Google C++ 风格指南 - 中文版

版本: 3.133

原作者: Benjy Weinberger
Craig Silverstein
Gregory Eitzmann
Mark Mentovai
Tashana Landray

翻译: YuleFox
yospaly

项目主页:

- Google Style Guide
- Google 开源项目风格指南 - 中文版

目录

Contents

译者前言

Google 经常会发布一些开源项目, 意味着会接受来自其他代码贡献者的代码. 但是如果代码贡献者的编程风格与 Google 的不一致, 会给代码阅读者和其他代码提交这造成不小的困扰. Google 因此发布了这份自己的编程风格, 使所有提交代码的人都能获知 Google 的编程风格.

翻译初衷:

> 规则的作用就是避免混乱. 但规则本身一定要权威, 有说服力, 并且是理性的. 我们所见过的大部分编程规范, 其内容或不够严谨, 或阐述过于简单, 或带有一定的武断性.
>
> Google 保持其一贯的严谨精神, 5 万汉字的指南涉及广泛, 论证严密. 我们翻译该系列指南的主因也正是其严谨性. 严谨意味着指南的价值不仅仅局限于它罗列出的规范, 更具参考意义的是它为了列出规范而做的谨慎权衡过程.
>
> 指南不仅列出你要怎么做, 还告诉你为什么要这么做, 哪些情况下可以不这么做, 以及如何权衡其利弊. 其他团队未必要完全遵照指南亦步亦趋, 如前面所说, 这份指南是 Google 根据自身实际情况打造的, 适用于其主导的开源项目. 其他团队可以参照该指南, 或从中汲取灵感, 建立适合自身实际情况的规范.
>
> 我们在翻译的过程中, 收获颇多. 希望本系列指南中文版对你同样能有所帮助.

我们翻译时也是尽力保持严谨, 但水平所限, bug 在所难免. 有任何意见或建议, 可与我们取得联系.

中文版修订历史:

- 2009-06 3.133 : YuleFox 的 1.0 版已经相当完善, 但原版在近一年的时间里, 其规范也发生了一些变化.

    > yospaly 与 YuleFox 一拍即合, 以项目的形式来延续中文版 : Google 开源项目风格指南 - 中文版项目.
    >
    > 主要变化是同步到 3.133 最新英文版本, 做部分勘误和改善可读性方面的修改, 并改进排版效果. yospaly 重新翻修, YuleFox 做后续评审.

- 2008-07 1.0 : 出自 YuleFox 的 Blog, 很多地方摘录的也是该版本.

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, 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.

1. 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.

