测量组C++编程风格指南

|  |  |  |
| --- | --- | --- |
| 文件状态  [√] 草稿文件  [ ] 正式文件  [ ] 更改正式文件 | 文件标识： |  |
| 当前版本： | 0.3 |
| 作 者： |  |
| 完成日期： | 2018年12月06日 |

版 本 历 史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本/状态 | 作者 | 参与者 | 起止日期 | 备注 |
| V 0.2  草稿文件 |  |  | 2018-12-04至  2018-12-06 | 参考《Google C++ Style Guide》 |
| V 0.3  草稿文件 |  |  | 2018-12-06 | 内容和格式整理 |
|  |  |  |  |  |
|  |  |  |  |  |

**目录**

[1. 头文件 6](#_Toc4352)

[1.1 Self-contained 头文件 6](#_Toc26899)

[1.2 #define 保护 6](#_Toc21026)

[1.3 前置声明 6](#_Toc25360)

[1.4 内联函数 7](#_Toc11303)

[2. 作用域 8](#_Toc31918)

[2.1 命名空间 8](#_Toc26980)

[2.2 匿名命名空间和静态变量 11](#_Toc9044)

[2.3 非成员函数、静态成员函数和全局函数 11](#_Toc25725)

[2.4 局部变量 12](#_Toc18053)

[2.5 静态和全局变量 13](#_Toc6293)

[3 类 14](#_Toc16346)

[3.1 构造函数的职责 14](#_Toc8155)

[3.2 隐式类型转换 15](#_Toc13276)

[3.3 可拷贝类型和可移动类型 16](#_Toc31612)

[3.4 结构体 VS 类 19](#_Toc31394)

[3.5 继承 19](#_Toc6223)

[3.6 多重继承 20](#_Toc22106)

[3.7 接口 21](#_Toc23987)

[3.8 存取控制 22](#_Toc12075)

[3.9 声明顺序 22](#_Toc16495)

[4 函数 22](#_Toc1805)

[4.1 参数顺序 22](#_Toc24991)

[4.2 编写简短函数 22](#_Toc11667)

[4.3 引用参数 23](#_Toc16877)

[4.4 函数重载 23](#_Toc15023)

[4.5 缺省参数 24](#_Toc5029)

[4.6 函数返回类型后置语法 25](#_Toc20735)

[5. 所有权与智能指针 26](#_Toc19052)

[5.1 总述 26](#_Toc27432)

[5.2 定义 26](#_Toc15771)

[5.3 优点 27](#_Toc21821)

[5.4 缺点 27](#_Toc6802)

[5.5 结论 28](#_Toc14931)

[6. 其他 C++ 特性 28](#_Toc3709)

[6.1 右值引用 28](#_Toc12200)

[6.2 缺省参数 29](#_Toc2911)

[6.3 变长数组 30](#_Toc12805)

[6.4 友元 30](#_Toc27234)

[6.5 运行时类型识别 30](#_Toc28492)

[6.6 类型转换 32](#_Toc32189)

[6.7 const语法 33](#_Toc17740)

[6.8 constexpr用法 34](#_Toc30382)

[6.9 整型 34](#_Toc26699)

[6.10 预处理宏 35](#_Toc26712)

[6.11 sizeof 35](#_Toc30452)

[6.12 auto 36](#_Toc13960)

[6.13 Lambda表达式 36](#_Toc11904)

[6.14 模板编程 36](#_Toc30639)

[7. 命名约定 37](#_Toc18768)

[7.1 通用命名规则 38](#_Toc19343)

[7.2 文件命名 38](#_Toc9057)

[7.3 类型命名 38](#_Toc29804)

[7.4 变量命名 39](#_Toc23351)

[7.5 常量命名 40](#_Toc11976)

[7.6 函数命名 40](#_Toc6954)

[7.7 命名空间命名 40](#_Toc25584)

[7.8 枚举命名 41](#_Toc22092)

[7.9 宏命名 41](#_Toc30171)

[7.10 命名规则的特例 41](#_Toc23864)

[8. 注释 41](#_Toc5939)

[8.1 注释风格 42](#_Toc4738)

[8.2 类注释 42](#_Toc30848)

[8.3 函数注释 42](#_Toc946)

[8.4 变量注释 44](#_Toc7115)

[8.5 实现注释 45](#_Toc4832)

[8.6 TODO 注释 47](#_Toc27188)

