Google C++ 编程规范

目录

一、头文件

通常，每一个.cc 文件（C++的源文件）都有一个对应的.h 文件（头文件），也有一些例外，如单元测试代码和只包含 main()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麻烦。

1. #define 的保护

所有头文件都应该使用#define 防止头文件被多重包含（multiple inclusion），命名格式当是：*<PROJECT>_<PATH>_<FILE>_H_*

为保证唯一性，头文件的命名应基于其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 按如下方式保护：

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

2. 头文件依赖

使用前置声明（forward declarations）尽量减少.h 文件中#include 的数量。

当一个头文件被包含的同时也引入了一项新的依赖（dependency），只要该头文件被修改，代码就要重新编译。如果你的头文件包含了其他头文件，这些头文件的任何改变也将导致那些包含了你的头文件的代码重新编译。因此，我们宁可尽量少包含头文件，尤其是那些包含在其他头文件中的。

使用前置声明可以显著减少需要包含的头文件数量。举例说明：头文件中用到类 File，但不需要访问 File 的声明，则头文件中只需前置声明 `class File;`无需`#include "file/base/file.h"`。

在头文件如何做到使用类 Foo 而无需访问类的定义？

1) 将数据成员类型声明为 Foo *或 Foo &；

2) 参数、返回值类型为 Foo 的函数只是声明（但不定义实现）；

3) 静态数据成员的类型可以被声明为 Foo，因为静态数据成员的定义在类定义之外。

另一方面，如果你的类是 Foo 的子类，或者含有类型为 Foo 的非静态数据成员，则必须为之包含头文件。

有时，使用指针成员（pointer members，如果是 scoped_ptr 更好）替代对象成员（object members）的确更有意义。然而，这样的做法会降低代码可读性及执行效率。如果仅仅为了少包含头文件，还是不要这样替代的好。

当然，.cc 文件无论如何都需要所使用类的定义部分，自然也就会包含若干头文件。

译者注：能依赖声明的就不要依赖定义。

3. 内联函数

只有当函数只有 10 行甚至更少时才会将其定义为内联函数（inline function）。

定义（Definition）：当函数被声明为内联函数之后，编译器可能会将其内联展开，无需按通常的函数调用机制调用内联函数。

优点：当函数体比较小的时候，内联该函数可以令目标代码更加高效。对于存取函数（accessor、mutator）以及其他一些比较短的关键执行函数。

缺点：滥用内联将导致程序变慢，内联有可能是目标代码量或增或减，这取决于被内联的函数的大小。内联较短小的存取函数通常会减少代码量，但内联一个很大的函数（译者注：如果编译器允许的话）将戏剧性的增加代码量。在现代处理器上，由于更好的利用指令缓存（instruction cache），小巧的代码往往执行更快。

结论：一个比较得当的处理规则是，不要内联超过 10 行的函数。对于析构函数应慎重对待，析构函数往往比其表面看起来要长，因为有一些隐式成员和基类析构函数（如果有的话）被调用！

另一有用的处理规则：内联那些包含循环或 switch 语句的函数是得不偿失的，除非在大多数情况下，这些循环或 switch 语句从不执行。

重要的是，虚函数和递归函数即使被声明为内联的也不一定就是内联函数。通常，递归函数不应该被声明为内联的（译者注：递归调用堆栈的展开并不像循环那么简单，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）。析构函数内联的主要原因是其定义在类的定义中，为了方便抑或是对其行为给出文档。

4. -inl.h 文件

复杂的内联函数的定义，应放在后缀名为-inl.h 的头文件中。

在头文件中给出内联函数的定义，可令编译器将其在调用处内联展开。然而，实现代码应完全放到.cc 文件中，我们不希望.h 文件中出现太多实现代码，除非这样做在可读性和效率上有明显优势。

如果内联函数的定义比较短小、逻辑比较简单，其实现代码可以放在.h 文件中。例如，存取函数的实现理所当然都放在类定义中。出于实现和调用的方便，较复杂的内联函数也可以放到.h 文件中，如果你觉得这样会使头文件显得笨重，还可以将其分离到单独的-inl.h 中。这样即把实现和类定义分离开来，当需要时包含实现所在的-inl.h 即可。

-inl.h 文件还可用于函数模板的定义，从而使得模板定义可读性增强。

要提醒的一点是，-inl.h 和其他头文件一样，也需要#define 保护。

5. 函数参数顺序（Function Parameter Ordering）

定义函数时，参数顺序为：输入参数在前，输出参数在后。

C/C++函数参数分为输入参数和输出参数两种，有时输入参数也会输出（译者注：值被修改时）。输入参数一般传值或常数引用（const references），输出参数或输入/输出参数为非常数指针（non-const pointers）。对参数排序时，将所有输入参数置于输出参数之前。不要仅仅因为是新添加的参数，就将其置于最后，而应该依然置于输出参数之前。

这一点并不是必须遵循的规则，输入/输出两用参数（通常是类/结构体变量）混在其中，会使得规则难以遵循。

6. 包含文件的名称及次序

将包含次序标准化可增强可读性、避免隐藏依赖（hidden dependencies，译者注：隐藏依赖主要是指包含的文件中编译时），次序如下：C 库、C++库、其他库的.h、项目内的.h。

项目内头文件应按照项目源代码目录树结构排列，并且避免使用 UNIX 文件路径.（当前目录）和..（父目录）。例如，google-awesome-project/src/base/logging.h 应像这样被包含：

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

dir/foo.cc 的主要作用是执行或测试 dir2/foo2.h 的功能，foo.cc 中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. 其他库头文件
5. 本项目内头文件

这种排序方式可有效减少隐藏依赖，我们希望每一个头文件独立编译。最简单的实现方式是将其作为第一个.h 文件包含在对应的.cc 中。

dir/foo.cc 和 dir2/foo2.h 通常位于相同目录下（像 base/basictypes_unittest.cc 和 base/basictypes.h），但也可在不同目录下。

相同目录下头文件按字母序是不错的选择。

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"
```

二、作用域

1. 命名空间（Namespaces）

在.cc 文件中，提倡使用不具名的命名空间（unnamed namespaces，译者注：不具名的命名空间就像不具名的类一样，似乎被介绍的很少:-(）。使用具名命名空间时，其名称可基于项目或路径名称，不要使用 using 指示符。

定义：命名空间将全局作用域细分为不同的、具名的作用域，可有效防止全局作用域的命名冲突。

优点：命名空间提供了（可嵌套）命名轴线（name axis，译者注：将命名分割在不同命名空间内），当然，类也提供了（可嵌套）的命名轴线（译者注：将命名分割在不同类的作用域内）。

举例来说，两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命名空间中，project1::Foo 和 project2::Foo 作为不同符号自然不会冲突。

缺点：命名空间具有迷惑性，因为它们和类一样提供了额外的（可嵌套的）命名轴线。在头文件中使用不具名的空间容易违背 C++的唯一定义原则（One Definition Rule (ODR)）。

结论：根据下文将要提到的策略合理使用命名空间。

1) 不具名命名空间（Unnamed Namespaces）

在.cc 文件中，允许甚至提倡使用不具名命名空间，以避免运行时的命名冲突：

```
namespace {                                  // .cc 文件中

// 命名空间的内容无需缩进
enum { UNUSED, EOF, ERROR };          // 经常使用的符号
bool AtEof() { return pos_ == EOF; }   // 使用本命名空间内的符号 EOF

}  // namespace
```

然而，与特定类关联的文件作用域声明在该类中被声明为类型、静态数据成员或静态成员函数，而不是不具名命名空间的成员。像上文展示的那样，不具名命名空间结束时用注释// namespace 标识。

不能在.h 文件中使用不具名命名空间。

2) 具名命名空间（Named Namespaces）

具名命名空间使用方式如下：

命名空间将除文件包含、全局标识的声明/定义以及类的前置声明外的整个源文件封装起来，以同其他命名空间相区分。

```
// .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
```

通常的.cc 文件会包含更多、更复杂的细节，包括对其他命名空间中类的引用等。

```
#include "a.h"

DEFINE_bool(someflag, false, "dummy flag");

class C;  // 全局命名空间中类 C 的前置声明
namespace a { class A; }  // 命名空间 a 中的类 a::A 的前置声明

namespace b {

...code for b...              // b 中的代码

}  // namespace b
```

不要声明命名空间 std 下的任何内容，包括标准库类的前置声明。声明 std 下的实体会导致不明确的行为，如，不可移植。声明标准库下的实体，需要包含对应的头文件。

最好不要使用 using 指示符，以保证命名空间下的所有名称都可以正常使用。

```
// 禁止——污染命名空间
using namespace foo;
```

在.cc 文件、.h 文件的函数、方法或类中，可以使用 using。

```
// 允许：.cc 文件中
// .h 文件中，必须在函数、方法或类的内部使用
using ::foo::bar;
```

在.cc 文件、.h 文件的函数、方法或类中，还可以使用命名空间别名。

```
// 允许：.cc 文件中
// .h 文件中，必须在函数、方法或类的内部使用
namespace fbz = ::foo::bar::baz;
```

2. 嵌套类（Nested Class）

当公开嵌套类作为接口的一部分时，虽然可以直接将他们保持在全局作用域中，但将嵌套类的声明置于命名空间中是更好的选择。

定义：可以在一个类中定义另一个类，嵌套类也称成员类（member class）。

```
class Foo {

private:
  // Bar 是嵌套在 Foo 中的成员类
  class Bar {
    ...
  };

