C++ Code Style Guide Line

背景

C++ 是 Google 大部分开源项目的主要编程语言. 正如每个 C++ 程序员都知道的, C++ 有很多强大的特性, 但这种强大不可避免的导致它走向复杂，使代码更容易产生 bug, 难以阅读和维护.

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

风格, 亦被称作可读性, 也就是指导 C++ 编程的约定. 使用术语 “风格” 有些用词不当, 因为这些习惯远不止源代码文件格式化这么简单.

使代码易于管理的方法之一是加强代码一致性. 让任何程序员都可以快速读懂你的代码这点非常重要. 保持统一编程风格并遵守约定意味着可以很容易根据 “模式匹配” 规则来推断各种标识符的含义. 创建通用, 必需的习惯用语和模式可以使代码更容易理解. 在一些情况下可能有充分的理由改变某些编程风格, 但我们还是应该遵循一致性原则，尽量不这么做.

本指南的另一个观点是 C++ 特性的臃肿. C++ 是一门包含大量高级特性的庞大语言. 某些情况下, 我们会限制甚至禁止使用某些特性. 这么做是为了保持代码清爽, 避免这些特性可能导致的各种问题. 指南中列举了这类特性, 并解释为什么这些特性被限制使用.

Google 主导的开源项目均符合本指南的规定.

注意: 本指南并非 C++ 教程, 我们假定读者已经对 C++ 非常熟悉.

头文件

通常每一个 `.cc` 文件都有一个对应的 `.h` 文件. 也有一些常见例外, 如单元测试代码和只包含 `main()` 函数的 `.cc` 文件.

正确使用头文件可令代码在可读性、文件大小和性能上大为改观.

下面的规则将引导你规避使用头文件时的各种陷阱.

Self-contained 头文件

> 头文件应该能够自给自足（self-contained,也就是可以作为第一个头文件被引入），以 `.h` 结尾。至于用来插入文本的文件，说到底它们并不是头文件，所以应以 `.inc` 结尾。不允许分离出 `-inl.h` 头文件的做法.

所有头文件要能够自给自足。换言之，用户和重构工具不需要为特别场合而包含额外的头文件。详言之，一个头文件要有 `define-guard`，统统包含它所需要的其它头文件，也不要求定义任何特别 symbols.

不过有一个例外，即一个文件并不是 self-contained 的，而是作为文本插入到代码某处。或者，文件内容实际上是其它头文件的特定平台（platform-specific）扩展部分。这些文件就要用 `.inc` 文件扩展名。

如果 `.h` 文件声明了一个模板或内联函数，同时也在该文件加以定义。凡是有用到这些的 `.cc` 文件，就得统统包含该头文件，否则程序可能会在构建中链接失败。不要把这些定义放到分离的 `-inl.h` 文件里（译者注：过去该规范曾提倡把定义放到 -inl.h 里过）。

有个例外：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 `.cc` 文件里。

#define 保护

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

为保证唯一性, 头文件的命名应该基于所在项目源代码树的全路径. 例如, 项目 `foo` 中的头文件 `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_
```

前置声明

> 尽可能地避免使用前置声明。使用 `#include` 包含需要的头文件即可。

定义:

- 所谓「前置声明」（forward declaration）是类、函数和模板的纯粹声明，没伴随着其定义.

优点:

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

缺点:

- 前置声明隐藏了依赖关系，头文件改动时，用户的代码会跳过必要的重新编译过程。
- 前置声明可能会被库的后续更改所破坏。前置声明函数或模板有时会妨碍头文件开发者变动其 API. 例如扩大形参类型，加个自带默认参数的模板形参等等。
- 前置声明来自命名空间 `std::` 的 symbol 时，其行为未定义。
- 很难判断什么时候该用前置声明，什么时候该用 `#include` 。极端情况下，用前置声明代替 `includes` 甚至都会暗暗地改变代码的含义：

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

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

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

- 前置声明了不少来自头文件的 symbol 时，就会比单单一行的 `include` 冗长。
- 仅仅为了能前置声明而重构代码（比如用指针成员代替对象成员）会使代码变得更慢更复杂.

结论：

- 尽量避免前置声明那些定义在其他项目中的实体.
- 函数：总是使用 `#include` .
- 类模板：优先使用 `#include` .

至于什么时候包含头文件，参见 [#include的路径及顺序]。

内联函数

> 只有当函数只有 10 行甚至更少时才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:

- 只要内联的函数体较小, 内联该函数可以令目标代码更加高效. 对于存取函数以及其它函数体比较短, 性能关键的函数, 鼓励使用内联.

缺点:

- 滥用内联将导致程序变得更慢. 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:

- 一个较为合理的经验准则是, 不要内联超过 10 行的函数. 谨慎对待析构函数, 析构函数往往比其表面看起来要更长, 因为有隐含的成员和基类析构函数被调用!
- 另一个实用的经验准则: 内联那些包含循环或 `switch` 语句的函数常常是得不偿失 (除非在大多数情况下, 这些循环或 `switch` 语句从不被执行).
- 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则是想把它的函数体放在类定义内, 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.

#include的路径及顺序

> 使用标准的头文件包含顺序可增强可读性, 避免隐藏依赖: 相关头文件, C 库, C++ 库, 其他库的 `.h`, 本项目内的 `.h`.

项目内头文件应按照项目源代码目录树结构排列, 避免使用 UNIX 特殊的快捷目录 `.` (当前目录) 或 `..` (上级目录). 例如, `google-awesome-project/src/base/logging.h` 应该按如下方式包含:

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

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

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

这种优先的顺序排序保证当 `dir2/foo2.h` 遗漏某些必要的库时， `dir/foo.cc` 或 `dir/foo_test.cc` 的构建会立刻中止。因此这一条规则保证维护这些文件的人们首先看到构建中止的消息而不是维护其他包的人们。

`dir/foo.cc` 和 `dir2/foo2.h` 通常位于同一目录下 (如 `base/basictypes_unittest.cc` 和 `base/basictypes.h`), 但也可以放在不同目录下.

按字母顺序对头文件包含进行二次排序是不错的主意。注意较老的代码可不符合这条规则，要在方便的时候改正它们。

您所依赖的 symbols 被哪些头文件所定义，您就应该包含（include）哪些头文件，[前置声明] 情况除外。比如您要用到 `bar.h` 中的某个 symbol, 哪怕您所包含的 `foo.h` 已经包含了 `bar.h`, 也照样得包含 `bar.h`, 除非 `foo.h` 有明确说明它会自动向您提供 `bar.h` 中的 symbol. 不过，凡是 cc 文件所对应的「相关头文件」已经包含的，就不用再重复包含进其 cc 文件里面了，就像 `foo.cc` 只包含 `foo.h` 就够了，不用再管后者所包含的其它内容。

举例来说, `google-awesome-project/src/foo/internal/fooserver.cc` 的包含次序如下:

```
#include "foo/public/fooserver.h" // 优先位置

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

#include <hash_map>
#include <vector>

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

例外：

有时，平台特定（system-specific）代码需要条件编译（conditional includes），这些代码可以放到其它 includes 之后。当然，您的平台特定代码也要够简练且独立，比如：

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

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

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

作用域

名字空间

> 鼓励在 `.cc` 文件内使用匿名名字空间. 使用具名的名字空间时, 其名称可基于项目名或相对路径. 禁止使用 using 指示（using-directive）。禁止使用内联命名空间（inline namespace）。

定义:

- 名字空间将全局作用域细分为独立的, 具名的作用域, 可有效防止全局作用域的命名冲突.

优点:

- 虽然类已经提供了（可嵌套的）命名轴线, 名字空间在这基础上又封装了一层.
- 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名字空间中, `project1::Foo` 和 `project2::Foo` 作为不同符号自然不会冲突.
- 内联命名空间会自动把内部的标识符放到外层作用域，比如：

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

`X::Y::foo()` 与 `X::foo()` 彼此可代替。内联命名空间主要用来保持跨版本的 ABI 兼容性。

缺点:

- 名字空间具有迷惑性, 因为它们和类一样提供了额外的 (可嵌套的) 命名轴线.
- 命名空间很容易令人迷惑，毕竟它们不再受其声明所在命名空间的限制。内联命名空间只在大型版本控制里有用。
- 在头文件中使用匿名空间导致违背 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.

匿名名字空间

- 在 `.cc` 文件中, 允许甚至鼓励使用匿名名字空间, 以避免运行时的命名冲突:

```
namespace {                             // .cc 文件中

// 名字空间的内容无需缩进
enum { kUNUSED, kEOF, kERROR };         // 经常使用的符号
bool AtEof() { return pos_ == kEOF; }   // 使用本名字空间内的符号 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, 而不是匿名名字空间的成员. 如上例所示, 匿名空间结束时用注释 `// namespace` 标识.

- 不要在 `.h` 文件中使用匿名名字空间.

具名的名字空间

具名的名字空间使用方式如下:

- 用名字空间把文件包含, 以及类的前置声明以外的整个源文件封装起来, 以区别于其它名字空间:

```
// .h 文件
namespace mynamespace {

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

} // namespace mynamespace
```

```
// .cc 文件
namespace mynamespace {

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

} // namespace mynamespace
```

通常的 `.cc` 文件包含更多, 更复杂的细节, 比如引用其他名字空间的类等.