[8.7 弃用注释 48](#_Toc32367)

[9. 格式 48](#_Toc17226)

[9.1 文件格式 48](#_Toc16258)

[9.2 非ASCII字符 48](#_Toc15408)

[9.3 空格还是制表位 48](#_Toc15636)

[9.4 函数声明与定义 48](#_Toc13517)

[9.5 Lambda表达式 50](#_Toc225)

[9.6 函数调用 50](#_Toc23451)

[9.7 列表初始化格式 51](#_Toc17533)

[9.8 条件语句 52](#_Toc15570)

[9.9 循环和开关选择语句 53](#_Toc8276)

[9.10 指针和引用表达式 54](#_Toc8191)

[9.11 布尔表达式 55](#_Toc13151)

[9.12 函数返回值 55](#_Toc20298)

[9.13 变量及数组初始化 55](#_Toc4566)

[9.14 预处理指令 56](#_Toc32250)

[9.15 类格式 56](#_Toc31275)

[9.16 构造函数初始值列表 57](#_Toc28656)

[9.17 命名空间格式化 58](#_Toc2825)

[9.18 水平留白 59](#_Toc21046)

## 头文件

### 1.1 Self-contained 头文件

头文件应该能够自给自足（self-contained，也就是可以作为第一个头文件被引入），以.h结尾。使用某头文件，不要求定义任何特别symbols，不要求使用者包含任何其他头文件，需要包含的该文件自己已经包含。

如果是一个小项目，可能察觉不到这样做的差异。但在讲求职责清晰，分工协作的大型项目下，它就会变得很有价值。

### 1.2 #define 保护

所有头文件都应该使用#define来防止头文件被多重包含，命名格式应当是：\_<PROJECT>\_<FILE>\_H\_。

例如，项目foo中的头文件src/bar/baz.h应按如下方式保护：

#ifndef \_FOO\_BAZ\_H\_

#define \_FOO\_BAZ\_H\_

...

#endif

### 1.3 前置声明

尽可能地避免使用前置声明。使用#include包含需要的头文件即可。

#### 1.3.1 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。

#### 1.3.2 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。

#### 1.3.3 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。
* 极端情况下，用前置声明代替includes甚至都会暗暗地改变代码的含义：

// b.h:

struct B {};

struct D : B {}

// good\_user.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。

#### 1.3.4 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。

### 1.4 内联函数

只有当函数只有10行甚至更少时才将其定义为内联函数。

#### 1.4.1 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。

#### 1.4.2 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。

#### 1.4.3 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。

#### 1.4.4 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。（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。

## 作用域

### 2.1 命名空间

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。建议不使用using指示（using-directive）。禁止使用内联命名空间（inline namespace）。

#### 2.1.1 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。

#### 2.1.2 优点

虽然类已经提供了（可嵌套的）命名轴线（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

{

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。

#### 2.1.3 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))。

#### 2.1.4 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。
* 用命名空间把文件包含，以及类的前置声明以外的整个源文件封装起来，以区别于其它命名空间:

// .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

// .cpp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.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

{

...

} // 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。
* 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

### 2.2 匿名命名空间和静态变量

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。

#### 2.2.1 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。

#### 2.2.2 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 非成员函数、静态成员函数和全局函数

使用静态成员函数或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。

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。

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

{

void Function1();

void Function2();

} // namespace foo\_bar

} // namespace myproject

而非

namespace myproject

{

class FooBar

{

public:

static void Function1();

static void Function2();

};

} // namespace myproject

### 2.4 局部变量

将函数变量尽可能置于最小作用域内，并在变量声明时进行初始化。

C++允许在函数的任何位置声明变量。我们提倡在尽可能小的作用域中声明变量，离第一次使用越近越好。这使得代码浏览者更容易定位变量声明的位置，了解变量的类型和初始值。特别是，应使用初始化的方式替代声明再赋值，比如：

int i;

i = f(); // 坏,初始化和声明分离

int j = g(); // 好,初始化时声明

vector<int> v;

v.push\_back(1); // 用花括号初始化更好

v.push\_back(2);

vector<int> v = {1，2}; // 好,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。

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);

}

### 2.5 静态和全局变量

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。

## 3 类

### 3.1 构造函数的职责

#### 3.1.1 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。

#### 3.1.2 定义

在构造函数中可以进行各种初始化操作。

#### 3.1.3 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。

#### 3.1.4 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。
* 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。

