文件编号	W-7.3-0047	版本	V1.0	页次	1/78
文件名称	西安中诺 C&C++编码规范				

西安中诺 C&C++编码规范

编制/日期：赵少芳 2021-11-25

审核/日期：张子敬、陈盎、张德强 2021-11-29

批准/日期：柴玉东 2022-05-10

受控状态：■受控文件　　□非受控文件

文件编号	W-7.3-0047	版本	V1.0	页次	2/78
文件名称	西安中诺 C&C++编码规范				

修订履历表

版本	时间	更新描述	审核人	修订者
V0.1	2021-11-29	初版	刘秋格、陈盎、张德强、柴玉东、张子敬、罗红娟	赵少芳
V1.0	2022-05-10	增加 C&C++编码规约	刘秋格、陈盎、张德强、柴玉东、张子敬、罗红娟、高峰	赵少芳

目录

1. 目的 7
2. 说明 7
3. 适用范围 7
4. 术语和定义 7
5. 角色、职责与权限 7
6. 头文件 8
6.1. Self-contained 头文件 8
6.2. #define 保护 8
6.3. 前置声明 8
6.4. 内联函数 9
6.5. Includes 的路径及顺序 10
7. 作用域 11
7.1. 命名空间 11
7.2. 匿名命名空间 13
7.3. 非成员函数、静态成员函数和全局函数 13
7.4. 局部变量 14
7.5. 静态和全局变量 15
8. 类 16
8.1. 构造函数的职责 16
8.2. 隐式类型转换 17
8.3. 可拷贝类型和可移动类型 18
8.4. 结构体 VS. 类 19
8.5. 继承 20
8.6. 多重继承 21
8.7. 接口 21
8.8. 运算符重载 22
8.9. 存取控制 23
8.10. 声明顺序 23
9. 函数 23
9.1. 参数顺序 23
9.2. 编写简短函数 24
9.3. 引用参数 24
9.4. 函数重载 25
9.5. 缺省参数 25
9.6. 函数返回类型后置语法 26

10. 来自 Google 的奇技.....27
10.1. 所有权与智能指针.....27
10.2. Cpplint.....29
11. 其他 C++ 特性.....29
11.1. 右值引用.....29
11.2. 友元.....30
11.3. Exceptions.....30
11.4. 运行时类型识别.....31
11.5. 类型转换.....32
11.6. 流.....33
11.7. 前置自增和自减.....34
11.8. const 用法.....34
11.9. constexpr 用法.....35
11.10. 整型.....36
11.11. 64 位下的可移植性.....37
11.12. 预处理宏.....38
11.13. 0, nullptr/NULL.....39
11.14. sizeof.....39
11.15. auto.....40
11.16. 列表初始化.....41
11.17. Lambda 表达式.....43
11.18. 模板元编程.....45
11.19. Boost 库.....46
11.20. std::hash.....47
11.21. C++ 11.....48
11.22. 非标准扩展.....48
11.23. 别名.....49
12. 命名约定.....51
12.1. 通用命名规则.....51
12.2. 文件命名.....51
12.3. 类型命名.....52
12.4. 变量命名.....52
12.4.1 普通变量命名.....53
12.4.2 类数据成员.....53
12.4.3 结构体变量.....53

12.5. 常量命名 53
12.6. 函数命名 54
12.7. 命名空间命名 54
12.8. 枚举命名 54
12.9. 宏命名 54
13. 注释 55
13.1. 注释风格 55
13.2. 文件注释 55
13.2.1 法律公告和作者信息 55
13.2.2 文件内容 55
13.3. 类注释 55
13.4. 函数注释 56
13.4.1 函数声明 56
13.4.2 函数定义 57
13.5. 变量注释 57
13.5.1 类数据成员 57
13.5.2 全局变量 58
13.6. 实现注释 58
13.6.1 代码前注释 58
13.6.2 行注释 58
13.6.3 函数参数注释 59
13.6.4 不允许的行为 59
13.7. 标点，拼写和语法 60
13.8. TODO 注释 60
13.9. 弃用注释 61
14. 格式 61
14.1. 行长度 61
14.2. 非 ASCII 字符 61
14.3. 空格还是制表位 62
14.4. 函数声明与定义 62
14.5. Lambda 表达式 64
14.6. 函数调用 64
14.7. 列表初始化格式 65
14.8. 条件语句 66

14.9. 循环和开关选择语句.....68
14.10. 指针和引用表达式.....69
14.11. 布尔表达式.....70
14.12. 函数返回值.....70
14.13. 变量及数组初始化.....71
14.14. 预处理指令.....71
14.15. 类格式.....72
14.16. 构造函数初始值列表.....73
14.17. 命名空间格式化.....73
14.18. 水平留白.....74
14.19. 垂直留白.....76
15. 规则特例.....76
15.1. 现有不合规范的代码.....76
15.2. Windows 代码.....77
16. 编码规约.....77
16.1. 内存，指针及资源.....77
16.2. 中断处理.....78
16.3. 变量，宏.....78
16.4. 数组，结构.....78
16.5. 类型，表达式.....78
16.6. 函数，类.....78
16.7. 多线程.....78
17. 总结.....78

文件编号	W-7.3-0047	版本	V1.0	页次	7/78
文件名称	西安中诺 C&C++编码规范				

1. 目的

无论你是个人开发还是团队，一个良好的代码规范，能够在项目当中发挥举足轻重的作用；它不仅能使你们的开发更加高效，而且还会减少 BUG 产生的概率，增强代码可维护性及稳定性。

为了统一公司 C 语言编码规范，提高公司 C 语言编码质量，制定本指南作为基线标准。

这份文档是中诺 C 语言编程风格规范的完整定义。当且仅当一个 C 语言源文件符合此文档中的规则， 我们才认为它符合中诺的 C 语言编程风格。

- 通过统一的表现形式，提高代码的可读性。
- 辅助和提高源代码的可移植性。
- 辅助和提高源代码的可维护性。
- 提供一个源代码质量保证应用的标准。

注意：与其它的编程风格指南一样，本篇所讨论的不仅仅是编码格式美不美观的问题， 同时也讨论一些约定及编码标准。本指南并非 C 语言教程。

2. 说明

本规范指南中的编码规约是必须强制执行的内容，其他内容作为基线标准。

3. 适用范围

西安中诺软件部编写代码的所有研发工程师，适用于西安中诺所有项目。

注意：本文档中的示例代码并不作为规范。也就是说，虽然示例代码是遵循西安中诺编程风格，但并不意味着这是展现这些代码的唯一方式。

4. 术语和定义

缩写和术语	定义
class	可表示一个普通类，枚举类，接口或是 annotation 类型(@interface)
member	可表示嵌套类，字段，方法，或者构造方法，即除初始块和注释外的所有内容
comment	只用来指代实现的注释(implementation comments)。我们不使用“documentation comments”一词，而是用 Javadoc。

5. 角色、职责与权限

角色	职责与权限
SPM	与客户沟通或自定义项目中使用的编码规则 周知软件研发团队，协助研发负责人对研发进行培训 抽查项目/产品中编码规范
SQA	抽查项目/产品中编码规范执行的状况
研发组长	组织参与项目的研发工程师学习项目/产品编码规范 通过 Review 等手段保证团队开发出符合规范的代码
研发工程师	项目/产品编码规范，并按照编码规范开发出高质量的代码

6. 头文件

通常每一个 C++源文件(.cc,.C,.cpp,.cxx) 文件都有一个对应的 .h 文件。也有一些例外，如单元测试代码和仅包含 main() 函数的较小的 .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。

下面的规则可规避使用头文件时的各种陷阱。

6.1. Self-contained 头文件

规则：头文件应该是独立的(Self-contained，解释：单独编译(Compile on their own))，并以 .h 为后缀。可被其它文件包含(#include)的非头文件应以 .inc 为后缀。

所有头文件都是独立的。换言之，用户和重构工具不需要为特定场合而包含额外的头文件。详言之，一个头须包含它所需要的所有其它头文件，并符合第 6.2 节之规定。

规则：模板和内联函数的声明及定义应放在同一个头文件中，凡是用到模板或内联函数的源文件必须包含定义它们的头文件，否则 Build 时可能会产生链接错误。不要把这些定义放到分离的-inl.h 文件里。

例外情况：如果某函数模板为所有相关模板参数显式实例化，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源文件里。

有极少数情况下，一些可包含文件并不是 self-contained 的，此类往往包含在一些不寻常的位置，例如在另一个文件中间。它们没有第 6.2 节规定的防多重包含保护，也没有包含一些先决条件。这些文件要用.inc 为扩展名。请谨慎使用这种做法，尽可能使用 Self-contained 的头文件。

6.2. #define 保护

规则：所有头文件都应使用**#define** 来防止被多重包含，推荐格式：*<PROJECT>_<PATH>_<FILE>_H_*。

为保证唯一性，头文件的命名应基于所在项目源代码树的全路径。例如，项目 foo 中的头文件 foo/src/bar/baz.h 可按如下方式保护：

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

6.3. 前置声明

规则：尽量避免使用前置声明，需要时，#include 所需的头文件即可。

定义：

「前置声明」(Forward declaration)是指类、结构体、函数或模板的纯粹声明，没伴随着其定义。

优点：

- 能够节省编译时间，因为#includes 会迫使编译器展开更多的文件，处理更多的输入；
- 能够节省不必要的重新编译的时间，#includes 会使代码因为头文件中无关的改动而被重新编译多次。

缺点：

- 会隐藏依赖关系，头文件改动时，代码会跳过必要的重新编译过程；
- 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、增加自带默认参数的模板形参或者迁移到新的命名空间；
- 前置声明来自命名空间 std:: 的 symbols 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。用前置声明代替#include 可能会改变代码的含义：

```
// 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*)；

- 前置声明了多个来自头文件的 symbols 时，就会比仅有一行的 #include 冗长；
- 仅仅为了能前置声明而构造代码（比如用指针成员代替对象成员）会使代码变得更慢更复杂。

结论：

- 尽量避免对定义在其他项目中的实体进行前置声明；
- 使用在头文件中声明的函数时，应采用#include 方式；
- 使用类模板时，优先使用 #include 来包含头文件。

关于什么时候包含头文件，参见第 9.5 节。

6.4. 内联函数

规则：函数体较小时，比如代码少于 10 行，可将其定义为内联函数。

定义：

当函数被声明为内联函数之后，编译器会将其内联展开，而不是按通常的函数调用机制进行调用。

优点：

内联函数的函数体较小时，目标代码的执行效率会更高。存取器(Accessors)、修改器(Mutators)、函数体小的函数和性能关键的函数，尽量使用内联方式。

缺点：

滥用内联会导致程序变得更慢。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器函数通常会减少代码大小，但内联一个相当大的函数将戏剧性的增加代码大小。现代处理器由于更好的利用了指令缓存，小巧的代码往往执行更快。

结论：

不要内联超过 10 行的函数。析构函数通常都会有隐含的代码，所以函数体会比表面看起来要长，内联析构函数时一定要注意这一点。

尽量不要内联那些包含循环或 switch 语句的函数，除非这些循环或 switch 语句在大多数情况下都不会被执行。

有些函数即使声明为内联的也不一定会被编译器内联，这点很重要。比如虚函数和递归函数就不会被正常内联。通常，递归函数不应该声明成内联函数。虚函数内联的主要原因是想把它的函数体放在类定义内，或许是为了图个方便，抑或是当作文档描述其行为，比如精短的存取函数。

6.5. Includes 的路径及顺序

规则：标准的头文件包含顺序为：相关头文件，C 库，C++库，其他库，本项目内的.*h*。使用标准头文件包含顺序能增强可读性，避免隐藏依赖。

规则：项目内头文件应按照项目源代码目录树结构排列，尽量避免使用特殊的快捷目录. （当前目录）或.. （上级目录）。

例如，xxx-project/src/base/logging.h 应按如下方式包含头文件：

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

又如，dir/foo.cc 或者 dir/foo_test.cc 的主要作用是实现或测试 dir2/foo2.h 的功能，包含头文件的次序如下：

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

采用这种优先的顺序，在 dir2/foo2.h 遗漏某些必要的库时，dir/foo.cc 或 dir/foo_test.cc 的编译会立刻中止。因此这一条规则能确保使用这些文件的人员首先看到编译中止的消息。

规则：按字母顺序对头文件进行排序。

规则：应包含定义了当前文件依赖的所有 symbols 定义的头文件，前置声明的情况除外。

例如，某个文件中用到了 bar.h 中的某个 symbol，即使该文件已包含的头文件 foo.h 中已经包含了 bar.h，也需要在该文件中包含 bar.h，除非 foo.h 有明确说明它会自动提供 bar.h 中的 symbol。

又如，xxx-project/src/foo/internal/fooserver.cc 的包含次序如下:

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

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

#include <hash_map>
#include <vector>

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

规则：系统/平台特定代码要用条件包含(conditional includes)，此类代码应放到其它 includes 语句之后。同时，此类代码应简洁、独立。

例如：

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

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

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

7. 作用域

7.1. 命名空间

命名空间使用规则：

- 命名空间的命名规则请参照 12.7。
- 命名空间结束的地方应该有注释，如下面的例子所示。
- 一般情况下，命名空间应包含源代码中除 include 语句、gflags 声明/定义以及在其它命名空间中定义的类的前置声明以外的所有内容。