};
```

优点：当嵌套（成员）类只在被嵌套类（enclosing class）中使用时很有用，将其置于被嵌套类作用域作为被嵌套类的成员不会污染其他作用域同名类。可在被嵌套类中前置声明嵌套类，在.cc 文件中定义嵌套类，避免在被嵌套类中包含嵌套类的定义，因为嵌套类的定义通常只与实现相关。

缺点：只能在被嵌套类的定义中才能前置声明嵌套类。因此，任何使用 Foo::Bar*指针的头文件必须包含整个 Foo 的声明。

结论：不要将嵌套类定义为 public，除非它们是接口的一部分，比如，某个方法使用了这个类的一系列选项。

3. 非成员函数（Nonmember）、静态成员函数（Static Member）和全局函数（Global Functions）

使用命名空间中的非成员函数或静态成员函数，尽量不要使用全局函数。

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，将非成员函数置于命名空间中可避免对全局作用域的污染。

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依赖时更是如此。

结论：

有时，不把函数限定在类的实体中是有益的，甚至需要这么做，要么作为静态成员，要么作为非成员函数。非成员函数不应依赖于外部变量，并尽量置于某个命名空间中。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。

定义于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连接依赖；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。

如果你确实需要定义非成员函数，又只是在.cc 文件中使用它，可使用不具名命名空间或 static 关联（如 `static int Foo() {...}`）限定其作用域。

4. 局部变量（Local Variables）

将函数变量尽可能置于最小作用域内，在声明变量时将其初始化。

C++允许在函数的任何位置声明变量。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。这使得代码易于阅读，易于定位变量的声明位置、变量类型和初始值。特别是，应使用初始化代替声明+赋值的方式。

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

注意：gcc 可正确执行 for (int i = 0; i < 10; ++i)（i 的作用域仅限 for 循环），因此其他 for 循环中可重用 i。if 和 while 等语句中，作用域声明（scope declaration）同样是正确的。

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

注意：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;  // 构造函数和析构函数分别调用 1000000 次！
  f.DoSomething(i);
}
```

类似变量放到循环作用域外面声明要高效的多：

```
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);
}
```

5. 全局变量（Global Variables）

class 类型的全局变量是被禁止的，内建类型的全局变量是允许的，当然多线程代码中非常数全局变量也是被禁止的。永远不要使用函数返回值初始化全局变量。

不幸的是，全局变量的构造函数、析构函数以及初始化操作的调用顺序只是被部分规定，每次生成有可能会有变化，从而导致难以发现的 bugs。

因此，禁止使用 class 类型的全局变量（包括 STL 的 string, vector 等等），因为它们的初始化顺序有可能导致构造出现问题。内建类型和由内建类型构成的没有构造函数的结构体

可以使用，如果你一定要使用 class 类型的全局变量，请使用单件模式（singleton pattern）。

对于全局的字符串常量，使用 C 风格的字符串，而不要使用 STL 的字符串：

```
const char kFrogSays[] = "ribbet";
```

虽然允许在全局作用域中使用全局变量，使用时务必三思。大多数全局变量应该是类的静态数据成员，或者当其只在.cc 文件中使用时，将其定义到不具名命名空间中，或者使用静态关联以限制变量的作用域。

记住，静态成员变量视作全局变量，所以，也不能是 class 类型！

译者：这一篇主要提到的是作用域的一些规则，总结一下：

1. .cc 中的不具名命名空间可避免命名冲突、限定作用域，避免直接使用 using 提示符污染命名空间；

2. 嵌套类符合局部使用原则，只是不能在其他头文件中前置声明，尽量不要 public；

3. 尽量不用全局函数和全局变量，考虑作用域和命名空间限制，尽量单独形成编译单元；

4. 多线程中的全局变量（含静态成员变量）不要使用 class 类型（含 STL 容器），避免不明确行为导致的 bugs。

作用域的使用，除了考虑名称污染、可读性之外，主要是为降低耦合度，提高编译、执行效率。

三、类

类是 C++中基本的代码单元，自然被广泛使用。本节列举了在写一个类时要做什么、不要做什么。

1. 构造函数（Constructor）的职责

构造函数中只进行那些没有实际意义的（trivial，译者注：简单初始化对于程序执行没有实际的逻辑意义，因为成员变量的“有意义”的值大多不在构造函数中确定）初始化，可能的话，使用 Init()方法集中初始化为有意义的（non-trivial）数据。

定义：在构造函数中执行初始化操作。

优点：排版方便，无需担心类是否初始化。

缺点：在构造函数中执行操作引起的问题有：

1) 构造函数中不易报告错误，不能使用异常。

2) 操作失败会造成对象初始化失败，引起不确定状态。

3) 构造函数内调用虚函数，调用不会派发到子类实现中，即使当前没有子类化实现，将来仍是隐患。

4) 如果有人创建该类型的全局变量（虽然违背了上节提到的规则），构造函数将在 main() 之前被调用，有可能破坏构造函数中暗含的假设条件。例如，gflags 尚未初始化。

结论：如果对象需要有意义的（non-trivial）初始化，考虑使用另外的 Init()方法并（或）增加一个成员标记用于指示对象是否已经初始化成功。

2. 默认构造函数（Default Constructors）

如果一个类定义了若干成员变量又没有其他构造函数，需要定义一个默认构造函数，否则编译器将自动生产默认构造函数。

定义：新建一个没有参数的对象时，默认构造函数被调用，当调用 new[]（为数组）时，默认构造函数总是被调用。

优点：默认将结构体初始化为"不可能的"值，使调试更加容易。

缺点：对代码编写者来说，这是多余的工作。

结论：

如果类中定义了成员变量，没有提供其他构造函数，你需要定义一个默认构造函数（没有参数）。默认构造函数更适合于初始化对象，使对象内部状态（internal state）一致、有效。

提供默认构造函数的原因是：如果你没有提供其他构造函数，又没有定义默认构造函数，编译器将为你自动生成一个，编译器生成的构造函数并不会对对象进行初始化。

如果你定义的类继承现有类，而你又没有增加新的成员变量，则不需要为新类定义默认构造函数。

3. 明确的构造函数（Explicit Constructors）

对单参数构造函数使用 C++关键字 explicit。

定义：通常，只有一个参数的构造函数可被用于转换（conversion，译者注：主要指隐式转换，下文可见），例如，定义了 Foo::Foo(string name)，当向需要传入一个 Foo 对象的函数传入一个字符串时，构造函数 Foo::Foo(string name)被调用并将该字符串转换为一个 Foo 临时对象传给调用函数。看上去很方便，但如果你并不希望如此通过转换生成一个新对象的话，麻烦也随之而来。为避免构造函数被调用造成隐式转换，可以将其声明为 explicit。

优点：避免不合时宜的变换。

缺点：无。

结论：

所有单参数构造函数必须是明确的。在类定义中，将关键字 explicit 加到单参数构造函数前：
explicit Foo(string name);

例外：在少数情况下，拷贝构造函数可以不声明为 explicit；特意作为其他类的透明包装器的类。类似例外情况应在注释中明确说明。

4. 拷贝构造函数（Copy Constructors）

仅在代码中需要拷贝一个类对象的时候使用拷贝构造函数；不需要拷贝时应使用 DISALLOW_COPY_AND_ASSIGN。

定义：通过拷贝新建对象时可使用拷贝构造函数（特别是对象的传值时）。

优点：拷贝构造函数使得拷贝对象更加容易，STL 容器要求所有内容可拷贝、可赋值。

缺点：C++中对象的隐式拷贝是导致很多性能问题和 bugs 的根源。拷贝构造函数降低了代码可读性，相比按引用传递，跟踪按值传递的对象更加困难，对象修改的地方变得难以捉摸。

结论：

大量的类并不需要可拷贝，也不需要一个拷贝构造函数或赋值操作（assignment operator）。不幸的是，如果你不主动声明它们，编译器会为你自动生成，而且是 public 的。

可以考虑在类的 private 中添加空的（dummy）拷贝构造函数和赋值操作，只有声明，没有定义。由于这些空程序声明为 private，当其他代码试图使用它们的时候，编译器将报错。为了方便，可以使用宏 DISALLOW_COPY_AND_ASSIGN：

```
// 禁止使用拷贝构造函数和赋值操作的宏
// 应在类的 private:中使用
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
  TypeName(const TypeName&);               \
  void operator=(const TypeName&)

class Foo {
public:
  Foo(int f);
  ~Foo();

private:
  DISALLOW_COPY_AND_ASSIGN(Foo);
};
```

如上所述，绝大多数情况下都应使用 DISALLOW_COPY_AND_ASSIGN，如果类确实需要可拷贝，应在该类的头文件中说明原由，并适当定义拷贝构造函数和赋值操作，注意在 operator=中检测自赋值（self-assignment）情况。

在将类作为 STL 容器值得时候，你可能有使类可拷贝的冲动。类似情况下，真正该做的是使用指针指向 STL 容器中的对象，可以考虑使用 std::tr1::shared_ptr。

5. 结构体和类（Structs vs. Classes）

仅当只有数据时使用 struct，其它一概使用 class。

在 C++中，关键字 struct 和 class 几乎含义等同，我们为其人为添加语义，以便为定义的数据类型合理选择使用哪个关键字。

struct 被用在仅包含数据的消极对象（passive objects）上，可能包括有关联的常量，但没有存取数据成员之外的函数功能，而存取功能通过直接访问实现而无需方法调用，这儿提到的方法是指只用于处理数据成员的，如构造函数、析构函数、Initialize()、Reset()、Validate()。

如果需要更多的函数功能，class 更适合，如果不确定的话，直接使用 class。

如果与 STL 结合，对于仿函数（functors）和特性（traits）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。

6. 继承（Inheritance）

使用组合（composition，译者注，这一点也是 GoF 在《Design Patterns》里反复强调的）通常比使用继承更适宜，如果使用继承的话，只使用公共继承。

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（implementation inheritance），子类继承父类的实现代码；接口继承（interface inheritance），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重用基类代码减少了代码量。由于继承是编译时声明（compile-time declaration），编码者和编译器都可以理解相应操作并发现错误。接口继承可用于程序上增强类的特定 API 的功能，在类没有定义 API 的必要实现时，编译器同样可以侦错。

缺点：对于实现继承，由于实现子类的代码在父类和子类间延展，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，还要区分基类的物理轮廓（physical layout）。

结论：

所有继承必须是 public 的，如果想私有继承的话，应该采取包含基类实例作为成员的方式作为替代。

不要过多使用实现继承，组合通常更合适一些。努力做到只在“是一个”（"is-a"，译者注，其他"has-a"情况下请使用组合）的情况下使用继承：如果 Bar 的确“是一种”Foo，才令 Bar 是 Foo 的子类。

