Team-Xmbot-Service-Robot C++风格指南

0. 前言

0.1 版本

- 1.0版本(2016.5.1)：缪宇飚(myyerrol)创建团队 C++ 代码开发风格指南。本文档参考了 ROS 和 Google 的 C++ 风格指南，并根据实际的需求，对内容进行了适当的精简和改进。新队员应该认真学习本指南，掌握 C++ 基本的开发风格。如果有细节不统一的地方或者对本文档某处不是很认同，请在组内讨论统一之后，修改本指南。因为文档排版使用的是 Markdown 纯文本标记语言，也请后来者遵循本文档的开发方式，使用 Markdown 来修改、添加内容。

0.2 背景

C++ 是 ROS 项目开发的主要编程语言。正如每个 C++ 开发者所知道的，C++ 有很多强大的特性，但这种强大不可避免的导致它走向复杂，使代码更容易产生 Bug，难以阅读和维护。

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

Team-Xmbot-Service-Robot（晓萌家庭服务机器人团队）作为开源项目，需要团队队员贡献代码，但是如果队员之间的代码编程风格不一致，便会给团队其他模块负责人造成不小的困扰。我们认为整洁、一致的代码风格可以使代码更加可读、更加可调试以及更加可维护。因此，我们应该编写优雅的代码使其不仅能够在现在发挥作用，而且在将来的若干年之后其依旧能够存在、可以被复用、或者是能够被未来的新队员改进。

1. 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。

1.1 #define 保护

> **Tip**
>
> 所有头文件都应该使用 `#define` 来防止头文件被多重包含，命名格式当是：`<NAME>_H` 。

为保证唯一性， 头文件的命名应该与头文件的命名一致。例如，`xm_arm_robot_hardware.h` 头文件可按如下方式保护：

```
#ifndef XM_ARM_ROBOT_HARDWARE_H
#define XM_ARM_ROBOT_HARDWARE_H
…
#endif // XM_ARM_ROBOT_HARDWARE_H
```

1.2 `#include` 的路径及顺序

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

`#include` 中包含头文件的次序如下：

1. ROS 系统文件
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

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

举例来说，头文件包含次序如下：

```
#include <ros/ros.h>
#include <ros/callback_queue.h>
#include <hardware_interface/robot_hw.h>
#include <hardware_interface/joint_command_interface.h>
#include <hardware_interface/joint_state_interface.h>
#include <controller_manager/controller_manager.h>
#include <realtime_tools/realtime_buffer.h>
#include <realtime_tools/realtime_publisher.h>
#include <std_msgs/Float64.h>
#include <xm_arm_msgs/xm_ArmSerialDatagram.h>

#include <sstream>
#include <vector>
#include <string>
#include <map>
#include <boost/shared_ptr.hpp>
```

2. 作用域

2.1 名字空间

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

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

优点：
虽然类已经提供了（可嵌套的）命名轴线，名字空间在这基础上又封装了一层。举例来说，两个不同项目的全局作用域都有一个类 `Foo`，这样在编译或运行时造成冲突。如果每个项目将代码置于不同名字空间中，`project1::Foo` 和 `project2::Foo` 作为不同符号自然不会冲突。

缺点：

- 名字空间具有迷惑性，因为它们和类一样提供了额外的（可嵌套的）命名轴线。
- 命名空间很容易令人迷惑，毕竟它们不再受其声明所在命名空间的限制。内联命名空间

只在大型版本控制里有用。
- 在头文件中使用匿名空间导致违背 C++ 的唯一定义原则。

结论：
根据下文将要提到的策略合理使用命名空间。

2.1.1 匿名名字空间

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

```
namespace {
enum { kUNUSED, kEOF, kERROR };
bool AtEof() { return pos_ == kEOF; }
...
} // namespace
```

然而，与特定类关联的文件作用域声明在该类中被声明为类型，静态数据成员或静态成员函数，而不是匿名名字空间的成员。如上例所示，匿名空间结束时用注释 `// namespace` 标识。

2.1.2 具名的名字空间

具名的名字空间使用方式如下：

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

```
namespace mynamespace {
class MyClass
{
public:
    void Foo();
    ...
private:
    int foo;
    ...
};
} // namespace mynamespace
```