#### 3.1.5 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。

### 3.2 隐式类型转换

#### 3.2.1 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。

#### 3.2.2 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。

#### **3.2.3 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。

#### **3.2.4 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。

#### **3.2.5 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};) 。

### 3.3 可拷贝类型和可移动类型

#### 3.3.1 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。

#### 3.3.2 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。

#### 3.3.3 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。

#### 3.3.4 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。

#### 3.3.5 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;

### 3.4 结构体 VS 类

#### 3.4.1 总述

仅当只有数据成员时使用struct，其它一概使用class。

#### 3.4.2 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。

### 3.5 继承

#### 3.5.1 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。

#### **3.5.2 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。

#### **3.5.3 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。

#### **3.5.4 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。

#### **3.5.5 结论**

所有继承必须是public的。如果你想使用私有继承，你应该替换成把基类的实例作为成员对象的方式。

不要过度使用实现继承。组合常常更合适一些。尽量做到只在“是一个”的情况下使用继承；如果Bar的确“是一种” Foo，Bar才能继承Foo。

必要的话，析构函数声明为 virtual。如果你的类有虚函数，则析构函数也应该为虚函数。

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。

### 3.6 多重继承

#### 3.6.1 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。

#### **3.6.2** 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。

#### **3.6.3** 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。

#### **3.6.**4 **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。

#### **3.6.5** 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。

### 3.7 接口

接口是指满足特定条件的类，这些类以Interface为后缀（不强制）。

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。

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。

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。

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。

### 3.8 存取控制

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)。

### 3.9 声明顺序

将相似的声明放在一起，将public部分放在最前。

类定义一般应以public开始，后跟protected，最后是private。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。只有那些普通的，或非性能关键且短小的函数可以内联在函数定义中。参见内联函数一节。

## 4 函数

### 4.1 参数顺序

函数的参数顺序为：输入参数在先，后跟输出参数。

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。在排列参数顺序时，将所有的输入参数置于输出参数之前。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，有时候为了与其他函数保持一致，你可能不得不有所变通。

### 4.2 编写简短函数

我们倾向于编写简短，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，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。

### 4.3 引用参数

所有按引用传递的参数，不作为输出参数时必须加上const。

### 4.4 函数重载

#### 4.4.1 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。

#### 4.4.2 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);

};

#### 4.4.3 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。

#### 4.4.4 缺点

如果函数单靠不同的参数类型而重载 (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。

#### 4.4.5 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。

### 4.5 缺省参数

#### 4.5.1 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。

#### **4.5.2 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”。

#### **4.5.3 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。

#### **4.5.4 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。

### 4.6 函数返回类型后置语法

#### 4.6.1 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。

#### **4.6.2 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。

#### **4.6.3 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);

#### **4.6.4 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。

#### **4.6.5 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。

## 所有权与智能指针

### 5.1 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。

### 5.2 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。

### **5.3 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。

### **5.4 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。

### **5.5 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);

不要使用 std::auto\_ptr，使用std::unique\_ptr代替它。

## 其他 C++ 特性

### 6.1 右值引用

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

#### 6.1.1 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。

#### 6.1.2 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。

#### 6.1.3 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。

#### 6.1.4 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。

### 6.2 缺省参数

我们不允许使用缺省函数参数，少数极端情况除外。尽可能改用函数重载。

#### 6.2.1 优点

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

#### 6.2.2 缺点

缺省参数会干扰函数指针，害得后者的函数签名（function signature）往往对不上所实际要调用的函数签名。即在一个现有函数添加缺省参数，就会改变它的类型，那么调用其地址的代码可能会出错，不过函数重载就没这问题了。此外，缺省参数会造成臃肿的代码，毕竟它们在每一个调用点（call site）都有重复。函数重载正好相反，毕竟它们所谓的「缺省参数」只会出现在函数定义里。

#### 6.2.3 结论

由于缺点并不是很严重，有些人依旧偏爱缺省参数胜于函数重载。所以除了以下情况，我们要求必须显式提供所有参数。

其一，位于.cpp文件里的静态函数或匿名空间函数，毕竟都只能在局部文件里调用该函数了。

其二，可以在构造函数里用缺省参数，毕竟不可能取得它们的地址。

其三，可以用来模拟变长数组。

// 通过空 AlphaNum 以支持四个形参

Std::string StrCat(const AlphaNum &a, const AlphaNum &b = gEmptyAlphaNum，

const AlphaNum &c = gEmptyAlphaNum, const AlphaNum &d = gEmptyAlphaNum);