必要的话，令析构函数为 virtual，必要是指，如果该类具有虚函数，其析构函数应该为虚函数。

译者注：至于子类没有额外数据成员，甚至父类也没有任何数据成员的特殊情况下，析构函数的调用是否必要是语义争论，从编程设计规范的角度看，在含有虚函数的父类中，定义虚析构函数绝对必要。

限定仅在子类访问的成员函数为 protected，需要注意的是数据成员应始终为私有。

当重定义派生的虚函数时，在派生类中明确声明其为 virtual。根本原因：如果遗漏 virtual，阅读者需要检索类的所有祖先以确定该函数是否为虚函数（译者注，虽然不影响其为虚函数的本质）。

7. 多重继承（Multiple Inheritance）

真正需要用到多重实现继承（multiple implementation inheritance）的时候非常少，只有当最多一个基类中含有实现，其他基类都是以 Interface 为后缀的纯接口类时才会使用多重继承。

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，多重实现继承可令你重用更多代码。

缺点：真正需要用到多重实现继承的时候非常少，多重实现继承看上去是不错的解决方案，通常可以找到更加明确、清晰的、不同的解决方案。

结论：只有当所有超类（superclass）除第一个外都是纯接口时才能使用多重继承。为确保它们是纯接口，这些类必须以 Interface 为后缀。

注意：关于此规则，Windows 下有种例外情况（译者注，将在本译文最后一篇的规则例外中阐述）。

8. 接口（Interface）

接口是指满足特定条件的类，这些类以 Interface 为后缀（非必需）。

定义：当一个类满足以下要求时，称之为纯接口：

1) 只有纯虚函数（"=0"）和静态函数（下文提到的析构函数除外）；

2) 没有非静态数据成员；

3) 没有定义任何构造函数。如果有，也不含参数，并且为 protected；

4) 如果是子类，也只能继承满足上述条件并以 Interface 为后缀的类。

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，必须为之声明虚析构函数（作为第 1 条规则的例外，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 12.4 节。

优点：以 Interface 为后缀可令他人知道不能为该接口类增加实现函数或非静态数据成员，这一点对于多重继承尤其重要。另外，对于 Java 程序员来说，接口的概念已经深入人心。

缺点：Interface 后缀增加了类名长度，为阅读和理解带来不便，同时，接口特性作为实现细节不应暴露给客户。

结论：。只有在满足上述需要时，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。

9. 操作符重载（Operator Overloading）

除少数特定环境外，不要重载操作符。

定义：一个类可以定义诸如+、/等操作符，使其可以像内建类型一样直接使用。

优点：使代码看上去更加直观，就像内建类型（如 int）那样，重载操作符使那些 Equals()、Add()等黯淡无光的函数名好玩多了。为了使一些模板函数正确工作，你可能需要定义操作符。

缺点：虽然操作符重载令代码更加直观，但也有一些不足

1) 混淆直觉，让你误以为一些耗时的操作像内建操作那样轻巧；

2) 查找重载操作符的调用处更加困难，查找 Equals()显然比同等调用==容易的多；

3) 有的操作符可以对指针进行操作，容易导致 bugs，Foo + 4 做的是一件事，而&Foo + 4 可能做的是完全不同的另一件事，对于二者，编译器都不会报错，使其很难调试；

4) 重载还有令你吃惊的副作用，比如，重载操作符&的类不能被前置声明。

结论：

一般不要重载操作符，尤其是赋值操作（operator=）比较阴险，应避免重载。如果需要的话，可以定义类似 Equals()、CopyFrom()等函数。

然而，极少数情况下需要重载操作符以便与模板或"标准"C++类衔接（如 operator<<(ostream&, const T&)），如果被证明是正当的尚可接受，但你要尽可能避免这样做。尤其是不要仅仅为了在 STL 容器中作为 key 使用就重载 operator==或 operator<，取而代之，你应该在声明容器的时候，创建相等判断和大小比较的仿函数类型。

有些 STL 算法确实需要重载 operator==时可以这么做，不要忘了提供文档说明原因。

参考**拷贝构造函数**和**函数重载**。

10. 存取控制（Access Control）

将数据成员私有化，并提供相关存取函数，如定义变量 foo_及取值函数 foo()、赋值函数 set_foo()。

存取函数的定义一般内联在头文件中。

参考**继承**和**函数命名**。

11. 声明次序（Declaration Order）

在类中使用特定的声明次序：public:在 private:之前，成员函数在数据成员（变量）前。

定义次序如下：public:、protected:、private:，如果那一块没有，直接忽略即可。

每一块中，声明次序一般如下：

1) typedefs 和 enums；

2) 常量；

3) 构造函数；

4) 析构函数；

5) 成员函数，含静态成员函数；

6) 数据成员，含静态数据成员。

宏 DISALLOW_COPY_AND_ASSIGN 置于 private:块之后，作为类的最后部分。参考**拷贝构造函数**。

.cc 文件中函数的定义应尽可能和声明次序一致。

不要将大型函数内联到类的定义中，通常，只有那些没有特别意义的或者性能要求高的，并且是比较短小的函数才被定义为内联函数。更多细节参考译文第一篇的**内联函数**。

12. 编写短小函数（Write Short Functions）

倾向于选择短小、凝练的函数。

长函数有时是恰当的，因此对于函数长度并没有严格限制。如果函数超过 40 行，可以考虑在不影响程序结构的情况下将其分割一下。

即使一个长函数现在工作的非常好，一旦有人对其修改，有可能出现新的问题，甚至导致难以发现的 bugs。使函数尽量短小、简单，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、调试困难，或者你需要使用其中的一小块，考虑将其分割为更加短小、易于管理的若干函数。

译者：关于类的注意事项，总结一下：

1. 不在构造函数中做太多逻辑相关的初始化；

2. 编译器提供的默认构造函数不会对变量进行初始化，如果定义了其他构造函数，编译器不再提供，需要编码者自行提供默认构造函数；

3. 为避免隐式转换，需将单参数构造函数声明为 explicit；

4. 为避免拷贝构造函数、赋值操作的滥用和编译器自动生成，可目前声明其为 **private** 且无需实现；

5. 仅在作为数据集合时使用 **struct**；

6. 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** 关键字，虽然编译器允许不这样做；

7. 避免使用多重继承，使用时，除一个基类含有实现外，其他基类均为纯接口；

8. 接口类类名以 **Interface** 为后缀，除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，不提供构造函数，提供的话，声明为 **protected**；

9. 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；

10. 存取函数一般内联在头文件中；

11. 声明次序：**public->protected->private**；

12. 函数体尽量短小、紧凑，功能单一。

四、Google 特有的风情

Google 有很多自己实现的使 C++代码更加健壮的技巧、功能，以及有异于别处的 C++的使用方式。

1. 智能指针（Smart Pointers）

如果确实需要使用智能指针的话，scoped_ptr 完全可以胜任。在非常特殊的情况下，例如对 STL 容器中对象，你应该只使用 std::tr1::shared_ptr，任何情况下都不要使用 auto_ptr。

“智能”指针看上去是指针，其实是附加了语义的对象。以 scoped_ptr 为例，scoped_ptr 被销毁时，删除了它所指向的对象。shared_ptr 也是如此，而且，shared_ptr 实现了引用计数（reference-counting），从而只有当它所指向的最后一个对象被销毁时，指针才会被删除。

一般来说，我们倾向于设计对象隶属明确的代码，最明确的对象隶属是根本不使用指针，直接将对象作为一个域（field）或局部变量使用。另一种极端是引用计数指针不属于任何对象，这样设计的问题是容易导致循环引用或其他导致对象无法删除的诡异条件，而且在每一次拷贝或赋值时连原子操作都会很慢。

虽然不推荐这么做，但有些时候，引用计数指针是最简单有效的解决方案。

译者注：看来，Google 所谓的不同之处，在于尽量避免使用智能指针:D，使用时也尽量局部化，并且，安全第一。

五、其他 C++特性

1. 引用参数（Reference Arguments）

所以按引用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，形参（parameter）必须为指针，如 int foo(int *pval)。在 C++中，函数还可以声明引用形参：int foo(int &val)。

优点：定义形参为引用避免了像(*pval)++这样丑陋的代码，像拷贝构造函数这样的应用也是必需的，而且不像指针那样不接受空指针 NULL。

缺点：容易引起误解，因为引用在语法上是值却拥有指针的语义。

结论：

函数形参表中，所有引用必须是 const：

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

事实上这是一个硬性约定：输入参数为值或常数引用，输出参数为指针；输入参数可以是常数指针，但不能使用非常数引用形参。

在强调参数不是拷贝而来，在对象生命期内必须一直存在时可以使用常数指针，最好将这些在注释中详细说明。bind2nd 和 mem_fun 等 STL 适配器不接受引用形参，这种情况下也必须以指针形参声明函数。

2. 函数重载（Function Overloading）

仅在输入参数类型不同、功能相同时使用重载函数（含构造函数），不要使用函数重载模仿缺省函数参数。

定义：可以定义一个函数参数类型为 const string&，并定义其重载函数类型为 const char*。

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

优点：通过重载不同参数的同名函数，令代码更加直观，模板化代码需要重载，同时为访问者带来便利。

缺点：限制使用重载的一个原因是在特定调用处很难确定到底调用的是哪个函数，另一个原因是当派生类只重载函数的部分变量会令很多人对继承语义产生困惑。此外在阅读库的客户端代码时，因缺省函数参数造成不必要的费解。

结论：如果你想重载一个函数，考虑让函数名包含参数信息，例如，使用 AppendString()、AppendInt()而不是 Append()。

3. 缺省参数（Default Arguments）

禁止使用缺省函数参数。

优点：经常用到一个函数带有大量缺省值，偶尔会重写一下这些值，缺省参数为很少涉及的例外情况提供了少定义一些函数的方便。

缺点：大家经常会通过查看现有代码确定如何使用 API，缺省参数使得复制粘贴以前的代码难以呈现所有参数，当缺省参数不适用于新代码时可能导致重大问题。

结论：所有参数必须明确指定，强制程序员考虑 API 和传入的各参数值，避免使用可能不为程序员所知的缺省参数。

4. 变长数组和 alloca（Variable-Length Arrays and alloca()）

禁止使用变长数组和 alloca()。

优点：变长数组具有浑然天成的语法，变长数组和 alloca()也都很高效。

缺点：变长数组和 alloca()不是标准 C++的组成部分，更重要的是，它们在堆栈（stack）上根据数据分配大小可能导致难以发现的内存泄漏："在我的机器上运行的好好的，到了产品中却莫名其妙的挂掉了"。