```
#include “a.h”

DEFINE_bool(someflag, false, “dummy flag”);

class C;                    // 全局名字空间中类 C 的前置声明
namespace a { class A; }    // a::A 的前置声明

namespace b {

        ...code for b...    // b 中的代码

} // namespace b
```

- 不要在名字空间 `std` 内声明任何东西, 包括标准库的类前置声明. 在 `std` 名字空间声明实体会导致不确定的问题, 比如不可移植. 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，以保证名字空间下的所有名称都可以正常使用.

```
// 禁止 -- 污染名字空间
using namespace foo;
```

- 在 `.cc` 文件, `.h` 文件的函数, 方法或类中, 可以使用 using 声明。

```
// 允许: .cc 文件中
// .h 文件的话，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, 允许使用名字空间别名.

```
// 允许: .cc 文件中
// .h 文件的话，必须在函数，方法或类的内部使用

namespace fbz = ::foo::bar::baz;

// 在 .h 文件里
namespace librarian {
//以下别名在所有包含了该头文件的文件中生效。
namespace pd_s = ::pipeline_diagnostics::sidetable;

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

注意在 .h 文件的别名对包含了该头文件的所有人可见，所以在公共头文件（在项目外可用）以及它们递归包含的其它头文件里，不要用别名。毕竟原则上公共 API 要尽可能地精简。

- 禁止用内联命名空间

嵌套类

> 当公有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 `namespaces` 内是更好的选择.

定义

- 在一个类内部定义另一个类; 嵌套类也被称为 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 (或成员) 类只被外围类使用时非常有用; 把它作为外围类作用域内的成员, 而不是去污染外部作用域的同名类. 嵌套类可以在外围类中做前置声明, 然后在 `.cc` 文件中定义, 这样避免在外围类的声明中定义嵌套类, 因为嵌套类的定义通常只与实现相关.

缺点:

- 嵌套类只能在外围类的内部做前置声明. 因此, 任何使用了 `Foo::Bar*` 指针的头文件不得不包含类 `Foo` 的整个声明.

结论:

- 不要将嵌套类定义成公有, 除非它们是接口的一部分, 比如, 嵌套类含有某些方法的一组选项.

非成员函数、静态成员函数和全局函数

> 使用静态成员函数或名字空间内的非成员函数, 尽量不要用裸的全局函数.

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, 将非成员函数放在名字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, 当它们需要访问外部资源或具有重要的依赖关系时更是如此.

结论:

- 有时, 把函数的定义同类的实例脱钩是有益的, 甚至是必要的. 这样的函数可以被定义成静态成员, 或是非成员函数. 非成员函数不应依赖于外部变量, 应尽量置于某个名字空间内. 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 `namespaces`。

- 定义在同一编译单元的函数, 被其他编译单元直接调用可能会引入不必要的耦合和链接时依赖; 静态成员函数对此尤其敏感. 可以考虑提取到新类中, 或者将函数置于独立库的名字空间内.
- 如果你必须定义非成员函数, 又只是在 `.cc` 文件中使用它, 可使用匿名 `namespaces` 或 `static` 链接关键字 (如 `static int Foo() {...}`) 限定其作用域.

局部变量

> 将函数变量尽可能置于最小作用域内, 并在变量声明时进行初始化.

C++ 允许在函数的任何位置声明变量. 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. 这使得代码浏览者更容易定位变量声明的位置, 了解变量的类型和初始值. 特别是，应使用初始化的方式替代声明再赋值, 比如:

```
int i;
i = f(); // 坏——初始化和声明分离
```

```
int j = g(); // 好——初始化时声明
```

```
vector<int> v;
v.push_back(1); // 用花括号初始化更好
v.push_back(2);
```

```
vector<int> v = {1, 2}; // 好——v 一开始就初始化
```

注意, GCC 可正确实现了 `for (int i = 0; i < 10; ++i)` (`i` 的作用域仅限 `for` 循环内), 所以其他 `for` 循环中可以重新使用 `i`. 在 `if` 和 `while` 等语句中的作用域声明也是正确的, 如:

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

> warning 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);
}
```

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);
}
```

静态和全局变量

> 禁止使用 `class` 类型的静态或全局变量：它们会导致难以发现的 bug 和不确定的构造和析构函数调用顺序。不过 `constexpr` 变量除外，毕竟它们又不涉及动态初始化或析构。

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

静态变量的构造函数、析构函数和初始化的顺序在 C++ 中是不确定的，甚至随着构建变化而变化，导致难以发现的 bug. 所以除了禁用类类型的全局变量，我们也不允许用函数返回值来初始化 POD 变量，除非该函数不涉及（比如 getenv() 或 getpid() ）不涉及任何全局变量。（函数作用域里的静态变量除外，毕竟它的初始化顺序是有明确定义的，而且只会在指令执行到它的声明那里才会发生。）

同理，全局和静态变量在程序中断时会被析构，无论所谓中断是从 `main()` 返回还是对 `exit()` 的调用。析构顺序正好与构造函数调用的顺序相反。但既然构造顺序未定义，那么析构顺序当然也就不定了。比如，在程序结束时某静态变量已经被析构了，但代码还在跑——比如其它线程——并试图访问它且失败；再比如，一个静态 string 变量也许会在一个引用了前者的其它变量析构之前被析构掉。

改善以上析构问题的办法之一是用 `quick_exit()` 来代替 `exit()` 并中断程序。它们的不同之处是前者不会执行任何析构，也不会执行 `atexit()` 所绑定的任何 handlers. 如果您想在执行 `quick_exit()` 来中断时执行某 handler（比如刷新 log），您可以把它绑定到 `_at_quick_exit()`. 如果您想在 `exit()` 和 `quick_exit()` 都用上该 handler, 都绑定上去。

综上所述，我们只允许 POD 类型的静态变量，即完全禁用 `vector` (使用 C 数组替代) 和 `string` (使用 `const char []`)。

如果您确实需要一个 `class` 类型的静态或全局变量，可以考虑在 `main()` 函数或 `pthread_once()` 内初始化一个指针且永不回收。注意只能用 raw 指针，别用智能指针，毕竟后者的析构函数涉及到上文指出的不定顺序问题。

类

类是 C++ 中代码的基本单元. 显然, 它们被广泛使用. 本节列举了在写一个类时的主要注意事项.

构造函数的职责

> 不要在构造函数中进行复杂的初始化 (尤其是那些有可能失败或者需要调用虚函数的初始化).

定义:

- 在构造函数体中进行初始化操作.

优点:

- 排版方便, 无需担心类是否已经初始化.

缺点:

- 在构造函数中执行操作引起的问题有:
  - 构造函数中很难上报错误, `尽量避免使用异常`.
  - 操作失败会造成对象初始化失败，进入不确定状态.
  - 如果在构造函数内调用了自身的虚函数, 这类调用是不会重定向到子类的虚函数实现. 即使当前没有子类化实现, 将来仍是隐患.
  - 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), 构造函数将先 `main()` 一步被调用, 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:

- 构造函数不得调用虚函数, 如果需要虚函数完成初始化工作,可构造完成后,调用相应的函数.

初始化

> 如果类中定义了成员变量, 则必须在类中为每个类提供初始化函数或定义一个构造函数. 若未声明构造函数, 则编译器会生成一个默认的构造函数, 这有可能导致某些成员未被初始化或被初始化为不恰当的值.

定义:

- `new` 一个不带参数的类对象时, 会调用这个类的默认构造函数. 用 `new[]` 创建数组时, 默认构造函数则总是被调用. 在类成员里面进行初始化是指声明一个成员变量的时候使用一个结构例如 `int _count = 17` 或者 `string _name{"abc"}` 来替代 `int _count` 或者 `string _name` 这样的形式.

优点:

- 用户定义的默认构造函数将在没有提供初始化操作时将对象初始化. 这样就保证了对象在被构造之时就处于一个有效且可用的状态, 同时保证了对象在被创建时就处于一个显然"不可能"的状态, 以此帮助调试.

缺点:

- 对代码编写者来说, 这是多余的工作.
- 如果一个成员变量在声明时初始化又在构造函数中初始化, 有可能造成混乱, 因为构造函数中的值会覆盖掉声明中的值.

结论:

- 简单的初始化用类成员初始化完成, 尤其是当一个成员变量要在多个构造函数里用相同的方式初始化的时候.如果你的类中有成员变量没有在类里面进行初始化, 而且没有提供其它构造函数, 你必须定义一个 (不带参数的) 默认构造函数. 把对象的内部状态初始化成一致 / 有效的值无疑是更合理的方式. 这么做的原因是: 如果你没有提供其它构造函数, 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合理的初始化. 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.

显式构造函数

> 对单个参数的构造函数使用 C++ 关键字 `explicit`.

定义:

- 通常, 如果构造函数只有一个参数, 可看成是一种隐式转换. 打个比方, 如果你定义了 `Foo::Foo(string name)`, 接着把一个字符串传给一个以 `Foo` 对象为参数的函数, 构造函数 `Foo::Foo(string name)` 将被调用, 并将该字符串转换为一个 `Foo` 的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`.
- 除单参数构造函数外, 这一规则也适用于除第一个参数以外的其他参数都具有默认参数的构造函数, 例如 Foo::Foo(string name, int id = 42).

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都必须是显式的. 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: 在极少数情况下, 拷贝构造函数可以不声明成 `explicit`. 作为其它类的透明包装器的类也是特例之一. 类似的例外情况应在注释中明确说明.
- 最后, 只有 std::initializer_list 的构造函数可以是非 explicit, 以允许你的类型结构可以使用列表初始化的方式进行赋值. 例如:

```
MyType m = {1, 2};
MyType MakeMyType() { return {1, 2}; }
TakeMyType({1, 2});
```

可拷贝类型和可移动类型

> 如果你的类型需要, 就让它们支持拷贝 / 移动. 否则, 就把隐式产生的拷贝和移动函数禁用.

定义:

- 可拷贝类型允许对象在初始化时得到来自相同类型的另一对象的值, 或在赋值时被赋予相同类型的另一对象的值, 同时不改变源对象的值. 对于用户定义的类型, 拷贝操作一般通过拷贝构造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 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. 这同时也防止了客户端与实现在非作用域内的交互, 使得它们更容易被理解与维护. 这样的对象可以和需要传值操作的通用 API 一起使用, 例如大多数容器.
- 拷贝 / 移动构造函数与赋值操作一般来说要比它们的各种替代方案, 比如 Clone(), CopyFrom() or 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 = 默认. 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.
- 移动操作允许隐式且高效地将源数据转移出右值对象. 这有时能让代码风格更加清晰.

缺点:

- 许多类型都不需要拷贝, 为它们提供拷贝操作会让人迷惑, 也显得荒谬而不合理. 为基类提供拷贝 / 赋值操作是有害的, 因为在使用它们时会造成对象切割. 默认的或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, 尤其是对那些所用的语言约定或强制要求传引用的程序员来说更是如此. 同时, 这从一定程度上说会鼓励过度拷贝, 从而导致性能上的问题.

结论:

- 如果需要就让你的类型可拷贝 / 可移动. 作为一个经验法则, 如果对于你的用户来说这个拷贝操作不是一眼就能看出来的, 那就不要把类型设置为可拷贝. 如果让类型可拷贝, 一定要同时给出拷贝构造函数和赋值操作的定义. 如果让类型可拷贝, 同时移动操作的效率高于拷贝操作, 那么就把移动的两个操作 (移动构造函数和赋值操作) 也给出定义. 如果类型不可拷贝, 但是移动操作的正确性对用户显然可见, 那么把这个类型设置为只可移动并定义移动的两个操作.
- 建议通过 `= default` 定义拷贝和移动操作. 定义非默认的移动操作目前需要异常. 时刻记得检测默认操作的正确性. 由于存在对象切割的风险, 不要为任何有可能有派生类的对象提供赋值操作或者拷贝 / 移动构造函数 (当然也不要继承有这样的成员函数的类). 如果你的基类需要可复制属性, 请提供一个 `public virtual Clone()` 和一个 `protected` 的拷贝构造函数以供派生类实现.
- 如果你的类不需要拷贝 / 移动操作, 请显式地通过 `= delete` 或其他手段禁用之.

委派和继承构造函数

> 在能够减少重复代码的情况下使用委派和继承构造函数.

定义:

- 委派和继承构造函数是由 C++11 引进为了减少构造函数重复代码而开发的两种不同的特性. 通过特殊的初始化列表语法, 委派构造函数允许类的一个构造函数调用其他的构造函数. 例如:

```
X::X(const string& name) : name_(name) {
        ...
}

X::X() : X("") { }
```

- 继承构造函数允许派生类直接调用基类的构造函数, 一如继承基类的其他成员函数, 而无需重新声明. 当基类拥有多个构造函数时这一功能尤其有用. 例如:

```
class Base {
public:
        Base();
        Base(int n);
        Base(const string& s);
        ...
};

class Derived : public Base {
public:
        using Base::Base;  // Base's constructors are redeclared here.
};
```

- 如果派生类的构造函数只是调用基类的构造函数而没有其他行为时, 这一功能特别有用.

优点:

- 委派和继承构造函数可以减少冗余代码, 提高可读性.
- 委派构造函数对 Java 程序员来说并不陌生.

缺点:

- 使用辅助函数可以预估出委派构造函数的行为.
- 如果派生类和基类相比引入了新的成员变量, 继承构造函数就会让人迷惑, 因为基类并不知道这些新的成员变量的存在.

结论:

- 只在能够减少冗余代码, 提高可读性的前提下使用委派和继承构造函数. 如果派生类有新的成员变量, 那么使用继承构造函数时要小心. 如果在派生类中对成员变量使用了类内部初始化的话, 继承构造函数还是适用的.

结构体 VS. 类

> 仅当只有数据时使用 struct, 其它一概使用 class.

说明:

- 在 C++ 中 struct 和 class 关键字几乎含义一样. 我们为这两个关键字添加我们自己的语义理解, 以便未定义的数据类型选择合适的关键字.
- struct 用来定义包含数据的被动式对象, 也可以包含相关的常量, 但除了存取数据成员之外, 没有别的函数功能. 并且存取功能是通过直接访问位域, 而非函数调用. 除了构造函数, 析构函数, Initialize(), Reset(), Validate() 等类似的函数外, 不能提供其它功能的函数.
- 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, 对于仿函数和 trait 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.

继承

> 使用组合 (composition) 常常比使用继承更合理. 如果使用继承的话, 定义为 `public` 继承.

定义:

- 当子类继承基类时, 子类包含了父基类所有数据及操作的定义. C++ 实践中, 继承主要用于两种场合: 实现继承 (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, 还要区分基类的实际布局.

结论:

- 所有继承必须是 `public` 的. 如果你想使用私有继承, 你应该替换成把基类的实例作为成员对象的方式.
- 不要过度使用实现继承. 组合常常更合适一些. 尽量做到只在 “是一个” 的情况下使用继承: 如果 `Bar` 的确 “是一种” Foo, `Bar` 才能继承 `Foo`.
- 必要的话, 析构函数声明为 `virtual`. 如果你的类有虚函数, 则析构函数也应该为虚函数. 注意 `数据成员在任何情况下都必须是私有的 <....>`_.
- 当重载一个虚函数, 在衍生类中把它明确的声明为 `virtual`. 理论依据: 如果省略 `virtual` 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.

多重继承

> 真正需要用到多重实现继承的情况少之又少. 只在以下情况我们才允许多重继承: 最多只有一个基类是非抽象类; 其它基类都是以 `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 (见 [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 实现 继承的情况少之又少. 多重实现继承看上去是不错的解决方案, 但你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.

> 关于该规则, Windows 下有个 特例.

接口

> 接口是指满足特定条件的类, 这些类以 `Interface` 为后缀 (不强制).

定义:

- 当一个类满足以下要求时, 称之为纯接口:
  - 只有纯虚函数 (“ `=0` “) 和静态函数 (除了下文提到的析构函数).
  - 没有非静态数据成员.
  - 没有定义任何构造函数. 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, 必须为之声明虚析构函数 (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 12.4 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. 另外, 对于 Java 程序员来说, 接口的概念已是深入人心.

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用户.

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.

运算符重载

> 除少数特定环境外，不要重载运算符.

定义:

- 一个类可以定义诸如 `+` 和 `/` 等运算符, 使其可以像内建类型一样直接操作.

优点:

- 使代码看上去更加直观, 类表现的和内建类型 (如 `int`) 行为一致. 重载运算符使 `Equals()`, `Add()` 等函数名黯然失色. 为了使一些模板函数正确工作, 你可能必须定义操作符.

缺点:

- 虽然操作符重载令代码更加直观, 但也有一些不足:
  - 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.
  - 更难定位重载运算符的调用点, 查找 `Equals()` 显然比对应的 `==` 调用点要容易的多.
  - 有的运算符可以对指针进行操作, 容易导致 bug. `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;
- 重载还有令你吃惊的副作用. 比如, 重载了 `operator&` 的类不能被前置声明.

结论:

- 一般不要重载运算符. 尤其是赋值操作 (`operator=`) 比较诡异, 应避免重载. 如果需要的话, 可以定义类似 `Equals()`, `CopyFrom()` 等函数.
- 然而, 极少数情况下可能需要重载运算符以便与模板或 “标准” C++ 类互操作 (如 `operator<<(ostream&, const T&)`). 只有被证明是完全合理的才能重载, 但你还是要尽可能避免这样做. 尤其是不要仅仅为了在 STL 容器中用作键值就重载 `operator==` 或 `operator<`; 相反, 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, 你可以这么做, 记得别忘了在文档中说明原因.

参考 拷贝构造函数 和 [函数重载].

存取控制

> 将 所有 数据成员声明为 `private`, 并根据需要提供相应的存取函数. 例如, 某个名为 `foo_` 的变量, 其取值函数是 `int foo() const`. 还可能需要一个赋值函数 `foo(int v)`.
>
> 特例是, 静态常量数据成员 (一般写做 kFoo) 不需要是私有成员.
>
> 一般在头文件中把存取函数定义成内联函数.
>
> 参考 [继承] 和 [函数命名]

声明顺序

> 在类中使用特定的声明顺序: `public:` 在 `private:` 之前, 成员函数在数据成员 (变量) 前;

- 类的访问控制区段的声明顺序依次为: `public:`, `protected:`, `private:`. 如果某区段没内容, 可以不声明.
- 每个区段内的声明通常按以下顺序:
    - `typedefs` 和枚举
    - 常量
    - 构造函数
    - 析构函数
    - 成员函数, 含静态成员函数
    - 数据成员, 含静态数据成员
- 友元声明应该放在 private 区段. 如果用宏 DISALLOW_COPY_AND_ASSIGN 禁用拷贝和赋值, 应当将其置于 private 区段的末尾, 也即整个类声明的末尾. 参见可拷贝类型和可移动类型.
- `.cc` 文件中函数的定义应尽可能和声明顺序一致.
- 不要在类定义中内联大型函数. 通常, 只有那些没有特别意义或性能要求高, 并且是比较短小的函数才能被定义为内联函数. 更多细节参考 [内联函数].

编写简短函数

> 倾向编写简短, 凝练的函数.

- 我们承认长函数有时是合理的, 因此并不硬性限制函数的长度. 如果函数超过 40 行, 可以思索一下能不能在不影响程序结构的前提下对其进行分割.
- 即使一个长函数现在工作的非常好, 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 bug. 使函数尽量简短, 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段代码, 考虑将其分割为更加简短并易于管理的若干函数.

其他 C++ 特性

引用参数

> 所有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, 参数必须为指针, 如 `int foo(int *pval)`. 在 C++ 中, 函数还可以声明引用参数: `int foo(int &val)`.

优点:

- 定义引用参数防止出现 `(*pval)++` 这样丑陋的代码. 像拷贝构造函数这样的应用也是必需的. 而且更明确, 不接受 `NULL` 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);
```