1.1. #define 保护

> **Tip**
>
> 所有头文件都应该使用 `#define` 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, 头文件的命名应该依据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_
```

1.2. 头文件依赖

> **Tip**
>
> 能用前置声明的地方尽量不使用 `#include`.

当一个头文件被包含的同时也引入了新的依赖, 一旦该头文件被修改, 代码就会被重新编译. 如果这个头文件又包含了其他头文件, 这些头文件的任何改变都将导致所有包含了该头文件的代码被重新编译. 因此, 我们倾向于减少包含头文件, 尤其是在头文件中包含头文件.

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: 如果头文件中用到类 `File`, 但不需要访问 `File` 类的声明, 头文件中只需前置声明 `class File;` 而无须 `#include "file/base/file.h"`.

不允许访问类的定义的前提下, 我们在一个头文件中能对类 `Foo` 做哪些操作?

- 我们可以将数据成员类型声明为 `Foo *` 或 `Foo &`.
- 我们可以将函数参数 / 返回值的类型声明为 `Foo` (但不能定义实现).
- 我们可以将静态数据成员的类型声明为 `Foo`, 因为静态数据成员的定义在类定义之外.

反之, 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, 则必须包含 `Foo` 所在的头文件.

有时, 使用指针成员 (如果是 `scoped_ptr` 更好) 替代对象成员的确是明智之选. 然而, 这会降低代码可读性及执行效率, 因此如果仅仅为了少包含头文件, 还是不要这么做的好.

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.

1.3. 内联函数

> **Tip**
>
> 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:

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. (YuleFox 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). 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.

1.4. -inl.h文件

Tip
复杂的内联函数的定义, 应放在后缀名为 `-inl.h` 的头文件中.

内联函数的定义必须放在头文件中, 编译器才能在调用点内联展开定义. 然而, 实现代码理论上应该放在 `.cc` 文件中, 我们不希望 `.h` 文件中有太多实现代码, 除非在可读性和性能上有明显优势.

如果内联函数的定义比较短小, 逻辑比较简单, 实现代码放在 `.h` 文件里没有任何问题. 比如, 存取函数的实现理所当然都应该放在类定义内. 出于编写者和调用者的方便, 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, 也可以把它萃取到单独的 `-inl.h` 中. 这样把实现和类定义分离开来, 当需要时包含对应的 `-inl.h` 即可。

`-inl.h` 文件还可用于函数模板的定义. 从而增强模板定义的可读性.

别忘了 `-inl.h` 和其他头文件一样, 也需要 `#define` 保护.

1.5. 函数参数的顺序

Tip
定义函数时, 参数顺序依次为: 输入参数, 然后是输出参数.

C/C++ 函数参数分为输入参数, 输出参数, 和输入/输出参数三种. 输入参数一般传值或传 `const` 引用, 输出参数或输入/输出参数则是非-`const` 指针. 对参数排序时, 将只输入的参数放在所有输出参数之前. 尤其是不要仅仅因为是新加的参数, 就把它放在最后; 即使是新加的只输入参数也要放在输出参数.

这条规则并不需要严格遵守. 输入/输出两用参数 (通常是类/结构体变量) 把事情变得复杂, 为保持和相关函数的一致性, 你有时不得不有所变通.

1.6. `#include` 的路径及顺序

Tip
使用标准的头文件包含顺序可增强可读性, 避免隐藏依赖: 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:

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

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都是可被独立编译的 (yospaly 译注: 即该头文件本身已包含所有必要的显式依赖), 最简单的方法是将其作为第一个 `.h` 文件 `#included` 进对应的 `.cc`.

`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 (yospaly 译注: 之前已经按头文件类别排过序了).

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"
```

译者 (YuleFox) 笔记

1. 避免多重包含是学编程时最基本的要求;
2. 前置声明是为了降低编译依赖，防止修改一个头文件引发多米诺效应;
3. 内联函数的合理使用可提高代码执行效率;
4. `-inl.h` 可提高代码可读性 (一般用不到吧:D);
5. 标准化函数参数顺序可以提高可读性和易维护性 (对函数参数的堆栈空间有轻微影响, 我以前大多是相同类型放在一起);
6. 包含文件的名称使用 `.` 和 `..` 虽然方便却易混乱, 使用比较完整的项目路径看上去很清晰, 很条理, 包含文件的次序除了美观之外, 最重要的是可以减少隐藏依赖, 使每个头文件在 “最需要编译” (对应源文件处 :D) 的地方编译, 有人提出库文件放在最后, 这样出错先是项目内的文件, 头文件都放在对应源文件的最前面, 这一点足以保证内部错误的及时发现了.

2. 作用域

2.1. 名字空间

> **Tip**
>
> 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. 不要使用 *using 关键字*.

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 (YuleFox 注: 将命名分割在不同类的作用域内), 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.

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.

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.

2.1.1. 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.

2.1.2. 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, gflags 的声明/定义, 以及类的前置声明以外的整个源文件封装起来, 以区别于其它名字空间:

```
// .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*`` *关键字*, 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*`` *关键字*.

```
// 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;
```

2.2. 嵌套类

Tip

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名字空间内是更好的选择.

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.

2.3. 非成员函数, 静态成员函数, 和全局函数

Tip

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, 不如使用命名空间.

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, 可使用匿名名字空间或 `static` 链接关键字 (如 `static int Foo() {...}`) 限定其作用域.

2.4. 局部变量

Tip

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
nt j = g(); // 好—初始化时声明
```