2.2 非成员函数、静态成员函数和全局函数

> **Tip**
>
> 使用静态成员函数或名字空间内的非成员函数，尽量不要用裸的全局函数。

优点：
某些情况下，非成员函数和静态成员函数是非常有用的，将非成员函数放在名字空间内可避免污染全局作用域。

缺点：
将非成员函数和静态成员函数作为新类的成员或许更有意义，当它们需要访问外部资源或具有重要的依赖关系时更是如此。

结论：

- 有时，把函数的定义同类的实例脱钩是有益的，甚至是必要的。这样的函数可以被定义成静态成员，或是非成员函数。非成员函数不应依赖于外部变量，应尽量置于某个名字空间内。相比单纯为了封装若干不共享任何静态数据的静态成员函数而创建类，不如使用 `namespaces`。
- 定义在同一编译单元的函数，被其他编译单元直接调用可能会引入不必要的耦合和链接时依赖，静态成员函数对此尤其敏感。可以考虑提取到新类中，或者将函数置于独立库的名字空间内。
- 如果你必须定义非成员函数，又只是在 `.cpp` 文件中使用它，可使用匿名namespaces或 `static` 链接关键字（如 `static int Foo() {...}`）限定其作用域。

2.3 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;
// 警告：初始化和声明分离。
i = f();
// 正确：初始化时声明。
int j = g();

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

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

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

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

> **Warning**
>
> 如果变量是一个对象，每次进入作用域都要调用其构造函数，每次退出作用域都要调用其析构函数。

```
// 低效的实现。
for (int i = 0; i < 1000000; i++)
{
    // 构造函数和析构函数分别调用 1000000 次！
    Foo f;
    f.DoSomething(i);
}
```

```
// 高效的实现。
// 构造函数和析构函数只调用 1 次。
Foo f;
for (int i = 0; i < 1000000; i++)
{
    f.DoSomething(i);
}
```

3. 类

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

3.1 构造函数的职责

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

定义：
在构造函数体中进行初始化操作。

优点：
排版方便，无需担心类是否已经初始化。

缺点：
在构造函数中执行操作引起的问题有：

- 构造函数中很难上报错误，不能使用异常。
- 操作失败会造成对象初始化失败，进入不确定状态。
- 如果在构造函数内调用了自身的虚函数，这类调用是不会重定向到子类的虚函数实现。即使当前没有子类化实现，将来仍是隐患。
- 如果有人创建该类型的全局变量（虽然违背了上节提到的规则），构造函数将先 `main()` 一步被调用，有可能破坏构造函数中暗含的假设条件。

结论：
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初始化，考虑使用明确的 `Init()` 方法或使用工厂模式。

3.2 初始化

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

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

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

缺点：

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

结论：

- 简单的初始化用类成员初始化完成，尤其是当一个成员变量要在多个构造函数里用相同的方式初始化的时候。
- 如果你的类中有成员变量没有在类里面进行初始化，而且没有提供其它构造函数，你必须定义一个（不带参数的）默认构造函数。把对象的内部状态初始化成一致/有效的值无疑是更合理的方式。这么做的原因是：如果你没有提供其它构造函数，又没有定义默认构造函数，编译器将为你自动生成一个。编译器生成的构造函数并不会对对象进行合理的初始化。
- 如果你定义的类继承现有类，而你又没有增加新的成员变量，则不需要为新类定义默认构造函数。

3.3 结构体 VS 类

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

结论：

- 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` 。
- 注意，类和结构体的成员变量使用不同的命名规则。

3.4 继承

> **Tip**
>
> 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，即子类继承父类的实现代码。接口继承，即子类仅继承父类的方法名称。

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

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

结论：

- 所有继承必须是 `public` 的。如果你想使用私有继承，你应该替换成把基类的实例作为成员对象的方式。
- 不要过度使用实现继承。组合常常更合适一些。尽量做到只在 “是一个”的情况下使用继

承：如果 `Bar` 的确 “是一种“ Foo，`Bar` 才能继承 `Foo`。
- 必要的话，析构函数声明为 `virtual`。如果你的类有虚函数，则析构函数也应该为虚函数。注意**数据成员在任何情况下都必须是私有的**。
- 当重载一个虚函数，在衍生类中把它明确的声明为 `virtual`。理论依据：如果省略 `virtual` 关键字，代码阅读者不得不检查所有父类，以判断该函数是否是虚函数。

