MINIEYE-代码风格指南

Apr 17, 2023

Contents

CHAPTER 1

MINIEYE 代码风格指南

这是 MINIEYE C++ 代码风格指南，它吸收了：

- Google Style Guide：https://google.github.io/styleguide
- 社区对 Google 代码风格指南的翻译：https://github.com/zh-google-styleguide/zh-google-styleguide
- 一份古老而不完备的《C++ 编码规范》。
- 车辆组现行的代码风格指南。
- 《融合组 C++ 代码规范》。
- MINIEYE 内部评审。

1.1 维护和贡献

除了上述既有代码规范，下列作者也提供了修改意见。

- 张世权
- 黄剑
- 唐壤

你可以在 https://git.minieye.tech/vehicle_dev/code-style.git 找到这份风格指南的代码。

1.2 2023-04-12 评审

本规范在 2023 年 04 月 12 日经过下列成员组成的评审委员会评审，合并了评审修改意见。

- 杨华良
- 谭首锋
- 徐焕东
- 黄剑
- 周建
- 刘平
- 朱爱晨
- 徐涵
- 黄国勇

组织和记录：

- 卢珊

本次评审的决定详见：*2023-04-12 评审*。

CHAPTER 2

C++ 风格指南 - 内容目录

2.1 C++ 标准版本

使用 C++11 标准，不使用非标准扩展。

除非：

1. 外部 SDK 头文件要求更高版本标准。
2. 对非标准扩展的使用限制在可移植接口内。

特别是：

- 禁用 `#pragma once`： 使用保护宏代替。
- 禁用指定字段初始化:

```
Foo f = {.a = 1, .b = 2};
```

- 允许属性扩展，例如 `__attribute__` 指令，但是需要包装在可移植的宏里。

2.2 头文件

通常每一个 `.cpp` 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 `.cpp` 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.

2.2.1 Self-contained 头文件

Tip: 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，一个头文件要有头文件保护，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 .cpp 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。

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 .cpp 文件里。

2.2.2 头文件保护

所有头文件都应该有 #define 保护来防止头文件被多重包含，特别是禁止使用 pragma once 。

一个代码仓库的一般布局是:

```
include/foo/bar.h
src/baz.h
src/submod/baz.h
```

在 include 目录下的头文件可能会发布到其他人的仓库中。因此对它的引入保护需要包含项目的名字。上述三个文件分对应的引入保护宏是:

```
INCLUDE_FOO_BAR_H_
SRC_BAZ_H_
SRC_SUBMOD_BAZ_H_
```

使用下面的格式保护起来:

```
#ifndef {GUARD}
#define {GUARD}
...
#endif  // {GUARD}
```

2.2.3 前置声明

Tip: 除了 pimpl idiom，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.

pimpl idiom 指的是分离实现细节和接口的实现方法，例如:

```
// include/foo/bar.h

class Bar {
 public:
  class Impl;

 private:
  Impl *impl_;
};
```

而 `PublicInterface::Impl` 的定义则包裹在内部头文件 `src/bar.h` 里。

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 `#include` 甚至都会暗暗地改变代码的含义:

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

// good_user.cpp:
#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* 。

因为 pimpl idiom 实际上只在类内出现前置声明，并不存在前叙的各种缺点，又对平台一致的代码很重要，因此不禁止 pimpl idiom 的使用。

2.2.4 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

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. (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.

2.2.5 #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:

1. **相关头文件**
2. C 标准库
3. C++ 标准库
4. 平台 SDK 或者开源代码的头文件
5. 其他库的 `.h`
6. 本项目内的公开的 `.h`
7. 本项目内私有的 `.h`

只有 C 和 C++ 标准库使用尖括号引入，其他使用引号引入。

当 foo.h 包含 foo.cpp 的公开符号声明，foo.cpp 包含 foo.h 的声明的定义，则 foo.h 是 foo.cpp 的相关头文件。foo_test.cpp 包含对 foo.h 的测试，则 foo.h 也是 foo_test.cpp 的 ** 相关头文件 **。

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, 项目 `foo` 中的 `foo/src/base/logging.h` 应该按如下方式包含:

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

而 `foo/include/foo/log.h` 应该按照如下方式包含:

```
#include "foo/log.h"
```

基于这样的规范，可以仅查看头文件就明白头文件所在的位置范围:

```
#include "src/..."       // 本项目私有头文件
#include "{module}/..."  // 名为 module 的项目（永远不能将 src 当作项目名称）
#include "{file}.h"      // 不符合规范
```

按字母顺序分别对每种类型的头文件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。

您所依赖的符号 (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 (forward declarations) 情况除外。比如您要用到 `bar.h` 中的某个符号, 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 cpp 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 cpp 文件里面了，就像 `foo.cpp` 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, `awesome-project/src/foo/internal/fooserver.cpp` 的包含次序如下:

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

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

(continues on next page)

(continued from previous page)

```
#include <hash_map>
#include <vector>

#include "src/base/basictypes.h"
#include "src/base/commandlineflags.h"
#include "src/foo/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 "src/base/port.h"  // For LANG_CXX11.

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

2.3 作用域

2.3.1 命名空间

Tip: 鼓励在 `.cpp` 文件内使用匿名命名空间或 `static` 声明. 使用具名的命名空间时, 其名称可基于项目名或相对路径. 禁止使用 using 指示（using-directive）。禁止使用内联命名空间（inline namespace）。

定义

命名空间将全局作用域细分为独立的, 具名的作用域, 可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), 命名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命名空间中, `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 {
```

(continues on next page)

(continued from previous page)

```
void foo();

}  // namespace Y
}  // namespace X
```

`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

命名空间具有迷惑性, 因为它们使得区分两个相同命名所指代的定义更加困难。

内联命名空间很容易令人迷惑，毕竟其内部的成员不再受其声明所在命名空间的限制。内联命名空间只在大型版本控制里有用。

有时候不得不多次引用某个定义在许多嵌套命名空间里的实体，使用完整的命名空间会导致代码的冗长。

在头文件中使用匿名空间导致违背 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.

- 遵守 命名空间命名 中的规则。
- 像之前的几个例子中一样，在命名空间的最后注释出命名空间的名字。
- 用命名空间把除了头文件包含、gflags 的声明/定义 以及其他命名空间的类的前置声明以外的整个源文件封装起来:

```
// .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
```

```
// .cpp 文件
namespace mynamespace {

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

(continues on next page)

(continued from previous page)

```
} // namespace mynamespace
```

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

```
#include "a.h"

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

namespace a {

...code for a...          // 左对齐

} // namespace a
```

- 不要在命名空间 std 内声明任何东西, 包括标准库的类前置声明. 在 std 命名空间声明实体是未定义的行为, 会导致如不可移植. 声明标准库下的实体, 需要包含对应的头文件.
- 不应该使用 *using* 指示引入整个命名空间的标识符号。

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

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

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

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

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

- 禁止用内联命名空间

2.3.2 匿名命名空间和静态变量

Tip: 在 .cpp 文件中定义一个不需要被外部引用的变量时，可以将它们放在匿名命名空间或声明为 static。但是不要在 .h 文件中这么做。

定义

所有置于匿名命名空间的声明都具有内部链接性，函数和变量可以经由声明为 static 拥有内部链接性，这意味着你在这个文件中声明的这些标识符都不能在另一个文件中被访问。即使两个文件声明了完全一样名字的标识符，它们所指向的实体实际上是完全不同的。

结论

推荐、鼓励在 .cpp 中对于不需要在其他地方引用的标识符使用内部链接性声明，但是不要在 .h 中使用。

匿名命名空间的声明和具名的格式相同，在最后注释上 namespace：

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

2.3.3 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命名空间内的非成员函数，尽量不要用裸的全局函数. 将一系列函数直接置于命名空间中，不要用类的静态方法模拟出命名空间的效果，类的静态方法应当和类的实例或静态数据紧密相关.

优点

某些情况下，非成员函数和静态成员函数是非常有用的，将非成员函数放在命名空间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，应尽量置于某个命名空间内. 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 。举例而言，对于头文件 myproject/foo_bar.h，应当使用