- 事实上这在 Google Code 是一个硬性约定: 输入参数是值参或 `const` 引用, 输出参数为指针. 输入参数可以是 `const` 指针, 但决不能是非 `const` 的引用参数，除非用于交换，比如 `swap()`.
- 有时候，在输入形参中用 `const T*` 指针比 `const T&` 更明智。比如：
    - 您会传 null 指针。
    - 函数要把指针或对地址的引用赋值给输入形参。
- 总之大多时候输入形参往往是 `const T&`. 若用 `const T*` 说明输入另有处理。所以若您要用 `const T*`, 则应有理有据，否则会害得读者误解。

右值引用

> 只在定义移动构造函数与移动赋值操作时使用右值引用. 不要使用 `std::forward`.

定义:

- 右值引用是一种只能绑定到临时对象的引用的一种, 其语法与传统的引用语法相似. 例如, `void f(string&& s)`; 声明了一个其参数是一个字符串的右值引用的函数.

优点:

- 用于定义移动构造函数 (使用类的右值引用进行构造的函数) 使得移动一个值而非拷贝之成为可能. 例如, 如果 `v1` 是一个 `vector<string>`, 则 `auto v2(std::move(v1))` 将很可能不再进行大量的数据复制而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.
- 右值引用能实现可移动但不可拷贝的类型, 这一特性对那些在拷贝方面没有实际需求, 但有时又需要将它们作为函数参数传递或塞入容器的类型很有用.
- 要高效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, 不要使用 `std::forward` 功能函数. 你可能会使用 `std::move` 来表示将值从一个对象移动而不是复制到另一个对象.

函数重载

> 若要用好函数重载，最好能让读者一看调用点（call site）就胸有成竹，不用花心思猜测调用的重载函数到底是哪一种。该规则适用于构造函数。

定义:

- 你可以编写一个参数类型为 `const string&` 的函数, 然后用另一个参数类型为 `const char*` 的函数重载它:

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

优点:

- 通过重载参数不同的同名函数, 令代码更加直观. 模板化代码需要重载, 同时为使用者带来便利.

缺点:

- 如果函数单单靠不同的参数类型而重载（acgtyrant 注：这意味着参数数量不变），读者就得十分熟悉 C++ 五花八门的匹配规则，以了解匹配过程具体到底如何。另外，当派生类只重载了某个函数的部分变体，继承语义容易令人困惑。

结论:

- 如果您打算重载一个函数, 可以试试改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()`.

缺省参数

> 我们不允许使用缺省函数参数，少数极端情况除外。尽可能改用函数重载。

优点:

- 当您有依赖缺省参数的函数时，您也许偶尔会修改修改这些缺省参数。通过缺省参数，不用再为个别情况而特意定义一大堆函数了。与函数重载相比，缺省参数语法更为清晰，代码少，也很好地区分了「必选参数」和「可选参数」。

缺点:

- 缺省参数会干扰函数指针，害得后者的函数签名（function signature）往往对不上所实际要调用的函数签名。即在一个现有函数添加缺省参数，就会改变它的类型，那么调用其地址的代码可能会出错，不过函数重载就没这问题了。此外，缺省参数会造成臃肿的代码，毕竟它们在每一个调用点（call site）都有重复（acgtyrant 注：我猜可能是因为调用函数的代码表面上看来省去了不少参数，但编译器在编译时还是会在每一个调用代码里统统补上所有默认实参信息，造成大量的重复）。函数重载正好相反，毕竟它们所谓的「缺省参数」只会出现在函数定义里。

结论:

- 由于缺点并不是很严重，有些人依旧偏爱缺省参数胜于函数重载。所以除了以下情况，我们要求必须显式提供所有参数（acgtyrant 注：即不能再通过缺省参数来省略参数了）。
  - 其一，位于 `.cc` 文件里的静态函数或匿名空间函数，毕竟都只能在局部文件里调用该函数了。
  - 其二，可以在构造函数里用缺省参数，毕竟不可能取得它们的地址。
  - 其三，可以用来模拟变长数组。

```
// 通过空 AlphaNum 以支持四个形参
string StrCat(const AlphaNum &a,
const AlphaNum &b = gEmptyAlphaNum,
const AlphaNum &c = gEmptyAlphaNum,
const AlphaNum &d = gEmptyAlphaNum);
```

变长数组和 alloca()

> 我们不允许使用变长数组和 `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, 它们根据数据大小动态分配堆栈内存, 会引起难以发现的内存越界 bugs: “在我的机器上运行的好好的, 发布后却莫名其妙的挂掉了”.

结论:

- 改用更安全的分配器（allocator），就像 `std::vector` 或 `std::unique_ptr<T[]>`.

友元

> 我们允许合理的使用友元类及友元函数.

- 通常友元应该定义在同一文件内, 避免代码读者跑到其它文件查找使用该私有成员的类. 经常用到友元的一个地方是将 `FooBuilder` 声明为 `Foo` 的友元, 以便 `FooBuilder` 正确构造 `Foo` 的内部状态, 而无需将该状态暴露出来. 某些情况下, 将一个单元测试类声明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, 尤其是如果你只允许另一个类访问该类的私有成员时. 当然, 大多数类都只应该通过其提供的公有成员进行互操作.

异常

> 我们不使用 C++ 异常.