结论：

使用安全的分配器（allocator），如 scoped_ptr/scoped_array。

5. 友元（Friends）

允许合理使用友元类及友元函数。

通常将友元定义在同一文件下，避免读者跑到其他文件中查找其对某个类私有成员的使用。经常用到友元的一个地方是将 FooBuilder 声明为 Foo 的友元，FooBuilder 以便可以正确构造 Foo 的内部状态，而无需将该状态暴露出来。某些情况下，将一个单元测试用类声明为待测类的友元会很方便。

友元延伸了（但没有打破）类的封装界线，当你希望只允许另一个类访问某个成员时，使用友元通常比将其声明为 public 要好得多。当然，大多数类应该只提供公共成员与其交互。

6. 异常（Exceptions）

不要使用 C++异常。

优点：

1）异常允许上层应用决定如何处理在底层嵌套函数中发生的"不可能发生"的失败，不像出错代码的记录那么模糊费解；

2) 应用于其他很多现代语言中，引入异常使得 C++与 Python、Java 及其他与 C++相近的语言更加兼容；

3) 许多 C++第三方库使用异常，关闭异常将导致难以与之结合；

4) 异常是解决构造函数失败的唯一方案，虽然可以通过工厂函数（factory function）或 Init()方法模拟异常，但他们分别需要堆分配或新的"非法"状态；

5) 在测试框架（testing framework）中，异常确实很好用。

缺点：

1) 在现有函数中添加 throw 语句时，必须检查所有调用处，即使它们至少具有基本的异常安全保护，或者程序正常结束，永远不可能捕获该异常。例如：if f() calls g() calls h()，h 抛出被 f 捕获的异常，g 就要当心了，避免没有完全清理；

2) 通俗一点说，异常会导致程序控制流（control flow）通过查看代码无法确定：函数有可能在不确定的地方返回，从而导致代码管理和调试困难，当然，你可以通过规定何时何地如何使用异常来最小化的降低开销，却给开发人员带来掌握这些规定的负担；

3) 异常安全需要 RAII 和不同编码实践。轻松、正确编写异常安全代码需要大量支撑。允许使用异常；

4) 加入异常使二进制执行代码体积变大，增加了编译时长（或许影响不大），还可能增加地址空间压力；

5) 异常的实用性可能会刺激开发人员在不恰当的时候抛出异常，或者在不安全的地方从异常中恢复，例如，非法用户输入可能导致抛出异常。如果允许使用异常会使得这样一篇编程风格指南长出很多（译者注，这个理由有点牵强:-(）！

结论：

从表面上看，使用异常利大于弊，尤其是在新项目中，然而，对于现有代码，引入异常会牵连到所有依赖代码。如果允许异常在新项目中使用，在跟以前没有使用异常的代码整合时也是一个麻烦。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多少要大一点，迁移过程会比较慢，也容易出错。我们也不相信异常的有效替代方案，如错误代码、断言等，都是严重负担。

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。因为我们希望使用 Google 上的开源项目，但项目中使用异常会为此带来不便，因为我们也建议不要在 Google 上的开源项目中使用异常，如果我们需要把这些项目推倒重来显然不太现实。

对于 Windows 代码来说，这一点有个例外（等到最后一篇吧:D）。

译者注：对于异常处理，显然不是短短几句话能够说清楚的，以构造函数为例，很多 C++ 书籍上都提到当构造失败时只有异常可以处理，Google 禁止使用异常这一点，仅仅是为了自身的方便，说大了，无非是基于软件管理成本上，实际使用中还是自己决定。

7. 运行时类型识别（Run-Time Type Information, RTTI）

我们禁止使用 RTTI。

定义：RTTI 允许程序员在运行时识别 C++类对象的类型。

优点：

RTTI 在某些单元测试中非常有用，如在进行工厂类测试时用于检验一个新建对象是否为期望的动态类型。

除测试外，极少用到。

缺点：运行时识别类型意味著设计本身有问题，如果你需要在运行期间确定一个对象的类型，这通常说明你需要重新考虑你的类的设计。

结论：

除单元测试外，不要使用 RTTI，如果你发现需要所写代码因对象类型不同而动作各异的话，考虑换一种方式识别对象类型。

虚函数可以实现随子类类型不同而执行不同代码，工作都是交给对象本身去完成。

如果工作在对象之外的代码中完成，考虑双重分发方案，如 Visitor 模式，可以方便的在对象本身之外确定类的类型。

如果你认为上面的方法你掌握不了，可以使用 RTTI，但务必请三思，不要去手工实现一个貌似 RTTI 的方案（RTTI-like workaround），我们反对使用 RTTI，同样反对贴上类型标签的貌似类继承的替代方案（译者注，使用就使用吧，不使用也不要造轮子:D）。

8. 类型转换（Casting）

使用 `static_cast<>()`等 C++的类型转换，不要使用 `int y = (int)x` 或 `int y = int(x);`。

定义：C++引入了有别于 C 的不同类型的类型转换操作。

优点：C 语言的类型转换问题在于操作比较含糊：有时是在做强制转换（如`(int)3.5`），有时是在做类型转换（如`(int)"hello"`）。另外，C++的类型转换查找更容易、更醒目。

缺点：语法比较恶心（nasty）。

结论：使用 C++风格而不要使用 C 风格类型转换。

1) static_cast：和 C 风格转换相似可做值的强制转换，或指针的父类到子类的明确的向上转换；

2) const_cast：移除 const 属性；

3) reinterpret_cast：指针类型和整型或其他指针间不安全的相互转换，仅在你对所做一切了然于心时使用；

4) dynamic_cast：除测试外不要使用，除单元测试外，如果你需要在运行时确定类型信息，说明设计有缺陷（参考 **RTTI**）。

9. 流（Streams）

只在记录日志时使用流。

定义：流是 printf()和 scanf()的替代。

优点：有了流，在输出时不需要关心对象的类型，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），打开、关闭对应文件时，流可以自动构造、析构。

缺点：流使得 pread()等功能函数很难执行，如果不使用 printf 之类的函数而是使用流很难对格式进行操作(尤其是常用的格式字符串%.*s),流不支持字符串操作符重新定序(%1s),而这一点对国际化很有用。

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，代码一致性胜过一切，不要在代码中使用流。

拓展讨论：

对这一条规则存在一些争论，这儿给出深层次原因。回忆唯一性原则（Only One Way）：我们希望在任何时候都只使用一种确定的 I/O 类型，使代码在所有 I/O 处保持一致。因此，我们不希望用户来决定是使用流还是 printf + read/write，我们应该决定到底用哪一种方式。把日志作为例外是因为流非常适合这么做，也有一定的历史原因。

流的支持者们主张流是不二之选，但观点并不是那么清晰有力，他们所指出流的所有优势也正是其劣势所在。流最大的优势是在输出时不需要关心输出对象的类型，这是一个亮点，也是一个不足：很容易用错类型，而编译器不会报警。使用流时容易造成的一类错误是：

```
cout << this;  // Prints the address
cout << *this;  // Prints the contents
```

编译器不会报错，因为<<被重载，就因为这一点我们反对使用操作符重载。

有人说 printf 的格式化丑陋不堪、易读性差，但流也好不到哪儿去。看看下面两段代码吧，哪个更加易读？

```
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”，这儿可以，其他地方呢？而且不要忘了，我们的目标是使语言尽可能小，而不是添加一些别人需要学习的新的内容。

每一种方式都是各有利弊，“没有最好，只有更好”，简单化的教条告诫我们必须从中选择其一，最后的多数决定是 printf + read/write。

10. 前置自增和自减（Preincrement and Predecrement）

对于迭代器和其他模板对象使用前缀形式（++i）的自增、自减运算符。

定义：对于变量在自增（++i 或 i++）或自减（--i 或 i--）后表达式的值又没有没用到的情况下，需要确定到底是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（++i）通常要比后置自增（i++）效率更高，因为后置的自增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，拷贝的代价是比较大的。既然两种自增方式动作一样（译者注，不考虑表达式的值，相信你知道我在说什么），为什么不直接使用前置自增呢？

缺点：C 语言中，当表达式的值没有使用时，传统的做法是使用后置自增，特别是在 for 循环中，有些人觉得后置自增更加易懂，因为这很像自然语言，主语（i）在谓语动词（++）前。

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

11. const 的使用（Use of const）

我们强烈建议你在任何可以使用的情况下都要使用 const。

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修改（如 const int foo），为类中的函数加上 const 限定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：人们更容易理解变量是如何使用的，编辑器可以更好地进行类型检测、更好地生成代码。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。即使是在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：如果你向一个函数传入 const 变量，函数原型中也必须是 const 的（否则变量需要 const_cast 类型转换），在调用库函数时这尤其是个麻烦。

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，更好的尽早发现错误。因此，我们强烈建议在任何可以使用的情况下使用 const：

1) 如果函数不会修改传入的引用或指针类型的参数，这样的参数应该为 const；

2) 尽可能将函数声明为 const，访问函数应该总是 const，其他函数如果不会修改任何数据成员也应该是 const，不要调用非 const 函数，不要返回对数据成员的非 const 指针或引用；

3) 如果数据成员在对象构造之后不再改变，可将其定义为 const。

然而，也**不要对 const 过度使用**，像 const int * const * const x;就有些过了，即便这样写精确描述了 x，其实写成 const int** x 就可以了。

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

const 位置：

有人喜欢 int const *foo 形式不喜欢 const int* foo，他们认为前者更加一致因此可读性更好：遵循了 const 总位于其描述的对象（int）之后的原则。但是，一致性原则不适用于此，"不要过度使用"的权威抵消了一致性使用。将 const 放在前面才更易读，因为在自然语言中形容词（const）是在名词（int）之前的。

这是说，我们提倡 const 在前，并不是要求，但要兼顾代码的一致性！

12. 整型（Integer Types）

C++内建整型中，唯一用到的是 int，如果程序中需要不同大小的变量，可以使用<stdint.h>中的精确宽度（precise-width）的整型，如 int16_t。

定义：C++没有指定整型的大小，通常人们认为 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，在需要确定大小的整型时可以使用它们代替 short、unsigned long long 等，在 C 整型中，只使用 int。适当情况下，推荐使用标准类型如 size_t 和 ptrdiff_t。

最常使用的是，对整数来说，通常不会用到太大，如循环计数等，可以使用普通的 int。你可以认为 int 至少为 32 位，但不要认为它会多于 32 位，需要 64 位整型的话，可以使用 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，除非你是在表示一个位组（bit pattern）而不是一个数值。即使数值不会为负值也不要使用无符号类型，使用断言（assertion，译者注，这一点很有道理，计算机只会根据变量、返回值等有无符号确定数值正负，仍然无法确定对错）来保护数据。

无符号整型：

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，类型表明了数值取值形式。但是，在 C 语言中，这一优点被由其导致的 bugs 所淹没。看看：

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

上述代码永远不会终止！有时 gcc 会发现该 bug 并报警，但通常不会。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（type-promotion scheme，C 语言中各种内建类型之间的提升转换关系）会致使无符号类型的行为出乎你的意料。

因此，使用断言声明变量为非负数，不要使用无符号型。

13. 64 位下的可移植性（64-bit Portability）

代码在 64 位和 32 位的系统中，原则上应该都比较友好，尤其对于输出、比较、结构对齐（structure alignment）来说：

1) printf()指定的一些类型在 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。不幸的是，MSVC 7.1 并非全部支持，而且标准中也有所遗漏。所以有时我们就不得不自己定义丑陋的版本（使用标准风格要包含文件 inttypes.h）：