```
namespace myproject {
namespace foo_bar {
```

(continues on next page)

(continued from previous page)

```
void Function1();
void Function2();

}  // namespace foo_bar
}  // namespace myproject
```

而非

```
namespace myproject {

class FooBar {
 public:
  static void Function1();
  static void Function2();
};

}  // namespace myproject
```

定义在同一编译单元的函数, 被其他编译单元直接调用可能会引入不必要的耦合和链接时依赖; 静态成员函数对此尤其敏感. 可以考虑提取到新类中, 或者将函数置于独立库的命名空间内.

如果你必须定义非成员函数, 又只是在 `.cpp` 文件中使用它, 可使用匿名命名空间 或 `static` 链接关键字 (如 `static int Foo() {...}`) 限定其作用域.

2.3.4 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
```

```
int j = g(); // 好——初始化时声明
```

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

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

属于 if, while 和 for 语句的变量应当在这些语句中声明，这样子这些变量的作用域就被限制在这些语句中了，举例而言:

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

Warning: 有一个例外, 如果变量是一个对象, 每次进入作用域都要调用其构造函数, 每次退出作用域都要调用其析构函数. 这会导致效率降低.

```
// 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);
}
```

2.3.5 静态和全局变量

Tip: 禁止定义静态储存周期非 POD 变量，禁止使用含有副作用的函数初始化 POD 全局变量，因为多编译单元中的静态变量执行时的构造和析构顺序是未明确的，这将导致代码的不可移植。

禁止使用类的 静态储存周期 变量：由于构造和析构函数调用顺序的不确定性，它们会导致难以发现的 bug 。不过 constexpr 变量除外，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++ 中是只有部分明确的，甚至随着构建变化而变化，导致难以发现的 bug. 所以除了禁用类类型的全局变量，我们也不允许用函数返回值来初始化 POD 变量，除非该函数（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。

2.3.6 线程局部变量

不在函数内定义的 `thread_local` 变量必须用编译期常量初始化。选择 `thread_local` 而不是其他方式来定义线程局部变量。

定义

变量可以用 `thread_local` 标识符声明。

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

这样的变量实际是一堆变量的集合，因此不同线程访问它时，这些线程实际上是在访问不同的对象。`thread_local` 变量在很多方面与静态存储变量十分相似。例如，它们可以定义在名字空间作用域，可以定义在函数内，可以定义为静态类成员，但是不能作为不同类成员。

`thread_local` 变量实例初始化跟静态变量初始化很相似，除了它们在不同线程里单独初始化，而不是在程序启动时初始化。这意味着 `thread_local` 变量声明在函数内时是安全的，但是其他 `thread_local` 变量与静态变量一样受到初始化顺序问题（和其他问题）的影响。

`thread_local` 变量实例在线程它们的线程结束之前不会销毁，因此它们不会像静态变量一样受到析构顺序影响。

优点

1. 线程局部数据对于数据竞争来说是天然安全的(因为通常只有一个线程可以访问它)，因此 `thread_local` 在并发编程中非常有用。
2. `thread_local` 是创建线程局部数据的唯一标准支持。

缺点

1. 访问线程局部变量可能执行一些不可预期和不可控制的代码。
2. 线程局部代码都是全局变量，有全局变量所有的缺点，除了没有线程安全问题。
3. 线程局部变量消耗的内存随着线程增加而增加（在最坏的情况下），这可能在程序中占用很多内存。
4. 非静态的数据成员无法声明为 `thread_local` 。
5. `thread_local` 可能比如特定的编译器指令高效。

结论

在函数内的 thread_local 变量没有安全问题，可以无限制的使用。注意可以用函数中声明的 thread_local 来模拟类或者名字空间内的 thread_local，你可以定义一个函数或者静态成员函数来暴露线程局部变量:

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

类内或者名字空间内的 thread_local 变量必须使用编译期常量初始化（即不能有动态初始化）。

在各种定义线程局部数据的机制中，首选 thread_local 。

2.4 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.

2.4.1 构造函数的职责

总述

不要在构造函数中调用虚函数, 也不要在无法报出错误时进行可能失败的初始化.

定义

在构造函数中可以进行各种初始化操作.

优点

- 无需考虑类是否被初始化.
- 经过构造函数完全初始化后的对象可以为 const 类型, 也能更方便地被标准容器或算法使用.

缺点

- 如果在构造函数内调用了自身的虚函数, 这类调用是不会重定向到子类的虚函数实现. 即使当前没有子类化实现, 将来仍是隐患.
- 在没有使程序崩溃 (因为并不是一个始终合适的方法) 或者使用异常 (因为已经被禁用 了) 等方法的条件下, 构造函数很难上报错误
- 如果执行失败, 会得到一个初始化失败的对象, 这个对象有可能进入不正常的状态, 必须使用 bool IsValid() 或类似这样的机制才能检查出来, 然而这是一个十分容易被疏忽的方法.
- 构造函数的地址是无法被取得的, 因此, 举例来说, 由构造函数完成的工作是无法以简单的方式交给其他线程的.

结论

构造函数不允许调用虚函数. 如果代码允许, 直接终止程序是一个合适的处理错误的方式. 否则, 考虑用 `Init()` 方法或工厂函数.

构造函数不得调用虚函数, 或尝试报告一个非致命错误. 如果对象需要进行有意义的 (non-trivial) 初始化, 考虑使用明确的 Init() 方法或使用工厂模式. Avoid `Init()` methods on objects with no other states that affect which public methods may be called (此类形式的半构造对象有时无法正确工作).

2.4.2 隐式类型转换

总述

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

定义

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

除了语言所定义的隐式类型转换, 用户还可以通过在类定义中添加合适的成员定义自己需要的转换. 在源类型中定义隐式类型转换, 可以通过目的类型名的类型转换运算符实现 (例如 `operator bool()`). 在目的类型中定义隐式类型转换, 则通过以源类型作为其唯一参数 (或唯一无默认值的参数) 的构造函数实现.

`explicit` 关键字可以用于构造函数或 (在 C++11 引入) 类型转换运算符, 以保证只有当目的类型在调用点被显式写明时才能进行类型转换, 例如使用 `cast`. 这不仅作用于隐式类型转换, 还能作用于 C++11 的列表初始化语法:

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

void Func(Foo f);
```

此时下面的代码是不允许的:

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

这一代码从技术上说并非隐式类型转换, 但是语言标准认为这是 `explicit` 应当限制的行为.

优点

- 有时目的类型名是一目了然的, 通过避免显式地写出类型名, 隐式类型转换可以让一个类型的可用性和表达性更强.
- 隐式类型转换可以简单地取代函数重载.
- 在初始化对象时, 列表初始化语法是一种简洁明了的写法.

缺点

- 隐式类型转换会隐藏类型不匹配的错误. 有时, 目的类型并不符合用户的期望, 甚至用户根本没有意识到发生了类型转换.
- 隐式类型转换会让代码难以阅读, 尤其是在有函数重载的时候, 因为这时很难判断到底是哪个函数被调用.
- 单参数构造函数有可能会被无意地用作隐式类型转换.
- 如果单参数构造函数没有加上 `explicit` 关键字, 读者无法判断这一函数究竟是要作为隐式类型转换, 还是作者忘了加上 `explicit` 标记.
- 并没有明确的方法用来判断哪个类应该提供类型转换, 这会使得代码变得含糊不清.
- 如果目的类型是隐式指定的, 那么列表初始化会出现和隐式类型转换一样的问题, 尤其是在列表中只有一个元素的时候.

结论

在类型定义中, 类型转换运算符和单参数构造函数都应当用 `explicit` 进行标记. 一个例外是, 拷贝和移动构造函数不应当被标记为 `explicit`, 因为它们并不执行类型转换. 对于设计目的就是用于对其他类型进行透明包装的类来说, 隐式类型转换有时是必要且合适的. 这时应当联系项目组长并说明特殊情况.

不能以一个参数进行调用的构造函数不应当加上 `explicit`. 接受一个 `std::initializer_list` 作为参数的构造函数也应当省略 `explicit`, 以便支持拷贝初始化 (例如 `MyType m = {1, 2};`).

2.4.3 可拷贝类型和可移动类型

总述

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

定义

可拷贝类型允许对象在初始化时得到来自相同类型的另一对象的值, 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<int>` 就是一个可移动但不可复制的对象的例子. 对于用户定义的类型, 移动操作一般是通过移动构造函数和移动赋值操作符实现的.

拷贝 / 移动构造函数在某些情况下会被编译器隐式调用. 例如, 通过传值的方式传递对象.

优点

可移动及可拷贝类型的对象可以通过传值的方式进行传递或者返回, 这使得 API 更简单, 更安全也更通用. 与传指针和引用不同, 这样的传递不会造成所有权, 生命周期, 可变性等方面的混乱, 也就没必要在协议中予以明确. 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 `= default`. 这种方式很简洁, 也保证所有数据成员都会被复制. 拷贝与移动构造函数一般也更高效, 因为它们不需要堆的分配或者是单独的初始化和赋值步骤, 同时, 对于类似 省略不必要的拷贝 这样的优化它们也更加合适.

移动操作允许隐式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. 单件类型 (`Registerer`), 与特定的作用域相关的类型 (`Cleanup`), 与其他对象实体紧耦合的类型 (`Mutex`) 从逻辑上来说都不应该提供拷贝操作. 为基类提供拷贝 / 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, 反之亦然. 如果让类型可移动, 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.

如果定义了拷贝/移动操作, 则要保证这些操作的默认实现是正确的. 记得时刻检查默认操作的正确性, 并且在文档中说明类是可拷贝的且/或可移动的.

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

 private:
  Field field_;
};
```

由于存在对象切割的风险, 不要为任何有可能有派生类的对象提供赋值操作或者拷贝 / 移动构造函数 (当然也不要继承有这样的成员函数的类). 如果你的基类需要可复制属性, 请提供一个 `public virtual Clone()` 和一个 `protected` 的拷贝构造函数以供派生类实现.

如果你的类不需要拷贝 / 移动操作, 请显式地通过在 `public` 域中使用 `= delete` 或其他手段禁用之.

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

(continues on next page)

(continued from previous page)

```
MyClass& operator=(const MyClass&) = delete;
```

2.4.4 结构体 VS. 类

总述

仅当只有数据成员时使用 `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, 而非函数调用. 除了构造函数, 析构函数, `Initialize()`, `Reset()`, `Validate()` 等类似的用于设定数据成员的函数外, 不能提供其它功能的函数.

如果需要更多的函数功能, `class` 更适合. 如果拿不准, 就用 `class`.

为了和 STL 保持一致, 对于仿函数等特性可以不用 `class` 而是使用 `struct`.

注意: 类和结构体的成员变量使用不同的命名规则.

2.4.5 结构体 VS. `std::pair` 和 `std::tuple`

如果成员可以取有意义的名字，则使用 `struct` 而不使用 `pair` 或者 `tuple` 。

2.4.6 继承

总述

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

定义

当子类继承基类时, 子类包含了父基类所有数据及操作的定义. C++ 实践中, 继承主要用于两种场合: 实现继承, 子类继承父类的实现代码; 接口继承, 子类仅继承父类的方法名称.

优点

实现继承通过原封不动的复用基类代码减少了代码量. 由于继承是在编译时声明, 程序员和编译器都可以理解相应操作并发现错误. 从编程角度而言, 接口继承是用来强制类输出特定的 API. 在类没有实现 API 中某个必须的方法时, 编译器同样会发现并报告错误.

缺点

对于实现继承, 由于子类的实现代码散布在父类和子类间之间, 要理解其实现变得更加困难. 子类不能重写父类的非虚函数, 当然也就不能修改其实现. 基类也可能定义了一些数据成员, 因此还必须区分基类的实际布局.

结论

所有继承必须是 `public` 的. 如果你想使用私有继承, 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.

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

对于重载的虚函数或虚析构函数, 使用 `override`, 或 (较不常用的) `final` 关键字显式地进行标记. 较早 (早于 C++11) 的代码可能会使用 `virtual` 关键字作为不得已的选项. 因此, 在声明重载时, 请使用 `override`, `final` 或 `virtual` 的其中之一进行标记. 标记为 `override` 或 `final` 的析构函数如果不是对基类虚函数的重载的话, 编译会报错, 这有助于捕获常见的错误. 这些标记起到了文档的作用, 因为如果省略这些关键字, 代码阅读者不得不检查所有父类, 以判断该函数是否是虚函数.

2.4.7 多重继承

总述

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. 有时多重实现继承看上去是不错的解决方案, 但这时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.

2.4.8 接口

总述

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

定义

当一个类满足以下要求时, 称之为纯接口:

- 只有纯虚函数 ("`=0`") 和静态函数 (除了下文提到的析构函数).
- 没有非静态数据成员.
- 没有定义任何构造函数. 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, 接口属性作为实现细节不应暴露给用户.

结论

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

2.4.9 运算符重载

总述

除少数特定环境外, 不要重载运算符, 也不要创建用户定义字面量.

定义

C++ 允许用户通过使用 `operator` 关键字 对内建运算符进行重载定义 , 只要其中一个参数是用户定义的类型. `operator` 关键字还允许用户使用 `operator""` 定义新的字面运算符, 并且定义类型转换函数, 例如 `operator bool()`.

优点

重载运算符可以让代码更简洁易懂, 也使得用户定义的类型和内建类型拥有相似的行为. 重载运算符对于某些运算来说是符合语言习惯的名称 (例如 ==, <, =, <<), 遵循这些语言约定可以让用户定义的类型更易读, 也能更好地和需要这些重载运算符的函数库进行交互操作.

对于创建用户定义的类型的对象来说, 用户定义字面量是一种非常简洁的标记.

缺点

- 要提供正确, 一致, 不出现异常行为的操作符运算需要花费不少精力, 而且如果达不到这些要求的话, 会导致令人迷惑的 Bug.
- 过度使用运算符会带来难以理解的代码, 尤其是在重载的操作符的语义与通常的约定不符合时.
- 函数重载有多少弊端, 运算符重载就至少有多少.
- 运算符重载会混淆视听, 让你误以为一些耗时的操作和操作内建类型一样轻巧.
- 对重载运算符的调用点的查找需要的可就不仅仅是像 grep 那样的程序了, 这时需要能够理解 C++ 语法的搜索工具.
- 如果重载运算符的参数写错, 此时得到的可能是一个完全不同的重载而非编译错误. 例如: `foo < bar` 执行的是一个行为, 而 `&foo < &bar` 执行的就是完全不同的另一个行为了.
- 重载某些运算符本身就是有害的. 例如, 重载一元运算符 `&` 会导致同样的代码有完全不同的含义, 这取决于重载的声明对某段代码而言是否是可见的. 重载诸如 `&&`, `||` 和 `,` 会导致运算顺序和内建运算的顺序不一致.
- 运算符从通常定义在类的外部, 所以对于同一运算, 可能出现不同的文件引入了不同的定义的风险. 如果两种定义都链接到同一二进制文件, 就会导致未定义的行为, 有可能表现为难以发现的运行时错误.
- 用户定义字面量所创建的语义形式对于某些有经验的 C++ 程序员来说都是很陌生的.