3.5 声明顺序

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

类的访问控制区段的声明顺序依次为：`public:`、`protected:`、`private:`。如果某区段没内容，可以不声明。

每个区段内的声明通常按以下顺序：

1. `typedefs` 和枚举
2. 常量
3. 构造函数
4. 析构函数
5. 成员函数，含静态成员函数
6. 数据成员，含静态数据成员

`.cpp` 文件中函数的定义应尽可能和声明顺序一致。

3.6 编写简短函数

> **Tip**
>
> 倾向编写简短，凝练的函数。

我们承认长函数有时是合理的，因此并不硬性限制函数的长度。如果函数超过 40 行，可以思索一下能不能在不影响程序结构的前提下对其进行分割。

即使一个长函数现在工作的非常好，一旦有人对其修改，有可能出现新的问题。甚至导致难以发现的 Bug。使函数尽量简短，便于他人阅读和修改代码。

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

4. 其他 C++ 特性

4.1 引用参数

> **Tip**
>
> 所有按引用传递的参数必须加上 `const`。

定义：
在 C 语言中，如果函数需要修改变量的值，参数必须为指针，如 `int foo(int *pval)`。在 C++ 中，函数还可以声明引用参数：`int foo(int &val)`。

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

缺点：
容易引起误解，因为引用在语法上是值变量却拥有指针的语义。

结论：
函数参数列表中，所有引用参数都必须是 `const`：

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

事实上这是一个硬性约定：输入参数是值参或 `const` 引用，输出参数为指针。输入参数可以是 `const` 指针，但决不能是非 `const` 的引用参数，除非用于交换，比如 `swap()`。

有时候，在输入形参中用 `const T*` 指针比 `const T&` 更明智。比如：

- 您会传 `null` 指针。
- 函数要把指针或对地址的引用赋值给输入形参。

总之大多时候输入形参往往是 `const T&`。若用 `const T*` 说明输入另有处理。所以若您要用 `const T*`，则应有理有据，否则会害得读者误解。

4.2 函数重载

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

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

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

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

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

结论：
如果您打算重载一个函数，可以试试改在函数名里加上参数信息。例如，用 `AppendString()` 和 `AppendInt()` 等，而不是一口气重载多个 `Append()`。

4.3 缺省参数

> **Tip**
>
> 我们不允许使用缺省函数参数，少数极端情况除外。尽可能改用函数重载。

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

缺点：
缺省参数会干扰函数指针，害得后者的函数签名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。

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：

- 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。
- 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。
- 可以用来模拟变长数组。

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

4.4 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。

4.5 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，不用管那些含糊且容易出错的错误代码。
- 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或 `Init()` 方法代替异常。
- 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。
- 启用异常会增加二进制文件数据，延长编译时间（或许影响小），还可能加大地址空间的压力。
- 滥用异常会变相鼓励开发者去捕捉不合时宜，或本来就已经没法恢复的「伪异常」。比如，用户的输入不符合格式要求时，也用不着抛异常。如此之类的伪异常列都列不完。

结论：

- 从表面上看来，使用异常利大于弊，尤其是在新项目中。但是对于现有代码，引入异常会牵连到所有相关代码。如果新项目允许异常向外扩散，在跟以前未使用异常的代码整合时也将是个麻烦。因为现有的大多数 C++ 代码都没有异常处理，引入带有异常处理的新代码相当困难。
- 鉴于现有代码不接受异常，在现有代码中使用异常比在新项目中使用的代价多少要大一些。迁移过程比较慢，也容易出错。我们不相信异常的使用有效替代方案，如错误代码、断言等会造成严重负担。因此我们不建议使用异常。

4.6 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的时候。
- 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。

4.7 流

> **Tip**
>
> 只在记录日志时使用流。

定义：
流用来替代 `printf()` 和 `scanf()`。

优点：
有了流，在打印时不需要关心对象的类型。不用担心格式化字符串与参数列表不匹配（虽然在 GCC 中使用 `printf` 也不存在这个问题）。流的构造和析构函数会自动打开和关闭对应的文件。