```
// 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 *（或其他指针类型）	%lx	%p	
int64_t	%qd, %lld	%"PRId64"	
uint64_t	%qu, %llu, %llx	%"PRIu64", %"PRIx64"	
size_t	%u	%"PRIuS", %"PRIxS"	C99 指定%zu
ptrdiff_t	%d	%"PRIdS"	C99 指定%zd

注意宏 PRI*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，在使用宏 PRI*时同样可以在%后指定长度等信息。例如，

printf("x = %30"PRIuS"\n", x)在 32 位 Linux 上将被扩展为 printf("x = %30" "u" "\n", x)，编译器会处理为 printf("x = %30u\n", x)。

2) 记住 sizeof(void *) != sizeof(int)，如果需要一个指针大小的整数要使用 intptr_t。

3) 需要对结构对齐加以留心，尤其是对于存储在磁盘上的结构体。在 64 位系统中，任何拥有 int64_t/uint64_t 成员的类/结构体将默认被处理为 8 字节对齐。如果 32 位和 64 位代码共用磁盘上的结构体，需要确保两种体系结构下的结构体的对齐一致。大多数编译器提供了调整结构体对齐的方案。gcc 中可使用__attribute__((packed))，MSVC 提供了#pragma pack()和__declspec(align())（译者注，解决方案的项目属性里也可以直接设置）。

4) 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;
```

5) 如果你确实需要 32 位和 64 位系统具有不同代码，可以在代码变量前使用。（尽量不要这么做，使用时尽量使修改局部化）。

14. 预处理宏（Preprocessor Macros）

使用宏时要谨慎，尽量以内联函数、枚举和常量代替之。

宏意味着你和编译器看到的代码是不同的，因此可能导致异常行为，尤其是当宏存在于全局作用域中。

值得庆幸的是，C++中，宏不像 C 中那么必要。宏内联效率关键代码（performance-critical code）可以内联函数替代；宏存储常量可以 const 变量替代；宏"缩写"长变量名可以引用替代；使用宏进行条件编译，这个……，最好不要这么做，会令测试更加痛苦（#define 防止头文件重包含当然是个例外）。

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（如字符串化（stringifying，译者注，使用#）、连接（concatenation，译者注，使用##）等等）。但在使用前，仔细考虑一下能不能不使用宏实现同样效果。

译者注：关于宏的高级应用，可以参考　　。

下面给出的用法模式可以避免一些使用宏的问题，供使用宏时参考：

1) 不要在.h 文件中定义宏；

2) 使用前正确#define，使用后正确#undef；

3) 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；

4) 不使用会导致不稳定的 C++构造（unbalanced C++ constructs，译者注）的宏，至少文档说明其行为。

15. 0 和 NULL（0 and NULL）

整数用 0，实数用 0.0，指针用 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 还是 NULL，Bjarne Stroustrup 建议使用最原始的 0，我们建议使用看上去像是指针的 NULL，事实上一些 C++编译器（如 gcc 4.1.0）专门提供了 NULL 的定义，可以给出有用的警告，尤其是 sizeof(NULL)和 sizeof(0)不相等的情况。

字符（串）用'\0'，不仅类型正确而且可读性好。

16. sizeof（sizeof）

尽可能用 sizeof(*varname*)代替 sizeof(*type*)。

使用 sizeof(*varname*)是因为当变量类型改变时代码自动同步，有些情况下 sizeof(*type*)或许有意义，还是要尽量避免，如果变量类型改变的话不能同步。

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

17. Boost 库（Boost）

只使用 Boost 中被认可的库。

定义：Boost 库集是一个非常受欢迎的、同级评议的（peer-reviewed）、免费的、开源的 C++库。

优点：Boost 代码质量普遍较高、可移植性好，填补了 C++标准库很多空白，如型别特性（type traits）、更完善的绑定（binders）、更好的智能指针，同时还提供了 TR1（标准库的扩展）的实现。

缺点：某些 Boost 库提倡的编程实践可读性差，像元程序（metaprogramming）和其他高级模板技术，以及过度"函数化"（"functional"）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，我们只允许使用 Boost 特性的一个成熟子集，当前，这些库包括：

1) Compressed Pair：boost/compressed_pair.hpp；

2) Pointer Container：boost/ptr_container 不包括 ptr_array.hpp 和序列化（serialization）。

我们会积极考虑添加可以的 Boost 特性，所以不必拘泥于该规则。

译者：关于 C++特性的注意事项，总结一下：

1. 对于智能指针，安全第一、方便第二，尽可能局部化（scoped_ptr）；

2. 引用形参加上 const，否则使用指针形参；

3. 函数重载的使用要清晰、易读；

4. 鉴于容易误用，禁止使用缺省函数参数（值得商榷）；

5. 禁止使用变长数组；

6. 合理使用友元；

7. 为了方便代码管理，禁止使用异常（值得商榷）；

8. 禁止使用 **RTTI**，否则重新设计代码吧；

9. 使用 **C++**风格的类型转换，除单元测试外不要使用 **dynamic_cast**；

10. 使用流还 **printf + read/write**，**it is a problem**；

11. 能用前置自增/减不用后置自增/减；

12. const 能用则用，提倡 **const** 在前；

13. 使用确定大小的整型，除位组外不要使用无符号型；

14. 格式化输出及结构对齐时，注意 **32** 位和 **64** 位的系统差异；

15. 除字符串化、连接外尽量避免使用宏；

16. 整数用 **0**，实数用 **0.0**，指针用 **NULL**，字符（串）用**'\0'**；

17. 用 **sizeof(varname)**代替 **sizeof(type)**；

18. 只使用 **Boost** 中被认可的库。

六、命名约定

最重要的一致性规则是命名管理，命名风格直接可以直接确定命名实体是：类型、变量、函数、常量、宏等等，无需查找实体声明，我们大脑中的模式匹配引擎依赖于这些命名规则。

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，所以不管你怎么想，规则总归是规则。

1. 通用命名规则（General Naming Rules）

函数命名、变量命名、文件命名应具有描述性，不要过度缩写，类型和变量应该是名词，函数名可以用"命令性"动词。

如何命名：

尽可能给出描述性名称，不要节约空间，让别人很快理解你的代码更重要，好的命名选择：

```
int num_errors;                 // Good.
int num_completed_connections;  // Good.
```

丑陋的命名使用模糊的缩写或随意的字符：

```
int n;                          // Bad - meaningless.
int nerr;                       // Bad - ambiguous abbreviation.
int n_comp_conns;               // Bad - ambiguous abbreviation.
```

类型和变量名一般为名词：如 FileOpener、num_errors。

函数名通常是指令性的，如 OpenFile()、set_num_errors()，访问函数需要描述的更细致，要与其访问的变量相吻合。

缩写：

除非放到项目外也非常明了，否则不要使用缩写，例如：

```
// Good
// These show proper names with no abbreviations.
int num_dns_connections;  // Most people know what "DNS" stands for.
int price_count_reader;   // OK, price count. Makes sense.