结论

只有在意义明显, 不会出现奇怪的行为并且与对应的内建运算符的行为一致时才定义重载运算符. 例如, `|` 要作为位或或逻辑或来使用, 而不是作为 shell 中的管道.

只有对用户自己定义的类型重载运算符. 更准确地说, 将它们和它们所操作的类型定义在同一个头文件中, `.cpp` 中和命名空间中. 这样做无论类型在哪里都能够使用定义的运算符, 并且最大程度上避免了多重定义的风险. 如果可能的话, 请避免将运算符定义为模板, 因为此时它们必须对任何模板参数都能够作用. 如果你定义了一个运算符, 请将其相关且有意义的运算符都进行定义, 并且保证这些定义的语义是一致的. 例如, 如果你重载了 <, 那么请将所有的比较运算符都进行重载, 并且保证对于同一组参数, < 和 > 不会同时返回 `true`.

建议不要将不进行修改的二元运算符定义为成员函数. 如果一个二元运算符被定义为类成员, 这时隐式转换会作用于右侧的参数却不会作用于左侧. 这时会出现 `a < b` 能够通过编译而 `b < a` 不能的情况, 这是很让人迷惑的.

不要为了避免重载操作符而走极端. 比如说, 应当定义 ==, =, 和 << 而不是 `Equals()`, `CopyFrom()` 和 `PrintTo()`. 反过来说, 不要只是为了满足函数库需要而去定义运算符重载. 比如说, 如果你的类型没有自然顺序, 而你要将它们存入 `std::set` 中, 最好还是定义一个自定义的比较运算符而不是重载 <.

不要重载 `&&`, `||`, `,` 或一元运算符 `&`. 不要重载 `operator""`, 也就是说, 不要引入用户定义字面量.

类型转换运算符在隐式类型转换 一节有提及. = 运算符在可拷贝类型和可移动类型 一节

有提及. 运算符 << 在流 一节有提及. 同时请参见函数重载 一节, 其中提到的的规则对运算符重载同样适用.

2.4.10 存取控制

总述

将 所有数据成员声明为 `private`, 除非是 `static const` 类型成员 (遵循常量命名规则). 出于技术上的原因, 在使用 Google Test 时我们允许测试固件类中的数据成员为 `protected`.

2.4.11 声明顺序

总述

将相似的声明放在一起, 将 `public` 部分放在最前.

说明

类定义一般应以 `public:` 开始, 后跟 `protected:`, 最后是 `private:`. 省略空部分.

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

不要将大段的函数定义内联在类定义中. 通常，只有那些普通的, 或性能关键且短小的函数可以内联在类定义中. 参见内联函数 一节.

2.5 函数

2.5.1 输入和输出

总述

我们倾向于按值返回，否则按引用返回。避免返回指针，除非它可以为空.

说明

C++ 函数由返回值提供天然的输出，有时也通过输出参数（或输入/输出参数）提供. 我们倾向于使用返回值而不是输出参数：它们提高了可读性，并且通常提供相同或更好的性能.

C/C++ 中的函数参数或者是函数的输入, 或者是函数的输出, 或兼而有之. 非可选输入参数通常是值参或 `const` 引用, 非可选输出参数或输入/输出参数通常应该是引用（不能为空）. 对于可选的参数, 通常使用 `std::optional` 来表示可选的按值输入，使用 `const` 指针来表示可选的其他输入. 使用非常量指针来表示可选输出和可选输入/输出参数.

避免定义需要 `const` 引用参数去超出生命周期的函数，因为 `const` 引用参数与临时变量绑定. 相反，要找到某种方法来消除生命周期要求（例如，通过复制参数），或者通过 `const` 指针传递它并记录生命周期和非空要求.

在排序函数参数时，将所有输入参数放在所有输出参数之前. 特别要注意, 在加入新参数时不要因为它们是新参数就置于参数列表最后, 而是仍然要按照前述的规则, 即将新的输入参数也置于输出参数之前.

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

2.5.2 编写简短函数

总述

我们倾向于编写简短, 凝练的函数.

说明

我们承认长函数有时是合理的, 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, 以便于他人阅读和修改代码.

在处理代码时, 你可能会发现复杂的长函数. 不要害怕修改现有代码: 如果证实这些代码使用 / 调试起来很困难, 或者你只需要使用其中的一小段代码, 考虑将其分割为更加简短并易于管理的若干函数.

2.5.3 引用参数

总述

所有按引用传递的参数必须加上 `const`.

定义

在 C 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, 函数还可以声明为引用参数: `int foo(int &val)`.

优点

定义引用参数可以防止出现 `(*pval)++` 这样丑陋的代码. 引用参数对于拷贝构造函数这样的应用也是必需的. 同时也更明确地不接受空指针.

缺点

容易引起误解, 因为引用在语法上是值变量却拥有指针的语义.

结论

函数参数列表中, 所有引用参数都必须是 `const`:

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

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, 除非特殊要求, 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:

- 可能会传递空指针.

- 函数要把指针或对地址的引用赋值给输入形参.

总而言之, 大多时候输入形参往往是 `const T&`. 若用 `const T*` 则说明输入另有处理. 所以若要使用 `const T*`, 则应给出相应的理由, 否则会使得读者感到迷惑.

2.5.4 函数重载

总述

若要使用函数重载, 则必须能让读者一看调用点就胸有成竹, 而不用花心思猜测调用的重载函数到底是哪一种. 这一规则也适用于构造函数.

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*` 的函数对其进行重载:

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

优点

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

缺点

如果函数单靠不同的参数类型而重载 (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, 如果派生类只重载了某个函数的部分变体, 继承语义就容易令人困惑.

结论

如果打算重载一个函数, 可以试试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()`. 如果重载函数的目的是为了支持不同数量的同一类型参数, 则优先考虑使用 `std::vector` 以便使用者可以用列表初始化 指定参数.

2.5.5 缺省参数

总述

只允许在非虚函数中使用缺省参数, 且必须保证缺省参数的值始终一致. 缺省参数与函数重载 遵循同样的规则. 一般情况下建议使用函数重载, 尤其是在缺省函数带来的可读性提升不能弥补下文中所提到的缺点的情况下.

优点

有些函数一般情况下使用默认参数, 但有时需要又使用非默认的参数. 缺省参数为这样的情形提供了便利, 使程序员不需要为了极少的例外情况编写大量的函数. 和函数重载相比, 缺省参数的语法更简洁明了, 减少了大量的样板代码, 也更好地区别了 “必要参数” 和 “可选参数” .

缺点

缺省参数实际上是函数重载语义的另一种实现方式, 因此所有*不应当使用函数重载的理由* 也都适用于缺省参数.

虚函数调用的缺省参数取决于目标对象的静态类型, 此时无法保证给定函数的所有重载声明的都是同样的缺省参数.

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

缺省参数会干扰函数指针, 导致函数签名与调用点的签名不一致. 而函数重载不会导致这样的问题.

结论

对于虚函数, 不允许使用缺省参数, 因为在虚函数中缺省参数不一定能正常工作. 如果在每个调用点缺省参数的值都有可能不同, 在这种情况下缺省函数也不允许使用. (例如, 不要写像 `void f(int n = counter++);` 这样的代码.)