```
// In the .h file
namespace mynamespace {
```

```
// All declarations are within the namespace scope.
// Notice the lack of indentation.
class MyClass {
 public:
  ...
  void Foo();
};

} // namespace mynamespace

// In the .cc file
namespace mynamespace {

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

} // namespace mynamespace
```

更复杂的.cc 文件可能包含更多的细节，比如引用其他命名空间的类等。

```
#include "a.h"

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

namespace a {

using ::foo::bar;

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

} // namespace a
```

- 不要在命名空间 std 内声明任何东西，包括标准库中类的前置声明。在 std 命名空间声明实体会导致不确定的问题，比如不可移植。
- 不要使用 using 语句来导出一个命名空间中所有可用的名字。

```
// Forbidden -- This pollutes the namespace.
using namespace foo;
```

- 除了明确标注只在内部使用的命名空间外，不要在命名空间的范围内使用命名空间别名，因为导入到头文件中命名空间的任何内容都将成为该文件输出的公共 API 的一部分。

```
// 减少对.cc 文件一些常用名称的访问
namespace baz = ::foo::bar::baz;
// 减少对.h 文件一些常用名称的访问
namespace librarian {
namespace impl { // Internal, not part of the API.
namespace sidetable = ::pipeline_diagnostics::sidetable;
} // namespace impl

inline void my_inline_function() {
  // namespace alias local to a function (or method).
  namespace baz = ::foo::bar::baz;
  ...
}
} // namespace librarian
```

- 不要使用内联(inline)命名空间。

7.2. 匿名命名空间

规则：.cc 文件中无需外部引用的定义，应放到匿名命名空间中或者声明为 static，但不要在.h 文件中这样用。

定义：

匿名命名空间内的声明具有内链接的性质，函数和变量的内链接可通过将其声明为 static 来实现。具有内链接性质的实体都不能在文件外访问。如果不同文件声明相同名称的具有内链接性质的实体，那么这两个实体是完全独立的。

规则：像命名空间那样格式化匿名命名空间，匿名空间结束时用注释 // namespace 标识.

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

7.3. 非成员函数、静态成员函数和全局函数

规则：非成员函数应放在命名空间中，尽量避免使用全局函数。函数应该用命名空间来分组，而不是用类来分组。

优点：

某些情况下，非成员函数和静态成员函数会非常有用，将非成员函数放在命名空间内可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，尤其是当它们需要访问外部资源或具有严重的外部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成静态成员，或是非成员函数。非成员函数不应依赖于外部变量，应尽量置于某个命名空间内。不共享任何静态数据的函数应放入到命名空间中，而不是作为静态成员函数用类来封装它们。

例如：在 myproject/foo_bar.h 的头文件中可这样写

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

但不要像下面这样

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

规则：如果必须定义非成员函数，并且只在.cc 文件中使用它，应使用内链接限定其作用域。

7.4. 局部变量

规则：函数中的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。

规则：虽然 C++允许在函数的任何位置声明变量，不过还是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。这会使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。

规则：应在变量声明时对其初始化，而不是先声明再赋值。

例如：

```
int i;
i = f();      //坏 -- 初始化和声明分离
int j = g();  // 好 -- 初始化时声明

std::vector<int> v;
v.push_back(1);  // 用花括号初始化更好
v.push_back(2);
std::vector<int> v = {1, 2};  // 好 -- v 一开始就初始化
```

在 if, while 和 for 语句中所需的变量通常也是在这些语句中声明，因此这限制了这些变量的作用域。如下：

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

特别强调：如果变量是一个对象，每次进入作用域都要调用其构造函数，每次退出作用域都要调用其析构函数。

```
// 低效的实现:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // My ctor and dtor get called 1000000 times each.
  f.DoSomething(i);
}
```

在循环作用域外面声明这类变量要高效的多：

```
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);
```

7.5. 静态和全局变量

规则：禁止使用 class 类型的静态或全局变量，这会导致难以发现的 bug 和不确定的构造和析构函数调用顺序。不过 constexpr 变量除外，因为这种方式不涉及动态初始化或析构。

规则：静态生存周期的对象（包括：全局变量、静态变量、静态类成员变量和函数静态变量）都必须是原生数据类型 (POD : Plain Old Data)，POD 类型包括：int、char、float、指针、POD 类型的数组和结构体。

规则：由于类的构造函数和静态变量的初始化的顺序在 C++定义不完整，甚至会随着 Build 而改变，这会导致难以发现的 Bug。因此禁止使用函数返回值来初始化非局部变量，除非该函数（比如：getenv()或者 getpid()）不依赖于任何其它全局变量。例外，函数作用域内的静态 POD 变量可用函数返回值来初始化，因为这种情形的初始化定义是明确的，只有在指令执行到变量的声明那里才会发生函数调用。

规则：程序从 main()返回或调用 exit()时，全局变量和静态变量会被析构。析构顺序与构造顺序相反，C++中析构顺序定义不完整。例如，程序结束时某个静态变量已经被析构了，但其它线程的代码还在运行，并且很有可能会访问已被释放的静态变量；再例如，一个静态 string 类型变量的析构函数可能会在引用了它的那个变量被析构前被调用。因此，请使用 quick_exit()来终止程序，因为这个函数不会执行任何析构，也不会执行 atexit()所绑定的任何处理程序。

规则：只使用 POD 类型的静态变量，用 C 数组代替 std::vector，用 const char []代替 string。

规则：如果确实需要一个 class 类型的静态或全局变量，可在 main()函数或 pthread_once()内初始化一个指针且永不回收。注意只能用原始指针，勿使用智能指针，因为智能指针的析构函数会涉及到析构顺序问题。

8. 类

类是 C++中代码的基本单元，本章列举了在写一个类时的主要注意事项。

8.1. 构造函数的职责

规则：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：

在构造函数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。
- 经过构造函数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：

- 如果在构造函数内调用了自身的虚函数，这种调用不会被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。
- 除了程序崩溃（因为并不是一个始终合适的方法）或者使用异常（因为已经被 禁用了），构造函数很难上报错误。
- 如果执行失败，会得到一个初始化失败的对象，这个对象有可能进入不正常的状态，必须使用 bool IsValid() 或类似这样的机制才能检查出来，然而这是一个十分容易被疏忽的方法。
- 构造函数的地址无法被获取，因此，由构造函数完成的工作无法以简单的方式交给其它线程。

结论

不允许在构造函数中调用虚函数。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。否则，可使用 Init()方法或工厂函数。如果一个对象没有其他的能影响调用公共方法的状态，则应避免对其使用 Init()方法（此类形式的半构造对象有时无法正确工作）。

8.2. 隐式类型转换

规则：不要定义隐式类型转换。对于转换运算符和单参数构造函数，请使用 explicit 关键字修饰。

定义：

隐式类型转换允许一个某种类型（源类型）的对象被用于需要另一种类型（目的类型）的位置，例如，将一个 int 类型参数传递给需要 double 类型参数的函数。

除了语言所定义的隐式类型转换，还可通过在类定义中添加合适的成员来定义所需的类型转换。

explicit 关键字可以用于构造函数或（在 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给出时才会进行类型转换，例如使用 cast。这不仅作用于隐式类型转换，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);
```