// Bad!
// Abbreviations can be confusing or ambiguous outside a small group.
int wgc_connections;  // Only your group knows what this stands for.
int pc_reader;        // Lots of things can be abbreviated "pc".
```

不要用省略字母的缩写：

```
int error_count;  // Good.
int error_cnt;    // Bad.
```

2. 文件命名（File Names）

文件名要全部小写，可以包含下划线（_）或短线（-），按项目约定来。

可接受的文件命名：

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

C++文件以.cc 结尾，头文件以.h 结尾。

不要使用已经存在于/usr/include 下的文件名（译者注，对 UNIX、Linux 等系统而言），如 db.h。

通常，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。如果代码比较长，可以放到以-inl.h 结尾的文件中。对于包含大量内联代码的类，可以有三个文件：

```
url_table.h      // The class declaration.
```

```
url_table.cc      // The class definition.
url_table-inl.h   // Inline functions that include lots of code.
```

参考第一篇**-inl.h** 文件一节。

3. 类型命名（Type Names）

类型命名每个单词以大写字母开头，不包含下划线：MyExcitingClass、MyExcitingEnum。

所有类型命名——类、结构体、类型定义（typedef）、枚举——使用相同约定，例如：

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

// typedefs
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// enums
enum UrlTableErrors { ...
```

4. 变量命名（Variable Names）

变量名一律小写，单词间以下划线相连，类的成员变量以下划线结尾，如 my_exciting_local_variable、my_exciting_member_variable_。

普通变量命名：

举例：

```
string table_name;  // OK - uses underscore.
string tablename;   // OK - all lowercase.
string tableName;   // Bad - mixed case.
```

类数据成员：

结构体的数据成员可以和普通变量一样，不用像类那样接下划线：

```
struct UrlTableProperties {
  string name;
  int num_entries;
}
```

结构体与类的讨论参考第三篇**结构体 vs.类**一节。

全局变量：

对全局变量没有特别要求，少用就好，可以以 g_或其他易与局部变量区分的标志为前缀。

5. 常量命名（Constant Names）

在名称前加 k：kDaysInAWeek。

所有编译时常量（无论是局部的、全局的还是类中的）和其他变量保持些许区别，k 后接大写字母开头的单词：

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

6. 函数命名（Function Names）

普通函数（regular functions，译者注，这里与访问函数等特殊函数相对）大小写混合，存取函数（accessors and mutators）则要求与变量名匹配：MyExcitingFunction()、MyExcitingMethod()、my_exciting_member_variable()、set_my_exciting_member_variable()。

普通函数：

函数名以大写字母开头，每个单词首字母大写，没有下划线：

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

存取函数：

存取函数要与存取的变量名匹配，这儿摘录一个拥有实例变量 num_entries_的类：

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

 private:
  int num_entries_;
};
```

其他短小的内联函数名也可以使用小写字母，例如，在循环中调用这样的函数甚至都不需要缓存其值，小写命名就可以接受。

译者注：从这一点上可以看出，小写的函数名意味着可以直接内联使用。

7. 命名空间（Namespace Names）

命名空间的名称是全小写的，其命名基于项目名称和目录结构：google_awesome_project。

关于命名空间的讨论和如何命名，参考第二篇**命名空间**。

8. 枚举命名（Enumerator Names）

枚举值应全部大写，单词间以下划线相连：MY_EXCITING_ENUM_VALUE。

枚举名称属于类型，因此大小写混合：UrlTableErrors。

```
enum UrlTableErrors {
  OK = 0,
  ERROR_OUT_OF_MEMORY,
```

```
    ERROR_MALFORMED_INPUT,
  };
```

9. 宏命名（Macro Names）

你并不打算使用宏，对吧？如果使用，像这样：MY_MACRO_THAT_SCARES_SMALL_CHILDREN。

参考第四篇**预处理宏**，通常是不使用宏的，如果绝对要用，其命名像枚举命名一样全部大写、使用下划线：

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

10. 命名规则例外（Exceptions to Naming Rules）

当命名与现有 C/C++实体相似的对象时，可参考现有命名约定：

bigopen()
　　函数名，参考 open()
uint
　　typedef 类型定义
bigpos
　　struct 或 class，参考 pos
sparse_hash_map
　　STL 相似实体；参考 STL 命名约定
LONGLONG_MAX
　　常量，类似 INT_MAX

译者：命名约定就相对轻松许多，在遵从代码一致性、可读性的前提下，略显随意：

1. 总体规则：不要随意缩写，如果说 ChangeLocalValue 写作 ChgLocVal 还有情可原的话，把 ModifyPlayerName 写作 MdfPlyNm 就太过分了，除函数名可适当为动词外，其他命名尽量使用清晰易懂的名词；

2. 宏、枚举等使用全部大写+下划线；

3. 变量（含类、结构体成员变量）、文件、命名空间、存取函数等使用全部小写+下划线，类成员变量以下划线结尾，全局变量以 g_开头；

4. 普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线；

5. 参考现有或相近命名约定。

七、注释

注释虽然写起来很痛苦，但对保证代码可读性至为重要，下面的规则描述了应该注释什么、注释在哪儿。当然也要记住，注释的确很重要，但最好的代码本身就是文档（self-documenting），类型和变量命名意义明确要比通过注释解释模糊的命名好得多。

注释是为别人（下一个需要理解你的代码的人）而写的，认真点吧，那下一个人可能就是你！

1. 注释风格（Comment Style）

使用//或/* */，统一就好。

//或/* */都可以，//只是用的更加广泛，在如何注释和注释风格上确保统一。

2. 文件注释（File Comments）

在每一个文件开头加入版权公告，然后是文件内容描述。

法律公告和作者信息：

每一文件包含以下项，依次是：

1) 版权（copyright statement）：如 Copyright 2008 Google Inc.；

2) 许可版本（license boilerplate）：为项目选择合适的许可证版本，如 Apache 2.0、BSD、LGPL、GPL；

3) 作者（author line）：标识文件的原始作者。

如果你对其他人创建的文件做了重大修改，将你的信息添加到作者信息里，这样当其他人对该文件有疑问时可以知道该联系谁。

文件内容：

每一个文件版权许可及作者信息后，都要对文件内容进行注释说明。

通常，.h 文件要对所声明的类的功能和用法作简单说明，.cc 文件包含了更多的实现细节或算法讨论，如果你感觉这些实现细节或算法讨论对于阅读有帮助，可以把.cc 中的注释放到.h 中，并在.cc 中指出文档在.h 中。

不要单纯在.h 和.cc 间复制注释，复制的注释偏离了实际意义。

3. 类注释（Class Comments）

每个类的定义要附着描述类的功能和用法的注释。

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

```
//    delete iter;
class GargantuanTable_Iterator {
  ...
};
```

如果你觉得已经在文件顶部详细描述了该类，想直接简单的来上一句“完整描述见文件顶部”的话，还是多少在类中加点注释吧。

如果类有任何同步前提（synchronization assumptions），文档说明之。如果该类的实例可被多线程访问，使用时务必注意文档说明。

4. 函数注释（Function Comments）

函数声明处注释描述函数功能，定义处描述函数实现。

函数声明：

注释于声明之前，描述函数功能及用法，注释使用描述式（"Opens the file"）而非指令式（"Open the file"）；注释只是为了描述函数而不是告诉函数做什么。通常，注释不会描述函数如何实现，那是定义部分的事情。

函数声明处注释的内容：

1) inputs（输入）及 outputs（输出）；

2) 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；

3) 如果函数分配了空间，需要由调用者释放；

4) 参数是否可以为 NULL；

5) 是否存在函数使用的性能隐忧（performance implications）；

6) 如果函数是可重入的（re-entrant），其同步前提（synchronization assumptions）是什么？

举例如下：

```
// 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;
```

但不要有无谓冗余或显而易见的注释，下面的注释就没有必要加上“returns false otherwise”，因为已经暗含其中了：

```
// Returns true if the table cannot hold any more entries.
bool IsTableFull();
```

注释构造/析构函数时，记住，读代码的人知道构造/析构函数是什么，所以“destroys this object”这样的注释是没有意义的。说明构造函数对参数做了什么（例如，是否是指针的所有者）以及析构函数清理了什么，如果都是无关紧要的内容，直接省掉注释，析构函数前没有注释是很正常的。

函数定义：

每个函数定义时要以注释说明函数功能和实现要点，如使用的漂亮代码、实现的简要步骤、如此实现的理由、为什么前半部分要加锁而后半部分不需要。

不要从.h 文件或其他地方的函数声明处直接复制注释，简要说明函数功能是可以的，但重点要放在如何实现上。

5. 变量注释（Variable Comments）

通常变量名本身足以很好说明变量用途，特定情况下，需要额外注释说明。

类数据成员：

每个类数据成员（也叫实例变量或成员变量）应注释说明用途，如果变量可以接受 NULL 或-1 等警戒值（sentinel values），须说明之，如：

```
private:
 // Keeps track of the total number of entries in the table.
 // Used to ensure we do not go over the limit. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

全局变量（常量）：

和数据成员相似，所有全局变量（常量）也应注释说明含义及用途，如：

```
// The total number of tests cases that we run through in this regression
test.
const int kNumTestCases = 6;
```

6. 实现注释（Implementation Comments）

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释。

代码前注释：

出彩的或复杂的代码块前要加注释，如：

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++) {
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

行注释：

比较隐晦的地方要在行尾加入注释，可以在代码之后空两格加行尾注释，如：

```
// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
  return;  // Error already logged.
```

注意，有两块注释描述这段代码，当函数返回时注释提及错误已经被记入日志。

前后相邻几行都有注释，可以适当调整使之可读性更好：

```
...
DoSomething();                  // Comment here so the comments line up.
DoSomethingElseThatIsLonger();  // Comment here so there are two spaces between
                                // the code and the comment.
...
```

NULL、true/false、1、2、3……：

向函数传入、布尔值或整数时，要注释说明含义，或使用常量让代码望文知意，比较一下：

```
bool success = CalculateSomething(interesting_value,
                                  10,
                                  false,
                                  NULL);  // What are these arguments??
```

和：

```
bool success = CalculateSomething(interesting_value,
                                  10,     // Default base value.
```

```
                                    false,  // Not the first time we're calling
this.
                                    NULL);  // No callback.
```

使用常量或描述性变量：

```
const int kDefaultBaseValue = 10;
const bool kFirstTimeCalling = false;
Callback *null_callback = NULL;
bool success = CalculateSomething(interesting_value,
                                  kDefaultBaseValue,
                                  kFirstTimeCalling,
                                  null_callback);
```

不要：

注意永远不要用自然语言翻译代码作为注释，要假设读你代码的人 C++比你强:D：

```
// Now go through the b array and make sure that if i occurs,
// the next element is i+1.
...        // Geez.  What a useless comment.
```

7. 标点、拼写和语法（Punctuation, Spelling and Grammar）

留意标点、拼写和语法，写的好的注释比差的要易读的多。

注释一般是包含适当大写和句点（.）的完整的句子，短一点的注释（如代码行尾的注释）可以随意点，依然要注意风格的一致性。完整的句子可读性更好，也可以说明该注释是完整的而不是一点不成熟的想法。

虽然被别人指出该用分号（semicolon）的时候用了逗号（comma）有点尴尬。清晰易读的代码还是很重要的，适当的标点、拼写和语法对此会有所帮助。

8. TODO 注释（TODO Comments）

对那些临时的、短期的解决方案，或已经够好但并不完美的代码使用 TODO 注释。

这样的注释要使用全大写的字符串 TODO，后面括号（parentheses）里加上你的大名、邮件地址等，还可以加上冒号（colon）：目的是可以根据统一的 TODO 格式进行查找：