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

结论：

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

拓展讨论：

- 对这一条规则存在一些争论，这儿给出点深层次原因。回想一下唯一性原则：我们希望在任何时候都只使用一种确定的 I/O 类型，使代码在所有 I/O 处都保持一致。因此，我们不希望用户来决定是使用流还是 `printf + read/write`。相反，我们应该决定到底用哪一种方式。把日志作为特例是因为日志是一个非常独特的应用，还有一些是历史原因。
- 流的支持者们主张流是不二之选，但观点并不是那么清晰有力。他们指出的流的每个优势也都是其劣势。流最大的优势是在输出时不需要关心打印对象的类型。这是一个亮点。同时，也是一个不足：你很容易用错类型，而编译器不会报警。使用流时容易造成的这类错误：

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

由于 `<<` 被重载，编译器不会报错。就因为这一点我们反对使用操作符重载。

有人说 `printf` 的格式化丑陋不堪，易读性差，但流也好不到哪儿去。看看下面两段代码吧，实现相同的功能，哪个更清晰？

```
cerr << "Error connecting to '" << foo->bar()->hostname.first
     << ":" << foo->bar()->hostname.second << ":" << strerror(errno);

fprintf(stderr, "Error connecting to '%s:%u:%s",
        foo->bar()->hostname.first, foo->bar()->hostname.second,
        strerror(errno));
```

你可能会说，”把流封装一下就会比较好了”，这儿可以，其他地方呢？而且不要忘了，我们的目标是使语言更紧凑，而不是添加一些别人需要学习的新装备。

每一种方式都是各有利弊，“没有最好，只有更适合”。简单性原则告诫我们必须从中选择其一，最后大多数决定采用 `printf + read/write`。

4.8 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）。

4.9 `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`。

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

优点：
大家更容易理解如何使用变量。编译器可以更好地进行类型检测，相应地，也能生成更好的代码。人们对编写正确的代码更加自信，因为他们知道所调用的函数被限定了能或不能修改变量值。即使是在无锁的多线程编程中，人们也知道什么样的函数是安全的。

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

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

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

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

`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！

4.10 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，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，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！

4.11 `nullptr` 和 `NULL`

> **Tip**
>
> 整数用 `0` ，实数用 `0.0` ，指针用 `nullptr` 或 `NULL` ，字符（串）用 `'\0'` 。

4.12 `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;
memset(&data , 0 , sizeof(data));
```

4.13 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。

5. 命名约定

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

命名规则具有一定随意性， 但相比按个人喜好命名， 一致性更重， 所以不管你怎么想， 规则总归是规则。

5.1 通用命名规则

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

```
int price_count_reader;
int num_errors;
int num_dns_connections;
```

5.2 文件命名

> **Tip**
>
> 文件名要全部小写， 可以包含下划线（ `_` ）。

可接受的文件命名：

- xm_arm_robot_hardware.cpp
- xm_arm_gripper_control.cpp
- xm_arm_trajectory_control.cpp

C++ 文件要以 `.cpp` 结尾， 头文件以 `.h` 结尾。

通常应尽量让文件名更加明确. `xm_arm_trajectory_control.cpp` 就比 `xm_arm_control.cpp` 要好。 `.h` 和 `.cpp` 要成对出现， 如 `xm_arm_robot_hardware.h` 和 `xm_arm_robot_hardware.cpp`。

5.3 类型命名

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

所有类型命名 —— 类、结构体、类型定义（`typedef`）、枚举 —— 均使用相同约定，而且名字应该是名词性的。

5.4 变量命名

> **Tip**
>
> 变量名一律小写，单词之间用下划线连接。类的成员变量以下划线结尾，但结构体的就不用，如：`a_local_variable`、`a_struct_data_member`、`a_class_data_member_`。

普通变量命名：

```
string table_name;
string tablename;
```

类数据成员：

不管是静态的还是非静态的，类数据成员都可以和普通变量一样，但要接下划线。