此时下面的代码是不允许的：

```
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：

- 当目的类型非常明确时，可避免显式地写出类型名，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，目的类型并不符合期望，甚至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。
- 单参数构造函数有可能会被无意地用作隐式类型转换。

- 如果单参数构造函数没有加上 explicit 关键字，使用时无法判断这一函数究竟是要作为隐式类型转换，还是当初忘了加上 explicit 标记。
- 没有明确的方法可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函数都应当用 explicit 进行标记。例外情况是，拷贝和移动构造函数不应当被标记为 explicit，因为它们并不执行类型转换。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。

不能以一个参数进行调用的构造函数不应当加上 explicit。含有 std::initializer_list 作为参数的构造函数也应当省略 explicit，以便支持拷贝初始化（例如 MyType m = {1, 2};）。

8.3. 可拷贝类型和可移动类型

规则：仅当明确需要时，才使用拷贝/移动。否则，隐式产生的拷贝和移动函数应禁用。

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对象的值，或在赋值时被赋予相同类型的另一对象，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函数与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对象的例子。对于用户定义的类型，移动操作一般是通过移动构造函数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者返回，这使得 API 更简单，更安全也更通用。与传指针和引用不同，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，也就没必要在协议中予以明确。同时也可防止客户端与实现在非局部交互，使得它们更容易被理解，维护以及被编译器优化。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。并且它们允许在类型组合中具有更多的灵活性。

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()或者 Swap()，更容易定义，因为它们能通过编译器产生，无论是隐式的还是通过 = default。这种方式很简洁，也保证所有数据成员都会被复制。拷贝与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是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这样的优化它们也更加合适。

移动操作允许隐式、高效地将源数据转移出右值对象。这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迷惑，也显得荒谬而不合理。singleton 类型(Registerer)、与特定的作用域相关的类型 (Cleanup)、或与其它对象实体紧耦合的类型 (Mutex) 从逻辑上来说都不应该提供拷贝操作。为支持多态的基类提供拷贝/赋值操作是有害的，因为在使用它们时会造成对象切割。默认的或者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并且难以诊断出的错误。

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。这会让人迷惑，尤其是对那些所用语言约定或强制要求传递引用的程序员来说更是如此。同时，这从一定程度上会鼓励过度拷贝，从而导致性能上的问题。

结论：

如果对于用户来说这个拷贝移动操作不是一眼就能看出来的，并且也没有不好的影响那就可以提供。如果定义了拷贝或者移动构造函数，那就要定义相应的赋值操作，反之亦然。如果类型可拷贝，那么就不要定义移动操作除非它们的效率远高于拷贝操作。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两个操作。

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的且/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类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()方法和一个 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 或其他手段禁用之。

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

8.4. 结构体 VS. 类

规则：仅当只有数据成员时使用 struct，其它一概使用 class。

在 C++ 中 struct 和 class 关键字几乎含义一样。

规则：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。除了构造函数、析构函数、Initialize()、Reset()、Validate()等类似的用于设定数据成员的函数外，不能提供其它功能的函数。

规则：如果需要更多的函数功能，class 更适合。如果拿不准，就用 class。为了和 STL 保持一致，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。

8.5. 继承

规则：使用 Composition 常常比用继承更合理。如果使用继承的话，定义为 public 继承。

定义：

当子类继承基类时，子类包含父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：

继承通过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在父类和子类间之间，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，应该把基类的实例作为成员对象。不要过度使用继承。Composition 常常更合适一些。尽量做到只在"is-a"的情况下使用继承：如果 Bar 的确 “is-a” Foo，Bar 才能继承 Foo。

必要的话，析构函数应声明为 virtual。如果你的类有虚函数，则析构函数也应该为虚函数。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 关键字显式地进行标记。较早（早于 C++11）的代码可能会使用 virtual 关键字作为不得已的选项。因此，在声明重载时，请使用 override，final 或 virtual 的其中之一进行标记。标记为 override 或 final 的析构函数如果不是对基类虚函数的重载的话，编译会报错，这有助于捕获常见的错误。这些标记起到了文档的作用，因为如果省略这些关键字，代码阅读者不得不检查所有父类，以判断该函数是否是虚函数。

8.6. 多重继承

规则：真正需要用到多重继承的情况少之又少。只在以下情况才允许多重继承：最多只有一个基类是非抽象类；其它基类都是以 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别开来。

优点：

相比单继承，多重实现继承可以复用更多的代码。

缺点：

真正需要用到多重继承的情况少之又少。有时多重实现继承看上去是不错的解决方案，但通常也可以找到一个更明确，更清晰的不同解决方案。

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。为确保它们是纯接口，这些类必须以 `Interface` 为后缀。

8.7. 接口

规则：接口是指满足特定条件的类，这些类最好以 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：

- 只包含纯虚函数和静态函数。
- 没有非静态数据成员。
- 没有定义任何构造函数。如果有，也不能带有参数，并且必须为 protected。
- 如果它是一个子类，也只能从满足上述条件并以 `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。为确保接口类的所有实现可被正确销毁，必须为其声明虚析构函数（作为上述第 1 条规则的特例，析构函数不能是纯虚函数）。具体细节可参考 Stroustrup 的 *The C++ Programming Language*，3rd edition 第 9.4 节。

优点：

以 Interface 为后缀可以提醒其他人不要为该接口类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暴露给用户。

结论：

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

8.8. 运算符重载

规则：除少数特定环境外，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自定义的类型。operator 关键字还允许使用 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，也使得自定义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习惯的（例如 ==，<，=，<=），遵循这些约定可以让自定义类型更易读，也能更好地和需要这些重载运算符的函数库进行交互操作。

对于创建自定义类型的对象来说，自定义字面量是一种非常简洁的标记。

缺点

- 要提供正确，一致，不出现异常行为的运算符重载需要花费不少精力，一旦达不到这些要求的话，会导致令人迷惑的 Bug。
- 过度使用运算符重载可能会使代码难以理解，尤其是在重载的运算符的语义与通常的约定不符合时。
- 函数重载有多少弊端，运算符重载就至少有多少。
- 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。
- 查找重载运算符的调用点需要的可不仅仅是像 grep 那样的程序，这需要能够理解 C++语法的搜索工具。
- 如果重载运算符的参数写错，得到的可能是一个完全不同的重载而非编译错误。例如：foo < bar 执行的是一个行为，而 &foo < &bar 执行的就是完全不同的另一个行为了。
- 重载某些运算符本身就是有害的。例如，重载一元运算符&会导致同样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。重载诸如&&，||和逗号运算符会导致运算顺序和内建运算的顺序不一致。
- 运算符通常定义在类的外部，所以对于同一运算，可能出现不同的文件引入了不同的定义的风险。如果两种定义都链接到同一二进制文件，就会导致未定义的行为，有可能发生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。

只对自定义类型重载运算符。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够使用定义的运算符，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。如果定义了一个运算符，请将其相关且有意义的运算符都进行定义，并且保证这些定义的语义是一致的。例如，如果重载了<，那么请将所有的比较运算符都进行重载，并且保证对于同一组参数，<和>不会同时返回 true。

建议将不修改内容的二元运算符定义为非成员函数。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的参数而不会作用于左侧。这时会出现 a < b 能够通过编译而 b < a 却不能的情况，这是很让人迷惑的。

不要为了避免重载操作符而走极端。比如说，应当定义==、=、和<<而不是 Equals()，CopyFrom()和 PrintTo()。反过来，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，如果类型没有自然顺序，而需要将它们存入 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&&,||,逗号运算符或一元运算符&。不要重载 operator""，也就是说，不要引入用户定义字面量。

8.9. 存取控制

规则：所有数据成员应声明为 private，除非是 static const 类型成员。

8.10. 声明顺序

规则：相似的声明须放在一起，public 部分放在最前面。

说明：

类定义一般应以 public:开始，接下来是 protected:，最后是 private:。

在各个部分中，建议将类似的声明放在一起，并且建议采用如下顺序：类型（包括 typedef，using 和嵌套的结构体与类），常量，工厂函数，构造函数，赋值运算符，析构函数，其它函数，数据成员。

不要将大段的函数定义内联在类定义中。只有那些普通的，或非性能关键且短小的函数可以内联在函数定义中。

9. 函数

9.1. 参数顺序

规则：函数的参数顺序为：输入类型参数，输出类型参数。

C/C++中的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/输出参数则一般为非 const 指针。在排列参数顺序时，将所有的输入参

数置于输出参数之前。特别要注意，在加入新参数时不要因为它们是新参数就置于参数列表最后，而要按照前述的规则，即将新的输入参数也置于输出参数之前。

这并非一个硬性规定。输入/输出参数（通常是类或结构体）让这个问题变得复杂。并且，有时候为了其他函数保持一致，可能不得不有所变通。

9.2. 编写简短函数

规则：函数尽可能简短、功能单一。

规则：长函数有时是合理的，因此不硬性限制函数的长度。如果函数超过 40 行，可以考虑在不影响程序结构的前提下对其进行分割。

对于运行良好的长函数，一旦对其修改，有可能出现新问题，甚至导致难以发现的 bug。所以尽量使函数简短，以便于他人阅读和修改代码。

在处理代码时，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。

9.3. 引用参数

规则：所有按引用方式传递的参数必须加上 const。

定义：

在 C 语言中，如果函数需要修改变量的值，参数必须为指针，如 `int foo(int *pval)`。在 C++中，函数还可以声明引用类型的参数： `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之类的应用是必需的。同时也更明确地不接受空指针。

缺点：

容易引起误解，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);
```

有时候，在输入形参中用 `const T*` 指针比 `const T&` 更明智。比如：

- 需要传递空指针的场景。
- 函数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&。若用 const T*则说明输入另有处理。所以若要使用 const T*，则应给出相应的理由，否则会让使用者感到迷惑。

9.4. 函数重载

规则：函数重载必须清晰易懂，尽可能做到不让使用者猜测调用的重载函数到底是哪一种。这一规则也适用于构造函数。

定义：

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。而对模板化代码重载也可能是有必要的，这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，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，可以尝试在函数名里加上参数信息。例如，用 AppendString() 和 AppendInt() 等，而不是一口气重载多个 Append()。如果重载函数的目的是为了支持不同数量的同一类型参数，则优先考虑使用 std::vector 以便使用者可以用列表初始化指定参数。

9.5. 缺省参数

规则：仅在非虚函数中使用缺省参数，且必须保证缺省参数的值始终一致。缺省参数与函数重载遵循同样的规则。一般情况下建议使用函数重载。

优点：

有些函数一般情况下使用默认参数，但有时又需要使用非默认值的参数。缺省参数为这种情形提供了便利，使程序员不需要为了较少的例外情况编写大量的函数。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，因此所有不该使用函数重载的理由也都适用于缺省参数。

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都声明同样的缺省参数。

缺省参数在每个调用点都要进行重新求值，这会造成生成的代码迅速膨胀。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，导致函数签名与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，在这种情况下也不允许使用缺省函数。例如，不要写像 `void f(int n = counter++);` 这样的代码。

在其他情况下，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。

9.6. 函数返回类型后置语法

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

定义：

C++ 现在允许两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。例如:

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

C++11 引入了这一新的形式，可在函数名前使用 `auto` 关键字，在参数列表之后后置返回类型。例如:

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

后置返回类型为函数作用域。对于像 `int` 这样简单的类型，两种写法没有区别。但对于复杂的情况，例如类作用域中的类型声明或者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 表达式返回值的唯一方式。某些情况下，编译器可以自动推导出 Lambda 表达式的返回类型，但并不是在所有的情况下都能实现。即使编译器能够自动推导，显式地指定返回类型也更加明了。

有时在函数参数列表之后指定返回类型，会更简单，也更易读，尤其是在返回类型依赖于模板参数时。例如：

```
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);
```

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，因此对大部分人来说会比较陌生。

在已有的代码中一般会有大量的函数声明，很难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的语法或者新旧混用。在这种情况下，只使用一种版本相对来说是更规整的形式。

结论:

在大部分情况下，应当继续使用以往的函数声明写法，即将返回类型置于函数名前。只有在必要的时候(如 Lambda 表达式)， 或是使用后置语法能够简化书写并且提高易读性的时候，才使用返回类型后置语法。但是后一种情况一般来说是很少见的，大部分时候都出现在相当复杂的模板代码中，而多数情况下不鼓励写这样复杂的模板代码。

10. 来自 Google 的奇技

Google 用了很多自己实现的技巧/工具使 C++代码更加健壮，他们使用 C++的方式可能和你在其它地方见到的有所不同。

10.1. 所有权与智能指针

规则：动态分配的对象最好有单一且固定的所有者，并通过智能指针传递所有权。

定义:

所有权是一种登记/管理动态内存和其它资源的技术。动态分配对象的所有者是一个对象或函数，后者负责确保前者无用时就自动将其销毁。所有权有时可以共享，此时就由最后一个所有者来负责销毁它。甚至也可以不用共享，在代码中直接把所有权传递给其它对象。

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。一些智能指针类型被用来做所有权登记工作的自动化，确保执行销毁任务。std::unique_ptr 是 C++11 新推出的一种智能指针类型，用来表示动态分配的对象的独一无二的所有权；当 std::unique_ptr 离开作用域时，对象就会被销毁。std::unique_ptr 不能被复制，但可以把它移动给新所有者。std::shared_ptr 同样表示动态分配对象的所有权，但可以被共享，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，对象也会被销毁。

优点：

- 如果没有清晰的所有权逻辑，不可能管理好动态分配的内存。
- 传递对象的所有权，开销比复制要小。
- 传递所有权也比”借用”指针或引用要简单，毕竟它省去了一起协调对象生命周期的工作。
- 如果所有权逻辑清楚，文档不紊乱的话，那么用智能指针可以改善可读性
- 通过智能指针可以不用手动完成所有权的登记工作，简化了代码，也免去了大量的类错误。
- 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：

- 必须用指针（不管是智能的还是原生的）来表示和传递所有权。指针语义比值语义更复杂，特别是在 API 里，不仅要关心所有权，还要顾及别名，生命周期，可变性以及其它大大小小的问题。
- 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，就会迫使客户端不得不使用单一的内存管理模型。
- 如果使用智能指针，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，它毕竟刚刚推出，容易迷惑一些程序员。
- 如果原本的所有权设计已经够完善了，那么若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，开销可能相当大。
- 某些极端情况下（例如循环引用），所有权被共享的对象永远不会被销毁。
- 智能指针并不能够完全代替原生指针。

结论：

如果必须使用动态分配，则尽量将所有权保持在分配者手中。如果其它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。最好使用 std::unique_ptr 来明确所有权传递，例如：

```
std::unique_ptr<Foo> FooFactory();
```

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

如果没有充足的理由，则不要使用所有权共享。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> ）时候，才能这么做。如果确实要使用共享所有权，建议于使用 std::shared_ptr。

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

10.2. Cpplint

规则：使用 cpplint.py 检查代码风格错误。

Cpplint.py 是一个用来分析源文件，能检查出多种风格错误的工具。它不并完美，甚至还会漏报和误报，但它仍然是一个非常有用的工具。在行尾加//NOLINT，或在上一行加//NOLINTNEXTLINE，可以忽略报错。

大部分项目可能需要定制 cpplint.py，如果项目没有提供，你可以单独下载 cpplint.py。

11. 其他 C++ 特性

11.1. 右值引用

规则：只在定义移动(move)构造函数与移动(move)赋值操作时使用右值引用，或者在 perfect forwarding 时使用它。

定义：

右值引用是一种只能绑定到临时对象的一种引用，其语法与传统的引用语法相似。

例如，

```
void f(string&& s);
```

声明了一个参数为字符串的右值引用函数。

优点：

- 用于定义移动构造函数（参数是类的右值引用的构造函数）可移动一个值而非拷贝它。例如，如果 v1 是一个 std::vector<string>，则 auto v2(std::move(v1))将很可能不再进行大量的数据复制而只是简单地进行指针操作，在某些情况下这将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成为可能，无论其参数是否是临时对象都能正常工作（这就是所谓的 perfect forwarding）。
- 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放入容器的类型很有用。
- 要高效率地使用某些标准库类型，例如 std::unique_ptr，std::move 是必需的。