### 6.3 变长数组

我们不允许使用变长数组。

变长数组具有浑然天成的语法。变长数组也都很高效。

变长数组不是标准 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”。

改用更安全的分配器（allocator），就像std::vector或std::unique\_ptr<T[]>。

### 6.4 友元

我们允许合理的使用友元类及友元函数。

通常友元应该定义在同一文件内，避免代码读者跑到其它文件查找使用该私有成员的类。经常用到友元的一个地方是将FooBuilder声明为Foo的友元，以便FooBuilder正确构造Foo的内部状态，而无需将该状态暴露出来。某些情况下，将一个单元测试类声明成待测类的友元会很方便。

友元扩大了 (但没有打破) 类的封装边界。某些情况下，相对于将类成员声明为public，使用友元是更好的选择，尤其是如果你只允许另一个类访问该类的私有成员时。当然，大多数类都只应该通过其提供的公有成员进行互操作。

### 6.5 运行时类型识别

我们禁止使用 RTTI。

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。

#### 6.5.1 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;

...

}

#### 6.5.2 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。

#### 6.5.3 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。

### 6.6 类型转换

使用C++的类型转换，如 static\_cast<>()。不要使用int y = (int)x或int y = int(x)等转换方式。

C++采用了有别于C的类型转换机制，对转换操作进行归类。

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。

缺点是恶心的语法。

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进行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参见“运行时类型识别”。

### 6.7 const语法

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。

不要使用int const \*foo形式，使用const int\* foo形式。

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; };）。

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。

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。

const变量，数据成员，函数和参数为编译时类型检测增加了一层保障；便于尽早发现错误。因此，我们强烈建议在任何可能的情况下使用const:

* 如果函数不会修改传你入的引用或指针类型参数，该参数应声明为 const。
* 尽可能将函数声明为 const。访问函数应该总是 const。其他不会修改任何数据成员，未调用非const函数，不会返回数据成员非const 指针或引用的函数也应该声明成 const。
* 如果数据成员在对象构造之后不再发生变化，可将其定义为 const。

然而，也不要发了疯似的使用 const。像 const int \* const \* const x; 就有些过了，虽然它非常精确的描述了常量 x。关注真正有帮助意义的信息: 前面的例子写成 const int\*\* x 就够了。

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

const的位置，有人喜欢 int const \*foo 形式，不喜欢 const int\* foo，他们认为前者更一致因此可读性也更好，遵循了const 总位于其描述的对象之后的原则。但是一致性原则不适用于此，“不要过度使用” 的声明可以取消大部分你原本想保持的一致性。将 const 放在前面才更易读，因为在自然语言中形容词 (const) 是在名词 (int) 之前。

### 6.8 constexpr用法

在C++11里，用constexpr来定义真正的常量，或实现常量初始化。可用于编译时常量的应用场景。

### 6.9 整型

C++ 内建整型中，仅使用 int。如果程序中需要不同大小的变量，可以使用 <stdint.h> 中长度精确的整型，如 int16\_t。如果您的变量可能不小于 2^31 (2GiB)，就用 64 位变量比如int64\_t。此外要留意，哪怕您的值并不会超出int所能够表示的范围，在计算过程中也可能会溢出。所以拿不准时，干脆用更大的类型。

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。

<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（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!

### 6.10 预处理宏

使用宏时要非常谨慎，尽量以内联函数，枚举和常量代替之。

宏意味着你和编译器看到的代码是不同的。这可能会导致异常行为，尤其因为宏具有全局作用域。

值得庆幸的是，C++ 中，宏不像在C中那么必不可少。以往用宏展开性能关键的代码，现在可以用内联函数替代。用宏表示常量可被 const 变量代替。用宏“缩写”长变量名可被引用代替。用宏进行条件编译，这个，千万别这么做，会令测试更加痛苦 (#define 防止头文件重包含当然是个特例)。

宏可以做一些其他技术无法实现的事情，在一些代码库 (尤其是底层库中) 可以看到宏的某些特性 (如用#字符串化，用##连接等等)。但在使用前，仔细考虑一下能不能不使用宏达到同样的目的。