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);
}
```

2.5. 静态和全局变量

> **Tip**
>
> 禁止使用 `class` 类型的静态或全局变量: 它们会导致很难发现的 bug 和不确定的构造和析构函数调用顺序.

静态生存周期的对象, 包括全局变量, 静态变量, 静态类成员变量, 以及函数静态变量, 都必须是原生数据类型 (POD : Plain Old Data): 只能是 *int*, *char*, *float*, 和 *void*, 以及 POD 类型的数组/结构体/指针. 永远不要使用函数返回值初始化静态变量; 不要在多线程代码中使用非 `const` 的静态变量.

不幸的是, 静态变量的构造函数, 析构函数以及初始化操作的调用顺序在 C++ 标准中未明确定义, 甚至每次编译构建都有可能会发生变化, 从而导致难以发现的 bug. 比如, 结束程序时, 某个静态变量已经被析构了, 但代码还在跑 - 其它线程很可能 - 试图访问该变量, 直接导致崩溃.

所以, 我们只允许 POD 类型的静态变量. 本条规则完全禁止 `vector` (使用 C 数组替代), `string` (使用 `const char*`), 及其它以任意方式包含或指向类实例的东东, 成为静态变量. 出于同样的理由, 我们不允许用函数返回值来初始化静态变量.

如果你确实需要一个 `class` 类型的静态或全局变量, 可以考虑在 ``main()` 函数或 `pthread_once()` 内初始化一个你永远不会回收的指针.

> **Note**
>
> yospaly 译注:
>
> 上文提及的静态变量泛指静态生存周期的对象, 包括: 全局变量, 静态变量, 静态类成员变量, 以及函数静态变量.

译者 (YuleFox) 笔记

1. `cc` 中的匿名名字空间可避免命名冲突, 限定作用域, 避免直接使用 `using` 关键字污染命名空间;
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 bug.
5. 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, 提高编译/执行效率.

3. 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.

3.1. 构造函数的职责

Tip
构造函数中只进行那些没什么意义的 (trivial, YuleFox 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 “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()` 方法集中初始化有意义的 (non-trivial) 数据.

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, 不能使用异常.
- 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:

如果对象需要进行有意义的 (non-trivial) 初始化, 考虑使用明确的 `Init()` 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.

3.2. 默认构造函数

Tip
如果一个类定义了若干成员变量又没有其它构造函数, 必须定义一个默认构造函数. 否则编译器将自动生产一个很糟糕的默认构造函数.

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.

优点:

默认将结构体初始化为 “无效” 值, 使调试更方便.

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, 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.

3.3. 显式构造函数

Tip
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`.

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.

3.4. 拷贝构造函数

Tip

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; 大部分情况下都不需要, 此时应使用 `DISALLOW_COPY_AND_ASSIGN`.

定义:

拷贝构造函数在复制一个对象到新建对象时被调用 (特别是对象传值时).

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++ 中的隐式对象拷贝是很多性能问题和 bug 的根源. 拷贝构造函数降低了代码可读性, 相比传引用, 跟踪传值的对象更加困难, 对象修改的地方变得难以捉摸.

结论:

大部分类并不需要可拷贝, 也不需要一个拷贝构造函数或重载赋值运算符. 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:` 中添加拷贝构造函数和赋值操作的空实现, 只有声明, 没有定义. 由于这些空函数声明为 `private`, 当其他代码试图使用它们的时候, 编译器将报错. 方便起见, 我们可以使用 `DISALLOW_COPY_AND_ASSIGN` 宏:

```
// 禁止使用拷贝构造函数和 operator= 赋值操作的宏
// 应该类的 private: 中使用

#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
            TypeName(const TypeName&); \
            void operator=(const TypeName&)
```

在 `class foo:` 中:

```
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` 宏. 如果类确实需要可拷贝, 应在该类的头文件中说明原由, 并合理的定义拷贝构造函数和赋值操作. 注意在 `operator=` 中检测自我赋值的情况 (yospaly 注: 即 `operator=` 接收的参数是该对象本身).

为了能作为 STL 容器的值, 你可能有使类可拷贝的冲动. 在大多数类似的情况下, 真正该做的是把对象的 *指针* 放到 STL 容器中. 可以考虑使用 `std::tr1::shared_ptr`.

3.5. 结构体 VS. 类

Tip
仅当只有数据时使用 `struct`, 其它一概使用 `class`.

在 C++ 中 `struct` 和 `class` 关键字几乎含义一样. 我们为这两个关键字添加我们自己的语义理解, 以便为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 (field), 而非函数调用. 除了构造函数, 析构函数, `Initialize()`, `Reset()`, `Validate()` 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 (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.

3.6. 继承

Tip
使用组合 (composition, YuleFox 注: 这一点也是 GoF 在 <<Design Patterns>> 里反复强调的) 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” (“is-a”, YuleFox 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, `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.

3.7. 多重继承

Tip
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.

Note
关于该规则, Windows 下有个 *特例*.

3.8. 接口

Tip
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.

3.9. 运算符重载

Tip
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*.

3.10. 存取控制

Tip
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 `foo()`. 还可能需要一个赋值函数 `set_foo()`.