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。类似引用崩溃，移动构造函数的自动推导这样的规则都是很复杂的。

结论：

只在定义移动构造函数与移动赋值操作时使用右值引用（参见拷贝和移动类型），并且可以和 std::forward 一起使用从而实现完美转发。此外，可以使用 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。

11.2. 友元

规则：应该合理地使用友元类或友元函数。

规则：通常友元应该定义在同一文件内。

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成待测类的友元会很方便。

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，大多数类都只应通过其提供的公有成员进行互操作。

11.3. Exceptions

规则：尽量不使用 C++ Exceptions。

优点：

- Exceptions 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，而不用管那些含糊且容易出错的错误代码。
- 很多现代程序设计语言都支持 Exceptions。引入 Exceptions 使得 C++ 与 Python，Java 以及其它类似于 C++ 的语言更一脉相承。
- 有些第三方 C++ 库依赖 Exceptions，禁用 Exceptions 就不方便使用这些库了。
- Exceptions 是处理构造函数失败的唯一途径，虽然可以用工厂函数或 Init() 方法代替 Exceptions，但是前者要求在栈中分配内存，后者会导致刚创建的实例处于 ”无效“ 状态。
- 在测试框架里很好用。

缺点：

- 在现有函数中添加 throw 语句时，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，要么只能让异常向上传递，最终中断掉整个程序。举例，f() 调用 g()，g() 又调用 h()，且 h 抛出的异常被 f 捕获。当心 g，否则可能没有妥善地做好清理工作。
- 更常见的情况，Exceptions 会彻底扰乱程序的执行流程并难以判断，函数也许会在意料不到的地方返回。或许会加一大堆何时何处处理 Exceptions 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，为了避免让使用者去理解整个调用表，异常安全必须隔绝从持续状态写到 “提交” 状态的逻辑。
- 启用 Exceptions 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。
- 使用 Exceptions 带来的好处，会鼓励开发者去捕捉不合时宜的 Exceptions，或恢复不安全的 Exceptions。比如，用户的输入不符合格式要求时，也用不着抛 Exceptions。一般都会用规则指南来记录这些限制。

结论：

从表面上看来，使用 Exceptions 利大于弊，尤其是在新项目中。但是对于现有代码，引入 Exceptions 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 Exceptions 的代码整合时会比较麻烦，而且迁移过程比较慢，也容易出错。Exceptions 的有效替代方案，如错误代码，断言等，不一定会造成严重负担。

并不是基于哲学或道德层面反对使用 Exceptions，而是在实践的基础上。

这个禁令也适用于 C++11 中增加的 Exceptions 相关的特性，例如：noexcept，std::exception_ptr，和 std::nested_exception。

11.4. 运行时类型识别

规则：避免使用运行时类型识别（RTTI）。

定义：

RTTI 允许程序员在运行时识别 C++ 对象的类型。它通过使用 typeid 或者 dynamic_cast 来识别对象。

优点：

标准的 RTTI 替代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所期望的，甚至是不可取的，尤其是在一个已经广泛使用的或者成熟的代码中。

RTTI 在某些单元测试中非常有用。比如进行工厂类测试时，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 会使代码难以维护。它使得基于类型的判断树或者 switch 语句散布在代码各处。如果以后要进行修改，就必须检查它们。

结论：

RTTI 有合理的用途但容易被滥用，因此在使用时务必注意。在单元测试中可以使用 RTTI，但是在其他代码中请尽量避免。尤其是在新代码中，使用 RTTI 前务必三思。如果代码需要根据不同的对象类型执行不同的行为的话，请考虑用以下的两种替代方案之一：

- 虚函数可以根据子类类型的不同而执行不同代码。这相当于把工作交给了对象本身去处理。
- 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(Double-dispatch)的方案，例如使用访问者设计模式。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，那么就可以自由使用 dynamic_cast。在这种情况下，通常会使用 static_cast 作为一种替代方案。

基于类型的判断树说明代码已经偏离正轨了。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，会很难找到并修改所有受影响的代码块。

不要去手工实现一个类似 RTTI 的方案。反对 RTTI 的理由同样适用于这些方案，比如包含类型标签的类继承体系。而且，这些方案会掩盖当初的真实意图。

11.5. 类型转换

规则：使用 C++ 类型转换（如 static_cast<float>(double_value) ），或者括号来初始化算术类型的转换（如 int64 y = int64{1} << 42 ）时，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，有时是强制转换（如 (int)3.5），有时是类型转换（如 (int)"hello"）。花括号（“{}”）初始化和 C++ cast 可以有效的避免这个问题。另外，C++ 的类型转换在查找时更容易。

缺点：

C++风格的转换语法有些冗长和繁琐的。

结论：

不要使用 C 风格类型转换。在需要类型转换的时候，使用 C++ 风格。

- 使用括号来初始化算术类型的转化（如 int64{x}），这是最安全的方法，因为如果转化可能导致信息丢失，那么代码将不会被编译。这个语法也很简洁。
- 当某个类指针需要明确的向上转换为父类指针或从父类指针转换到子类指针时，用 static_cast 替代 C 风格的值转换。在后一种情况，需要确保对象实际上是子类的实例。
- 用 const_cast 去掉 const 限定符。
- 指针类型与整型或其它指针类型进行互相转换时，用 reinterpret_cast。

11.6. 流

规则：在合适的情况下，用简单的方式使用流。

定义：

流是 C++中标准的 I/O 抽象，如标准头 <iostream>所示例。

优点：

流运算符(<<, >>)提供了易学、可移植，可复用及可扩展的 API 来格式化 I/O，相反，printf 甚至不支持 string，也不支持用户定义的类型，而且也很难移植。此外，使用 printf 时却需要在一些功能稍微有差别的版本中选择，并使用大量转化指示符。

通过 std::cin，std::cout，std::cerr，和 std::clog，流可对 I/O 控制台提供非常好的支持。虽然 C 的 API 也可以做到这些，但是需要手动缓冲输入。

缺点：

- 流格式可以通过改变流的状态来配置。由于这样的改变是持续的，因此代码行为可能会受到之前流的影响，除非涉及的其他代码每次都恢复到已知的状态。用户代码不仅可以修改内在状态，还可以通过注册系统增加新的状态变量和行为。
- 由于以上问题，包括流代码的写法以及运算符重载（有可能会选择的与预期不同的重载），因此很难精确控制流输出。
- 采用一连串的<< 运算符来建立输出会影响代码国际化，因为这把单词顺序编写到代码中。并且流对本地化的支持也是有缺陷的。
- 流的 API 比较复杂，必须要有经验才能有效使用。
- 对于编译器来说处理多个<<的重载需要花费很大代价。当在大型代码库中这样使用时，需要占用词法分析和语义分析差不多 20%的时间。

结论：

对当前任务来说如果流是最佳工具，才能使用它，否则不要使用流。通常情况下，I/O 接口是点对点、本地的、可读的，它针对其他开发者，而不是终端用户。

对于面向外部用户或者处理不信任数据的 I/O 接口，应避免使用流。相反，可以用合适的模板库来处理一些诸如国际化，本地化和安全加强的问题。

11.7. 前置自增和自减

规则：对于迭代器和其它模板对象，应使用前缀形式的自增(++i)，自减运算符。

定义：

使用自增（++i 或 i++）或自减（--i 或 i--）运算时，如果表达式的值在运算后未使用，需要确定到底是使用前置还是后置的自增/减运算。

优点：

不考虑返回值的话，前置自增（++i）通常要比后置自增（i++）效率更高。因为后置自增（或自减）需要对表达式的值 i 进行一次拷贝。如果 i 是迭代器或其他非数值类型，拷贝的代价会比较大。

缺点：

在 C 开发中，当表达式的值未被使用时，传统的做法是使用后置自增，特别是在 for 循环中。

结论：

对简单数值（非对象），两种都无所谓。对迭代器和模板类型，使用前置自增（自减）。

11.8. const 用法

规则：强烈建议在任何可能的情况下都要使用 const。此外有时用 C++11 的 constexpr 更好。

定义：

在声明的变量或参数前加上关键字 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：

如何使用变量比较容易理解。编译器可以更好地进行类型检测，也能生成更好的代码。程序员对编写正确的代码更加自信，因为所调用的函数被限定了能或不能修改变量值。即使是在无锁的多线程编程中，也容易知道什么样的函数是安全的。

缺点：

const 是入侵性的：如果向一个函数传入 const 变量，函数原型声明中也必须有对应 const 参数（否则变量需要 const_cast 类型转换），在调用库函数时显得尤其麻烦。

结论：

const 变量，数据成员，函数和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：

- 如果一个函数确保不会修改传入的引用或指针类型参数，那么相应的函数参数应声明为常量引用(const T&)或者指向 const 的指针 (const T*) 。
- 尽可能将方法声明为 const 。Accessors 函数应该总是 const 类型。其它不会修改任何数据成员、未调用非 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成 const 。
- 如果数据成员在对象构造之后不再发生变化，可将其定义为 const 。

关键字 mutable 可以使用，但是在多线程中是不安全的，使用时首先要考虑线程安全。

11.9. constexpr 用法

在 C++11 里，用 constexpr 来定义真正的常量，或实现常量初始化。

定义：

变量可以被声明成 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成 constexpr ，以用来定义 constexpr 变量。

优点：

constexpr 允许用浮点表达式定义常量，不用再依赖字面量。也可以定义用户自定义类型常量；甚至也可以定义函数调用所返回的常量。

缺点：

若过早把变量优化成 constexpr 变量，将来又要把它改为常规变量时，会相当麻烦。

结论:

constexpr 特性使 C++ 在接口上实现了真正常量机制。用 constexpr 来定义常量以支持常量的函数。避免复杂的函数定义，以使其能够与 constexpr 一起使用。千万不要想靠 constexpr 来强制代码「内联」。

11.10. 整型

C++内建整型中，最好仅使用 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。如果变量可能不小于 2^31 (2GiB)，就用 64 位变量比如 int64_t。此外要留意，即使值并不会超出 int 所能够表示的范围，在计算过程中也可能会溢出。所以拿不准时，干脆用更大的类型。

定义：

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

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的不同而不同。

结论：

<stdint.h> 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 和 ptrdiff_t。

如果已知整数不会太大，可以使用 int ，如循环计数。在类似的情况下使用原生类型 int 。可以认为 int 至少为 32 位，但不要认为它会多于 32 位。如果需要 64 位整型，用 int64_t 或 uint64_t。

对于大整数，使用 int64_t。

不要使用 uint32_t 等无符号整型，除非是在表示一个位组而不是一个数值，或是需要定义二进制补码溢出。尤其是不要为了指出数值永不会为负，而使用无符号类型。相反，你应该使用断言来保护数据。

如果代码涉及容器返回的大小（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，总有意想不到的后果。

关于无符号整数:

一些教科书，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，在 C 语言中，这一优点被由其导致的 bug 所淹没。看看下面的例子：

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

上述循环永远不会退出！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合变量和无符号变量时。主要是 C 的类型提升机制会致使无符号类型的行为出乎意料。

因此，使用断言来指出变量为非负数，而不是使用无符号型！

11.11. 64 位下的可移植性

规则：代码应该对 64 位和 32 位系统友好。处理打印，比较，结构体对齐时应切记。

- 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，而且标准中也有所遗漏，所以有时不得不自己定义一个丑陋的版本（头文件 inttypes.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	使用	备注

void *(或其它指针类型)	%lx	%p	
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 %td

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) 来处理。

- 记住 sizeof(void *) != sizeof(int)。如果需要一个指针大小的整数要用 intptr_t 。
- 要非常小心地对待结构体对齐，尤其是要持久化到磁盘上的结构体。在 64 位系统中，任何含有 int64_t/uint64_t 成员的类/结构体，缺省都以 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，需要确保两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 中可使用__attribute__((packed))，MSVC 则提供了 #pragma pack() 和 __declspec(align())。
- 创建 64 位常量时使用 LL 或 ULL 作为后缀，如：

```
int64_t my_value = 0x123456789LL;
uint64_t my_mask = 3ULL << 48;
```