```
class TableInfo
{
...
private:
    string table_name_;
    string tablename_;
    static Pool<TableInfo> *pool_;
 };
```

结构体变量：

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

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

全局变量：

对全局变量没有特别要求，少用就好，但如果你要用，可以用 `g_` 标志作为前缀，以便更好的区分局部变量。

5.5 常量命名

> **Tip**
>
> 在全局或类里的常量名称前加 `k` ：kDaysInAWeek。且除去开头的 `k` 之外每个单词开头字母均大写。
>
> 所有编译时常量，无论是局部的，全局的还是类中的，和其他变量稍微区别一下。 `k` 后接大写字母开头的单词：

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

这规则适用于编译时的局部作用域常量，不过要按变量规则来命名也可以。

5.6 函数命名

> **Tip**
>
> 函数使用驼峰命名法（camelCased）：`getExcitingResult()`，`checkForErrors()`

函数或方法通常执行的是一个动作。因此，它们的名字应该能够反映出它们能做什么（动词性的）。使用`checkForErrors()`来代替`errorCheck()`，`dumpDataToFile()`代替`dataFile()`。

5.7 名字空间命名

> **Tip**
>
> 名字空间用小写字母命名：`xm_arm_namespace`。

5.8 枚举命名

> **Tip**
>
> 枚举的命名应当和宏一致：`ENUM_NAME`。

枚举值应该采用宏的命名方式。

```
enum AlternateUrlTableErrors
{
  OK = 0,
    OUT_OF_MEMORY = 1,
    MALFORMED_INPUT = 2
};
```

5.9 宏命名

> **Tip**
>
> 名字全部大写并且可加下划线，比如：`PI_ROUNDED`。

6. 注释

> **Tip**
>
> 在程序难以理解的地方加上注释，而且全部要用英文写，不要用中文。

注释虽然写起来很痛苦，但对保证代码可读性至关重要。下面的规则描述了如何注释以及在哪儿注释。当然也要记住：注释固然很重要，但最好的代码本身应该是自文档化。有意义的类型名和变量名，要远胜过要用注释解释的含糊不清的名字。

你写的注释是给代码读者看的：对下一个需要理解你的代码的人慷慨些吧， 因为下一个人可能就是你！

6.1 注释风格

> **Tip**
>
> 使用 `//` 或 `/* */`，统一就好。

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

6.2 文件注释

> **Tip**
>
> 在每一个文件开头加入版权公告，然后是文件内容描述。

法律公告和作者信息：

每个文件都应该包含以下项， 依次是：

- 版权声明 (比如，`Copyright (c) 2016, Team-Xmbot-Service-Robot`)

- 许可证：为项目选择合适的许可证版本（BSD）
- 作者：标识文件的原始作者。
- 时间：创建文件的时间。

如果你对原始作者的文件做了重大修改，将你的信息添加到作者信息里。这样当其他人对该文件有疑问时可以知道该联系谁。

举个例子：

```
/*********************************************************************
 *  Software License Agreement (BSD License)
 *
 *  Copyright (c) 2016, Team-Xmbot-Service-Robot
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of the Team-Xmbot-Service-Robot nor the names
 *     of its contributors may be used to endorse or promote products
 *     derived from this software without specific prior written
 *     permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTOR
S
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIREC
T,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDIN
G,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *********************************************************************/

// Authors: myyerrol
// Created: 2016.4.15
```

文件内容：

紧接着版权许可和作者信息之后，每个文件都要用注释描述文件内容。

通常 `.h` 文件要对所声明的类的功能和用法作简单说明。`.cpp` 文件通常包含了更多的实现细节或算法技巧讨论。如果你感觉这些实现细节或算法技巧讨论对于理解 `.h` 文件有帮助，可以将该注释挪到 `.h`，并在 `.cpp` 中指出文档在 `.h`。

不要简单的在 `.h` 和 `.cpp` 间复制注释，这种偏离了注释的实际意义。

6.3 类注释

> **Tip**
>
> 每个类的定义都要附带一份注释，描述类的功能和用法。

```
// Iterates over the contents of a GargantuanTable.
class GargantuanTableIterator
{
    ...
};
```

6.4 函数注释

> **Tip**
>
> 函数声明处注释描述函数功能，定义处描述函数实现。

函数声明：

注释位于声明之前，对函数功能及用法进行描述。注释使用叙述式而非指令式。注释只是为了描述函数，而不是命令函数做什么。通常，注释不会描述函数如何工作。那是函数定义部分的事情。

函数声明处注释的内容：

- 函数的输入输出。
- 对类成员函数而言：函数调用期间对象是否需要保持引用参数，是否会释放这些参数。
- 如果函数分配了空间，需要由调用者释放。
- 参数是否可以为 `NULL`。
- 是否存在函数使用上的性能隐患。
- 如果函数是可重入的，其同步前提是什么？

举例如下：

```
// Returns an iterator for this table.  It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
//
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