优点:

- 异常允许应用高层决定如何处理在底层嵌套函数中「不可能发生」的失败（failures），不用管那些含糊且容易出错的错误代码（acgtyrant 注：error code, 我猜是 C 语言函数返回的非零 int 值）。
- 很多现代语言都用异常。引入异常使得 C++ 与 Python, Java 以及其它类 C++ 的语言更一脉相承。
- 有些第三方 C++ 库依赖异常，禁用异常就不好用了。
- 异常是处理构造函数失败的唯一途径。虽然可以用工厂函数（acgtyrant 注：factory function, 出自 C++ 的一种设计模式，即「简单工厂模式」）或 `Init()` 方法代替异常, but these require heap allocation or a new “invalid” state, respectively.
- 在测试框架里很好用。

缺点:

- 在现有函数中添加 `throw` 语句时，您必须检查所有调用点。要么让所有调用点统统具备最低限度的异常安全保证，要么眼睁睁地看异常一路欢快地往上跑，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`, 否则会没妥善清理好。

- 还有更常见的，异常会彻底扰乱程序的执行流程并难以判断，函数也许会在您意料不到的地方返回。您或许会加一大堆何时何处处理异常的规定来降低风险，然而开发者的记忆负担更重了。
- 异常安全需要RAII和不同的编码实践. 要轻松编写出正确的异常安全代码需要大量的支持机制. 更进一步地说, 为了避免读者理解整个调用表, 异常安全必须隔绝从持续状态写到 “提交” 状态的逻辑. 这一点有利有弊 (因为你也许不得不为了隔离提交而混淆代码). 如果允许使用异常, 我们就不得不时刻关注这样的弊端, 即使有时它们并不值得.
- 启用异常会增加二进制文件数据，延长编译时间（或许影响小），还可能加大地址空间的压力。
- 滥用异常会变相鼓励开发者去捕捉不合时宜，或本来就已经没法恢复的「伪异常」。比如，用户的输入不符合格式要求时，也用不着抛异常。如此之类的伪异常列都列不完。

结论:

- 从表面上看来，使用异常利大于弊, 尤其是在新项目中. 但是对于现有代码, 引入异常会牵连到所有相关代码. 如果新项目允许异常向外扩散, 在跟以前未使用异常的代码整合时也将是个麻烦. 因为 Google 现有的大多数 C++ 代码都没有异常处理, 引入带有异常处理的新代码相当困难.
- 鉴于 Google 现有代码不接受异常, 在现有代码中使用异常比在新项目中使用的代价多少要大一些. 迁移过程比较慢, 也容易出错. 我们不相信异常的使用有效替代方案, 如错误代码, 断言等会造成严重负担.
- 我们并不是基于哲学或道德层面反对使用异常, 而是在实践的基础上. 我们希望在 Google 使用我们自己的开源项目, 但项目中使用异常会为此带来不便, 因此我们也建议不要在 Google 的开源项目中使用异常. 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, 有个 `特例` .

运行时类型识别

> 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. 它通过使用 `typeid` 或者 `dynamic_cast` 完成.

优点:

- RTTI 的标准替代 (下面将描述) 需要对有问题的类层级进行修改或重构. 有时这样的修改并不是我们所想要的, 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. 如果你需要在运行期间确定一个对象的类型, 这通常说明你需要考虑重新设计你的类.
- 随意地使用 RTTI 会使你的代码难以维护. 它使得基于类型的判断树或者 switch 语句散布在代码各处. 如果以后要进行修改, 你就必须检查它们.

结论:

- RTTI 有合理的用途但是容易被滥用, 因此在使用时请务必注意. 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. 如果你的代码需要根据不同的对象类型执行不同的行为的话, 请考虑用以下的两种替代方案之一查询类型:
- 虚函数可以根据子类类型的不同而执行不同代码. 这是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, 使用 dynamic_cast 也是一种替代方案.
- 基于类型的判断树是一个很强的暗示, 它说明你的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, 你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, 比如带类型标签的类继承体系. 而且, 这些方案会掩盖你的真实意图.

类型转换

> 使用 C++ 的类型转换, 如 `static_cast<>()`. 不要使用 `int y = (int)x` 或 `int y = int(x)` 等转换方式;

定义:

- C++ 采用了有别于 C 的类型转换机制, 对转换操作进行归类.

优点:

- C 语言的类型转换问题在于模棱两可的操作; 有时是在做强制转换 (如 `(int)3.5`), 有时是在做类型转换 (如 `(int)"hello"`). 另外, C++ 的类型转换在查找时更醒目.

缺点:

- 恶心的语法.

结论:

- 不要使用 C 风格类型转换. 而应该使用 C++ 风格.
    - 用 `static_cast` 替代 C 风格的值转换, 或某个类指针需要明确的向上转换为父类指针时.
    - 用 `const_cast` 去掉 `const` 限定符.
    - 用 `reinterpret_cast` 指针类型和整型或其它指针之间进行不安全的相互转换. 仅在你对所做一切了然于心时使用.
- 至于 `dynamic_cast` 参见 [运行时类型识别].

流

> 只在记录日志时使用流.

定义:

- 流用来替代 `printf()` 和 `scanf()`.

优点:

- 有了流, 在打印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). 流的构造和析构函数会自动打开和关闭对应的文件.

缺点:

- 流使得 `pread()` 等功能函数很难执行. 如果不使用 `printf` 风格的格式化字符串, 某些格式化操作 (尤其是常用的格式字符串 `%.*s`) 用流处理性能是很低的. 流不支持字符串操作符重新排序 (%1s), 而这一点对于软件国际化很有用.

结论:

- 不要使用流, 除非是日志接口需要. 使用 `printf` 之类的代替.
- 使用流还有很多利弊, 但代码一致性胜过一切. 不要在代码中使用流.

拓展讨论:

- 对这一条规则存在一些争论, 这儿给出点深层次原因. 回想一下唯一性原则 (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 处都保持一致. 因此, 我们不希望用户来决定是使用流还是 `printf + read/write`. 相反, 我们应该决定到底用哪一种方式. 把日志作为特例是因为日志是一个非常独特的应用, 还有一些是历史原因.
- 流的支持者们主张流是不二之选, 但观点并不是那么清晰有力. 他们指出的流的每个优势也都是其劣势. 流最大的优势是在输出时不需要关心打印对象的类型. 这是一个亮点. 同时, 也是一个不足: 你很容易用错类型, 而编译器不会报警. 使用流时容易造成的这类错误:

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

- 由于 `<<` 被重载, 编译器不会报错. 就因为这一点我们反对使用操作符重载.
- 有人说 `printf` 的格式化丑陋不堪, 易读性差, 但流也好不到哪儿去. 看看下面两段代码吧, 实现相同的功能, 哪个更清晰?

```
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更紧凑, 而不是添加一些别人需要学习的新装备.
- 每一种方式都是各有利弊, “没有最好, 只有更适合”. 简单性原则告诫我们必须从中选择其一, 最后大多数决定采用 `printf + read/write`.

前置自增和自减

> 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, 拷贝的代价是比较大的. 既然两种自增方式实现的功能一样, 为什么不总是使用前置自增呢?

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. 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), 两种都无所谓. 对迭代器和模板类型, 使用前置自增 (自减).

`const` 用法

> 我们强烈建议你在任何可能的情况下都要使用 `const` . 此外有时改用 C++11 推出的 constexpr 更好。

定义:

- 在声明的变量或参数前加上关键字 `const` 用于指明变量值不可被篡改 (如 `const int foo`). 为类中的函数加上 `const` 限定符表明该函数不会修改类成员变量的状态 (如 `class Foo { int Bar(char c) const; };`).

优点:

- 大家更容易理解如何使用变量. 编译器可以更好地进行类型检测, 相应地, 也能生成更好的代码. 人们对编写正确的代码更加自信, 因为他们知道所调用的函数被限定了能或不能修改变量值. 即使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:

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

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` :
    - 如果函数不会修改传你入的引用或指针类型参数, 该参数应声明为 `const` .
    - 尽可能将函数声明为 `const` . 访问函数应该总是 `const` . 其他不会修改任何数据成员, 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` .
- 然而, 也不要发了疯似的使用 `const` . 像 `const int * const * const x;` 就有些过了, 虽然它非常精确的描述了常量 `x` . 关注真正有帮助意义的信息: 前面的例子写成 `const int** x` 就够了.
- 关键字 `mutable` 可以使用, 但是在多线程中是不安全的, 使用时首先要考虑线程安全.
- `const` 的位置, 有人喜欢 `int const *foo` 形式, 不喜欢 `const int* foo` , 他们认为前者更一致因此可读性也更好: 遵循了 `const` 总位于其描述的对象之后的原则. 但是一致性原则不适用于此, “不要过度使用” 的声明可以取消大部分你原本想保持的一致性. 将 `const` 放在前面才更易读, 因为在自然语言中形容词 (`const`) 是在名词 (`int`) 之前.
- 这是说, 我们提倡但不强制 `const` 在前. 但要保持代码的一致性!

`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:

- 如今 constexpr 就可以定义浮点式的真・常量，不用再依赖字面值了；也可以定义用户自定义类型上的常量；甚至也可以定义函数调用所返回的常量。

缺点:

- 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，挺麻烦的；Current restrictions on what is allowed in constexpr functions and constructors may invite obscure workarounds in these definitions.

结论:

- 靠 constexpr 特性，方才实现了 C++ 在接口上打造真正常量机制的可能。好好用 constexpr 来定义真・常量以及支持常量的函数。Avoid complexifying function definitions to enable their use with constexpr. 千万别痴心妄想地想靠 constexpr 来强制代码「内联」。

整型

> C++ 内建整型中, 仅使用 `int`. 如果程序中需要不同大小的变量, 可以使用 `<stdint.h>` 中长度精确的整型, 如 `int16_t`.如果您的变量可能不小于 2^31 (2GiB), 就用 64 位变量比如 `int64_t`. 此外要留意，哪怕您的值并不会超出 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, 我们常常会使用 `int`, 如循环计数. 在类似的情况下使用原生类型 `int`. 你可以认为 `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, 除非你是在表示一个位组而不是一个数值, 或是你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.
- 如果您的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（acgtyrant 注：integer promotions, 比如 `int` 与 `unsigned int` 运算时，前者被提升为 `unsigned int` 而有可能溢出），总有意想不到的后果。
- 关于无符号整数:
    - 有些人, 包括一些教科书作者, 推荐使用无符号类型表示非负数. 这种做法试图达到自我文档化. 但是, 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你的意料.

- 因此, 使用断言来指出变量为非负数, 而不是使用无符号型!

64 位下的可移植性

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

- 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. 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 规定 `%zu` |
+————————-+————————+————————————+————————+
| `ptrdiff_t` | `%d` | `%"PRIdS"` | C99 规定 `%zd` |
+————————-+————————+————————————+————————+

- 注意 `PRI*` 宏会被编译器扩展为独立字符串. 因此如果使用非常量的格式化字符串, 需要将宏的值而不是宏名插入格式中. 使用 `PRI*` 宏同样可以在 `%` 后包含长度指示符. 例如, `printf("x = %30"PRIuS"\n", x)` 在 32 位 Linux 上将被展开为 `printf("x = %30" "u" "\n", x)`, 编译器当成 `printf("x = %30u\n", x)` 处理 (Yang.Y 注: 这在 MSVC 6.0 上行不通, VC 6 编译器不会自动把引号间隔的多个字符串连接一个长字符串).
- 记住 `sizeof(void *) != sizeof(int)`. 如果需要一个指针大小的整数要用 `intptr_t`.
- 你要非常小心的对待结构体对齐, 尤其是要持久化到磁盘上的结构体 (Yang.Y 注: 持久化 - 将数据按字节流顺序保存在磁盘文件或数据库中). 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;
```