11.12. 预处理宏

规则：使用宏时要非常谨慎，尤其是在头文件中，尽量以内联函数，枚举和常量代替之。宏的名称须用大写。命名具有项目特定前缀的宏。不要使用宏来定义 C++API。

宏意味着你和编译器看到的代码是不同的。这可能会导致异常行为，尤其因为宏具有全局作用域。

规则：由宏引入的问题在定义 C++API 部分时尤其严重，对于公共的 API 而言更是如此。当开发人员使用这个接口不正确时，编译器的每一个错误信息都需要解释宏是如何形成这个接口的，而重构和分析工具更新这个接口的难度会更大。因此，明确禁止以这种方式来使用宏。

例如，像下面这样使用就需要避免：

```
class WOMBAT_TYPE(Foo) {
  // ...

 public:
  EXPAND_PUBLIC_WOMBAT_API(Foo)

  EXPAND_WOMBAT_COMPARISONS(Foo, ==, <)
};
```

C++中，宏不像在 C 中那么必不可少。以往需用宏展开的性能关键的代码，现在可以用内联函数替代。用宏表示常量可被 const 变量代替。用宏 “缩写” 长变量名可被引用代替。

宏可以做一些其他技术无法实现的事情，在一些代码库（尤其是底层库中）可以看到宏的某些特性，并且一些特性(如 stringifying，连接等等)不是所有语言通用的。但在使用前，仔细考虑一下是否可以不使用宏达到同样的目的。

下面给出的用法模式可以避免使用宏带来的问题。如果使用宏，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，使用后要立即 #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，不然至少要附上文档说明其行为。
- 不要用 ## 处理函数，类和变量的名字。

最好不要从头文件里面导出宏（例如，在一个头文件里定义了宏，但没有在文件结束前对它#undef），如果从头文件里导出宏，那么它必须要有全局唯一名称。要实现这一点，它必须使用组成项目命名空间的名称前缀来命名。

11.13. 0, nullptr/NULL

规则：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，这一点是毫无争议的。对于指针（地址值），到底是用 0，NULL 还是 nullptr。C++11 项目用 nullptr，C++03 项目则用 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）用 '\0'，不仅类型正确而且可读性好。

11.14. sizeof

规则：尽可能用 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。用 sizeof(*type*) 处理不涉及任何变量的代码，比如处理来自外部或内部的数据格式，这时用变量就不合适了。

```
Struct data;
memset(&data, 0, sizeof(data));
memset(&data, 0, sizeof(Struct));
if (raw_size < sizeof(int)) {
  LOG(ERROR) << "compressed record not big enough for count: " << raw_size;
  return false;
}
```

11.15. auto

规则：如果类型对于可读性没有帮助，可以用 auto 绕过烦琐的、显而易见或者不重要的类型名，只要有助于可读性就可继续使用显式的类型声明。

优点：

- C++ 类型名有时比较长，特别是涉及模板或命名空间的时候。
- 在单个声明或者小段代码内重复 C++类型名称，对可读性没有帮助
- 有时候通过初始化表达式的类型来指定类型更安全，因为这会避免意外复制或者类型转化的可能性。

缺点：

类型够明显时，特别是初始化变量时，代码才会够一目了然。但以下就不一样了：

```
auto foo = x.add_foo();
auto i = y.Find(key);
```

如果 y 的类型不是很清楚，或者是在很多行之外声明的，那可能不容易看出结果类型。

必须会区分 auto 和 const auto &的不同之处，否则会复制错东西。

如果在接口里用 auto ，比如声明头文件里的一个常量，那么可能仅仅一时修改其值，也会导致类型变化。

结论：

auto 可以在增加可读性的情况下允许使用，特别是如下所述。不要用初始化列表来初始化 auto 类型变量。

下面情况允许或者鼓励使用 auto

- （鼓励）对于迭代器和其他 long/cluttery 类型名称，特别是在上下文可以很清楚地看出类型时候（调用 find, begin, 或者 end 为实例）

- （允许）在局部上下文（同一个表达式或者几行代码内）可以很明显看出类型时，指针的初始化或者调用 new 的智能指针通常都归结于这类。这正如在一个基于范围的循环中 auto 类型所作的那样。
- （鼓励）当使用基于范围的循环对 map 迭代时（因为通常假定正确的类型 std::pair<KeyType, ValueType>然而实际上是 std::pair<const KeyType, ValueType>）。与本地的 key 和 value 别名.first 和 .second 配对相当有益（常用 const-ref）

```
for (const auto& item : some_map) {
  const KeyType& key = item.first;
  const ValType& value = item.second;
  // 剩下的循环可以参考 key and value,
  // 避免了可能看到的问题类型
  // 在迭代中常见的额外复制
}
```

11.16. 列表初始化

规则：可以用列表初始化。

早在 C++ 03 里，聚合类型（aggregate types）就已经可以使用列表初始化，比如数组和不自带构造函数的结构体：

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

C++11 中，该特性得到进一步的推广，任何对象类型都可以使用列表初始化。如下：

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

//不考虑细节上的微妙差别，大致上相同。
//可以任选其一。

std::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。
std::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。
std::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(std::vector<int> v) {}
TestFunction2({1, 2, 3});
```

自定义类型也可以定义 std::initializer_list<T> 的构造函数和赋值运算符，以自动列表初始化：

```
class MyType {
 public:
  // std::initializer_list 专门接收 init 列表。
  //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，即使这些类型没有定义 std::initializer_list<T>的构造函数。

```
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(explicit)，就不能用 `= {}` 了
```

```
MyOtherType m{"b"};
```

千万不要直接使用列表初始化 auto 变量，下面一句，估计没人看得懂：

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

auto d = double{1.23}; //好，d 为 double
```

11.17. Lambda 表达式

规则：适当使用 Lambda 表达式。Lambda 离开当前作用域时，最好采用显式捕获。

定义：

Lambda 表达式是创建匿名函数对象的一种简易途径，常用于把函数当参数传递，例如：

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

此外 Lambda 表达式还允许通过名称或者默认捕获等方式来获取闭合作用域内变量。详细来讲，捕获方式分为值捕获和引用捕获。

```
int weight = 3;
int sum = 0;
//值捕获`weight`和引用捕获`sum`
std::for_each(v.begin(), v.end(), [weight, &sum](int x) {
  sum += weight * x;
});
```

默认捕获可以隐式捕获任何 Lambda 作用域内引用的变量，包括 this 指针，只要它的任何成员被使用。

```
const std::vector<int> lookup_table = ...;
std::vector<int> indices = ...;
//引用捕获`lookup_table`，按值排序`indices`
//查找`lookup_table`中的相关元素
std::sort(indices.begin(), indices.end(), [&](int a, int b) {
  return lookup_table[a] < lookup_table[b];
```

```
});
```

C++11 首次提出 Lambdas，还提供了一系列处理函数对象的工具，比如多态包装器（polymorphic wrapper）std::function。

优点：

- 传函数对象给 STL 算法，Lambdas 方式最为简易，可读性也好。
- 适当使用默认捕获可以减少冗余，并能突显重要的默认异常
- Lambdas，std::function 和 std::bind 可以搭配成通用回调机制；以有界函数作为参数的函数写起来也更容易了。

缺点：

- Lambdas 的变量捕获可能会造成悬浮指针 bugs，尤其在 lambda 离开当前作用域时。
- 默认值捕获不会阻止悬浮指针 bugs，所以可能被误导。值捕获不会引起深拷贝，因而它一般和引用捕获一样有生命周期问题。当通过值捕获'this'时，有一点特别让人迷惑，那就是'this'一般也是隐含使用。
- Lambdas 可能会失控；层层嵌套的匿名函数难以阅读。

结论：

- 慎用 Lambda 表达式，使用时按下面描述进行格式化。
- Lambda 离开当前作用域时，最好采用显式捕获。例如，不能写成下面这样：

```
{
  Foo foo;
  ...
  executor->Schedule([&] { Frobnicate(foo); })
  ...
}
//不好！事实上 lambda 引用了`foo`并且
//不仔细检查也很能很难发现`this`（如果`Frobnicate`是成员函数的话）。
//如果`foo`和封闭的对象本来就不存在的话，
//那么在函数返回后，再调用 lambda，可能就很糟。
```

要写成这样：

```
{
```

```
  Foo foo;
  ...
  executor->Schedule([&foo] { Frobnicate(foo); })
  ...
}
// 较好- 如果`Frobnicate`是成员函数，那么编译就会失败
// 并且很明显引用捕获`foo`是很危险的
```

- 只有当 Lambda 的生命周期明显比其它任何潜在的捕获方式更短，才可以使用默认引用捕获([&])
- 对于较短的 Lambda 表达式来说，使用默认值捕获([=])只是用来绑定几个变量的方法。一般来说，Lambda 表达式不要写太长或者太复杂，也不要采用默认值捕获。
- 匿名函数始终要简短，如果函数体超过了五行，就须使用命名 Lambda，或直接改为函数。
- 如需更好的可读性，就要显式写出 Lambda 的返回类型，就像 auto。

11.18. 模板元编程

规则：不要使用复杂的模板元编程。

定义：

C++模板实例化机制是图灵完备的，所以模板元编程可被用来实现任意编译时刻的类型域计算。

优点：

模板元编程能够实现非常灵活的类型安全的接口，并且性能极佳，一些常见的工具比如 Google Test，std::tuple，std::function 和 Boost.Spirit 都使用了模板元编程。

缺点：

模板元编程所使用的技巧对于 C++不是很熟练的人会比较晦涩、难懂。在复杂的地方使用模板的代码让人更不容易读懂，并且调试和维护起来都很麻烦。

模板元编程经常会导致非常不友好的编译时错误信息：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出现在错误信息里。

大量的使用模板元编程接口会让重构工具更难发挥用途。首先模板的代码会在很多上下文里面扩展开来，所以很难确认重构对所有这些展开的代码是否有用，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：

模板元编程有时候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。因此模板元编程最好只用在少量的基础组件或基础数据结构上，因为模板带来的额外的维护成本会被大量的使用分担掉。

在使用模板元编程或者其他复杂的模板技巧的时候，一定要再三考虑。需要考虑团队成员的平均水平是否能够读懂并且能够维护模板代码。或者一个非 C++ 程序员和一些只是在出错的时候偶尔看一下代码的人能够读懂这些错误信息或者能够跟踪函数的调用流程。如果使用递归的模板实例化、类型列表、元函数、表达式模板、依赖 SFINAE、sizeof 的 trick 来检查函数是否重载，这说明模板用的太多了，这些模板过于复杂，不推荐使用。

如果使用模板元编程，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漏。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，这样能提高接口的可读性。并且应该在这些使用模板的代码上添加详细的注释。注释应详细说明这些代码是怎么用的，这些模板生成出来的代码大概是什么样子的。还需要额外注意在用户错误使用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接口的一部分，所以代码必须调整到这些错误信息在用户看起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。

11.19. Boost 库

规则：只使用 Boost 中被认可的库。

定义：

Boost 库集 是一个广受欢迎，经过充分鉴定的，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，如型别特性，更完善的绑定器等。

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：

为了向阅读和维护代码的人员提供更好的可读性，只允许使用 Boost 一部分经认可的特性子集。目前允许使用以下库：

- Call Traits ： boost/call_traits.hpp
- Compressed Pair ： boost/compressed_pair.hpp
- The Boost Graph Library (BGL) ： boost/graph，除了序列化 (adj_list_serialize.hpp)、并行/分布式算法以及数据结构 (boost/graph/parallel/* 以及 boost/graph/distributed/*)。
- Property Map ： boost/property_map，除了并行/分布式属性映射 (boost/property_map/parallel/*)。
- Iterator ： boost/iterator

- 处理 Voronoi 图的构建的多边形库：boost/polygon/voronoi_builder.hpp、boost/polygon/voronoi_diagram.hpp 和 boost/polygon/voronoi_geometry_type.hpp
- Bimap ： boost/bimap
- Statistical Distributions and Functions ： boost/math/distributions
- Special Functions ： boost/math/special_functions
- Multi-index ： boost/multi_index
- Heap ： boost/heap
- The flat containers ： Container: boost/container/flat_map 和 boost/container/flat_set
- Intrusive ：boost/intrusive。
- The boost/sort library。
- Preprocessor ：boost/preprocessor。

以下库可以用，但由于其已经被 C++ 11 标准库取代，所以不再鼓励使用：

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

11.20. std::hash

规则：不要定义特殊的 std::hash

定义：

std::hash<T>是 C++11 哈希容器用来生成类型 T 的散列值的函数对象，除非用户明确指定了其他的哈希函数。例如 std::unordered_map<int, string>是一个哈希映射，用 std::hash<int>来生成它的散列值，然而 std::unordered_map<int, string, MyIntHash>却用 MyIntHash 来生成。

std::hash 被定义为适用所有整型，浮点，指针，以及 enum 类型，还包括一些标准库类型，例如 string 和 unique_ptr。也可以对自定义类型进行特制化处理从而使用这些功能。

优点：

由于不需要明确命名，std::hash 很容易使用，而且还可以简化代码。std::hash 特殊化是指如何对一个类型进行哈希的一套标准化方法。

缺点：

std::hash 很难特殊化(specialize)，这需要很多模板代码，更重要的是，在确保哈希输入的同时还要确保哈希算法的执行。由于哈希泛洪攻击的出现，低质量的哈希函数可能存在的安全漏洞，所以风险较高。

即使对于专家来说，由于不能对数据成员递归调用，复合类型的 std::hash 特殊化也是非常难以实现。高质量哈希算法可以保持大量的内部状态，并且减少 std::hash 返回 size_t 字节，这通常是计算最慢部分，最好不要多次执行。

正是由于这个问题，std::hash 不适用于 std::pair 或者 std::tuple，并且也不允许扩展语言来支持这些。

结论：

可以用 std::hash 支持非常规类型，但不要专门用于支持其他类型。如果需要 std::hash 不支持的密钥类型的哈希表，可以考虑暂时使用旧的哈希容器（例如 hash_map），哈希容器使用不同的默认哈希函数，这一点不受上述限制。

如果想使用标准哈希容器，就需要为密钥类型指定自定义哈希值，例如