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*

3.11. 声明顺序

Tip
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

宏 `DISALLOW_COPY_AND_ASSIGN` 的调用放在 `private:` 区段的末尾. 它通常是类的最后部分. 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*.

3.12. 编写简短函数

Tip
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.

译者 (YuleFox) 笔记

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, 可将其声明为 `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;

4. 来自 Google 的奇技

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

4.1. 智能指针

> **Tip**
>
> 如果确实需要使用智能指针的话, `scoped_ptr` 完全可以胜任. 你应该只在非常特定的情况下使用 `std::tr1::shared_ptr`, 例如 STL 容器中的对象. 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, 它会删除所指向的对象. `shared_ptr` 也是如此, 并且 `shared_ptr` 实现了引用计数, 所以最后一个 `shared_ptr` 对象析构时, 如果检测到引用次数为 0，就会销毁所指向的对象.

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作用域或局部变量使用. 另一种极端做法是, 引用计数指针不属于任何对象. 这种方法的问题是容易导致循环引用, 或者导致某个对象无法删除的诡异状态, 而且在每一次拷贝或赋值时连原子操作都会很慢.

虽然不推荐使用引用计数指针, 但有些时候它们的确是最简单有效的解决方案.

(YuleFox 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)

4.2. cpplint

> **Tip**
>
> 使用 `cpplint.py` 检查风格错误.

`cpplint.py` 是一个用来分析源文件, 能检查出多种风格错误的工具. 它不并完美, 甚至还会漏报和误报, 但它仍然是一个非常有用的工具. 用行注释 `// NOLINT` 可以忽略误报.

某些项目会指导你如何使用他们的项目工具运行 `cpplint.py`. 如果你参与的项目没有提供, 你可以单独下载 cpplint.py.

5. 其他 C++ 特性

5.1. 引用参数

> **Tip**
>
> 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, 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.

在以下情况你可以把输入参数定义为 `const` 指针: 你想强调参数不是拷贝而来的, 在对象生存周期内必须一直存在; 最好同时在注释中详细说明一下. `bind2nd` 和 `mem_fun` 等 STL 适配器不接受引用参数, 这种情况下你也必须把函数参数声明成指针类型.

5.2. 函数重载

> **Tip**
>
> 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). 不要用函数重载模拟 缺省函数参数.

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:

限制使用重载的一个原因是在某个特定调用点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了某个函数的部分变体, 会令很多人对继承的语义产生困惑. 此外在阅读库的用户代码时, 可能会因反对使用 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()`.

5.3. 缺省参数

> **Tip**

我们不允许使用缺省函数参数.

优点:

多数情况下, 你写的函数可能会用到很多的缺省值, 但偶尔你也会修改这些缺省值. 无须为了这些偶尔情况定义很多的函数, 用缺省参数就能很轻松的做到这点.

缺点:

大家通常都是通过查看别人的代码来推断如何使用 API. 用了缺省参数的代码更难维护, 从老代码复制粘贴而来的新代码可能只包含部分参数. 当缺省参数不适用于新代码时可能会导致重大问题.

结论:

我们规定所有参数必须明确指定, 迫使程序员理解 API 和各参数值的意义, 避免默默使用他们可能都还没意识到的缺省参数.

5.4. 变长数组和 alloca()

Tip

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:

使用安全的内存分配器, 如 `scoped_ptr` / `scoped_array`.

5.5. 友元

Tip

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.

5.6. 异常

Tip

我们不使用 C++ 异常.

优点:

- 异常允许上层应用决定如何处理在底层嵌套函数中 “不可能出现的” 失败, 不像错误码记录那么含糊又易出错;
- 很多现代语言都使用异常. 引入异常使得 C++ 与 Python, Java 以及其它 C++ 相近的语言更加兼容.
- 许多第三方 C++ 库使用异常, 禁用异常将导致很难集成这些库.
- 异常是处理构造函数失败的唯一方法. 虽然可以通过工厂函数或 `Init()` 方法替代异常, 但他们分别需要堆分配或新的 “无效” 状态;
- 在测试框架中使用异常确实很方便.

缺点:

- 在现有函数中添加 `throw` 语句时, 你必须检查所有调用点. 所有调用点得至少有基本的异常安全保护, 否则永远捕获不到异常, 只好 “开心的” 接受程序终止的结果. 例如, 如果 `f()` 调用了 `g()`, `g()` 又调用了 `h()`, `h` 抛出的异常被 `f` 捕获, `g` 要当心了, 很可能会因疏忽而未被妥善清理.