- 如果你确实需要 32 位和 64 位系统具有不同代码, 可以使用 `#ifdef _LP64` 指令来切分 32/64 位代码. (尽量不要这么做, 如果非用不可, 尽量使修改局部化)

预处理宏

> 使用宏时要非常谨慎, 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. 这可能会导致异常行为, 尤其因为宏具有全局作用域.
- 值得庆幸的是, C++ 中, 宏不像在 C 中那么必不可少. 以往用宏展开性能关键的代码, 现在可以用内联函数替代. 用宏表示常量可被 `const` 变量代替. 用宏 “缩写” 长变量名可被引用代替. 用宏进行条件编译... 这个, 千万别这么做, 会令测试更加痛苦 (`#define` 防止头文件重包含当然是个特例).
- 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 (如用 `#` 字符串化, 用 `##` 连接等等). 但在使用前, 仔细考虑一下能不能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; 如果你要宏, 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, 不然也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。

0, `nullptr` 和 `NULL`

> 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.

sizeof

> 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. 您或许会用 `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;
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;
}
```

auto

> 用 `auto` 绕过烦琐的类型名，只要可读性好就继续用，别用在局部变量之外的地方。

定义：

- C++11 中，若变量被声明成 `auto`，那它的类型就会被自动匹配成初始化表达式的类型。您可以用 `auto` 来复制初始化或绑定引用。

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

优点：

- C++ 类型名有时又长又臭，特别是涉及模板或命名空间的时候。就像：

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

- 返回类型好难读，代码目的也不够一目了然。重构其：

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

- 好多了。
- 没有 `auto` 的话，我们不得不在同一个表达式里写同一个类型名两次，无谓的重复，就像：

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

- 有了 auto, 可以更方便地用中间变量，显式编写它们的类型轻松点。

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

- 看不出其类型是啥，x 的类型声明恐怕远在几百行之外了。
- 程序员必须会区分 `auto` 和 `const auto&` 的不同之处，否则会复制错东西。
- auto 和 C++11 列表初始化的合体令人摸不着头脑：

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

- 它们不是同一回事——`x` 是 `int`，`y` 则是 `std::initializer_list<int>`. 其它一般不可见的代理类型（acgtyrant 注：normally-invisible proxy types, 它涉及到 C++ 鲜为人知的坑：Why is vector not a STL container）也有大同小异的陷阱。
- 如果在接口里用 `auto`，比如声明头文件里的一个常量，那么只要仅仅因为程序员一时修改其值而导致类型变化的话——API 要翻天覆地了。

结论：

- `auto` 只能用在局部变量里用。别用在文件作用域变量，命名空间作用域变量和类数据成员里。永远别列表初始化 `auto` 变量。
- `auto` 还可以和 C++11 特性「尾置返回类型（trailing return type）」一起用，不过后者只能用在 lambda 表达式里。

列表初始化

> 你可以用列表初始化。

- 早在 C++03 里，聚合类型（aggregate types）就已经可以被列表初始化了，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被列表初始化。示范如下：

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

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

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

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

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

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

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

- 用户自定义类型也可以定义接收 `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。
        // 得以值传递。
        MyType(std::initializer_list<int> init_list) {
                for (int i : init_list) append(i);
        }
        MyType& operator=(std::initializer_list<int> init_list) {
                clear();
                for (int i : init_list) append(i);
        }
};
MyType m{2, 3, 5, 7};
```

- 最后，列表初始化也适用于常规数据类型的构造，哪怕没有接收 `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（explict），您就不能用 `= {}` 了。
MyOtherType m{"b"};
```

- 千万别直接列表初始化 auto 变量，看下一句，估计没人看得懂：

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

```
auto d = double{1.23};  // 善哉 -- d 即为 double，并非 std::initializer_list.
```

- 至于格式化，参见 `braced-initializer-list-format`.

Lambda 表达式

> 适当使用 lambda 表达式。别用默认 lambda 捕获，所有捕获都要显式写出来。

定义：

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

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

- C++11 首次提出 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 最简易，可读性也好。
- Lambdas, `std::functions` 和 `std::bind` 可以搭配成通用回调机制（general purpose callback mechanism）；写接收有界函数为参数的函数也很容易了。

缺点：

- Lambdas 的变量捕获略旁门左道，可能会造成悬空指针。
- Lambdas 可能会失控；层层嵌套的匿名函数难以阅读。

结论：

- 按 format 小用 lambda 表达式怡情。
- 禁用默认捕获，捕获都要显式写出来。打比方，比起 `<a href="int x">=</a> {return x + n;}`，您该写成 `<a href="int x">n</a> {return x + n;}` 才对，这样读者也好一眼看出 `n` 是被捕获的值。
- 匿名函数始终要简短，如果函数体超过了五行，那么还不如起名（acgtyrant 注：即把 lambda 表达式赋值给对象），或改用函数。
- 如果可读性更好，就显式写出 lambd 的尾置返回类型，就像auto.

模板元编程

Boost 库

> 只使用 Boost 中被认可的库.

定义:

- Boost 库集 是一个广受欢迎, 经过同行鉴定, 免费开源的 C++ 库集.

优点:

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

缺点:

- 某些 Boost 库提倡的编程实践可读性差, 比如元编程和其他高级模板技术, 以及过度 “函数化” 的编程风格.

结论:

- 为了向阅读和维护代码的人员提供更好的可读性, 我们只允许使用 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` , except serialization (`adj_list_serialize.hpp`) and parallel/distributed algorithms and data structures(`boost/graph/parallel/*` and `boost/graph/distributed/*`)
    - Property Map : `boost/property_map.hpp`
    - The part of Iterator that deals with defining iterators: `boost/iterator/iterator_adaptor.hpp` , `boost/iterator/iterator_facade.hpp` , and `boost/function_output_iterator.hpp`
    - The part of Polygon that deals with Voronoi diagram construction and doesn't depend on the rest of Polygon: `boost/polygon/voronoi_builder.hpp` , `boost/polygon/voronoi_diagram.hpp` , and `boost/polygon/voronoi_geometry_type.hpp`
    - Bimap : `boost/bimap`
    - Statistical Distributions and Functions : `boost/math/distributions`
    - Multi-index : `boost/multi_index`
    - Heap : `boost/heap`
    - The flat containers from Container: `boost/container/flat_map` , and `boost/container/flat_set`
- 我们正在积极考虑增加其它 Boost 特性, 所以列表中的规则将不断变化.
- 以下库可以用，但由于如今已经被 C++ 11 标准库取代，不再鼓励：
    - Pointer Container : `boost/ptr_container` , 改用 std::unique_ptr
    - Array : `boost/array.hpp` , 改用 std::array

C++11

> 适当用 C++11（前身是 C++0x）的库和语言扩展，在贵项目用 C++11 特性前三思可移植性。

定义：

- C++11 有众多语言和库上的变革。

优点：

- 在二〇一四年八月之前，C++11 一度是官方标准，被大多 C++ 编译器支持。它标准化很多我们早先就在用的 C++ 扩展，简化了不少操作，大大改善了性能和安全。

缺点：

- C++11 相对于前身，复杂极了：1300 页 vs 800 页！很多开发者也不怎么熟悉它。于是从长远来看，前者特性对代码可读性以及维护代价难以预估。我们说不准什么时候采纳其特性，特别是在被迫依赖老实工具的项目上。
- 和 [Boost 库] 一样，有些 C++11 扩展提倡实则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，容易招致困惑以及迁移代价。
- C++11 特性除了个别情况下，可以用一用。除了本指南会有不少章节会加以讨若干 C++11 特性之外，以下特性最好不要用：
  - 尾置返回类型，比如用 `auto foo() -> int` 代替 `int foo()`. 为了兼容于现有代码的声明风格。
  - 编译时合数 `<ratio>`, 因为它涉及一个重模板的接口风格。
  - `<cfenv>` 和 `<fenv.h>` 头文件，因为编译器尚不支持。
  - 默认 lambda 捕获。

命名约定

- 最重要的一致性规则是命名管理. 命名风格快速获知名字代表是什么东东: 类型? 变量? 函数? 常量? 宏 … ? 甚至不需要去查找类型声明. 我们大脑中的模式匹配引擎可以非常可靠的处理这些命名规则.
- 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, 所以不管你怎么想, 规则总归是规则.

通用命名规则

> 函数命名，变量命名，文件命名要有描述性；少用缩写。

- 尽可能给有描述性的命名，别心疼空间，毕竟让代码易于新读者理解很重要。不要用只有项目开发者能理解的缩写，也不要通过砍掉几个字母来缩写单词。