```
std::unordered_map<MyKeyType, Value, MyKeyTypeHasher> my_map;
```

11.21. C++ 11

规则：使用 C++ 11 的库和语言扩展要谨慎，在项目用 C++ 11 特性前要考虑可移植性。

定义：

C++ 11 在语言和库上有大量的变更。

优点：

在 2014 年 8 月之前，C++ 11 是官方标准，被大多 C++ 编译器支持。它标准化了很多早就在用的 C++ 扩展，简化了不少操作，大大改善了性能和安全。

缺点：

C++ 11 相对于以前的标准，复杂性增加了。

和 Boost 库一样，有些 C++ 11 扩展提倡那些对可读性有害的编程实践——就像去除冗余检查（比如类型名）以帮助读者，或是鼓励模板元编程等等。有些扩展在功能上与原有机制冲突，容易招致困惑以及较高的迁移代价。

结论：

使用 C++ 11 特性要谨慎，以下特性最好不要用：

- 编译时 rational number <ratio>，因为它涉及一个重量级模板接口风格。
- <cfenv> 和 <fenv.h> 头文件，因为编译器尚不支持。
- 成员函数的引用限定符，例如 void X::Foo() & 或 void X::Foo() &&，因为要它们特性非常不清楚。

11.22. 非标准扩展

规则：除非特别指定，否则不可以使用 C++非标准扩展。

定义：

编译器支持不属于标准 C++的各种扩展，包括 GCC 的 `__attribute__`，内置函数 `__builtin_prefetch`，指定初始化 （如 `Foo f = {.field = 3}`），嵌入汇编`__COUNTER__`，`__PRETTY_FUNCTION__`，复合语句表达式(例如 `foo = ({ int x; Bar(&x); x })`，可变长度数组，`alloca()` 以及 `a?:b` 语法等。

优点：

- 非标准扩展可以提供标准 C++没有的一些有用特性。例如，有些人认为指定初始化比标准 C++特性（如构造函数）更易读。
- 编译器性能的重要导向可能只能用扩展来指定。

缺点：

- 非标准扩展不适用于所有编译器，使用非标准扩展会降低代码可移植性
- 即使非标准扩展在涉及的所有编译器都获得支持，但一般都未能良好定义。并且在编译器之间也可能存在一些细微的设置差异
- 非标准扩展使代码阅读者必须了解更多的语言特性

结论：

不要使用非标准扩展。可以使用非标准扩展实现的可移植包装类，这些可以通过指定的项目范围内可移植头文件来提供。

11.23. 别名

公共别名可让 API 使用者受益，应该有明确的书面定义。

定义：

有几个方法来创建其他实体的别名：

```
typedef Foo Bar;
using Bar = Foo;
using other_namespace::Foo;
```

像其他声明一样，头文件的别名声明属于头文件公共 API，除非在函数定义、类的私有定义或者明确标明的内部命名空间里。

优点：

- 别名可以简化长命名和复杂命名来改善可读性
- 别名可以通过在一个地方命名来减少 API 中类型命名重复，这便于以后更容易改变类型。

缺点：

- 当别名放在客户端代码可以引用的头文件时，会增加头文件 API 实体数目以及复杂性。
- 客户端容易相信公共别名的非预期的细节，这会造成变更困难。
- 不考虑对 API 的影响或者维护性的话，创建一个仅用于实现的公共别名是很有诱惑力的。
- 别名可引起命名冲突。
- 别名对陌生名字采用常用构造可以降低可读性。
- 类型别名可产生不明确的 API 契约，目前不确定是否别名与其别名的类型相同，有同样的 API 或者只能用指定的有限方法

结论：

如果仅仅是为了少写点代码，那么不要在公共 API 使用别名。

在定义公共别名时，应记录下新名字的目的，包括是否确保其始终与当前别名的类型一样，或者是否希望更有限的兼容性。这可以让用户知道是否可以将类型看做可替代的或者遵循更具体的规则，并且这有助于保留一定程度的自由来改变别名。

不要在公共 API 中使用命名空间别名。（另见命名空间）

例如，下面的例子说明了如何在客户端代码中使用别名。

```
namespace a {
//用来保存结构体成员值，DataPoint 可能从 Bar*变成某些内部类型
//客户端代码应该把它当成隐形指针
using DataPoint = foo::bar::Bar*;

//一次赋值，别名只是为了用户方便
using TimeSeries = std::unordered_set<DataPoint, std::hash<DataPoint>,
DataPointComparator>;
} // namespace a
```

而下面的别名没有体现预期的用途，并且其中一半都不适用客户端使用。

```
namespace a {
//不好：没有一个说明应该如何使用
using DataPoint = foo::bar::Bar*;
using std::unordered_set;  //差：只是为了局部方便
using std::hash;           //差：只是为了局部方便
typedef unordered_set<DataPoint, hash<DataPoint>, DataPointComparator> TimeSeries;
```

```
} // namespace a
```

当然，别名在局部使用还是不错的，包括函数定义，类的私有部分，明确注明的内部命名空间以及.cc 文件。

```
// 在一个.cc 文件里
using std::unordered_set;
```

12. 命名约定

命名管理是最重要的一致性规则。良好的命名风格能让使用者在无需查找类型声明的条件下快速地了解某个名字代表的含义：类型、变量、函数、常量、宏等。

命名规则具有一定随意性，但相比按个人喜好命名，一致性更重要，所以无论你认为它们是否重要，规则总归是规则。

12.1. 通用命名规则

规则：命名要有描述性，少用缩写。

规则：尽可能使用描述性的命名，不要担心空间占用问题，毕竟相比之下让代码易于理解更重要。不要用只有项目开发者能理解的缩写，也不要通过去掉几个字母来缩写单词。

```
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;       // 只有自己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
```

规则：一些特定的广为人知的缩写是允许的，例如用 i 表示迭代变量和用 T 表示模板参数。

12.2. 文件命名

规则：文件名全部小写，可以包含下划线（_）或连字符（-），这依赖于项目的约定。

可接受的文件命名示例：

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

- myusefulclass.cc
- myusefulclass_test.cc // _unittest 和_regtest 已弃用

规则：C++ 文件应以 .cc 结尾，头文件以 .h 结尾，专门 include 的文件则以 .inc 结尾。

规则：不要使用已经存在于 /usr/include 下的文件名，如 db.h。

规则：尽量让文件名更加明确。

例如：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。

规则：内联函数必须放在 .h 文件中。

12.3. 类型命名

规则：类型名的每个单词首字母均应大写，不要包含下划线。

例如：MyExcitingClass，MyExcitingEnum。

规则：所有类型命名（包括：类、结构体、类型别名、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。

例如：

```
//类和结构体
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

//类型定义
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// using 别名
using PropertiesMap = hash_map<UrlTableProperties *, string>;

// 枚举
enum UrlTableErrors { ...
```

12.4. 变量命名

规则：变量（包括函数参数）和数据成员名一律小写，单词之间用下划线连接。类的成员变量以下划线结尾，但结构体的就不用。

如：a_local_variable， a_struct_data_member，a_class_data_member_。

12.4.1 普通变量命名

举例：

```
string table_name;  // 好 - 用下划线
string tablename;   // 好 - 全小写

string tableName;   // 差 - 混合大小写
```

12.4.2 类数据成员

规则：不管是静态的还是非静态的，类数据成员都可以和普通变量一样，但要以下划线结尾。

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

12.4.3 结构体变量

规则：不管是静态的还是非静态的，结构体数据成员都可以和普通变量一样，不用像类那样以下划线结尾。

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

12.5. 常量命名

规则：声明为 constexpr 或 const 的变量，或在程序运行期间其值始终保持不变的量，命名时以 k 开头，大小写混合。

例如：

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

规则：所有静态存储类型的变量（例如静态变量或全局变量）都应当以此方式命名。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命名。

12.6. 函数命名

规则：常规函数使用大小写混合，accessors 和 mutators 函数则要与变量名匹配。

规则：一般来说，函数名的每个单词首字母大写（即 “驼峰命名法” 或 “帕斯卡命名法”），没有下划线。对于首字母缩写而成的单词，更倾向于将它们视作一个单词进行首字母大写（例如，写作 StartRpc() 而非 StartRPC())。

```
AddTableEntry()
DeleteUrl()
OpenFileOrDie()
```

12.7. 命名空间命名

规则：命名空间以小写字母命名。最高级命名空间的名字应取决于项目名称。注意避免嵌套命名空间和常见的顶级命名空间的名字发生冲突。

规则：命名空间名字尽量不要用缩写。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。

12.8. 枚举命名

规则：枚举命名应和常量或宏的命名一致，如：kEnumName 或是 ENUM_NAME。

规则：单独的枚举值应该采用常量的命名方式，宏方式的命名也可以接受。枚举名 UrlTableErrors（以及 AlternateUrlTableErrors）是类型，所以要用大小写混合的方式。

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

12.9. 宏命名

规则：尽量不要使用宏，如果要用，应这样命名： MY_MACRO_THAT_SCARES_SMALL_CHILDREN。

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

```
#define PI_ROUNDED 3.0
```

13. 注释

注释对保证代码可读性至关重要。下面的规则描述了如何注释以及在哪儿注释。当然也要记住：注释固然很重要，但最好的代码应当本身就是文档。有意义的类型名和变量名，要远胜过要用注释解释的含糊不清的名字。具体参考《西安中诺软件代码提交、注释、入库规范》进行。

13.1. 注释风格

规则：使用//或/* */，注释及注释风格应保持统一。

13.2. 文件注释

规则：每一个文件开头应加入版权公告。

规则：文件注释描述了该文件的内容。如果一个文件只声明、实现或测试了一个对象，并且这个对象已经在它的声明处进行了详细的注释，就没必要再加上文件注释。除此之外的其他文件都需要文件注释。

13.2.1 法律公告和作者信息

规则：每个文件都应该包含许可证引用，请为项目选择合适的许可证版本（比如：Apache 2.0，BSD，LGPL，GPL）。

规则：如果对原始作者的文件做了重大修改，请考虑删除原作者信息。

13.2.2 文件内容

规则：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致的说明，同时说明各概念之间的联系。一到两行的文件注释就足够了，对于每个概念的详细文档应当放在各个概念中，而不是文件注释中。

规则：不要在 .h 和 .cc 之间复制注释，这样的注释偏离了注释的实际意义。

13.3. 类注释

规则：每个类的定义都要附带一份注释，描述类的功能和用法。