但也要避免罗罗嗦嗦，或做些显而易见的说明。下面的注释就没有必要加上 “Returns false otherwise”，因为已经暗含其中了：

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

注释构造/析构函数时，切记读代码的人知道构造/析构函数是干啥的，所以 “Destroys this object” 这样的注释是没有意义的。注明构造函数对参数做了什么（例如，是否取得指针所有权）以及析构函数清理了什么。如果都是些无关紧要的内容，直接省掉注释。析构函数前没有注释是很正常的。

函数定义：

每个函数定义时要用注释说明函数功能和实现要点。比如说说你用的编程技巧，实现的大致步骤，或解释如此实现的理由，为什么前半部分要加锁而后半部分不需要。

不要从 `.h` 文件或其他地方的函数声明处直接复制注释。简要重述函数功能是可以的，但注释重点要放在如何实现上。

6.5 变量注释

> **Tip**
>
> 通常变量名本身足以很好说明变量用途。某些情况下，也需要额外的注释说明。

类数据成员：

每个类数据成员（也叫实例变量或成员变量）都应该用注释说明用途。如果变量可以接受 `NULL` 或 `-1` 等警戒值，须加以说明。比如：

```
private:
    // Keeps track of the total number of entries in the table.
    // Used to ensure we do not go over the limit. -1 means
    // that we don't yet know how many entries the table has.
    int num_total_entries_;
```

全局变量：

和数据成员一样，所有全局变量也要注释说明含义及用途。比如：

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

6.6 实现注释

> **Tip**
>
> 对于代码中巧妙的，晦涩的，有趣的，重要的地方加以注释。

代码前注释：

巧妙或复杂的代码段前要加注释。比如：

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++)
{
  x = (x << 8) + (*result)[i];
    (*result)[i] = x >> 1;
    x &= 1;
}
```

6.7 标点，拼写和语法

> **Tip**
>
> 注意标点，拼写和语法。写的好的注释比差的要易读的多。

注释的通常写法是包含正确大小写和结尾句号的完整语句。短一点的注释（如代码行尾注释）可以随意点，依然要注意风格的一致性。完整的语句可读性更好，也可以说明该注释是完整的，而不是一些不成熟的想法。

虽然被别人指出该用分号时却用了逗号多少有些尴尬，但清晰易读的代码还是很重要的。正确的标点，拼写和语法对此会有所帮助。

7. 格式

代码风格和格式确实比较随意，但一个项目中所有人遵循同一风格是非常容易的。个体未必同意下述每一处格式规则，但整个项目服从统一的编程风格是很重要的，只有这样才能让所有人能很轻松的阅读和理解代码。

7.1 行长度

> **Tip**
>
> 每一行代码字符数不超过 80。

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

优点：
提倡该原则的人主张强迫他们调整编辑器窗口大小很野蛮。很多人同时并排开几个代码窗口，根本没有多余空间拉伸窗口。大家都把窗口最大尺寸加以限定，并且 80 列宽是传统标准。为什么要改变呢？

缺点：
反对该原则的人则认为更宽的代码行更易阅读。80 列的限制是上个世纪 60 年代的大型机的古板缺陷。现代设备具有更宽的显示屏，很轻松的可以显示更多代码。

结论：
80 个字符是最大值。

特例：

- 如果一行注释包含了超过 80 字符的命令或 URL，出于复制粘贴的方便允许该行超过 80 字符。
- 包含长路径的 `#include` 语句可以超出80列。但应该尽量避免。
- 头文件保护，可以无视该原则.

7.2 非 ASCII 字符

> **Tip**
>
> 尽量不使用非 ASCII 字符，使用时必须使用 UTF-8 编码。

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

7.3 空格还是制表位

> **Tip**
>
> 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符表，你应该设置编辑器将制符表转为空格。

7.4 函数声明与定义

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

函数看上去像这样：

```
ReturnType ClassName::FunctionName(Type par_name, 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,
    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。
- 如果返回类型那个与函数声明或定义分行了，不要缩进。
- 左圆括号总是和函数名在同一行。

- 函数名和左圆括号间没有空格。
- 圆括号与参数间没有空格。
- 如果其它风格规则允许的话，右大括号总是单独位于函数最后一行，或者与左大括号同一行。
- 右大括号和左大括号间总是有一个空格。
- 函数声明和定义中的所有形参必须有命名且一致。
- 所有形参应尽可能对齐。
- 换行后的参数保持 4 个空格的缩进。