```
int price_count_reader;    // 无缩写
int num_errors;            // “num” 本来就很常见
int num_dns_connections;   // 人人都知道 “DNS” 是啥
```

```
int n;                     // 莫名其妙。
int nerr;                  // 怪缩写。
int n_comp_conns;          // 怪缩写。
int wgc_connections;       // 只有贵团队知道是啥意思。
int pc_reader;             // "pc" 有太多可能的解释了。
int cstmr_id;              // 有删减若干字母。
```

文件命名

> 文件名要全部小写, 可以包含下划线 (`_`) 或连字符 (`-`). 按项目约定来. 如果并没有项目约定，”_” 更好。

- 可接受的文件命名:
  - my_useful_class.cc
  - my-useful-class.cc
  - myusefulclass.cc
  - muusefulclass_test.cc // `_unittest` 和 `_regtest` 已弃用。
- C++ 文件要以 `.cc` 结尾, 头文件以 `.h` 结尾. 专门插入文本的文件则以 `.inc` 结尾，参见[Self-contained 头文件]。

- 不要使用已经存在于 `/usr/include` 下的文件名 (Yang.Y 注: 即编译器搜索系统头文件的路径), 如 `db.h`.
- 通常应尽量让文件名更加明确. `http_server_logs.h` 就比 `logs.h` 要好. 定义类时文件名一般成对出现, 如 `foo_bar.h` 和 `foo_bar.cc`, 对应于类 `FooBar`.

内联函数必须放在 `.h` 文件中. 如果内联函数比较短, 就直接放在 `.h` 中.

类型命名

- 类型名称的每个单词首字母均大写, 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 —— 均使用相同约定. 例如:

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

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

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

变量命名

> 变量名一律小写, 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_`.

- 普通变量命名:

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

```
string tableName;   // 差 - 混合大小写。
```

- 类数据成员：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, 但要接下划线。

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

- 结构体变量：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, 不用像类那样接下划线:

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

- 结构体与类的讨论参考 [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, 但如果你要用, 可以用 `g_` 或其它标志作为前缀, 以便更好的区分局部变量.

常量命名

> 在全局或类里的常量名称前加 `k` : kDaysInAWeek. 且除去开头的 `k` 之外每个单词开头字母均大写。

- 所有编译时常量, 无论是局部的, 全局的还是类中的, 和其他变量稍微区别一下. `k` 后接大写字母开头的单词:

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

- 这规则适用于编译时的局部作用域常量，不过要按变量规则来命名也可以。

函数命名

> 常规函数使用大小写混合, 取值和设值函数则要求与变量名匹配: `MyExcitingFunction()`, `MyExcitingMethod()`, `my_exciting_member_variable()`, `set_my_exciting_member_variable()`.

- 常规函数:
    - 函数名的每个单词首字母大写, 没有下划线。
    - 如果您的某函数出错时就要直接 crash, 那么就在函数名加上 OrDie. 但这函数本身必须集成在产品代码里，且平时也可能会出错。

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

- 取值和设值函数:
    - 取值（Accessors）和设值（Mutators）函数要与存取的变量名匹配. 这儿摘录一个类, `num_entries_` 是该类的实例变量:

```
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_;
};
```

- 其它非常短小的内联函数名也可以用小写字母, 例如. 如果你在循环中调用这样的函数甚至都不用缓存其返回值, 小写命名就可以接受.

名字空间命名

> 名字空间用小写字母命名, 并基于项目名称和目录结构: `google_awesome_project`.

- 关于名字空间的讨论和如何命名, 参考 [名字空间] 一节.

枚举命名

> 枚举的命名应当和 常量 或 宏 一致: `kEnumName` 或是 `ENUM_NAME`.

- 单独的枚举值应该优先采用 常量 的命名方式. 但 宏 方式的命名也可以接受. 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,
};
```

- 2009 年 1 月之前, 我们一直建议采用 宏 的方式命名枚举值. 由于枚举值和宏之间的命名冲突, 直接导致了很多问题. 由此, 这里改为优先选择常量风格的命名方式. 新代码应该尽可能优先使用常量风格. 但是老代码没必要切换到常量风格, 除非宏风格确实会产生编译期问题.

宏命名

> 你并不打算使用宏, 对吧? 如果你一定要用, 像这样命名: `MY_MACRO_THAT_SCARES_SMALL_CHILDREN`.

- 通常 不应该 使用宏. 如果不得不用, 其命名像枚举命名一样全部大写, 使用下划线::

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

注释

- 注释虽然写起来很痛苦, 但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, 但最好的代码本身应该是自文档化. 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.
- 你写的注释是给代码读者看的: 下一个需要理解你的代码的人. 慷慨些吧, 下一个人可能就是你!

注释风格

> 使用 `//` 或 `/* */`, 统一就好.

- `//` 或 `/* */` 都可以; 但 `//` 更 常用. 要在如何注释及注释风格上确保统一.

文件注释

> 在每一个文件开头加入版权公告, 然后是文件内容描述.

- 每个文件都应该包含以下项, 依次是:
    - 版权声明 (比如, `Copyright 2008 Google Inc.`)
    - 许可证. 为项目选择合适的许可证版本 (比如, Apache 2.0, BSD, LGPL, GPL)
    - 作者: 标识文件的原始作者.
    - 如果你对原始作者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:
    - 紧接着版权许可和作者信息之后, 每个文件都要用注释描述文件内容.
    - 通常, `.h` 文件要对所声明的类的功能和用法作简单说明. `.cc` 文件通常包含了更多的实现细节或算法技巧讨论, 如果你感觉这些实现细节或算法技巧讨论对于理解 `.h` 文件有帮助, 可以将该注释挪到 `.h`, 并在 `.cc` 中指出文档在 `.h`.
    - 不要简单的在 `.h` 和 `.cc` 间复制注释. 这种偏离了注释的实际意义.

类注释

> 每个类的定义都要附带一份注释, 描述类的功能和用法.

```
// 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” 也不打紧, 但务必确保有这类注释.
- 如果类有任何同步前提, 文档说明之. 如果该类的实例可被多线程访问, 要特别注意文档说明多线程环境下相关的规则和常量使用.

函数注释

> 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: 注释位于声明之前, 对函数功能及用法进行描述. 注释使用叙述式 (“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.
    - 如果函数分配了空间, 需要由调用者释放.
    - 参数是否可以为 `NULL`.
    - 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:

```
// 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;
```

- 但也要避免罗罗嗦嗦, 或做些显而易见的说明. 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, 切记读代码的人知道构造/析构函数是干啥的, 所以 “destroys this object” 这样的注释是没有意义的. 注明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.
- 函数定义: 每个函数定义时要用注释说明函数功能和实现要点. 比如说说你用的编程技巧, 实现的大致步骤, 或解释如此实现的理由, 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. 简要重述函数功能是可以的, 但注释重点要放在如何实现上.

变量注释

> 通常变量名本身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.

- 类数据成员: 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. 如果变量可以接受 `NULL` 或 `-1` 等警戒值, 须加以说明. 比如:

```
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: 和数据成员一样, 所有全局变量也要注释说明含义及用途. 比如:

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

实现注释

> 对于代码中巧妙的, 晦涩的, 有趣的, 重要的地方加以注释.