```
// 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 {
  ...
};
```

类注释应当为读者理解如何使用与何时使用类提供足够的信息，同时应当提醒读者在正确使用此类时应当考虑的因素。如果类有任何同步前提，请用文档说明。如果该类的实例可被多线程访问，要特别在文档中说明多线程环境下如何使用相关的规则和常量。

规则：如果类的声明和定义分开了(例如分别放在了 .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在一起。

13.4. 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。

13.4.1 函数声明

规则：每个函数声明的前面都应加上注释来描述函数的功能和用途。函数的功能简单且明显时可省略注释。注释应使用叙述式而非指令式，因为它只是为了描述函数，而不是命令函数做什么。通常，注释不会描述函数如何工作，那是函数定义部分的事情。

函数声明处注释的内容:

- 函数的输入输出。
- 对类成员函数而言：函数调用期间对象是否需要保持引用参数，是否会释放这些参数。
- 函数是否分配了必须由调用者释放的内存。
- 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?

举例如下:

```
//返回表的迭代器，完成后，调用者需要删除这个迭代器。
//一旦 GargantuanTable 对象删掉后，对其创建的迭代器就不能使用。
//
//迭代器应该在表的开始位置初始化
//
//这个方法等价于：
//    Iterator* iter = table->NewIterator();
```

```
//    iter->Seek("");
//    return iter;
//如果准备立即在返回的迭代器中搜素其他位置，
//使用 NewIterator()更快，还能避免额外搜索
Iterator* GetIterator() const;
```

但也要避免罗罗嗦嗦，或者对显而易见的内容进行说明。下面的注释就没有必要加上 “否则返回 false”，因为已经暗含其中了：

```
//如果表不能容纳更多记录就返回真
bool IsTableFull();
```

注释构造/析构函数时，切记读代码的人知道构造/析构函数的功能，所以 “销毁这一对象” 这样的注释是没有意义的。应当注明的是构造函数对参数做了什么（例如，是否取得指针所有权）以及析构函数清理了什么。如果都是些无关紧要的内容，直接省掉注释。析构函数前没有注释是很正常的。

13.4.2 函数定义

规则：如果函数的实现过程中用到了某些很巧妙的方式，则应在函数定义处应当加上解释性的注释。例如，所使用的编程技巧，实现的大致步骤，或解释如此实现的理由。举个例子，可以说明为什么函数的前半部分要加锁而后半部分不需要。

规则：不要从.h 文件或其它地方的函数声明处直接复制注释。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。

13.5. 变量注释

通常变量名应足以很好说明变量用途。某些情况下，也需要额外的注释说明。

13.5.1 类数据成员

规则：每个类数据成员（也叫实例变量或成员变量）都应该用注释说明用途。如果非变量的参数（例如：特殊值、数据成员之间的关系或生命周期等）不能够用类型与名称明确推理出其用途，则应当加上注释。然而，如果变量类型与变量名已经足以描述一个变量(int num_events_;)，那么就不再需要加上注释。

特别地，如果变量可以接受 nullptr 或-1 等警戒值，须加以说明。比如:

```
private:
 //用于边界检查表访问，-1 表示还不知道这个表有多少条记录
 int num_total_entries_;
```

13.5.2 全局变量

规则：全局变量要注释说明含义及用途，以及作为全局变量的原因。

比如：

```
//在回归测试里面要运行的全部测试案例数目
const int kNumTestCases = 6;
```

13.6. 实现注释

规则：代码中巧妙的、晦涩的、有趣的或重要的地方都应加以注释。

13.6.1 代码前注释

规则：巧妙或复杂的代码段前要加注释。

比如：

```
//结果除以 2，考虑到加法运算对 x 的叠加
for (int i = 0; i < result->size(); i++) {
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

13.6.2 行注释

规则：比较隐晦的地方要在行尾加入注释，注释时应在行尾空两格。

比如：

```
//如果内存足够，数据部分也可以内存映射
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
  return;  //错误已经记录
```

注意，这里用了两段注释分别描述这段代码的作用，提示函数返回时错误已经被记入日志。

如果需要连续进行多行注释，可以使之对齐获得更好的可读性：

```
DoSomething();                    //此处注释保持对齐
DoSomethingElseThatIsLonger();  //代码和注释之间两个空格
{ //新起一行时，注释前要有空格
  //并且要和接下来的注释和代码对齐
  DoSomethingElse();  //通常在每行注释前两个空格
```

```
}
std::vector<string> list{
                    //列表注释描述下一个元素...
                    "First item",
                    // ..并且应该适当对齐
                    "Second item"};
DoSomething(); /* 对块末尾的注释，留一个空格就好 */
```

13.6.3 函数参数注释

规则：如果函数参数的意义不明显，考虑用下面的方式进行弥补：

- 如果参数是一个字面常量，并且这一常量在多处函数调用中被使用，用以推断它们一致，应当用一个常量名让这一约定变得更明显，并且保证这一约定不会被打破。
- 考虑更改函数的签名，让某个 bool 类型的参数变为 enum 类型，这样可以让这个参数的值能够表达其意义。
- 如果某个函数有多个配置选项，可以考虑定义一个类或结构体以保存所有的选项，并传入类或结构体的实例。这样的方法有许多优点，例如这样的选项可以在调用处用变量名引用，这样就能清晰地表明其意义。同时也减少了函数参数的数量，使得函数调用更易读也易写。除此之外，以这样的方式，如果使用其他的选项，就无需对调用点进行更改。
- 用命名变量代替大段而复杂的嵌套表达式。
- 万不得已时，才考虑在调用点用注释阐明参数的意义。

比如下面的示例的对比：

```
//这些参数是什么含义?
const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);
```

和

```
ProductOptions options;
options.set_precision_decimals(7);
options.set_use_cache(ProductOptions::kDontUseCache);
const DecimalNumber product =
    CalculateProduct(values, options, /*completion_callback=*/nullptr);
```

哪个更清晰一目了然。

13.6.4 不允许的行为

规则：不要描述显而易见的现象，永远不要用自然语言翻译作为代码注释。

规则：注释应当解释代码为什么要这么做和代码的目的，或者最好是让代码自文档化。

比较这样的注释：

```
// 寻找向量元素 <-- 差: 这太明显了!
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

和这样的注释：

```
//处理"element"除非它已经被处理了
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

自文档化的代码根本就不需要注释。上面例子中的注释对下面的代码来说就是毫无必要的：

```
if (!IsAlreadyProcessed(element)) {
  Process(element);
}
```

13.7. 标点，拼写和语法

规则：注意标点，拼写和语法，写得好的注释比差的要易读得多。

注释的通常写法是包含正确大小写和结尾句号的完整叙述性语句。大多数情况下，完整的句子比句子片段可读性更高。短一点的注释，比如代码行尾注释，可以随意点，但依然要注意风格的一致性。

13.8. TODO 注释

规则：对那些临时的，短期的解决方案，或已经够好但仍不完美的代码使用 TODO 注释。

规则：TODO 注释要使用全大写的字符串 TODO，在随后的圆括号里写上作者名字，邮件地址，bug ID，或其它身份标识和与这一 TODO 相关的 issue。主要目的是让添加注释的人（也是可以请求提供更多细节的人）可根据规范的 TODO 格式进行查找。

```
// TODO(kl@gmail。com):使用"*"连接运算符
// TODO(Zeke) change this to use relations。改为使用关系
// TODO(bug 12345):删除“最后访问者”特性
```

如果加 TODO 是为了在 “将来某一天做某事”，可以附上一个非常明确的时间 “Fix by November 2005”），或者一个明确的事项（“Remove this code when all clients can handle XML responses.”）。

13.9. 弃用注释

规则：通过弃用注释（DEPRECATED comments）以标记某接口点已弃用。

规则：可以写上包含全大写的 DEPRECATED 的注释，以标记某接口为弃用状态。注释可以放在接口声明前，或者同一行。

在 DEPRECATED 一词后，在括号中留下名字，邮箱地址以及其他身份标识。

弃用注释应当包涵简短而清晰的指引，以帮助其他人修复其调用点。在 C++ 中，可以将一个弃用函数改造成一个内联函数，这一函数将调用新的接口。

14. 格式

每个人都可能有自己的代码风格和格式，但如果一个项目中的所有人都遵循同一风格的话，这个项目就能更顺利地进行。每个人未必能同意下述的每一处格式规则，而且其中的不少规则需要一定时间的适应，但整个项目服从统一的编程风格是很重要的，只有这样才能让所有人轻松地阅读和理解代码。

14.1. 行长度

规则：每一行代码字符数不超过 80。

这条规则是有争议的，但很多已有代码都遵照这一规则，因此一致性更重要。

优点：

提倡该原则的人认为强迫他们调整编辑器窗口大小是很野蛮的行为。很多人同时并排开几个代码窗口，根本没有多余的空间拉伸窗口。大家都把窗口最大尺寸加以限定，并且 80 列宽是传统标准。

缺点：

反对该原则的人则认为更宽的代码行更易阅读。80 列的限制是上个世纪 60 年代的大型机的古板缺陷，现代设备具有更宽的显示屏，可以很轻松地显示更多代码。

结论：

每行代码 80 个字符是最大值。

例外：

如果无法在不降低易读性的条件下进行断行，那么注释行可以超过 80 个字符，这样可以方便复制粘贴。例如，带有命令示例或 URL 的行可以超过 80 个字符。

原生的字符串可能包含超过 80 个字符的内容，除了测试代码，这样的文字应当放在文件头部。

包含长路径的#include 语句可以超出 80 列。

头文件保护可无视该原则。

14.2. 非 ASCII 字符

规则：尽量不使用非 ASCII 字符，不得不用时必须使用 UTF-8 编码。

即使是英文，也不应将用户界面的文本硬编码到源代码中，因此非 ASCII 字符应当很少被用到。特殊情况下可以适当包含此类字符。例如，代码分析外部数据文件时，可以适当硬编码数据文件中作为分隔符的非 ASCII 字符串，更常见的是（不需要本地化的）单元测试代码可能包含非 ASCII 字符串。此类情况下，应使用 UTF-8 编码，因为很多工具都可以理解和处理 UTF-8 编码。

十六进制编码也可以，能增强可读性的情况下尤其鼓励，比如"\xEF\xBB\xBF"，或者更简洁地写作"\uFEFF"，在 Unicode 中是零宽度，无间断的间隔符号，如果不用十六进制直接放在 UTF-8 格式的源文件中，是看不到的。

使用 u8 前缀把带\uXXXX 转义序列的字符串字面值编码成 UTF-8。不要用于本身就带 UTF-8 字符的字符串字面值上，因为如果编译器不把源代码识别成 UTF-8，输出就会出错。

别用 C++11 的 char16_t 和 char32_t，它们和 UTF-8 文本没有关系， wchar_t 同理，除非你写的代码要调用 Windows API，后者广泛使用了 wchar_t。

14.3. 空格还是制表位

规则：只使用空格，每次缩进 2 个空格。

使用空格缩进，不要在代码中使用制表符，应该设置编辑器将制表符转为空格。

14.4. 函数声明与定义

规则：返回类型和函数名应在同一行，参数也尽量放在同一行，如果放不下就对形参分行，分行方式与函数调用一致。

函数看上去像这样：

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

如果同一行文本太多，放不下所有参数：

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1,
Type par_name2, Type par_name3) {
  DoSomething();
  ...
}
```

甚至连第一个参数都放不下：

```
ReturnType
LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
```

```
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}
```

注意以下几点：

- 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&);
};
```

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*/) {}
```

```
//差 - 如果将来有人要实现，很难猜出变量的作用。
void Circle::Rotate(double) {}
```

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();
```

14.5. Lambda 表达式

规则：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。

```
int x = 0;
auto x_plus_n = [&x](int n) -> int { return x + n; }
```

短 lambda 就写得和内联函数一样。

```
std::set<int> blacklist = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(), [&blacklist](int i) {
                return blacklist.find(i) != blacklist.end();
            }),
            digits.end());
```