- 更普遍的情况是, 如果使用异常, 光凭查看代码是很难评估程序的控制流: 函数返回点可能在你意料之外. 这回导致代码管理和调试困难. 你可以通过规定何时何地如何使用异常来降低开销, 但是让开发人员必须掌握并理解这些规定带来的代价更大.
- 异常安全要求同时采用 RAII 和不同编程实践. 要想轻松编写正确的异常安全代码, 需要大量的支撑机制配合. 另外, 要避免代码读者去理解整个调用结构图, 异常安全代码必须把写持久化状态的逻辑部分隔离到 “提交” 阶段. 它在带来好处的同时, 还有成本 (也许你不得不为了隔离 “提交” 而整出令人费解的代码). 允许使用异常会驱使我们不断为此付出代价, 即使我们觉得这很不划算.
- 启用异常使生成的二进制文件体积变大, 延长了编译时间 (或许影响不大), 还可能增加地址空间压力;
- 异常的实用性可能会怂恿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, 处理非法用户输入时就不应该抛出异常. 如果我们要完全列出这些约束, 这份风格指南会长出很多!

结论:

从表面上看, 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*.

(YuleFox 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)

5.7. 运行时类型识别

Tip
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. 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.

除测试外, 极少用到.

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.

结论:

除单元测试外, 不要使用 RTTI. 如果你发现自己不得不写一些行为逻辑取决于对象类型的代码, 考虑换一种方式判断对象类型.

如果要实现根据子类类型来确定执行不同逻辑代码, 虚函数无疑更合适. 在对象内部就可以处理类型识别问题.

如果要在对象外部的代码中判断类型, 考虑使用双重分派方案, 如访问者模式. 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真的掌握不了, 你可以使用 RTTI, 但务必请三思 :-) . 不要试图手工实现一个貌似 RTTI 的替代方案, 我们反对使用 RTTI 的理由, 同样适用于那些在类型继承体系上使用类型标签的替代方案.

5.8. 类型转换

Tip
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.
- `dynamic_cast` 测试代码以外不要使用. 除非是单元测试, 如果你需要在运行时确定类型信息, 说明有 设计缺陷.

5.9. 流

Tip

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

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`.

5.10. 前置自增和自减

Tip
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).

5.11. `const` 的使用

Tip
我们强烈建议你在任何可能的情况下都要使用 `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! (yospaly 注: 也就是不要在一些地方把 `const` 写在类型前面, 在其他地方又写在后面, 确定一种写法, 然后保持一致.)

5.12. 整型

Tip

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`.

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.

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!

5.13. 64 位下的可移植性

Tip

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 (yospaly 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 (yospaly 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())` (YuleFox 注, 解决方案的项目属性里也可以直接设置).
- 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)

5.14. 预处理宏

Tip

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.

5.15. 0 和 NULL

Tip

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) 对 `NULL` 进行了特殊的定义, 可以给出有用的警告信息, 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.

5.16. sizeof

Tip

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. 某些情况下 `sizeof(type)` 或许有意义, 但还是要尽量避免, 因为它会导致变量类型改变后不能同步.

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

Warning

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

5.17. Boost 库

Tip

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, 同时还提供了 `TR1` (标准库扩展) 的实现.

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:

- Compressed Pair : `boost/compressed_pair.hpp`
- Pointer Container : `boost/ptr_container` (序列化除外)
- Array : `boost/array.hpp`
- The Boost Graph Library (BGL) : `boost/graph` (序列化除外)
- Property Map : `boost/property_map.hpp`
- Iterator 中处理迭代器定义的部分 : `boost/iterator/iterator_adaptor.hpp`, `boost/iterator/iterator_facade.hpp`, 以及 `boost/function_output_iterator.hpp`

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.

6. 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.

6.1. 通用命名规则

Tip

函数命名, 变量命名, 文件命名应具备描述性; 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:

尽可能给出描述性的名称. 不要节约行空间, 让别人很快理解你的代码更重要. 好的命名风格:

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

糟糕的命名使用含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 (确切的说它们应该是命令), 如 `OpenFile()`, `set_num_errors()`. 取值函数是个特例 (在 *函数命名* 处详细阐述), 函数名和它要取值的变量同名.