- 代码前注释: 巧妙或复杂的代码段前要加注释. 比如:

```
// 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. 在行尾空两格进行注释. 比如:

```
// 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, 这里用了两段注释分别描述这段代码的作用, 和提示函数返回时错误已经被记入日志.
- 如果你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();                  // Comment here so the comments line up.
DoSomethingElseThatIsLonger();  // Comment here so there are two spaces between
                                // the code and the comment.
{       // One space before comment when opening a new scope is allowed,
        // thus the comment lines up with the following comments and code.
        DoSomethingElse();  // Two spaces before line comments normally.
}
```

- NULL, true/false, 1, 2, 3...: 向函数传入 `NULL`, 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. 例如, 对比:

```
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.
```

- 或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);
```

- 不允许: 注意 永远不要 用自然语言翻译代码作为注释. 要假设读代码的人 C++ 水平比你高, 即便他/她可能不知道你的用意:

```
// 现在, 检查 b 数组并确保 i 是否存在,
// 下一个元素是 i+1.
...        // 天哪. 令人崩溃的注释.
```

标点, 拼写和语法

> 注意标点, 拼写和语法; 写的好的注释比差的要易读的多.

- 注释的通常写法是包含正确大小写和结尾句号的完整语句. 短一点的注释 (如代码行尾注释) 可以随意点, 依然要注意风格的一致性. 完整的语句可读性更好, 也可以说明该注释是完整的, 而不是一些不成熟的想法.
- 虽然被别人指出该用分号时却用了逗号多少有些尴尬, 但清晰易读的代码还是很重要的. 正确的标点, 拼写和语法对此会有所帮助.

TODO 注释

> 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.

- `TODO` 注释要使用全大写的字符串 `TODO`, 在随后的圆括号里写上你的大名, 邮件地址, 或其它身份标识. 冒号是可选的. 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. 添加 `TODO` 注释并不意味着你要自己来修正.

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

- 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.”).

弃用注释

> 通过弃用注释（ `DEPRECATED` comments ）以标记某接口点（ interface points ）已弃用。

- 您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，留下您的名字，邮箱地址以及括号补充。
- 仅仅标记接口为 `DEPRECATED` 并不会让大家不约而同地弃用，您还得亲自主动修正调用点（ callsites ），或是找个帮手。
- 修正好的代码应该不会再涉及弃用接口点了，着实改用新接口点。如果您不知从何下手，可以找标记弃用注释的当事人一起商量。

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. 个体未必同意下述每一处格式规则, 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能很轻松的阅读和理解代码.

行长度

> 每一行代码字符数不超过 80.

- 我们也认识到这条规则是有争议的, 但很多已有代码都已经遵照这一规则, 我们感觉一致性更重要.

优点:

- 提倡该原则的人主张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代码窗口, 根本没有多余空间拉伸窗口. 大家都把窗口最大尺寸加以限定, 并且 80 列宽是传统标准. 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.

特例:

- 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允许该行超过 80 字符.
- 包含长路径的 `#include` 语句可以超出80列. 但应该尽量避免.
- 头文件保护 可以无视该原则.

非 ASCII 字符

> 尽量不使用非 ASCII 字符, 使用时必须使用 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"` 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.
- 用 `u8` 前缀以把带 `uXXXX` 转义序列的字符串字面值编码成 UTF-8. 不要用在本身就带 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，除非您写的代码要调用 Windows API, 后者有用到 `wchar_t` 扩展。

空格还是制表位

> 只使用空格, 每次缩进 2 个空格.

- 我们使用空格缩进. 不要在代码中使用制符表. 你应该设置编辑器将制符表转为空格.

函数声明与定义

> 返回类型和函数名在同一行, 参数也尽量放在同一行，如果放不下就对形参分行。

- 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 空格缩进
         Type par_name2,
         Type par_name3) {
             DoSomething();  // 2 空格缩进
             ...
        }
```

- 注意以下几点:
    - 如果返回类型和函数名在一行放不下，分行。
    - 如果返回类型那个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;
    - 如果其它风格规则允许的话，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。
    - 右大括号和左大括号间总是有一个空格;
    - 函数声明和定义中的所有形参必须有命名且一致;
    - 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;
- 如果有些参数没有用到, 在函数定义处将参数名注释起来:

```
// 接口中形参恒有命名。
class Shape {
public:
        virtual void Rotate(double radians) = 0;
}

// 声明中形参恒有命名。
class Circle : public Shape {
public:
       virtual void Rotate(double radians);
}

// 定义中注释掉无用变量。
void Circle::Rotate(double /*radians*/) {}
```

```
// 差 - 如果将来有人要实现，很难猜出变量是干什么用的。
void Circle::Rotate(double) {}
```

Lambda 表达式

> 其它函数怎么格式化形参和函数体，Lambda 表达式就怎么格式化；捕获列表同理。

- 若用引用捕获，在变量名和 `&` 之间不留空格。

```
int x = 0;
auto add_to_x = [&x](int n) { 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());
```

函数调用

> 要么一行写完函数调用，要么在圆括号里对参数分行，要么参数另起一行且缩进四格。如果没有其它顾虑的话，尽可能精简行数，比如把多个参数适当地放在同一行里。

- 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);
```

- 参数也可以放在次行，缩进四格：

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

- 把多个参数放在同一行，是为了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，我们更看重可读性，且后者比较好办：
- 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 retval = DoSomething(my_heuristic, x, y, z);
```

- 或者放着不管，补充上注释：

```
bool retval = 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，就这么办。

- 此外，如果一系列参数本身就有一定的结构，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);
```

列表初始化格式

> 您平时怎么格式化函数调用，就怎么格式化列表。

- 如果列表初始化伴随着名字，比如类型或变量名，您可以当名字是函数、{} 是函数调用的括号来格式化它。反之，就当它有个长度为零的名字。

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

// 当不得不断行时。
SomeFunction(
        {"assume a zero-length name before {"},
         some_other_function_parameter);
SomeType variable{
        some, other, values,
        {"assume a zero-length name before {"},
        SomeOtherType{
            "Very long string requiring the surrounding breaks.",
            some, other values},
        SomeOtherType{"Slightly shorter string",
                      some, other, values}};
SomeType variable{
        "This is too long to fit all in one line"};
MyType m = {  // 注意了，您可以在 { 前断行。
        superlongvariablename1,
        superlongvariablename2,
        {short, interior, list},
        {interiorwrappinglist,
         interiorwrappinglist2}};
```

条件语句

> 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. 哪种都可以, 但 保持一致性. 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它文件. 还在徘徊的话, 就不要加空格了.

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

- 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){    // 变本加厉地差。
```

```
if (condition) {  // 可
```

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.

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.

Windows 代码

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

- 如果你习惯使用 Windows 编码风格, 这儿有必要重申一下某些你可能会忘记的指南:
    - 不要使用匈牙利命名法 (比如把整型变量命名成 `iNum`). 使用 Google 命名约定, 包括对源文件使用 `.cc` 扩展名.
    - Windows 定义了很多原生类型的同义词 , 如 `DWORD` , `HANDLE` 等等. 在调用 Windows API 时这是完全可以接受甚至鼓励的. 但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 warnings 当作 errors 处理.
    - 不要使用 `#pragma once` ; 而应该使用 Google 的头文件保护规则. 头文件保护的路径应该相对于项目根目录 (Yang.Y 注: 如 `#ifndef SRC_DIR_BAR_H_` , 参考 #define 保护 一节).
    - 除非万不得已, 不要使用任何非标准的扩展, 如 `#pragma` 和 `__declspec` . 允许使用 `__declspec(dllimport)` 和 `__declspec(dllexport)` ; 但你必须通过宏来使用, 比如 `DLLIMPORT` 和 `DLLEXPORT` , 这样其他人在分享使用这些代码时很容易就去掉这些扩展.
- 在 Windows 上, 只有很少的一些情况下, 我们可以偶尔违反规则:
    - 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, 你可能不得不使用多重实现继承.
    - 虽然代码中不应该使用异常, 但是在 ATL 和部分 STL (包括 Visual C++ 的 STL) 中异常被广泛使用. 使用 ATL 时, 应定义 `_ATL_NO_EXCEPTIONS` 以禁用异常. 你要研究一下是否能够禁用 STL 的异常, 如果无法禁用, 启用编译器

异常也可以. (注意这只是为了编译 STL, 自己代码里仍然不要含异常处理.)
    - 通常为了利用头文件预编译, 每个每个源文件的开头都会包含一个名为 `StdAfx.h` 或 `precompile.h` 的文件. 为了使代码方便与其他项目共享, 避免显式包含此文件 (`precompile.cc`), 使用 `/FI` 编译器选项以自动包含.
    - 资源头文件通常命名为 `resource.h`, 且只包含宏的, 不需要遵守本风格指南.

结束语

> 运用常识和判断力, 并 保持一致.

- 编辑代码时, 花点时间看看项目中的其它代码, 并熟悉其风格. 如果其它代码中 `if` 语句使用空格, 那么你也要使用. 如果其中的注释用星号 (*) 围成一个盒子状, 你同样要这么做.
- 风格指南的重点在于提供一个通用的编程规范, 这样大家可以把精力集中在实现内容而不是表现形式上. 我们展示了全局的风格规范, 但局部风格也很重要, 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, 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, 关于编码风格写的够多了; 代码本身才更有趣. 尽情享受吧!