下面给出的用法模式可以避免使用宏带来的问题，如果你要宏，尽可能遵守：

* 不要在.h文件中定义宏。
* 在马上要使用时才进行#define，使用后要立即#undef。
* 不要只是对已经存在的宏使用#undef，选择一个不会冲突的名称；
* 不要试图使用展开后会导致 C++构造不稳定的宏，不然也至少要附上文档说明其行为。
* 不要用##处理函数，类和变量的名字。

### 6.11 sizeof

尽可能用 sizeof(varname) 代替 sizeof(type)。

使用 sizeof(varname) 是因为当代码中变量类型改变时会自动更新。可以使用 sizeof(type) 处理不涉及任何变量的代码。

### 6.12 auto

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。

### 6.13 Lambda表达式

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

禁用默认捕获，捕获都要显式写出来。打比方，比如 [=] (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。

**6.14 模板编程**

不要使用复杂的模板编程

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。

#### 6.14.1 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

#### 6.14.2 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。

#### 6.14.**3** 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漏。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。

## 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，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。

### 7.1 通用命名规则

函数命名，变量命名，文件命名要有描述性; 少用缩写。

尽可能使用描述性的命名，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。

### 7.2 文件命名

文件名要全部小写，可以包含下划线 \_ 。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。

### 7.3 类型命名

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

所有类型命名：类，结构体，类型定义 (typedef)，枚举，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>;

// 枚举

enum UrlTableErrors { ...

### 7.4 变量命名

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\_。

#### 7.4.1 普通变量命名

string table\_name; // 好，用下划线。

string tablename; // 好，全小写。

string tableName; // 差，混合大小写

#### 7.4.2 类数据成员

class TableInfo

{

private:

string tablename\_; // 好。

static Pool<TableInfo>\* pool\_; // 好。

};

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，但要后接下划线。

#### 7.4.3 结构体变量

不管是静态的还是非静态的，结构体数据成员都可以和普通变量一样，不用像类那样后接下划线：

struct UrlTableProperties

{

string name;

int num\_entries;

static Pool<UrlTableProperties>\* pool;

};

### 7.5 常量命名

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;

所有具有静态存储类型的变量 (例如静态变量或全局变量) 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。

### 7.6 函数命名

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

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()

### 7.7 命名空间命名

命名空间以小写字母命名。最高级命名空间的名字取决于项目名称。要注意避免嵌套命名空间的名字之间和常见的顶级命名空间的名字之间发生冲突。

顶级命名空间的名称应当是项目名或者是该命名空间中的代码所属的团队的名字。命名空间中的代码，应当存放于和命名空间的名字匹配的文件夹或其子文件夹中。

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)。

### 7.8 枚举命名

枚举的命名应当和常量一致：kEnumName。

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，

};

### 7.9 宏命名

建议不要使用宏，如果你一定要用，其命名像枚举命名一样全部大写，使用下划线。

例如：#define PI\_ROUNDED 3.0

### 7.10 命名规则的特例

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

## 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!

对于代码中巧妙的，晦涩的，有趣的，重要的地方加以注释。根据使用对象确定注释语言，比如开源到国际平台的使用英文，公司内部使用的使用中文。大家英文水平参差不齐，规定不允许使用英文注释，否则，代码注释的可用性大大降低。

### 8.1 注释风格

单行注释使用//，独立的多行注释（中间不夹杂代码）使用/\* \*/。

### 8.2 类注释

每个类的定义都要附带一份注释，描述类的功能和用法，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

{

...

};

\*/

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。

### 8.3 函数注释

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。

#### 8.3.1 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。

#### 8.3.2 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。

**8.4 变量注释**

通常变量名本身足以很好说明变量用途。某些情况下，也需要额外的注释说明。

#### 8.4.1 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\_;

#### 8.4.2 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;

### 8.5 实现注释

#### 8.5.1 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;

}

#### 8.5.2 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。\*/

**8.5.3 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。

**8.5.4 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);

}

### 8.6 TODO 注释

对那些临时的，短期的解决方案，或已经够好但仍不完美的代码使用TODO注释。

TODO注释要使用全大写的字符串TODO，在随后的圆括号里写上你的名字，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。”)。

### 8.7 弃用注释

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。

## 格式

### 9.1 文件格式

文件格式必须使用UTF-8格式。不使用UTF-8 BOM格式，不使用gb2312格式。

文件中换行符必须使用 \n。

### 9.2 非ASCII字符

代码中，尽量不使用非ASCII字符，使用时必须使用UTF-8编码。

### 9.3 空格还是制表位

缩进只使用空格，每次缩进4个空格。

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。

### 9.4 函数声明与定义

返回类型和函数名在同一行，参数也尽量放在同一行，如果放不下就对形参分行,分行方式与函数调用一致。

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();

...

