或者 从 Foo* 到 const Foo* 。 C++通常会 自动进行这类 转换

需要去除 const 属性，请使用 const_cast

使用 reinterpret_cast 将指针类型 与整数 和 其他指针类型 (包括 void*) 进行不安全的 转换。 仅当你知道 自己在做什么 并且 了解 混叠问题时 才使用这个选项。 另外，请考虑 替代的 absl::bit_cast。

使用 absl::bit_cast 来解释 使用 相同大小的 不同类型的值的 原始位，例如 将 double 的位 解释为 int64_t

==========

https://en.cppreference.com/w/cpp/language/copy_elision

==========

==========

==========