缩写:

除非该缩写在其它地方都非常普遍, 否则不要使用. 例如:

```
// Good
// These show proper names with no abbreviations.
int num_dns_connections;  // 大部分人都知道 "DNS" 是啥意思.
int price_count_reader;   // OK, price count. 有意义.
```

Warning

```
// 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".
```

永远不要用省略字母的缩写:

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

6.2. 文件命名

Tip

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.

可接受的文件命名:

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

C++ 文件要以 `.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 (yospaly 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. 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, 可以使用三个文件:

```
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 *-inl.h 文件* 一节.

6.3. 类型命名

Tip

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 { ...
```

6.4. 变量命名

Tip

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, 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.
```

Warning

```
string tableName;   // Bad - mixed case.
```

结构体变量:

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 *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.

6.5. 常量命名

Tip
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, 和其他变量稍微区别一下. k 后接大写字母开头的单词::

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

6.6. 函数命名

Tip
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:

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

取值和设值函数:

取值和设值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.

6.7. 名字空间命名

Tip
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.

6.8. 枚举命名

Tip
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.

6.9. 宏命名

> **Tip**
>
> 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`.

参考 *预处理宏* `<preprocessor-macros>`; 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
```

6.10. 命名规则的特例

> **Tip**
>
> 如果你命名的实体与已有 C/C++ 实体相似, 可参考现有命名策略.

`bigopen()`:

函数名, 参照 `open()` 的形式

`uint`:

`typedef`

`bigpos`:

`struct` 或 `class`, 参照 `pos` 的形式

`sparse_hash_map`:

STL 相似实体; 参照 STL 命名约定

`LONGLONG_MAX`:

常量, 如同 `INT_MAX`

7. 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!

7.1. 注释风格

Tip
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.

7.2. 文件注释

Tip
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.

7.3. 类注释

Tip
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.

7.4. 函数注释

> **Tip**
>
> 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.

7.5. 变量注释

> **Tip**
>
> 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;
```

7.6. 实现注释

Tip

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 betwee
                                // 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:

Warning

```
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
                                  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:

Warning

```
// 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.
```

7.7. 标点, 拼写和语法

Tip

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.

7.8. TODO 注释

Tip

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.”).

译者 (YuleFox) 笔记

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.

8. 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.

另外, 我们写了一个 emacs 配置文件 来帮助你正确的格式化代码.

8.1. 行长度

Tip
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.

8.2. 非 ASCII 字符

Tip
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. (yospaly 注: `"\xEF\xBB\xBF"` 通常用作 UTF-8 with BOM 编码标记)

8.3. 空格还是制表位

Tip
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.

8.4. 函数声明与定义

Tip
返回类型和函数名在同一行, 参数也尽量放在同一行.

函数看上去像这样:

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

```
    ...
}
```

如果同一行文本太多, 放不下所有参数:

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

甚至连第一个参数都放不下:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
        Type par_name1,  // 4 space indent
        Type par_name2,
        Type par_name3) {
    DoSomething();  // 2 space indent
    ...
}
```

注意以下几点:

- 返回值总是和函数名在同一行;
- 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;
- 右大括号总是单独位于函数最后一行;
- 右圆括号和左大括号间总是有一个空格;
- 函数声明和实现处的所有形参名称必须保持一致;
- 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;

如果函数声明成 `const`, 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*/) {}
```

Warning

```
// Bad - if someone wants to implement later, it's not clear what the
// variable means.
void Circle::Rotate(double) {}
```

8.5. 函数调用

Tip

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);
```

如果函数参数很多, 出于可读性的考虑可以在每行只放一个参数:

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

如果函数名非常长, 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);
  }
```

8.6. 条件语句

Tip

倾向于不在圆括号内使用空格. 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, 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) {  // no spaces inside parentheses
  ...  // 2 space indent.
} else {  // The else goes on the same line as the closing brace.
  ...
}
```

如果你更喜欢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` 和左圆括号间都有个空格. 右圆括号和左大括号之间也要有个空格:

Warning

```
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 {.
```

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

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

如果语句有 `else` 分支则不允许:

Warning

```
// 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, 如果你喜欢用也没问题; 复杂的条件或循环语句用大括号可读性会更好. 也有一些项目要求 `if` 必须总是使用大括号:

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

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

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

Warning

```
// 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;
}
```

8.7. 循环和开关选择语句

Tip

`switch` 语句可以使用大括号分段. 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, 取决于你的个人喜好. 如果用的话, 要按照下文所述的方法.