在其他情况下, 如果缺省参数对可读性的提升远远超过了以上提及的缺点的话, 可以使用缺省参数. 如果仍有疑惑, 就使用函数重载.

2.5.6 函数返回类型后置语法

总述

只有在常规写法 (返回类型前置) 不便于书写或不便于阅读时使用返回类型后置语法.

定义

C++ 现在允许两种不同的函数声明方式. 以往的写法是将返回类型置于函数名之前. 例如:

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

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

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

后置返回类型为函数作用域. 对于像 `int` 这样简单的类型, 两种写法没有区别. 但对于复杂的情况, 例如类域中的类型声明或者以函数参数的形式书写的类型, 写法的不同会造成区别.

优点

后置返回类型是显式地指定*Lambda 表达式* 的返回值的唯一方式. 某些情况下, 编译器可以自动推导出 Lambda 表达式的返回类型, 但并不是在所有的情况下都能实现. 即使编译器能够自动推导, 显式地指定返回类型也能让读者更明了.

有时在已经出现了的函数参数列表之后指定返回类型, 能够让书写更简单, 也更易读, 尤其是在返回类型依赖于模板参数时. 例如:

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

对比下面的例子:

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

缺点

后置返回类型相对来说是非常新的语法, 而且在 C 和 Java 中都没有相似的写法, 因此可能对读者来说比较陌生.

在已有的代码中有大量的函数声明, 你不可能把它们都用新的语法重写一遍. 因此实际的做法只能是使用旧的语法或者新旧混用. 在这种情况下, 只使用一种版本是相对来说更规整的形式.

结论

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

2.6 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.

2.6.1 所有权与智能指针

总述

动态分配出的对象最好有单一且固定的所有主, 并通过智能指针传递所有权.

定义

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

智能指针是一个通过重载 `*` 和 `->` 运算符以表现得如指针一样的类. 智能指针类型被用来自动化所有权的登记工作, 来确保执行销毁义务到位. std::unique_ptr 是 C++11 新推出的一种智能指针类型, 用来表示动态分配出的对象的独一无二的所有权; 当 `std::unique_ptr` 离开作用域时, 对象就会被销毁. `std::unique_ptr` 不能被复制, 但可以把它移动 (move) 给新所有主. std::shared_ptr 同样表示动态分配对象的所有权, 但可以被共享, 也可以被复制; 对象的所有权由所有复制者共同拥有, 最后一个复制者被销毁时, 对象也会随着被销毁.

优点

- 如果没有清晰、逻辑条理的所有权安排, 不可能管理好动态分配的内存.
- 传递对象的所有权, 开销比复制来得小, 如果可以复制的话.
- 传递所有权也比”借用”指针或引用来得简单, 毕竟它大大省去了两个用户一起协调对象生命周期的工作.
- 如果所有权逻辑条理, 有文档且不紊乱的话, 可读性会有很大提升.
- 可以不用手动完成所有权的登记工作, 大大简化了代码, 也免去了一大波错误之恼.
- 对于 const 对象来说, 智能指针简单易用, 也比深度复制高效.

缺点

- 不得不用指针（不管是智能的还是原生的）来表示和传递所有权. 指针语义可要比值语义复杂得许多了, 特别是在 API 里：这时不光要操心所有权, 还要顾及别名, 生命周期, 可变性以及其它大大小小的问题.
- 其实值语义的开销经常被高估, 所以所有权传递带来的性能提升不一定能弥补可读性和复杂度的损失.
- 如果 API 依赖所有权的传递, 就会害得客户端不得不用单一的内存管理模型.
- 如果使用智能指针, 那么资源释放发生的位置就会变得不那么明显.
- `std::unique_ptr` 的所有权传递原理是 C++11 的 move 语法, 后者毕竟是刚刚推出的, 容易迷惑程序员.
- 如果原本的所有权设计已经够完善了, 那么若要引入所有权共享机制, 可能不得不重构整个系统.
- 所有权共享机制的登记工作在运行时进行, 开销可能相当大.
- 某些极端情况下 (例如循环引用), 所有权被共享的对象永远不会被销毁.
- 智能指针并不能够完全代替原生指针.

结论

如果必须使用动态分配, 那么更倾向于将所有权保持在分配者手中. 如果其他地方要使用这个对象, 最好传递它的拷贝, 或者传递一个不用改变所有权的指针或引用. 倾向于使用 `std::unique_ptr` 来明确所有权传递, 例如:

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

如果没有很好的理由, 则不要使用共享所有权. 这里的理由可以是为了避免开销昂贵的拷贝操作, 但是只有当性能提升非常明显, 并且操作的对象是不可变的（比如说 `std::shared_ptr<const Foo>`）时候, 才能这么做. 如果确实要使用共享所有权, 建议于使用 `std::shared_ptr`.

不要使用 `std::auto_ptr`, 使用 `std::unique_ptr` 代替它.

2.6.2 Cpplint

总述

使用 `cpplint.py` 检查风格错误.

说明

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

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.

2.7 其他 C++ 特性

2.7.1 右值引用

Tip: 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.

2.7.2 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.

2.7.3 异常

Tip: 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，但是前者要求在堆栈分配内存，后者会导致刚创建的实例处于”无效“状态。
- 在测试框架里很好用。

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。

2.7.4 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. 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.

2.7.5 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.

至于 dynamic_cast 参见运行时类型识别.

2.7.6 IO 流

Tip: 不使用 IO 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. 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.

2.7.7 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).

2.7.8 const 用法

Tip: 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`:

- 如果函数不会修改你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! (Yang.Y 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.)

2.7.9 constexpr 用法

Tip: 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；当前对 constexpr 函数和构造函数中允许的限制可能会导致这些定义中解决的方法模糊。

结论

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。避免复杂的函数定义，以使其能够与 constexpr 一起使用。千万别痴心妄想地想靠 constexpr 来强制代码「内联」。

2.7.10 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`. 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!

2.7.11 6.15. 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:
```

(continues on next page)

(continued from previous page)

```
// 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。

- 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())` (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)

2.7.12 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.
- 不要用 `##` 处理函数，类和变量的名字。

2.7.13 0, `nullptr` 和 `NULL`

Tip: 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.

2.7.14 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. 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));
```

Warning:

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

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

2.7.15 auto

Tip: 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，比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<bool> 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。

2.7.16 列表初始化

Tip: 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"};
```

(continues on next page)

(continued from previous page)

```
// 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：

Warning:

```
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，参见列表初始化格式.

2.7.17 Lambda 表达式

Tip: 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 `[=](int x) {return x + n;}`, 您该写成 `[n](int x) {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.

2.7.18 模板编程

Tip: 不要使用复杂的模板编程

定义:

模板编程指的是利用 c++ 模板实例化机制是图灵完备性, 可以被用来实现编译时刻的类型判断的一系列编程技巧

优点:

模板编程能够实现非常灵活的类型安全的接口和极好的性能, 一些常见的工具比如 Google Test, std::tuple, std::function 和 Boost.Spirit. 这些工具如果没有模板是实现不了的

缺点:

- 模板编程所使用的技巧对于使用 c++ 不是很熟练的人是比较晦涩, 难懂的. 在复杂的地方使用模板的代码让人更不容易读懂, 并且 debug 和维护起来都很麻烦
- 模板编程经常会导致编译出错的信息非常不友好: 在代码出错的时候, 即使这个接口非常的简单, 模板内部复杂的实现细节也会在出错信息显示. 导致这个编译出错信息看起来非常难以理解.
- 大量的使用模板编程接口会让重构工具 (Visual Assist X, Refactor for C++ 等等) 更难发挥用途. 首先模板的代码会在很多上下文里面扩展开来, 所以很难确认重构对所有的这些展开的代码有用, 其次有些重构工具只对已经做过模板类型替换的代码的 AST 有用. 因此重构工具对这些模板实现的原始代码并不有效, 很难找出哪些需要重构.