}

注意以下几点:

* 如果返回类型和函数名在一行放不下，分行。
* 如果返回类型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。
* 左大括号相对最后一个参数所在行另起新行。
* 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。
* 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。

### 9.5 Lambda表达式

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。

int x = 0;

auto add\_to\_x = [&x](int n) { x += n; };

若用引用捕获，在变量名和&之间不留空格。

digits.erase(std::remove\_if(digits.begin()，digits.end()，

[&blacklist](int i)

{

return blacklist。find(i) != blacklist。end();

}，

digits.end());

### 9.6 函数调用

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

函数调用遵循如下形式：

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

如果同一行放不下，可断为多行，后面每一行都从与函数名对齐位置缩进4个空格，左圆括号后和右圆括号前不要留空格：

bool retval = DoSomething(averyveryveryverylongverylongverylongargument1,

argument2, argument3);

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);

### **9.7 列表初始化格式**

按照函数调用格式进行列表初始化。

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。

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

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"

};

MyType m =

{

superlongvariablename1，

superlongvariablename2，

{short，interior，list}，

{interiorwrappinglist，

interiorwrappinglist2}

};

### 9.8 条件语句

在圆括号和条件之间不要加空格。注意所有情况下if和左圆括号间都有个空格。

if (condition)

{

...

}

else if (...)

{

...

}

else

{

...

}

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。

if (condition)

DoSomething(); // 4空格缩进。

if (condition)

{

DoSomething(); // 4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;

}

// 好，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;

}

### 9.9 循环和开关选择语句

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。

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:

{

...

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。

### 9.10 指针和引用表达式

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。

### 9.11 布尔表达式

如果一个布尔表达式太长要断行，断行方式要统一一下。

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)

{

...

}

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。

### 9.12 函数返回值

不要在return表达式里加上非必须的圆括号。

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！

### 9.13 变量及数组初始化

使用 =，() 和 {} 均可。

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。

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。

### 9.14 预处理指令

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。

// 好，指令从行首开始

if (lopsided\_score)

{

#if DISASTER\_PENDING // 正确，从行首开始

DropEverything();

# if NOTIFY // 差，#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();

}

### 9.15 类格式

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

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: // 无缩进

MyClass(); // 标准的四空格缩进

explicit MyClass(int var);

~MyClass() {}

void SomeFunction();

void SomeFunctionThatDoesNothing() {}

void SetSomeVar(int var) { some\_var\_ = var; }

int SomeVar() const { return some\_var\_; }

private:

bool SomeInternalFunction();

int some\_var\_;

int some\_other\_var\_;

};

注意事项:

* 所有基类名应尽量与子类名放在同一行。
* 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。

### 9.16 构造函数初始值列表

构造函数初始化列表放在同一行或按四格缩进并排多行。

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

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)

{

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) {}

### 9.17 命名空间格式化

命名空间内容不缩进。声明嵌套命名空间时，每个命名空间都独立成行。

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 {

### 9.18 水平留白

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。

void f(bool b)

{

...

int i = 0; // 分号前不加空格。

}

// 列表初始化中大括号内不加空格。

int x[] = {0，1};

// 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)。(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)。

#### 9.18.1 循环和条件语句

if (b)

{

...

}

else

{

...

}

while (test) {} // 圆括号内部不紧邻空格。

switch (i) // 圆括号内部不紧邻空格。

{

...

}

for (int i = 0; i < 5; ++i) // 圆括号内部不紧邻空格。

{

...

}

switch ( i ) // 差，圆括号内部不紧邻空格。

{

...

}

if ( test ) // 差，圆括号内部不紧邻空格。

{

...

}

for ( int i = 0; i < 5; ++i ) // 差，圆括号内部不紧邻空格。

{

...

}

for (; i < 5; ++i) // 循环体内 ; 后恒有空格，; 前不可以有空格。

{

}

switch (i)

{

case 1: // switch case 的冒号前无空格。

...

case 2: break; // 如果冒号后有代码，加个空格。

}

#### 9.18.2 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)

...

#### 9.18.3 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。

vector<char \*> x;