14.6. 函数调用

规则：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个空格。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。

函数调用遵循如下形式：

```
bool result = DoSomething(argument1, argument2, argument3);
```

如果同一行放不下，可断为多行，后面每一行都和第一个实参对齐，左圆括号后和右圆括号前不要留空格：

```
bool result = DoSomething(averyveryveryverylongargument1,
```

```
                                argument2, argument3);
```

参数也可以放在次行，缩进四个空格：

```
if (...) {
  ...
  ...
  if (...) {
    bool result = DoSomething(
        argument1, argument2,  //  4 空格缩进
        argument3, argument4);
    ...
  }
```

把多个参数放在同一行以减少函数调用所需的行数，除非影响到可读性。有人认为把每个参数都独立成行，不仅更好读，而且方便编辑参数。不过，比起所谓的参数编辑，应更看重可读性，且后者大多数问题可以用以下技巧解决：

如果一些参数本身就是略复杂的表达式，且降低了可读性，那么可以直接创建临时变量描述该表达式，并传递给函数：

```
int my_heuristic = scores[x] * y + bases[x];
bool result = DoSomething(my_heuristic, x, y, z);
```

或者放着不管，补充上注释：

```
bool result = DoSomething(scores[x] * y + bases[x],  // Score heuristic.
                          x, y, z);
```

如果某参数独立成行，对可读性更有帮助的话，那也可以如此做。参数的格式处理应当以可读性而非其他作为最重要的原则。

此外，如果一系列参数本身就有一定的结构，并且对可读性也很重要，可以酌情地按其结构来决定参数格式：

```
//通过 3x3 矩阵转换 widget.
my_widget.Transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

14.7. 列表初始化格式

规则：怎么格式化函数调用，就怎么格式化列表初始化。

如果列表初始化伴随着名字，比如类型或变量名，格式化时将名字视作函数调用名，{} 视作函数调用的括号。如果没有名字，就视作名字长度为零。

```
//一行列表初始化示范.
return {foo, bar};
functioncall({foo, bar});
std::pair<int, int> p{foo, bar};

//当不得不断行时.
SomeFunction(
    {"assume a zero-length name before {"},// 假设在 { 前有长度为零的名字.
    some_other_function_parameter);
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {"},// 假设在 { 前有长度为零的名字.
    SomeOtherType{
        "Very long string requiring the surrounding breaks.", // 非常长的字符串, 前后都需要断
行.
        some, other values},
    SomeOtherType{"Slightly shorter string",// 稍短的字符串.
                  some, other, values}};
SomeType variable{
    "This is too long to fit all in one line"};//字符串过长, 因此无法放在同一行.
MyType m = {  //注意了, 可以在 { 前断行.
    superlongvariablename1,
    superlongvariablename2,
    {short, interior, list},
    {interiorwrappinglist,
     interiorwrappinglist2}};
```

14.8. 条件语句

规则：倾向于不在圆括号内使用空格，关键字 if 和 else 另起一行。

对基本条件语句有两种可以接受的格式。一种在圆括号和条件之间有空格，另一种没有。

最常见的是没有空格的格式。哪一种都可以，最重要的是保持一致。如果是在修改一个文件，参考当前已有格式。如果是写新的代码，参考目录下或项目中其它文件。

```
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 {
```

```
  ...
}
```

如果你更喜欢在圆括号内部加空格:

```
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.
  ...
}
```

注意所有情况下 if 和左圆括号间都有个空格。右圆括号和左大括号之间也要有个空格:

```
if(condition) {   //差 - IF 后面没空格.
if (condition){   //差 - { 前面没空格.
if(condition){    //更差.
```

```
if (condition) {  //好 - IF 和 { 都与空格紧邻.
```

如果能增强可读性，简短的条件语句允许写在同一行。只有当语句简单并且没有使用 else 子句时使用:

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

如果语句有 else 分支则不允许:

```
//不允许 - 当有 ELSE 分支时 IF 块却写在同一行
if (x) DoThis();
else DoThat();
```

通常，单行语句不需要使用大括号，如果喜欢用也没问题。复杂的条件或循环语句用大括号可读性会更好。也有一些项目要求 if 必须总是使用大括号:

```
if (condition)
  DoSomething();  // 2 空格缩进.

if (condition) {
  DoSomething();  // 2 空格缩进.
}
```

但如果语句中某个 if-else 分支使用了大括号的话，其它分支也必须使用:

```
//不可以 - IF 有大括号 ELSE 却没有.
if (condition) {
```

```
  foo;
} else
  bar;

//不可以 - ELSE 有大括号 IF 却没有.
if (condition)
  foo;
else {
  bar;
}
```

```
//只要其中一个分支用了大括号，两个分支都要用上大括号.
if (condition) {
  foo;
} else {
  bar;
}
```

14.9. 循环和开关选择语句

规则：switch 语句可使用括号分段，以表明 cases 之间不是连在一起的。在单语句循环里，括号可用可不用。空循环体应使用括号或 continue。

switch 语句中的 case 块可以使用大括号也可以不用。如果用的话，要按照下文所述的方法。

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：

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

```
}
```

在单语句循环里，括号可用可不用：

```
for (int i = 0; i < kSomeNumber; ++i)
  printf("I love you\n");

for (int i = 0; i < kSomeNumber; ++i) {
  printf("I take it back\n");
}
```

空循环体应使用一对空括号{}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) {}  //好 - 空循环体.
while (condition) continue;  //好 - contunue 表明没有逻辑.
```

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

14.10. 指针和引用表达式

规则：.或->前后不要有空格，指针/地址操作符（*，&）之后不能有空格。

下面是指针和引用表达式的正确使用范例：

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

注意：

- 在访问成员时，.或->前后不能有空格。
- 指针操作符* 或 & 后没有空格。

在声明指针变量或参数时，*与类型/变量名直接最好不要都留有空格：

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

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

```
//好，对可读性有帮助
int x, y;
```

可以在同一个声明中声明多个变量，但其中包含指针或者引用的话就不行，因为这样很容易引起误解。

```
int x, *y;  //不可以 - 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
```

在单个文件内要保持风格一致，所以，如果是修改现有文件，要遵照该文件的风格。

14.11. 布尔表达式

规则：如果一个布尔表达式超过标准行宽，断行方式要统一。

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。可以考虑额外插入圆括号，合理使用的圆括号可增强可读性。此外，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。

14.12. 函数返回值

规则：不要在 return 表达式里加上非必须的圆括号。

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

```
return result;                  // 返回值很简单，没有圆括号.
//可以用圆括号把复杂表达式圈起来，改善可读性。
return (some_long_condition &&
        another_condition);
```

```
return (value);                 // 不能写 var = (value);;
```

```
return(result);                 // return 不是函数!
```

14.13. 变量及数组初始化

规则：用 =、 () 和 {} 均可.

可以用 =、 () 和 {}，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"};
```

具有 std::initializer_list 构造函数的类型，使用列表初始化时要非常小心，非空列表初始化会优先调用 std::initializer_list 的构造函数，不过空列表初始化除外，后者原则上会调用默认构造函数。为了强制禁用非 std::initializer_list 构造函数，请改用圆括号。

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

此外，列表初始化不允许整型类型的四舍五入，这可以用来避免一些类型编程错误。

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

14.14. 预处理指令

规则：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               //不必要 - # 后跟空格
    NotifyClient();
# endif
#endif
    BackToNormal();
  }
```

```
//差 - 指令缩进
 if (lopsided_score) {
   #if DISASTER_PENDING  //错！ - "#if" 应该放在行开头
   DropEverything();
   #endif                    //错！ - "#endif" 不要缩进
   BackToNormal();
 }
```

14.15. 类格式

规则：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）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_;
};
```

注意事项：

- 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一节。

14.16. 构造函数初始值列表

规则：构造函数初始化列表放在同一行或按四个空格缩进并排多行。

下面两种初始值列表方式都可以接受：

```
//如果所有东西能放在同一行:
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) {}
```

14.17. 命名空间格式化

规则：命名空间内容不缩进。

命名空间不要增加额外的缩进层次，例如：

```
namespace {
```

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

}  // namespace
```

不要在命名空间内缩进：

```
namespace {

  //错, 缩进多余了.
  void foo() {
    ...
  }

}  // 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 {
```

14.18. 水平留白

规则：水平留白的使用依代码中的位置决定。不要在行尾添加没意义的留白。

通用

```
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.
  ...
```

添加冗余的留白会给其他人编辑时造成额外负担。因此，行尾不要留空格。如果确定一行代码已经修改完毕，将多余的空格去掉，或者在专门清理空格时去掉。

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) {}  // 圆括号内部不留空格.
switch (i) {
for (int i = 0; i < 5; ++i) {
// 循环和条件语句的圆括号里的空格可有可无,
// 总之要一致.
switch ( i ) {
if ( test ) {
for ( int i = 0; i < 5; ++i ) {
// For 循环语句在分号后要有空格，分号前也可能有空格.
for ( ; i < 5 ; ++i) {
  ...

// 基于范围的循环里内 : 前后面一定要有空格.
for (auto x : counts) {
  ...
}
switch (i) {
  case 1:          // switch case 的冒号前无空格.
    ...
  case 2: break;  // 如果冒号后有代码，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.
std::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
std::vector<char *> x;
```

14.19. 垂直留白

规则：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。

基本原则是：同一屏可以显示的代码越多，越容易理解程序的控制流。当然，过于密集的代码块和过于疏松的代码块同样难以阅读。但通常是垂直留白越少越好。

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。

15. 规则特例

前面说明的编程习惯基本都是强制性的。但所有优秀的规则都允许例外，这里就是探讨这些特例。

15.1. 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。

当修改其他风格的代码时，为了与代码原有风格保持一致可以不使用本指南约定。记住，一致性也包括原有的一致性。

15.2. 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。如果习惯使用 Windows 编码风格，这儿有必要重申一下某些可能会忘记的指南：

- 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。即便如此，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。
- 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，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，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。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。

16. 编码规约

16.1. 内存，指针及资源

16.1.1 内存和资源必须遵从谁申请谁释放的原则。

16.1.2 内存或资源释放函数和该内存或资源的申请函数必须配套使用。

注意在函数的各个返回分支都需要释放，特别是异常处理也不要遗漏。

16.1.3 申请内存或资源必须先判断内存或资源有效性。

16.1.4 内存或字符串拷贝前必须进行长度有效性判断，避免内存越界。

16.1.5 禁止引用已释放的内存和资源。

16.1.6 指针使用前，必须进行有效性判定，避免使用空指针。

16.1.7 指针内存释放后，对应指针必须置空。

16.1.8 释放结构体/数组/各类数据容器指针前，必须先释放成员指针。

16.2. 中断处理

16.2.1 中断处理过程中，不允许调用可能导致系统进入睡眠状态的函数。

16.2.2 中断处理过程中，不允许调用调度器，放弃处理器的控制权。

16.3. 变量，宏

16.3.1 严禁使用未被初始化的变量作为右值。

16.3.2 用宏对表达式进行定义时，最外层必须使用括号。

16.4. 数组，结构

16.4.1 数组使用前，必须进行数组下标合法性判断。

16.4.2 数组分配必须考虑最大情况，避免导致空间不够。

16.5. 类型，表达式

16.5.1 类型强制转换必须要防止越界。

16.5.2 运算符使用必须加括号进行保护，禁止使用默认优先级。

16.5.3 对加，减，乘，除运算必须要进行溢出保护。

16.5.4 除法运算，必须对除零操作做有效保护。

16.6. 函数，类

16.6.1 对函数传入参数必须进行有效性检查。

16.6.2 禁止返回函数中定义的局部指针变量。

16.6.3 类中申请的成员内存，在类析构前必须释放。

16.7. 多线程

多线程编程，对于要求线程安全的变量资源，必须加锁保护，避免出现多任务访问异常导致的内存越界。

17. 总结

其实代码规范只是一个 Guideline，不是一定要采取某种特定的风格来编写代码。在编写的代码的时候，能有一个 Guideline（准则）或者说是一个约定，我们共同遵守这样的约定，来达到我一开始说的代码规范性所带来的意义。

正所谓，“离娄之明，公输子之巧，不以规矩，不能成方圆。”