结论:

- 模板编程有时候能够实现更简洁更易用的接口, 但是更多的时候却适得其反. 因此模板编程最好只用在少量的基础组件, 基础数据结构上, 因为模板带来的额外的维护成本会被大量的使用给分担掉
- 在使用模板编程或者其他复杂的模板技巧的时候, 你一定要再三考虑一下. 考虑一下你们团队成员的平均水平是否能够读懂并且能够维护你写的模板代码. 或者一个非 c++ 程序员和一些只是在出错的时候偶尔看一下代码的人能够读懂这些错误信息或者能够跟踪函数的调用流程. 如果你使用递归的模板实例化, 或者类型列表, 或者元函数, 又或者表达式模板, 或者依赖 SFINAE, 或者 sizeof 的 trick 手段来检查函数是否重载, 那么这说明你模板用的太多了, 这些模板太复杂了, 我们不推荐使用
- 如果你使用模板编程, 你必须考虑尽可能的把复杂度最小化, 并且尽量不要让模板对外暴露. 你最好只在实现里面使用模板, 然后给用户暴露的接口里面并不使用模板, 这样能提高你的接口的可读性. 并且你应该在这些使用模板的代码上写尽可能详细的注释. 你的注释里面应该详细的包含这些代码是怎么用的, 这些模板生成出来的代码大概是什么样子的. 还需要额外注意在用户错误使用你的模板代码的时候需要输出更人性化的出错信息. 因为这些出错信息也是你的接口的一部分, 所以你的代码必须调整到这些错误信息在用户看起来应该是非常容易理解, 并且用户很容易知道如何修改这些错误

2.7.19 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。

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

2.8 命名约定

最重要的一致性规则是命名管理. 命名的风格能让我们在不需要去查找类型声明的条件下快速地了解某个名字代表的含义: 类型, 变量, 函数, 常量, 宏, 等等, 甚至. 我们大脑中的模式匹配引擎非常依赖这些命名规则.

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

2.8.1 通用命名规则

总述

函数命名, 变量命名, 文件命名要有描述性; 少用缩写.

说明

尽可能使用描述性的命名, 别心疼空间, 毕竟相比之下让代码易于新读者理解更重要. 不要用只有项目开发者能理解的缩写, 也不要通过砍掉几个字母来缩写单词.

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

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

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

模板参数的命名应当遵循对应的分类: 类型模板参数应当遵循类型命名 的规则, 而非类型模板应当遵循变量命名 的规则.

2.8.2 文件命名

总述

文件名要全部小写, 可以包含下划线 (_) 或连字符 (-), 依照项目的约定. 如果没有约定, 那么 "_" 更好.

说明

可接受的文件命名示例:

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

C++ 文件要以 `.cpp` 结尾, 头文件以 `.h` 结尾. 专门插入文本的文件则以 `.inc` 结尾, 参见头文件自足.

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.cpp`, 对应于类 `FooBar`.

内联函数定义必须放在 `.h` 文件中. 如果内联函数比较短, 就直接将实现也放在 `.h` 中.

动态库和静态库使用 snake case 命名，例如:

```
libmy_useful_library.so
```

2.8.3 类型命名

总述

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

说明

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

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

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

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

(continues on next page)

(continued from previous page)

```
// 枚举
enum UrlTableErrors { ...
```

2.8.4 变量命名

总述

变量 (包括函数参数) 和数据成员名一律小写, 单词之间用下划线连接. 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_`.

说明

普通变量命名

举例:

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

string tableName;  // 差 - 混合大小写
```

类数据成员命名

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_;  // 好 - 后加下划线.
  string tablename_;   // 好.
  static Pool<TableInfo>* pool_;  // 好.
};
```

结构体变量命名

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;
  static Pool<UrlTableProperties>* pool;
};
```

结构体与类的使用讨论, 参考*结构体 vs. 类*.

2.8.5 常量命名

总述

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

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

说明

所有具有静态存储类型的变量 (例如静态变量或全局变量, 参见 存储类型) 都应当以此方式命名. 对于其他存储类型的变量, 如自动变量等, 这条规则是可选的. 如果不采用这条规则, 就按照一般的变量命名规则.

2.8.6 函数命名

总述

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().

说明

一般来说, 函数名的每个单词首字母大写 (即“驼峰变量名”或“帕斯卡变量名”), 没有下划线. 对于首字母缩写的单词, 更倾向于将它们视作一个单词进行首字母大写 (例如, 写作 StartRpc() 而非 StartRPC()).

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

(同样的命名规则同时适用于类作用域与命名空间作用域的常量, 因为它们是作为 API 的一部分暴露对外的, 因此应当让它们看起来像是一个函数, 因为在这时, 它们实际上是一个对象而非函数的这一事实对外不过是一个无关紧要的实现细节.)

取值和设值函数的命名与变量一致. 一般来说它们的名称与实际的成员变量对应, 但并不强制要求. 例如 int count() 与 void set_count(int count).

2.8.7 命名空间命名

总述

命名空间命名需要个参考包名和目录名, 注意 *不使用缩写作为名称* 的规则同样适用于命名空间. 命名空间中的代码极少需要涉及命名空间的名称, 因此没有必要在命名空间中使用缩写.

要避免嵌套的命名空间与常见的顶级命名空间发生名称冲突. 由于名称查找规则的存在, 命名空间之间的冲突完全有可能导致编译失败. 尤其是, 不要创建嵌套的 std 命名空间. 建议使用更独特的项目标识符 (websearch::index, websearch::index_util) 而非常见的极易发生冲突的名称 (比如 websearch::util).

对于 internal 命名空间, 要当心加入到同一 internal 命名空间的代码之间发生冲突 (由于内部维护人员通常来自同一团队, 因此常有可能导致冲突). 在这种情况下, 请使用文件名以使得内部名称独一无二 (例如对于 frobber.h, 使用 websearch::index::frobber_internal).

2.8.8 枚举命名

总述

枚举的命名应当和常量 一致，而不是跟宏。

说明

单独的枚举值应该采用常量 的命名方式。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,
};
```

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.

2.8.9 宏命名

总述

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.

说明

参考预处理宏; 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
```

2.8.10 命名规则的特例

总述

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 型实体; 参照 STL 命名约定

LONGLONG_MAX: 常量, 如同 INT_MAX

2.9 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, 但最好的代码应当本身就是文档. 有意义的类型名和变量名, 要远胜过要用注释解释的含糊不清的名字.

你写的注释是给代码读者看的, 也就是下一个需要理解你的代码的人. 所以慷慨些吧, 下一个读者可能就是你!

2.9.1 注释风格

总述

使用 // 或 /* */, 统一就好.

说明

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

2.9.2 文件注释

总述

在每一个文件开头加入版权公告:

```
// Copyright [year] MINIEYE
```

例如，现在是 2023 年，应该写做:

```
// Copyright 2023 MINIEYE
```

年份部分标记了版权开始的时间，从而不必在 2024 年更新上面的版权公告。

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

文件内容

如果一个 .h 文件声明了多个概念, 则文件注释应当对文件的内容做一个大致的说明, 同时说明各概念之间的联系. 一个一到两行的文件注释就足够了, 对于每个概念的详细文档应当放在各个概念中, 而不是文件注释中.

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

2.9.3 类注释

总述

每个类的定义都要附带一份注释, 描述类的功能和用法, 除非它的功能相当明显.

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

说明

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

如果你想用一小段代码演示这个类的基本用法或通常用法, 放在类注释里也非常合适.

如果类的声明和定义分开了 (例如分别放在了 `.h` 和 `.cpp` 文件中), 此时, 描述类用法的注释应当和接口定义放在一起, 描述类的操作和实现的注释应当和实现放在一起.

2.9.4 函数注释

总述

函数声明处的注释描述函数功能; 定义处的注释描述函数实现.

函数声明

基本上每个函数声明处前都应当加上注释, 描述函数的功能和用途. 只有在函数的功能简单而明显时才能省略这些注释 (例如, 简单的取值和设值函数). 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.
- 函数是否分配了必须由调用者释放的空间.
- 参数是否可以为空指针.
- 是否存在函数使用上的性能隐患.

- 如果函数是可重入的, 其同步前提是什么?

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

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

注释函数重载时, 注释的重点应该是函数中被重载的部分, 而不是简单的重复被重载的函数的注释. 多数情况下, 函数重载不需要额外的文档, 因此也没有必要加上注释.

注释构造/析构函数时, 切记读代码的人知道构造/析构函数的功能, 所以 “销毁这一对象” 这样的注释是没有意义的. 你应当注明的是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

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

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.

2.9.5 变量注释

总述

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.

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. 如果有非变量的参数 (例如特殊值, 数据成员之间的关系, 生命周期等) 不能够用类型与变量名明确表达, 则应当加上注释. 然而, 如果变量类型与变量名已经足以描述一个变量, 那么就不再需要加上注释.

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

```
private:
 // Used to bounds-check table accesses. -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, 以及作为全局变量的原因. 比如:

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

2.9.6 实现注释

总述

对于代码中巧妙的, 晦涩的, 有趣的, 重要的地方加以注释.

说明

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();  // 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.
}
std::vector<string> list{
    // Comments in braced lists describe the next element...
    "First item",
    // .. and should be aligned appropriately.
    "Second item"};
DoSomething(); /* For trailing block comments, one space is fine. */
```

函数参数注释

如果函数参数的意义不明显, 考虑用下面的方式进行弥补:

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

比如下面的示例的对比:

```
// What are these arguments?
const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);
```

和

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

哪个更清晰一目了然.

不允许的行为

不要描述显而易见的现象, 永远不要用自然语言翻译代码作为注释, 除非即使对深入理解 C++ 的读者来说代码的行为都是不明显的. 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:

你所提供的注释应当解释代码 为什么要这么做和代码的目的, 或者最好是让代码自文档化.

比较这样的注释:

```
// Find the element in the vector.  <-- 差: 这太明显了!
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

和这样的注释:

```
// Process "element" unless it was already processed.
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

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

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

2.9.7 标点, 拼写和语法

总述

注意标点, 拼写和语法; 写的好的注释比差的要易读的多.

说明

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

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很大帮助.

2.9.8 TODO 注释

总述

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名字, 邮件地址, bug ID, 或其它身份标识和与这一 TODO 相关的 issue. 主要目的是让添加注释的人 (也是可以请求提供更多细节的人) 可根据规范的 TODO 格式进行查找. 添加 TODO 注释并不意味着你要自己来修正, 因此当你加上带有姓名的 TODO 时, 一般都是写上自己的名字.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
// TODO(bug 12345): remove the "Last visitors" feature
```

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.”).

2.9.9 弃用注释

总述

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

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, 在括号中留下您的名字, 邮箱地址以及其他身份标识.

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

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.

2.10 格式

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

为了帮助你正确的格式化代码, 我们写了一个 emacs 配置文件.

2.10.1 格式化工具

使用 clang-format (至少 v14.0.0) 格式化新增的代码，并且使用下面的配置:

```
---
Language:                        Cpp
BasedOnStyle:                    Google
ColumnLimit:                     120
```

特别注意，如果某个项目不遵守这个格式，需要随着修改逐步的转变代码格式，但不能一次性粗暴的提交转变，那会让合并代码变得非常困难。

- 例一，当某个提交新增了一部分代码，将新增代码格式化为正确的格式。
- 例二，当某个提交修改了一部分代码，将附近代码，例如一个不长的函数内部，或者一个循环内部，格式化为正确的格式。基本上，在同一个作用域（大括号内）的代码视作是附近代码。

下文的格式化要求如果跟这份配置的表现不同，则遵照配置的要求，并可以提交修改下面的配置。但应该注意，并不是所有格式要求都能被 clang-format 修改好，往往写代码时养成良好的格式化习惯也有助于保持头脑清晰，因此你还是需要阅读这些格式化要求。

2.10.2 行长度

每一行代码字符数不超过 120.

120 个字符是最大值.

如果无法在不伤害易读性的条件下进行断行, 那么注释行可以超过 120 个字符, 这样可以方便复制粘贴. 例如, 带有命令示例或 URL 的行可以超过 120 个字符.

包含长路径的 `#include` 语句可以超出 120 列.

头文件保护 可以无视该原则.

2.10.3 非 ASCII 字符

总述

尽量不使用非 ASCII 字符, 使用时必须使用 UTF-8 编码.

说明

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

十六进制编码也可以, 能增强可读性的情况下尤其鼓励——比如 `"\xEF\xBB\xBF"`, 或者更简洁地写作 `u8"\uFEFF"`, 在 Unicode 中是 零宽度无间断的间隔符号, 如果不用十六进制直接放在 UTF-8 格式的源文件中, 是看不到的.

使用 u8 前缀把带 uXXXX 转义序列的字符串字面值编码成 UTF-8. 不要用在本身就带 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, 除非你写的代码要调用 Windows API, 后者广泛使用了 wchar_t.

2.10.4 空格还是制表位

总述

只使用空格, 每次缩进 2 个空格.

说明

我们使用空格缩进. 不要在代码中使用制表符. 你应该设置编辑器将制表符转为空格.

2.10.5 函数声明与定义

总述

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, 分行方式与函数调用 一致.

说明

函数看上去像这样:

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

如果同一行文本太多, 放不下所有参数:

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

甚至连第一个参数都放不下:

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

注意以下几点:

- 使用好的参数名.
- 只有在参数未被使用或者其用途非常明显时, 才能省略参数名.
- 如果返回类型和函数名在一行放不下, 分行.
- 如果返回类型与函数声明或定义分行了, 不要缩进.
- 左圆括号总是和函数名在同一行.
- 函数名和左圆括号间永远没有空格.
- 圆括号与参数间没有空格.
- 左大括号总在最后一个参数同一行的末尾处, 不另起新行.
- 右大括号总是单独位于函数最后一行, 或者与左大括号同一行.
- 右圆括号和左大括号间总是有一个空格.
- 所有形参应尽可能对齐.
- 缺省缩进为 2 个空格.
- 换行后的参数保持 4 个空格的缩进.

未被使用的参数, 或者根据上下文很容易看出其用途的参数, 可以省略参数名:

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

未被使用的参数如果其用途不明显的话, 在函数定义处将参数名注释起来:

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

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

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

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

属性, 和展开为属性的宏, 写在函数声明或定义的最前面, 即返回类型之前:

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

2.10.6 Lambda 表达式

总述

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

说明

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());
```

2.10.7 函数调用

总述

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.

说明

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 (...) {
```

(continues on next page)

(continued from previous page)

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

把多个参数放在同一行以减少函数调用所需的行数, 除非影响到可读性. 有人认为把每个参数都独立成行, 不仅更好读, 而且方便编辑参数. 不过, 比起所谓的参数编辑, 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, 那也可以如此做. 参数的格式处理应当以可读性而非其他作为最重要的原则.

