

coding style

与其说是代码风格，不如说是代码规范

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。

`.cc` 是 unix 系统常用的 c++ 文件

`.cpp` 是非 unix 系统常用的 c++ 文件

! Tip

所有头文件都应该使用 `#define` 来防止头文件被多重包含，命名格式应当是：

```
<PROJECT>_<PATH>_<FILE>.h .
```

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

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

! Tip

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。

! 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。

对于目前的编译器来说，不需要自己定义内联函数

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: 相关头文件, C 库, C++ 库, 其他库的 .h, 本项目内的 .h.

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

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

又如, `dir/foo.cc` 或 `dir/foo_test.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

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

5. 在 `#include` 中插入空行以分割相关头文件, C 库, C++ 库, 其他库的 `.h` 和本项目内的 `.h` 是个好习惯。

鼓励在 `.cc` 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) 。

- 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, 以区别于其它命名空间:

```
// .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
```

更复杂的 `.cc` 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
```

`gflag`这一条不需要看

- 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;
```

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 `.cc` 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
```

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 [2.1. 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 {
void Function1();
void Function2();
} // namespace foo_bar
} // namespace myproject
```

而非

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

上面这一点很关键，类的静态方法不应该模拟命名空间的效果，而是与静态数据紧密相关。

💡 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。这使得代码浏览器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正常地声明，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);
}
```

! 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。

! Note

Xris 译注：

同一个编译单元内是明确的，静态初始化优先于动态初始化，初始化顺序按照声明顺序进行，销毁则逆序。不同的编译单元之间初始化和销毁顺序属于未明确行为 (unspecified behaviour)。

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);
```

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. 如果让类型可拷贝, 同时移动操作的效率高于拷贝操作, 那么就把移动的两个操作 (移动构造函数和赋值操作) 也给出定义. 如果类型不可拷贝, 但是移动操作的正确性对用户显然可见, 那么把这个类型设置为只可移动并定义移动的两个操作.

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

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.

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.

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.

有关重载运算符, 尽量不用就好

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, `.cc` 中和命名空间中. 这样做无论类型在哪里都能够使用定义的运算符, 并且最大程度上避免了多重定义的风险. 如果可能的话, 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域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, 其中提到的规则对运算符重载同样适用.

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.

类的小结

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;

4.2. 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.

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.

4. 其实我主张指针／地址操作符与变量名紧邻, `int* a, b` vs `int *a, b`, 新手会误以为前者的 `b` 是 `int *` 变量, 但后者就不一样了, 高下立判.

缩进用4个空格, 不用2个

操作符两侧有空格 for中的分号后有空格 if和for与左括号之间有空格

不违背上面条件情况下看这个

<https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/formatting/#id9>

7 8 9章

if else 尽量用大括号并且大括号换行