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
```

如果加上是为了在"将来某一天做某事"，可以加上一个特定的时间（"Fix by November 2005"）或事件（"Remove this code when all clients can handle XML responses."）。

译者：注释也是比较人性化的约定了：

1. 关于注释风格，很多 **C++**的 **coders** 更喜欢行注释，**C coders** 或许对块注释依然情有独钟，或者在文件头大段大段的注释时使用块注释；

2. 文件注释可以炫耀你的成就，也是为了捅了篓子别人可以找你；

3. 注释要言简意赅，不要拖沓冗余，复杂的东西简单化和简单的东西复杂化都是要被鄙视的；

4. 对于 **Chinese coders** 来说，用英文注释还是用中文注释，**it is a problem**，但不管怎样，注释是为了让别人看懂，难道是为了炫耀编程语言之外的你的母语或外语水平吗；

5. 注释不要太乱，适当的缩进才会让人乐意看，但也没有必要规定注释从第几列开始（我自己写代码的时候总喜欢这样），**UNIX/LINUX** 下还可以约定是使用 **tab** 还是 **space**，个人倾向于 **space**；

6. TODO 很不错，有时候，注释确实是为了标记一些未完成的或完成的不尽如人意的地方，这样一搜索，就知道还有哪些活要干，日志都省了。

八、格式

代码风格和格式确实比较随意，但一个项目中所有人遵循同一风格是非常容易的，作为个人未必同意下述格式规则的每一处，但整个项目服从统一的编程风格是很重要的，这样做才能让所有人在阅读和理解代码时更加容易。

1. 行长度（Line Length）

每一行代码字符数不超过 80。

我们也认识到这条规则是存有争议的，但如此多的代码都遵照这一规则，我们感觉一致性更重要。

优点：提倡该原则的人认为强迫他们调整编辑器窗口大小很野蛮。很多人同时并排开几个窗口，根本没有多余空间拓宽某个窗口，人们将窗口最大尺寸加以限定，一致使用 80 列宽，为什么要改变呢？

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，很轻松的可以显示更多代码。

结论：80 个字符是最大值。例外：

1) 如果一行注释包含了超过 80 字符的命令或 URL，出于复制粘贴的方便可以超过 80 字符；

2) 包含长路径的可以超出 80 列，尽量避免；

3) 头文件保护（防止重复包含第一篇）可以无视该原则。

2. 非 ASCII 字符（Non-ASCII Characters）

尽量不使用非 ASCII 字符，使用时必须使用 UTF-8 格式。

哪怕是英文，也不应将用户界面的文本硬编码到源代码中，因此非 ASCII 字符要少用。特殊情况下可以适当包含此类字符，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；更常用的是（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 格式，因为很多工具都可以理解和处理其编码，十六进制编码也可以，尤其是在增强可读性的情况下——如"\xEF\xBB\xBF"是 Unicode 的 zero-width no-break space 字符，以 UTF-8 格式包含在源文件中是不可见的。

3. 空格还是制表位（Spaces vs. Tabs）

只使用空格，每次缩进 2 个空格。

使用空格进行缩进，不要在代码中使用 tabs，设定编辑器将 tab 转为空格。

4. 函数声明与定义（Function Declarations and Definitions）

返回类型和函数名在同一行，合适的话，参数也放在同一行。

函数看上去像这样：

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

如果同一行文本较多，容不下所有参数：

```
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：

1) 返回值总是和函数名在同一行；

2) 左圆括号（open parenthesis）总是和函数名在同一行；

3) 函数名和左圆括号间没有空格；

4) 圆括号与参数间没有空格；

5) 左大括号（open curly brace）总在最后一个参数同一行的末尾处；

6) 右大括号（close curly brace）总是单独位于函数最后一行；

7) 右圆括号（close parenthesis）和左大括号间总是有一个空格；

8) 函数声明和实现处的所有形参名称必须保持一致；

9) 所有形参应尽可能对齐；

10) 缺省缩进为 2 个空格；

11) 独立封装的参数保持 4 个空格的缩进。

如果函数为 const 的，关键字 const 应与最后一个参数位于同一行。

```
// Everything in this function signature fits on a single line
ReturnType FunctionName(Type par) const {
  ...
}

// This function signature requires multiple lines, but
// the const keyword is on the line with the last parameter.
ReturnType ReallyLongFunctionName(Type par1,
                                  Type par2) const {
  ...
}
```

如果有些参数没有用到，在函数定义处将参数名注释起来：

```
// Always have named parameters in interfaces.
class Shape {
 public:
  virtual void Rotate(double radians) = 0;
}

// Always have named parameters in the declaration.
class Circle : public Shape {
 public:
  virtual void Rotate(double radians);
}
```

```
// Comment out unused named parameters in definitions.
void Circle::Rotate(double /*radians*/) {}
// Bad - if someone wants to implement later, it's not clear what the
// variable means.
void Circle::Rotate(double) {}
```

译者注：关于 UNIX/Linux 风格为什么要把左大括号置于行尾（.cc 文件的函数实现处，左大括号位于行首），我的理解是代码看上去比较简约，想想行首除了函数体被一对大括号封在一起之外，只有右大括号的代码看上去确实也舒服；Windows 风格将左大括号置于行首的优点是匹配情况一目了然。

5. 函数调用（Function Calls）

尽量放在同一行，否则，将实参封装在圆括号中。

函数调用遵循如下形式：

```
bool retval = 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 retval = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

如果函数参数比较多，可以出于可读性的考虑每行只放一个参数：

```
bool retval = DoSomething(argument1,
                          argument2,
                          argument3,
                          argument4);
```

如果函数名太长，以至于超过行最大长度，可以将所有参数独立成行：

```
if (...) {
  ...
  ...
  if (...) {
    DoSomethingThatRequiresALongFunctionName(
        very_long_argument1,  // 4 space indent
        argument2,
        argument3,
        argument4);
  }
```

6. 条件语句（Conditionals）

更提倡不在圆括号中添加空格，关键字 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，一种没有。

最常见的是没有空格的格式，那种都可以，还是**一致性**为主。如果你是在修改一个文件，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他文件的格式，还在徘徊的话，就不要加空格了。

```
if (condition) {  // no spaces inside parentheses
  ...  // 2 space indent.
} else {  // The else goes on the same line as the closing brace.
  ...
}
```

如果你倾向于在圆括号内部加空格：

```
if ( condition ) {  // spaces inside parentheses - rare
  ...  // 2 space indent.
} else {  // The else goes on the same line as the closing brace.
  ...
}
```

注意所有情况下 if 和左圆括号间有个空格，右圆括号和左大括号（如果使用的话）间也要有个空格：

```
if(condition)     // Bad - space missing after IF.
if (condition){   // Bad - space missing before {.
if(condition){    // Doubly bad.
if (condition) {  // Good - proper space after IF and before {.
```

有些条件语句写在同一行以增强可读性，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 分支是不允许的：

```
// Not allowed - IF statement on one line when there is an ELSE clause
if (x) DoThis();
else DoThat();
```

通常，单行语句不需要使用大括号，如果你喜欢也无可厚非，也有人要求 if 必须使用大括号：

```
if (condition)
```

```
  DoSomething();  // 2 space indent.

if (condition) {
  DoSomething();  // 2 space indent.
}
```

但如果语句中哪一分支使用了大括号的话，其他部分也必须使用：

```
// Not allowed - curly on IF but not ELSE
if (condition) {
  foo;
} else
  bar;

// Not allowed - curly on ELSE but not IF
if (condition)
  foo;
else {
  bar;
}

// Curly braces around both IF and ELSE required because
// one of the clauses used braces.
if (condition) {
  foo;
} else {
  bar;
}
```

7. 循环和开关选择语句（Loops and Switch Statements）

switch 语句可以使用大括号分块；空循环体应使用{}或 continue。

switch 语句中的 case 块可以使用大括号也可以不用，取决于你的喜好，使用时要依下文所述。

如果有不满足 case 枚举条件的值，要总是包含一个 default（如果有输入值没有 case 去处理，编译器将报警）。如果 default 永不会执行，可以简单的使用 assert：

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

```
      break;
    }
    default: {
      assert(false);
    }
  }
```

空循环体应使用{}或 continue，而不是一个简单的分号：

```
while (condition) {
  // Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // Good - empty body.
while (condition) continue;  // Good - continue indicates no logic.
while (condition);  // Bad - looks like part of do/while loop.
```

8. 指针和引用表达式（Pointers and Reference Expressions）

句点（.）或箭头（->）前后不要有空格，指针/地址操作符（*、&）后不要有空格。

下面是指针和引用表达式的正确范例：

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

注意：

1) 在访问成员时，句点或箭头前后没有空格；

2) 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：

```
// These are fine, space preceding.
char *c;
const string &str;

// These are fine, space following.
char* c;    // but remember to do "char* c, *d, *e, ...;"!
const string& str;
char * c;  // Bad - spaces on both sides of *
const string & str;  // Bad - spaces on both sides of &
```

同一个文件（新建或现有）中起码要保持一致。

译者注：个人比较习惯与变量紧挨的方式。

9. 布尔表达式（Boolean Expressions）

如果一个布尔表达式超过标准行宽（80 字符），如果断行要统一一下。

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) {
  ...
}
```

两个逻辑与（&&）操作符都位于行尾，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。

译者注：个人比较习惯逻辑运算符位于行首，逻辑关系一目了然，各人喜好而已，至于加不加圆括号的问题，如果你对优先级了然于胸的话可以不加，但可读性总是差了些。

10. 函数返回值（Return Values）

return 表达式中不要使用圆括号。

函数返回时不要使用圆括号：

```
return x;  // not return(x);
```

11. 变量及数组初始化（Variable and Array Initialization）

选择=还是()。

需要做二者之间做出选择，下面的形式都是正确的：

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

12. 预处理指令（Preprocessor Directives）

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。

```
// Good - directives at beginning of line
  if (lopsided_score) {
#if DISASTER_PENDING      // Correct -- Starts at beginning of line
    DropEverything();
#endif
```