此外, 如果一系列参数本身就有一定的结构, 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);
```

2.10.8 列表初始化格式

总述

您平时怎么格式化函数调用, 就怎么格式化*列表初始化*.

说明

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

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

// 当不得不断行时.
SomeFunction(
    {"assume a zero-length name before {"},  // 假设在 { 前有长度为零的名字.
```

(continues on next page)

(continued from previous page)

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

2.10.9 条件语句

总述

倾向于不在圆括号内使用空格. 关键字 `if` 和 `else` 另起一行.

说明

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

最常见的是没有空格的格式. 哪一种都可以, 最重要的是 保持一致. 如果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. 还在犹豫的话, 就不要加空格了.

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

如果你更喜欢在圆括号内部加空格:

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

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

```
if(condition)     // 差 - IF 后面没空格.
if (condition){   // 差 - { 前面没空格.
if(condition){    // 变本加厉地差.
```

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

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

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

如果语句有 `else` 分支则不允许:

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

通常, 单行语句不需要使用大括号, 如果你喜欢用也没问题; 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 空格缩进.

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
}
```

2.10.10 循环和开关选择语句

总述

switch 语句可以使用大括号分段, 以表明 cases 之间不是连在一起的. 在单语句循环里, 括号可用可不用. 空循环体应使用 {} 或 continue.

说明

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, 编译器将给出 warning). 如果 default 应该永远执行不到, 简单的加条 assert:

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

在单语句循环里, 括号可用可不用:

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

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

空循环体应使用 {} 或 continue, 而不是一个简单的分号.

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

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

2.10.11 指针和引用表达式

总述

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.

说明

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:

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

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

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

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.

2.10.12 布尔表达式

总述

如果一个布尔表达式超过标准行宽, 断行方式要统一一下.

说明

下例中, 逻辑与 (&&) 操作符总位于行尾:

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

注意, 上例的逻辑与 (`&&`) 操作符均位于行尾. 这个格式在 Google 里很常见, 虽然把所有操作符放在开头也可以. 可以考虑额外插入圆括号, 合理使用的话对增强可读性是很有帮助的. 此外, 直接用符号形式的操作符, 比如 `&&` 和 ~, 不要用词语形式的 `and` 和 `compl`.

2.10.13 函数返回值

总述

不要在 `return` 表达式里加上非必须的圆括号.

说明

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

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

```
return (value);                // 毕竟您从来不会写 var = (value);
return(result);                // return 可不是函数!
```

2.10.14 变量及数组初始化

总述

用 =, `()` 和 `{}` 均可.

说明

您可以用 =, `()` 和 `{}`, 以下的例子都是正确的:

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

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

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

此外, 列表初始化不允许整型类型的四舍五入, 这可以用来避免一些类型上的编程失误.

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

2.10.15 预处理指令

总述

预处理指令不要缩进, 从行首开始.

说明

即使预处理指令位于缩进代码块中, 指令也应从行首开始.

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

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

2.10.16 类格式

总述

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

说明

类声明 (下面的代码中缺少注释, 参考类注释) 的基本格式如下:

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

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

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

 private:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
};
```

注意事项:

- 所有基类名应在 120 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.

2.10.17 构造函数初始值列表

总述

构造函数初始化列表放在同一行或按四格缩进并排多行.

说明

下面两种初始值列表方式都可以接受:

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

// 如果不能放在同一行,
// 必须置于冒号后, 并缩进 4 个空格
```

(continues on next page)

(continued from previous page)

```
MyClass::MyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
  DoSomething();
}

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

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

2.10.18 命名空间格式化

总述

命名空间内容不缩进.

说明

命名空间 不要增加额外的缩进层次，例如:

```
namespace {

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

}  // namespace
```

不要在命名空间内缩进:

```
namespace {

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

}  // namespace
```

声明嵌套命名空间时，每个命名空间都独立成行.

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

2.10.19 水平留白

总述

水平留白的使用根据在代码中的位置决定. 永远不要在行尾添加没意义的留白.

通用

```
void f(bool b) {  // 左大括号前总是有空格.
  ...
int i = 0;  // 分号前不加空格.
// 列表初始化中大括号内的空格是可选的.
// 如果加了空格, 那么两边都要加上.
int x[] = { 0 };
int x[] = {0};

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

添加冗余的留白会给其他人编辑时造成额外负担. 因此, 行尾不要留空格. 如果确定一行代码已经修改完毕, 将多余的空格去掉; 或者在专门清理空格时去掉（尤其是在没有其他人在处理这件事的时候). (Yang.Y 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) {          // if 条件语句和循环语句关键字后均有空格.
} else {          // else 前后有空格.
}
while (test) {}   // 圆括号内部不紧邻空格.
switch (i) {
for (int i = 0; i < 5; ++i) {
switch ( i ) {    // 循环和条件语句的圆括号里可以与空格紧邻.
if ( test ) {     // 圆括号, 但这很少见. 总之要一致.
for ( int i = 0; i < 5; ++i ) {
for ( ; i < 5 ; ++i) {  // 循环里内 ; 后恒有空格, ;  前可以加个空格.
```

(continues on next page)

(continued from previous page)

```
switch (i) {
  case 1:          // switch case 的冒号前无空格.
    ...
  case 2: break;   // 如果冒号有代码, 加个空格.
```

操作符

```
// 赋值运算符前后总是有空格.
x = 0;

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

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

模板和转换

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

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

2.10.20 垂直留白

总述

垂直留白越少越好.

说明

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.

下面的规则可以让加入的空行更有效:

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

2.11 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.

2.11.1 现有不合规范的代码

总述

对于现有不符合既定编程风格的代码可以网开一面.

说明

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, 一致性也包括原有的一致性.

2.12 结束语

运用常识和判断力, 并且 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, 那么你同样要这么做.

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. 我们展示的是一个总体的的风格规范, 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, 也让打乱读者在阅读代码时的节奏, 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!

2.13 2023-04-12 评审

本节列出 2023-04-12 评审的决定。按照少数服从多数的原则，记录了审议。注意本节内容不会跟之前正文冲突，也不会记录评审的比分。

- 前置声明：建议尽量不使用前向声明。
- 格式化工具：显式约定了代码格式化工具（clang-format），并且约定了它的配置文件。
- 行长度: 要求 120 而不是 80 或者 100。
- 头文件保护: 头文件必须有宏保护，隐含了禁止使用 `#pragma once`。
- 空格还是制表位：要求缩进 2 个空格，而不是 4 个。
- C++ 标准版本：要求 C++ 11，而不是 C++ 14。
- 行注释：要求名字空间结尾注释是:

```
} // namespace NAMESPACE
```

而不是:

```
} // namespace NAMESPACE
```

- 条件语句：使用下面的风格:

```
if (condition) {
  Code;
} else {
  Code;
}
```

而不是:

```
if (condition)
{
  Code;
}
else
{
  Code;
}
```

- 文件命名：C++ 代码文件的后缀名是.cpp 而不是.cc。
- 编写简短函数：建议的函数规模是小于 80 行。
- 常量命名：枚举变量命名，使用下面的命名方法:

```
kDaysInAWeek       // 3. "k" and pascal case.
```

- 函数命名：访问函数和修改函数使用变量的命名方法与一般的类成员函数不同，使用:

```
// G/V
int Foo::set_count(int count);
int Foo::count() const;
```

而不是:

```
// O
int Foo::SetCount(int count);
int Foo::GetCount() const;
```

- 类数据成员命名：要求成员变量和类静态成员变量采取后加下划线的格式:

```
int member_;
static int static_member_;
```

而不是:

```
// O
int m_member;
static int s_static_member;
```