如果有不满足 `case` 条件的枚举值, `switch` 应该总是包含一个 `default` 匹配 (如果有输入值没有 case 去处理, 编译器将报警). 如果 `default` 应该永远执行不到, 简单的加条 `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.
```

Warning

```
while (condition);  // Bad - looks like part of do/while loop.
```

8.8. 指针和引用表达式

Tip

句点或箭头前后不要有空格. 指针/地址操作符 (`*, &`) 之后不能有空格.

下面是指针和引用表达式的正确使用范例:

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

注意:

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

在声明指针变量或参数时, 星号与类型或变量名紧挨都可以:

```
// 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;
```

Warning

```
char * c;  // Bad - spaces on both sides of *
const string & str;  // Bad - spaces on both sides of &
```

在单个文件内要保持风格一致, 所以, 如果是修改现有文件, 要遵照该文件的风格.

8.9. 布尔表达式

Tip

如果一个布尔表达式超过 *标准行宽*, 断行方式要统一一下.

下例中, 逻辑与 (`&&`) 操作符总位于行尾:

```
if (this_one_thing > this_other_thing &&
    a_third_thing == a_fourth_thing &&
    yet_another & last_one) {
  ...
}
```

注意, 上例的逻辑与 (&&) 操作符均位于行尾. 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.

8.10. 函数返回值

Tip

return 表达式中不要用圆括号包围.

函数返回时不要使用圆括号:

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

8.11. 变量及数组初始化

Tip

用 = 或 () 均可.

在二者中做出选择; 下面的方式都是正确的:

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

8.12. 预处理指令

Tip

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();
  }
```

Warning

```
// 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();
  }
```

8.13. 类格式

Tip

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

类声明 (对类注释不了解的话, 参考 *类注释*) 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);
};
```

注意事项:

- 所有基类名应在 80 列限制下尽量与子类名放在同一行.
- 关键词 `public:`, `protected:`, `private:` 要缩进 1 个空格.
- 除第一个关键词 (一般是 `public`) 外, 其他关键词前要空一行. 如果类比较小的话也可以不空.
- 这些关键词后不要保留空行.
- `public` 放在最前面, 然后是 `protected`, 最后是 `private`.
- 关于声明顺序的规则请参考 *声明顺序* 一节.

8.14. 初始化列表

Tip

构造函数初始化列表放在同一行或按四格缩进并排几行.

下面两种初始化列表方式都可以接受:

```
// 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();
  ...
}
```

8.15. 名字空间格式化

Tip

名字空间内容不缩进.

名字空间 不要增加额外的缩进层次, 例如:

```
namespace {

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

}  // namespace
```

不要缩进名字空间:

Warning

```
namespace {

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

```
} // namespace
```

8.16. 水平留白

Tip

水平留白的使用因地制宜. 永远不要在行尾添加没意义的留白.

常规:

```
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, 行尾不要留空格. 如果确定一行代码已经修改完毕, 将多余的空格去掉; 或者在专门清理空格时去掉（确信没有其他人在处理). (yospaly 注: 现在大部分代码编辑器稍加设置后, 都支持自动删除行首/行尾空格, 如果不支持, 考虑换一款编辑器或 IDE)

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 < <.
```

8.17. 垂直留白

Tip

垂直留白越少越好.

这不仅仅是规则而是原则问题了: 不在万不得已, 不要使用空行. 尤其是: 两个函数定义之间的空行不要超过 2 行, 函数体首尾不要留空行, 函数体中也不要随意添加空行.

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

Warning

函数首尾不要有空行

```
void Function() {

  // Unnecessary blank lines before and after

}
```

Warning

代码块首尾不要有空行

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

}
if (condition) {

  // Unnecessary blank line before
}
```

`if-else` 块之间空一行是可以接受的:

```
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 (YuleFox) 笔记

0. 对于代码格式, 因人, 系统而异各有优缺点, 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.
11. 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.

9. 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.

9.1. 现有不合规范的代码

> **Tip**
>
> 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.

9.2. Windows 代码

> **Tip**
>
> 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 (YuleFox 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 (yospaly 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.

10. 结束语

> **Tip**
>
> 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!

```
Revision 3.133

Benjy Weinberger
Craig Silverstein
Gregory Eitzmann
Mark Mentovai
Tashana Landray
```