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

13. 类格式（Class Format）

声明属性依次序是 public:、protected:、private:，每次缩进 1 个空格（译者注，为什么不是两个呢？也有人提倡 private 在前，对于声明了哪些数据成员一目了然，还有人提倡依逻辑关系将变量与操作放在一起，都有道理:-)）。

类声明（对类注释不了解的话，参考第六篇中的**类注释**一节）的基本格式如下：

```
class MyClass : public OtherClass {
 public:      // Note the 1 space indent!
  MyClass();  // Regular 2 space indent.
  explicit MyClass(int var);
  ~MyClass() {}

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

 private:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
  DISALLOW_COPY_AND_ASSIGN(MyClass);
};
```

注意：

1) 所以基类名应在 80 列限制下尽量与子类名放在同一行；

2) 关键词 public:、protected:、private:要缩进 1 个空格（译者注，MSVC 多使用 tab 缩进，且这三个关键词没有缩进）；

3) 除第一个关键词（一般是 public）外，其他关键词前空一行，如果类比较小的话也可以不空；

4) 这些关键词后不要空行；

5) public 放在最前面，然后是 protected 和 private；

6) 关于声明次序参考第三篇**声明次序**一节。

14. 初始化列表（Initializer Lists）

构造函数初始化列表放在同一行或按四格缩进并排几行。

两种可以接受的初始化列表格式：

```
// When it all fits on one line:
MyClass::MyClass(int var) : some_var_(var), some_other_var_(var + 1) {
```

或

```
// When it requires multiple lines, indent 4 spaces, putting the colon on
// the first initializer line:
MyClass::MyClass(int var)
    : some_var_(var),             // 4 space indent
      some_other_var_(var + 1) {  // lined up
  ...
  DoSomething();
  ...
}
```

15. 命名空间格式化（Namespace Formatting）

命名空间内容不缩进。

命名空间不添加额外缩进层次，例如：

```
namespace {

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

}  // namespace
```

不要缩进：

```
namespace {
```

```
  // Wrong.  Indented when it should not be.
  void foo() {
    ...
  }

}  // namespace
```

16. 水平留白（Horizontal Whitespace）

水平留白的使用因地制宜。不要在行尾添加无谓的留白。

普通：

```
void f(bool b) {  // Open braces should always have a space before them.
  ...
int i = 0;  // Semicolons usually have no space before them.
int x[] = { 0 };  // Spaces inside braces for array initialization are
int x[] = {0};    // optional.  If you use them, put them on both sides!
// Spaces around the colon in inheritance and initializer lists.
class Foo : public Bar {
 public:
  // For inline function implementations, put spaces between the braces
  // and the implementation itself.
  Foo(int b) : Bar(), baz_(b) {}  // No spaces inside empty braces.
  void Reset() { baz_ = 0; }  // Spaces separating braces from
implementation.
  ...
```

添加冗余的留白会给其他人编辑时造成额外负担，因此，不要加入多余的空格。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（确信没有其他人在使用）。

循环和条件语句：

```
if (b) {          // Space after the keyword in conditions and loops.
} else {          // Spaces around else.
}
while (test) {}   // There is usually no space inside parentheses.
switch (i) {
for (int i = 0; i < 5; ++i) {
switch ( i ) {    // Loops and conditions may have spaces inside
if ( test ) {     // parentheses, but this is rare.  Be consistent.
for ( int i = 0; i < 5; ++i ) {
for ( ; i < 5 ; ++i) {  // For loops always have a space after the
  ...                   // semicolon, and may have a space before the
                        // semicolon.
```

```
switch (i) {
  case 1:         // No space before colon in a switch case.
    ...
  case 2: break;  // Use a space after a colon if there's code after it.
```

操作符：

```
x = 0;              // Assignment operators always have spaces around
                    // them.
x = -5;             // No spaces separating unary operators and their
++x;                // arguments.
if (x && !y)
  ...
v = w * x + y / z;  // Binary operators usually have spaces around them,
v = w*x + y/z;      // but it's okay to remove spaces around factors.
v = w * (x + z);    // Parentheses should have no spaces inside them.
```

模板和转换：

```
vector<string> x;           // No spaces inside the angle
y = static_cast<char*>(x);  // brackets (< and >), before
                            // <, or between >( in a cast.
vector<char *> x;           // Spaces between type and pointer are
                            // okay, but be consistent.
set<list<string> > x;       // C++ requires a space in > >.
set< list<string> > x;      // You may optionally make use
                            // symmetric spacing in < <.
```

17. 垂直留白（Vertical Whitespace）

垂直留白越少越好。

这不仅仅是规则而是原则问题了：不是非常有必要的话就不要使用空行。尤其是：不要在两个函数定义之间空超过 2 行，函数体头、尾不要有空行，函数体中也不要随意添加空行。

基本原则是：同一屏可以显示越多的代码，程序的控制流就越容易理解。当然，过于密集的代码块和过于疏松的代码块同样难看，取决于你的判断，但通常是越少越好。

函数头、尾不要有空行：

```
void Function() {

  // Unnecessary blank lines before and after

}
```

代码块头、尾不要有空行：

```
while (condition) {
  // Unnecessary blank line after

}
if (condition) {

  // Unnecessary blank line before
}
```

if-else 块之间空一行还可以接受：

```
if (condition) {
  // Some lines of code too small to move to another function,
  // followed by a blank line.

} else {
  // Another block of code
}
```

译者：首先说明，对于代码格式，因人、因系统各有优缺点，但同一个项目中遵循同一标准还是有必要的：

1. 行宽原则上不超过 **80** 列，把 **22** 寸的显示屏都占完，怎么也说不过去；

2. 尽量不使用非 **ASCII** 字符，如果使用的话，参考 **UTF-8** 格式（尤其是 **UNIX/Linux** 下，**Windows** 下可以考虑宽字符），尽量不将字符串常量耦合到代码中，比如独立出资源文件，这不仅仅是风格问题了；

3. UNIX/Linux 下无条件使用空格，**MSVC** 的话使用 **Tab** 也无可厚非；

4. 函数参数、逻辑条件、初始化列表：要么所有参数和函数名放在同一行，要么所有参数并排分行；

5. 除函数定义的左大括号可以置于行首外，包括函数/类/结构体/枚举声明、各种语句的左大括号置于行尾，所有右大括号独立成行；

6. ./->操作符前后不留空格，***/&**不要前后都留，一个就可，靠左靠右依各人喜好；

7. 预处理指令/命名空间不使用额外缩进，类/结构体/枚举/函数/语句使用缩进；

8. 初始化用**=**还是**()**依个人喜好，统一就好；

9. return 不要加**()**；

10. 水平/垂直留白不要滥用，怎么易读怎么来。

九、规则之例外

前面说明的编码习惯基本是强制性的，但所有优秀的规则都允许例外。

1. 现有不统一代码（Existing Non-conformant Code）

对于现有不符合既定编程风格的代码可以网开一面。

当你修改使用其他风格的代码时，为了与代码原有风格保持一致可以不使用本指南约定。如果不放心可以与代码原作者或现在的负责人员商讨，记住，一致性包括原有的一致性。

2. Windows 代码（Windows Code）

Windows 程序员有自己的编码习惯，主要源于 Windows 的一些头文件和其他 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++编码给出一个单独的指导方案。

如果你一直使用 Windows 编码风格的，这儿有必要重申一下某些你可能会忘记的指南（译者注，我怎么感觉像在被洗脑:D）：

1) 不要使用匈牙利命名法(Hungarian notation，如定义整型变量为 iNum)，使用 Google 命名约定，包括对源文件使用.cc 扩展名；

2) Windows 定义了很多原有内建类型的同义词（译者注，这一点，我也很反感），如 DWORD、HANDLE 等等，在调用 Windows API 时这是完全可以接受甚至鼓励的，但还是尽量使用原来的 C++类型，例如，使用 const TCHAR *而不是 LPCTSTR；

3) 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 warnings 当作 errors 处理**；

4) 不要使用#pragma once;作为**包含保护**，使用 C++标准**包含保护**，**包含保护**的文件路径包含到项目树顶层（译者注，#include<prj_name/public/tools.h>）；

5) 除非万不得已，否则不使用任何不标准的扩展，如#pragma 和__declspec，允许使用__declspec(dllimport)和__declspec(dllexport)，但必须通过 DLLIMPORT 和 DLLEXPORT 等宏，以便其他人在共享使用这些代码时容易放弃这些扩展。

在 Windows 上，只有很少一些偶尔可以不遵守的规则：

1) 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，为了执行 COM 或 ATL/WTL 类及其接口时可以使用多重实现继承；

2) 虽然**代码中不应使用异常**，但在 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 以屏蔽异常，你要研究一下是否也屏蔽掉 STL 的异常，如果不屏蔽，开启编译器异常也可以，注意这只是为了编译 STL，自己仍然不要写含异常处理的代码；

3) 通常每个项目的每个源文件中都包含一个名为 StdAfx.h 或 precompile.h 的头文件方便头文件预编译，为了使代码方便与其他项目共享，避免显式包含此文件（precompile.cc 除外），使用编译器选项/FI 以自动包含；

4) 通常名为 resource.h、且只包含宏的资源头文件，不必拘泥于此风格指南。

十、团队合作

参考常识，**保持一致**。

编辑代码时，花点时间看看项目中的其他代码并确定其风格，如果其他代码 if 语句中使用空格，那么你也要使用。如果其中的注释用星号（*）围成一个盒子状，你也这样做：

```
/**********************************
* Some comments are here.
* There may be many lines.
**********************************/
```

编程风格指南的使用要点在于提供一个公共的编码规范，所有人可以把精力集中在实现内容而不是表现形式上。我们给出了全局的风格规范，但局部的风格也很重要，如果你在一个文件中新加的代码和原有代码风格相去甚远的话，这就破坏了文件本身的整体美观也影响阅读，所以要尽量避免。

好了，关于编码风格写的差不多了，代码本身才是更有趣的，尽情享受吧！

Benjy Weinberger
Craig Silverstein
Gregory Eitzmann
Mark Mentovai
Tashana Landray