如果有些参数没有用到，在函数定义处将参数名注释起来：

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

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

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

> **Warning**

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

7.5 函数调用

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

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

如果同一行放不下，可断为多行，后面每一行都和第一个实参对齐，左圆括号后和右圆括号前不要留空格：

```
bool retval = DoSomething(averyveryveryverylongargument1 ,
                          argument2 ,  argument3);
```

参数也可以放在次行，缩进四格：

```
if (...)
{
  DoSomething(
        argument1, argument2,
        argument3, argument4);
}
```

把多个参数放在同一行，是为了减少函数调用所需的行数，除非影响到可读性。有人认为把每个参数都独立成行，不仅更好读，而且方便编辑参数。不过，比起所谓的参数编辑，我们更看重可读性，且后者比较好办。

如果一些参数本身就是略复杂的表达式，且降低了可读性。那么可以直接创建临时变量描述该表达式，并传递给函数：

```
int my_heuristic = scores[x] * y + bases[x];
bool retval = DoSomething(my_heuristic ,  x ,  y ,  z);
```

如果某参数独立成行，对可读性更有帮助的话，就这么办。

此外，如果一系列参数本身就有一定的结构，可以酌情地按其结构来决定参数格式：

```
// 通过 3x3 矩阵转换 widget。
my_widget.Transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

7.6 条件语句

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

最常见的是没有空格的格式。哪种都可以，但保持一致性。如果你是在修改一个文件， 参考当前已有格式。如果是写新的代码， 参考目录下或项目中其它文件。还在徘徊的话，就不要加空格了。

```
// 正确：圆括号里没空格紧邻。
if (condition)
{
    ...
}
else
{
    ...
}
```

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

> **Warning**

```
// 警告：if 后面没空格。
if(condition)
```

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

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

如果语句有 `else` 分支则不允许：

> **Warning**

```
// 警告：当有 else 分支时，if 块却只有一行。
if (x) DoThis();
else DoThat();
```

要求 `if-else` 必须总是使用大括号，哪怕 `if` 或 `else` 语句只有一行：

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

7.7 循环和开关选择语句

> **Tip**
>
> `switch` 语句必须使用大括号分段，以表明 `case` 之间不是连在一起的。在单语句循环里，括号必须使用。空循环体应使用 `{}` 或 `continue`。

`switch` 语句中的 `case` 块要使用大括号。

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

```
switch (var)
{
    case 0:
    {   ...
        break;
    }
    default:
    {
        assert(false);
    }
}
```

在单语句循环里，括号也必须使用：

```
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。

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

> **Warning**

```
// 警告：看起来仅仅只是 while 的部分之一。
while (condition);
```

7.8 指针和引用表达式

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

下面是指针和引用表达式的正确使用范例：

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

注意：

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

在声明指针变量或参数时，星号要与变量名紧挨着。

```
// 正确：空格前置。
char *c;
const string &str;
```

> **Warning**

```
// 警告：* 两边都有空格。
char * c;
// 警告：& 两边都有空格。
const string & str;
```

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。

7.9 布尔表达式

> **Tip**
>
> 如果一个布尔表达式超过标准行宽，断行方式要统一一下。

下例中， 逻辑与 (`&&`) 操作符总位于行尾：

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

注意，上例的逻辑与 (`&&`) 操作符均位于行尾。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` 。

7.10 函数返回值

> **Tip**
>
> `return` 表达式里时没必要都用圆括号。

假如您写 `x = epr` 时本来就会加上括号，那 `return expr;` 也可如法炮制。

函数返回时不要使用圆括号：

```
// 正确：返回值很简单，没有圆括号。
return result;
// 正确：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);
```

> **Warning**

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

7.11 变量及数组初始化

> **Tip**
>
> 用 `=`、`()`、`{}` 均可。

您可以用 `=`、`()`、`{}`，以下都对：

```
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);
vector<int> v{100, 1};
```

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。

```
// 正确：pi == 3。
int pi(3.14);
// 错误：不合适的转换。
int pi{3.14};
```

7.12 预处理指令

> **Tip**
>
> 预处理指令不要缩进，从行首开始。

即使预处理指令位于缩进代码块中，指令也应从行首开始。

```
    if (lopsided_score)
    {
// 正确：从行开头起。
#if DISASTER_PENDING
        DropEverything();
#endif
        BackToNormal();
    }
```

Warning

```
    if (lopsided_score)
    {
        // 警告："#if" 应该放在行开头。
        #if DISASTER_PENDING
        DropEverything();
        // 警告："#endif" 不要缩进。
        #endif
        BackToNormal();
    }
```

7.13 类格式

> **Tip**
>
> 访问控制块的声明依次序是 `public：`、`protected：`、`private：`，没有缩进。

类声明的基本格式如下：

```
class MyClass : public OtherClass
{
public:
    MyClass();
    ~MyClass() {}
    void SomeFunction();
    void SomeFunctionThatDoesNothing() {}
    void set_some_var(int var) { some_var_ = var; }
    int some_var() const { return some_var_; }
private:
    bool SomeInternalFunction();
private:
    int some_var_;
    int some_other_var_;
};
```

注意事项：

- 所有基类名应在 80 列限制下尽量与子类名放在同一行。
- 关键词 `public:` 、 `protected:` 、 `private:` 没有缩进。
- 关键词块之间不要有空格。
- 函数与变量如果属于同一个访问控制权限，则要分别写。顺序是：先函数，后变量（如上面例子中最后两个 private）。
- 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`.

7.14 构造函数初始值列表

> **Tip**
>
> 构造函数初始值列表按四格缩进并排几行。

```
// 断成多行，缩进四格，冒号放在第一行。
MyClass::MyClass(int var)
    : some_var_(var),
      some_other_var_(var + 1)
{
    DoSomething();
    ...
}
```

7.15 水平留白

> **Tip**
>
> 水平留白的使用因地制宜。永远不要在行尾添加没意义的留白。

常规：

```
// 分号前不加空格。
int i = 0;
// 大括号内部与空格紧邻。
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:
    // 大括号里面是空的话，不加空格。
    Foo(int b) : Bar(), baz_(b) {}
    // 用空格把大括号与实现分开。
    void Reset() { baz_ = 0; }
    ...
}
```

添加冗余的留白会给其他人编辑时造成额外负担。因此，行尾不要留空格。如果确定一行代码已经修改完毕，将多余的空格去掉。或者在专门清理空格时去掉（确信没有其他人在处理)。

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 、switch 和 for/while 关键字后均有空格。
if (b) {}
switch (i) {}
while (test) {}
for (int i = 0; i < 5; ++i) {}
switch (i)
{
    // case 的冒号前无空格。
    case 1:
    ...
    // case 的冒号后有代码，加个空格。
    case 2: break;
    ...
}
```

操作符：

```
// 赋值操作系统前后恒有空格。
x = 0;

// 其它二元操作符也前后恒有空格。
// 圆括号内部不紧邻空格。
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，>( 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;

set<list<string> > x;
// 可以加上一对对称的空格。
set< list<string> > x;
```

7.16 垂直留白

> **Tip**
>
> 垂直留白越少越好。

这不仅仅是规则而是原则问题了：不在万不得已，不要使用空行。尤其是两个函数定义之间的空行不要超过 2 行（最好是1行），函数体首尾不要留空行，函数体中也不要随意添加空行。

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

空行心得如下：

- 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。

8. 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。

8.1 现有不合规范的代码

> **Tip**
>
> 对于现有不符合既定编程风格的代码可以网开一面。

当你修改使用其他风格的代码时，为了与代码原有风格保持一致可以不使用本指南约定。如果不放心可以与代码原作者或现在的负责人员商讨。记住，一致性包括原有的一致性。

9. 结束语

> **Tip**
>
> 运用常识和判断力，并保持一致。

编辑代码时，花点时间看看项目中的其它代码，并熟悉其风格。如果其它代码中 `if` 语句使用空格，那么你也要使用。如果其中的注释用星号 (*) 围成一个盒子状，你同样要这么做。

风格指南的重点在于提供一个通用的编程规范，这样大家可以把精力集中在实现内容而不是表现形式上。我们展示了全局的风格规范，但局部风格也很重要，如果你在一个文件中新加的代码和原有代码风格相去甚远，这就破坏了文件本身的整体美观，也影响阅读，所以要尽量避免。

好了，关于编码风格写的够多了。代码本身才更有趣，尽情享受吧！